UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

GRADO EN INGENIERÍA EN INFORMÁTICA

Infraestructura como Código (IaC) para un sistema de microservicios Cloud.

Alejandro Gálvez Ruiz

Julio, 2023

INFRAESTRUCTURA COMO CÓDIGO (IAC) PARA UN SISTEMA DE MICROSERVICIOS CLOUD.

UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

Tecnologías y Sistemas de Información

GRADO EN INGENIERÍA EN INFORMÁTICA

INGENIERÍA DE COMPUTADORES

Infraestructura como Código (IaC) para un sistema de microservicios Cloud.

Autor: Alejandro Gálvez Ruiz

Tutor académico: David Villa Alises

Julio, 2023

Alejandro Gálvez Ruiz

Ciudad Real – Spain

TRIBUNAL:

Presidente:

Vocal:

Secretario:

FECHA DE DEFENSA:

CALIFICACIÓN:

PRESIDENTE	VOCAL	SECRETARIO
Fdo.:	Fdo.:	Fdo.:

Resumen

La llegada de los entornos Cloud supuso un nuevo paradigma en el desarrollo de *software* y en la creación de infraestructuras más escalables y flexibles. Sin embargo, esto también ha generado un aumento de la complejidad y de la cantidad de servicios en la nube que deben ser creados y gestionados.

Este proyecto tiene como objetivo analizar y comprender las ventajas que ofrece el uso de la Infraestructura como Código en la provisión de servicios en diferentes entornos Cloud en comparación con los métodos tradicionales, como la Interfaz de Línea de Comandos (CLI) o las interfaces web.

Para ello, se hará uso de Terraform, una de las herramientas más relevantes en este ámbito, para llevar a cabo una provisión y despliegue de una arquitectura para un sistema de microservicios en cada uno de los tres entornos Cloud más importantes actualmente: AWS, Azure y GCP. De esta manera, este trabajo permitirá no solo entender la importancia de la automatización en la provisión de infraestructura, sino también comparar los diferentes servicios que ofrece cada una de estas plataformas Cloud y comprender las diferencias al aprovisionar entre una y otra.

Abstract

The emergence of Cloud environments marked a new paradigm in *software* development and the creation of more scalable and flexible infrastructures. However, this has also led to an increase in the complexity and the number of cloud services that need to be created and managed.

This project aims to analyse and understand the advantages offered by Infrastructure as Code in the provision of services in different Cloud environments, compared to traditional methods such as Command Line Interface (CLI) or web interfaces.

To achieve this, Terraform, one of the most relevant tools in this field, will be used to provision and deploy an architecture for a microservices system in each of the three most important Cloud environments today: AWS, Azure and GCP. In this way, this work will not only allow us to understand the importance of automation in infrastructure provisioning, but also to compare different services offered by each of these Cloud platforms and understand the differences when provisioning between them.

Agradecimientos

A *mi padre* y a *mi madre*, que no solo me dieron la vida, sino que día a día me demuestran como vivirla.

A *mis abuelos*, que siempre serán una prueba incondicional del cariño y amor de los seres humanos.

A *mi hermano*, que siempre ha estado ahí para servirme de ejemplo.

A *mis amigos*, por hacer más divertidos todos estos años.

A *mis profesores*, que durante toda mi etapa educativa han dejado su huella en mi de alguna u otra manera.

A *Paco*, por ofrecerme su ayuda en cualquier inconveniente que me ha surgido durante la realización de este proyecto.

A *NTT Data*, por brindarme la oportunidad de realizar este trabajo con ellos. En especial a *mis compañeros*, que día a día hicieron más entretenida mi estancia.

Gracias a todos.

Alejandro Gálvez Ruiz

A mis abuelos

Índice general

Índice de cuadros

Índice de figuras

Índice de listados

Listado de acrónimos

NIST	Instituto Nacional de Estándares y Tecnologia
CLI	Command Line Interface
AWS	Amazon Web Services
CPU	Unidad Central de Procesamiento
GCP	Google Cloud Platform
IaC	Infraestrcuture as Code
IaaS	Infraestrcuture-as-a-Service
PaaS	Platform-as-a-Service
SaaS	Software-as-a-Service
DSL	Lenguaje de Dominio Específico
HCL	HashiCorp Configuration Language
API	Application Programming Interfaces
RPC	Remote Procedure Call
JSON	JavaScript Object Notation
EC2	Elastic Compute Cloud
AMI	Imágenes de máquina de Amazon
HVM	Hardware Virtual Machine
CDK	Cloud Development Kit
EKS	Elastic Kubernetes Service
HTTP	Protocolo de Transferencia de Hipertexto
SSH	Secure Shell
VPC	Virtual Private Cloud
VM	Virtual Machine
TCP	Protocolo de Control de Transmisión
UDP	Protocolo de Datagramas de Usuario
ICMP	Protocolo de Control de Mensajes de Internet
SO	Sistema Operativo
ALB	Application Load Balancer
URL	Uniform Resource Locator
MIG	Grupos de Instancias Administradas
RDS	Relational Database Service
SSD	Solid-State Drive

IOPS	Input/Output Operations Per Second
GB	GigaByte
RDP	Remoite Desktop Protocol
ECR	Elastic Container Registry
ECS	Elastic Container Service
FQDN	Fully Qualified Domain Name
NEG	Network Endpoint Group
HCP	Computación de Altas Prestaciones
DNS	Domain Name System
SLA	Service Level Agreement
FaaS	Function as a Service
OCI	Open Container Initiative
RPM	Red Hat Package Manager
ACL	Network Access Control List
IoT	Internet of Things
NAT	Traducción de Direcciones de Red

Capítulo 1

Introducción

AÑOS atrás, durante la «edad de hierro» de la infraestructura, el crecimiento de la misma estaba limitado por capacidades físicas y económicas. Instalar un servidor requería en primer lugar un espacio físico suficiente para albergar todo su *hardware*, a la vez que lo suficientemente adaptado para desempeñar sus funciones con normalidad (suministro eléctrico, temperatura adecuada, etc.). Esto acarreaba a su vez un cargo económico bastante importante; además de requerir del tiempo necesario para su instalación, configuración y correcto funcionamiento [4].

Aún con todo esto en cuenta, las organizaciones que conseguían poseer este tipo de infraestructura podían verse en la situación de que su despliegue no fuera suficiente para satisfacer todas las necesidades de una masa de usuarios que era cada vez mayor. Por lo que ante la imposibilidad de ampliar esta arquitectura, acababan por permitir que múltiples usuarios compartieran recursos, resultando en sistemas menos seguros y personalizables [4].

La llegada de las Máquinas Virtuales abrió el camino para solventar muchos de estos problemas. Tener la capacidad de crear múltiples entornos virtuales sobre un mismo sistema físico permitía desarrollar varias operaciones simultáneamente en más de un sistema operativo, y con su propia memoria y unidad de procesamiento, para desarrollar así soluciones mucho más personalizables para cada usuario [5].

Este fue el primer paso dado hacia la virtualización, que junto al desarrollo de los sistemas distribuidos, paralelos, centralizados y la llegada del *Grid Computing* dieron como resultado el *Cloud Computing* [6].

1.1 *Cloud Computing*

El *Cloud Computing* consiste en el acceso bajo demanda y distribuido a una serie de recursos informáticos que pueden ir desde simples aplicaciones hasta enormes cantidades de servidores con altas capacidades computacionales. El NIST establece cinco características fundamentales sobre este modelo [7]:

- *Autoservicio bajo demanda*. Cualquier consumidor tiene la capacidad de aprovisionarse unilateralmente de estos recursos computacionales, es decir, de manera automática

y sin interacción humana de por medio con el proveedor.

- *Acceso amplio a la red*. Todas los recursos están disponibles desde la red y pueden ser accesibles desde gran variedad de plataformas, desde móviles a computadoras.
- *Agrupación de recursos*. Todos los recursos hacen uso de un modelo de multitenencia para servir a múltiples consumidores, asignando y reasignando dinámicamente cada uno de sus componentes para satisfacer las necesidades de cada usuario. Además, existe un cierto nivel de independencia de la ubicación, pues el consumidor no sabe con exactitud la localización de dichos recursos, aunque esta si que pueda ser expresada a un alto nivel como a través de regiones.
- *Rápida elasticidad*. Los recursos pueden escalar fácilmente para satisfacer la demanda. A vistas del consumidor, los recursos disponibles para aprovisionar parecen ilimitados y listos para ser adquiridos en cualquier momento.
- *Servicio medido*. Los recursos pueden ser monitorizados, controlados e informados tanto al proveedor como al consumidor con el fin de controlar ciertas métricas como el ancho de banda, el almacenamiento o el uso de CPU.

1.1.1 Modelos de servicios

Dependiendo del nivel de control y de abstracción que ofrecen al usuario, estos servicios se dividen principalmente en tres modelos [7].

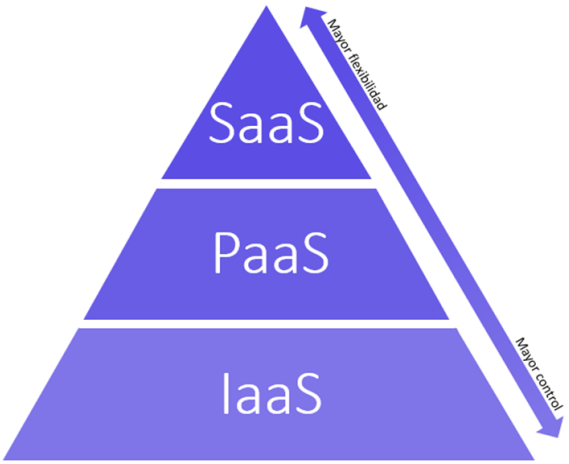

Figura 1.1: Pirámide con los modelos de *Cloud Computing*.
Cuanto más en la base mayor control y menor flexibilidad, y cuanto más en el pico mayor flexibilidad y menor control.

- *Infraestructure-as-a-Service* (IaaS). Constituye el nivel más «crudo», pues ofrece directamente una serie de recursos *hardware* tales como servidores, redes o almacenamiento. Es por lo tanto el nivel con mayor nivel de flexibilidad y control, pero a su vez

el que mayor responsabilidad requiere.
En este nivel podríamos encontrar por ejemplo las Máquinas Virtuales.

- *Platform-as-a-Service* (PAAS). Abstrae al usuario de conocer las necesidades *hardware* de su sistema, ofreciendo ciertos entornos configurables donde no es necesario administrar la infraestructura subyacente. De este modo, permiten al consumidor centrarse en el desarrollo y administración de sus aplicaciones.
  En este nivel podríamos encontrar por ejemplo los contenedores.

- *Software-as-a-Service* (SAAS). Es el servicio más extendido, pues consiste en aplicaciones que funcionan en remoto en una arquitectura *Cloud* totalmente abstracta para los consumidores, y que acaban siendo accedidas a través de navegadores web o cualquier otra aplicación cliente.
  En este nivel podríamos encontrar por ejemplo los servicios de correo electrónico.

1.1.2 Modelos de despliegue

En función del entorno *cloud* distinguimos varios tipos de nubes [7].

- *Cloud pública*. La infraestructura desplegada en la nube es accesible al público.

- *Cloud comunitaria*. Entornos restringidos a organizaciones que tienen inquietudes similares. Puede estar administrada por una o más organizaciones en la comunidad, terceras personas o una combinación de ambas.

- *Cloud privada*. La infraestructura es usada exclusivamente por una organización, que puede o no ser su propietaria.

- *Cloud híbrida*. Es una combinación de cualquiera de los modelos anteriores.

1.1.3 Ventajas

La llegada del *Cloud Computing* ha traído consigo numerosas ventajas con respecto al anterior paradigma.

- *Reducción de costes y tiempo*. Debido a que ya no es necesario albergar tus propios recursos *hardware*, se reduce drásticamente el esfuerzo que suponía comprar, instalar, configurar y mantener tu propia infraestructura.

- *Mejora del escalado*. Ya no existe la necesidad de instalar nuevo *hardware* para escalar tu sistema. Los entornos *cloud* ofrecen mecanismos para automatizar esta característica en función de los requisitos del sistema, mejorando así la disponibilidad y la fiabilidad.

- *Personalización*. Es posible ofrecer una solución única y más personalizable a cada usuario.

1.1.4 Estado actual

Desde que Salesforce lanzó el primer servicio en la nube en 1999, la industria ha crecido exponencialmente. Muchas empresas como Amazon, Google o Microsoft han lanzado sus propios servicios *Cloud* convirtiéndose así en uno de los sectores con mayor crecimiento dentro de las TIC.

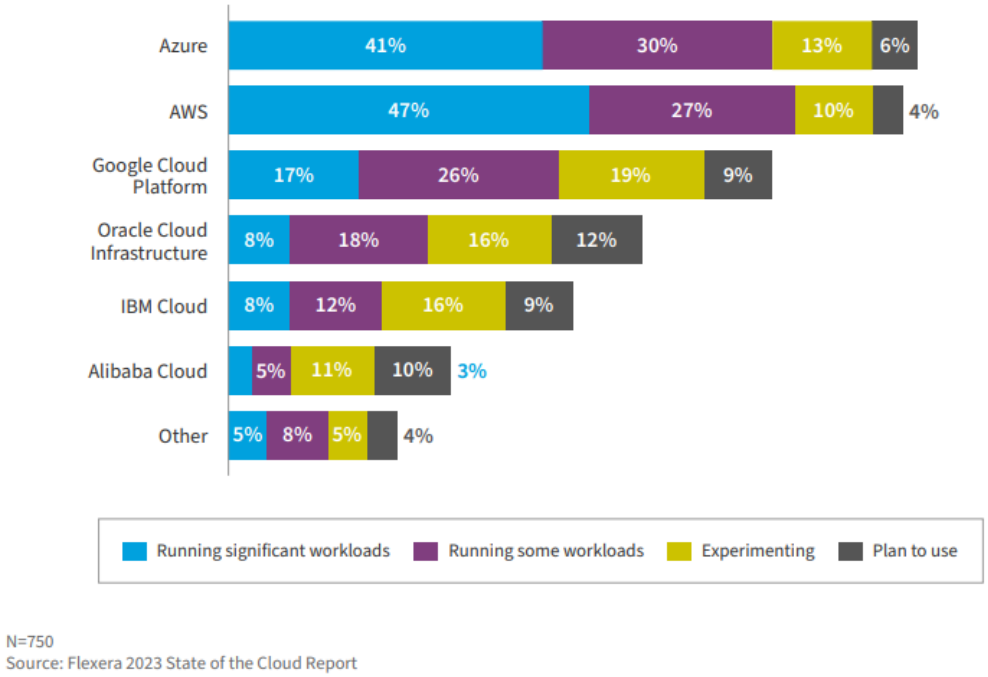

Figura 1.2: Gráfico que refleja el proveedor usado por las empresas ([1])

Este crecimiento se vio aún más acelerado con la llegada del COVID. La Pandemia ha demostrado una vez más el gran valor que tiene esta tecnología, donde su alta disponibilidad, su capacidad de recuperación o sus bajos costes, han supuesto factores diferenciales. Tal ha sido su importancia que 9 de cada 10 empresas admite haber aumentado su uso a causa de la Pandemia [8].

Y es que la computación en la nube ha demostrado ser una excelente aliada para todo tipo de campos como el teletrabajo, la telemedicina, la Inteligencia Artifical, la Realidad Virtual y Aumentada o el *Internet of Things*, entre otros.

Sin embargo, esto ha traído consigo nuevos retos. Atrás quedaron los años donde un simple *mainframe* bastaba para dar soporte a unos cuantos usuarios. La expansión desorbitada de Internet en estos últimos tiempos ha dado lugar a sistemas cada vez más y más complejos, compuestos por múltiples recursos redundantes y de distintos proveedores *Cloud*. Esto supone que los métodos tradicionales de aprovisionamiento como las interfaces gráficas web o los CLI acaben siendo demasiado tediosos cuando las dimensiones de la arquitectura son muy grandes. Es en este contexto donde nace la Infraestructura como Código (IAC).

Capítulo 2

Objetivos

A partir de los antecedentes mencionados en el capítulo anterior, se procede a describir detalladamente los objetivos generales y específicos que se pretenden abordar en el desarrollo de este Trabajo de Fin de Grado.

2.1 Objetivo general

El objetivo general del proyecto consiste en hacer un estudio de una de las herramientas más demandadas de la Infraestructura como Código como es Terraform, a través de la creación de una arquitectura para un sistema de microservicios en cada uno de los tres entornos *Cloud* más importantes: AWS, Azure y GCP.

2.2 Objetivos específicos

Se van a definir una serie de sub-objetivos cuya consecución supondrá el cumplimiento del objetivo general previamente mencionado.

2.2.1 Análisis de servicios *Cloud*

Realizar un análisis sobre los distintos servicios ofrecidos por cada proveedor *Cloud*, con el fin de tener una mejor visión de la arquitectura que se puede construir. De este modo, el alumno será capaz de identificar y seleccionar los servicios de cada plataforma que serán necesarios para la construcción de la arquitectura deseada.

2.2.2 Estudio sobre Terraform

Realizar una investigación a fondo sobre la herramienta de Terraform con el fin de entender su funcionamiento, capacidades, características y buenas prácticas.

2.2.3 Diseño y construcción de cada arquitectura

Se llevará a cabo un diseño y planificación sobre la arquitectura a crear en cada entorno *Cloud*, definiendo las especificaciones y los recursos necesarios para cada una de ellas, con el objetivo de desplegarlas y validarlas.

2.2.4 Comparativa de los entornos

Realización de una comparativa entre los tres entornos a través de la información recabada durante los anteriores sub-objetivos sobre los servicios ofrecidos, las características de cada uno de ellos y su coste asociado.

Capítulo 3

Antecedentes

3.1 Infraestructura como Código

La Infraestructura como Código (IaC) consiste en describir mediante código la infraestructura *hardware* o *software* deseada, de manera que esta pueda ser fácilmente creada, destruida, redimensionada o remplazada. Puede comprender todo el proceso de creación de un entorno, desde la construcción de servidores y sus instancias, hasta la configuración de su sistema operativo y el *software* que correrá en ellos. [9]

3.1.1 Metodologías

Dentro del mundo de la IAC existen dos tipos de metodologías:

- *Imperativas*. Se define explícitamente, por ejemplo mediante *scripts*, los pasos y el orden para conseguir tu estado deseado. Esto otorga un mayor control pero también obliga a cuidar mejor las dependencias entre las infraestructuras, gestionar los errores, etc.
- *Declarativas*. Se define el estado que se desea conseguir, y será la propia herramienta la que se encargue automáticamente de cómo obtenerlo. Dentro de este campo entran muchas herramientas de aprovisionamiento como Terraform.

3.1.2 Categorías de herramientas

A continuación se listan las distintas categorías de herramientas IAC que existen. [9]

Ad hoc scripts

Se hacen uso de *scripts* para automatizar un conjunto de tareas que normalmente que harían manualmente; desde la configuración *software* de un servidor, hasta el aprovisionamiento de recursos en la Nube a través de los comandos ofrecidos por las CLI de cada proveedor *Cloud*. Si bien puede ser útil para tareas pequeñas y simples, es un enfoque bastante alejado de las bases de la Infraestructura como Código, puesto que puede ser poco reutilizable y propenso a errores.

Herramientas de Configuración

Herramientas orientadas a configurar el *software* dentro de cierta infraestructura. Por lo tanto son mas apropiadas para instalaciones donde ya se ha aprovisionado de un

hardware, y ahora es necesario gestionar y configurar el *software* que se va a instalar en él. Por ejemplo, encontramos herramientas como Ansible.

Herramientas de *Server Templating*

Si en el anterior enfoque primero era necesario aprovisionarse de un *hardware* para luego instalar el *software* dentro de él, en las herramientas de *Server Templating* inicialmente se crea una imagen de dicho *software* prediseñado para que más adelante sea utilizado por cualquier *hardware* que lo solicite. Es decir, se crean plantillas con un sistema operativo determinado, un *software* preconfigurado y cualquier otro fichero necesario, y estas serán usados a la hora de instanciar cualquier tipo de *hardware*. Así, encontramos herramientas como Docker, Packer o Vagrant.

Herramientas de Orquestación

Herramientas orientadas a definir el despliegue, actualización, monitorización, escalado y distribución del trafico dentro de una determinado sistema. Por ejemplo Kubernetes para orquestar contenedores.

Herramientas de Aprovisionamiento

Mientras que las herramientas de Configuración y *Server Templating* definían el *software* que correrá en cada *hardware*, las herramientas de Aprovisionamientos se encargan de aprovisionar dicha infraestructura, como servidores, bases de datos o redes virtuales. Pueden ser de dos tipos:

- *Cloud Specific*. Herramientas proporcionadas por proveedores *Cloud*, y por ende muy útiles para sus servicios, pero sin utilidad para el resto. Por ejemplo Cloud Formation (Amazon), Azure Resource Manager (Microsoft) o Google Cloud Deployment Manager.
- *Cloud Agnostic*. Herramientas independientes del servicio *Cloud*. Por ejemplo Terraform o Pulumi.

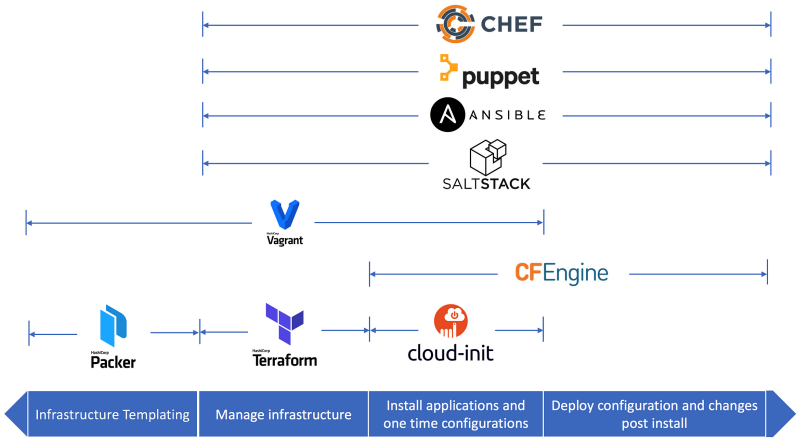

Figura 3.1: Diagrama que muestra las distintas herramientas IAC y su alcance ([2])

3.1.3 Ventajas de la Infraestructura como Código

Definir la infraestructura de un sistema mediante código permite obtener muchas de las ventajas que ya nos brindaba de por si el desarrollo de *software* [9, 10].

- *Mayor rapidez y seguridad en el despliegue*. Cambiar el aprovisionamiento de recursos de un proceso humano y manual a uno automatizado por computadoras es, en muchos sentidos, bastante más rápido, consistente y seguro.
- *Mayor reusabilidad*. El código puede ser empaquetado para ser reutilizado las veces que haga falta, facilitando así el despliegue constante y manual de recursos idénticos.
- *Mejor escalabilidad*. Gracias a la rapidez de modificación del código, la infraestructura se vuelve fácilmente escalable.
- *Mayor control de la infraestructura*. Lo que ves es lo que tienes.
- *Facilita el trabajo en equipo*. Al trabajar con código es mucho más sencillo compartir el trabajo dentro de un equipo.
- *Control de versiones*. Al describir tu infraestructura mediante ficheros, es factible mantener un control de versiones de cada uno de ellos.
- *Validación de la infraestructura*. Es posible diseñar *tests* automáticos para validar la infraestructura ideada.

3.2 Terraform

Terraform es una herramienta de IAC de código abierto desarrollado por HashiCorp para el aprovisionamiento de infraestructura mediante código legible en archivos de configuración. Entre sus características más destacables se encuentran las siguientes [9].

- *Código abierto*.
- *Herramienta de aprovisionamiento*. Como ya se comentó previamente, Terraform es una herramienta orientada enteramente a aprovisionar infraestructura. Si bien permite un cierto grado de configuración, es mejor dejar este trabajo en herramientas creadas específicamente para ello como Ansible.
- *Cloud-agnostic*. Terraform permite trabajar con distintos entornos *Cloud*, tanto de manera conjunta como individual.
- *Infraestructura inmutable*. Terraform trabaja con el concepto de inmutabilidad, lo que significa que sus recursos creados nunca serán modificados. Esto se traduce en que cualquier cambio realizado sobre sus servicios significará la obsolescencia del antiguo recurso, que poco a poco irá migrando su tráfico hacia una nueva instancia con los cambios aplicados. Esto supone una gran ventaja frente a enfoques mutables donde los cambios se realizan sobre las mismas instancias, puediendo generar así un cierto *configuration drift*.

- *Metodologia declarativa*. Terraform utiliza un estilo declarativo donde el usuario escribe cual es su estado deseado y será la propia plataforma la que se encargue de alcanzar ese estado. Esto supone una ventaja frente a enfoques más procedurales donde un pequeño cambio supondria la aplicación de un nuevo archivo de configuración.
- *Lenguaje de Dominio Específico(*DSL*)*. A diferencia de muchos de sus competidores, Terraform hace uso de un DSL llamado HashiCorp Configuration Language (HCL) para describir sus archivos de configuración.
- *Servicio gratuito y premium*. Terraform ofrece un servicio gratuito, que puede ser más que suficiente para llevar una infraestructura a producción, a la vez que ofrece un servicio de pago en Terraform Cloud para proyectos mucho más exigentes.

3.2.1 ¿Cómo es la arquitectura de Terraform?

Lógicamente hablando, Terraform está dividido en dos partes [11].

- *Terraform Core*. Es la base de Terraform y está escrita en Go. Contiene su Command Line Interface (CLI) y un motor de grafos, por lo que sus principales funciones son:
  - Leer e interpolar los archivos de configuración.
  - Gestionar el estado de los recursos a través de un archivo que mencionaremos a continuación.
  - Construye un grafo de dependencias entre los recursos.
- *Terraform Plugins*. Son un conjunto de *plugins* escritos en Go que sirven de puente entre Terraform Core y las API de cada entorno *Cloud*. Existen dos tipos:
  - *Providers*. Sirven para aprovisionar recursos dentro de un entorno. Aunque es posible crear *Providers* de uso privado, la mayoría de estos se pueden descargar desde repositorios públicos como Terraform Registry. Así, es posible encontrar distintos tipos de *Providers* [12].
    - *Providers oficiales*. Desarrollados y mantenidos por HashiCorp.
    - *Providers partner*. Desarrollados y mantenidos por empresas de terceros que usan su propia API y que han pasado previamente un control.
    - *Providers de la comunidad*. Desarrollados y mantenidos por grupos de usuarios ajenos.
    - *Providers archivados*. *Providers* que ya están depreciados.
  - *Provisioners*. Se utilizan para realizar determinadas acciones dentro de una máquina, local o remota, con el objetivo de prepararla para realizar un servicio, como por ejemplo instalar cierto *software*, subir ficheros, etc. Sin embargo, su uso no está recomendado pues, como ya se mención anteriormente en la sección 3.1.2, Terraform es una herramienta de aprovisionamiento de infraestructura, no de configuración [13].

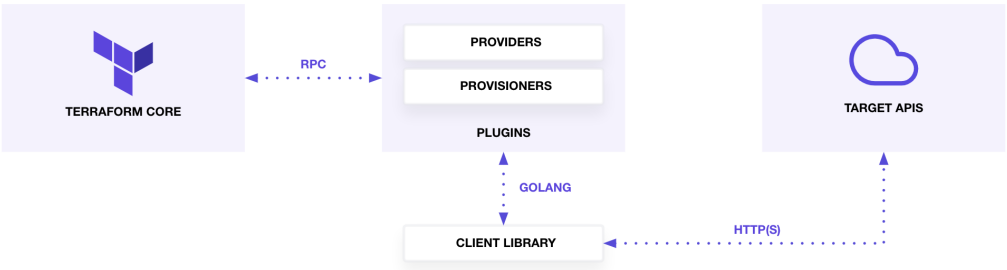

Figura 3.2: Diagrama que muestra la arquitectura subyacente de Terraform ([3])

Cada una de estas partes corresponden a procesos separados que se comunican mediante RPC. De modo que cuando Terraform Core lo requiera, este se lo comunicará a cada uno de estos *plugins* que, a su vez, hará lo propio con las API necesarias de cada entorno, normalmente a través de librerías cliente, tal y como podemos ver en la Figura 3.2.

3.2.2 ¿Cómo trabaja Terraform?

Una de las principales diferencias que tiene Terraform frente a otras herramientas de IAC es su modo de trabajo.

Ficheros *State*

Terraform hace uso de un fichero JSON llamado *State* para representar el estado actual de tu infraestructura. De este modo, cualquier cambio que se realice sobre dicha infraestructura, se llevará a cabo en comparación con este archivo [14].
Esto es un aspecto fundamental a tener en cuenta por tres razones principales [14, 9].

- Una vez se decide aprovisionar infraestructura mediante Terraform, realizar cambios desde fuera, como a través de la interfaz web, puede causar problemas graves en un proyecto, pues estos no se verán reflejados en dicho archivo *State*.
- Trabajar en grupos de varias personas sobre una misma infraestructura requiere de usar mecanismos de sincronización para que todos trabajen sobre un mismo archivo *State* sin generar conflictos.
- La información sensible como contraseñas o certificados que se hayan especificado quedarán guardados en dicho fichero.

Por todo esto, Terraform ofrece soluciones como trabajar en entornos remotos llamados *backends*, que puede ser propios como Terraform Cloud, o externos como un *S3 Bucket* de AWS [15].
Entre las ventajas que existe hacer uso de *backends* encontramos las siguientes [15, 9].

- Facilita la colaboración al compartir un mismo fichero *State* entre grupos de varias

personas. Terraform soluciona, además, cualquier problema de sincronización que pudiera existir si más de una persona intentan modificar un mismo *State* gracias al uso de *locks*, en caso de que su plataforma *backend* lo permita.

- Protege la información del fichero *State* encriptándola, en caso de que el entorno *backend* lo permita.
- Guarda un control de versiones así, como un historial de ejecuciones.

Ciclo de vida

El ciclo de vida de un proyecto de Terraform consta de las siguientes partes [16].

1. *Inicializar*. El proyecto es inicializado mediante el comando `terraform init`, que automáticamente descarga e instala todos los módulos y *plugins* de los proveedores especificados en el archivo de configuración.
2. *Planificar*. Mediante el comando `terraform plan` se realiza una previsualización de los cambios que se planean realizar sobre la infraestructura.
3. *Aplicar*. Los cambios son ejecutados al lanzar el comando `terraform apply`, que comparará el fichero *State* para crear, actualizar o destruir los recursos pertinentes.
4. *Destruir*. Toda infraestructura previamente creada y guardada en el archivo *State* es destruida a través del comando `terraform destroy`.

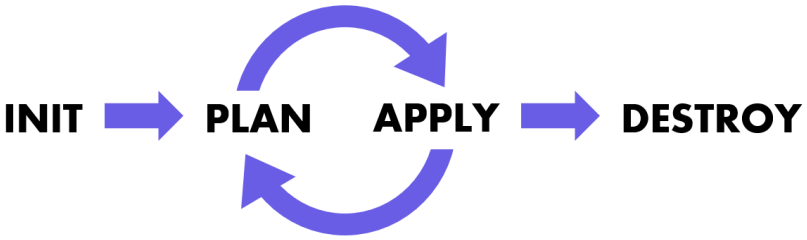

Figura 3.3: Ciclo de vida de un proyecto en Terraform.

3.2.3 HashiCorp Configuration Language (HCL)

Si bien Terraform permite crear archivos de configuración en formato JSON, su uso no está recomendado y es utilizado principalmente para crear archivos programáticamente [17]. El lenguaje principal es HashiCorp Language (HCL).
A lo largo de esta sección se explicarán brevemente las bases de este lenguaje.

Bloques

Los archivos de configuración de este lenguaje están formados por bloques que siguen la estructura que se muestra en el Listado 3.1.

```
<BLOCK_TYPE> "<BLOCK_LABEL>" {
  <IDENTIFIER> = <EXPRESSION> #Arguments
}
```

Listado 3.1: Estructura de bloques en HCL

Todos los bloques están formados por los siguientes elementos [18, 9].

- *Tipo de bloque*. Existen distintos tipos de bloques, más adelante se enumerarán algunos de ellos.
- *Etiquetas de bloque*. Especifican el tipo elegido. Dependiendo del mismo, pueden requerir de muchas o ninguna etiqueta.
- *Argumentos*. Están formados por un `<IDENTIFIER>` que es el nombre del argumento, y un `<EXPRESSION>` que es su valor. Como se explicará en la Sección 3.2.3, pueden ser de muchos tipos, desde `string` hasta otros bloques, creando así una jerarquía de bloques. A excepción de un pequeño número de meta-argumentos, dependen tanto del tipo de bloque como de sus etiquetas.

Por otro lado, cada bloque puede tener una serie de *atributos* para referenciar ciertos aspectos del mismo en otros bloques.

`terraform` El bloque `terraform` es un bloque especial de configuración utilizado para configurar ciertos aspectos de Terraform [19].

```
terraform {
  <IDENTIFIER> = <EXPRESSION> #Arguments
}
```

Listado 3.2: Estructura del bloque `provider` en HCL

Como se puede ver en el Listado 3.2, su estrcutura es muy básica puesto que no contienen ningún `BLOCK_LABEL`, sino únicamento argumentos para configurar Terraform como los siguientes [19].

- El `backend` donde se almacenará el archivo *State*. Tal y como se comentó en la Sección 3.2.2, este puede ser en remoto, como un `s3`; o `local`, que es la opción por defecto.
- El `required_providers`, que configura los diferentes proveedores que se van a utilizar en dicho archivo de configuración, especificando la fuente del *plugin* que se descargará y su versión.

```
terraform {
  backend "s3" {
    bucket = "my-terraform-state"
    key    = "terraform.tfstate"
```

```
    region = "us-east-1"
  }
  required_providers {
    aws = {
      source  = "registry.terraform.io/hashicrop/aws"
      version = "> 3.0.0"
    }
  }
}
```

Listado 3.3: Ejemplo de uso del bloque `terraform` donde se configura un *bucket* de *S3* llamado "`my-terraform-state`". Además, se especifica el uso de un *plugin* de `aws`[3] que se obtendrá a través de `registry.terraform.io/hashicrop/aws`[4] con cualquier versión superior a la `3.0.0`.

Como se puede observar en el Listado 3.3, los dos argumentos utilizados son bloques en si mismos, con su propia estructura y argumentos definidos. Esta es la jerarquía de bloques que se mencionó anteriormente.

`provider` Como ya se mencionó en la Sección 3.2.1, los proveedores son *plugins* externos a Terraform que nos permiten interactuar con cualquier servicio. Dentro de este bloque es posible configurar ciertos aspectos de todos aquellos entornos que fueron previamente añadidos en el bloque `terraform` [20].

```
provider "<PROVIDER_NAME>" {
  <IDENTIFIER> = <EXPRESSION> #Arguments
}
```

Listado 3.4: Estructura del bloque `provider` en HCL

Como se puede ver en el Listado 3.4, siguen la siguiente estructura [20].

- `<PROVIDER_NAME>`. Corresponde al nombre del proveedor asignado previamente en el bloque `terraform` al incluir los provedores a utilizar en `required_providers`.
- `Argumentos`. Configuran aspectos del entorno, por ejemplo especificando una región, o las credenciales necesarias para acceder a sus servicios. Sin embargo, es necesario refererirse a la documentación de cada uno de los proveedores para conocer sus argumentos específicos[5].

```
provider "aws" {
```

[3] Si bien es posible elegir cualquier otro tipo de nombre, es una buena práctica usar el recomendado por el proveedor

[4] Esta fuente corresponde al repositorio público de AWS en Terraform Registry. En la práctica, se suele omitir el *hostname* `registry.terraform.io`, pues Terraform lo añade automáticamente, y solo es usado en caso de utilizar algún otro tipo de repositorio

[5] Terraform permite crear más de un bloque `provider` por cada entorno, por ejemplo para que cada uno use una región distinta. Para ello es necesario dotarles de un meta-argumento `alias` que sí es común para todos

```
2     region = "eu-west-1"
3   }
```

Listado 3.5: Ejemplo de uso del bloque `provider` para configurar un proveedor de AWS en la región de `eu-west-1`[7]

resource Los `resource` o recursos, son un tipo de bloque fundamental en Terraform. Estos representan una infraestructura de cualquier proveedor incluido previamente, como una una red virtual o una instancia de máquina virtual [21].

```
1   resource "<RESOURCE_TYPE>" "<RESOURCE_NAME>" {
2     <IDENTIFIER> = <EXPRESSION> #Arguments
3   }
```

Listado 3.6: Estructura del bloque `resource` en HCL

Como se puede ver en el Listado 3.6, siguen la siguiente estructura [21].

- `<RESOURCE_TYPE>`. Es el tipo de recurso que ofrece cada `provider`, por lo tanto habrá que referirse a su documentación para conocer todas sus opciones.
- `<RESOURCE_NAME>`. Es el nombre que se asignará a dicho `resource` dentro de ese archivo de configuración de Terraform.
- `Argumentos`. Configuran el recurso a crear. Como ya ocurría con el bloque `provider`, la mayoría de argumentos que puede recibir cada `resource` dependen del `provider` que lo implementa, por lo tanto es necesario referirse a su documentación para conocerlos con detalle. Sin embargo, existen una serie de meta-argumentos que cambian ligeramente su comportamiento y que son comunes a todos bloques de este tipo. Son los siguientes [21].
  - `depends_on`. Si bien Terraform gestiona automáticamente las dependencias entre recursos, es posible hacer uso de este argumento para especificar algunas explícitamente.
  - `count`. Permite crear múltiples instancias de un mismo recurso.
  - `for_each`. Similar a `count` pero haciendo uso de listas o mapas.
  - `provider`. Este meta-argumento especifica cual de todos los `provider` utilizar, en caso de que se haya añadido más de uno.
  - `lifecycle`. Consiste en un bloque que permite configurar ciertos aspectos más específicos del ciclo de vida de un recurso:
    - `create_before_destroy`. En el caso de que para actualizar un recurso sea necesario destruirlo y crearlo nuevamente, este meta-argumento establece si se creará un remplazo antes de su destrucción.

[7] Este identificador corresponde con la región de Europa (Irlanda)

- `prevent_destroy`. Establece si proteger al recurso de ser destruido bajo cualquier concepto.
- `ignore_changes`. Lista de argumentos cuyos cambios serán ignorados por parte de Terraform.
- `replace_triggered_by`. Lista de argumentos cuyos cambios harán que dicho recurso sea remplazado por otro nuevo.

Por último, los `resource` tienen una serie de atributos definidos en su documentación correspondiente, y que pueden ser referenciados en otros bloques a través de: `<RESOURCE_TYPE>.<RESOURCE_NAME>.<ATTRIBUTE>`.

```
resource "aws_instance" "this" {
  count         = 3

  ami           = "ami-0b46fb811cb1d78e9"
  instance_type = "t2.micro"

  tags = {
    Name = "myEC2-${count.index}"
  }

}
```

Listado 3.7: Ejemplo de uso del bloque `resource` para crear 3 instancias de EC2 en AWS con una AMI[9] y un tipo de instancia específico

data De la misma manera que los `provider` ofrecen `resource` para crear, actualizar o eliminar infraestructura, estos también ofrecen bloques `data` para obtener información de estos servicios [22].

```
data "<DATA_TYPE>" "<DATA_NAME>" {
  <IDENTIFIER> = <EXPRESSION> #Arguments
}
```

Listado 3.8: Estructura del bloque `data` en HCL

Como se puede ver en el Listado 3.8, siguen la siguen estructura [22].

- `<DATA_TYPE>`. Es el tipo de dato que se va a obtener de cada `provider`, por lo tanto habrá que referirse a su documentación para conocer todas sus opciones.
- `<DATA_NAME>`. Es el nombre que se asignará a dicho `data` dentro de ese archivo de configuración Terraform.
- `Argumentos`. Configuran la información a obtener. Dependen del `provider`, por lo que habrá que referirse a su documentación para conocerlos en detalle.

[9] Corresponde a un Ubuntu 20.04

Al igual que ocurría con los bloques `resource`, cuentan con una serie de atributos definidos en su documentación, que en este caso corresponderán con la información obtenida desde el entorno. Se puede referenciar mediante: `data.<DATA_TYPE>.<DATA_NAME>.<ATTRIBUTE>`

```
data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
}

resource "aws_instance" "this" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t1.micro"
}
```

Listado 3.9: Ejemplo de uso del bloque `data` para obtener en AWS una AMI de Ubuntu 20.04 de Canonical con HVM como tipo de virtualización, que luego será utilizada para crear una instancia de EC2

Expresiones

La expresiones en Terraform se utilizan para representar los valores que toman los argumentos dentro del archivo de configuración. Estos pueden ser de varios tipos[10] como [23]:

- *Tipos primitivos*.
  - `string`
  - `number`
  - `bool`
  - `null`
- *Tipos complejos*. Están compuestos de varios tipos primitivos
  - `list` o `tuple`. Consiste en una serie de valores encerrados por corchetes y separados por comas que pueden tomar cualquier tipo, como por ejemplo `["a" , 2, true]`.
  - `set`
  - `map` o `object`. Conjunto de pares `<KEY>=<VALUE>`.

[10] En la práctica, Terraform acaba convirtiendo todos los tipos a `string`. Sin embargo, es recomendable utilizar cada tipo para su uso correspondiente

Sin embargo, HCL nos ofrece mecanismos algo más complejos para trabajar con todos estos tipos como operadores, funciones, condicionales, bucles o bloques dinámicos.

Módulos

Terraform organiza sus proyectos en contenedores de recursos llamados módulos. Todo proyecto de Terraform tiene siempre al menos un módulo llamado *módulo raíz*, que a su vez puede llamar a otros módulos llamados *módulos hijos* [24].
En resumidas cuentas, su funcionamiento es similar al de los métodos en los lenguajes de programación tradicionales. De la misma forma que estos lenguajes declaran un método `main` que realiza ciertas acciones y que a su vez puede llamar a otros métodos, todo proyecto Terraform tiene un *módulo raíz* que contiene varios `resource` y que a su vez puede llamar a otros *módulos hijos*.
Asimismo, los módulos pueden recibir variables de entrada y retornar unos valores de salida, que se referencian con `module.<MODULE_NAME>.<OUTPUT_VARIABLE>`.

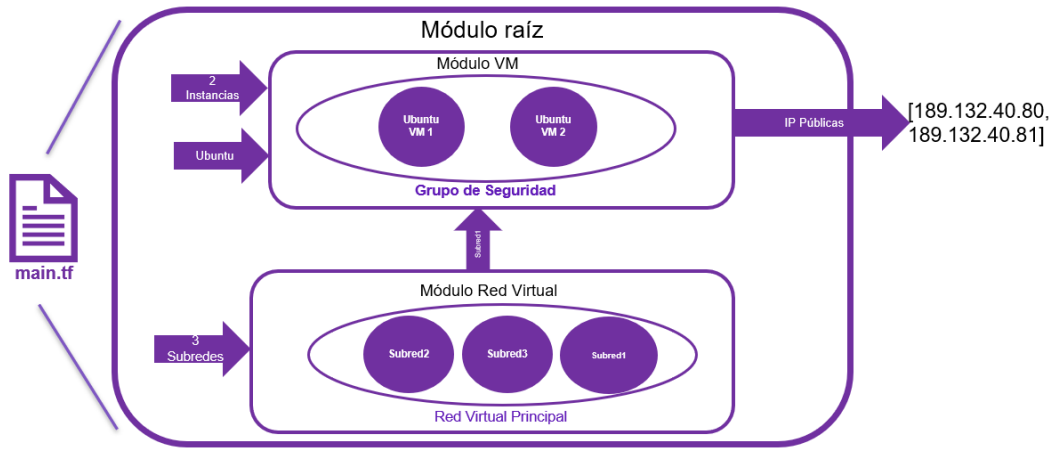

Figura 3.4: Ejemplo de uso de módulos en Terraform

En la Figura 3.4 podemos ver un ejemplo de uso de esta característica. En este caso, un *módulo raíz* llama a dos *módulos hijos* para crear, en primer lugar, una red virtual con un número de subredes especificado mediante argumento, y en segundo lugar, otra serie de máquinas virtuales con una configuración especificada también mediante argumentos.
Observa como el módulo *VM* reutiliza la variable de salida del módulo *Red Virtual*, y el módulo raíz a su vez reutiliza la salida de este último para devolver las IPs públicas de las máquinas virtuales creadas.

El Listado 3.10 muestra el código equivalente a dicho ejemplo en HCL. Como se puede ver, los `module` son un nuevo tipo de bloque muy similar a los que ya se vieron anteriormente. Reciben como argumento una serie de variables de entrada especificados en cada módulo, aunque también pueden recibir una serie de meta-argumentos de los que ya se habló anteriormente, como `count`, `depends_on` o `for_each`, además de uno nuevo y muy importante,

`source` [25]. Este argumento determina la localización de dichos módulos. En este caso se encuentran dentro de un directorio local, pero cualquier usuario puede publicar módulos en repositorios como Terraform Registry para que sean reutilizados por otros [26].

```
module "virtual_network" {
  source = "./modules/virtual_network"

  number_subnets = 3
}

module "VM" {
  source = "./modules/vm"

  number_instances = 2
  image = data.images.ubuntu
  subnet = module.virtual_network.subnets[1]
}

output "public_ips" {
  value = module.vm.public_ips
}
```

Listado 3.10: Código en HCL de la Figura 3.4

Valores de entrada, salida y locales

Como ya se comentó en la anterior sección, los módulos pueden recibir una serie de valores de entrada y de salida, pero también pueden contener valores locales.

Valores de entrada Sirven como parámetros de entrada para los módulos de Terraform y se crean a partir del bloque `variable`, tal como se muestra en el Listado 3.11, donde contienen los siguientes elementos [27].

- `<VARIABLE_NAME>`. Es el nombre de la variable de entrada.
- `Arguments`. Se definen los siguientes argumentos:
  - `default`. Valor por defecto asignado en caso de que no se le atribuya ningún otro.
  - `type`. Determina su tipo de los que ya se mencionó en la Sección 3.2.3.
  - `description`. Añade una descripción informativa a la variable.
  - `validation`. Permite añadir reglas lógicas que validen los valores asignados a las variables.
  - `sensitive`. Indica si el argumento contiene datos sensibles que no deban ser mostrados en pantalla al realizar un `terraform plan` o un `terraform apply`.
  - `nulleable`. Indica si la variable puede tomar valor `null`.

Se referencian siguiendo la sintaxis: `var.<VARIABLE_NAME>`.

```
variable "<VARIABLE_NAME>" {
  <IDENTIFIER> = <EXPRESSION> #Arguments
}
```

Listado 3.11: Estructura del bloque `variable` en HCL

Estas variables podrán ser *seteadas* de varias formas, como manualmente al llamar un módulo (véase Liastdo 3.10) o al lanzar un `terraform apply`, mediante variables de entorno `TF_VAR_<VARIABLE_NAME>` o desde un fichero `.tfvars`.

```
variable "image" {
  type        = string
  description = "Image ID to use for the virtual machines."
  validation {
    condition     = length(var.image) > 4 && substr(var.image, 0, 4) == "ami-"
    error_message = "Image value must be a valid AMI id, starting with \"ami-\"."
  }
  sensitive       = false
  nulleable       = false
}
```

Listado 3.12: Ejemplo del bloque `variable` en HCL

Valores de salida Son los variables de retorno para los módulos, de manera que dicha información pueda ser reutilizada por otros módulos o mostrada en pantalla. Se crean a partir del bloque `output` tal como se muestra en el Listado 3.13, donde contienen los siguientes elementos [28].

- `<LOCALS_NAME>`. Es el nombre de la variable local.
- `Arguments`. Se definen los siguientes argumentos:
  - `value`. Referencia al atributo que servirá como valor para la variable de salida.
  - `precondition`. Establece una precondición a ser cumplida para que exista esta variable.
  - `description`. Añade una descripción informativa a la variable.
  - `sensitive`. Indica si el valor contiene datos sensibles que no deban ser mostrados en pantalla al realizar un `terraform plan` o un `terraform apply`.
  - `depends_on`. Establece una dependencia con algún `resource` en caso de ser necesario.

```
output "<OUTPUT_NAME>" {
  <IDENTIFIER> = <EXPRESSION> #Arguments
}
```

Listado 3.13: Estructura del bloque `output` en HCL

Es posible consultar estos valores de salida una vez se ejecuta un `terraform apply` haciendo uso de `terraform output [OUTPUT_NAME]`, o reutilizarlos en otros módulos con `<MODULE_NAME>.<OUTPUT_NAME>`.

```
output "public_ips" {
  value       = aws_instance.server.private_ip
  description = "Public IP address of the EC2 instance."
  sensitive   = false
}
```

Listado 3.14: Ejemplo del bloque `output` en HCL

Variables locales Son variables reutilizables dentro de un mismo módulo. Se crean a partir del bloque `locals` que sigue la estructura mostrada en el Listado 3.15, donde contienen los siguientes elementos [29].

- `Arguments`. En este caso los argumentos del bloque representan a cada una de las variables locales que se van a crear, donde `<IDENTIFIER>` es el nombre de la variable, y `<EXPRESSION>` su valor.

```
locals {
  <IDENTIFIER> = <EXPRESSION> #Arguments
}
```

Listado 3.15: Estructura del bloque `locals` en HCL

Se referencian a través de `local.<IDENTIFIER>`.

```
locals {
  image                  = "Ubuntu_20.04"
  number_instances       = 3
}
```

Listado 3.16: Ejemplo del bloque `locals` en HCL

3.2.4 Terraform frente a sus alternativas

Como ya se comentó en la sección 3.1.2, existen una gran variedad de tipos de herramientas de Infraestructura como Código. A lo largo de este apartado, se compararán algunas de estas como alternativas a Terraform.

Interfaces web

Una metodología de trabajo clásica para aprovisionar infraestructura es hacer uso de las interfaces web que facilitan los proveedores *Cloud* y que guían al usuario en todo momento en los pasos necesarios para aprovisionarse de la infraestructura deseada. Se trata de la manera más simple y sencilla de crear y gestionar recursos, y por lo tanto es mucho más apropiada

para principiantes en la plataforma que no tienen demasiados conocimientos sobre la misma, o para proyectos pequeños que requieran rápidamente de una infraestructura más básica.

Sin embargo, según la magnitud del entorno aumenta, su uso puede volverse muy tedioso. Su incapacidad a la hora de automatizar el aprovisionamiento y las pocas opciones ofrecidas para replicar infraestructura, acaban dificultando bastante la creación de entornos complejos.

A pesar de todo esto, siguen siendo una herramienta imprescindible para gestionar los recursos que se han aprovisionado previamente mediante Terraform.

Scripts

Todos los servicios *Cloud* ofrecen una serie de herramientas de CLI para aprovisionar infraestructura mediante comandos. Esto permite la creación de *scripts* que solventarían algunos de los problemas que ofrecían las interfaces web, como la automatización de infraestructura, permitiéndonos crear así entornos algo más complejos más facilmente.

Sin embargo, y como ya se comentó en la sección 3.1.2, hacer uso de *scripts* puede traer consigo ciertos problemas graves. Estas herramientas requieren un conocimiento mucho más avanzado del entorno Cloud y de su CLI, de las dependencias de sus recursos, de sus distintas versiones, etc. Esto acaba resultando en archivos demasiados complejos de construir y mantener.
Terraform soluciona la mayoría de estos problemas al ofrecer una sintaxis unificada para todos los entornos *Cloud*, donde solo seria necesario aprender las características propias de cada recurso, abstrayendo así al usuario de conocer las dependencias entre estos. De este modo, el foco vuelve a estar en la infraestructura y no en la herramienta.

Herramientas de configuración

Chef, Puppet y Ansible son algunos ejemplos de herramientas de configuración que, en ciertos casos, nos pueden permitir obtener un cierto grado de aprovisionamiento pero cuyo uso general es la configuración de la infraestructura [30]. Por lo tanto, estas no son alternativas a herramientas de aprovisionamiento como Terraform sino, como veremos más adelante, herramientas complementarias que pueden trabajar conjuntamente.

Herramientas de aprovisionamiento *Cloud-Specific*

Herramientas como CloudFormation, Azure Resource Manager o Google Cloud Deployment Manager, permiten, al igual que Terraform, la creación de infraestructura a través de archivos de configuración, y por lo tanto ofrecen muchas de las ventajas que ya encontrábamos en esta herramienta.

Sin embargo, estas ventajas chocan directamente con el inconveniente de estar diseñadas para trabajar con un *Cloud* en específico, donde Terraform destacaba enormemente ofrecien-

do un entorno unificado para todos los servicios *Cloud* [31].

Cloud Development Kit (CDK)

Además de las herramientas de aprovisionamiento que se mencionaron anteriormente, algunos entornos Cloud como AWS ofrecen un *framework* para definir infraestructura mediante lenguajes de programación tradicionales como Java, Python o TypeScript. Estos, no dejan de ser transpiladores, es decir, acaban traduciendo el código fuente de estos lenguajes a un formato intermedio entendido por sus herramientas de aprovisionamiento propias. Por lo que son una capa de abstracción más para aquellos que no quieran aprender la forma de trabajo de todas estas herramientas [32].

Esto puede suponer una ventaja debido a que desaparece la necesidad de aprender una nueva herramienta, y facilita el uso de mecanismos propios de estos lenguajes tradicionales como los bucles y los condicionales, y la creación de *tests*.
Sin embargo, de nuevo encontramos los problemas que suponía trabajar con un entorno diseñado para funcionar con una única plataforma *Cloud*. Frente a esto HashiCorp está desarrollando actualmente su propio CDK para aquellos que prefieran trabajar mediante lenguajes de programación tradicionales [33].

Otras herramientas de aprovisionamiento *Cloud-Agnostic*

Al igual que hace Terraform, existen otras herramientas que trabajan con independencia del entorno *Cloud*. La más famosa de ellas es Pulumi.

Esta herramienta de código abierto guarda muchas similitudes con respecto a Terraform, como sus paradigmas de programación declarativa y de infraestructura inmutable. Su principal diferencia frente a la herramienta de HashiCorp es el uso de lenguajes de programación de uso general como son Python, JavaScript o TypeScript, al igual que hacian los CDK pero sin transpilador de por medio [34].
Es por todo esto, que Pulumi se está postulando como la mayor alternativa a Terraform.

Por contra, Pulumi es una herramienta realmente nueva y sin una comunidad tan grande como la de Terraform, que ya se ha estandarizado bastante dentro de la industria.
Además, muchos profesionales concuerdan en que, pese a que Pulumi ofrece un servicio gratuito, su versión premium es imprescindible para ser usado en producción [9].

Por lo tanto, la decisión de elegir Terraform o Pulumi es puramente personal. Terraform es una herramienta mucho más madura y con un gran servicio gratuito, pero es posible que muchos usuarios prefieran no tener que aprender un lenguaje nuevo y consideren que usar los lenguajes de programación clásicos es la mejor opción para su proyecto. Aunque, como ya se comentó anteriormente, Terraform se encuentra actualmente diseñando su propio CDK para esta masa de usuarios que no vean oportuno aprender su HCL. Por lo tanto, aún está por ver cuál será el futuro de Pulumi cuando se acabe lanzando esta herramienta.

3.2.5 Patrones comunes de trabajo

Tal y como se mencionó anteriormente, muchas de las herramientas de IAC vistas no son alternativas a Terraform, sino herramientas complementarias que pueden trabajar conjuntamente con ella para ofrecer un producto aún mejor. A lo largo de esta sección se explicarán algunos de los patrones de trabajo más comunes entre estas herramientas.

Terraform + Herramientas de Configuración

Terraform es una herramienta fantástica para el aprovisionamiento de infraestructura, sin embargo, y aunque como se mencionó si que ofrece mecanismos para la configuración de dicha infraestructura, su finalidad principal no es esta.
Por lo tanto, es común usar Terraform para aprovisionarnos de la infraestructura necesaria, como una serie de máquinas virtuales, a la vez que hacemos uso de herramientas de configuración como Ansible para instalar todo el *software* necesario dentro de la infraestructura contratada [9].

Terraform + Herramientas de *Server Templating*

También es posible hacer uso de Terraform con herramientas de *Server Templating*. De este modo se utilizarían estas herramientas para configurar la imagen virtual a partir de la cual se crearían la infraestructura a través de Terraform.
Así, a diferencia del anterior patrón donde se aprovisionaba de una infraestructura, y luego se instalaba y configuraba su *software*, aquí es posible precargar todo aquello en una imagen virtual que será utilizada al crear la infraestructura a través de Terraform [9].

Terraform + Herramientas de Orquestación

También es posible hacer uso de Terraform para el aprovisionamiento de por ejemplo, unos clústers de EKS en AWS; para luego usar una herramienta de orquestación como sería Kubernetes en este caso, para definir cómo nuestra aplicación es desplegada y gestionada en dicho servicio *Cloud* [9].

Capítulo 4

Metodología

A lo largo de este capítulo se abordará la metodología de trabajo empleada para la correcta planificación y desarrollo del proyecto.

4.1 Metodología ágil

Ágil es un conjunto de métodos y metodologías que comparten una serie de valores y principios aplicables a muchas áreas dentro de la ingeniería del *software* como la gestión de proyectos o el diseño y arquitectura *software* [35].

Tradicionalmente, las empresas hacían uso de metodologías en cascada a la hora de gestionar sus proyectos. Este consistía en un proceso mucho más lineal donde se recogían unos requisitos, se realizaba un diseño y se implementaba un *software*. Sin embargo, con el paso del tiempo, el proceso de creación de *software* se volvió demasiado complejo, las necesidades del cliente cambiaban, y la metodología en cascada quedaba anticuada [36].

Es entonces en 2001, cuando un grupo de expertos se reunieron para buscar una solución a este problema, y nació el término «ágil». Todos coincidieron en que las empresas estaban tan centradas en planificar y documentar sus sistemas que habían perdido el foco en lo realmente importante, complacer a sus clientes. Su reacción fue la creación del *Manifiesto Ágil*, un compendio de 12 principios que agrupan 4 valores que cambiaron por completo el modo en que se gestionan los proyectos [36].

- *Individuos e iteraciones por encima de procesos y herramientas*. Es más importante contar con un buen grupo de trabajo con los conocimientos técnicos adecuados, facilidad de adaptación y la capacidad de trabajar en equipo que las herramientas y los procesos.

- *Software funcionando por encima de la documentación extensiva*. Si bien la documentación es una parte importante del proceso de creación de *software*, estos deben ser cortos y limitarse a lo fundamental.

- *Colaboración con el cliente por encima de la negociación contractual*. Más que un ambiente de enfrentamiento en el cual las partes buscan su beneficio propio, se busca la participación constante del cliente, desde el comienzo hasta la culminación del

proyecto, y su interacción con el equipo de desarrollo.

- *Respuesta ante el cambio por encima de seguir un plan*. Dada la naturaleza cambiante de la tecnología y la sociedad moderna, se busca la creaciones de planes flexibles para poder adaptarse a los cambios que puedan surgir.

4.1.1 Metodología ágil frente a metodologías tradicionales

Como se puede observar en la comparativa realizada en la Tabla 4.1, las metodologías tradicionales ofrecen una forma de trabajo mucho más estricta y con menor implicación entre los miembros del grupo de trabajo.

Parámetro	Ágil	Tradicional
Adaptabilidad	Más adaptable y flexible para adaptarse a las necesidades que vayan surgiendo	Planificación más estricta y predecible
Requisitos	Requisitos pueden cambiar a lo largo del proyecto	Requisitos no cambian durante el proyecto
Implicación del cliente	Alta	Baja
Tamaño de grupos	Pequeños	Grandes
Comunicación	Comunicación informal en encuentros cara a cara	Comunicación formal a través de escritos
Acercamiento	Iterativo	Linear
Modelo de desarrollo	Entrega de evolutivos	Ciclo de vida
Riesgo	Mayor	Menor

Cuadro 4.1: Comparativa metodologías ágiles y tradicionales

Esto puede suponer en muchos aspectos un riesgo algo menor, pues siempre se tiene en mente cual es el objetivo final del proyecto y cuales son los pasos precisos para alcanzarlos. Sin embargo, también requiere de un conocimiento de la materia en la que se va a trabajar mucho mayor, con el fin de realizar todas las estimaciones con la máxima exactitud, además de saber con precisión cuales son los límites de tu proyecto.

4.2 Metodología de desarrollo de este proyecto

Tras los puntos expuestos en los apartados anteriores, y dada la naturaleza del Trabajo de Fin de Grado, se ha optado por seguir una metodología ágil.

En primer lugar, porque al desarrollar el trabajo en el marco de unas prácticas, el alumno desarrollará su proyecto acompañado tanto de un tutor de la escuela, como de un tutor de empresa que podrán guiar al alumno a lo largo de cortas reuniones diarias o semanales sobre el desarrollo de su proyecto.
Y en segundo lugar, porque al tratar sobre tecnologías algo desconocidas para el alumno, es

difícil crear una primera planificación estricta, ya que es posible que surjan distintos cambios a lo largo del proyecto.

Por lo tanto, la primera tarea que se llevó a cabo fue definir el alcance del proyecto que se puede consultar en la Sección 2.1. A continuación se desarrolló un *backlog* o lista de tareas a realizar, que fue ampliándose y especificándose a medida que el proyecto avanzaba. A estas tareas se les asignó un nivel de esfuerzo medido en días laborables (6 horas/día) y se fueron organizando en ciclos de trabajo llamados *sprint*.

4.2.1 *Sprint 1*. Familiarización con los entornos

Debido a que el alumno carecía de conocimientos en algunos de los entornos con los que se iba a trabajar, se decidió dedicar un *Sprint* a la familiarización con los entornos *Cloud* y con Terraform.

El objetivo final de este *Sprint* es la creación de un documento informativo sobre Terraform y sus características. Para ello se dividió el *Sprint* en las siguientes tareas:

- *Introducción a la* IAC. Aprendizaje sobre qué es la Infraestrcuture as Code (IAC), cuáles son sus ventajas, características y distintas herramientas que abarca.
  Estimación de esfuerzo: 2.

- *Introducción al Cloud Computing*. Aprendizaje básico sobre en qué consiste el *Cloud Computing*.
  Estimación de esfuerzo: 2.

- *Familiarización con* AWS. Estudio sobre las bases del entorno *Cloud* de Amazon y los distintos servicios que ofrece. Probar su métodos de aprovisionamiento clásico mediante la interfaz web para compararlo frente a métodos IAC.
  Estimación de esfuerzo: 3.

- *Familiarización con Azure*. Estudio sobre las bases del entorno *Cloud* de Microsoft y los distintos servicios que ofrece. Probar su métodos de aprovisionamiento clásico mediante la interfaz web para compararlo frente a métodos IAC.
  Estimación de esfuerzo: 3.

- *Familiarización con Google Cloud Platform*. Estudio sobre las bases del entorno *Cloud* de Google y los distintos servicios que ofrece. Probar su métodos de aprovisionamiento clásico mediante la interfaz web para compararlo frente a métodos IAC.
  Estimación de esfuerzo: 3.

- *Estudio sobre Terraform*. Estudio sobre la herramienta que se va a utilizar durante la práctica. Aprendizaje sobre sus características, su funcionamiento y alternativas.
  Estimación de esfuerzo: 6.

4.2.2 *Sprint 2*. Creación primer nivel de la arquitectura:

Tras obtener una base sobre los conceptos necesarios en los entornos y herramientas con los que se va a trabajar, se comienza a realizar un primer nivel básico sobre la arquitectura prevista.

El objetivo final de este *Sprint* es la creación de un nivel base de infraestructura formado únicamente por una máquina virtual pública y accesible directamente desde Internet, y que correrá un servidor web en el puerto 8080. Para ello se dividió el *Sprint* en 3 tareas, una por entorno *Cloud*.

- *Planificación y desarrollo de un primer nivel básico en* AWS.
  Estimación de esfuerzo: 1,5.
- *Planificación y desarrollo de un primer nivel básico en Azure.*
  Estimación de esfuerzo: 1,5.
- *Planificación y desarrollo de un primer nivel básico en* GCP.
  Estimación de esfuerzo: 1,5.

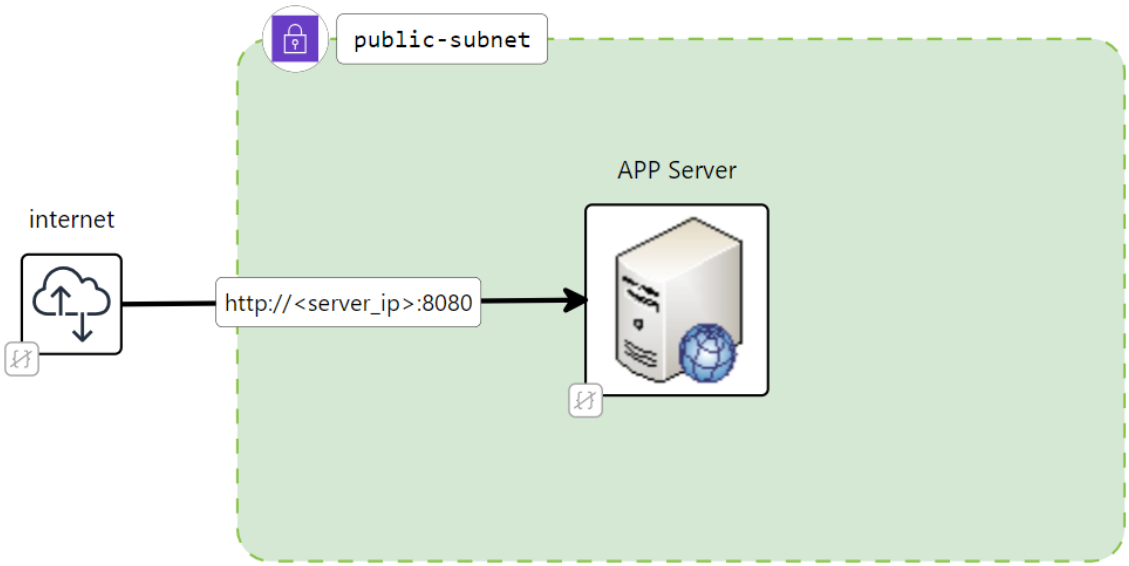

Figura 4.1: Diagrama de la arquitectura a replicar en el *Sprint 2*.

4.2.3 *Sprint 3*. Creación de un segundo nivel de la arquitectura.

Durante este nivel se pretende mejorar la disponibilidad del nivel previamente planteado. Por lo tanto, el objetivo final de este *Sprint* es la sustitución de la máquina virtual por un grupo de instancias con capacidad de escalar dependiendo del estado de las mismas. Además, se añadirá un balanceador de carga que opere en la capa de aplicación y que distribuya el tráfico web entra las distintas máquinas. Para ello se dividió el *Sprint* en 3 tareas, una por entorno *Cloud*.

- *Planificación y desarrollo del segundo nivel en* AWS.
  Estimación de esfuerzo: 2,3.
- *Planificación y desarrollo del segundo nivel en Azure.*
  Estimación de esfuerzo: 2,3.
- *Planificación y desarrollo del segundo nivel en* GCP.
  Estimación de esfuerzo: 2,3.

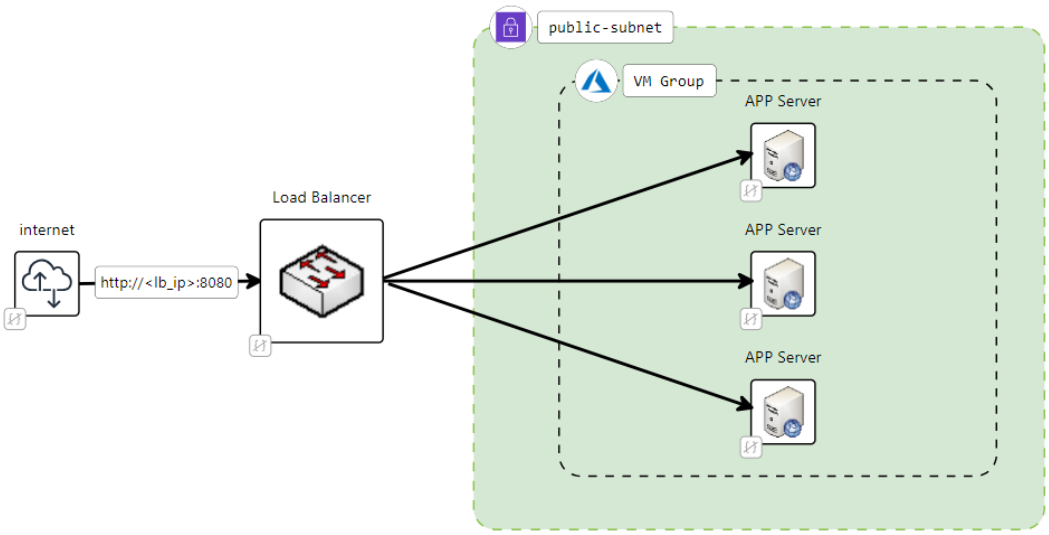

Figura 4.2: Diagrama de la arquitectura a replicar en el *Sprint 3*.

4.2.4 *Sprint 4*. Creación de un tercer nivel de la arquitectura.

Durante este nivel se pretende añadir algo más de complejidad a la arquitectura y a los servicios web desplegados.
Por lo tanto, el objetivo final de este *sprint* es la adición de una base de datos que únicamente será accesible a través de las máquinas virtuales. Además, se desarrollará un microservicio básico con *Spring Boot* que será desplegado en las diferentes instancias con el fin de probar el correcto funcionamiento de la base de datos.

- *Planificación y desarrollo del tercer nivel en* AWS.
  Estimación de esfuerzo: 2,3.
- *Planificación y desarrollo del tercer nivel en Azure.*
  Estimación de esfuerzo: 2,3.
- *Planificación y desarrollo del tercer nivel en* GCP.
  Estimación de esfuerzo: 2,3.
- *Creación y despliegue de un microservicio básico en cada una de las arquitecturas construidas.*

Estimación de esfuerzo: 2.

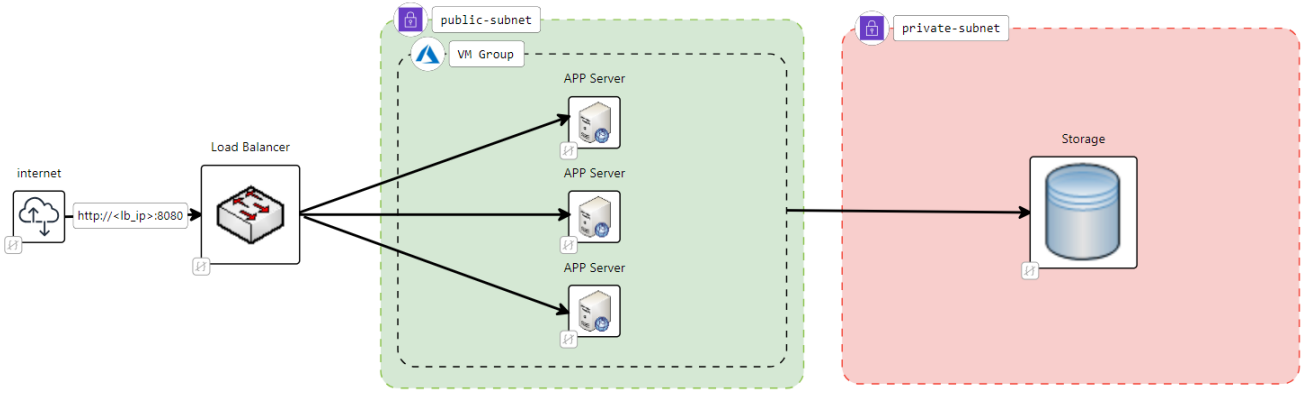

Figura 4.3: Diagrama de la arquitectura a replicar en el *Sprint 4*.

4.2.5 *Sprint 5*. Creación de un cuarto nivel de la arquitectura.

Durante este nivel se busca aumentar la seguridad de las máquinas virtuales creadas hasta el momento.

Por lo tanto, el objetivo final de este *Sprint* es la sustitución de la subred pública de servidores por una privada, de modo que únicamente sean accesibles mediante HTTP a través del balanceador de carga, y mediante SSH a través de un nuevo nodo *jumpbox* que solo aceptará peticiones de este tipo.

- *Planificación y desarrollo del cuarto nivel en* AWS.
  Estimación de esfuerzo: 2,3.
- *Planificación y desarrollo del cuarto nivel en Azure.*
  Estimación de esfuerzo: 2,3.
- *Planificación y desarrollo del cuarto nivel en* GCP.
  Estimación de esfuerzo: 2,3.

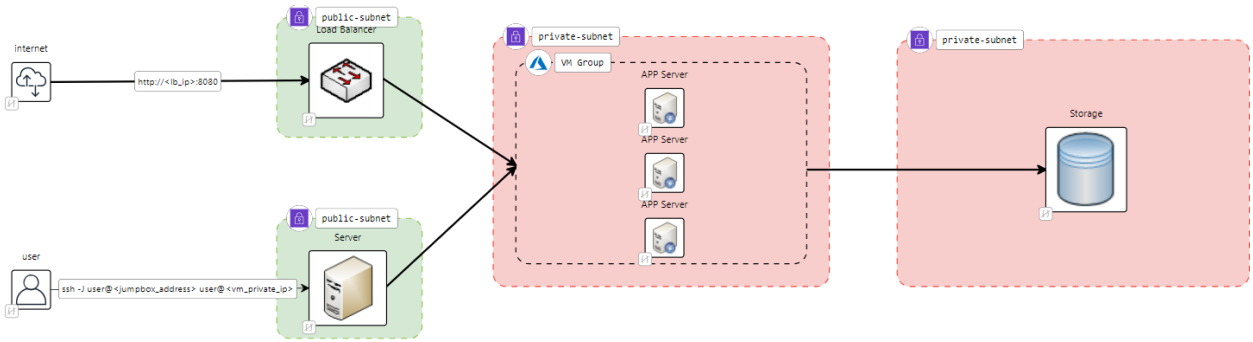

Figura 4.4: Diagrama de la arquitectura a replicar en el *Sprint 5*.

4.2.6 *Sprint 6*. Creación de un quinto nivel de la arquitectura.

A través de este último nivel se pretende aumentar algo más la complejidad del sistema. Por lo tanto, el objetivo final de este *Sprint* es la sustitución de las máquinas virtuales por servicios de contenedores Docker. Además, se desarrollará un nuevo microservicio de autenticación de manera que haya tres tipos de comunicaciones en el sistema: usuario-servicio, servicio-servicio, servicio-base de datos.

- *Planificación y desarrollo del quinto nivel en* AWS.
  Estimación de esfuerzo: 2,3.
- *Planificación y desarrollo del quinto nivel en Azure.*
  Estimación de esfuerzo: 2,3.
- *Planificación y desarrollo del quinto nivel en* GCP.
  Estimación de esfuerzo: 2,3.
- *Creación y despliegue de un microservicio de autenticación en cada una de las arquitecturas construidas.*
  Estimación de esfuerzo: 2.

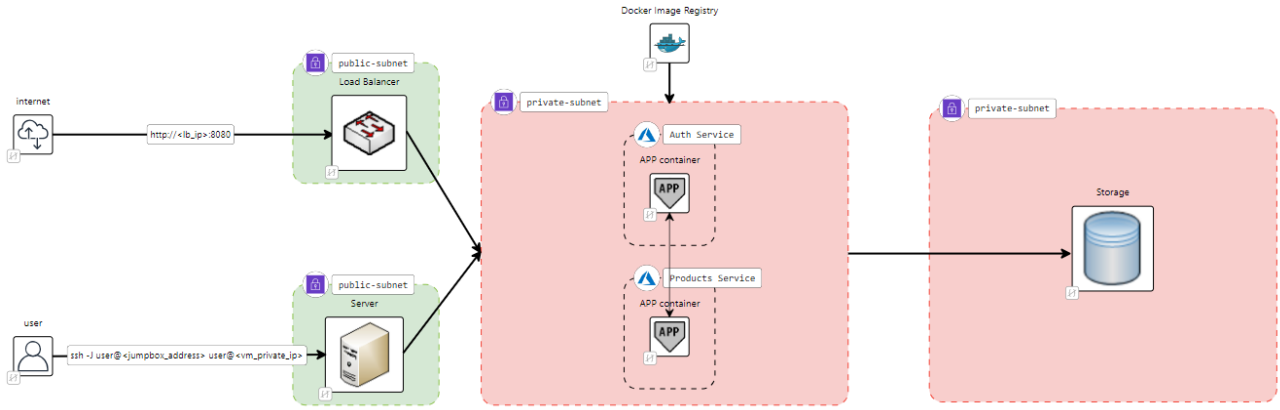

Figura 4.5: Diagrama de la arquitectura a replicar en el *Sprint 6*.

4.2.7 *Sprint 7*. Comparación de los distintos entornos.

Con este último *Sprint* se pretende realizar una comparación final entre los tres entornos con los que se ha trabajado a lo largo de este proyecto.
El objetivo final es la creación de un documento que contenga una comparación sobre el funcionamiento, características, componentes y coste de cada uno de los servicios utilizados durante el trabajo.

- *Comparación de los servicios de cómputo mediante máquinas virtuales.*
  Estimación de esfuerzo: 2.
- *Comparación de los servicios de red.*
  Estimación de esfuerzo: 2.

- *Comparación de los servicios de base de datos*.
  Estimación de esfuerzo: 2.
- *Comparación de los servicio de contenedores*.
  Estimación de esfuerzo: 2.

Capítulo 5

Análisis, Diseño e Implementación de las Arquitecturas

A lo largo de este capítulo se abordará el análisis, diseño e implementación de una arquitectura de microservicios mediante Terraform en tres entornos Cloud: AWS, Azure y GCP, guiado a través de distintos niveles tal y como se comentó en el capítulo anterior. Si bien se explicarán ciertos detalles de los `resource` usados de cada proveedor, no se dará una explicación exhaustiva de todos sus posibles argumentos y posibilidades, pues para ello ya existe su documentación correspondiente que puede ser accedida a través de: `https://registry.terraform.io/`.

5.1 Primer nivel de la arquitectura

5.1.1 Amazon Web Services (AWS)

Recursos necesarios

VPC Es el servicio de redes virtuales de AWS donde se encuentran el resto de recursos. Está compuesto principalmente por los siguientes elementos [37]:

- *Subredes*. Se trata de una separación lógica dentro de la VPC, con una serie de direcciones IP privadas propias. Cada *Subred* tiene asociada una *Tabla de Enrutamiento* que especifica las rutas permitidas para el tráfico saliente de la *Subred*. Para proteger el tráfico, pueden ser:

  - *Públicas*. La *Subred* tiene una ruta directa hacia el *Gateway de Internet*.
  - *Privada*. La *Subred* no tiene ruta directa hacia el *Gateway de Internet*. Por lo tanto requerirían de una *Puerta NAT* para su salida hacia fuera de la VPC.

- *Tablas de Enrutamiento*. Conjunto de reglas que determinan hacia dónde se dirige el tráfico. Por defecto, al crear una VPC se crea consigo una tabla principal que distribuye el tráfico dentro de la misma y que será asociada a todas las *Subredes* creadas. Sin embargo, es posible crear tablas personalizadas nuevas.

- *Gateway de Internet*. Recurso que permite la comunicación entre los recursos de la VPC e Internet.

EC2 Las EC2 son las máquinas virtuales de AWS. A continuación se mencionan sus parámetros más importantes [38].

- *Tipo de instancia*. Es el entorno virtual donde se creará dicha VM, es decir, su capacidad de cómputo, memoria y red.
- *Imágenes de máquina de Amazon (*AMI*)*. Son plantillas con una configuración *software* (Sistema operativo, servidor de aplicaciones y aplicaciones) a partir de las cuales se crearán las instancias.
- *Pares de claves*. Requeridas para el inicio de sesión seguro en las instancias.
- *Interfaz de Red*. Tarjeta de red virtual con la configuración de red.

Grupos de Seguridad Un grupo de seguridad es como un *firewall* orientado a instancias que controla el tráfico entrante a uno o un conjunto de recursos. En ellos se especifica una serie de reglas que regulan dicho tráfico en función de los siguientes parámetros [39].

- Si el tráfico es de entrada o salida.
- El tipo de protocolo del tráfico.
- Las IP y puertos fuente y destino.

Diseño de la arquitectura

Para la creación de este nivel se van a diseñar dos módulos, uno para representar el EC2 y otro para la VPC.

Como se puede observar en la Figura 5.1, la VPC estará formada por una *Subred* pública que tendrá asociada una *Tabla de Rutas* con una ruta hacia el *Gateway de Internet*, para permitir la entrada de tráfico hacia una EC2, que se encontrará dentro de la misma.

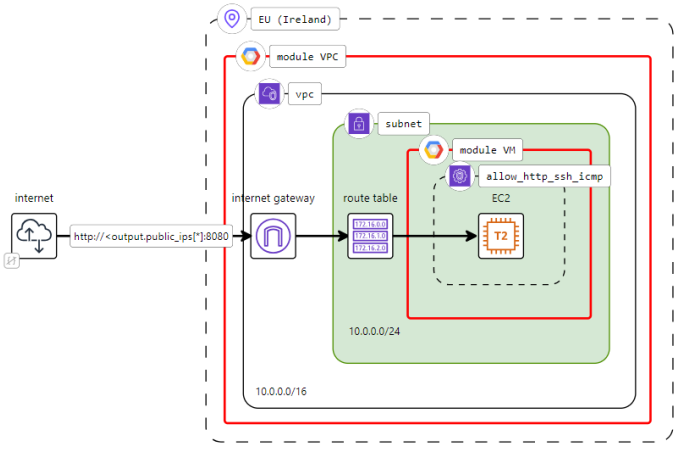

Figura 5.1: Diagrama del primer nivel de la arquitectura en AWS

Implementación de la arquitectura

Como se mencionó, el módulo raíz hace uso de dos módulos.

VPC Módulo que recibe como variables de entrada un bloque de direcciones IP para la VPC y otra lista de sub-bloques cuyos elementos serán utilizados para crear las *Subredes*. De esta forma, se hacen uso de los siguientes `resource`.

- `aws_vpc`. Crea una VPC en el bloque de direcciones asignado en `cidr_block`.
- `aws_subnet`. Crea una *Subred* en la VPC asociada mediante `vpc_id`, y con el sub-bloque asignado en `cidr_block`. Además se ha activado el argumento `map_public_ip_on_launch` para asignar una dirección IP pública a todas las instancias que se creen dentro de esta *Subred*.
- `aws_internet_gateway`. Crea un *Gateway de Internet* en la VPC designada en `vpc_id`.
- `aws_route_table`. Crea una *Tabla de Rutas* en el `aws_vpc` designado en `vpc_id`, y asociado a un `aws_subnet` a través del `resource` `aws_route_table_association`. Para crear las rutas, hace uso del bloque `route`. En este caso una a Internet a través del `aws_internet_gateway`.

Por último, este módulo retorna los identificadores del `aws_vpc` y los `aws_subnet` creadas, para que pueden ser asociadas a otros `resource`.

EC2 Este módulo recibe de entrada los `aws_vpc` y `aws_subnet` donde se crearán estas máquinas; el bloque de direcciones de dicha VPC, para permitir su tráfico con el resto de componentes; un puerto donde desplegar el servidor web; y un `number_instances` para determinar el numero de instancias a lanzar. Así, se hace uso de principalmente los siguientes `resource`:

- `aws_instance`. Crea las instancias de EC2 según muchas de las características comentadas en la Sección 5.1.1, como el `network_interface`, el `instance_type` y el `ami`, en este caso obtenidos mediante los bloques `data` `aws_ec2_instance_types` y `aws_ami`, para obtener un tipo gratuito y un Ubuntu 20.04 respectivamente. También se encuentran argumentos como `user_data`, para asignar una serie de comandos que serán lanzados en cuanto inicie la máquina, en este caso un servidor web; o el `key_name`, que asigna una clave SSH obtenida a través del `resource` `aws_key_pair`.
- `aws_network_interface`. Crea una *Interfaz de Red* con una `aws_subnet` y unos `aws_security_groups` que serán asociados a la EC2 a la que se asigne.
- `aws_security_group`. Crea un *Grupo de Seguridad* con una serie de reglas definidas mediante los bloques `ingress` y `egress`, que especifican las opciones definidas en la

Sección 5.1.1. En este caso, y como ya definimos en el diseño, se permitirá el tráfico interno y el externo mediante ICMP, SSH y HTTP a través del puerto del servidor.

Finalmente retorna la dirección IP de las instancias creadas.

5.1.2 Azure

Recursos necesarios

Azure Virtual Network Es la *Red Virtual* donde se encuentran los recursos de Azure y que permite que se comuniquen de forma segura entre ellos, con Internet y con redes locales.

Es posible reservar una serie de direcciones IP dentro de la *Virtual Network* (VNet) para crear una *Subred*. De forma predeterminada, Azure ya enruta el tráfico entre subredes, redes virtuales conectadas, redes locales e Internet, por lo que todos los recursos a los que se les asigne una dirección IP pública tendrán salida a Internet automáticamente. Sin embargo, es posible crear *Tablas de Rutas* que sustituyan a estas predeterminadas. [40]

También es posible asignar grupos de seguridad a estas *Subredes*, que crean reglas para limitar el tráfico entrante y saliente a cada una de ellas según una serie de parámetros [41]:

- *Prioridad*. Cuanto más bajo, mayor prioridad tendrá la regla.
- *Protocolo*. Puede ser TCP, UDP, ICMP o *Any*.
- *Acción*. Si deniega o permite.
- *Origen* y *Destino*. IP y puertos de origen y destino.

Azure Virtual Machines Son, como su nombre indica, máquinas virtuales en la nube. Hay ciertas características que se tienen que tener en cuenta a la hora de crearlas. [42]

- *Tamaño*. Indica la potencia de la máquina virtual, como la capacidad de procesamiento, la memoria o el ancho de banda de red.
- *Imagen*. *Software* base del que partirá la VM.
- *Autenticación*. Establece el modo de autenticación a la máquina, mediante SSH o contraseña.
- *Disco*. Tanto el tamaño y tipo del *Disco del* SO, como el de otros *Discos de datos*.
- *Interfaz de Red*. Establece la configuración de red.

Diseño de la arquitectura

Para la creación de este nivel se van a diseñar dos módulos, uno para la *Virtual Network* y otro para la *Virtual Machine*.

Como se puede observar en la Figura 5.2, la *Virtual Network* contendrá una *Subnet* pública con un *Grupo de Seguridad* que permita el tráfico HTTP y SSH. Por otro lado, la *Virtual*

Machine será una máquina Linux dentro de esta, con una Interfaz de Red con IP pública.

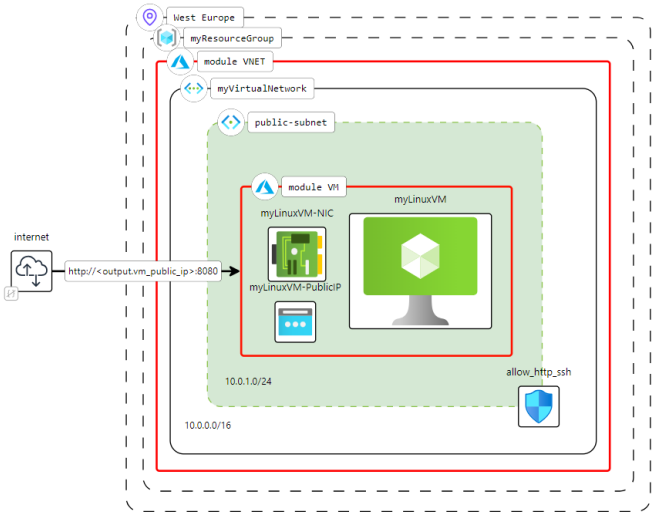

Figura 5.2: Diagrama del primer nivel de la arquitectura en Azure

Implementación de la arquitectura

En Azure todos los recursos pertenecen a un grupo de recursos, que en Terraform se construye con `azurerm_resource_group`. Por lo tanto, este debe ser asignado a todos los `resource` usados en este proyecto, aunque por simplicidad se omitirá su mención así como el de su localización al hablar de los parámetros de entrada de cada módulo en Azure.
De igual forma, y como se mencionó anteriormente, el módulo raíz hace uso de dos módulos nuevos.

Módulo VNET Recibe como parámetros de entrada el bloque de direcciones asignadas a la *VNet* y a sus *Subredes*, así como el puerto donde se desplegará el servidor para permitir su tráfico. Contiene los siguientes recursos.

- `azurerm_virtual_network`. Crea una VNet dado un espacio de direcciones a través del argumento `address_space`.
- `azurerm_subnet`. Crea una *Subred* dentro de la `azurerm_virtual_network` especificada en el argumento `virtual_network_name` y con un rango de direcciones asignado en `address_prefixes`.
- `azurerm_network_security_group`. Crea una serie de reglas de seguridad a través del bloque `security_group`, con todas las características mencionadas en la Sección 5.1.2, y se asocia a una `virtual_network_name` a través de `azurerm_subnet_network_security_group_association`. En este caso se permite todo el tráfico al puerto 22 (SSH) y al del servidor web.

Por último retorna los `azurerm_subnet` creados.

Módulo VM Recibe de entrada el `azurerm_subnet` donde se encontrará, el puerto donde desplegar el servidor y el número de instancias a crear. Contiene tres `resource`:

- `azurerm_linux_virtual_machine`. Corresponde a una máquina virtual Linux que recibe como argumentos muchas de las características mencionadas en la Sección 5.1.2. Como el el `size` o tamaño de la instancia, el `source_image_reference` o imagen, el `os_disk`, con características del disco del SO, y `admin_ssh_key`, para configurar su acceso mediante SSH. Por último se hace uso de `custom_data` para ejecutar un *script* que lance un servidor web justo al iniciar la instancia.
- `azurerm_network_interface`. Interfaz de red que será asociada con el `azurerm_linux_virtual_machine`. Entre sus argumentos se encuentran el bloque `ip_configuration`, donde se especifica principalmente el `azurerm_subnet` y un `azurerm_public_ip`.
- `azurerm_public_ip`. IP pública que será asociada al `azurerm_network_interface`.

Retorna la dirección pública de las VM creadas.

5.1.3 GCP

Recursos necesarios

Virtual Private Cloud (VPC) Una Virtual Private Cloud (VPC) es una versión virtual de una red física en GCP.
Estas son recursos globales, y por lo tanto no están asociadas con ninguna región o zona en particular. Tampoco definen, a diferencia de otros entornos, ningún rango de direcciones. Para ello, es posible crear varias *Subredes* que sí son regionales y que sí tienen su propio rango de direcciones IP. [43]

Para dirigir el tráfico dentro de esta VPC, se hace uso de *Rutas* que en GCP pueden ser de principalmente de dos tipos [44].

- *Rutas predeterminadas*.
  - *Ruta hacia Internet*. Creada automáticamente con cualquier VPC, enruta el tráfico hacia Internet. Se puede borrar o modificar.
  - *Rutas de subred*. Creadas automáticamente con una *Subred*, enrutan el tráfico entre subredes. No se pueden ni borrar ni modificar.
- *Rutas personalizadas*. Permite crear rutas estáticas o dinámicas ligadas a una VPC.

Firewall Son un conjunto de reglas que limitan el tráfico, no solo entrante y saliente de la VPC, sino también el interno entre instancias de la misma. En estas reglas se especifica principalmente [45]:

- *Dirección del tráfico*. Interno o Externo.
- *Protocolo*.

- *Acción a realizar*. Permitir o Denegar.
- *Origen* y *Destino*.

Instancias de *Compute Engine* Son instancias de máquinas virtuales dentro de Google Cloud Platform que vienen determinadas por su *Tipo de máquina*, una *Imagen* con un *software* preconfigurado dentro de un *Disco*, una *Interfaz de Red* con la configuración de Red, y un *Modo de acceso*. [46]

Diseño de la arquitectura

Para este nivel se desarrollarán tres módulos, uno por cada uno de los recursos mencionados anteriormente.

Como se puede apreciar en la Figura 5.3, el primer módulo consistirá en una VPC que contendrá una *Subred* pública, y donde se encontrará a su vez otro módulo representando una instancia de *Compute Engine*. Por último, la VPC tendrá a su vez asociado un módulo con un conjunto de reglas *Firewall* para limitar el tráfico y permitir únicamente el tráfico interno, y el externo a través de SSH, ICMP y HTTP en el puerto que correrá el servidor web.

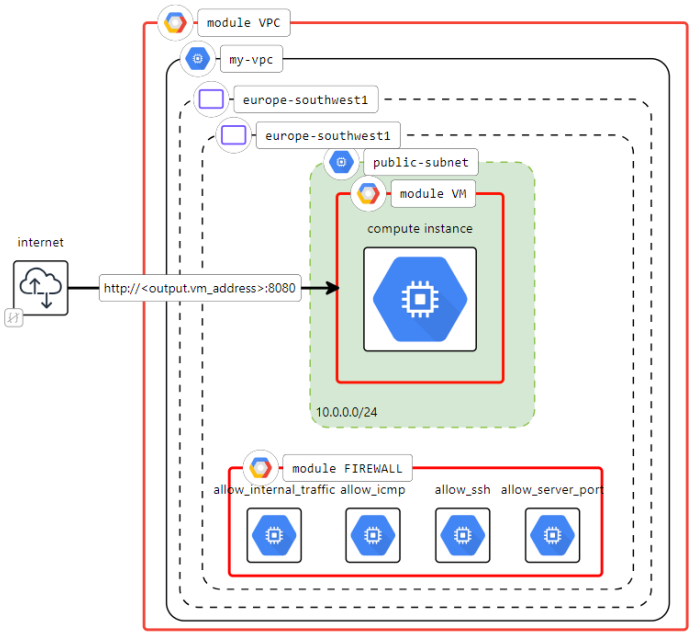

Figura 5.3: Diagrama del primer nivel de la arquitectura en GCP

Implementación de la arquitectura

En esta sección se detallará la implementación de los tres módulos creados diseñados anteriormente.

Previamente, GCP necesita la habilitación de cada servicio o API que se vaya a utilizar en un proyecto. Esto es posible hacerlo directamente desde su interfaz web o a través del recurso `google_project_service` en Terraform. Por lo tanto, y para tener que evitar crear continuamente dicho recurso cada vez que se vaya a hacer uso de un nuevo servicio, se ha creado un módulo aparte llamado `SERVICES` que recibe una lista con los servicios a activar, y que se encargará de crear automáticamente dicho `resource` por cada uno de sus elementos.

`VPC` Este módulo recibe como variables de entrada una lista de bloques que se usarán para crear *Subredes*, y el puerto donde correrá el servidor para permitir su tráfico. Así, hace uso de dos `resource`:

- `google_compute_network`. Crea una VPC global. Debido a que estas crean una serie de *Subredes* automáticamente, se deshabilitará esta opción con `auto_create_subnetworks` en `false`.
- `google_compute_subnetwork`. Crea una serie de *Subredes* dentro del `google_compute_network` determinado en `network`, y en un rango de direcciones asignado mediante `ip_cidr_range`.

Por último retorna el conjunto de `google_compute_subnetwork` creados para que sean asociados a otros `resource` en el VPC.

A su vez, este módulo hace uso de otro módulo que ya se mencionó anteriormente, `FIREWALL`.

`FIREWALL` Este módulo creará una serie de reglas ya mencionadas en la Sección 5.1.3 a través del `resource` `google_compute_firewall`. Este recurso hace uso de bloques `allow` o `deny` para permitir o denegar el tráfico según las características ya descritas.

`VM` Este módulo recibe el `google_compute_subnetwork` donde se encuentra, el puerto donde lanzará un servidor web, y el número de VM a crear. Para ello, se hace uso de un único `resource`, `google_compute_instance`.
Este recurso determina mediante argumentos muchas de las características mencionadas en la Sección 5.1.3, como el `machine_type`; el `boot_disk`, donde se especifica la *Imagen*; o el bloque `network_interface`, que asocia la instancia a un `google_compute_subnetwork`, y le asigna una IP pública al añadirle el bloque `access_config`.
Por ultimo, también hace uso de algunos argumentos como `metadata_startup_script`, para incluir un *script* que lance un servidor web al iniciar, o el bloque `metadata` para añadir alguna otra característica como una clave SSH a través de la cual conectarse a dicha instancia.

5.2 Segundo nivel de la arquitectura

5.2.1 Amazon Web Services (AWS)

Recursos necesarios

Auto Scaling Servicio de AWS que permite crear colecciones de instancias de EC2 y que está formado principalmente por tres elementos [47].

- *Grupos*. Es la colección de instancias que se administran y escalan como conjunto.
- *Plantilla de Configuración*. Plantilla base que contiene las características a partir de las cuales se crearán cada instancia.
- *Opciones de escalado*. Es posible ajustar un máximo, mínimo y número deseado de instancias. Además se pueden llevar a cabo diferentes maneras de escalado.
  - *Mantener un número fijo de instancias*. El servicio comprueba periódicamente que las instancias están en buen estado y las reemplaza en caso contrario. Para ello, existen varios tipos de comprobaciones de estado.
    - *EC2*. En este tipo de comprobaciones, se verifica el estado de las instancias EC2 directamente. Es decir, se evalúa si la instancia está funcionando o no, y si la instancia está disponible para procesar solicitudes.
    - *ELB*. Se evalúa si el balanceador de carga puede enrutar el tráfico correctamente a las instancias.
  - *Escalado manual*. Consiste en cambiar manualmente el mínimo, máximo y capacidad deseada.
  - *Escalado por programación*. El escalado se realiza en función de la fecha y hora
  - *Escalado bajo demanda*. Se establecen ciertas métricas para que el grupo escale en función de sus valores.
  - *Escalado predictivo*. Escalado automático en función de patrones diarios y semanales de flujo de tráfico.

Elastic Load Balancing Es el servicio de balanceo de carga de AWS. Este permite distribuir automáticamente el tráfico recibido entre diferentes recursos asociados [48]. Existen distintos tipos [49].

- *Application Load Balancer*. Distribuye el tráfico en la capa 7 (HTTP).
- *Network Load Balancer*. Distribuye el tráfico en la capa 4 (TCP/UDP)
- *Gateway Load Balancer*. Se utiliza para enrutar el tráfico entre redes virtuales.
- *Classic Load Balancer*. Distribuye tráfico en la capa 4 y 7, y además da soporte al servicio depreciado *EC2-Classic*.

Debido a que nuestro diseño distribuye tráfico a un servidor web, lo mejor será hacer uso de un Application Load Balancer (ALB), que en AWS está formado por los siguientes componentes [50]:

- Un *Listener*, que recibe las peticiones de conexión en un protocolo y puerto asignado. Contiene una serie de reglas que determinan como se distribuye el tráfico entrante a cada *Target Group*.
- Un *Target Group*, que son uno o más servidores que recibirán el tráfico que ha filtrado el *Listener* previamente.
- Un *Health Check*, que comprueba el estado de un *Target Group*.

Diseño de la arquitectura

Lo primero que se puede observar en la Figura 5.4 es que el número de subredes ha aumentado. Esto es debido a que tanto el *Grupo de Auto Scaling* como el *Application Load Balancer* deben aprovisionarse siempre en al menos dos *Subredes* pertenecientes a Zonas de Disponibilidad diferentes. Esto supondrá además una mejor disponibilidad para el caso de que alguna de las regiones del proveedor fallen.
Además, se sustituirá el viejo módulo de `VM` por un nuevo `ASG` que contendrá un *Grupo de Auto Scaling* que irá lanzando instancias a partir de una plantilla prediseñada.
Por último, se añadirá un nuevo módulo `LB` que contará con los elementos descritos en la Sección 5.2.1 y que distribuirá el tráfico entre las instancias del módulo anterior.

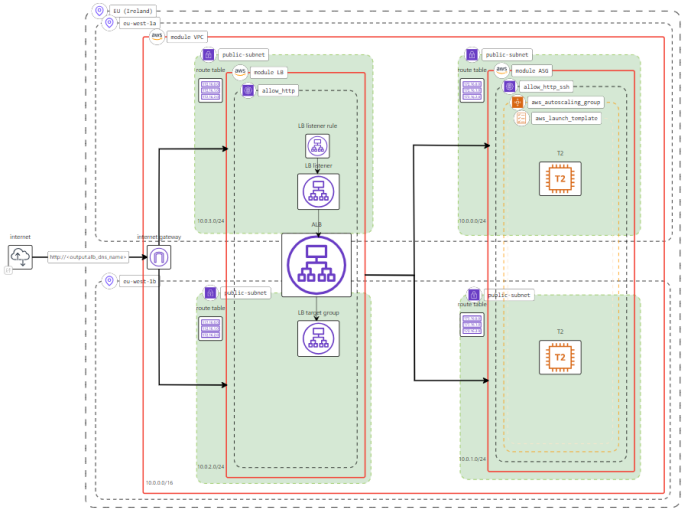

Figura 5.4: Diagrama del segundo nivel de la arquitectura en AWS

Implementación de la arquitectura

En esta sección se describirá la implementación de los módulos mencionados anteriormente.

ASG Módulo que recibe como variables del entrada los `aws_vpc` y `aws_subnet` donde actúa, el puerto donde lanzar un servidor web, y un mínimo y máximo de instancias, con el fin de cambiar estos valores directamente desde el módulo raíz. Además, recibe una lista de los *Target Group* que apuntan a este *Grupo de Auto Scaling*. Está formado por los siguientes `resource`.

- `aws_launch_template`. Crea una *Plantilla de Configuración* a través de muchos de los argumentos ya usados al crear un `aws_instance`, como el *Tipo de Instancia* o su *Imagen*.
- `aws_autoscaling_group`. Crea un *Grupo de Auto Scaling* en unos `aws_subnet` determinados en `vpc_zone_identifier`, y con un `aws_launch_template` asociado con el bloque `launch_template`.
  Para el escalado del grupo, se ha determinado un número mínimo y máximo de instancias, y se ha especificado `health_check_type` como `ELB`, lo que significa que el balanceador de carga será el encargado de evaluar el estado de estas máquinas.
- `aws_security_group`, *Grupo de seguridad* igual al usado anteriormente en el EC2.

LB Este módulo tiene como variables de entrada los `aws_vpc` y `aws_subnet` donde se encuentra, así como el puerto del servidor al que redirigirá el tráfico. Esta formado por los siguientes `resource`.

- `aws_lb`. Crea un *Balanceador de Carga* que, como especifica el argumento `load_balancer_type`, será de aplicación.
- `aws_lb_listener`. Crea un *Listener* que recibirá el tráfico en el `port` en el que corre el servidor y en el `protocol` HTTP. Además, a través del bloque `default_action` se define la acción por defecto a realizar en el caso de que no encontrar destinatario, en este caso que muestre un error `404`.
- `aws_lb_listener_rule`. Crea una regla para dirigir el tráfico de un `aws_lb_listener` especificado en `listener_arn`. Allí se establecen a través de bloques `condition` y `action` que acciones tomar y en función de qué parámetros. En este caso, redirigir todo el tráfico a un `aws_lb_target_group`.
- `aws_lb_target_group`. Crea un *Target Group* que recibirá el tráfico del protocolo y en el puerto especificado en `protocol` y `port` respectivamente. Además, se le asigna un bloque `health_check` que representa un *Health Check* y que comprobará el estado de dicho `aws_lb_target_group` a través de la ruta especificada en `path`, junto a otro serie de parámetros. En este caso simplemente comprobará que el ruta a '`/`' devuelve un código `200`.
- `aws_security_group`. *Grupo de seguridad* que permitirá todo el tráfico dirigido hacia el puerto que corre el servidor.

5.2.2 Azure

Recursos necesarios

Azure Application Gateway Azure ofrece varios servicios de balanceadores de carga que se pueden observar en el Cuadro 5.1.

En este caso, y debido a que vamos a distribuir tráfico web a nivel regional, usaremos el

Servicio	Alcance	Tráfico Recomendado
Azure Front Door	Global	HTTP(S)
Traffic Manager	Global	No HTTP(S)
Application Gateway	Regional	HTTP(S)
Azure Load Balancer	Regional	No HTTP(S)

Cuadro 5.1: Servicios de balanceo de carga ofrecidos en Azure [51]

Application Gateway. Este, toma decisiones sobre como distribuir el tráfico en función de distintos atributos HTTP, como la URL o las *cookies*, y consta de los siguientes elementos [52].

- *Frontend IP*. Dirección que recibe las peticiones HTTP.
- *Listener*. Comprueba las solicitudes de conexión entrantes mediante el puerto, protocolo, *host* y dirección IP.
- *Grupo Backend*. Grupo de instancias que recibirán el tráfico.
- *Configuración* HTTP. Configuración usada por el *Application Gateway* para enrutar el tráfico a cada *Grupo Backend*.
- *Reglas de enrutamiento*. Reglas que enlazan un *Listener* con un *Grupo Backend* y su *Configuración* HTTP.
- *Sondeo de estado*. Supervisa el estado del *Grupo Backend*.

Azure Virtual Machine Scale Set Servicio que permite crear y gestionar un conjunto de *Máquinas Virtuales*, lo que supone grandes ventajas.

- Mayor facilidad a la hora de crear y administrar varias VM, pues todas serán creadas a partir de una misma configuración.
- Aumenta la disponibilidad, pues en caso de fallo en una VM, otra puede sustituirla temporalmente.
- Admite medidas de escalado automático, tanto manual, como bajo demanda, por programación y predictivo.

Diseño de la arquitectura

Como se observa en la Figura 5.5, este nivel consta de dos *Subredes*, cada una alojando a un módulo nuevo.
Por un lado, el viejo VM será sustituido por un nuevo `SS`, que representará el *Azure Virtual Machine Scale Set*. Por otro lado, el módulo `AG`, que representará un *Application Gateway*, y que distribuirá el tráfico entre las distintas instancias del módulo anterior.

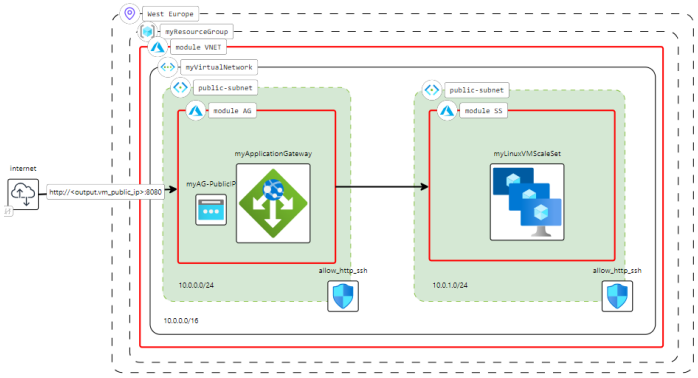

Figura 5.5: Diagrama del segundo nivel de la arquitectura en Azure

Implementación de la arquitectura

En esta sección se describirá la implementación de los módulos presentados anteriormente.

AG Este módulo recibe de entrada un `azurerm_subnet` donde se desplegará, así como un puerto para el *Grupo Backend* donde distribuirá el tráfico.
Hace uso principalmente del `resource` de `azurerm_application_gateway`, que crea muchos de los elementos mencionados en la Sección 5.2.2 a través de sus argumentos.

- Los bloques `gateway_ip_configuration`, `frontend_port` y `frontend_ip_configuration` configuran respectivamente su `azurerm_subnet` y puerto e IP del *Frontend*.
- `http_listener`. Configura el *Listener* para recibir tráfico HTTP a través del *Frontend* anterior.
- `backend_address_pool` Crea un *Grupo Backend* que será reutilizado en el módulo `SS` para su asociación con un *Virtual Machine Scale Set*.
- `backend_http_settings`. Determina la *Configuración* HTTP para seleccionar el puerto al que enrutar el tráfico del `backend_address_pool`.
- `request_routing_rule`. Enlaza todos los elementos anteriores.
- `probe`. Crea un *Sondeo de estado* que verificará el estado del *backend* especificado mediante `host`, que en el supuesto de que estar configurado para un único sitio, como

en este caso, tomará el valor de `127.0.0.1`; e intentará obtener una respuesta del `path` especificado.

Como valores de retorno se encuentran el `backend_address_pool`, para su utilización en el siguiente módulo, y la dirección IP pública de este *Application Gateway*.

SS Este módulo recibe como parámetros de entrada el `azurerm_subnet` donde se desplegará, el número de instancias a crear, el puerto donde se lanzará el servidor web y el `backend_address_pool` del módulo anterior al que se asociará.
Utiliza únicamente un solo `resource`, `azurerm_linux_virtual_machine_scale_set`, que hace uso de sus argumentos para determinar las características con las que se instanciarán sus máquinas. Muchas de estas, son las mismas que ya encontrábamos al crear un `azure_linux_virtul_machine`, como el `sku` o tamaño de la instnacia, el `source_image_reference` o imágen a usar, un `network_interface` donde se asociarán el `backend_address_pool`, entre otros.También incluye un argumento `instances` para establecer el número de instancias a desplegar.

5.2.3 GCP

Recursos necesarios

Cloud Load Balancing Servicio que distribuye el tráfico entre distintas instancias. Existen muchas opciones en función de determinadas características como si el tráfico es *Interno* o *Externo*, si el alcance es *Regional* o *Global* o el tipo de tráfico que distribuirán [53].

En este caso, al desplegar un servidor web, bastará con un *Balanceador de Carga* HTTP*, regional externo*, que está formado por varios elementos [54].

- *Subred de solo proxy*. Proporciona una serie de direcciones IP que serán usados para ejecutar *proxies*.
- *Reglas de forwarding externo*. Enrutan el tráfico en función de la dirección IP, puerto y protocolo, hacia un *Proxy de Destino* HTTP.
- *Proxy de Destino* HTTP. Consulta un *Mapa de* URL para dirigir el tráfico a los *Backends*.
- *Mapas de* URL. Usado por el *Proxy de Destino* HTTP para realizar una determinado enrutamiento dependiendo de atributos HTTP como la ruta de la solicitud, las *cookies* o los encabezados.
- *Backends*. Instancias a las que se distribuye el tráfico.
- *Verificaciones de estado*. Supervisa el estado del *backend* periódicamente.

Grupos de Instancias Un grupo de instancias es un conjunto de máquinas virtuales y que se pueden administrar como una sola entidad. Existen dos tipos. [55]

- *Grupos de Instancias No Administrados*. Servicio base para la creación de agrupaciones de instancias heterogéneas.
- Grupos de Instancias Administradas (MIG). Permiten añadir escalabilidad y reparación automática a las instancias, entre otras opciones.

Por lo tanto, para este proyecto se hará uso de un MIG, que presentan principalmente las siguientes características [55].

- *Reparación automática*. Ya sea a través de verificaciones sobre las propias instancias o a través de un *Verificador de Aplicación*, el MIG se asegurará siempre que las VM estén disponibles.
- Posibilidad de crear un MIG zonal o regional, en caso de querer aumentar la disponibilidad.
- *Escalado automático*. Es posible escalar el MIG manualmente, en función de la demanda, por programación, o predictivamente.
- *Actualización automática*. Cualquier actualización se replicará automáticamente entre todas las instancias, pudiendo controlar en todo momento la velocidad y el alcance para minimizar el tiempo de interrupción.

Diseño de la arquitectura

Para este nivel se crearán dos nuevos módulos, uno por servicio nuevo mencionado anteriormente.

Como se observa en la Figura 5.6, este nivel sustituye el viejo módulo `VM` por un `MIG` que representará los *Grupos de Instancias Administradas*, y que constará con una plantilla a partir de las cuales crear las instancias, un escalador automático y un sondeo de estado. Por otro lado, se creará un módulo `LB` con el *Balanceador de Carga* que distribuirá el tráfico a dichas instancias.

Implementación de la arquitectura

En primer lugar, el módulo `VPC` ahora también permite crear, a través del parámetro de entrada `proxy_subnets`, un `google_compute_subnetwork` con su argumento `purpose` de tal manera que esté dedicada exclusivamente para *proxies* regionales, y por lo tanto, será donde deba localizarse el *Balanceador de Carga*, así como se especificó en la Sección 5.2.3.
Por otro lado, se describirá la implementación de los dos nuevos módulos.

`MIG` Este módulo recibe como variable de entrada el `google_compute_subnetwork` donde actúa, y el puerto donde lanzará el servidor web. Por otro lado, hace uso de varios `resource`.

- `google_compute_instance_template`. Crea una plantilla que será usada para la creación del resto de instancias. Para ello, en sus argumentos se definen muchas de las ca-

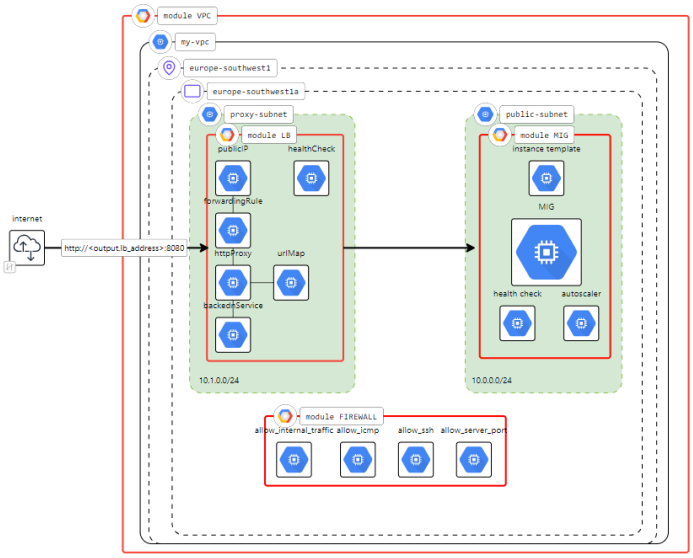

Figura 5.6: Diagrama del segundo nivel de la arquitectura en GCP

racterísticas que ya se usaban al crear un `google_compute_instance`. Además, Google recomienda añadir ciertos permisos sobre `cloud-platform` para que todo funcione correctamente. Para ello se ha definido el bloque `service_account` que da los permisos necesarios al *email* del servicio.

- `google_compute_instance_group_manager`. Crea un MIG que lanzará instancias a partir de un `google_compute_instance_template` designado en el bloque `version`, y que tendrá asociado un `google_compute_health_check` en el bloque `auto_healing_policies`. Además, crea un puerto con nombre en el argumento `port_name` que será utilizado por el balanceador de carga para distribuir el tráfico a dicho puerto.
- `google_compute_autoscaler`. Crea un *autoescalador* asociado con el `google_compute_instance_group_manager` mediante el argumento `target`, con una política de escalado determinada mediante el bloque `autoscaling_policy`. En este caso se establece un mínimo de 2 instancias que puede escalar a un máximo de 4 en caso de que el uso de CPU supere el 60 %.
- `google_compute_health_check`. Crea un *Sondeo de Estado* para verificar el estado de las máquina del grupo, mediante la comprobación de una ruta especificado en `http_health_check`, y con una frecuencia determinada mediante argumentos. En este caso se comprueba cada 5 segundos que la ruta '`/`' devuelva un código `200`.

Como variables de retorno se encuentra el grupo de instancias creado con el fin de ser asociado a un *Backend* en el *Balanceador de Carga*.

LB Este módulo recibe como argumentos el `google_compute_instance` que servirá como *Backend*, un puerto de dicho *Backend* donde comprobar su estado y el `google_compute_`-

`subnetwork` donde se desplegará. Además hace uso de los siguientes recursos.

- `google_compute_forwarding_rule`. Crea un *Balanceador de Carga* según el valor de `load_balancing_scheme`, en este caso externo. Además, determina la `network` en la que trabaja, y establece el IP y rango de puertos donde recibe el tráfico con `ip_address` y `port_range`.
- `google_compute_region_target_http_proxy`. Representa el *proxy* HTTP con un `google_compute_region_url_map` asociado con `url_map`.
- `google_compute_region_url_map`. Crea un *Mapa de* URL que determinará hacia donde distribuye el tráfico el `google_compute_region_target_http_proxy`. En este caso simplemente se establece una ruta por defecto hacia el único `google_compute_region_backend_service` a través del argumento `default_service`.
- `google_compute_region_backend_service`. Crea un *Backend* que estará asociado con el `google_compute_instance_group_manager` a través del bloque `backend`. Además es necesario especificar el `port_name` que se indicó en el `google_compute_instance_group_manager` para comunicarse con este. Por último asocia el *Backend* con un `google_compute_region_health_check` a través del argumento `health_checks`.
- `health_checks`. Igual que el ya creado en el módulo anterior, pero en este caso servirá para comprobar las instancias que no respondan y evitar que el tráfico se distribuya a estas.

Devuelve como valor de retorno la dirección IP del *Balanceador de Carga*.

5.3 Tercer nivel de la arquitectura

5.3.1 Amazon Web Services (AWS)

Recursos necesarios

Relational Database Service (*RDS*) RDS es un servicio de base de datos relacionales en la nube de AWS.

El componente más básico de este servicio es la instancia de base de datos. Esta es a *grosso modo* la máquina con un *hardware* predefinido y un entorno preparado para contener una o más bases de datos creadas por el usuario. Esta tiene una serie de características que hay que tener en cuenta en su creación. [56]

- *Motor de Base de Datos*. El *software* que se ejecuta en las instancias de base de datos, como MariaDB, MySQL, Microsoft SQL Server, Oracle o PostgreSQL.
- *Clase de instancia*. Determina características del *hardware* como el número de procesadores o el tamaño de RAM.
- *Almacenamiento*. Son los volúmenes de almacenamiento de las instancias. Existen principalmente dos tipos [57].

- SSD de uso general (`gp2` y `gp3`). Para instancias de tamaño medio.
- SSD de IOPS (`io1`). Para uso intensivo de operaciones de E/S.

RDS permite además la creación de réplicas que proporciona compatibilidad con la conmutación por error [58].

5.3.2 Diseño de la arquitectura

Como se puede observar en la Figura 5.7, se creará nuevo módulo para el RDS, que estará alojado dentro de dos *Subredes* privadas que únicamente será accesibles desde el módulo `MIG`.
Esto es debido, en primer lugar, porque todo RDS debe estar siempre en al menos dos *Subredes* de distinta Zona de Disponibilidad [59] y, en segundo lugar, para conseguir la replicación que se mencionó en el apartado anterior.

Por último, también se desplegará un microservicio muy sencillo desarrollado con Spring Boot en Java y que hará uso de esta RDS.

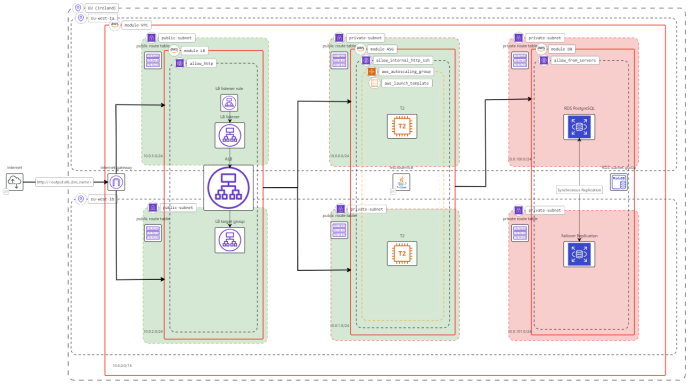

Figura 5.7: Diagrama del tercer nivel de la arquitectura en AWS

5.3.3 Implementación de la arquitectura

En primer lugar, el módulo `VPC` ahora también crea `aws_subnet` privadas mediante la variable de entrada `private_subnets`. Tiene dos diferencias principales con respecto a las anteriores públicas.

- El argumento `map_public_ip_on_launch` que antes estaba asignado en `true` por defecto y que asignaba una IP pública a sus instancias automáticamente, ahora está en `false`.
- No tiene ninguna `aws_route_table` hacia Internet, por lo que únicamente consta de una ruta por defecto hacia las instancias de su misma VPC.

En segundo lugar, se ha creado un microservicio muy sencillo que será desplegado en el módulo `ASG` mediante uno de los paradigmas de trabajo que se mencionó en la Sección 3.2.5, el uso de Packer para la creación de un entorno con el *software* preparado. De este mo-

do, ahora el módulo `ASG` hace uso de una `AMI` de Ubuntu 20.04 con OpenJDK-17 y con el microservicio listo para ser ejecutado directamente.

Por último se ha creado el nuevo módulo mencionado en la sección anterior.

DB Este módulo recibe como variables de entrada el `aws_vpc` y las `aws_subnet` donde actúa, los bloques de direcciones de las `aws_subnet` donde se lanzan los servidores y que podrán comunicarse con esta RDS, y el puerto donde lanzar la base de datos junto con su usuario y contraseña. Además hace uso de los siguientes `resource`.

- `aws_db_subnet_group`. Crea un grupo de las `aws_subnet` en las que se alojará un `aws_db_instance`. Estas deberán ser al menos dos y de diferente Zona de Disponibilidad.
- `aws_db_instance`. Crea una instancia de base de datos con muchas de las características mencionadas en la Sección 5.3.1.
  - `instance_class`. Determina el tipo de instancia, que en este caso será un `db.t3.micro`.
  - `engine`. Detemina el motor de la base de datos, que en este caso será PostgreSQL.
  - `storage_type` y `allocated_storage`. Determina el tipo y la cantidad de almacenamiento en GB.
  - `multi_az`. Activa la replicación en mútliples Zonas de Disponibilidad.

Retorna la dirección de dicho `aws_db_instance` para ser utilizada por el módulo `ASG`.

5.3.4 Azure

Recursos necesarios

Azure Database for PostgreSQL *Azure Database for PostgreSQL* es el servicio de base de datos en la nube basado en PostgreSQL ofrecido por Microsoft. Este servicio ofrece actualmente dos modelos: servidor único, que será retirado en los próximos años; y servidor flexible, que será el usado en este proyecto. [60]
En este modelo, el motor de la base de datos se ejecuta dentro de un contenedor en una VM de Linux, mientras que los datos se almacenan por separado y con redundancia local, garantizando la durabilidad de los mismos. Además, y de la misma forma que ocurría con RDS en AWS, permite crear réplicas en distintas zonas de disponibilidad para aumentar la tolerancia a fallos. [61]

Diseño de la arquitectura

Como se observa en la Figura 5.8, se va a crear un nuevo módulo `DB` que corresponderá con el servicio mencionado anteriormente y que será desplegado en una *Subred* privada alcanzable únicamente desde el módulo `MIG`. Este, además, desplegará en sus instancias un microservicio sencillo que se comunicará con la base de datos a crear.

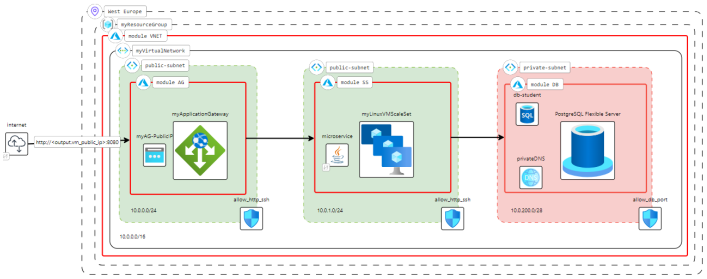

Figura 5.8: Diagrama del tercer nivel de la arquitectura en Azure

Implementación de la arquitectura

En primer lugar, el módulo `VNET` ahora crea un `azurerm_subnet` privado de uso exclusivo para la base de datos a través del argumento `db_subnet`. Para ello, dicho `resource` requiere de unos permisos especificados en su argumento `delegation`. Este `azurerm_subnet` tendrá asociado también un `azurerm_network_security_group` que solo permitará el tráfico al puerto en el que se lanza la base de datos y únicamente a través de las instancias del módulo `MIG`.

En segundo lugar, y al igual que ya se hizo en AWS, en el módulo `MIG` se ha sustituido la imagen anterior por una nueva generada mediante Packer. Esta nueva imagen contiene un entorno preparado para desplegar el código Java del microservicio creado.

Por último se ha implementado el módulo `DB`

DB Este módulo recibe de entrada el `azurerm_virtual_network` y `azurerm_subnet` donde se aloja, así como un nombre y contraseña de usuario de la base de datos. Hace uso de los siguientes `resource`.

- `azurerm_private_dns_zone_virtual_network_link`. Asigna un nombre de dominio privado que estará asociado con un `azurerm_postgresql_flexible_server` a través de `azurerm_private_dns_zone_virtual_network_link`, y que será utilizado por el resto de servicios de la *VNet* para acceder a este.
- `azurerm_postgresql_flexible_server`. Crea un *Servidor Flexible* de *Azure PostgreSQL* en una `azurerm_subnet` que debe de ser de uso específico como se comentó anteriormente, y que se especifica mediante el argumento `delegated_subnet_id`. Además se determina muchas sus características como `sku_name`, para el tipo de instancias; `version`, versión del motor a usar; `storage_mb`, para indicar los MB de almacenamiento; o `administrator_login` y `administrator_password`.
- `azurerm_postgresql_flexible_server_database`. Crea una base de datos en un `azurerm_postgresql_flexible_server` especificado en `server_id` y con nombre en `name`.

Retorna la dirección del `azurerm_private_dns_zone_virtual_network_link` para ser utilizada por el módulo `MIG`.

5.3.5 GCP

Recursos necesarios

Cloud SQL Cloud SQL es el servicio de bases de datos relaciones de GCP que permite administrar bases de datos de MySQL, PostgreSQL y SQL Server.
Este servicio consta de una VM que contiene uno de los motores de bases de datos ya mencionados, además de ciertos servicios de asistencia, como monitoreo o *logging*. Por otro lado, las bases de datos son almacenadas en un dispositivo de red escalable y duradero que se conecta a dicha máquina virtual. [62]
Por último, y al igual que ocurria en el resto de entornos, es posible activar una Alta Disponibilidad para crear réplicas que mejoren el *failover*. [62]

Diseño de la arquitectura

Como se puede observar en la Figura 5.9 , este diseño cambia drásticamente con respecto al mismo nivel en entornos anteriores.
En GCP, ciertos servicios o API ofrecidos por Google y terceros, entre los que se encuentra Cloud SQL, son creados dentro de una VPC propia llamada *Producer* VPC, en contraparte con la creada por los consumidores y llamada *Consumer* VPC. Por lo tanto, si se requiere que la instancia de base de datos tenga conexión con las máquinas *Consumer* VPC necesitaremos configurar un *network peering* o intercambio de tráfico entre ambas VPC a través de un *Private Service Connect* [63]. Para ello, será necesario reservar en la *Consumer* VPC un bloque de direcciones que no podrán ser asignados a ninguna instancia de la red, sino que serán utilizados para acceder a los servicios de la VPC de los productores de servicios.

Además, se desplegará un microservicio en el MIG que haga uso de las base de datos de la misma forma que ya se hizo en entornos anteriores.

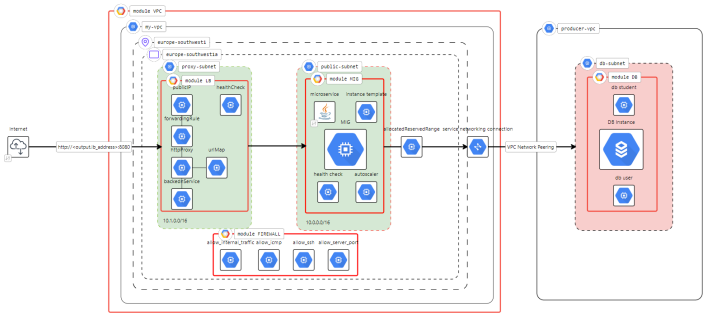

Figura 5.9: Diagrama del tercer nivel de la arquitectura en GCP

Implementación de la arquitectura

Para modelar del comportamiento mencionado en la sección anterior, se han añadido dos nuevos `resource` al módulo `VPC`.

- `google_compute_global_address`. Crea el rango de direcciones reservado para la VPC del servicio de la base de datos. Para ello se especifica mediante su argumento `purpose` que su uso será el intercambio de tráfico, y en `address_type` que serán direcciones internas.
- `google_service_networking_connection`. Crea el *Private Service Connect* para acceder a los servicios del *Producer* VPC a través de un `google_compute_global_address` especificado en `reserved_peering_ranges`. En caso de que dichos servicios sean de Google, en `service` habrá que especificar `servicenetworking.googleapis.com`

Por otro lado, el `google_compute_instance_template` del módulo `MIG` ahora utiliza la nueva imagen creada en Packer, y se ha creado un nuevo módulo `DB`

DB Este módulo recibe como variables de entrada el `google_compute_network` del *Consumer* VPC, desde donde se accederá, y un usuario y contraseña de la base de datos. Además hace uso de los siguientes `resource`.

- `google_sql_database_instance`. Crea una instancia de base de datos con el motor y versión especificado en `database_version`, en este caso la versión 13 de PostgreSQL. Además, en el bloque `settings` es posible configurar ciertas características de su *hardware*, como el tipo de máquina, el almacenamiento y la configuración IP. En este último apartado es importante configurar el `private_network` con la `google_compute_network` desde desde se accederá a la instancia de base de datos.
- `google_sql_database`. Crea una base de datos en el `google_sql_database_instance` especificada en `instance` y de nombre en `name`.
- `google_sql_user`. Crea un usuario en el `google_sql_database_instance` especificada en `instance`, y con nombre y contraseña especificados en `name` y `password`.

Devuelve como variables de retorno la dirección del `google_sql_database_instance` para ser utilizada por las instancias del módulo `MIG`.

5.4 Cuarto nivel de la arquitectura

5.4.1 Amazon Web Services (AWS)

Recursos necesarios

No serán necesarios recursos nuevos, pues se modelará el nodo *jumpbox* a través de una EC2 descrita en la Sección 5.1.1.

Diseño de la arquitectura

Como se puede observar en la Figura 5.10, se añadirá un nuevo módulo `JUMPBOX` que consistirá en una EC2 pública que permita el acceso a las instancias del grupo de autoescalado

a través de ella mediante SSH con el comando `ssh -J ubuntu@<jumpbox_ip>ubuntu@<vm_private_ip>`. Por otro lado, el módulo `ASG` ahora se encuentra dentro de una *Subred* privada, que únicamente permitirá conexiones SSH desde dicho nodo *Jumpbox*, y HTTP en el puerto del servidor a través del *Balanceador de Carga*.

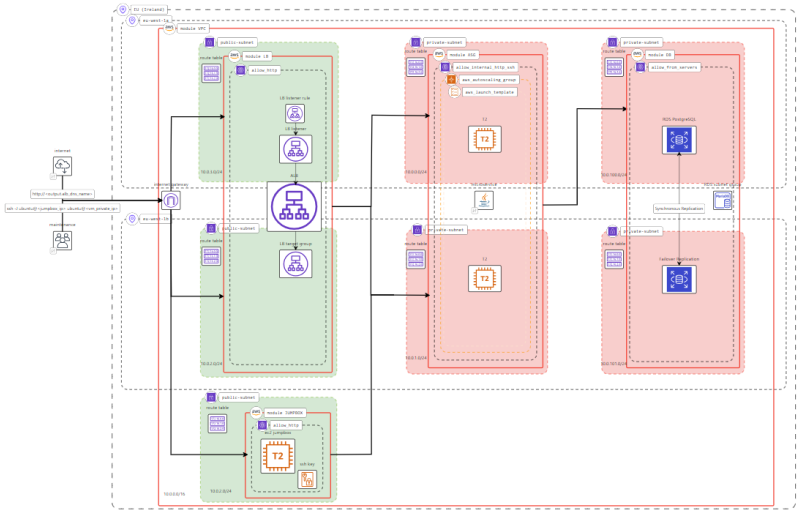

Figura 5.10: Diagrama del cuarto nivel de la arquitectura en AWS

Implementación de la arquitectura

Ahora el módulo `ASG` recibe en la variable de entrada `subnets` una serie de `aws_subnet` privadas.

Por otro lado, el nuevo módulo descrito previamente es muy similar al creado en la Sección 5.1.1. Recibe como argumentos el `aws_vpc` y `aws_subnet` donde se aloja y crea un `aws_instance` con un `aws_key_pair` y dentro de un `aws_security_group` que permite cualquier tráfico SSH entrante.
Por último retorna la dirección IP de dicha máquina.

5.4.2 Azure

Recursos necesarios

Azure Bastion Plataforma como servicio que permite crear un nodo *jumpbox* a través del cual conectarse a las VM mediante SSH o RDP, sin tener que gestionar ni administrar una máquina aparte. Este servicio admite también cierta configuración de escalado, para permitir más conexiones simultáneas. [64]

Diseño de la arquitectura

Como se puede observar en la Figura 5.11, el recurso previamente mencionada formará parte de un nuevo módulo llamado `BASTION` y que se encontrará dentro de una *Subred* llamada

`AzureBastionSubnet`. Esta *Subred* tiene unas ciertas exigencias que Micrososft expone en su documentación, como un tamaño de al menos `/26` y un grupo de seguridad con ciertas reglas específicas asociadas [64]. Además, ahora el módulo `SS` se encuentra dentro de una *Subred* privada que únicamente permitirá el tráfico SSH desde la *Subred* del nodo *Bastion*, y HTTP desde el *Application Gateway*.

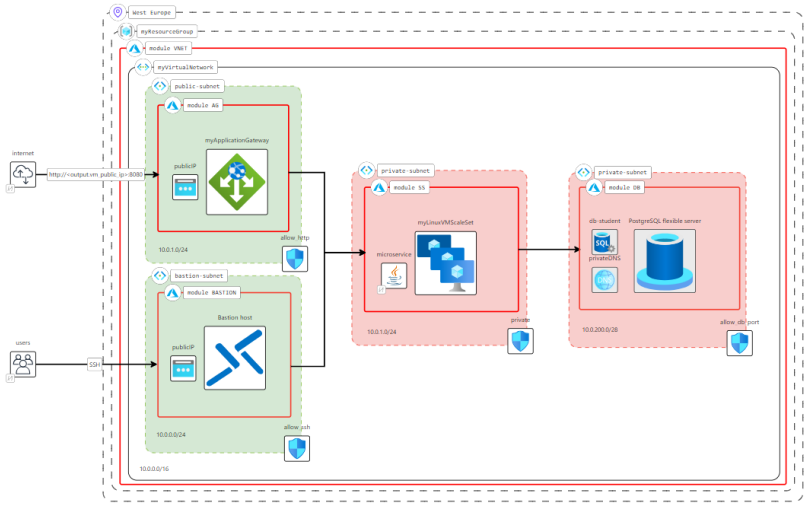

Figura 5.11: Diagrama del cuarto nivel de la arquitectura en Azure

Implementación de la arquitectura

Tal y como se explicó anteriormente, el módulo `VNET` ahora también permite crear un `azurerm_subnet` a través de la variable de entrada `bastion_subnet`. Este `resource` tendrá asociado un `azurerm_network_security_group` con las reglas que Microsoft exige en su documentación.
Por otro lado, el módulo `SS` ahora recibe como entrada en la variable *ss_subnet* un `azurerm_subnet` privado.

Finalmente, se ha implementado el módulo `BASTION`.

BASTION Este módulo recibe como entrada el `azurerm_subnet` donde se alojará un `azurerm_bastion_host`. En este `resource` se pueden configurar ciertos parámetros como el `sku` o la configuración de red con `ip_configuration`, donde se especifica el `azurerm_subnet` donde se encuentra y donde se le asocia un `azurerm_public_ip`.
Por último, devuelve la dirección de dicho `azurerm_bastion_host`.

5.4.3 Google Cloud Platform (GCP)

Recursos necesarios

Al igual que ocurria en AWS, no se requerirán de recursos nuevos, pues el nodo *jumpbox* se implementará desde una *Instancia de Compute Engine* descrita en la Sección 5.1.3.

Diseño de la arquitectura

Como se puede observar en la Figura 5.12, se incluirá un nuevo módulo `JUMPBOX` que contendrá una *Instancia de Compute Engine* a través de la cual conectarse a las instancias del módulo `MIG`, que ahora están en una *Subred* privada únicamente accesible mediante HTTP desde el *Balanceador de Carga*, y mediante SSH a través del nodo *jumpbox*.

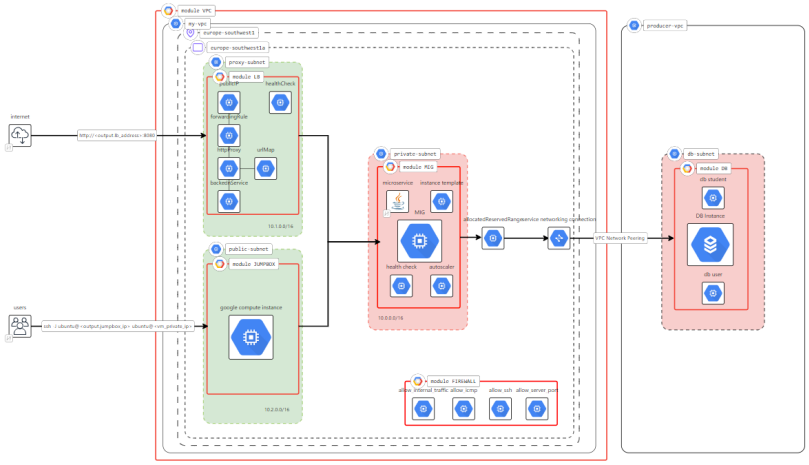

Figura 5.12: Diagrama del cuarto nivel de la arquitectura en GCP

Implementación de la arquitectura

Para eliminar la conectividad directa desde Internet al módulo `MIG`, se ha eliminado el bloque `access_config` en el `google_compute_instance_template`, de modo que a este no se le asigne ninguna dirección IP pública.
Por otro lado, se ha implementado el módulo mencionado anteriormente.

JUMPBOX Este módulo, muy similar al `VM` de la Sección 5.1.3 , recibe el `google_compute_network` y `google_compute_subnetwork` donde se aloja, para crear un `google_compute_instance` con un bloque `metadata` donde se incluye una clave `SSH` siguiendo la forma: `nombre_usuario:ssh_key.pub`.
Por último se retorna la dirección IP pública de la instancia creada.

5.5 Quinto nivel de la arquitectura

5.5.1 Amazon Web Services (AWS)

Recursos necesarios

Elastic Container Service (ECS) ECS es un servicio de orquestación de contenedores ofrecido por Amazon para la creación, gestión y escalado de aplicaciones en contenedores. Este servicio permite integrarse con otras herramientas de AWS como Elastic Container Registry (ECR), o con herramientas de terceros como Docker Registry, para facilitar a los desarrolladores la creación de sus aplicaciones al centrarse únicamente en su integración y no en el entorno. Tiene dos modelos de lanzamiento. [65]

- EC2. Implementado sobre instancias de EC2, más adecuado para cargas de trabajo que requieren de un uso constante de memoria y CPU. Se debe administrar su infraestructura.
- AWS *Fargate*. Opción sin servidor, más adecuado para cargas de trabajo pequeñas, con ráfagas ocasionales. No es necesario administrar su infraestructura.

En este caso se hará uso de la última opción, que está formada por los siguientes componentes [66].

- *Clústeres*. Es una agrupación lógica de *Tareas* o *Servicios*. Por lo tanto, las aplicaciones que corren en un mismo clúster comparten la infraestructura subyacente.
- *Contenedores* e *Imágenes*. Las *Imágenes* son plantillas creadas a partir de archivos *Dockerfile* que se almacenan en registros desde donde se pueden descargar para construir contenedores, que son la *Imagen* en funcionamiento.
- *Definición de tareas*. Archivo JSON que describe uno o varios contenedores que forman una aplicación. En él se especifican parámetros para el sistema operativo como variables de entorno, qué contenedores usar, que puertos abrir, etc.
- *Tareas*. Es la instancia creada a partir de una *Definición de Tarea* dentro de un *Clúster*. Se pueden ejecutar independientemente o como parte de un *Servicio*.
- *Servicio*. Establece un número de *Tareas* deseado en un *Clúster* y se encargará de mantenerlo de dicha manera.
- *Agente de contenedores*. Agente que se ejecuta dentro de cada Clúster y que envía información sobre le uso de recursos y las *Tareas* ejecutándose en cada uno de ellos.

Elastic Container Registry (ECR) Es el servicio de registro de *Imágenes* de Amazon. Este permite crear repositorios tanto públicos como privados donde almacenar *imágenes* de distinto tipo como Docker. Cada cuenta tiene un *Registro*, cada *Registro* tiene varios *Repositorios* y cada *Repositorio* tiene varias *Imágenes* que puede tener varios *tags*, aunque estos no pueden repetirse en un mismo *Repositorio*.

Diseño de la arquitectura

Como se puede observar en la Figura 5.13, se van a incluir dos nuevos módulos, uno por recurso anterior mencionado.
En primer lugar se ha desarrollado un nuevo microservicio de autenticación de modo que junto al anterior, requieran de comunicación mutua síncrona. Estos dos códigos serán subidos como *Imágenes* Docker a un nuevo módulo `ECR` que corresponderá al servicio de registro de contenedores de AWS. En segundo lugar, el anterior módulo `ASG` será sustituido por módulo `ECS`, que contendrá un par de *Servicios* de *AWS Fargate*, uno por microservicio, y que obtendrán sus *Imágenes* a través del módulo `ECR`.
Por último, y con el fin de que los *Servicios* de una red privada puedan obtener dichas *Imágenes* a través de un repositorio de Internet, se va a incluir un *NAT Gateway* que permita a las *Subredes* privadas obtener datos desde fuera de su VPC, pero manteniendo la imposibilidad de ser accesibles directamente desde el exterior.

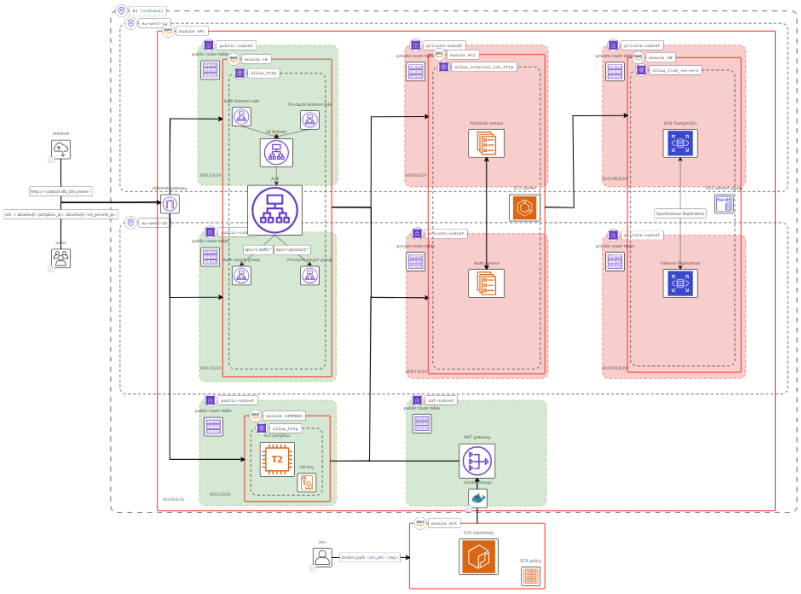

Figura 5.13: Diagrama del quinto nivel de la arquitectura en AWS

Implementación de la arquitectura

En primer lugar, el módulo `LB` ahora contiene dos `aws_lb_listener_rule` que distribuirán el tráfico en función de la ruta a dos `aws_lb_target_group`, uno por cada microservicio.
Por otro lado, el módulo `VPC` ahora crea un `aws_nat_gateway` en un `aws_subnet` público que será usado por los `aws_subnet` privados en su `aws_route_table` para permitirles obtener las *Imágenes* del repositorio desde Internet.
Además se han implementado los dos módulos ya mencionados.

ECS Este módulo recibe de entrada el `aws_vpc` y `aws_subnet` donde actúa, los `aws_lb_target_group` del módulo `LB` a los que pertenecerá cada uno de sus *Servicios*, el `aws_security_group` de donde únicamente podrán recibir tráfico, el URL del `aws_ecr_repository` de donde obtendrán las imágenes, y un conjunto de datos necesarios por los microservicios como la dirección de la base de datos, su usuario y contraseña o puerto donde correr. Además hace uso de los siguientes recursos.

- `aws_ecs_cluster`. Crea un *Clúster*.
- `aws_iam_role`. Los *AWS Fargate* requieren unos permisos de ejecución [67] que serán asignados mediante este `resource` y el bloque `data aws_iam_policy_document`.
- `aws_ecs_task_definition`. Crea una *Definición de tarea* para cada uno de los microservicios. En sus argumentos se asocian los permisos del `resource` anterior en `execution_role_arn`; se elige el modelo de lanzamiento en `requires_compatibilities`, en este caso `FARGATE`; y se especifica una definición en un archivo JSON con las características de la tarea en `container_definitions`.
- `aws_ecs_service`. Crea un *Servicio* por microservicio en el `aws_ecs_cluster` y con `aws_ecs_task_definition` definidos en `cluster` y `task_definition` respectivamente. Además se especifica de nuevo el modelo de lanzamiento en `launch_type` y el número de *Tareas* a crear en `desired_count`.
  Por último se configura la red en `network_configuration` y se asocia a un `aws_lb` a través de `load_balancer`.
- `aws_cloudwatch_log_group`. Se crea un grupo de registros para monitorear los `aws_ecs_service`.
- `aws_security_group`. Crea el *Grupo de Seguridad* para permitir el tráfico entrante únicamente desde el `aws_lb`.

ECR Crea un repositorio con `aws_ecr_repository` y le asigna una serie de políticas a través de `aws_ecr_lifecycle_policy`.
Retorna su dirección URL para ser reutilizada en el módulo `ECS`.

5.5.2 Azure

Recursos necesarios

Azure Containers App Azure Containers App es uno de los servicios de contenedores que ofrece Azure en su plataforma. En ellos es posible ejecutar cualquier aplicación contenerizada sin tener que preocuparse por el entorno de ejecución. Esta formado por los siguientes componentes. [68]

- *Entorno*. Son grupos de contenedores que comparten misma red virtual y que usualmente se encuentran relacionados.

- *Contenedores*. Son las imágenes en ejecución. Dentro de ellos se define la imagen a ejecutar, comandos de inicio, variables de entorno, cantidad de RAM y CPU asignados, definiciones de volúmenes o sondeos de estado.
- *Revisiones*. Controla las versiones de cada aplicación. Así, una revisión es una instantánea inmutable de una versión de un *Application Container*. De esta manera, es posible publicar una nueva versión de la aplicación y mantener al mismo tiempo la antigua, mientras el tráfico se va distribuyendo poco a poco hacia la nueva versión.

Azure Container Registry Azure Container Registry es el servicio de registro de Azure para almacenar y administriar imágenes privadas. Esta formado por.

- *Registros*. Conjunto de `Repositorios`.
- *Repositorios*. Conjunto de *Artefactos*.
- *Artefactos*. Una imagen que puede tener diferentes *tags*.

Diseño de la arquitectura

Como se puede observar en la Figura 5.14, cada uno de los servicios anteriores será un nuevo módulo.
El primero, el `ACA`, sustituirá al viejo `MIG` y creará un *Entorno* con con dos *Contenedores*, uno por microservicio, que se comunicarán entre ellos y con la base de datos.
Además, estos contenedores obtendrán sus *Imágenes* Docker a partir de un *Repositorio* situado en el otro módulo nuevo, el `ACR`. Como se puede ver, y a diferencia del nivel hecho en AWS, en este caso no es necesario hacer uso de un NAT *Gateway*, porque este viene integrado por defecto en toda su red [69].

Implementación de la arquitectura

En primer lugar, en el `azurerm_application_gateway` del módulo `AG` ahora hay una *Regla de Enrutamiento* con el `rule_type` en `PathBasedRouting`, de modo que hará uso de un *Mapa de Paths* para enrutar el tráfico a dos *Backends* en función de la ruta. Además, ahora el `backend_address_pool` usa como *Backend* el `ip_address` del *Entorno* del Container App; de la misma forma que se hace uso del FQDN de cada contenedor para comprobar su estado.
Por otro lado se han desarrollado ambos módulos diseñados previamente.

ACR Crea un repositorio privado a través de `azurerm_container_registry` y devuelve su dirección y usuario y contraseña para su acceso por el módulo `ACA`.

ACA Recibe la dirección y usuario y contraseña para el acceso al `azurerm_container_registry`, el `azurerm_subnet` donde se alojará, así como información requerida por los micro-

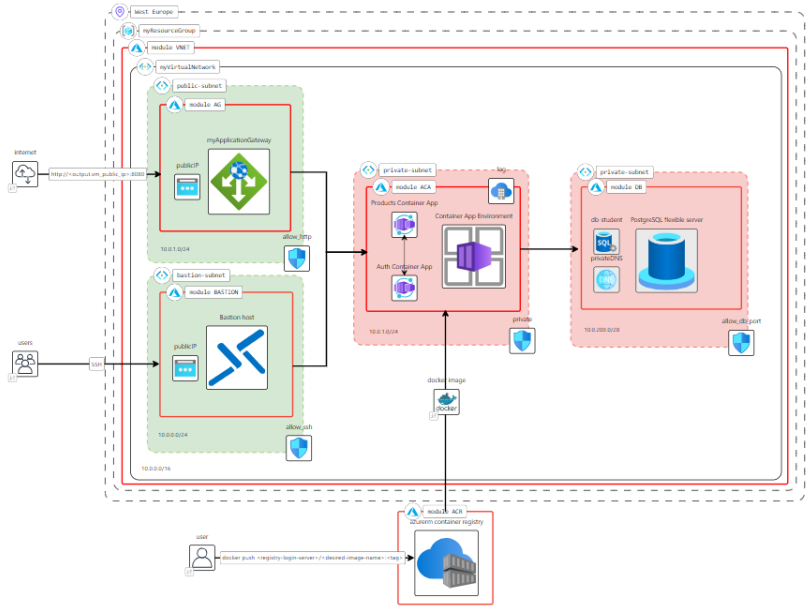

Figura 5.14: Diagrama del quinto nivel de la arquitectura en Azure

servicios como el puerto donde correrán o la dirección y usuario y contraseña del `azurerm_postgresql_flexible_server`. Crea los siguientes `resource`.

- `azurerm_container_app_environment`. Crea un *Entorno* en un `azurerm_subnet` especificado en `infrastructure_subnet_id` donde se desplegarán los `azurerm_container_app`. Además, se ha activado el argumento `internal_load_balancer_enabled` para permitir el tráfico desde un *Balanceador de Carga*.

- `azurerm_container_app`. Crea un *Contenedor* en un `azurerm_container_app_environment` identificado en `container_app_environment_id` para cada uno de los microservicios. En su bloque se especifica además el `registry` con la información del `azurerm_container_registry`; y el `template`, que es la plantilla que se usará para los despliegues con la potencia de cómputo o las variables de entorno, que en este caso además hacen uso de `secret` para proteger la información sensible.
  Por otro lado también se indica la configuración de entrada desde el bloque `ingress`, donde se especifica el puerto donde recibirá el tráfico y donde se han activado el argumento `external_enabled`, para permitir tráfico desde fuera de la VNet.

- `azurerm_log_analytics_workspace`. Crea un grupo de registros para monitorear el `azurerm_container_app_environment`.

Este módulo retorna la dirección IP del `azurerm_container_app_environment` y los FQDN de los dos `azurerm_container_app`, para que sean utilizados en el módulo `AG`.

5.5.3 Google Cloud Platform (GCP)

Recursos necesarios

Cloud Run Es un servicio para la ejecución de contenedores de manera escalable sobre la infraestructura de Google. Puede ser de dos tipos. [70]

- *Trabajos*. Ideal para código que se ejecuta y enseguida finaliza, como una secuencia de comandos.
- *Servicios*. Ideal para código que responde a solicitudes web o eventos.

Por lo tanto, se hará uso de este último.

Artifact Registry Es un servicio para el almacenamiento, no solo de imágenes Docker, sino también paquetes de lenguajes como Java o Python, y paquetes de SO como Debian. [71]

Diseño de la arquitectura

Como se puede observar en la Figura 5.15, el diseño difiere algo con respecto al resto de entornos. En primer lugar el módulo `MIG` ahora es sustituido por un módulo `CR` que representará los servicios de *Cloud Run*.
En Google, estos servicios son entornos sin servidores que no se encuentran en tu VPC. Por lo tanto, si se requiere que los recursos sin accesibilidad a Internet de una VPC puedan comunicarse con estos servicios, es necesario crear lo que se conoce como un *Conector* dentro de una *Subred* `/28` que controle el tráfico entre la VPC y los entornos sin servidores [72].
Por otra parte, también se añadirá el módulo `AR` para crear un repositorio a través del cual los servicios mencionados anteriormente obtendrán las imágenes Docker.

Implementación de la arquitectura

Como se mencionó, ahora el módulo `VPC` crea un conector a través de `google_vpc_access_connector` con un `google_compute_subnetwork` `/28` especificado en su bloque `network`. Es importante recalcar que este `google_compute_subnetwork` tiene activado el argumento `private_ip_google_access` debido a que sin él, los *Servicios* de *Cloud Run* creados si que podrían acceder a la VPC y sus servicios, pero no comunicarse entre ellos.
Por otro lado, ahora el módulo `LB` usa como *Backend* un Network Endpoint Group (NEG)[1] que se creará en el módulo `CR`. Esto supone que ciertos servicios como el `google_compute_region_health_check` dejen de estar disponibles. También se ha modificado el `google_compute_region_url_map` para distribuir el tráfico a cada microservicio en función de la URL.
Además se han creado los dos módulos mencionados previamente.

[1] Los NEG son conjuntos de servicios que deben agruparse en estos grupos para servir como *backend* a un *Balanceador de Carga* [73]

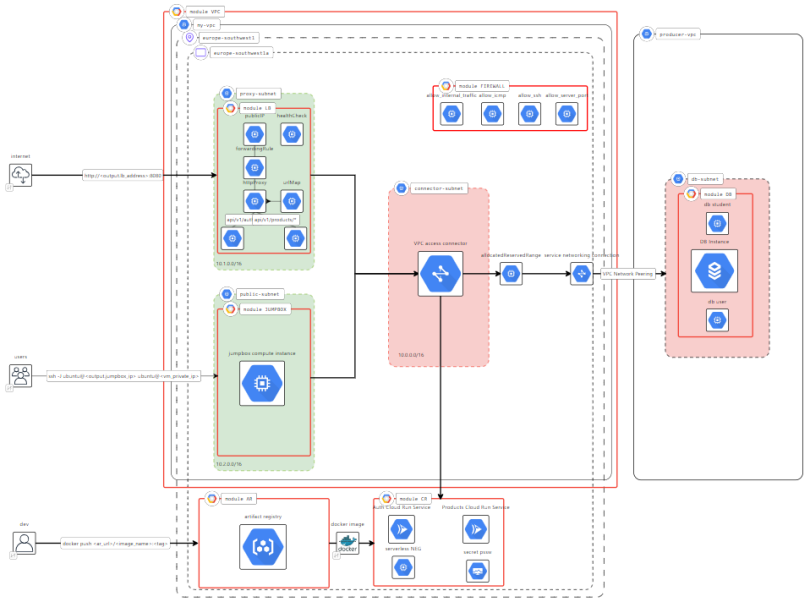

Figura 5.15: Diagrama del quinto nivel de la arquitectura en GCP

CR Recibe como parámetro el `google_vpc_access_connector` que usará para conectarse a la VPC, la URL del *Artifact Registry*, y una serie de datos necesarios por los microservicios como la dirección, los puertos donde correrán, y el usuario y la contraseña de la base de datos. Además, hace uso de los siguientes `resource`.

- `google_cloud_run_v2_service`. Crea un *Servicio* de *Cloud Run* por cada microservicio a partir de una plantilla definida en `template`, donde se especifica la ruta de la imagen a usar, las variables de entorno y el `google_vpc_access_connector` necesario para usar los servicios de la VPC. También define en `ingress` que el único modo de acceso a estos servicios será a través de un *Balanceador de Carga*.
- `google_compute_region_network_endpoint_group`. Al ser un entorno sin servidor externo a la VPC, este tipo de servicios debe organizarse en NEG para ser usado como *Backend* de un *Cloud Load Balancer*. Con este `resource` creas dicho grupo.
- `google_secret_manager_secret_iam_member` y `google_cloud_run_v2_service_iam_binding`. Serie de permisos necesarios para que los `google_cloud_run_v2_service` corran y puedan comunicarse entre ellos.

Finalmente retorna los `google_compute_region_network_endpoint_group` creados para ser reutilizados como *Backends* en el módulo `LB`.

AR Este módulo simplemente crea un *Artifact Registry* a través de `google_artifact_registry_repository` con un nombre especificado en `repository_id` y un tipo de *Artefactos* almacenados designado en `format`, en este caso Docker.

Capítulo 6

Comparativa de los entornos

A partir de toda la información recabada en las secciones anteriores, se va a proceder a realizar una comparativa entre los tres entornos *Cloud* con los que se ha estado trabajando.

6.1 Servicios de cómputo con máquinas virtuales

Los tres entornos ofrecen soluciones muy similares en este sentido con máquinas virtuales con prácticamente las mismas características.

6.1.1 Componentes

A la hora de crear una máquina virtual sin importar el entorno, es necesario determinar ciertos parámetros como la tarjeta de red, las claves de acceso, y los más importantes: el tipo de instancia y la imagen.

Tipos de instancias

Cada entorno *Cloud* ofrece una serie de tipos de instancias con un determinado número de CPU, memoria, almacenamiento y capacidad de red. Estas suelen ir agrupadas en función de su uso.

- *Propósito general*.
- *Optimizado para cómputo*.
- *Optimizados para memoria*.
- *Optimizados para almacenamiento*.
- *Computación acelerada*.
- *Computación* HCP.

AWS, con 74, y Azure, con 69, ofrecen una gran variedad de opciones con pequeñas variaciones para soluciones mucho más personalizables. GCP ofrece menos posibilidades, sin siquiera tener tipos enfocados a dos de los grupos mencionados anteriormente, *Optimizados para almacenamiento* y *Computación* HCP. Y aunque si bien ofrece alternativas dentro de otros modelos, estos no compiten con las mejor opciones de las otras dos plataformas.

Imágenes

Como ya se ha mencionado a lo largo del proyecto, las imágenes ofrecen una configuración *software* a partir de la cual se crearán las VM. Nuevamente, los tres entornos ofrecen soluciones muy similares ofreciendo una lista de imágenes propias ya preparadas o la posibilidad de crear unas personalizadas para su uso privado. Además, estos entornos ofrecen un mercado conocido como *Marketplace* donde, entre otros servicios, es posible adquirir imágenes desarrolladas por terceros. Se estima que el número de productos ofrecidos en Azure y AWS es similar con aproximádamente 17000, número que sobrepasa con creces a los 5000 estimados de GCP [74, 75, 76]. Si bien cantidad no tiene por qué reflejar calidad, esto es un gran indicador de hacia donde apuntan la mayoria de empresas de terceros.

6.1.2 Grupos de VM

Los tres entornos ofrecen la posibilidad de crear grupos de VM para que puedan escalar y ser gestionados conjuntamente.

En primer lugar, AWS y GCP permiten crear plantillas a partir de las cuales crear las VM automáticamente. Si bien Azure no ofrece esta posibilidad, las herramientas de aprovisionamiento como Terraform o las propias de cada *Cloud*, como Azure Resource Manager, ya serían más que suficientes para cubrir esta carencia.

En segundo lugar, tanto AWS como Azure y GCP ofrecen los mismos métodos de escalado. [47, 77, 78]

- *Escalado manual* cambiando el numero de instancias deseado.
- *Escalado bajo demanda* a partir de ciertas políticas que utilizan métricas sobre el estado del grupo.
- *Escalado programado* basado en horarios.
- *Escalado predictivo* que hace uso de *machine learning* para escalar un grupo en función de patrones de carga.

En tercer lugar, cada entorno gestiona la verificación del estado de las instancias de distintas maneras. En general existen dos mecanismos.

- *Comprobación directa del estado de la* VM, para verificar que está funcionando y no hay problemas *hardware* o *software* en ella.
- *Comprobación del balanceador de carga*, para verificar que la VM puede recibir solicitudes correctamente.

AWS únicamente permite utilizar una de los dos políticas, y en ambos caso las utiliza para reemplazar las VM si así lo determinan. [79]
Azure únicamente ofrece el segundo mecanismo a través de balanceadores de carga o una *extensión de estado de aplicación*. De igual manera, este mecanismo reemplaza las VM que

no pueden recibir solicitudes correctamente. [80]
GCP, por el contrario, ofrece y recomienda, la posibilidad de utilizar los dos mecanismos conjuntamente. De este modo, las comprobaciones desde el balanceador de carga serían más intensivas, ya que únicamente determinarían si una VM recibe tráfico de los usuarios, pero no la reemplaza. Por otro lado, el otro mecanismo llevaría consigo una política más conservadora puesto que sí que reemplazaría dichas VM en mal estado. [81]

6.1.3 Precio

En general, el precio de las VM variará en función de cada tipo de máquina, aunque existen distintos modelos para su compra que ofrecen diferentes descuentos.

- Bajo demanda. Modelo base donde se paga por la capacidad de computo usada por unidad de tiempo.
- Planes de ahorro. Es un compromiso de uso constante de una capacidad de computo fija por unidad de tiempo durante un plazo de años determinado a cambio de un descuento considerable.
- Reservada. Se reserva una instancia durante un plazo determinado de años que puede ser aprovisionada cuando se requiera y que conlleva un cierto descuento frente al modelo Bajo Demanda.
- Spot. Se utiliza una instancia que hace uso de la capacidad sobrante de una VM que esté disponible por un precio inferior con respecto al modelo bajo demanda.

Aunque es complicado encontrar tipos de máquinas con características exactamente iguales, en los Listados 6.1 y se han realizado comparaciones entre los tres entornos con tipos similares y con dos modelos de compra distintos.

	AWS EC2	**Azure VM**	**Instancias de *Cloud Computing***	**AWS precio ($/hora)**	**Azure precio ($/hora)**	**GCP precio ($/hora)**
Propósito general	t3.2xlarge	B8ms	e2-standard-8	$0,3328 *($0,2399)*	$0,3330 *($0,2243)*	$0.2681 *($0,1689)*
Optimizado para cómputo	c6g.2xlarge	F8s_v2	c2-standard-8	$0,2918 *($0,272)*	$0.4040 *($0,2285)*	$0,5380 *($0,3386)*
Optimizados para memoria	u-6tb1.56xlarge	Standard_M208ms_v2	m2-ultramem-208	$46.40 *($34,30)*	$44,6200 *($30,78)*	$42,19 *($25,45)*
Computación acelerada	p4d.24xlarge	ND96amsr_A100_v4	a2-ultragpu-8g	$32.77 *($24,21)*	$32.77 *($28,24)*	$40.22 *($29,36)*

Cuadro 6.1: Comparación de precios Bajo Demanda y *(Plan de Ahorro de 1 año)* con tipos de máquinas similares

6.2 Servicios de Red

6.2.1 Localizaciones

Todos los entornos *Cloud* se basan en una infraestructura de cientos de centros datos de datos distribuidos por todo el mundo. Cuanto mayor sea el número de regiones y de zonas de disponibilidad con estos centros y mejor distribuidos estén por el globo, mayor será la disponibilidad de nuestro sistema y su tiempo de respuesta.

AWS cuenta con 31 regiones, cada una con al menos 3 zonas de disponibilidad, hasta alcanzar las 99 zonas. Estas regiones se distribuyen principalmente entre Europa, Asia y Norte América, dejando a Sudamérica y África con únicamente una región cada una [82].

Azure cuenta con 38 regiones, que pueden alcanzar 3 o 4 zonas de disponibilidad. Estas regiones al igual que en AWS, están distribuidas principalmente en el hemisferio norte y Oceanía, con solo una región para África y Sudamérica, aunque en los próximos años hay prevista la creación de una nueva en Chile [83].

GCP ofrece 37 regiones, cada una con al menos 3 zonas de disponibilidad. Estas se distribuyen igualmente entre Europa, América y Asia, dejando 2 regiones en Sudamérica y ninguna en África, aunque espera abrir una nueva allí en los próximos años [84].

6.2.2 Redes y Subredes

Las Redes Virtuales en AWS son regionales, es decir, definen un rango de direcciones y actúan dentro de una región. Además, permiten crear subredes que son zonales, pues están ligados a una zona de disponibilidad dentro de dicha región. [37]
De igual manera, en Azure las Redes Virtuales son regionales y tienen un rango de direcciones. Sin embargo, y a diferencia de AWS, sus subredes no están dentro de ninguna zona de disponibilidad específica, sino que son los distintos servicios los que modelan esta característica. [40]
GCP usa enfoque distinto a los anteriores. Sus Redes Virtuales son globales, es decir, no están situadas en ninguna región específica ni contienen ningún rango de direcciones y dejan esta característica a sus subredes que, en cambio, sí que son regionales [85].

Por lo tanto, para comunicar regiones entre sí, AWS y Azure requieren de crear *network peerings* que intercambien tráfico entre sus redes virtuales. GCP, por contra, no necesita ninguna configuración extra pues, como se comentó en la Sección 5.1.3, automáticamente comunica las subredes regionales.

6.2.3 Limitaciones del tráfico

En cualquier caso, ninguno de los entornos hace una distinción directa entre subredes públicas o privadas, y cada recurso creado dentro de cada una puede tener o no una IP pública. Por lo tanto, este comportamiento se modela a través de su limitación del tráfico por medio de tablas de rutas o grupos de seguridad.

Tablas de rutas

Los tres entornos permiten crear tablas de rutas que están asociadas a redes virtuales y/o a sus subredes para distribuir su tráfico. Independientemente de la plataforma, al crear una Red Virtual se crean consigo unas rutas que limitan el tráfico. [86, 87, 44]

- El tráfico interno es modelado y permitido automáticamente independientemente del entorno.
- El tráfico externo, por contra, sí que difiere en función de la plataforma. Mientras que Azure y GCP crean rutas por defecto que permiten el tráfico hacia Internet, AWS requiere de la creación manual de un *Internet Gateway*, y de una ruta a través de este recurso para acceder a Internet.

Igualmente, es posible crear rutas que sustituyan a las ya creadas, con la pequeña excepción de GCP, que como se comentó en la Sección 5.1.3, al no poder eliminar las rutas de subred, no se puede eliminar la conectividad entre estas de este modo; y donde tampoco es posible crear rutas ligadas a subredes, por lo tanto es imposible modelar la creación de subredes públicas o privadas de esta forma.

Grupos de Seguridad

Como se ha mencionado, los grupos de seguridad permiten limitar el tráfico a través de una serie de reglas con parámetros que son iguales en todas las plataformas. Sin embargo, este servicio está ligado a diferentes recursos en función del entorno.

- En AWS, este servicio se llama Grupo de Seguridad, y está ligado a un conjunto de recursos, por lo tanto solo limitarán el tráfico de estos [39]. Sin embargo, y aunque no se han usado durante este proyecto, AWS también ofrece *firewalls* orientados a las subredes, los Network Access Control List (ACL) [88].
- En Azure, este servicio es conocido igualmente como Grupo de Seguridad, pero a diferencia de AWS, están ligados a subredes [41].
- En GCP no existen grupos de seguridad como tal, este comportamiento se modela a través del servicio *Firewall* que está asociado a toda la red virtual [45].

Por otro lado, el número de reglas creadas por defecto es distinto.

- En AWS, se crean un par de reglas para permitir el tráfico saliente pero denegar todo el entrante [39].
- En Azure, se permite el tráfico interno, el externo desde una balanceador de carga, y la salida hacia Internet [41].
- En GCP, no se crean ninguna regla de *Firewall* por defecto [45].

Independientemente de la plataforma, todas las reglas creadas en estos servicios son *stateful* [39, 41, 45].

6.2.4 Balanceador de Carga

Los balanceadores de carga son un recurso común en todos los entornos *clouds* para repartir el tráfico entre diferentes instancias.

Tipos

En general, los tres entornos ofrecen distintas opciones de balanceo de carga que trabajan tanto en la *Capa de Aplicación* como en la *Capa de Red*. Sin embargo, debido a la globalidad de sus redes virtuales, GCP es el único que ofrece un servicio de balanceo de carga global que permita distribuir el tráfico entre regiones. Aunque tanto AWS como Azure ofrecen alternativas para modelar este comportamiento.

- En AWS, se puede hacer uso de su servicio de DNS, *Route 53*, en conjunto con un Balanceador de Carga regional. [89]
- En Azure, se puede hacer uso la combinación de diferentes recursos como un Azure Front Door y un Application Gateway regional. [51]

Componentes

Los componentes que forman un Balanceador de Carga de Aplicación en los tres entornos difieren un poco, aunque en general pueden agruparse en tres partes principales que acaban ofreciendo las mismas opciones.

- *Frontend*. Reciben el trafico en una dirección, puerto y protocolo específico. En el caso de AWS y Azure estos se llaman *Listeners*, mientras que en GCP son *Reglas de Reenvio*.
- *Reglas de enrutamiento*. Reglas asociadas a cada *Frontend* creado que establece como distribuir el tráfico entre los *Backends* en función de una serie de condiciones. En AWS son conocidos como *Reglas de Listener*. En Azure como *Reglas de enrutamiento*, y hacen uso de *Configuraciones de* HTTP. Mientras que en GCP, corresponde al *Proxy de Destino* que utilizan *Mapas de* URL.
- *Backend*. Reciben el tráfico distribuido. En AWS son conocidos como *Target Groups*, mientras que en Azure y GCP son *Backends*.

6.2.5 Precio

En las tres plataformas no hay coste adicional por la creación de una Red Virtual, pero si por el tráfico que genera. Este depende en parte del servicio que genere el tráfico. Pero aproximadamente los precios son los siguientes [90, 91, 92].

El tráfico entrante desde Internet es gratuito en todos los entornos, mientras que el saliente puede ser consultado en el Cuadro 6.2.

El tráfico entre zonas de disponibilidad es igual en las tres plataformas, $0,01 por GB tanto

en entrada como en salida, a excepción de GCP, que no cobra la entrada.

El tráfico entre regiones son $0,02 por GB en AWS, mientras que en Azure y GCP varía entre las regiones, siendo la misma cantidad entre regiones Europeas pero pudiendo llegar a los $0,16 por GB para alcanzar otras regiones.

	AWS	Azure	GCP
Estándar	▪ Primeros 100 GB/mes gratuitos ▪ Siguientes 10 Tb/mes $0,09/GB ▪ Siguientes 40 TB/mes $0,085/GB ▪ Siguientes 100 TB/mes $0,07/GB ▪ Superior a 150 TB/mes $0,05/GB	▪ Primeros 100 GB/mes gratuitos ▪ Siguientes 10 Tb/mes $0,08/GB ▪ Siguientes 40 TB/mes $0,065/GB ▪ Siguientes 100 TB/mes $0,06/GB ▪ Siguientes 150 TB/mes $0,04/GB	▪ Primeros 1 TiB/mes $0,085/GB ▪ Siguientes 10 TiB/mes $0.065/GB ▪ Superior a 10TiB/mes $0.045/GB
Premium	n/a	▪ Primeros 100 GB/mes gratuitos ▪ Siguientes 10 Tb/mes $0,087/GB ▪ Siguientes 40 TB/mes $0,083/GB ▪ Siguientes 100 TB/mes $0,07/GB ▪ Superior a 150 TB/mes $0,05/GB	▪ Hasta 10 TiB/mes $0.12/GB ▪ Hasta 150 TiB/mes $0.11/GB ▪ Superior a 150 TiB/mes $0.08/GB

Cuadro 6.2: Comparación de precios del tráfico externo saliente desde regiones europeas a Internet

Azure y GCP ofrecen dos tipos de redes con distinta latencia y disponibilidad, aunque por defecto hacen uso de su versión *premium* [93, 92, 91]

6.3 Servicios de Bases de Datos SQL

Como se podrá comprobar en la siguiente sección, los tres proveedores ofrecen distintos servicios de bases de datos, sin embargo, el resto de comparaciones sobre escalabilidad, rendimiento, disponibilidad o precio se realizarán desde las bases de datos relacionales usadas en este proyecto.

6.3.1 Tipos

Los tres entornos ofrecen distintos servicios de bases de datos.

Bases de datos relacionales

AWS ofrece su servicio RDS con una mayor oferta para motores, tanto *open-source* (MySQL, PostgreSQL y MariaDB), como de terceros (Oracle y SQL Server). También ofrece como una versión *serverless* llamada Amazon Aurora compatible con MySQL y PostgreSQL. [94]

Azure ofrece Azure SQL que son una familia de servicios bases de datos que hacen uso de su propio motor, Microsoft SQL Server, lo que les aporta ciertas ventajas como la implementación de nuevas funcionalidades antes de su lanzamiento [95]. Aunque también ofrecen servicios para otros motores como MySQL, PostgreSQL, MariaDB [96].
Todos estos servicios mencionados pueden ser implementados mediante soluciones PaaS o IaaS, donde además es posible implementar bases de datos Oracle [97].

GCP por su parte ofrece Cloud SQL únicamente para MySQL, PostgreSQL y SQL Server; AlloyDB, un servicio adaptado exclusivamente para PostgreSQL; y una solución *bare metal* para Oracle.

Bases de datos no relacionales

AWS ofrece una gran variedad de servicios como DynamoDB, que es una base de datos de clave-valor y documentos; DocumentDB, que es una base de datos de documentos compatible con MongoDB; Nepturne, para gráficos; MemoryDB, servicio de base de datos en memoria duradero y compatible con Redis; ElasticCache para el almacenamiento *cache*; o Timestream, adaptado para el almacenamiento de millones de eventos diarios como en el IoT. [94]

Azure por su parte ofrece Azure Table Storage, un almacén de pares clave-valor y Azure Cosmos DB, que incluye varios modelos de datos como pares clave-valor, documentos, gráficos y columnas. También ofrece almacenamiento en memoria caché, Cache for Redis. [96]

GCP ofrece Firestore, para documentos; Cloud Bigtable, para pares clave-valor; y Memorystore, servicio de memoria escalable para Redis y Memcached. [98]

6.3.2 Rendimiento

En cuanto al almacenamiento, AWS ofrece la opción más personalizable a la vez que mayor prestaciones. Como ya se comentó anteriormente, dispone de varios tipos de almacenamiento[1], que pueden alcanzar hasta los 64 TB. [57]

- Los `gp2`, útiles para cargas de trabajo no tan grandes, ofrecen un rendimiento base de 3 IOPS por GB hasta alcanzar los 64 000. Además ofrece un *throughput* que varía desde los 128 hasta los 1 000 MiB/s en función del número de GB.

[1]Los valores mostrados pueden variar ligeramente en función del motor de base de datos usado, especialmente con SQL Server donde disminuyen bastante

- `gp3` es algo más potente y personalizable, pues ofrece un rendimiento base de 3 000 IOPS y 125 MiB/s hasta alcanzar los 400 GiB, donde permite escalar estos datos en un rango hasta los 64 000 IOPS y 4 000 MiB/s.
- `io1` es usado para cargas que requieran más potencia. De nuevo su rendimiento varía en función del almacenamiento, ofreciendo un rango de 1 000 a 19 950 IOPS y 500 MiB/s los primeros 400 GiB, para acabar permitiendo alcanzar los 256,000 IOPS y los 4,000 MiB/s una vez superado este límite.

Azure, por su parte, ofrece tamaños de memoria desde los 32 GiB con 120 IOPS hasta los 32 TiB con 20 000 IOPS. [99]

GCP también ofrece distintos tipos de discos persistentes[2] que escalan en función del número de GB y que ofrecen un máximo de 64 TiB cada uno. [100]

- *Estándar*. Permiten alcanzar los 7 500 IOPS y 1 200 MB/seg en lecturas, y los 15 000 IOPS y 400 MB/seg en escrituras.
- *Balanceado*. Escalan más rapidamente que los anteriores hasta alcanzar los 80 000 IOPS y 1 200 MB/seg en escrituras y lecturas.
- SSD. Escalan casi 5 veces más rápido que los anteriores hasta alcanzar los 100 000 IOPS y 1 200 MB/seg en escrituras y lecturas.
- *Extremo*. Son los únicos que no escalan con el tamaño, sino que permite elegir un rango de IOPS 2,500 a 120,000 IOPS, y una capacidad de procesamiento máxima de 4 000 MB/seg en lecturas y 3 000 MB/seg en escrituras.

Independientemente del entorno estos valores están limitados por el tipo de máquina escogido. Donde AWS ofrece tamaños de hasta 96 CPU virtuales y 768 GiB de memoria, Azure 96 vCPU y 721 GiB, y GCP 96 vCPU y 624 GB.

6.3.3 Escalabilidad

Los tres entornos ofrecen políticas de escalado muy similares, como aumentar o disminuir el tamaño de las instancias o su almacenamiento, para configurar un escalado vertical; o como la creación de replicas de lectura para el escalado horizontal. [101, 102, 62]

6.3.4 Disponibilidad

Al hablar de la disponibilidad hay dos características fundamentales a tener en cuenta, la Alta Disponibilidad y la Recuperación ante Desastres.

La Alta Disponibilidad permite crear copias en diferentes Zonas de Disponibilidad o en mismos centros locales para mejorar la tolerancia a fallos. Los tres entornos *Cloud* ofrecen

[2]También se hace una distinción entre discos zonales y regionales, que pueden ofrece una mayor disponibilidad a cambio de algo menos de rendimiento. Los datos ofrecidos en esta sección usan como referencia los discos zonales

estos servicios, aunque el SLA es distinto, siendo AWS y GCP los que menos tienen con 99,995 %, frente al 99,999 % de Azure [103, 104, 105].

La Recuperación ante Desastres permite la recuperación del sistema y la minimización de datos perdido para casos donde falla una región entera. Para ello de nuevo las tres plataformas ofrecen funcionalidades muy similares como la creación de réplicas de lectura en diferentes regiones o copias de seguridad manuales y automáticas diarias, donde GCP se impone en su tiempo de almacenamiento durante 1 año, frente a los 35 días máximos de AWS y Azure. [106, 107, 108]

6.3.5 Precio

Al igual que ocurría con las EC2, AWS cobra en función del tipo de instancia seleccionado. Este precio puede ser bajo demanda, donde se paga por uso; o por reserva, donde se reserva una instancia durante uno (25 - 37 % descuento) o tres (45 - 60 % descuento) años. Además, según el tipo de almacenamiento seleccionado se añade un precio extra [109].

- `gp2`. $0,127 GB/mes.
- `gp3`. $0,127 GB/mes, y $0,022 IOPS y $0,088 Mb/s al mes por encima del rendimiento base.
- `io1`. $0,138 GB/mes y $0,11 IOPS/mes.

Al igual que ocurría con AWS, en Azure el precio de su base de datos depende, en primera instancia, del tipo de base de datos que se haya seleccionado, con modelos de pago bajo demanda y de reserva de uno (40 % de ahorro) o tres (60 % de ahorro) años. Adicionalmente se cobrará el almacenamiento, $0,137 GB/mes [110].

GCP cobra por el número de vCPU y memoria utilizada cada hora, $0.04956 por número y $0.00840 por GB respectivamente. Al igual que ocurría en las anteriores plataformas, ofrecen tanto un modelo bajo demanda, como uno por reserva por uno (25 % de ahorro) o tres (52 % de ahorro) años. A esto se le suma el coste de almacenamiento, $0.170 GB/mes en SSD o $0.090 GB/mes en HDD [111].

En cualquiera los entornos, el uso de réplicas incurre su coste correspondiente tal y como si fueran una instancia más con su misma capacidad.

6.4 Servicios de Contenedores

Todos los entornos tienen un servicio de orquestación de contenedores de Kubernetes, pero a parte de este servicio, cada uno ofrece opciones adicionales que se asimilan y se diferencian en algunos aspectos.
AWS ofrece su propio servicio de orquestación llamado ECS, que permite la creación de instancias en máquinas EC2 y sin servidor como Fargate. Por otro lado también ofrecen un servicio FAAS, Lambda.

Azure, por su parte, ofrece una gran variedad de servicios como Azure Spring Apps, ideal para ejecutar aplicaciones desarrolladas en Spring; modelos FaaS como Aure Functions; Azure App Service, para el hospedaje de sitios web; Azure Container Instances, útil para pocos contenedores que no requieren de mucha comunicación y sin mucho margen de escalado; o Azure Container Apps, que ha sido utilizado durante este proyecto; entre muchos otros. [112]

Por último, GCP no ofrece tantas alternativas. Alguna herramienta FaaS como Cloud Functions o Firebase Cloud Functions (orientado a *backends* de dispositivos móviles) y App Engine o Cloud Run, servicios de contenedores *serverless* con mínimas diferencias entre ellos. [113]

Las siguientes comparaciones se realizarán sobre los servicios escogidos y utilizados durante este proyecto, AWS Fargate, Azure Container Apps y Cloud Run.

6.4.1 Componentes

Tanto el AWS Fargate como el Azure Container Apps tienen una arquitectura similar donde se crean clústers o entornos, que son agrupaciones lógicas de contenedores. Además, existen unas definiciones de tareas o revisiones que describen una serie de tareas o servicios, equivalentes a los *pods* en Kubernetes. [66, 68]

Cloud Run no presenta de un entorno común. Simplemente permite la creación de servicios cuya implementación se llevará a cabo a través de sucesivas revisiones, equivalentes a las definiciones de tareas. [114]

6.4.2 Escalado

AWS Fargate y Azure Container Apps ofrece la posibilidad de escalar horizontalmente en función de ciertas métricas. [115, 116]

Cloud Run escala automáticamente dentro de un mínimo y máximo establecido, para satisfacer todas las solicitudes que recibe, intentando mantener siempre aproximadamente un 60 % de uso máximo de CPU. Estos valores se encuentran entre 0 y 100, aunque es posible aumentarlo en casos excepcionales. Sin embargo, no permite hacer este escalado en función de ciertas métricas, como si hacían las anteriores plataformas. [117]

6.4.3 Recursos disponibles

AWS es la que mayor cantidad y más variedad de recursos ofrece, desde 0,25 vCPU y 0,5 GB de memoria hasta 16 vCPU y 120 GB [118]. Le sigue GCP, que permite crear contenedores de entre 512 MiB a 32 GiB de memoria y 1 a 8 CPU [119]. Por último, Azure únicamente ofrece desde 0,25 vCPU y 0,4 GB memoria hasta 2 vCPU y 4 GB, aunque se puede incrementar hablando con Soporte [120].

6.4.4 Registros de contenedores

Todos los servicios de contenedores de cada plataforma permiten obtener las imágenes desde registros de contenedores de terceros, como Docker Registry, o desde propios, donde cada proveedor ofrece su propio servicio, aunque muy similar entre ellos.

Los tres entornos ofrecen servicios con replicación multi-regional, aunque únicamente Azure tiene redundancia de zona. Además, los tres ofrecen la posibilidad de crear repositorios públicos o privados, que únicamente serán accedidos por aquellas cuentas con los permisos necesarios. También incluyen diferentes mecanismos de encriptación.

Google Cloud Platform (GCP) es la única de las tres que, a parte de ser compatible con formatos de imágenes de contenedor Docker V2 y Open Container Initiative (OCI), también permite almacenar paquetes de lenguaje como Java, Node.js y Python, y paquetes de SO como Debian y Red Hat Package Manager (RPM). [71]

Por su parte, Azure ofrece diferentes niveles donde muchas de las ventajas mencionadas anteriormente solo se encuentran en el último *tier*. [121]

Los precios de estos repositorios son muy similares. Tanto AWS como GCP cobran $0,10 por GB al mes, aunque AWS regala los primeros 50 en repositorios públicos. [122, 123] Azure, por su parte, cobra por días y en función del *tier*, siendo $1,667 por día la del *premium*. Aunque con la misma cantidad de GB que sus competidores, acaba alcanzando el mismo precio. [124]

6.4.5 Costes

Los precios de uso de estos servicios bajo demanda pueden comprobarse en la Tabla 6.3.

	AWS Fargate Linux/X86 *($/segundo)*	**AWS Fargate** Linux/ARM *($/segundo)*	**AWS Fargate** Windows/X86 *($/segundo)*	**Azure Container Apps** *($/segundo)*	**GCP Cloud Run** *($/segundo)*
vCPU/seg	$0,0000129	$0,00001034	$0,000042	$0,000034	$0,000024
GB/seg	$0,00000142	$0,00000114	$0,00000321	$0,0000043	$0,0000025
Peticiones	*n/a*	*n/a*	*n/a*	$0,56/millón	$0,40/millón

Cuadro 6.3: Precios de los servicios de contenedores utilizados
Los servicios de Azure y GCP muestran su precio con la CPU activa, pero en caso de no estar ejecutándose su precio de *vCPU/seg* disminuye a $0,000004 y $0,0000024 respectivamente [125, 126, 127]

Capítulo 7

Conclusiones

Para finalizar este documento se recogerá una serie de conclusiones sobre los objetivos planteados en el Capítulo 2, la demostración de las competencias de la tecnología cursada y, por último, se expondrán una serie de funcionalidades que podrían ser añadidas a la arquitectura construida en futuros trabajos.

7.1 Consecución de los objetivos

A continuación se detallará la consecución de los objetivos específicos propuestos en la Sección 2.2.

- *Análisis de los servicios Cloud*. A lo largo del Capítulo 5 se han realizado un análisis tanto de los recursos necesarios de cada entorno *Cloud*, así como su encaje dentro del diseño de la arquitectura. Por otro lado, para la realización del Capítulo 6 fue necesario hacer un análisis exhaustivo de cada uno de estos recursos.
- *Estudio sobre Terraform*. Tal y como se puede comprobar en la Sección 3.2, se ha realizado un estudio a fondo tanto de la arquitectura subyacento de Terraform, como de su lenguaje y forma de trabajo. Por otro lado, a lo largo del Capítulo 5 se han implementado distintos niveles de una arquitectura únicamente con esta herramienta. Por lo que queda claro que se ha obtenido un conocimiento avanzado en la misma.
- *Diseño y construcción de cada arquitectura Cloud*. Durante el Capítulo 5 se ha creado una arquitectura en cada uno de los entornos *Cloud* guiado a través de distintos niveles, cada uno ofreciendo una funcionalidad nueva. Cada uno de dichos niveles viene acompañado de un análisis de los recursos necesarios para su creación, un diseño la arquitectura a construir, y su implementación con Terraform.
- *Comparativa de los entornos*. Durante el Capítulo 6, se ha realizado una comparativa de los servicios utilizados durante el proyecto, orientado a conocer en detalle las diferencias en su funcionamiento, componentes, características y coste.

La consecución de todos estos objetivos ha dado como resultado la obtención del objetivo general planteado en la Sección 2.1. Pues se ha conseguido construir una arquitectura *Cloud* de distintos microservicios que se comunican entre ellos, con una base de datos y con los usuarios. Esta arquitectura ha conseguido replicarse en cada una de las plataformas ya

mencionadas y siempre haciendo uso de Terraform, herramienta de la que se ha obtenido un conocimiento bastante amplio.

7.2 Competencias

A continuación se justificarán la demostración de las competencias de la tecnología cursada por el alumno.

- *Capacidad para comprender, aplicar y gestionar la garantía y seguridad de los sistemas informáticos*. Durante el desarrollo del proyecto se han tomado distintas medidas para garantizar la seguridad del sistema, como la preocupación a la hora de proteger cada recurso situándolo en subredes públicas o privadas según corresponda, limitando su tabla de rutas, creando grupos de seguridad y reglas *firewall* para limitar su tráfico entrante y saliente, y haciendo uso de nodos *jumpbox* para limitar su acceso.
- *Capacidad para diseñar, desplegar, administrar y gestionar redes de computadores*. Uno de los puntos esenciales a la hora de desarrollar una arquitectura en la nube es configurar y gestionar las Redes Privadas Virtuales donde se encuentran sus recursos desplegados. Por lo tanto, uno de los retos que ha supuesto este proyecto ha sido el correcto diseño y creación de estas redes virtuales.
- *Capacidad para analizar, evaluar, seleccionar y configurar plataformas hardware para el desarrollo y ejecución de aplicaciones y servicios informáticos*. Las arquitecturas construidas requerían de la configuración de una serie de servicios hardware en la nube como Balanceadores de Carga, Puertas de Enlace, Máquinas Virtuales o Bases de Datos.

7.3 Trabajo futuro

Pese a lo consecución de los objetivos mencionados anteriormente, es posible añadir algunas mejoras sobre ciertos aspectos que, si bien se han cubierto en cierta manera, se ha hecho de una manera más superficial principalmente por falta de tiempo y espacio.

- *Añadir un sistema de monitorización más complejo*. Si bien a lo largo de la creación de algunos niveles de la arquitectura, se han añadido algún que otro sistema de monitorización, su uso ha estado más centrado en comprobar que la arquitectura construida funcionaba como se planeaba. Sin embargo, todos los entornos ofrecen mecanismos mucho más complejos para monitorear el estado tanto de sus recursos como de los servicios desplegados en ellos. Por lo tanto un posible futuro añadido sería crear un sistema de monitorización más complejo que pueda incluir también alarmas.
- *Creación de políticas de escalado más complejas*. De la misma forma, a lo largo de este proyecto, se han incluido ciertas métricas para el escalado automático de sus recursos en algunos diseños. Sin embargo, estas solían ser muy básicas como alcanzar

un umbral de uso de CPU.

Tal como se ha explicado durante secciones anteriores, estos entornos ofrecen mecanismos mucho más complejos que incluso pueden hacer uso de *machine learning* para una escalado mucho más adaptativo.

- *Mejora de la disponibilidad a través de la multiregionalidad*. Una de las características importantes de los entornos *Cloud* es su capacidad para crear sistemas con alta disponibilidad a través del uso de distintas regiones y zonas de disponibilidad. Si bien esto se ha considerado a lo largo del diseño, por falta de tiempo y espacio, únicamente AWS ha contado verdaderamente con un sistema desplegado en distintas zonas de disponibilidad, y, en ningún caso, se ha incluido disponibilidad multiregional.

ANEXOS

Anexo A

Repositorio del proyecto

A.1 Repositorio en GitHub

A lo largo del proyecto se ha hecho uso de GitHub para controlar las versiones del código de Terraform, Packer, Docker y los microservicios creados en Java.
Este se puede encontrar en `https://github.com/algaru01/tfg-repo`.

Referencias

[1] Flexera, "State of the cloud report 2023," tech. rep., 2023.

[2] R. Nayak, "When to use which infrastructure-as-code tool." https://dzone.com/articles/when-to-use-which-infrastructure-as-code-tool, 2019. (Visitado 05/06/2023).

[3] GitGuardian, "Data security-an introduction to aws kms and hashicorp vault." https://blog.gitguardian.com/talking-about-data-security-an-introduction-to-aws-kms-and-hashicorp-vault. (Visitado 20/06/2023).

[4] I. C. Team and I. Cloud, "A brief history of cloud computing." https://www.ibm.com/cloud/blog/cloud-computing-history. (Visitado 20/06/2023).

[5] N. Kilari, "Cloud computing - an overview evolution," *International Journal of Scientific Research in Computer Science, Engineering and Information Technology (IJSRCSEIT)*, vol. 3, pp. 149–152, 2018.

[6] Dar and D. Ravindran, "A comprehensive study on cloud computing," 04 2018.

[7] P. Mell and G. Tim, "The nist definition of cloud computing share to facebook," Tech. Rep. NIST Special Publication (SP) 800-145, National Institute of Standards and Technology, Gaithersburg, MD, 2011.

[8] Flexera, "State of the cloud report 2021," tech. rep., 2021.

[9] Y. Brikman, *Terraform: Up and Running, 3rd Edition*. O'Reilly Media, Inc., 2022.

[10] J. Klein and D. Reynolds, "Infrastructure as code–final report."

[11] H. Developer, "Plugin development - how terraform works with plugins | terraform | hashicorp developer." https://developer.hashicorp.com/terraform/plugin/how-terraform-works. (Visitado 02/06/2023).

[12] H. Developer, "Configuration language - providers." https://developer.hashicorp.com/terraform/language/providers. (Visitado 02/06/2023).

[13] H. Developer, “Configuration language - provisioners.” https://developer.hashicorp.com/terraform/language/resources/provisioners/syntax. (Visitado 04/06/2023).

[14] H. Developer, “Configuration language - state.” https://developer.hashicorp.com/terraform/language/state. (Visitado 20/06/2023).

[15] H. Developer, “Configuration language - backend.” https://developer.hashicorp.com/terraform/language/settings/backends/configuration. (Visitado 20/06/2023).

[16] H. Developer, “Terraform home - the core terraform workflow.” https://developer.hashicorp.com/terraform/tutorials/cli/init. (Visitado 20/06/2023).

[17] H. Developer, “Terraform home - json configuration syntax.” https://developer.hashicorp.com/terraform/language/syntax/json. (Visitado 20/06/2023).

[18] H. Developer, “Configuration language - configuration syntax.” https://developer.hashicorp.com/terraform/language/syntax/configuration. (Visitado 20/06/2023).

[19] H. Developer, “Configuration home - terraform settings.” https://developer.hashicorp.com/terraform/language/settings. (Visitado 20/06/2023).

[20] H. Developer, “Configuration home - provider configuration.” https://developer.hashicorp.com/terraform/language/providers/configuration. (Visitado 20/06/2023).

[21] H. Developer, “Configuration home - resource blocks.” https://developer.hashicorp.com/terraform/language/resources/syntax. (Visitado 20/06/2023).

[22] H. Developer, “Configuration language - data sources.” https://developer.hashicorp.com/terraform/language/data-sources. (Visitado 20/06/2023).

[23] H. Developer, “Configuration language - types and values.” https://developer.hashicorp.com/terraform/language/expressions/types. (Visitado 20/06/2023).

[24] H. Developer, “Configuration language - modules.” https://developer.hashicorp.com/terraform/language/modules. (Visitado 20/06/2023).

[25] H. Developer, “Configuration language - module blocks.” https://developer.hashicorp.com/terraform/language/modules/syntax. (Visitado 20/06/2023).

[26] H. Developer, “Configuration language - module sources.” https://developer.hashicorp.com/terraform/language/modules/sources. (Visitado 20/06/2023).

[27] H. Developer, “Configuration language - input variables.” https://developer.hashicorp.com/terraform/language/values/variables. (Visitado 20/06/2023).

[28] H. Developer, “Configuration language - output variables.” https://developer.hashicorp.com/terraform/language/values/outputs. (Visitado 20/06/2023).

[29] H. Developer, “Configuration language - local variables.” https://developer.hashicorp.com/terraform/language/values/locals. (Visitado 20/06/2023).

[30] H. Developer, “Intro - terraform vs. chef, puppet, etc..” https://developer.hashicorp.com/terraform/intro/vs/chef-puppet. (Visitado 20/06/2023).

[31] H. Developer, “Intro - terraform vs. cloudformation, heat, etc..” https://developer.hashicorp.com/terraform/intro/vs/chef-puppet. (Visitado 20/06/2023).

[32] Amazon, “¿qué es la aws cdk?.” (Visitado 21/06/2023).

[33] H. Developer, “Cdk for terraform.” https://developer.hashicorp.com/terraform/cdktf. (Visitado 21/06/2023).

[34] Pulumi, “Pulumi vs terraform.” https://www.pulumi.com/docs/concepts/vs/terraform. (Visitado 21/06/2023).

[35] A. Stellman and J. Greene, *Learning Agile*. O’Reilly Media, Inc., 2014.

[36] L. E. V. A. y Eliécer Herrera Uribe, “Del manifiesto ágil sus valores y principios,” 2007.

[37] Amazon, “Cómo funciona amazon vpc.” https://docs.aws.amazon.com/es_es/vpc/latest/userguide/how-it-works.html. (Visitado 11/06/2023).

[38] Amazon, “¿qué es amazon ec2?.” "https://docs.aws.amazon.com/es_es/AWSEC2/latest/UserGuide/concepts.html". (Visitado 19/06/2023).

[39] Amazon, “Control traffic to your aws resources using security groups.” https://docs.aws.amazon.com/vpc/latest/userguide/vpc-security-groups.html. (Visitado 11/06/2023).

[40] Microsoft, “What is azure virtual network?.” https://learn.microsoft.com/en-us/azure/virtual-network/virtual-networks-overview. (Visitado 11/06/2023).

[41] Microsoft, “Network security groups.” https://learn.microsoft.com/en-us/azure/virtual-network/network-security-groups-overview. (Visitado 11/06/2023).

[42] Microsoft, “Virtual machines in azure.” "https://learn.microsoft.com/en-us/azure/virtual-machines/overview". (Visitado 19/06/2023).

[43] Google, “Redes de vpc.” "https://cloud.google.com/vpc/docs/vpc". (Visitado 19/06/2023).

[44] Google, “Rutas.” https://cloud.google.com/vpc/docs/routes. (Visitado 11/06/2023).

[45] Google, “Vpc firewall rules.” https://cloud.google.com/firewall/docs/firewalls. (Visitado 11/06/2023).

[46] Google, “Instancias de máquina virtual.” "https://cloud.google.com/compute/docs/instances". (Visitado 19/06/2023).

[47] Amazon, “What is amazon ec2 auto scaling?.” https://docs.aws.amazon.com/autoscaling/ec2/userguide/what-is-amazon-ec2-auto-scaling.html. (Visitado 13/06/2023).

[48] Amazon, “¿qué es elastic load balancing?.” https://docs.aws.amazon.com/es_es/elasticloadbalancing/latest/userguide/what-is-load-balancing.html. (Visitado 12/06/2023).

[49] Amazon, “Características de elastic load balancing - comparaciones de productos.” https://aws.amazon.com/es/elasticloadbalancing/features/#Product_comparisons. (Visitado 12/06/2023).

[50] Amazon, “¿qué es un application load balancer?.” https://docs.aws.amazon.com/es_es/elasticloadbalancing/latest/application/introduction.html. (Visitado 12/06/2023).

[51] Microsoft, “Load-balancing options.” https://learn.microsoft.com/en-us/azure/architecture/guide/technology-choices/load-balancing-overview. (Visitado 12/06/2023).

[52] Microsoft, “Introducción a la configuración de application gateway.” https://learn.microsoft.com/es-es/azure/application-gateway/configuration-overview. (Visitado 12/06/2023).

[53] Google, “Descripción general de cloud load balancing.” https://cloud.google.com/load-balancing/docs/load-balancing-overview. (Visitado 12/06/2023).

[54] Google, “Descripción general del balanceador de cargas http(s) externo.” https://cloud.google.com/load-balancing/docs/https. (Visitado 12/06/2023).

[55] Google, “Grupos de instancias.” "https://cloud.google.com/compute/docs/instance-groups". (Visitado 19/06/2023).

[56] Amazon, “¿qué es amazon relational database service (amazon rds)?.” "https://docs.aws.amazon.com/es_es/AmazonRDS/latest/UserGuide/Welcome.html". (Visitado 17/06/2023).

[57] Amazon, “Amazon rds db instance storage.” https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Storage.html. (Visitado 17/06/2023).

[58] Amazon, “Configuración y administración de una implementación multi-az.” "https://docs.aws.amazon.com/es_es/AmazonRDS/latest/UserGuide/Concepts.MultiAZ.html". (Visitado 17/06/2023).

[59] Amazon, “Uso de una instancia de base de datos en una vpc.” "https: https://docs.aws.amazon.com/es_es/AmazonRDS/latest/UserGuide/USER_VPC.WorkingWithRDSInstanceinaVPC.html". (Visitado 19/06/2023).

[60] Microsoft, “¿qué es azure database for postgresql?.” "https://learn.microsoft.com/es-es/azure/postgresql/single-server/overview". (Visitado 17/06/2023).

[61] Microsoft, “Introducción al servidor flexible de azure database for postgresql..” https://learn.microsoft.com/es-es/azure/postgresql/flexible-server/overview. (Visitado 17/06/2023).

[62] Google, “¿qué es cloud sql?.” https://cloud.google.com/sql/docs/postgres/introduction. (Visitado 16/06/2023).

[63] Google, “Private service connect.” "https://cloud.google.com/vpc/docs/private-service-connect". (Visitado 19/06/2023).

[64] Microsoft, “Working with nsg access and azure bastion.” https://learn.microsoft.com/en-us/azure/bastion/bastion-nsg. (Visitado 13/06/2023).

[65] Amazon, “¿qué es amazon elastic container service?.” "https://docs.aws.amazon.com/es_es/AmazonECS/latest/developerguide/Welcome.html". (Visitado 19/06/2023).

[66] Amazon, “¿qué es aws fargate?.” https://docs.aws.amazon.com/es_es/AmazonECS/latest/userguide/what-is-fargate.html. (Visitado 19/06/2023).

[67] Amazon, “Amazon ecs task execution iam role.” https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_execution_IAM_role.html. (Visitado 14/06/2023).

[68] Microsoft, “Entornos de azure container apps.” https://learn.microsoft.com/es-es/azure/container-apps/environment. (Visitado 14/06/2023).

[69] Microsoft, “Configure ip addresses for an azure network interface.” https://learn.microsoft.com/en-us/azure/virtual-network/ip-services/virtual-network-network-interface-addresses?tabs=nic-address-portal. (Visitado 22/06/2023).

[70] Google, "¿qué es cloud run?." "https://cloud.google.com/run/docs/overview/what-is-cloud-run". (Visitado 19/06/2023).

[71] Google, "Formatos admitidos." https://cloud.google.com/artifact-registry/docs/supported-formats. (Visitado 14/06/2023).

[72] Google, "Acceso a vpc sin servidores." https://cloud.google.com/vpc/docs/serverless-vpc-access. (Visitado 06/06/2023).

[73] Google, "Descripción general de los grupos de extremos de red." "https://cloud.google.com/load-balancing/docs/negs". (Visitado 19/06/2023).

[74] L. S. Vailshery, "Global number of products and services offered on aws marketplace 2021, by category," tech. rep., 2021.

[75] L. S. Vailshery, "Global number of leading services offered on microsoft azure 2023, by category," tech. rep., 2023.

[76] L. S. Vailshery, "Global number of services offered on google cloud marketplace 2022, by category," tech. rep., 2022.

[77] Microsoft, "Overview of autoscale with azure set scali scale sets." https://learn.microsoft.com/en-us/azure/virtual-machine-scale-sets/virtual-machine-scale-sets-autoscale-overviews. (Visitado 19/06/2023).

[78] Google, "Ajuste de escala automático de grupos de instancias." https://cloud.google.com/compute/docs/autoscaler. (Visitado 19/06/2023).

[79] Amazon, "Comprobaciones de estado para instancias de auto scaling." https://docs.aws.amazon.com/es_es/autoscaling/ec2/userguide/ec2-auto-scaling-health-checks.html. (Visitado 12/06/2023).

[80] Microsoft, "Reparaciones automáticas de instancias de azure virtual machine scale sets." https://learn.microsoft.com/es-es/azure/virtual-machine-scale-sets/virtual-machine-scale-sets-automatic-instance-repairs. (Visitado 12/06/2023).

[81] Google, "Configura una verificación de estado de la aplicación y una reparación automática." https://cloud.google.com/compute/docs/instance-groups/autohealing-instances-in-migs. (Visitado 12/06/2023).

[82] Amazon, "Infraestructura global de aws." https://aws.amazon.com/es/about-aws/global-infrastructure/. (Visitado 17/06/2023).

[83] Microsoft, "Servicio de zona de disponibilidad y compatibilidad regional." https://learn.microsoft.com/es-es/

azure/reliability/availability-zones-service-support#azure-regions-with-availability-zone-support. (Visitado 17/06/2023).

[84] Google, "Ubicaciones de cloud." https://cloud.google.com/about/locations. (Visitado 17/06/2023).

[85] Google, "Recursos globales, regionales y zonales." https://cloud.google.com/compute/docs/regions-zones/global-regional-zonal-resources. (Visitado 11/06/2023).

[86] Amazon, "Configure route tables." https://docs.aws.amazon.com/vpc/latest/userguide/VPC_Route_Tables.html. (Visitado 11/06/2023).

[87] Microsoft, "Virtual network traffic routing." https://learn.microsoft.com/en-us/azure/virtual-network/virtual-networks-udr-overview. (Visitado 11/06/2023).

[88] Amazon, "Control traffic to subnets using network acls." https://docs.aws.amazon.com/vpc/latest/userguide/vpc-network-acls.html. (Visitado 11/06/2023).

[89] J. Barr, "Multi-region latency based routing now available for aws," *AWS News Blog*, 2012.

[90] B. Pal, S. Gorczynski, , and D. Schmidt, "Overview of data transfer costs for common architectures," *AWS Architecture Blog*, 2021.

[91] Google, "Virtual private cloud pricing." https://cloud.google.com/vpc/pricing. (Visitado 17/06/2023).

[92] Microsoft, "Bandwidth pricing." https://azure.microsoft.com/en-us/pricing/details/bandwidth. (Visitado 16/06/2023).

[93] Amazon, "Precios de las instancias bajo demanda de amazon ec2 - transferencia de datos." https://aws.amazon.com/es/ec2/pricing/on-demand/#Data_Transfer. (Visitado 16/06/2023).

[94] Amazon, "Base de datos." https://docs.aws.amazon.com/es_es/whitepapers/latest/aws-overview/database.html. (Visitado 17/06/2023).

[95] Microsft, "¿qué es azure sql?." https://learn.microsoft.com/es-es/azure/azure-sql/azure-sql-iaas-vs-paas-what-is-overview. (Visitado 17/06/2023).

[96] Microsft, "Tecnologías de base de datos relacionales en azure y aws." https://learn.microsoft.com/es-es/azure/architecture/aws-professional/databases. (Visitado 17/06/2023).

[97] Microsft, “Overview of oracle applications and solutions on azure.” https://learn.microsoft.com/en-us/azure/virtual-machines/workloads/oracle/oracle-overview. (Visitado 17/06/2023).

[98] P. Vergadia, “Your google cloud database options, explained,” *Google Cloud Blog*, 2021. (Visitado 17/06/2023).

[99] Microsoft, “Compute and storage options in azure database for postgresql - flexible server.” https://learn.microsoft.com/en-us/azure/postgresql/flexible-server/concepts-compute-storage. (Visitado 17/06/2023).

[100] Google, “Configurar los discos para cumplir con los requisitos de rendimiento.” https://cloud.google.com/compute/docs/disks/performance. (Visitado 17/06/2023).

[101] M. Yap and N. Gupta, “Scaling your amazon rds instance vertically and horizontally,” *AWS Database Blog*, 2016.

[102] Microsoft, “Scale operations in flexible server.” "https://learn.microsoft.com/en-us/azure/postgresql/flexible-server/how-to-scale-compute-storage-portal". (Visitado 17/06/2023).

[103] Amazon, “Acuerdo de nivel de servicios de amazon rds.” "https://d1.awsstatic.com/legal/amazonrdsservice/Amazon_RDS_Service_Level_Agreement__2022-03-09_Spanish.pdf". (Visitado 17/06/2023).

[104] Microsoft, “Service level agreements (sla) for online services.” "https://www.microsoft.com/licensing/docs/view/Service-Level-Agreements-SLA-for-Online-Services". (Visitado 17/06/2023).

[105] Google, “Cloud sql service level agreement (sla).” "https://cloud.google.com/sql/sla". (Visitado 17/06/2023).

[106] Amazon, “Resiliencia en amazon rds.” "https://aws.amazon.com/es/blogs/database/implementing-a-disaster-recovery-strategy-with-amazon-rds". (Visitado 17/06/2023).

[107] Amazon, “Overview of business continuity with azure database for postgresql - flexible server.” "https://learn.microsoft.com/en-us/azure/postgresql/flexible-server/concepts-business-continuity". (Visitado 17/06/2023).

[108] Google, “Acerca de las copias de seguridad de cloud sql.” "https://cloud.google.com/sql/docs/postgres/backup-recovery/backups". (Visitado 17/06/2023).

[109] Amazon, “Precios de amazon rds para postgresql.” https://aws.amazon.com/es/rds/postgresql/pricing. (Visitado 17/06/2023).

[110] Microsoft, “Precios de azure database for postgresql.” https://azure.microsoft.com/es-es/pricing/details/postgresql/flexible-server. (Visitado 17/06/2023).

[111] Google, “Precios de azure database for postgresql.” https://cloud.google.com/sql/pricing. (Visitado 17/06/2023).

[112] Microsoft, “Choose an azure compute service.” https://learn.microsoft.com/en-us/azure/architecture/guide/technology-choices/compute-decision-tree. (Visitado 12/06/2023).

[113] P. Vergadia and B. Dorsey, “Where should i run my stuff? choosing a google cloud compute option,” *Google Cloud Blog*, 2021.

[114] Google, “Modelo de recursos.” https://cloud.google.com/run/docs/resource-model. (Visitado 14/06/2023).

[115] Amazon, “Target tracking scaling policies.” https://docs.aws.amazon.com/AmazonECS/latest/developerguide/service-autoscaling-targettracking.html. (Visitado 14/06/2023).

[116] Azure, “Set scaling rules in azure container apps.” https://learn.microsoft.com/en-us/azure/container-apps/scale-app. (Visitado 14/06/2023).

[117] Google, “Información acerca del ajuste de escala automático de la instancia del contenedor.” https://cloud.google.com/run/docs/about-instance-autoscaling. (Visitado 14/06/2023).

[118] Amazon, “Amazon ecs on aws fargate.” https://docs.aws.amazon.com/AmazonECS/latest/developerguide/AWS_Fargate.html. (Visitado 14/06/2023).

[119] Google, “Límites de cpu.” https://cloud.google.com/run/docs/configuring/cpu. (Visitado 14/06/2023).

[120] Microsoft, “Quotas for azure container apps.” https://learn.microsoft.com/en-us/azure/container-apps/quotas. (Visitado 14/06/2023).

[121] Microsoft, “Azure container registry service tiers.” https://learn.microsoft.com/en-us/azure/container-registry/container-registry-skus. (Visitado 14/06/2023).

[122] Amazon, “Precios de amazon elastic container registry.” "https://aws.amazon.com/es/ecr/pricing". (Visitado 17/06/2023).

[123] Google, “Precios de artifact registry.” "https://cloud.google.com/artifact-registry/pricings". (Visitado 17/06/2023).

[124] Microsoft, “Precios de registro de contenedor.” "https://azure.microsoft.com/es-es/pricing/details/container-registry". (Visitado 17/06/2023).

[125] Amazon, “Precios de aws fargate.” https://aws.amazon.com/es/fargate/pricing. (Visitado 14/06/2023).

[126] Amazon, “Azure container apps pricing.” https://azure.microsoft.com/en-us/pricing/details/container-apps. (Visitado 14/06/2023).

[127] Google, “Precios de cloud run.” https://cloud.google.com/run/pricing. (Visitado 14/06/2023).

Este documento fue editado y tipografiado con LaTeX empleando
la clase **esi-tfg** (versión 0.20191022) que se puede encontrar en:
`https://github.com/UCLM-ESI/esi-tfg`