



viu

**Universidad
Internacional
de Valencia**

Infraestructura mediante Código vía Terraform en Amazon Web Services

Titulación:
Grado en Ingeniería
Informática

Curso académico
2024-2025

Alumno/a:Gagliardo, Miguel Angel
Pasaporte [REDACTED]
Director/a de TFG: Ángela Di Serio

Convocatoria: Primera



Agradecimientos

Quisiera agradecerles a los docentes de la VIU que se esfuerzan por forjar alumnos y elevan la vara de la educación en España y el mundo.

Vaya esta mención y dedicatoria en especial a aquellos docentes de quienes me llevo incontables aprendizajes esta dedicatoria especial, pero sobre todo a los que se toman la molestia de mantener los programas actualizados acorde a las tecnologías actuales y los requerimientos de un mercado laboral de IT que hoy en día es pujante y competitivo, tal vez como nunca lo fue antes.

Resumen

En el contexto actual de los entornos de desarrollo modernos, la gestión manual de infraestructura en la nube ha demostrado ser ineficiente, lenta y propensa a errores, lo que puede generar retrasos y vulnerabilidades en el proceso de entrega de software.

Esta problemática ha sido abordada por la Infraestructura como Código (IaC), que permite la automatización de la creación y gestión de infraestructura de manera replicable, segura y controlada.

En este Trabajo de Fin de Grado se demostrará cómo, mediante el uso de herramientas como Terraform y Amazon Web Services (AWS), se puede mejorar el proceso de despliegue, haciendo que la infraestructura sea fácilmente replicable, segura y auditável, un aspecto clave en entornos de trabajo colaborativos y modernos.

Palabras clave: Infrastructure as Code, IaC, Infraestructura como Código, Amazon Web Services, AWS, Cloud Computing, Terraform, Automatización, Despliegue, Infraestructura.

Abstract

In the current context of modern development environments, manual management of cloud infrastructure has proven to be inefficient, slow, and error-prone, leading to potential delays and vulnerabilities in the software delivery process.

This issue has been addressed by Infrastructure as Code (IaC), which enables the automation of infrastructure creation and management in a replicable, secure, and controlled manner.

This Final Degree Project will demonstrate how, using tools such as Terraform and Amazon Web Services (AWS), this deployment process can be improved by making infrastructure easily replicable, secure, and auditable - a key aspect in modern and collaborative work environments.

Keywords: Infrastructure as Code, IaC, Amazon Web Services, AWS, Cloud Computing, terraform, Automation, Deployment, Infrastructure.

Tabla de contenido

1. Introducción.....	9
Antecedentes.....	9
Tendencias en Infraestructura como Código	9
Planteamiento del Problema.....	10
Justificación	11
Objetivos.....	12
2. Marco Teórico.....	13
Computación en la Nube.....	13
Paradigmas de Computación en la Nube	14
Amazon Web Services (AWS)	15
Terraform e Infraestructura como Código	17
Contenedores.....	20
3. Metodología.....	22
Project Management.....	22
Herramientas.....	23
Limitaciones	24
4. Desarrollo	25
Trabajo en Sprints.....	25
Desarrollo de la Aplicación “Microblog”	26
Modelado de Datos.....	27
Diseño de la Infraestructura.....	28
Desarrollo de la Infraestructura como Código	30
Beneficios del Enfoque Iterativo con Terraform	30
Gestión de Credenciales para el Desarrollo Local.....	30
Configuración de Proveedores en Terraform.....	32
Gestión de dependencias en Terraform	33
Terraform y el manejo de estado	34
Gestión de credenciales a nivel aplicación	35
5. Resultados	39
Implementación Exitosa de la Infraestructura.....	39
Seguridad y Buenas Prácticas	42
Escalabilidad y Mantenibilidad	43
6. Conclusiones	44
Impacto en el Desarrollo y Despliegue de la Infraestructura.....	44
Automatización y Eficiencia Operativa	44
Consistencia y Reducción de Errores.....	44
Escalabilidad y Mantenimiento Simplificado.....	44
Áreas de Mejora y Trabajo Futuro	45
Reflexión Final.....	45
7. Referencias	47

Índice de Figuras

Figura 1. Crecimiento del Mercado de IaC en Entornos On-Premise vs Cloud. Fuente: https://market.us/report/infrastructure-as-code-market/	10
Figura 2. Ambientes de desarrollo de software en entornos ágiles. Fuente: https://docs.cloudbees.com/docs/cloudbees-cd/latest/plan/server-topology	11
Figura 3. Servicios de Computación en la Nube. Fuente: https://www.future-processing.com/blog/how-to-implement-cloud-computing/	13
Figura 4. Infraestructura global de Amazon Web Services. Fuente: https://aws.amazon.com/es/about-aws/global-infrastructure/	15
Figura 5. Datos de Datos de Cuota de Mercado de la Nube para Q1 2022. Fuente: https://www.theregister.com/2022/05/02/cloud_market_share_q1_2022/	16
Figura 6. Cuadrante de proveedores de servicios de infraestructura y plataforma en la nube. Fuente: https://aws.amazon.com/blogs/aws/aws-named-as-a-cloud-leader-for-the-10th-consecutive-year-in-gartners-infrastructure-platform-services-magic-quadrant/	17
Figura 7. Workflow de Terraform. Fuente: https://build5nines.com/terraform-workflow-process-explained/	18
Figura 8. Sprint Board en Github. Fuente: https://github.com/users/mgagliardo/projects/1	22
Figura 9. Repositorio del Trabajo de Final de Grado. Fuente: https://github.com/mgagliardo/viu-84giin-tfg/	23
Figura 10. Sprint 2 del Trabajo de Final de Grado. Fuente: https://github.com/users/mgagliardo/projects/1/views/2	25
Figura 11. Burn up Chart del Sprint del Trabajo de Final de Grado. Fuente: https://github.com/users/mgagliardo/projects/1/insights	26
Figura 12. Diagrama de Clases de la aplicación Microblog. Fuente: Elaboración Propia	27
Figura 13. Diagrama Entidad-Relación de la DB para la aplicación Microblog. Fuente: Elaboración Propia	28
Figura 14. Código de Terraform para explicitar la cantidad requerida de Contenedores. Fuente: Elaboración Propia	29
Figura 15. Diseño de la Infraestructura en AWS. Fuente: Elaboración Propia	29
Figura 16. Esquema de Autenticación para Terraform en AWS. Fuente: Elaboración Propia	31
Figura 17. Autenticación contra AWS utilizando roles de IAM. Fuente: Elaboración Propia	31
Figura 18. Declaración de un proveedor AWS en Terraform. Fuente: Elaboración Propia	32
Figura 19. Dependencias implícitas en Terraform. Fuente: Elaboración Propia	33
Figura 20. Dependencias explícitas en Terraform. Fuente: Elaboración Propia	33
Figura 21. Uso del remote tfstate file en Terraform. Fuente: Elaboración Propia	34
Figura 22. Gestión de credenciales en la aplicación. Fuente: Elaboración Propia	35
Figura 23. Generación aleatoria de la contraseña de la DB en Terraform. Fuente: Elaboración Propia	36

Figura 24. Guardado de secretos en Secrets Manager via Terraform. Fuente: Elaboración Propia	36
Figura 25. Creación de IAM role y asignación de permisos en Terraform. Fuente: Elaboración Propia	37
<i>Figura 26. Asignación de Variables de Entorno y referencia de Secretos en Terraform. Fuente: Elaboración Propia</i>	38
Figura 27. Output del plan de Terraform. Fuente: Elaboración Propia.....	39
Figura 28. Árbol de Dependencias de Outputs en Terraform. Fuente: Elaboración Propia	40
Figura 29. Tiempos de Creación de la Infraestructura via Terraform. Fuente: Elaboración Propia	40
Figura 30. URL del sitio web desplegado desde un navegador web. Fuente: Elaboración Propia	41
Figura 31. Tiempo de Destrucción de la Infraestructura via Terraform. Fuente: Elaboración Propia	41

Índice de Tablas

Tabla 1. Comparación entre Paradigmas de Computación en la Nube. Elaboración Propia.....	14
Tabla 2. Comparación entre Terraform y herramientas de IaC. Elaboración propia....	19
Tabla 3. Comparación entre Máquinas Virtuales y Contenedores. Elaboración propia.	20
Tabla 4. Comparación entre despliegues manuales y automatizados. Elaboración propia.....	42

1. Introducción

Antecedentes

Durante las últimas dos décadas, la gestión de infraestructura ha experimentado una transformación significativa, pasando de entornos e infraestructura física a entornos virtualizados y, más recientemente, a entornos completamente alojados en la nube. Esta evolución ha acompañado el proceso de escalado de aplicaciones de forma ágil y segura, reduciendo el costo operativo y facilitando el mantenimiento.

Dado el auge del desarrollo ágil y la necesidad de despliegues continuos, la gestión manual de infraestructura en la nube eventualmente también se volvió un cuello de botella. Ante esta problemática nace el concepto de infraestructura como código (Morris, 2025), que permite definir y gestionar entornos de infraestructura de manera automatizada, replicable y auditável, minimizando errores humanos y mejorando la eficiencia en los procesos de despliegue.

Así, de manera análoga a lo que ocurre en el desarrollo de software en aplicaciones y servicios, la infraestructura como código nos permite definir de manera declarativa el estado de la infraestructura que queremos alcanzar y acelerando, junto con los entornos cloud, los tiempos de delivery y agilizando el proceso de desarrollo de productos.

Tendencias en Infraestructura como Código

Inicialmente, los proveedores de computación en la nube desarrollaron soluciones nativas como **AWS CloudFormation** (Amazon Web Services, 2023) y **Azure Resource Manager**, que permiten definir infraestructura usando plantillas JSON o YAML. Más adelante, herramientas agnósticas como **Terraform** ganaron popularidad por su capacidad de trabajar con múltiples proveedores de manera agnóstica, usando un lenguaje declarativo común (DSL). Otras alternativas como **Pulumi** y **AWS CDK** surgieron posteriormente, permitiendo definir infraestructura utilizando lenguajes de programación tradicionales como Python o TypeScript.

La adopción de Infraestructura como Código muestra una clara tendencia de crecimiento en el mercado IT, tanto en soluciones on-premise como basadas en la nube, proyectando un significativo aumento en soluciones basadas en la nube, como se ve en la Figura 1.

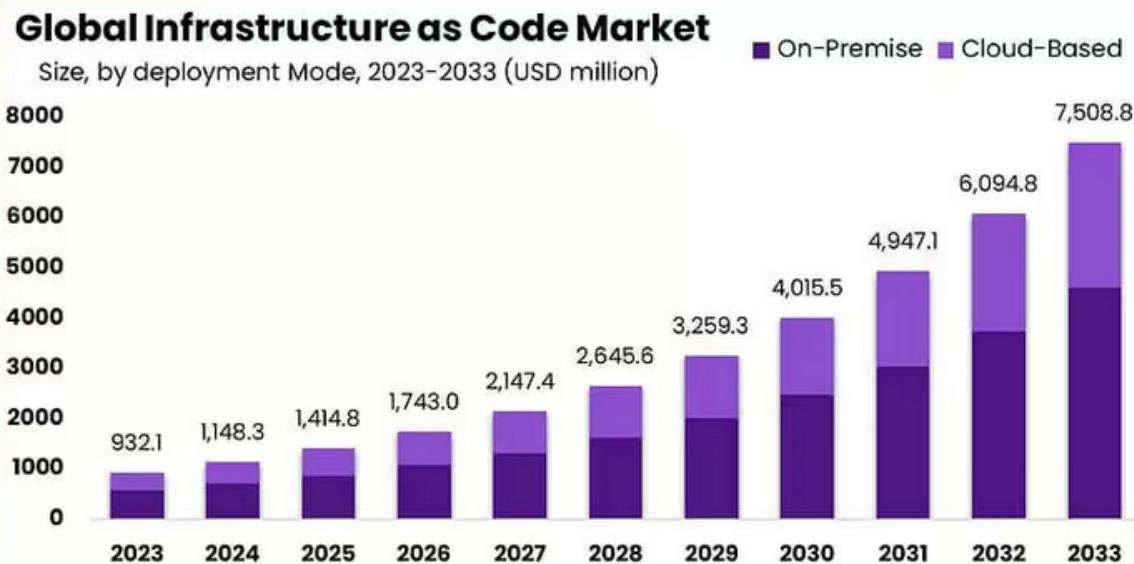


Figura 1. Crecimiento del Mercado de IaC en Entornos On-Premise vs Cloud.

Fuente: <https://market.us/report/infrastructure-as-code-market/>

Las tendencias en Infraestructura como Código reflejan la evolución continua del campo. La integración con **inteligencia artificial** está transformando la manera en que se gestiona la infraestructura, con herramientas como **GitHub Copilot** o **Cursor**, que asisten en la escritura de código de infraestructura y sistemas de ML que optimizan automáticamente configuraciones de recursos. La seguridad automatizada se ha vuelto crucial con soluciones como **Checkov** o **Snyk**, que analizan continuamente el código de infraestructura en busca de vulnerabilidades. Además, el surgimiento de **plataformas low-code y no-code** como **las últimas actualizaciones en Cloudformation** o **Pulumi** está democratizando el acceso a la IaC, permitiendo a equipos con menos experiencia técnica gestionar infraestructura compleja **en la nube**.

Planteamiento del Problema

En el desarrollo de software en entornos ágiles, existe una gran necesidad de contar con ambientes “replicables” para desarrollo, testing/UAT y producción. Metodologías como Continuous Delivery, como se ve en la Figura 2, requieren entornos similares para que, al ejecutar las fases de testing, se espere que todo funcione como se espera.

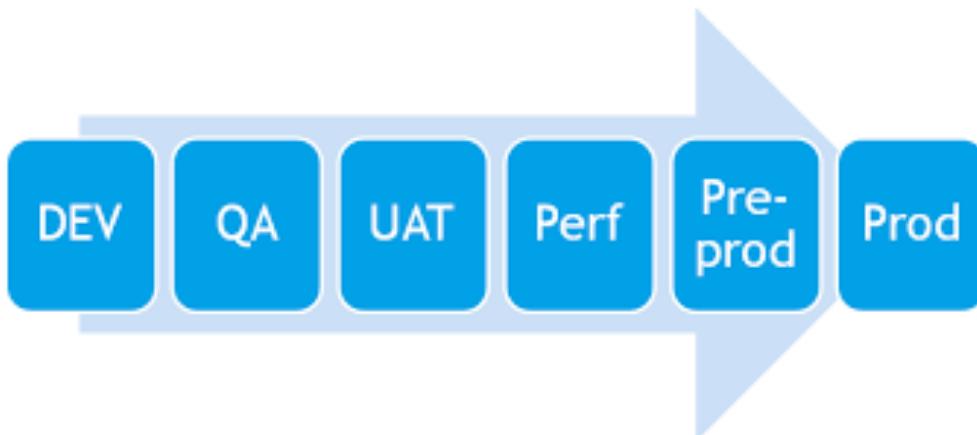


Figura 2. Ambientes de desarrollo de software en entornos ágiles.

Fuente: <https://docs.cloudbees.com/docs/cloudbees-cd/latest/plan/server-topology>

Sin embargo, la gestión manual de infraestructura sigue siendo común en muchos entornos, lo que introduce retrasos, riesgos operativos debido a errores humanos, falta de consistencia y, además, retrasa la entrega continua que debería acompañar el ritmo del desarrollo de software.

Aunque existen y se utilizan herramientas de Infraestructura como Código como Terraform, muchas veces no se integran con buenas prácticas, ni con una mentalidad orientada a escalar la arquitectura a futuro o mantenerla segura.

En este trabajo se propone una solución efectiva que, utilizando las mejores prácticas y herramientas correctamente, resuelva todas estas limitaciones.

Justificación

La gestión de infraestructura es una habilidad clave para ingenieros informáticos en el contexto actual de computación en la nube y entornos virtualizados. La gestión manual de infraestructura, aún común en muchas organizaciones, tiende a generar inconsistencias entre entornos y es propensa a errores humanos, lo que impacta directamente en la estabilidad y calidad del software desplegado. Esta problemática ha convertido la automatización de infraestructura en una competencia altamente demandada en el mercado laboral, siendo fundamental para la empleabilidad de futuros ingenieros informáticos.

Bajo esta premisa, la infraestructura como código ha demostrado ser un componente esencial en el ciclo de vida de desarrollo de aplicaciones, y su aplicación mejora tanto la calidad del software cuando se aplica correctamente y de manera segura, como los tiempos de delivery al acelerar los tiempos de despliegue.

Este trabajo contribuye con un enfoque práctico a demostrar cómo, utilizando las mejores prácticas en seguridad y automatización podemos automatizar y replicar entornos en AWS, usando Terraform.

Esto resulta fundamental en el ámbito del Grado de Ingeniería Informática y útil especialmente para ingenieros y empresas que requieren entornos consistentes de desarrollo, testing/UAT y producción, lo cual repercute directamente en la estabilidad y calidad del software desplegado.

Objetivos

Este trabajo tiene como objetivo general demostrar la utilidad de la automatización de infraestructura en la nube, utilizando Terraform como herramienta de Infraestructura como Código en Amazon Web Services (AWS) como entorno en la nube para crear entornos de desarrollo, prueba y producción replicables, seguros y auditables, capaces de soportar aplicaciones de manera eficiente y escalable.

Por el lado de los objetivos específicos:

- Explicar los principios fundamentales de la Infraestructura como Código y sus ventajas frente a la gestión manual de infraestructura.
- Diseñar e implementar una infraestructura en AWS utilizando Terraform para alojar una aplicación de ejemplo.
- Aplicar buenas prácticas de seguridad en la infraestructura desplegada, incluyendo la gestión de claves, configuración de redes segura e identidades de confianza.
- Automatizar el despliegue de recursos en la nube mediante Terraform, asegurando la replicabilidad y auditabilidad del proceso.
- Validar la infraestructura creada mediante pruebas de funcionamiento.

El marco teórico incluirá definiciones y una introducción a herramientas clave como **AWS, Terraform y Contenedores**. La **Metodología** detallará el desarrollo del entorno y la estrategia de trabajo implementada, mientras que la sección de **Desarrollo** explicará la implementación práctica de la aplicación y su infraestructura. Finalmente, las **últimas dos secciones** ofrecerán la evaluación de resultados y un análisis final.

2. Marco Teórico

Computación en la Nube

De acuerdo con la definición del **National Institute of Standards and Technology** (NIST, 2011) la computación en la nube o “la nube” es un modelo que permite el acceso a la red, ubicuo, práctico y bajo demanda a un conjunto compartido de recursos informáticos configurables como redes, servidores, almacenamiento, aplicaciones y servicios que pueden aprovisionarse y liberarse rápidamente con un esfuerzo de gestión o mínima interacción con el proveedor de dichos servicios.

Este modelo tiene cinco características esenciales:

- **Servicio bajo demanda**
- **Amplio acceso a la red**
- **Agrupación de Recursos**
- **Elasticidad Rápida**
- **Servicio Medido**

Por otro lado, el modelo cloud se compone de tres modelos de servicio:

- **Software as a Service (SaaS)**
- **Platform as a Service (PaaS)**
- **Infrastructure as a Service (IaaS)**

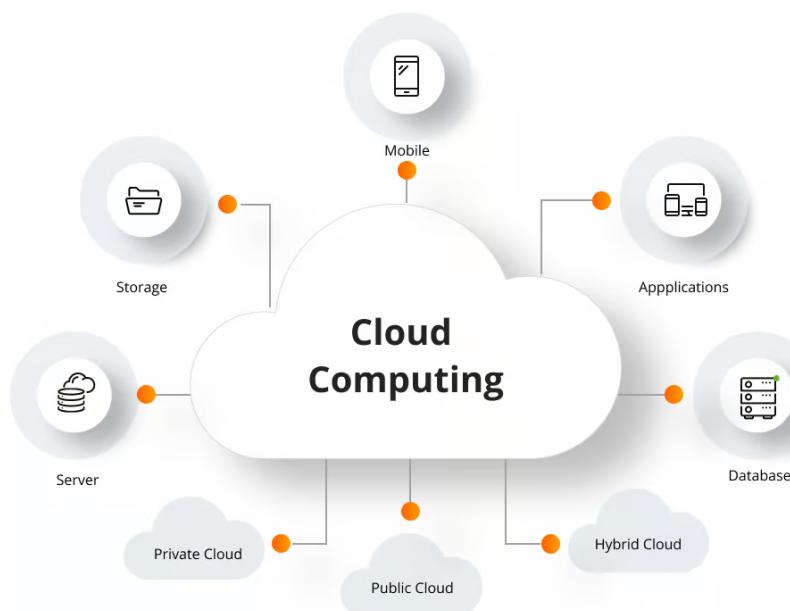


Figura 3. Servicios de Computación en la Nube.

Fuente: <https://www.future-processing.com/blog/how-to-implement-cloud-computing/>

Paradigmas de Computación en la Nube

La evolución de la computación en la nube ha dado lugar a diversos paradigmas que responden a diferentes necesidades empresariales y técnicas:

- **Cloud Nativa (Cloud Native):** Representa un enfoque para construir aplicaciones diseñadas específicamente para aprovechar las ventajas de la computación en la nube. Las aplicaciones cloud native, como en este proyecto, se caracterizan por ser contenerizadas, dinámicamente orquestadas y gestionadas mediante prácticas de DevOps (Wilsenach, 2015).
- **Cloud Híbrida (Hybrid Cloud):** Combina la infraestructura física (on-premise) con servicios en la nube pública, permitiendo a las organizaciones mantener sistemas críticos en sus propios centros de datos, principalmente por motivos de seguridad, mientras aprovechan la escalabilidad y flexibilidad de la nube.
- **Serverless Computing:** Representa un modelo donde el proveedor de la nube gestiona automáticamente la infraestructura subyacente, permitiendo a los desarrolladores centrarse exclusivamente en el código de la aplicación. En este proyecto, utilizamos parte de este paradigma a través de servicios como AWS Fargate y AWS RDS, los cuales nos abstraen de la gestión de la infraestructura necesaria para ejecutar e la aplicación.

La Tabla 1 representa una comparación entre estos tres enfoques principales.

Tabla 1. Comparación entre Paradigmas de Computación en la Nube. Elaboración Propia.

Características	Cloud Nativa	Cloud Híbrida	Serverless Computing
Infraestructura	Totalmente en la Nube	Combinación de On-Premise y Nube	Gestionada Automáticamente sobre el Proveedor
Diseño	Aplicaciones Contenerizadas	Sistemas Críticos On-Premise	Enfocado en el Código de Aplicación
Orquestación	Dinámica	Mixta	Automática
Gestión	Prácticas Devops	División entre Local y Nube	Mínima Gestión de Infraestructura
Escalabilidad	Alta	Media	Alta y Automática
Control	Total sobre la Infraestructura Cloud	Mixto entre Local y Nube	Limitado a Nivel Aplicación
Caso de Uso Típico	Aplicaciones Modernas	Organizaciones con Sistemas Legacy	Funciones y Microservicios

Estos paradigmas modernos han influido significativamente en el diseño la infraestructura para esta solución, permitiendo crear una arquitectura flexible, escalable y fácil de mantener, aprovechando las mejores características de cada modelo según las necesidades específicas del proyecto.

Amazon Web Services (AWS)

Tomando como base la definición antes mencionada, Amazon lanza en 2006 Amazon Web Services (Wittig & Wittig, 2003), una plataforma de servicios web que ofrece soluciones de cómputo, almacenado y redes bajo diferentes niveles de abstracción, en la cual es posible implementar todos los modelos de servicio antes mencionados (IaaS, PaaS, SaaS) para desplegar sitios web, aplicaciones corporativas o realizar minado de datos. El término “web services” en AWS hace referencia a que dichos servicios pueden ser controlados por una interfaz web, que puede ser utilizada por personas o sistemas mediante una interfaz de usuario.

Los servicios más conocidos de AWS son EC2 (Elastic Compute Cloud o Computo Elástico en la nube) que ofrece servidores virtuales y S3 (Simple Storage Service o Servicio de Almacenamiento Simple) que ofrece capacidad almacenamiento prácticamente ilimitada. Todos los servicios de AWS se ofrecen bajo un modelo de pago por uso, y cada servicio tiene un modelo de pago distinto, por ejemplo, EC2 se paga por cada minuto de uso, mientras que S3 se abona por GigaByte de almacenamiento al mes.

Amazon Web Services tiene múltiples centros de datos alrededor del mundo como se muestra en la Figura 4, distribuidos en América, Europa, Asia y Oceanía. Lo que nos da la posibilidad de desplegar servidores o almacenar datos en Japón de la misma manera que lo podemos hacer en Europa o América, y con la misma velocidad. Esto no solo muestra la consistencia y potencia del servicio si no también permite a las empresas y organizaciones desplegar infraestructura a nivel global en muy poco tiempo y con un esfuerzo mínimo.

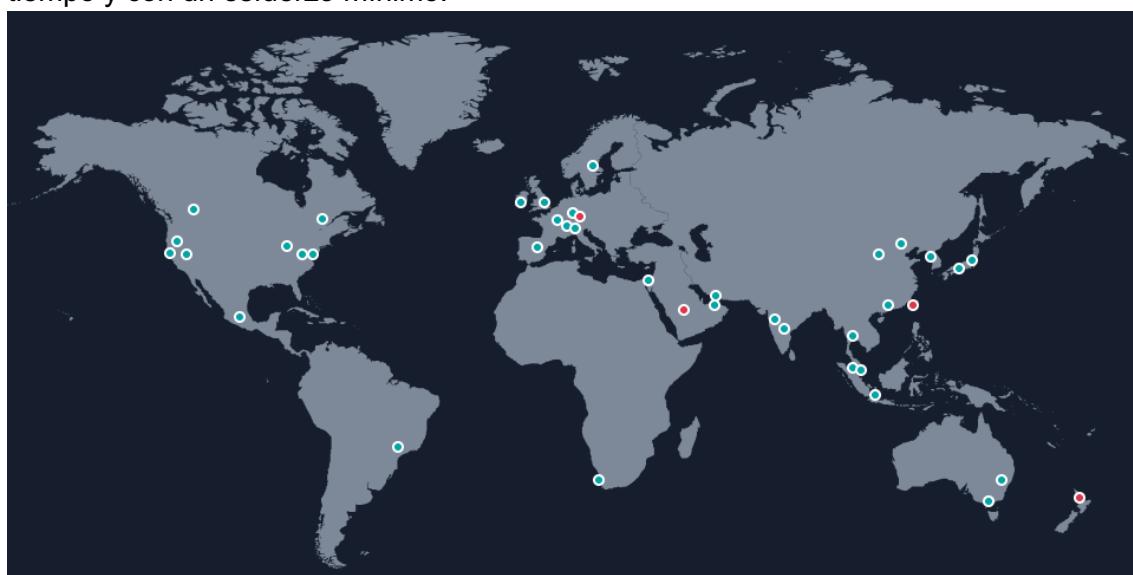


Figura 4. Infraestructura global de Amazon Web Services.
Fuente: <https://aws.amazon.com/es/about-aws/global-infrastructure/>

Por otro lado, AWS no solo suministra los servicios de computación en la nube antes mencionados (IaaS, PaaS, SaaS), si no que provee herramientas para administrar y desarrollar sobre sus servicios llamadas SDKs (Software Development Kit), estas permiten a los desarrolladores autenticarse y comunicarse con las APIs (Application Programmable Interfaces) públicas de Amazon Web Services e integrar sus propias aplicaciones con un abanico de lenguajes de programación tales como Java, Javascript, Python, Golang, entre tantos otros.

Por último, Amazon Web Services es no solo es la empresa líder en el mercado como se ve en la Figura 5, si no que es uno de los proveedores más antiguos, proveyendo de servicios de Cloud Computing. Esta madurez se traduce en estabilidad, extensa documentación y una comunidad activa tanto en foros propios de AWS y externas como StackOverflow o Reddit, lo cual facilita el aprendizaje y resolución de problemas.

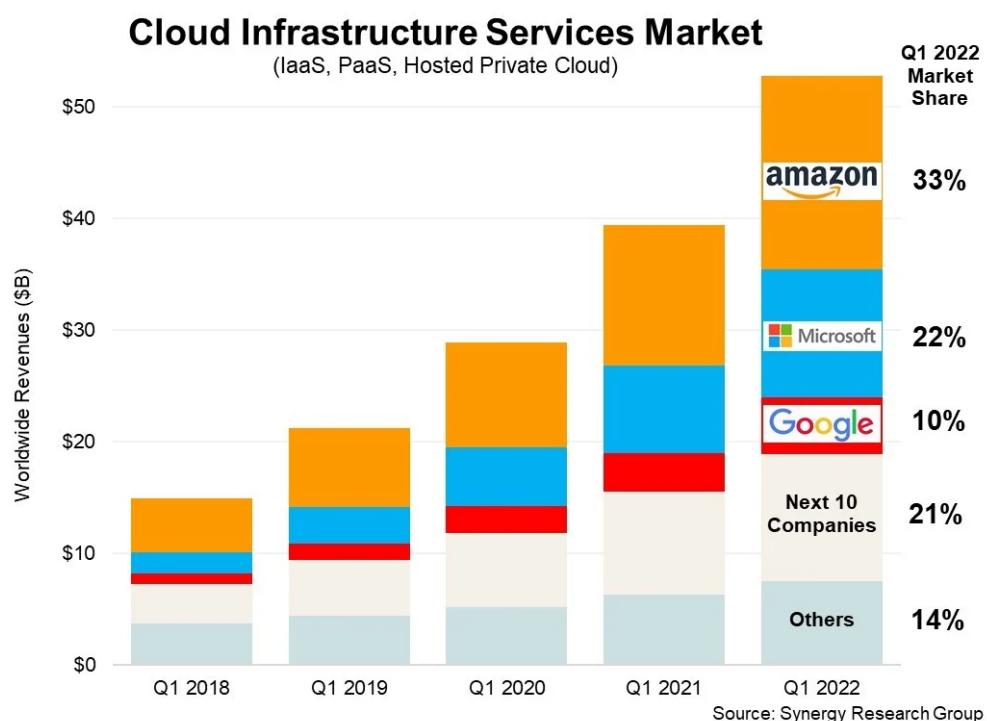


Figura 5. Datos de Cuota de Mercado de la Nube para Q1 2022.
Fuente: https://www.theregister.com/2022/05/02/cloud_market_share_q1_2022/

En este contexto, AWS se presenta no solo como un proveedor de infraestructura, sino también como una plataforma preparada para automatización avanzada. Esta característica lo convierte en un entorno ideal para aplicar herramientas de Infraestructura como Código, como Terraform.

La elección de Amazon Web Services (AWS) como entorno de trabajo para este proyecto se fundamenta en su posición como el proveedor de servicios en la nube más consolidado del mercado, con una participación líder a nivel global, como se ve en la Figura 6. Su infraestructura altamente distribuida y robusta garantiza disponibilidad, escalabilidad y seguridad, características esenciales para el desarrollo

de aplicaciones modernas. Además, AWS cuenta con una extensa documentación oficial y una comunidad activa, lo que facilita su aprendizaje y adopción.



Figura 6. Cuadrante de proveedores de servicios de infraestructura y plataforma en la nube.

Fuente: <https://aws.amazon.com/blogs/aws/aws-named-as-a-cloud-leader-for-the-10th-consecutive-year-in-gartners-infrastructure-platform-services-magic-quadrant/>

Terraform e Infraestructura como Código

La creciente digitalización aumenta la demanda de servicios en la nube, generando presión en los equipos técnicos para mantener la infraestructura. Esto suele llevarlos a adoptar un enfoque conservador para evitar inestabilidad.

Esta situación genera dos efectos:

- Se desaprovechan las ventajas de la nube: Elasticidad, aprovisionamiento rápido y el escalado automático.

- Los usuarios finales y áreas de negocio esperan beneficios como mayor velocidad y disponibilidad continua.

Para evitar que los equipos técnicos omitan los procesos de control, surge la **automatización en la nube**, que permite despliegues controlados y replicables mediante APIs y SDKs, facilitando el cumplimiento de los requisitos organizacionales de control de cambios y gestión de riesgos.

En este contexto es que surge la Infraestructura como Código (IaC), que se define como la gestión de infraestructura mediante código en lugar de herramientas manuales. Esto elimina inconsistencias y permite aplicar principios de desarrollo de software: versionado, pruebas, automatización y peer review.

La evolución de IaC comenzó con soluciones personalizadas, seguida por AWS CloudFormation (Amazon Web Services, 2023) usando JSON/YAML. En 2014, Mitchell Hashimoto creó Terraform (Hashimoto, 2021), una alternativa de código abierto que funcionaba con múltiples proveedores cloud, como se ve en la Figura 7.

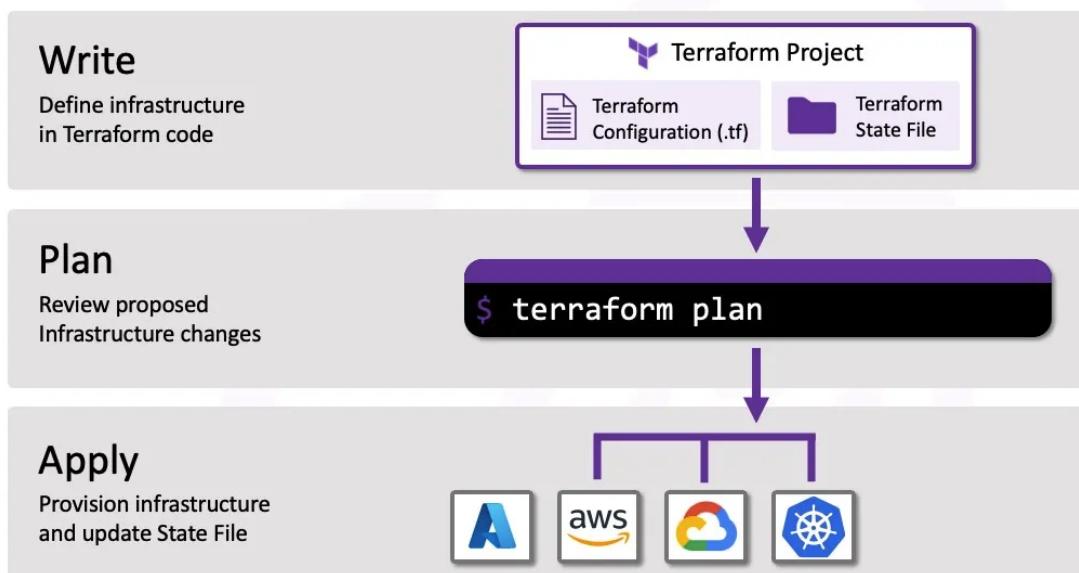


Figura 7. Workflow de Terraform.
Fuente: <https://build5nines.com/terraform-workflow-process-explained/>

Lo novedoso de Terraform no era solamente que fuese Código Abierto, si no que a diferencia de sus competidores posee:

- **Lenguaje propio:** HCL (Hashicorp Configuration Language), un DSL (Domain Specific Language) desarrollado por Hashicorp. A diferencia de herramientas específicas por proveedor, Terraform busca una **abstracción común** y una **sintaxis declarativa**, lo que facilita entender el “qué” de la infraestructura deseada, y no tanto el “cómo”.
- **Soporte multi-cloud y agnóstico:** Terraform nace con la intención de ser multi-cloud y agnóstico dado que no está limitado a AWS, sino que también permite gestionar infraestructura de más de tres mil proveedores (Hashicorp, 2023)

- **Componibilidad:** El usuario puede separar porciones de código y generar módulos agnósticos y reutilizables, análogos a las clases en un lenguaje de programación regular.
- **Pluggable:** Dado que es Open Source cualquier usuario puede crear su propio conector a un proveedor cloud que no esté aún soportado y agregarlo al ecosistema.
- **Amplia comunidad:** Terraform rápidamente construyó una enorme comunidad que no sólo compartía soluciones si no que comentaba y discutía y votaba nuevas características y participaba en la corrección de errores de la herramienta.
- **Estado:** Terraform introduce **tfstate** (Terraform State), un archivo en formato JSON que contiene el estado actual de la estructura desplegada en el proveedor, y que permite visualizar **de antemano** a través de un **plan** cual va a ser el estado alcanzado de la infraestructura.

La Tabla 2 compara las tres soluciones más populares en la actualidad. Esta evaluación detallada permite a los equipos de desarrollo e infraestructura seleccionar la herramienta más adecuada para sus necesidades específicas y entornos de trabajo.

Tabla 2. Comparación entre Terraform y herramientas de IaC. Elaboración propia.

Características	Terraform	Pulumi	AWS CDK
Soporte Multi-Cloud	✓	✓	✗
Manejo de Estado	✓	✓	✓
Licencia OpenSource	✓	✓	✗
No requiere código	✓	✗	✗
Gratis	✓	✗	✓
Extensible	✓	✗	✗
Lenguaje	HCL (DSL)	Multi-Lang	Multi-Lang
Curva de Aprendizaje	Baja	Alta	Alta

Por su flexibilidad, enfoque agnóstico y comunidad, Terraform se presenta como una herramienta idónea para explorar los principios de Infraestructura como Código. A lo largo de este trabajo se utilizará para definir, desplegar y gestionar recursos en la nube de AWS de forma reproducible y controlada, permitiendo así demostrar la viabilidad de este enfoque.

Contenedores

La utilización de contenedores fue fundamental para este proyecto, específicamente a la hora de empaquetar la aplicación **microblog**. Utilizando Docker (Kane & Matthias, 2023) se generó una imagen que incluye el código fuente de la aplicación Python junto con todas sus dependencias, garantizando así una ejecución consistente en cualquier entorno.

En cuanto a la **ejecución de los contenedores** se decidió utilizar **AWS Fargate**, un servicio de cómputo de tipo **serverless** que elimina la necesidad de aprovisionar y gestionar servidores, permitiendo **ejecutar contenedores sin preocuparse por la infraestructura subyacente**. Esto también permite aprovechar la escalabilidad horizontal añadiendo más instancias del contenedor según se necesite, mientras se mantiene una eficiencia de recursos superior al compartir el sistema operativo del host.

Para **almacenar las imágenes de contenedores** de manera segura se optó por **Amazon Elastic Container Registry (ECR)**, un **registro privado** donde se depositan las imágenes de los contenedores y desde donde AWS Fargate las recupera para su ejecución nuevamente, gestionando automáticamente la infraestructura subyacente.

La decisión de utilizar contenedores sobre máquinas virtuales tradicionales (Allison, 2020) se fundamentó en ventajas prácticas que se materializaron durante el desarrollo, como se puede visualizar en la Tabla 3. Los contenedores en **AWS Fargate** se inician en segundos comparado con el tiempo que requiere una máquina virtual tradicional, y el uso de la misma imagen del contenedor tanto en entorno local como en producción elimina problemas de inconsistencia en el desarrollo de la aplicación.

Tabla 3. Comparación entre Máquinas Virtuales y Contenedores. Elaboración propia.

Características	Máquinas Virtuales	Contenedores
Peso	Pesado	Ligero
Uso de recursos OS	Independiente	Compartido
Tiempo de Inicio	Lento	Rápido
Portabilidad	Baja	Alta
Uso de memoria	Alto	Bajo
Instancias por Host	10-100	100-1000
Seguridad	Alta	Moderada

Esta arquitectura basada en contenedores no solo facilitó el despliegue y la gestión de la aplicación, sino que también estableció una base sólida para futuras mejoras,

permitiendo actualizaciones sin tiempo de inactividad y una utilización óptima de recursos.

Finalmente, La integración con los servicios nativos de AWS demostró ser una decisión acertada, proporcionando una plataforma robusta y escalable para la aplicación.

3. Metodología

Project Management

Para este Trabajo de Fin de grado se implementó un marco de trabajo ágil basado en Scrum. Las razones para hacerlo son:

- Scrum permite **entregas incrementales y feedback continuo** a través de sprints de dos semanas.
- **Facilita la adaptación** a cambios durante el proyecto.
- **Proporciona transparencia** en el avance del proyecto, tanto para el alumno como para el tutor.

Estructura de los Sprints:

- **Duración:** Dos semanas por cada Sprint.
- **Planificación:** Tareas definidas como Issues en Github.
- **Seguimiento:** Reuniones periódicas con el tutor.
- **Métricas:** Burn Up charts nativos de Github.

La Figura 8 ilustra parte del Sprint Board utilizado para el seguimiento de tareas, donde cada Sprint está identificado con un color distintivo y las tareas están organizadas según su estado de progreso. Esta herramienta facilitó la visualización del avance del proyecto y permitió mantener un control efectivo sobre los diferentes componentes del trabajo.

The screenshot shows a GitHub interface for a 'Sprint Tracker'. On the left, there's a sidebar titled 'Labels' with the following items:

- Sprint 1 (Yellow circle)
- Sprint 2 (Light Green circle)
- Sprint 3 (Red circle)
- Sprint 4 (Blue circle)
- Sprint 5 (Teal circle)
- documentation (Blue circle)
- feature (Green circle)

Next to each label is a number indicating the count of tasks: 3, 6, 3, 1, 1, 2, and 3 respectively. Below the sidebar, there's a link 'Show empty values'.

The main area is titled 'Trabajo de Fin de Grado - Sprint Tracker' and shows two columns of sprints:

- [FINALIZADO] Sprint 1
- [FINALIZADO] Sprint 2 - 22/04/2025 - 05/05/2025

For Sprint 2, there is a search bar with the query '-status:"In Progress",Todo Backlog'. Below it, a list of tasks is shown, each with a checkmark icon and a user profile picture:

- Done (6) - This has been completed
 - viu-84giin-tfg #16 Checkpoint meeting #2 con tutor
 - viu-84giin-tfg #8 Obtener cuenta de AWS para TFG
 - viu-84giin-tfg #2 Dockerizar app
 - viu-84giin-tfg #1 Crear codigo app en python
 - viu-84giin-tfg #3 Crear codigo de terraform para deployment
 - viu-84giin-tfg #7 Dominio de la app. Route53 vs Cloudflare

Figura 8. Sprint Board en Github.
Fuente: <https://github.com/users/magliardo/projects/1>

Herramientas

En cuanto a las herramientas utilizadas:

- **Computador Personal:** Para desarrollo local, tanto de la aplicación como de la contenerización de la misma
- **Visual Studio Code:** IDE que se integra con la herramienta de IaC Terraform y el código de la app creada en Python a través de plugins de código abierto.
- **Github:** El standard de facto para crear repositorios y hostear documentación en Markdown. Provee integración nativa con proyectos y la posibilidad de generar Sprint Boards y Burn Up charts (Figuras 7, 8 y 9).
- **Terraform:** CLI nativa para la validación y aplicación de código generado utilizando el IDE.
- **Docker:** Para el empaquetamiento de la aplicación desarrollada en Python en un único contenedor.
- **Cuenta de Amazon Web Services:** Cuenta personal de AWS para la aplicación del código y validación de la infraestructura a desplegar sobre la misma.

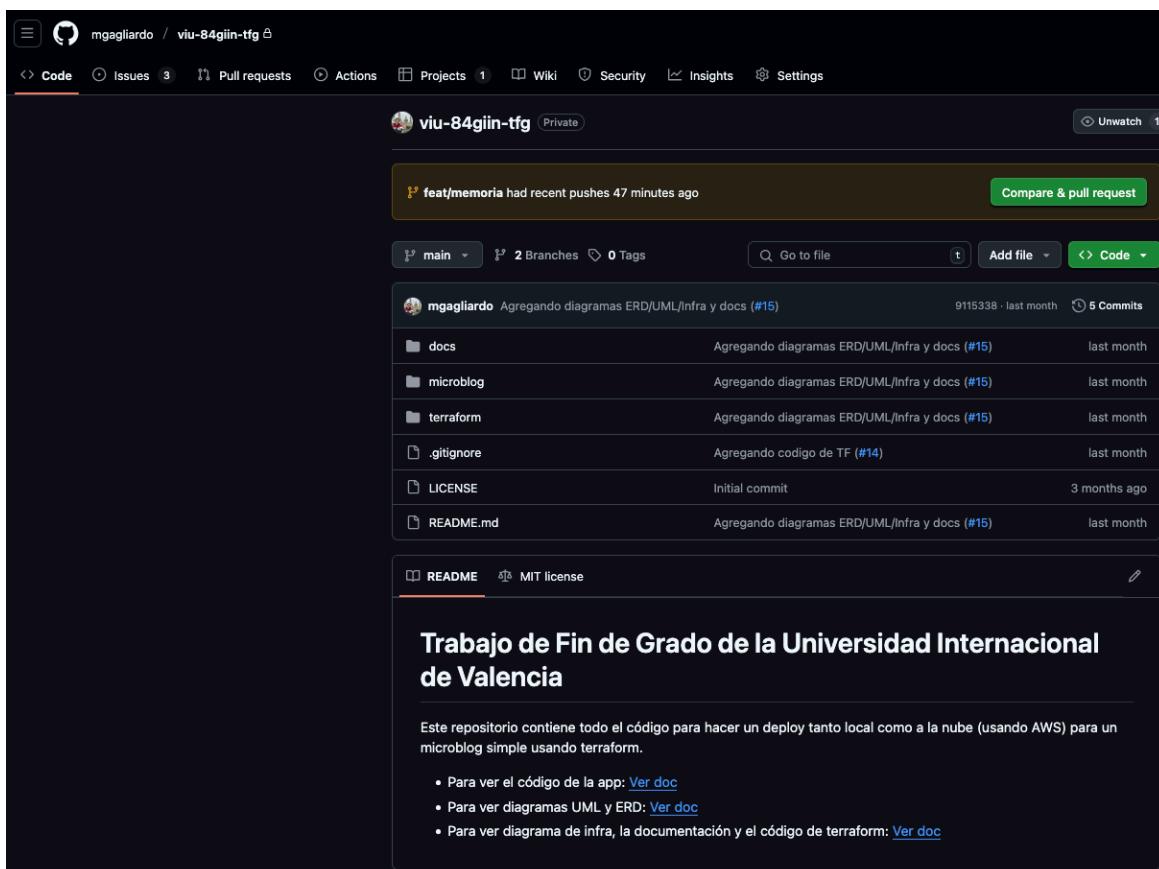


Figura 9. Repositorio del Trabajo de Final de Grado.
Fuente: <https://github.com/mgagliardo/viu-84giin-tfg/>

Limitaciones

Algunas de las limitaciones encontradas en el manejo del proyecto, fueron:

- El **desarrollo de la aplicación** microblog fue **deliberadamente simplificado** al no ser el foco principal de este trabajo, limitando la complejidad de casos de prueba para la infraestructura.
- La infraestructura mantiene un **número fijo de contenedores**, sin capacidad de auto-escalado, lo que limita la adaptabilidad en cambios en la demanda.
- No se implementaron **pruebas automatizadas de infraestructura**, lo cual limita la validación sistemática de los cambios realizados.
- La **ausencia de un pipeline de CI/CD** completo limita la automatización total del proceso de despliegue y validación.
- La no implementación de un **sistema formal de estimación** para los issues del sprint como por ejemplo puntuaciones como, por ejemplo cartas, Fibonacci u otro método.

Al ser un proyecto unipersonal, algunas prácticas de Scrum son lógicamente limitadas como por ejemplo el backlog grooming. Por esta misma razón, los sprints fueron lógicamente más flexibles que en un entorno profesional.

4. Desarrollo

Trabajo en Sprints

Para el desarrollo del proyecto como se ha mencionado anteriormente se ha optado por un modelo agile, ejecutando la totalidad del proyecto en 5 sprints de 2 semanas cada uno, como se puede visualizar en la Figura 10:

1. **Sprint 1 - 07/04/2025 - 21/04/2025:** Donde mayormente se diseñó la aplicación y se realizó el primer draft en el desarrollo esta.
2. **Sprint 2 - 22/04/2025 - 05/05/2025:** Utilizado para refinar la aplicación y dockerizarla, así como la creación de la cuenta de AWS donde va a ser hosteada y el desarrollo del código de Terraform.
3. **Sprint 3 - 06/05/2025 - 19/05/2025:** Se hicieron mejoras menores en el código de Terraform y se finalizaron todas las pruebas de este.
4. **Sprint 4 - 20/05/2025 - 03/06/2025:** Se trabajó mayormente en refinar y finalizar, así como preparar la ponencia para el Trabajo Final de Grado.
5. **Sprint 5 - 04/06/2025 – 18/06/2025:** Idem Sprint 4.

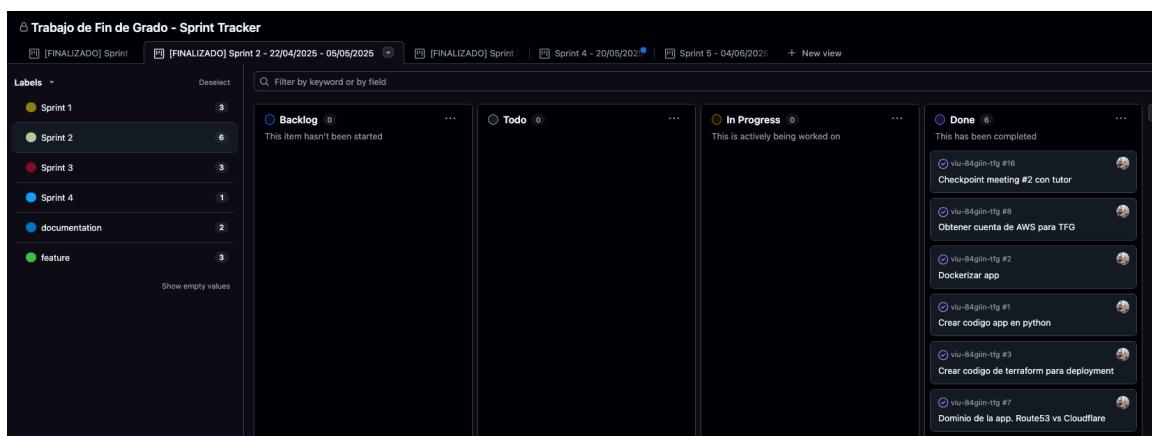


Figura 10. Sprint 2 del Trabajo de Final de Grado.
Fuente: <https://github.com/users/mgagliardo/projects/1/views/2>

La razón para utilizar sprints es principalmente la de poder enfocar el desarrollo hacia la iteración y el progreso acumulativo, donde el punto de partida de cada nuevo sprint era la finalización del anterior tratando de, en una suerte de retrospectiva, entender que se hizo bien, que se hizo mal y que cambios se debían hacer para continuar. Los checkpoints con el tutor fueron útiles también en este punto para entender hacia donde se debía enfocar el desarrollo de la memoria y la presentación final.

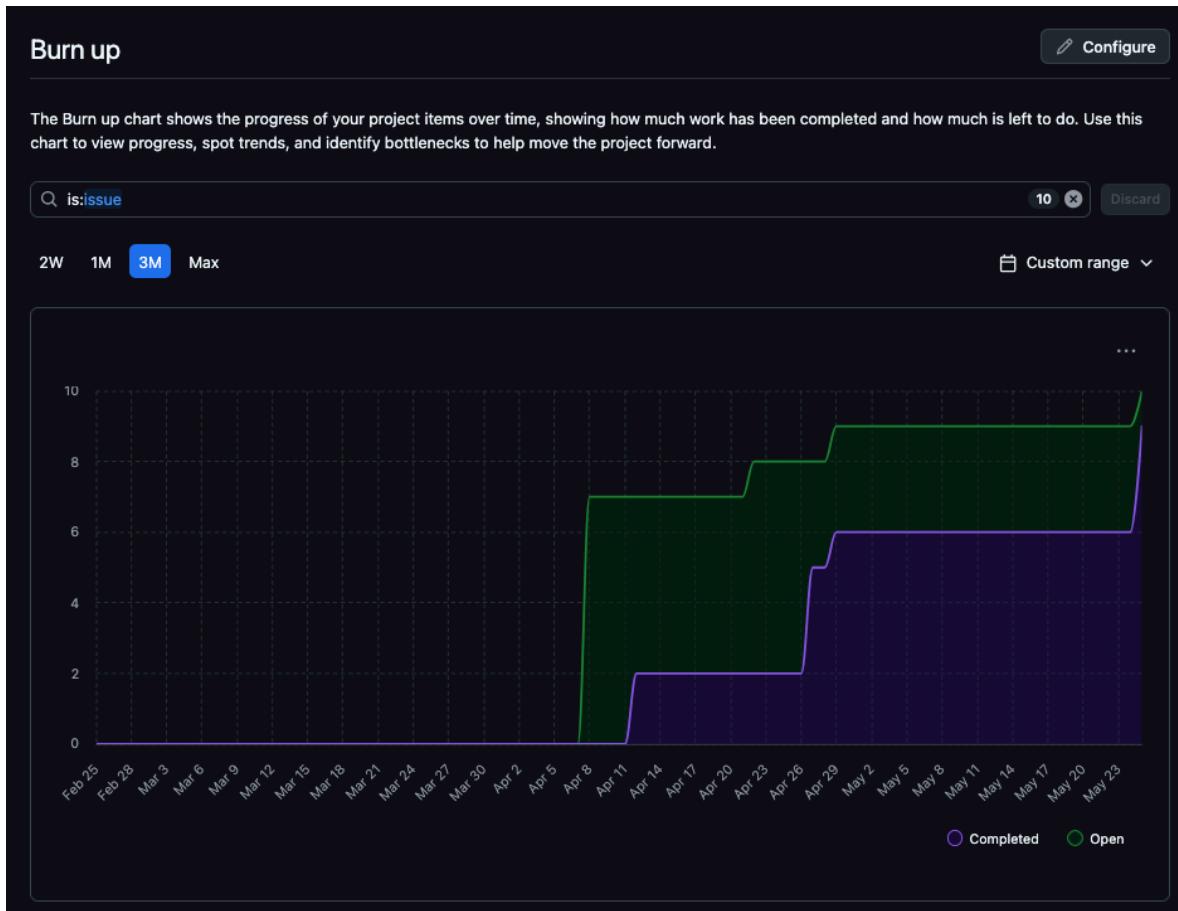


Figura 11. Burn up Chart del Sprint del Trabajo de Final de Grado.

Fuente: <https://github.com/users/mgagliardo/projects/1/insights>

Desarrollo de la Aplicación “Microblog”

Para las pruebas de Terraform en AWS, se desarrolló una aplicación sencilla de tipo microblog: Una plataforma donde cada usuario, dándose de alta utilizando un mail y contraseña, puede acceder, generar posteos y seguir a otros usuarios así como enviarles mensajes privados, de manera similar a Google Blogs o Twitter

Para el desarrollo se optó por una metodología de Rapid Development dado que no era el foco ni la prioridad de este trabajo. Este enfoque permitió realizar un prototipo en 2 sprints, buscando más el feedback y testear la respuesta de la aplicación para lograr un MVP lo más rápido posible.

Se utilizó Python con su popular **Web Framework Flask**, que posee un setup sencillo y una curva de aprendizaje baja. La Figura 11 muestra el diagrama de clases de la aplicación, generado utilizando **Mermaid** (Github, 2022), un framework basado en **MarkDown** que es un lenguaje de marcado para escribir documentos.

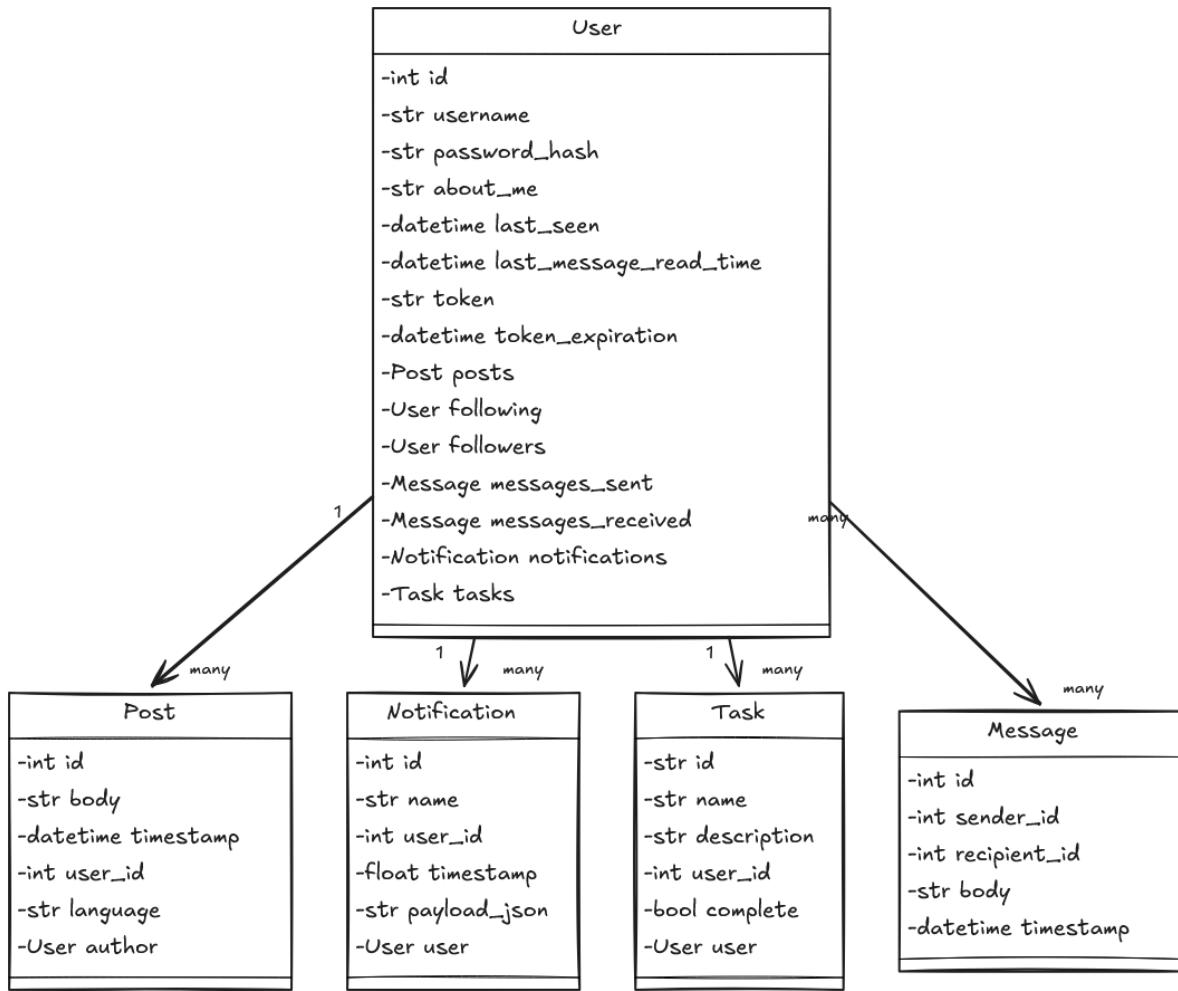


Figura 12. Diagrama de Clases de la aplicación Microblog. Fuente: Elaboración Propia

Modelado de Datos

Dado que los usuarios de la aplicación van a realizar las funcionalidades típicas de ABM (Alta, Baja y Modificación) de usuarios, posteos y mensajes, se optó por una base de datos de tipo relacional como PostgreSQL por ser escalable, extensible y ser probablemente la que posee la comunidad más robusta en la actualidad, superando a sus competidores directos (MySQL, MariaDB, SQL Server).

Python posee un ORM muy robusto llamado SQLAlchemy, que se conecta con el web framework Flask a través de una librería llamada Flask-SQLAlchemy y que permite de manera muy sencilla levantar una base de datos, requiriendo un nivel de diseño bajo dado que, nuevamente, no es el foco principal de este Trabajo de Fin de Grado.

Finalmente, para el diseño del diagrama Entidad-Relación se utilizó chartdb (ChartDB, 2025), una herramienta de diseño de diagramas de bases de datos, que se puede ver ilustrado en la Figura 13 a continuación.

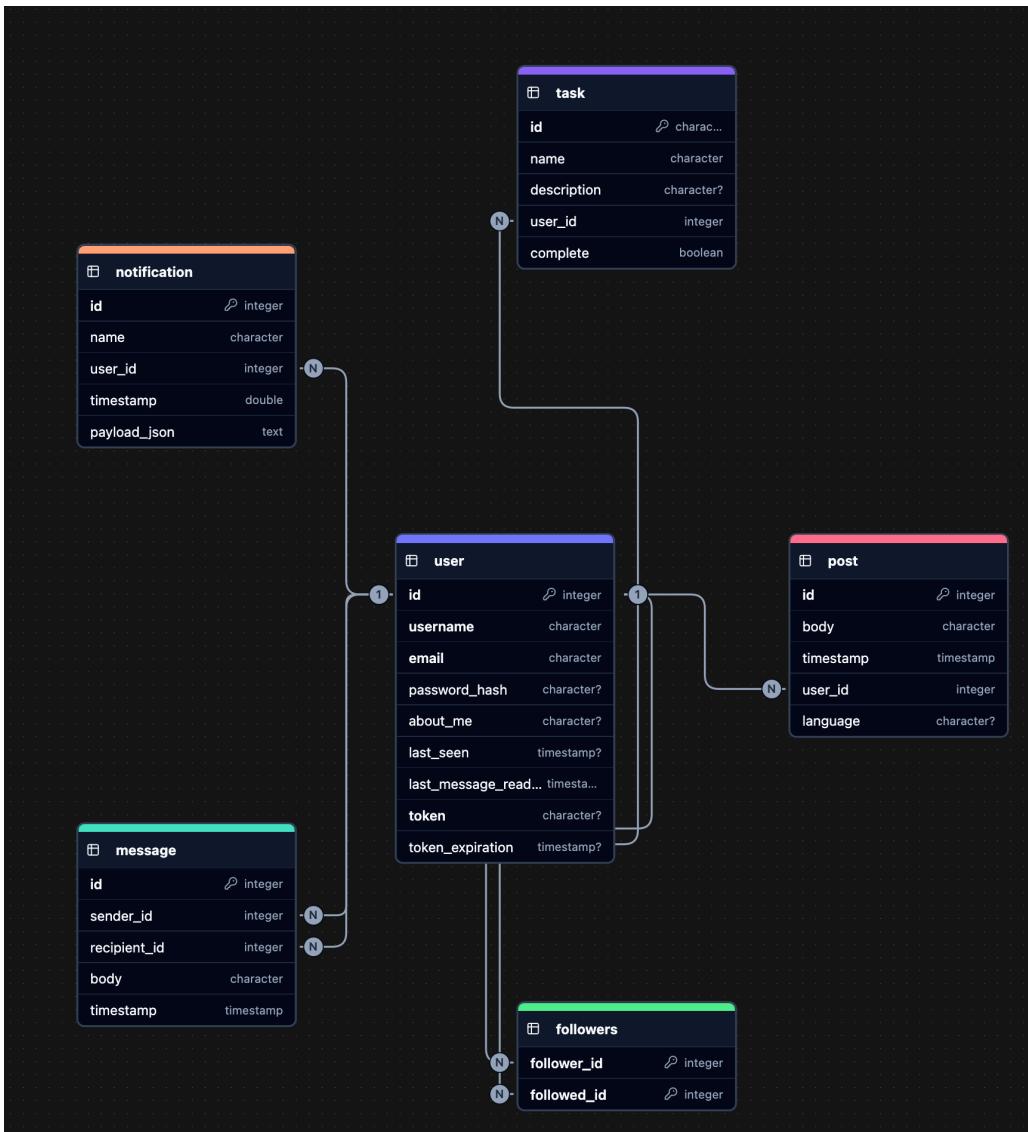


Figura 13. Diagrama Entidad-Relación de la DB para la aplicación Microblog. Fuente: Elaboración Propia

Diseño de la Infraestructura

Para el diseño de la infraestructura se optó por contenerizar la aplicación, de tal manera que no hiciera falta un setup adicional de dependencias dado el pre-empaquetado que esto supone, y también aprovechando que el servicio de backend no maneja estado. Adicionalmente, el paradigma de contenedores se ajusta perfectamente a los servicios que AWS ofrece. En este aspecto, se utilizó un registro de contenedores para guardar las imágenes como los mismos como es ECR (Elastic Container Repository), y para ejecutarlos un servicio on-demand como es Fargate, pudiendo lanzar cuantas copias de la aplicación como queramos con un cambio sencillo en el código de Terraform como se muestra en la Figura 14, y la carga del tráfico entrante será siempre balanceado gracias a que están conectadas a un平衡ador de carga (ALB).

```

resource "aws_ecs_service" "microblog" {
  name          = "${var.environment}-${var.service}-service"
  cluster       = aws_ecs_cluster.microblog.id
  task_definition = aws_ecs_task_definition.microblog.arn
  desired_count    = 2
  force_new_deployment = true
  launch_type     = "FARGATE"
}

```

Cantidad solicitada de contenedores a ejecutar

Figura 14. Código de Terraform para explicitar la cantidad requerida de Contenedores.
Fuente: Elaboración Propia

Para la conexión con la base de datos PostgreSQL, se ha optado por Amazon RDS (Relational Database Service), un servicio escalable que facilita la gestión y mantenimiento de bases de datos en la nube. Las credenciales se almacenan de manera segura y se inyectan dinámicamente al contenedor en tiempo de ejecución mediante AWS Secrets Manager, utilizando variables de entorno para evitar su exposición en el código o configuración estática.

Asimismo, la infraestructura sigue un modelo de acceso a red basado en el principio de mínimo privilegio (Amazon Web Services, 2025), asegurando un aislamiento adecuado de los servicios:

- El Balanceador de Carga se despliega en subredes públicas, actuando como el único componente con acceso directo a Internet y sirviendo como punto de entrada para los usuarios.
- Los contenedores en AWS Fargate y la base de datos en RDS se alojan en subredes privadas, sin conectividad externa directa, dado que no requieren acceso a Internet para su funcionamiento.

Esta arquitectura de red segmentada reduce la superficie de ataque y fortalece la postura de seguridad general, garantizando que sólo los servicios esenciales puedan interactuar con el entorno público de manera controlada, como se ve en la Figura 15.

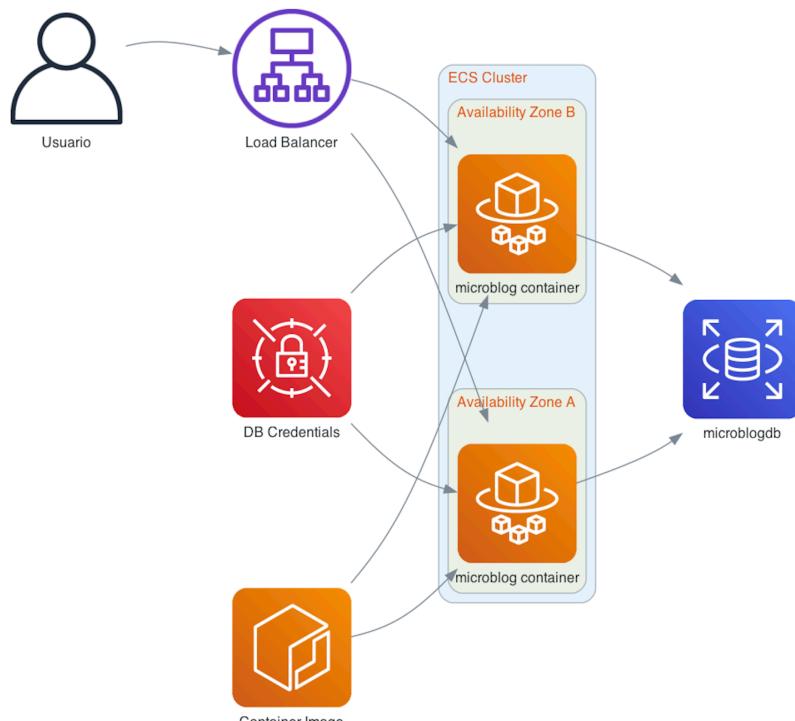


Figura 15. Diseño de la Infraestructura en AWS. Fuente: Elaboración Propia

Desarrollo de la Infraestructura como Código

Asimismo, la infraestructura sigue un modelo de acceso a red basado en el principio de mínimo privilegio, asegurando un aislamiento adecuado de los servicios:

El desarrollo de infraestructura como código utilizando Terraform en AWS se presta perfectamente a un enfoque iterativo y ágil como Rapid Development, de la misma manera que se hizo con la aplicación microblog. Al haber estructurado el desarrollo en sprints, se pudo construir el código infraestructura de manera incremental, alineándose con prácticas DevOps modernas:

- Se **inició** con una prueba de concepto que contenía la configuración básica de la red en AWS, definiendo una VPC, subredes (subnets) y grupos de seguridad (security groups) necesarios.
- Las **subsiguientes iteraciones** se centraron en añadir capas adicionales de infraestructura como los contenedores en Fargate, el balanceador de carga, la base de datos PostgreSQL en RDS y las credenciales en SecretsManager.
- **Cada tarea y sprint** permitieron la validación y prueba de los componentes desplegados, facilitando la detección temprana de problemas y la optimización continua.

Beneficios del Enfoque Iterativo con Terraform

Terraform ofrece funcionalidades clave que optimizan este proceso, como:

- Planificación de cambios con **terraform plan**, permitiendo visualizar modificaciones antes de aplicarlas.
- Aplicación incremental con **terraform apply**, asegurando una implementación controlada y reproducible.
- **Modularización del código**, lo que en proyectos colaborativos permite que distintos equipos trabajen en paralelo en diversos componentes, acelerando el desarrollo y mejorando la mantenibilidad a largo plazo.

Aunque en este caso se trató de un trabajo unipersonal, este enfoque modular es clave en entornos empresariales, donde la gestión eficiente de infraestructura es esencial para la escalabilidad y estabilidad del sistema.

Gestión de Credenciales para el Desarrollo Local

Una práctica común y necesaria al trabajar con Terraform de manera local es la configuración de credenciales de AWS específicas para la máquina de desarrollo. Esto se hace posible mediante dos herramientas:

1. La **configuración inicial del perfil de usuario local para AWS** implica introducir un par de claves (de acceso y secreta) a través de la línea de

comandos una única vez, utilizando la CLI proporcionada por AWS. Estas claves se guardarán en un archivo de configuración para su posterior gestión.

2. Configuradas las **credenciales**, las mismas son **leídas** por Terraform a través de un **proveedor (provider)** de AWS, explicadas en la siguiente sección.

La Figura 16 muestra el flujo de este tipo de setup.



Figura 16. Esquema de Autenticación para Terraform en AWS. Fuente: Elaboración Propia

Los **beneficios** de este enfoque son:

- **Configuración rápida y sencilla** para desarrollo local.
- **Facilidad** a la hora de hacer pruebas e iteración durante el desarrollo.
- **No requiere infraestructura adicional.**

Aun así, las **desventajas** más claras son:

- Las **credenciales almacenadas localmente** suponen un **riesgo de seguridad**.
- **Difícil gestión y rotación de credenciales en equipos grandes.**
- **No proporciona un registro centralizado de accesos.**

Analizándolo desde una perspectiva de IT en entornos avanzados y de desarrollo colaborativo, es recomendable utilizar alguna de las siguientes alternativas junto con políticas de privilegios mínimos para los usuarios, además de rotación habitual de credenciales:

- **Uso de roles en AWS Identity and Access Management (IAM):** Permite el uso de roles **temporales**, que son especialmente útiles en entornos de Integración Contínua (CI) y Despliegue Contínuo (CD), como se muestra en la Figura 17.

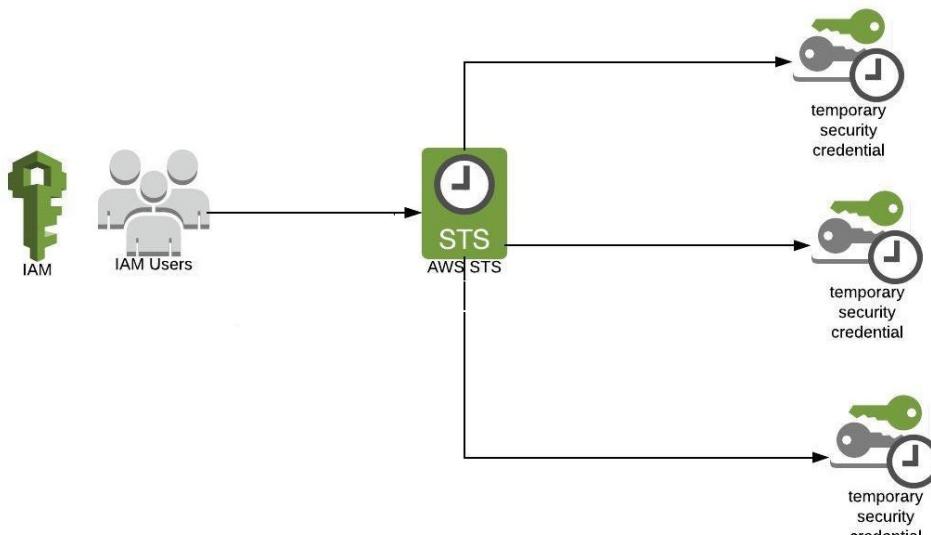


Figura 17. Autenticación contra AWS utilizando roles de IAM. Fuente: Elaboración Propia

- **AWS Single Sign-On (SSO):** Aplicaciones y servicios tercerizados que permiten autenticación de manera centralizada, se otorgan credenciales temporales que reemplazan a aquellas locales.
- **Manejo de Secretos Centralizados:** En lugar de guardar los secretos de manera local, utilizar un servicio centralizado como AWS Secrets Manager, HashiCorp Vault, entre otros.

Configuración de Proveedores en Terraform

Un proveedor (provider) en Terraform es una porción explícita del código (Figura 18) que le permite al usuario interactuar con una herramienta externa o servicio, tal como AWS. Un proveedor es análogo a un intérprete en un lenguaje de programación, ya que traduce las declaraciones de la configuración (escritas en HCL) a las llamadas y operaciones necesarias para modificar un esquema de infraestructura.

```
terraform {  
    required_providers {  
        # Definición del proveedor  
        aws = {  
            source = "hashicorp/aws"  
        }  
    }  
}  
  
# Configuración del proveedor  
provider "aws" {  
    region = var.region  
}
```

Figura 18. Declaración de un proveedor AWS en Terraform. Fuente: Elaboración Propia.

En el caso específico de AWS, es el componente que permite a Terraform comunicarse con los servicios de Amazon Web Services. Al ejecutar comandos como **plan** y **apply**, Terraform utilizará el proveedor AWS para:

- **Leer la configuración:** Obtener detalles sobre cómo conectarse (credenciales, región de AWS) y qué recursos gestionar.
- **Validar el código HCL:** Terraform utiliza el proveedor para validar que la estructura del código es válida tanto sintáctica como semánticamente.
- **Interactuar con APIs de AWS:** En este paso terraform utiliza las credenciales antes proporcionadas para realizar llamadas a la API de AWS creando, leyendo, modificando y/o eliminando los recursos definidos en la configuración.

Los proveedores son esenciales dado que permiten a Terraform gestionar diversos **servicios de infraestructura**, convirtiéndola en una herramienta no sólo **agnóstica**, si no **modular** y con capacidades **multi-proveedor al mismo tiempo bajo un mismo repositorio de código**, permitiendo definir recursos bajo un lenguaje común (HCL) sin importar si están en AWS, Azure, Google Cloud Platform, o incluso en sistemas locales como Docker, Kubernetes, o VMware.

Gestión de dependencias en Terraform

En Terraform cada recurso explicitado en el código al ser creado genera una salida (output), esto es, valores específicos de los recursos gestionados tal como su nombre, su identificador único en AWS (ARN, de las siglas Amazon Resource Name), entre otros.

Terraform construye automáticamente un árbol de dependencias entre recursos basándose en las **referencias existentes en el código**. Cuando un recurso necesita información de otro para su creación (por ejemplo, una instancia EC2 que necesita un VPC), Terraform establece una **dependencia implícita y asegura el orden correcto de creación**.

Estas dependencias se pueden establecer de dos formas:

- **Implícitas:** A través de referencias directas entre recursos usando atributos de salida (**outputs**), como se muestra en la Figura 19.

```
resource "aws_alb" "main" {  
    name          = "${var.environment}-${var.service}-alb"  
    subnets       = module.vpc.public_subnets  
    security_groups = [aws_security_group.microblog_alb_sg.id]  
}  
  
Dependencias Implícitas: Referencias directas de otros recursos usando sus Outputs
```

Figura 19. Dependencias implícitas en Terraform. Fuente: Elaboración Propia

- **Explícitas:** Mediante la directiva **depends_on**, cuando la dependencia no es detectable automáticamente. Como se muestra en la Figura 20.

```
resource "aws_ecs_task_definition" "microblog" [  
    family          = "${var.environment}-${var.service}-task"  
    execution_role_arn = aws_iam_role.ecs_task_execution_role.arn  
  
    depends_on = [null_resource.build_push_docker_img]  
]  
  
Dependencias Explícitas: Declaración de recursos que deben ser generados previos a este
```

Figura 20. Dependencias explícitas en Terraform. Fuente: Elaboración Propia

Terraform y el manejo de estado

El manejo del estado es un componente crucial en Terraform, dado que es la manera en que esta herramienta mantiene un **registro** del estado actual de la infraestructura desplegada. Este puede gestionarse de dos maneras distintas:

- **Local**
 - Almacenado por defecto en un archivo **terraform.tfstate**
 - **Simple** para desarrollo individual
 - **No recomendado** para trabajo en equipo
- **Remoto (Remote Backend)**
 - Se almacena el estado en un **servicio remoto** como AWS S3.
 - Permite **colaboración segura** en equipo.
 - Implementa **bloqueos** para prevenir modificaciones simultáneas.
 - Mantiene un **historial de cambios**.

Como podemos ver, el uso de un backend remoto es considerado una mejor práctica en entornos profesionales, ya que garantiza la consistencia y seguridad del estado de la infraestructura.

Para este proyecto se utiliza **AWS S3** como **backend remoto**, que proporciona **encriptación simétrica AES-256** para datos en reposo (at rest), **versionado del estado** para mantener un **historial de cambios**, **alta disponibilidad** y **durabilidad** inherente al servicio. Esta configuración, **complementada con políticas de IAM** (Amazon Web Services, 2025) para el control de acceso, asegura que el estado de la infraestructura permanezca **seguro, accesible y protegido contra pérdidas accidentales**.

La Figura 21 muestra el flujo de lectura/escritura en Terraform, manteniendo el registro en el tfstate file.

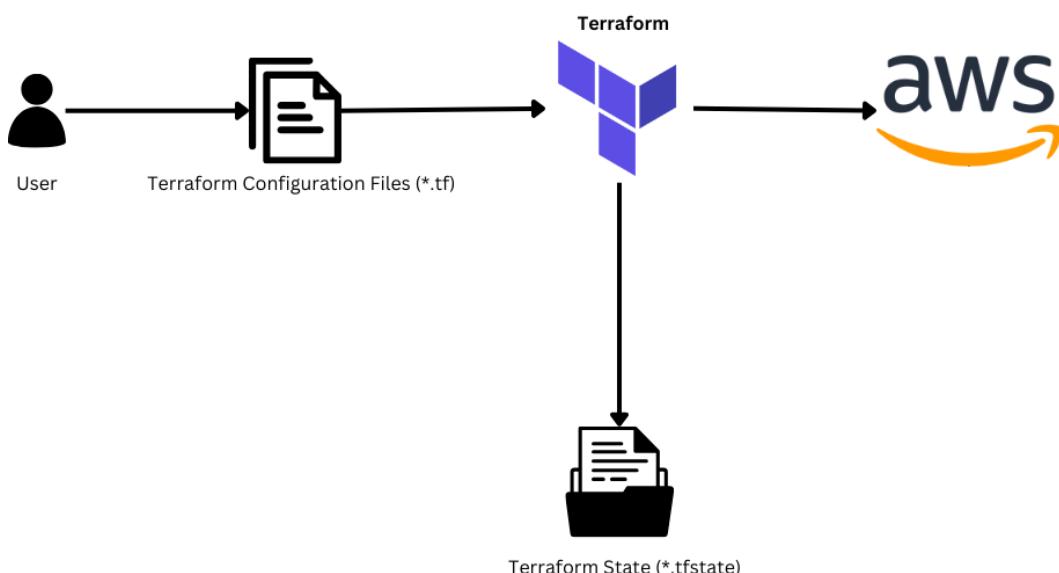


Figura 21. Uso del remote tfstate file en Terraform. Fuente: Elaboración Propia

Gestión de credenciales a nivel aplicación

La gestión segura de credenciales es crucial en aplicaciones más aún si están contenerizadas como en este caso, y AWS ofrece una solución robusta mediante la integración de Fargate con AWS SecretsManager, un servicio de claves seguras que las guarda en formato key/value. Esta implementación permite injectar secretos como variables de entorno en los contenedores en tiempo de ejecución (runtime), evitando el almacenamiento de credenciales sensibles en el código (hardcoded) o embebidas en la imagen de este. Este flujo de acceso a la información en tiempo real puede verse en la Figura 22.

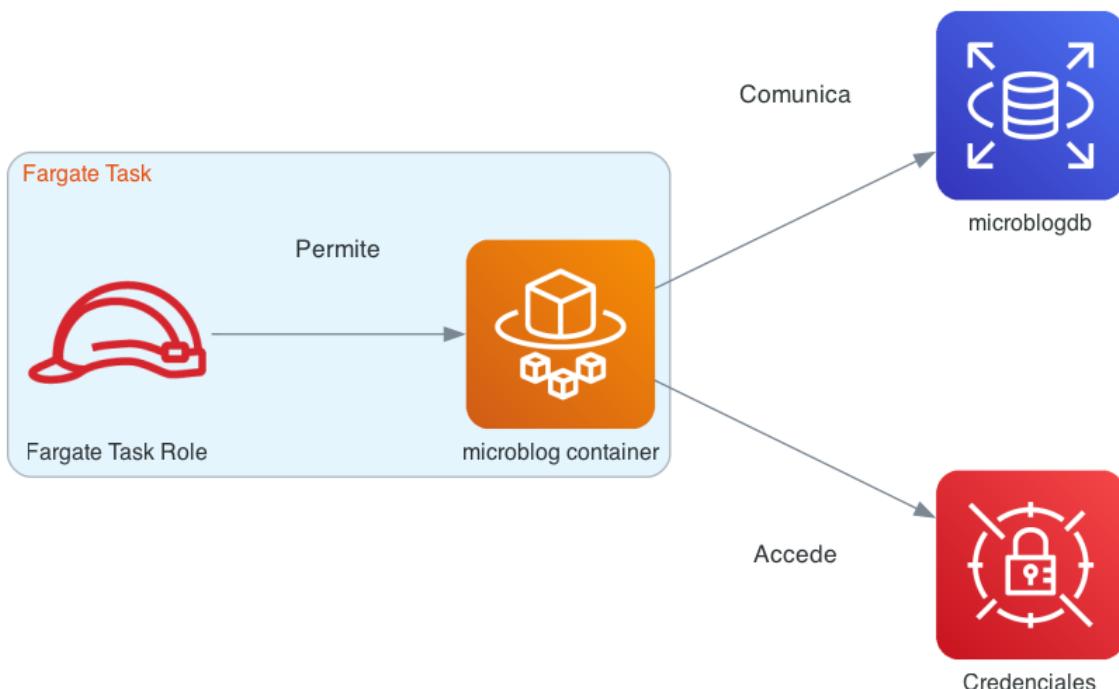


Figura 22. Gestión de credenciales en la aplicación. Fuente: Elaboración Propia

La configuración en Terraform requiere varios componentes trabajando en conjunto:

1. Primero, la contraseña de la base de datos se genera **de manera aleatoria** en Terraform de tal forma que **nunca podremos visualizarla**, y se le asigna a dicha DB. Como se ve en la Figura 23.

```

resource "random_password" "db" {
  length      = 16
  special     = true
  override_special = "!#$%&*()_+=[]{}<>;?"
}

keepers = {
  # Para generar un nuevo password cambiar la versión (e.g. v2, v3)
  version = "v2"
}

module "microblog_db" {
  source  = "terraform-aws-modules/rds/aws"
  version = "~> 6.12.0"

  identifier          = "${var.environment}-${var.service}-db"
  instance_use_identifier_prefix = true

  engine      = "postgres"
  engine_version = "16.6"
  instance_class = "db.t4g.micro"
  family       = "postgres16"

  db_name    = var.service
  username   = var.service
  port       = 5432

  allocated_storage = 100
  storage_encrypted = true

  multi_az        = false
  publicly_accessible = false

  manage_master_user_password = false
  password      = random_password.db.result
}

```

The diagram shows the Terraform code for generating a random password and assigning it to a database module. A red arrow points from the 'random_password' resource definition to the 'password' assignment in the 'microblog_db' module. Another red arrow points from the 'password' assignment to the 'random_password.db.result' value, indicating the path of the generated password.

Figura 23. Generación aleatoria de la contraseña de la DB en Terraform.

Fuente: Elaboración Propia

2. Luego, dicha contraseña generada de manera aleatoria se adjunta al secreto a almacenar (en este caso, **la URL de conexión a la base de datos**) en **AWS Secrets Manager** de forma segura, como muestra la Figura 24.

```

1  resource "aws_secretsmanager_secret" "db_secret" { ← Creación del secreto en SecretsManager
2    name           = "${var.environment}/${var.service}/microblog"
3    description    = "Database credentials for ${var.environment}-${var.service} DB"
4    recovery_window_in_days = 0
5  }
6
7
8  resource "aws_secretsmanager_secret_version" "db_secret_version" { ← Guardado de la URL de conexión a la DB
9    secret_id      = aws_secretsmanager_secret.db_secret.id
10   secret_string = jsonencode({ database_url: "postgres://${var.service}:${random_password.db.result}@${aws_route53_record.microblog_db.name}:5432/microblog" })
11 }

```

The diagram shows the Terraform code for creating a secret in AWS Secrets Manager and associating it with a database URL. Red arrows point from the 'aws_secretsmanager_secret' resource to the 'secret_id' variable and from the 'secret_string' assignment to the 'database_url' value, illustrating the connection between the two components.

Figura 24. Guardado de secretos en Secrets Manager via Terraform. Fuente: Elaboración Propia

3. Siguiente, se define un IAM Role que es un tipo de ACL (Access Control List) específico para el contenedor a ejecutar que incluye una regla que le otorga **permisos de lectura limitados** específicamente al secreto requerido, como muestra la Figura 25.

```
17  resource "aws_iam_role_policy_attachment" "ecs_task_execution_role_policy_attachment" {
18    role      = aws_iam_role.ecs_task_execution_role.name
19    policy_arn = "arn:aws:iam::aws:policy/service-role/AmazonECSTaskExecutionRolePolicy"
20  }
21
22  resource "aws_iam_policy" "secrets_access_policy" {
23    name      = "SecretsAccessPolicy"
24    description = "Allow access to specific secrets for ECS tasks"
25
26    policy = jsonencode({
27      Version = "2012-10-17",
28      Statement = [
29        {
30          Effect = "Allow",
31          Action = [
32            "secretsmanager:GetSecretValue",
33            "secretsmanager:DescribeSecret",
34          ],
35          Resource = aws_secretsmanager_secret.db_secret.arn
36        },
37        {
38          Effect = "Allow",
39          Action = [
40            "ecr:*",
41          ],
42          Resource = aws_secretsmanager_secret.db_secret.arn
43        }
44      ]
45    })
46  }
```

Definición de IAM Role

Política de Acceso (ACL)

Figura 25. Creación de IAM role y asignación de permisos en Terraform. Fuente: Elaboración Propia

4. Finalmente, en la configuración del contenedor se configura la **variable de entorno** utilizando el parámetro **valueFrom**, que hace referencia al secreto almacenado y automáticamente accede al mismo, sin necesidad de visualizarlo ni codificarlo en ningún momento dado que es un proceso interno en AWS. Esto se ve indicado en la Figura 26.

```
resource "aws_ecs_task_definition" "microblog" {  
    family           = "${var.environment}-${var.service}-task"  
    execution_role_arn = aws_iam_role.ecs_task_execution_role.arn  
  
    network_mode      = "awsvpc"  
    requires_compatibilities = ["FARGATE"]  
  
    cpu      = var.task_cpu  
    memory  = var.task_memory  
  
    lifecycle {  
        # Reemplaza `aws_ecs_task_definition` cada vez que `aws_secretsmanager_secret.db_secret` es reemplazado  
        # Esto es, cada vez que se genera una versión nueva del secreto  
        replace_triggered_by = [ aws_secretsmanager_secret_version.db_secret_version ]  
    }  
  
    container_definitions = jsonencode([  
        {  
            name      = "${var.environment}-${var.service}-app"  
            image     = local.ecr_docker_image  
            cpu       = var.task_cpu  
            memory   = var.task_memory  
            essential = true  
            portMappings = [  
  
                {  
                    containerPort = var.application_port  
                    hostPort     = var.application_port  
                }  
            ]  
            secrets = [  
                {  
                    name      = "DATABASE_URL"  
                    valueFrom = "${aws_secretsmanager_secret.db_secret.arn}:database_url::"  
                }  
            ]  
        }  
    ]  
}
```

Asignación del IAM role al contenedor

Fuerza al contenedor a reiniciar si se genera una nueva contraseña de la DB

Definición (Características) del contenedor a ejecutar

Asignación de Variable de Entorno

Identificador del Contenedor a Ejecutar

Figura 26. Asignación de Variables de Entorno y referencia de Secretos en Terraform.
Fuente: Elaboración Propia

Esta arquitectura proporciona **varios beneficios de seguridad**:

- El principio de mínimo privilegio se mantiene al **limitar el acceso del contenedor solo a los secretos necesarios**.
- **Las credenciales nunca se exponen** en el código o en los archivos de configuración, ni siquiera al usuario ejecutando Terraform.
- Cualquier rotación de secretos en SecretsManager se refleja automáticamente en los nuevos contenedores **sin necesidad de modificar la configuración del contenedor**.

5. Resultados

Implementación Exitosa de la Infraestructura

La implementación de la infraestructura en AWS mediante Terraform demostró ser significativamente más eficiente que un despliegue **manual tradicional**.

En primer lugar, porque con exactitud podemos saber exactamente los recursos se van a desplegar dada la utilidad del **terraform plan** (Figura 27), que nos muestra la cantidad y el tipo de recursos de infraestructura:

```
# module.microblog_db.module.db_subnet_group.aws_db_subnet_group.this[0] will be created
+ resource "aws_db_subnet_group" "this" {
    + arn                  = (known after apply)
    + description          = "prod-microblog-db subnet group"
    + id                   = (known after apply)
    + name                 = (known after apply)
    + name_prefix          = "prod-microblog-db-"
    + subnet_ids           = (known after apply)
    + supported_network_types = (known after apply)
    + tags                 = {
        + "CreatedBy"   = "Terraform"
        + "Environment" = "prod"
        + "Name"        = "prod-microblog-db"
        + "Service"     = "microblog"
    }
    + tags_all              = {
        + "CreatedBy"   = "Terraform"
        + "Environment" = "prod"
        + "Name"        = "prod-microblog-db"
        + "Service"     = "microblog"
    }
    + vpc_id                = (known after apply)
}

Plan: 54 to add, 0 to change, 0 to destroy.

Changes to Outputs:
+ ecs_lb_dns_name = (known after apply)
```

Figura 27. Output del plan de Terraform. Fuente: Elaboración Propia

La gestión de outputs entre componentes fue crucial para mantener una arquitectura modular y segura. Por ejemplo, el ID de la VPC creada se utiliza como input para la creación de subredes privadas y públicas, y el secreto generado de manera aleatoria para la base de datos se utiliza como input para SecretsManager y para setear en la base de datos, a su vez dicho secreto guardado en SecretsManager es necesario para la referencia de permisos en el IAM Role del contenedor, como muestra la Figura 28. Esta capacidad de Terraform de enlazar outputs entre recursos no solo reduce errores de configuración manual, sino que también facilita cambios futuros al mantener las referencias actualizadas automáticamente.



Figura 28. Árbol de Dependencias de Outputs en Terraform. Fuente: Elaboración Propia.

Por otro lado, el tiempo de **despliegue completo** de la infraestructura se **redujo** de aproximadamente **2 horas y media** (estimado para despliegue manual) a **menos de 15 minutos utilizando Terraform**, aproximadamente un 92% más rápido, como muestra la Figura 29.

```

Apply complete! Resources: 54 added, 0 changed, 0 destroyed.

Outputs:

ecs_lb_dns_name = "prod-microblog-alb-1020002464.us-east-1.elb.amazonaws.com"

real    12m18.258s
user    0m45.444s
sys     0m8.958s
  
```

Figura 29. Tiempos de Creación de la Infraestructura via Terraform. Fuente: Elaboración Propia

Para **validar** que la infraestructura creada es funcional, basta con localizar el output **ecs_lb_dns_name**, que es la URL de ingreso a la aplicación, un ejemplo de esto se ve claramente en la Figura 30.

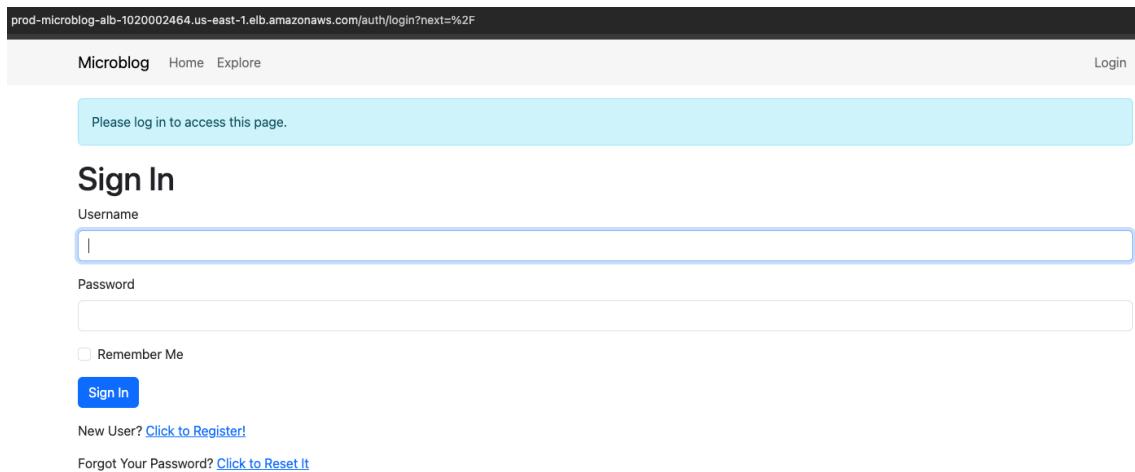


Figura 30. URL del sitio web desplegado desde un navegador web. Fuente: Elaboración Propia

A la hora de **destruir** la infraestructura, el tiempo total de destrucción **ronda los 17 minutos**, hacerlo esto de manera manual lleva aproximadamente 1 hora y media, o sea un 71% más rápido utilizando Infraestructura como código, como muestra la Figura 31.

```
Destroy complete! Resources: 54 destroyed.

real    16m46.595s
user    0m13.310s
sys     0m2.815s
```

Figura 31. Tiempo de Destrucción de la Infraestructura via Terraform. Fuente: Elaboración Propia

Además de todas estas mejoras **cuantitativas**, hay otros **beneficios menos perceptibles mas no menos importantes** como:

- **Eliminación de errores humanos** comunes en configuraciones manuales.
- **Consistencia** garantizada entre ambientes de desarrollo, pruebas y producción.
- **Capacidad de despliegue y cambios** de ambientes completos.
- **Control de versiones** efectivo de la infraestructura utilizando Git.
- **Documentación** autogenerada en forma de código.
- **Reducción de los tiempos de aprendizaje** para los técnicos dado que, a pesar de ser deseable, ya no necesitan conocer cada detalle de la infraestructura cloud que están desplegando.

La Tabla 4 muestra de manera comparativa las mejoras cualitativas y cuantitativas percibidas al utilizar Infraestructura como Código para desplegar la aplicación, microblog, creada para este proyecto.

Tabla 4. Comparación entre despliegues manuales y automatizados. Elaboración propia.

Características	Despliegue Manual	Despliegue con laC
Tiempo de Despliegue	210 minutos	15 minutos
Tiempo de Destrucción	90 minutos	17 minutos
Replicable	No	Si
Factores de Error Humano	Si	No
Uso de memoria	Alto	Bajo
Requiere Conocimientos Previos de AWS	Si	No
Documentación Auto-generada	No	Sí
Auditable	No	Sí
Versionable	No	Sí

Seguridad y Buenas Prácticas

La implementación se desarrolló siguiendo el principio de “seguridad por diseño”, incorporando múltiples capas de seguridad y siguiendo las **buenas prácticas recomendadas por AWS**.

1. Gestión de Accesos, Secretos y Credenciales:

- a. Implementación de AWS Secrets Manager como **solución centralizada para el manejo de secretos**, eliminando la necesidad de almacenar credenciales en código o variables de entorno.
- b. Proporciona un sistema de rotación de credenciales que minimiza el riesgo de exposición y reduce la intervención humana.
- c. Integración con la aplicación de microblog mediante roles de IAM, evitando el uso de credenciales estáticas para el acceso a los recursos.
- d. Implementación del principio de privilegio mínimo mediante roles de IAM específicos limitados para el servicio de microblog.

2. Seguridad en la red:

- a. Arquitectura de red segmentada utilizando subnets públicas y privadas.
- b. Segregación de accesos basados en grupos de seguridad, limitando el acceso a los recursos necesarios
- c. Acceso a internet restringido sólo a recursos que lo requieren explícitamente.

3. Gestión Segura de la Infraestructura:

- a. Estado de Terraform (tfstate) almacenado en S3 con encriptación en reposo.
- b. Versionado del estado para mantener un historial de cambios y facilitar reverisiones (rollbacks).

- c. Bloqueos de estado que previenen modificaciones concurrentes, particularmente útil en entornos corporativos o de trabajo en equipo.

Escalabilidad y Mantenibilidad

El diseño implementado demostró ser altamente escalable y mantenible, tanto en el código como en la infraestructura desplegada:

- **Capacidad de escalar horizontalmente** la aplicación mediante ECS Fargate con un simple cambio en la cantidad de contenedores a ejecutar.
- En la misma línea, el **balanceo de carga** distribuye el tráfico entrante en cuantos contenedores haya, independientemente de la cantidad.
- **Recuperación** ante fallos de contenedores dadas las capacidades de Fargate de reponerlos automáticamente.
- Código **modular** y fácilmente **reutilizable** que facilita la **replicación** de la infraestructura en otros entornos (desarrollo, UAT, producción, etc).
- **Facilidad de actualización y modificación** de la infraestructura.

6. Conclusiones

Impacto en el Desarrollo y Despliegue de la Infraestructura

La implementación de la Infraestructura como Código (IaC) via Terraform ha tenido un impacto significativo en la evolución del ciclo de desarrollo y despliegue de entornos en la nube.

Automatización y Eficiencia Operativa

La automatización del proceso de aprovisionamiento ha reducido drásticamente los tiempos de despliegue, optimizando la disponibilidad de recursos en cada etapa del desarrollo. Esto no sólo supone un ahorro considerable de tiempo, sino que también elimina tareas manuales repetitivas, permitiendo a los equipos enfocarse en actividades estratégicas, como el diseño de arquitectura siguiendo mejores prácticas de seguridad y escalabilidad.

Consistencia y Reducción de Errores

La capacidad de definir infraestructura de manera declarativa y versionada garantiza que los entornos de desarrollo, pruebas y producción sean coherentes y replicables, minimizando errores de configuración que suelen surgir en despliegues manuales. Además, herramientas como **terraform plan** y **terraform apply** permiten visualizar cambios antes de aplicarlos, aumentando la previsibilidad y el control sobre la infraestructura.

Escalabilidad y Mantenimiento Simplificado

Terraform facilita la gestión dinámica de recursos, permitiendo escalar la infraestructura en función de las necesidades del sistema sin intervención manual. Este enfoque mejora la resiliencia operativa y optimiza el mantenimiento a largo plazo, ya que cualquier ajuste o actualización puede aplicarse de manera segura y reproducible.

Con estos beneficios, IaC no solo transforma la forma en que se gestiona la infraestructura, sino que también fortalece la agilidad del desarrollo y mejora la estabilidad de los sistemas en producción, alineándose con los principios DevOps y de automatización en la nube.

Áreas de Mejora y Trabajo Futuro

Si bien este Trabajo de Fin de Grado ha logrado establecer una base sólida para la automatización de infraestructura, existen varias áreas de oportunidad para futuras iteraciones y mejoras.

Una primera área de mejora sería la implementación de **pruebas automatizadas de infraestructura utilizando modelos de Inteligencia Artificial**. Dichos modelos podrían analizar patrones históricos de fallos y comportamientos anómalos para predecir posibles problemas antes de que ocurran, permitiendo una validación más proactiva de la configuración y el funcionamiento de los recursos desplegados.

Estas pruebas permitirían validar de manera sistemática la configuración y el funcionamiento de los recursos desplegados, garantizando que cada cambio en la infraestructura mantenga la integridad y seguridad del sistema.

Otra mejora significativa sería la **implementación de auto-escalado dinámico potenciado por Inteligencia Artificial**. Actualmente, la infraestructura mantiene un número fijo de contenedores, pero un sistema de auto-escalado basado en machine learning podría predecir patrones de uso y ajustar proactivamente la cantidad de contenedores antes de los picos de demanda, optimizando tanto el rendimiento como los costos operativos.

La integración con sistemas de **Integración Continua y Despliegue Continuo (CI/CD)** como GitHub Actions (Laster, 2023), podría enriquecerse con herramientas de Inteligencia Artificial para análisis de código y seguridad. Dichas herramientas podrían detectar automáticamente vulnerabilidades potenciales, sugerir optimizaciones en la configuración de la infraestructura y garantizar el cumplimiento de las mejores prácticas en seguridad.

Adicionalmente, se podría implementar un sistema de monitoreo inteligente respaldado por Inteligencia Artificial para detectar anomalías en tiempo real, identificar patrones de comportamiento inusuales y sugerir acciones correctivas de manera automática, mejorando la capacidad de respuesta ante incidentes.

Estas mejoras propuestas no solo aumentarían la robustez de la solución, sino que también la acercarían más a los estándares actuales de la industria para entornos de producción a gran escala, aprovechando el potencial de la IA para volver la infraestructura más inteligente, predictiva y autónoma.

Reflexión Final

Los avances en Desarrollo de la Infraestructura y Eficiencia Operativa han sido claves para consolidar una Cultura DevOps más madura, donde la gestión de infraestructura sigue los mismos principios de rigor y disciplina que el desarrollo de software.

Esta evolución representa un cambio cultural fundamental: Los entornos tecnológicos dejan de ser administrados de manera manual y fragmentada, para convertirse en sistemas automatizados, reproducibles y auditables, alineados con la filosofía de Infraestructura como Código (IaC).

Gracias a este enfoque, los equipos de desarrollo y operaciones pueden colaborar de manera más efectiva, impulsando la agilidad y calidad en la entrega de software moderno. La estandarización y automatización de procesos no sólo reducen errores y tiempos de despliegue, sino que también refuerzan la estabilidad y seguridad de los entornos en producción. Este modelo de trabajo no es sólo una optimización técnica, sino una evolución estructural que redefine la forma en que se conciben, implementan y gestionan los sistemas en la nube.

7. Referencias

Allison, R. (2020). The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers. Association for Computing Machinery.

<https://dl.acm.org/doi/fullHtml/10.1145/3365199>

Amazon Web Services. (2023). The history and future roadmap of the AWS CloudFormation Registry.

<https://aws.amazon.com/es/blogs/devops/cloudformation-coverage/>

Amazon Web Services. (2025a). Prepare for least-privilege permissions.

<https://docs.aws.amazon.com/IAM/latest/UserGuide/getting-started-reduce-permissions.html>

Amazon Web Services. (2025b). Security best practices in IAM.

<https://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html>

Amazon Web Services. (n.d.). AWS Well-Architected Framework.

<https://aws.amazon.com/architecture/well-architected/>

ChartDB. (2025). Welcome to ChartDB.

<https://docs.chartdb.io/docs/welcome>

GitHub. (2024). Include diagrams in your Markdown files with Mermaid.

<https://github.blog/developer-skills/github/include-diagrams-markdown-files-mermaid/>

HashiCorp. (2021). The Story of HashiCorp Terraform with Mitchell Hashimoto.

<https://www.hashicorp.com/en/resources/the-story-of-hashicorp-terraform-with-mitchell-hashimoto>

HashiCorp. (2023). HashiCorp Terraform ecosystem passes 3,000 providers with over 250 partners.

<https://www.hashicorp.com/en/blog/hashicorp-terraform-ecosystem-passes-3-000-providers-with-over-250-partners>

Kane, S. P., & Matthias, K. (2023). Docker: Up & Running (3rd ed.). O'Reilly Media.

Kernighan, B. W., & McIlroy, M. D. (1979). UNIX Time-Sharing System: UNIX Programmer's Manual (7th ed., Vol. 1). Bell Telephone Laboratories Incorporated.

Laster, B. (2023). Learning GitHub Actions. O'Reilly Media.

Morris, K. (2025). Infrastructure as Code (3rd ed.). O'Reilly Media.

National Institute of Standards and Technology. (2011). The NIST Definition of Cloud Computing. NIST SP 800-145.
<https://csrc.nist.gov/pubs/sp800/145/final>

Wilsenach, R. (2015). DevOps Culture. Martin Fowler Blog.
<https://martinfowler.com/bliki/DevOpsCulture.html>

Wittig, A., & Wittig, M. (2023). Amazon Web Services in Action (3rd ed.). Manning Publications.

National Institute of Standards and Technology. (2011). The NIST Definition of Cloud Computing. NIST SP 800-145.
<https://csrc.nist.gov/pubs/sp800/145/final>

Wittig, A., & Wittig, M. (2023). Amazon Web Services in Action (3rd ed.). Manning Publications.