****

**Infraestructura mediante Código via Terraform en Amazon Web Services**

|  |  |  |
| --- | --- | --- |
| **Titulación:**  Grado en Ingeniería Informática  Curso académico  2024-2025 | **Alumno/a:**Gagliardo, Miguel Angel  **Pasaporte:** AAI072641  **Director/a de TFG:** Ángela Di Serio | **Convocatoria:** Primera |

**Dedicatoria**

Este Trabajo de Fin de Grado es el resultado del esfuerzo de años de trabajo y estudio, pero sobre todo del soporte de familia:

* En primer lugar a mi padre, **Miguel**,quién está en Argentina y verá esto de manera remota, pero que siempre me exigió ir por más, porque siempre hay que apuntar más alto.
* Por otro lado, y aunque nunca lo vaya a ver quisiera dedicarle este trabajo a mi madre, **Norma**, que falleció hace casi 9 años y no va a verme finalizar mi título de grado, pero que siempre quiso que su hijo tenga uno.
* Pero sobre todo quisiera dedicarle este Trabajo de Fin de Grado en particular a mi pareja y compañera de vida, **Pamela**, quien me ha soportado dándome todo el apoyo que necesito para que logre llegar donde estoy, después de tantos años.

**Agradecimientos**

Quisiera agradecerle a los docentes de la VIU que se esfuerzan por forjar alumnos y elevan la vara de la educación en España y en 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 fué 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 auditable, 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](#_Toc135392734)

Antecedentes [9](#_Toc135392742)

Planteamiento del Problema [9](#_Toc135392742)  
Justificación [10](#_Toc135392742)  
Objetivos [10](#_Toc135392742)

[2. Marco teórico 12](#_Toc135392739)

[Computación En La Nube 12](#_Toc135392740)

[Amazon Web Services (AWS) 12](#_Toc135392740)

Terraform e [Infraestructura como Código 18](#_Toc135392740)

Contenedores [21](#_Toc135392740)

[3. Metodología 23](#_Toc135392741)

Project Management [23](#_Toc135392742)  
Herramientas [24](#_Toc135392742)  
Limitaciones [25](#_Toc135392742)

[4. Desarrollo 26](#_Toc135392741)

Trabajo en Sprints26  
Desarrollo de la Aplicación "Microblog" [27](#_Toc135392742)  
Modelado de Datos [28](#_Toc135392742)

Diseño de la Infraestructura [29](#_Toc135392742)  
Desarrollo de la Infraestructura como Código [32](#_Toc135392742)

Gestión de Credenciales para el Desarrollo Local [33](#_Toc135392742)  
Configuración de Proveedores en Terraform [34](#_Toc135392742)

Gestión de Dependencias en Terraform [36](#_Toc135392742)  
Terraform y el Manejo de Estado [37](#_Toc135392742)

Gestión de Credenciales a Nivel Aplicación  [38](#_Toc135392742)

[5. Resultados 43](#_Toc135392744)

Implementación Exitosa de la Infraestructura  [43](#_Toc135392742)  
Seguridad y Buenas Prácticas [46](#_Toc135392742)  
Escalabilidad y Mantenibilidad [47](#_Toc135392742)

[6. Conclusiones 47](#_Toc135392745)

Impacto en el Desarrollo de la Infraestructura [47](#_Toc135392742)  
Eficiencia Operativa [48](#_Toc135392742)  
Áreas de Mejora y Trabajo Futuro [48](#_Toc135392742)  
Reflexión Final [49](#_Toc135392742)

[7. Referencias 50](#_Toc135392746)

**Índice de figuras**

Figura 1. Ambientes de desarrollo de software en entornos ágiles. Fuente:

*https://docs.cloudbees.com/docs/cloudbees-cd/latest/plan/server-topology* 9

Figura 2. Servicios de Computación en la Nube. Fuente:

*https://www.future-processing.com/blog/how-to-implement-cloud-computing/* 12

Figura 3. Arquitectura Global de Amazon Web Services. Fuente:

*https://aws.amazon.com/es/about-aws/global-infrastructure* 13

Figura 4. Datos de Cuota de Mercado de la Nube para Q1 2022. Fuente:

https://www.theregister.com/2022/05/02/cloud\_market\_share\_q1\_2022 14

Figura 5. Cuadrante para 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/ 15

Figura 6. Workflow de Terraform. Fuente:

https://build5nines.com/terraform-workflow-process-explained/ 16

Figura 7. Sprint Board en Github. Fuente:

https://github.com/users/mgagliardo/projects/1/views/2 19

Figura 8. Repositorio del Trabajo de Final de Grado. Fuente:

https://github.com/mgagliardo/viu-84giin-tfg/ 20  
Figura 9. Sprint 2 del Trabajo de Final de Grado. Fuente:

https://github.com/mgagliardo/viu-84giin-tfg/ 22

Figura 10. Burn up Chart del Sprint del Trabajo de Final de Grado. Fuente: https://github.com/mgagliardo/viu-84giin-tfg/ 23

Figura 11. Diagrama de clases de la aplicación Microblog. Fuente:

Elaboración Propia 24

Figura 12. Diagrama Entidad-Relación de la DB para la aplicación Microblog. Fuente: Elaboración Propia 25

Figura 13. Código de Terraform para explicitar la cantidad requerida de contenedores. Fuente: Elaboración Propia 26

Figura 14. Diseño de la Infraestructura en AWS. Fuente:

Elaboración Propia 26

Figura 15. Esquema de Autenticación para Terraform en AWS. Fuente:

Elaboración Propia 28

Figura 16. Autenticación contra AWS utilizando roles de IAM. Fuente:

Elaboración Propia 28

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

Elaboración Propia 29

Figura 18. Dependencias implícitas en Terraform. Fuente:

Elaboración Propia 30

Figura 19. Dependencias explícitas en Terraform. Fuente:

Elaboración Propia 31  
Figura 20. Gestión de credenciales en la aplicación. Fuente:

Elaboración Propia 32

Figura 21. Gestión de credenciales en la aplicación. Fuente:

Elaboración Propia 32

Figura 22. Generación aleatoria de la contraseña de la DB en Terraform. Fuente:

Elaboración Propia 33

Figura 23. Guardado de secretos en Secrets Manager via Terraform. Fuente:

Elaboración Propia 33

Figura 24. Creación de IAM role y asignación de permisos en Terraform. Fuente:

Elaboración Propia 34

Figura 25. Asignación de Variables de Entorno y referencia de Secretos en Terraform. Fuente: Elaboración Propia 35

Figura 26. Output del Plan de Terraform. Fuente:

Elaboración Propia 36  
Figura 27. Tiempos de Creación de la Infraestructura via Terraform. Fuente: Elaboración Propia 37

Figura 28. URL del sitio web desplegado desde un navegador web. Fuente: Elaboración Propia 37  
Figura 28. Tiempos de Destrucción de la Infraestructura via Terraform. Fuente: Elaboración Propia 37

**Índice de tablas**

[Tabla 1. Comparación entre Terraform y herramientas de IaC. Elaboración propia  
… 17](#_Toc6399146)

Tabla 2. Comparación entre Máquinas Virtuales y Contenedores. Elaboración propia. 18  
Tabla 3. Comparación entre despliegues manuales y automatizados. Elaboración propia 39

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 servidores e infraestructura en la nube eventualmente también se volvió un cuello de botella. Ante esta problemática nace el concepto de infraestructura como código, que permite definir y gestionar entornos de infraestructura de manera automatizada, replicable y auditable, 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.

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 requieren entornos similares para que, al ejecutar las fases de testing, se espere que todo funcione como se espera.

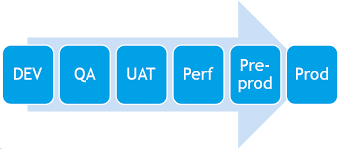


Figura 1. Ambientes de desarrollo de software en entornos ágiles.

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.

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 Codigo 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.

En este trabajo en el marco teórico se presentaran **definiciones e** **introducciones a las herramientas** utilizadas: AWS, Terraform y Contenedores. Bajo la sección de metodología se explicara el **desarrollo del entorno** y la **metodología de trabajo implementada**, y en la sección de **desarrollo** se explicara cómo fue la implementación practica tanto de la aplicación como los contenedores e Infraestructura como Código para, finalmente en las últimas dos secciones dar la evaluación de los resultados y concluir con un análisis final.

1. Marco Teórico

En este capitulo se presentan los fundamentos teoricos y conceptuales relevantes en los cuales se baso el desarrollo de este trabajo.

Computación en la Nube

De acuerdo a la definición del National Institute of Standards and Technology (NIST)1 la computacion en la nube o “la nube” es un modelo que permite el acceso a la red, ubicuo, practico y bajo demanda a un conjunto compartido de recursos informaticos configurables como redes, servidores, almacenamiento, aplicaciones y servicios que pueden aprovisionarse y liberarse rápidamente con un esfuerzo de gestion o interacion minimo con el proveedor de dichos servicios.

Este modelo tiene cinco caracteristicas esenciales:

* *Servicio bajo demanda.*
* *Amplio acceso a la red.*
* *Agrupacion 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)*

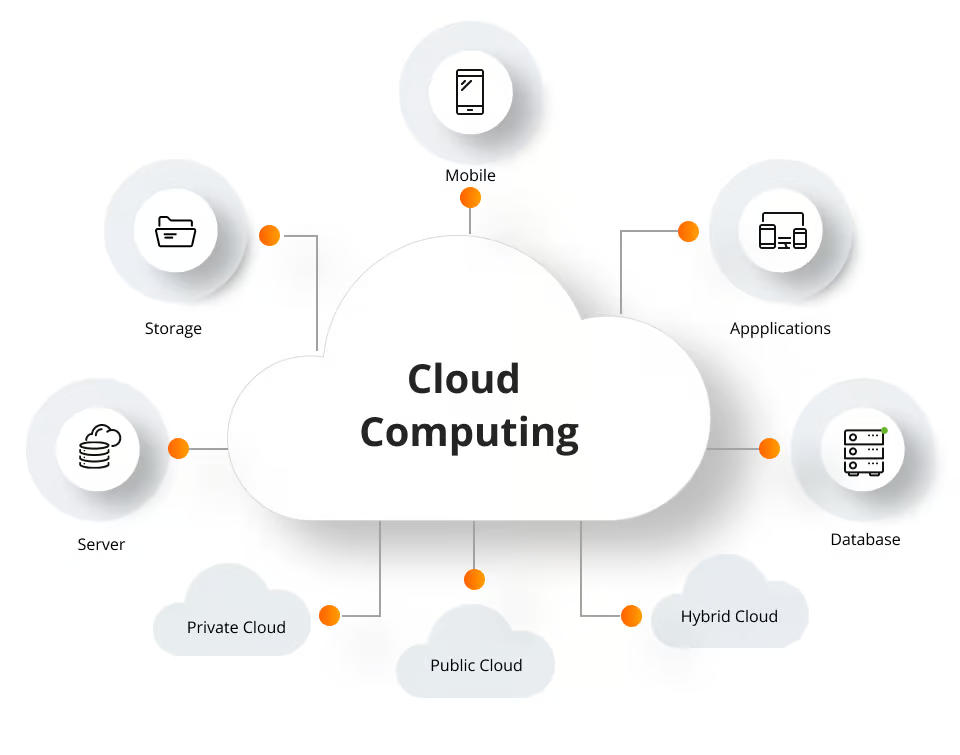


Figura 2. Servicios de Computación en la Nube.

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

Amazon Web Services (AWS)

Tomando como base la definición antes vista, Amazon lanza en 2006 Amazon Web Services (AWS)2, 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 termino “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 Elastico en la nube) que ofrece servidores virtuales y S3 (Simple Storage Service o Servicio de Almacenamiento Simple) que ofrece capacidad almacenamiento practicamente 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 multiples centros de datos alrededor del mundo, distribuidos en America, Europa, Asia y Oceania. Lo que nos da la posibilidad de desplegar servidores o almacenar datos en Japon de la misma manera que lo podemos hacer en Europa o America, y con la misma velocidad. Esto no solo muestra la consistencia y potencia del servicio si no tambien permite a las empresas y organizciones dsplegar infraestructura a nivel global en muy poco tiempo y con un esfuerzo minimo.



Figura 3. 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 computacion 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) publicas de Amazon Web Services e integrar sus propias aplicaciones con un abanico de lenguajes de programacion tales como Java, Javascript, Python, Golang, entre tantos otros.

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

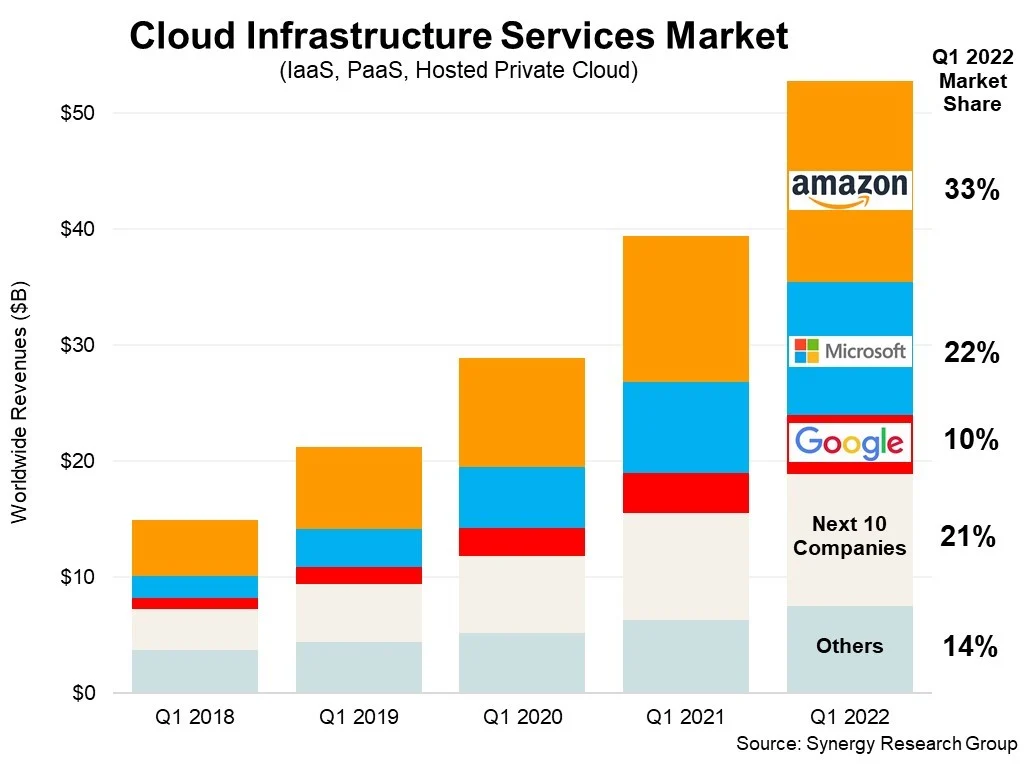


Figura 4. Datos de Datos de Cuota de Mercado de la Nube para Q1 2022

Fuente: https://www.theregister.com/2022/05/02/cloud\_market\_share\_q1\_2022/

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 responde a las razones antes mencionadas:** AWS es el proveedor de servicios en la nube más consolidado del mercado y una participación líder en términos de cuota global. Esto se refleja en una infraestructura robusta, altamente distribuida y con vasta documentación tanto oficial como de su comunidad que facilita su uso y aprendizaje. Además y como veremos en el siguiente apartado, Terraform se integra íntimamente con AWS siendo uno de los proveedores con mayor mantenimiento que cubre la mayoría de sus servicios y permite desplegar recursos de manera declarativa, controlada y reproducible.



Figura 5. 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 CloudFormation5 en 2011 (usando JSON/YAML). En 2014, Mitchell Hashimoto creó Terraform, una alternativa de código abierto que funcionaba con múltiples proveedores cloud

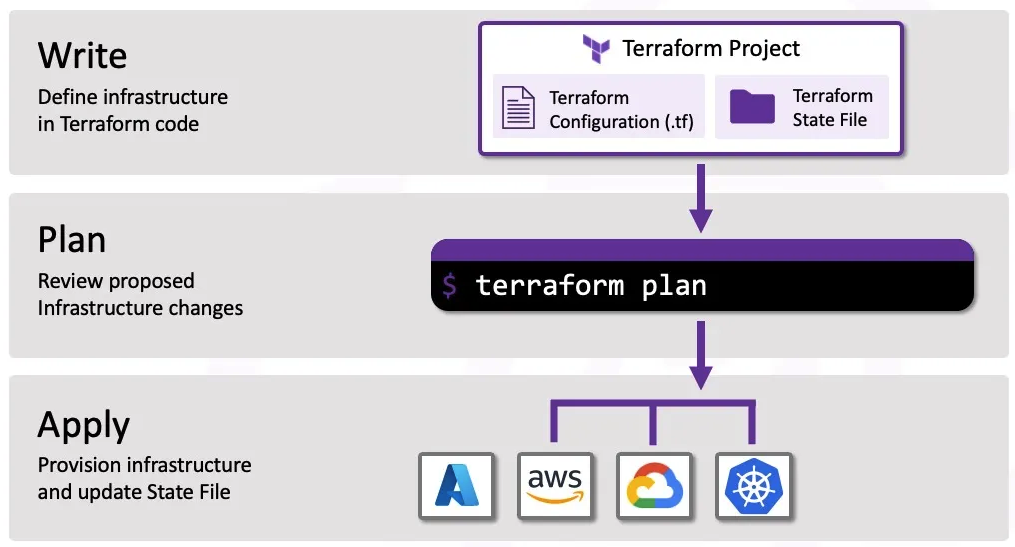


Figura 6. Workflow de Terraform

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 proveedores7.
* **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.

Tabla 1. Comparación entre Terraform y herramientas de IaC. Elaboración propia.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Caracteristicas** | **Terraform** | **Pulumi** | **AWS CDK** | **Ansible** |
| **Soporte Multi-Cloud** | ✅ | ✅ | ❌ | ✅ |
| **Manejo de Estado** | ✅ | ✅ | ✅ | ❌ |
| **Licencia OpenSource** | ✅ | ✅ | ❌ | ✅ |
| **No require código** | ✅ | ❌ | ❌ | ❌ |
| **Gratuito** | ✅ | ❌ | ✅ | ✅ |
| **Extensible** | ✅ | ❌ | ❌ | ✅ |
| **Lenguaje** | HCL (DSL) | Multi-Lang | Multi-Lang | Python |
| **Curva de Aprendizaje** | Baja | Alta | Alta | Media |

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 tecnología conocida como contenedores surge de la necesidad de paralelizar y aislar procesos en computadores. Esta tecnología evoluciona desde la **virtualización**, introducida por IBM en 19608, estableciendo dos requisitos fundamentales:

* **Aislamiento:** Los programas no deben verse entre sí.
* **Limitación de recursos:** Los programas **no pueden** utilizar tantos recursos nativos de la máquina como deseen (CPU, Memoria, Red), deben existir límites.

Docker10, empresa pionera en contenerización, la define como “unidades estándar de software que empaquetan código y dependencias para garantizar una ejecución consistente en diferentes entornos”. El componente fundamental de un contenedor es una **imagen**, que es una forma de empaquetado que incluye todo lo necesario para ejecutar una aplicación, diferenciándose del empaquetado tradicional por ser **agnóstica al sistema operativo**.

Las imágenes se transforman en contenedores durante su ejecución, garantizando un funcionamiento uniforme independiente de la infraestructura. Sus características principales son:

* **Estándar:** Docker creó el estándar de la industria para contenedores, por lo que pueden ser portables en cualquier lugar.
* **Ligero:** Los contenedores comparten el kernel del sistema operativo de la máquina y, por lo tanto, no requieren un sistema operativo por aplicación, lo que impulsa una mayor eficiencia del servidor y reduce los costos de servidor y licencias.
* **Seguro:** Las aplicaciones son más seguras en contenedores y Docker ofrece las capacidades de aislamiento predeterminadas más sólidas de la industria.

Tabla 2. Comparación entre Máquinas Virtuales y Contenedores. Elaboración propia.

|  |  |  |
| --- | --- | --- |
| **Caracteristicas** | **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 |

1. Metodología

Project Management

Para este Trabajo de Fin de grado se implementó un marco de trabajo ágil basado en Scrum. Las razones 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 Sprint.
* Planificación: Tareas definidas como Issues en Github.
* Seguimiento: Reuniones periódicas con el tutor.
* Métricas: Burn Up charts nativos de Github.

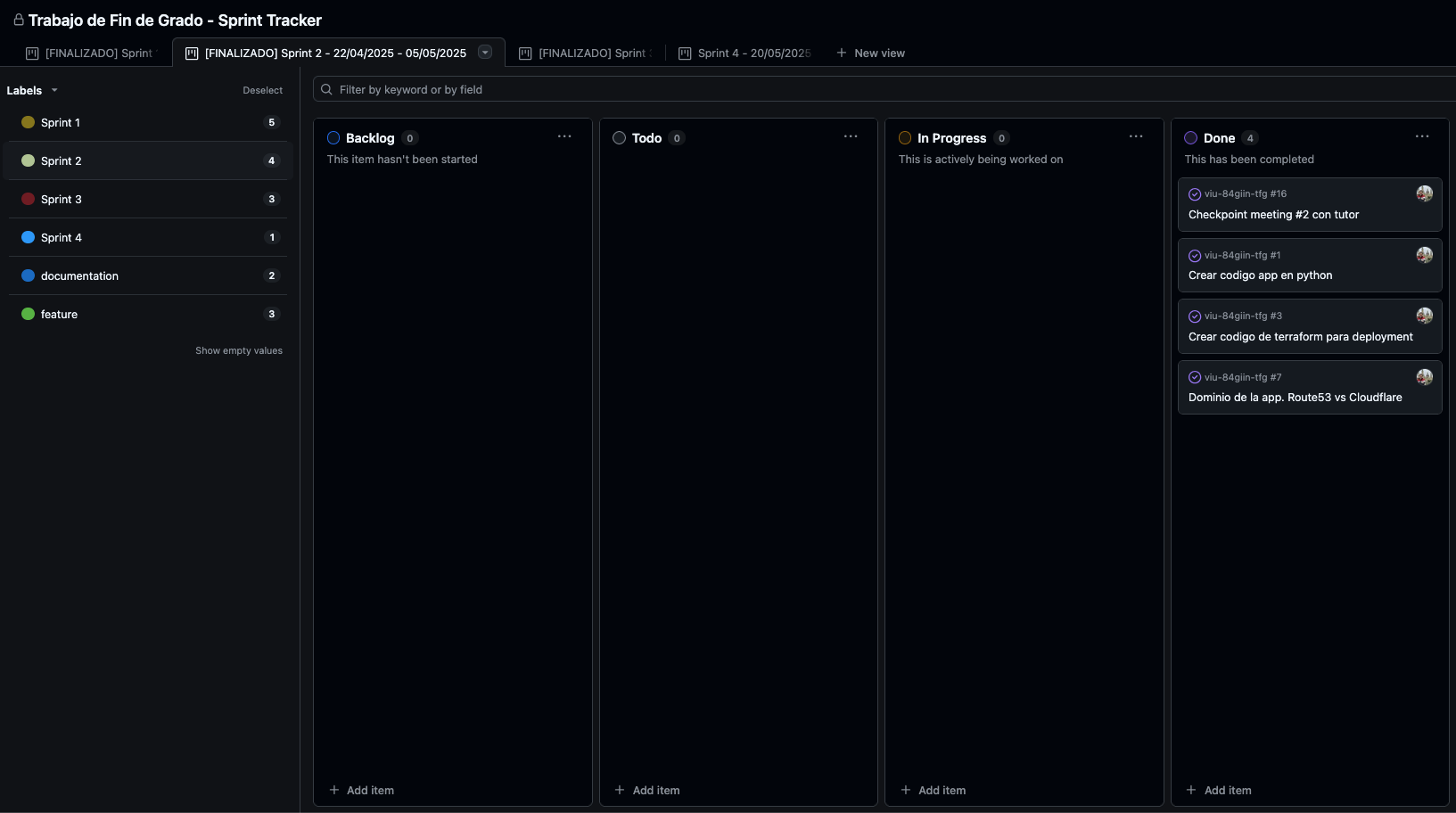


Figura 7. Sprint Board en Github

Fuente: https://github.com/mgagliardo/viu-84giin-tfg

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 abierto11,12
* 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 (Fig. 7, Fig. 8, Fig. 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.

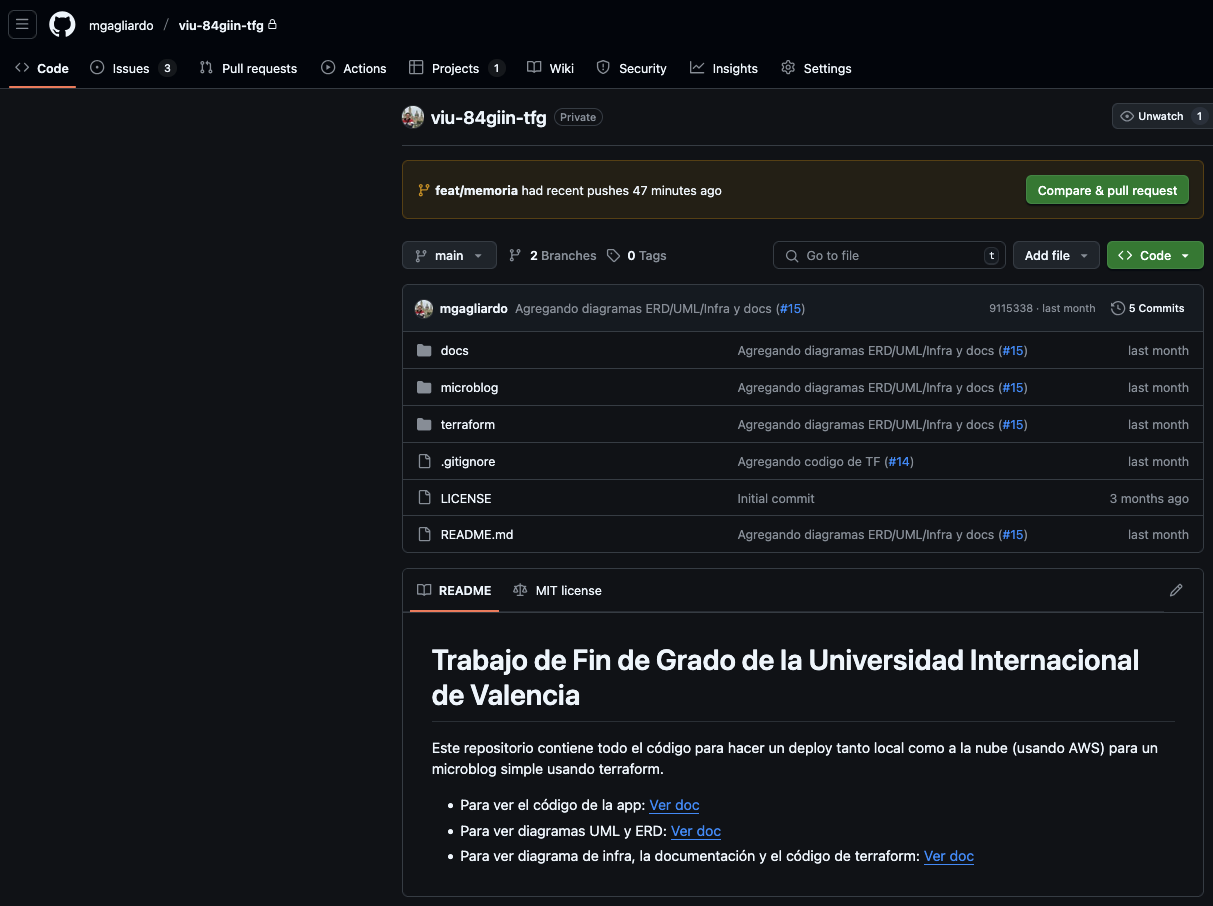


Figura 8. Repositorio del Trabajo de Final de Grado

Fuente: https://github.com/mgagliardo/viu-84giin-tfg/

Limitaciones

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

* La no implementación de un sistema formal de estimación para los issues del sprint como por ejemplo puntuaciones, por ejemplo cartas, Fibonacci u otro método.
* Al ser un proyecto unipersonal, algunas practices 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.

1. 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:

1. Sprint 1 - 07/04/2025 - 21/04/2025: Donde mayormente se diseño la aplicación y se realizó el primer draft en el desarrollo de la misma.
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 la misma.
4. Sprints 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.

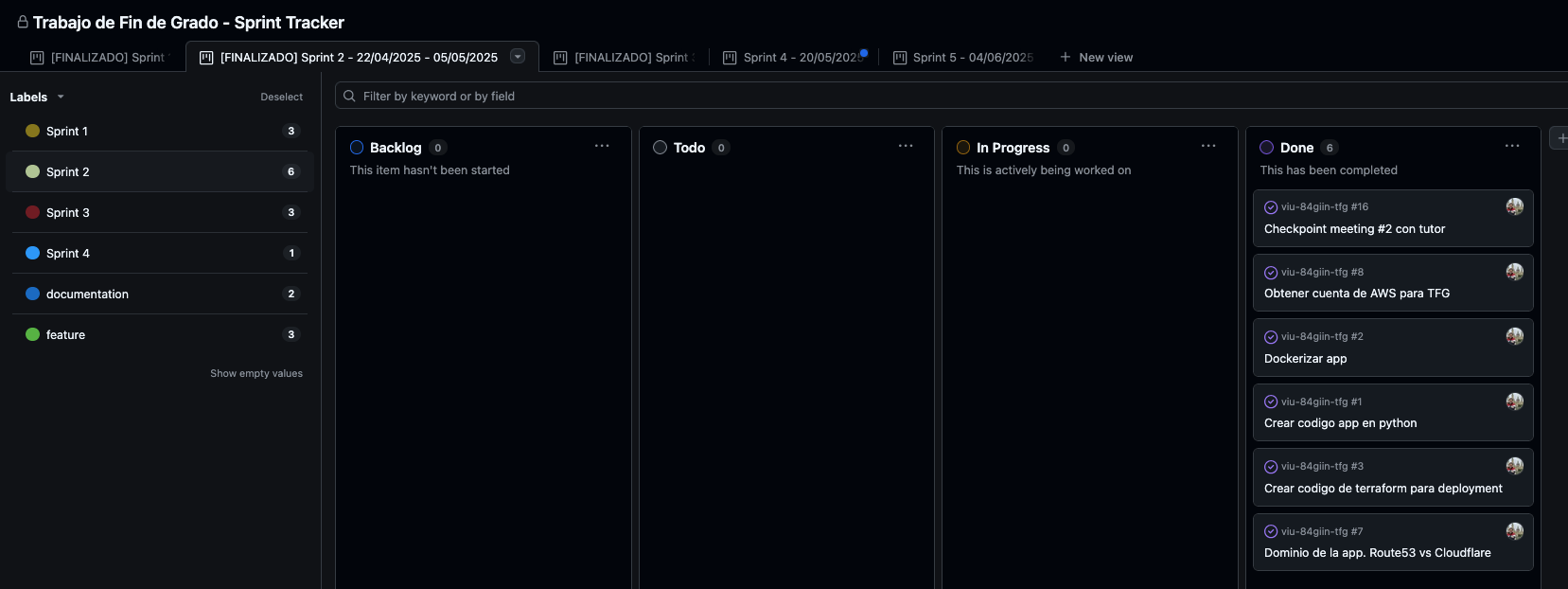


Figura 9. Sprint 2 del Trabajo de Final de Grado

Fuente: https://github.com/mgagliardo/viu-84giin-tfg

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.

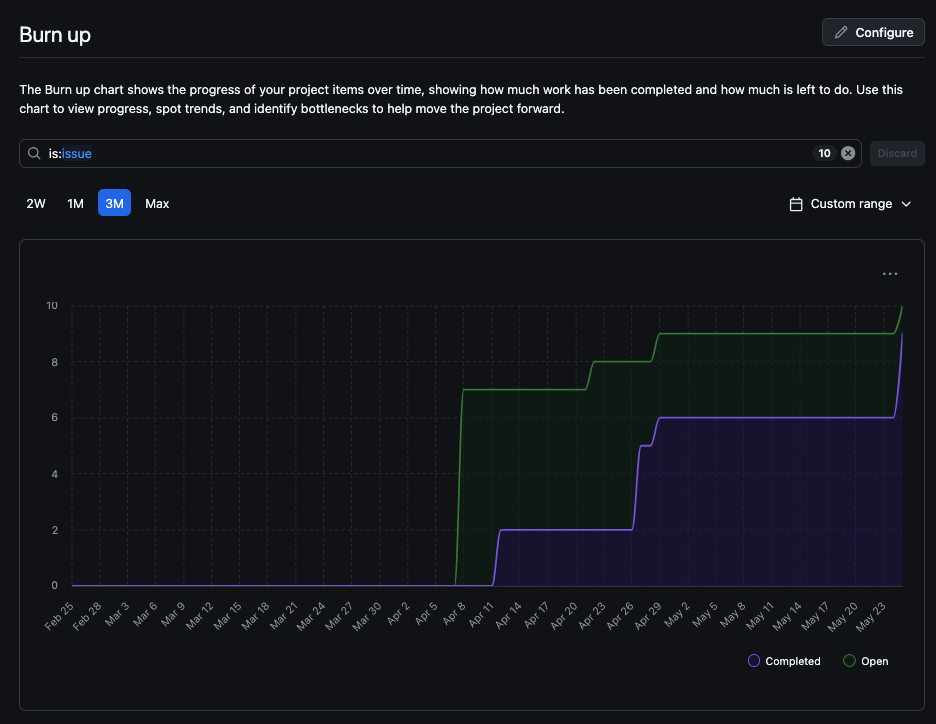


Figura 10. Burn up Chart del Sprint del Trabajo de Final de Grado

Fuente: https://github.com/mgagliardo/viu-84giin-tfg

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, dandosé 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**13, un framework basado en **MarkDown** que es un lenguaje de marcado para escribir documentos.

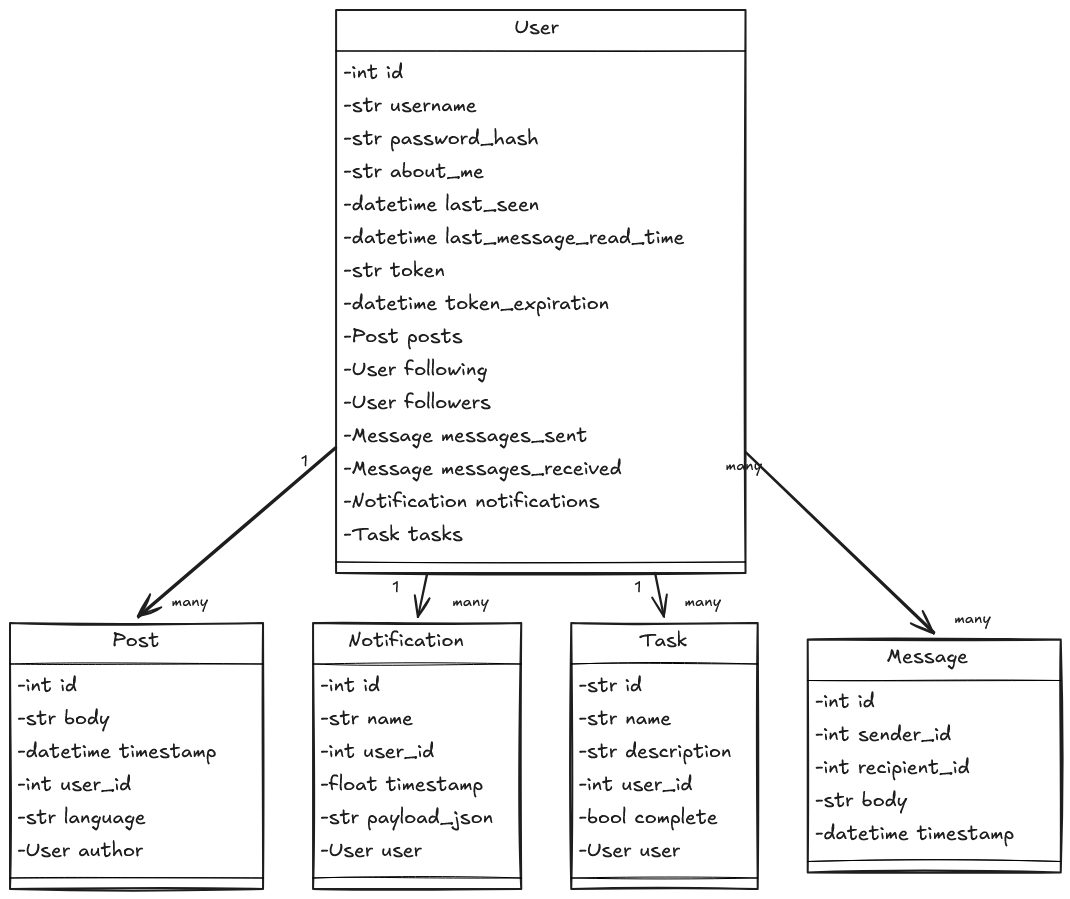


Figura 11. Diagrama de Clases de la aplicación Microblog

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 **chartdb13**, una herramienta de diseño de diagramas de bases de datos.

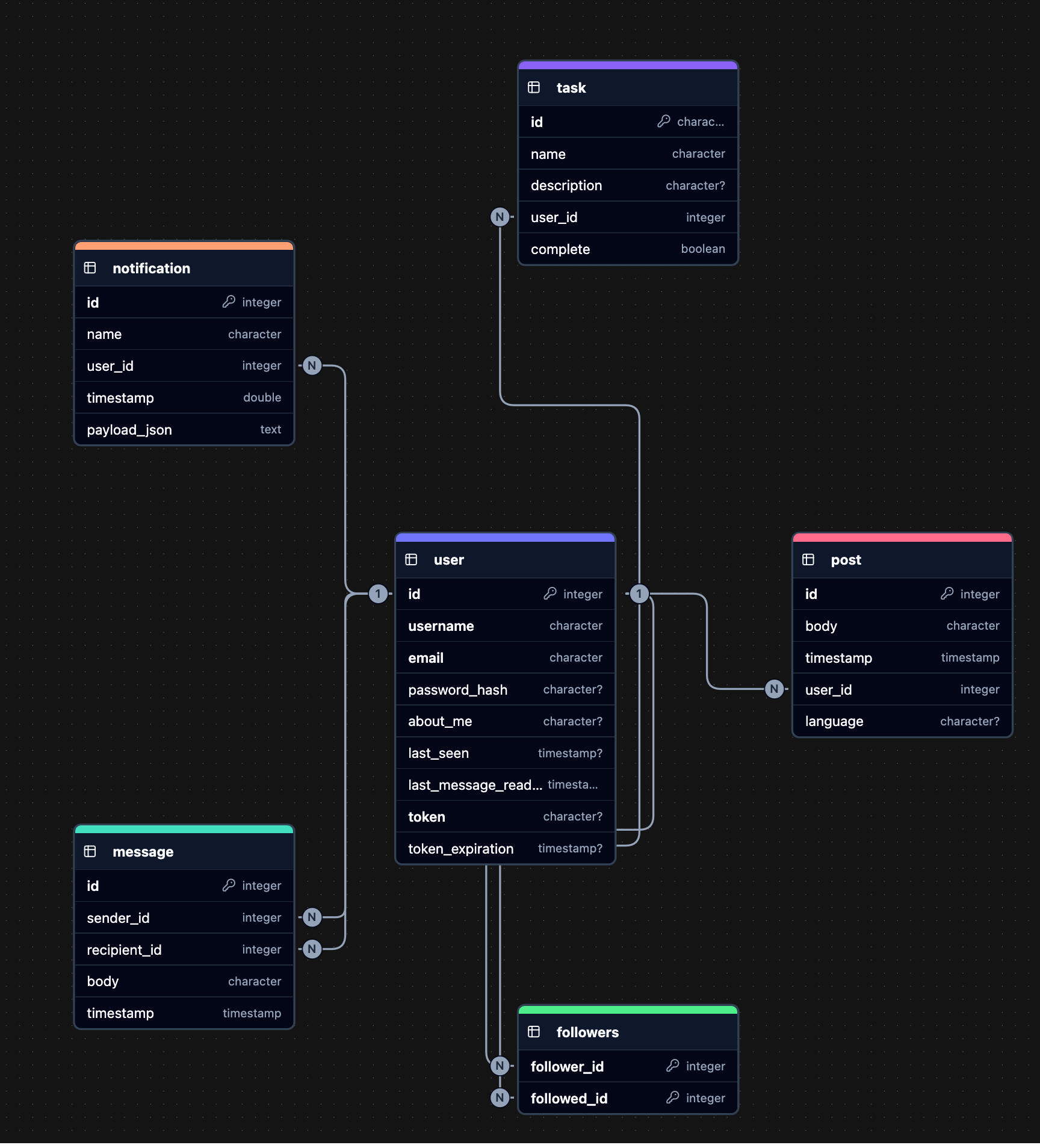
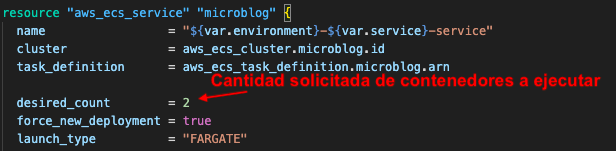
****

Figura 12. 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, y la carga del tráfico entrante será siempre balanceado gracias a que están conectadas a un balanceador de carga (ALB).



*Figura 13. Código de Terraform para explicitar la cantidad requerida de contenedores*

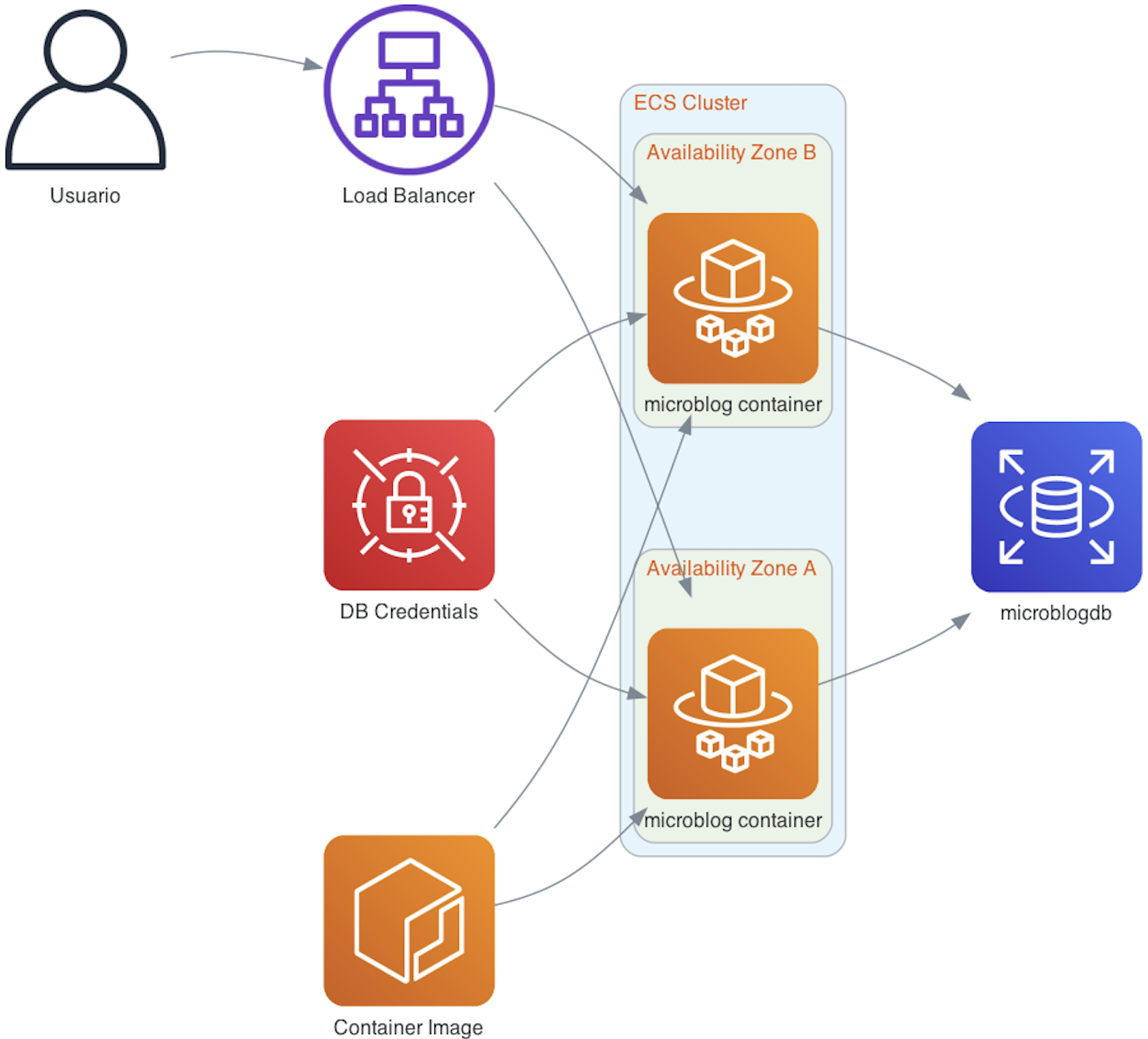
*Fuente: Elaboración Propia*

Para la conexión a la base de datos de PostgreSQL se optó por un servicio escalable como es RDS (Relational Database Service). Finalmente, las credenciales se guardan de manera segura y se pasan en runtime al contenedor a través de AWS SecretsManager simplemente a través de una variable de entorno.

Por último, la infraestructura implementa un modelo de acceso a red basado en el principio de mínimo privilegio:

* El Balanceador de Carga se ubica en subredes públicas, siendo el único componente con acceso directo a Internet al ser el punto de entrada para los usuarios.
* Los contenedores en Fargate y la base de datos en RDS se despliegan en subredes privadas sin acceso a Internet, ya que no requieren conectividad externa para su funcionamiento.

Esta arquitectura de red segmentada minimiza la superficie de ataque y mejora la postura de seguridad general de la aplicación.



*Figura 14. Diseño de la Infraestructura en AWS*

*Fuente: Elaboración Propia*

Desarrollo de la Infraestructura como Código

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 task y sprint** permitieron la validación y prueba de los componentes desplegados, facilitando la detección temprana de problemas y la optimización continua.

Este enfoque iterativo se beneficia enormemente de las características de Terraform, como la planificación de cambios (**terraform plan**) y la aplicación incremental (**terraform apply**). Además, la **modularización** del código Terraform, aunque no fue este caso dado que es un trabajo unipersonal, permite a los equipos trabajar en paralelo en diferentes componentes de la infraestructura, acelerando el desarrollo y mejorando la mantenibilidad del código a largo plazo.

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 **leidas** por Terraform a través de un **proveedor** **(provider)** de AWS, explicadas en la siguiente sección.



*Figura 15. Esquema de Autenticación para Terraform en AWS*

*Fuente: Elaboración Propia*

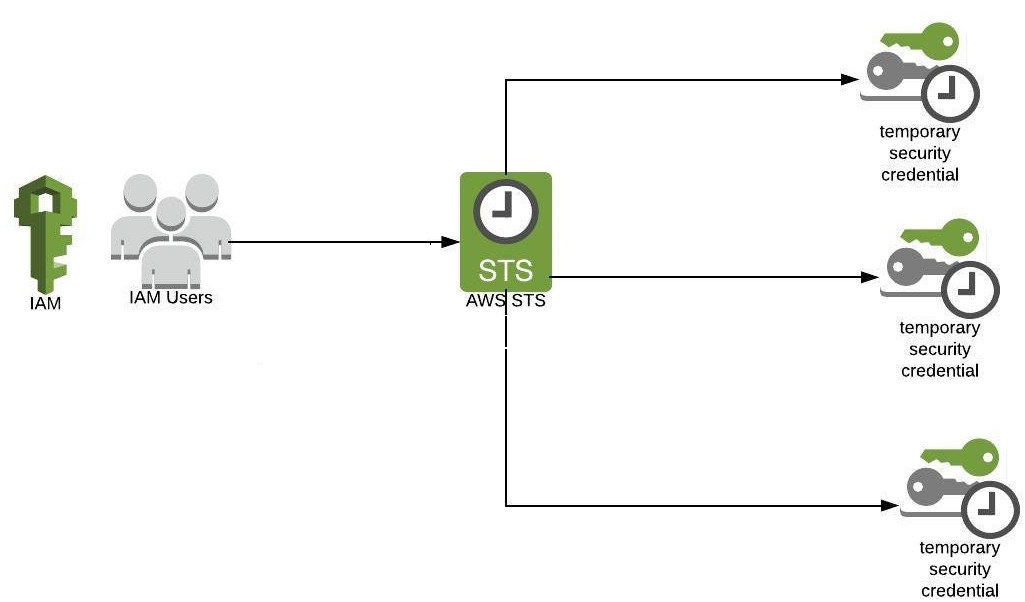
Los **beneficios** de este enfoque son:

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

Aún 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 mínimos14** 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).

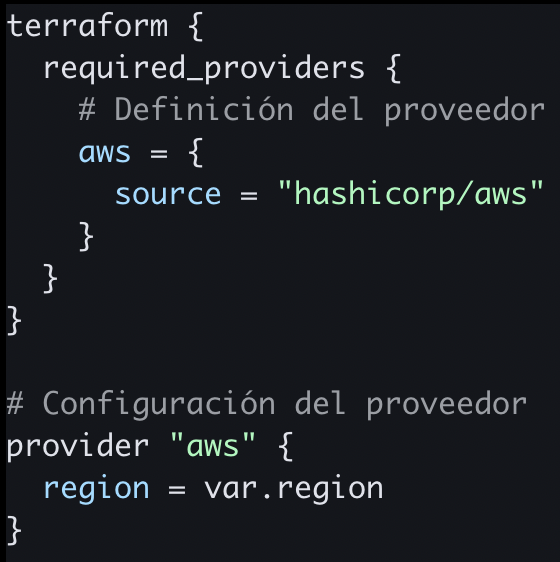
*Figura 16. Autenticación contra AWS utilizando roles de IAM*

*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 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.



*Figura 17. 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,** Terraformutilizará 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 sintática 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**,convirtiéndolaen 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.

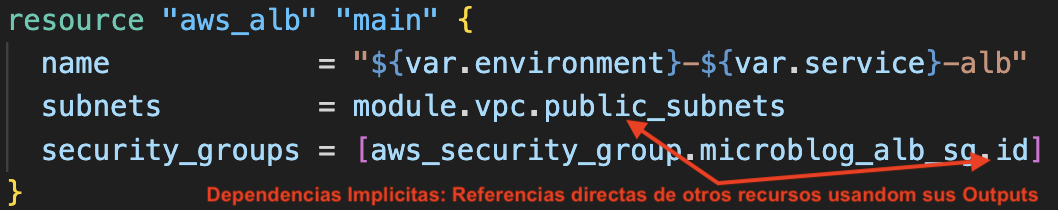
Gestión de dependencias en Terraform

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 unívoco 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:

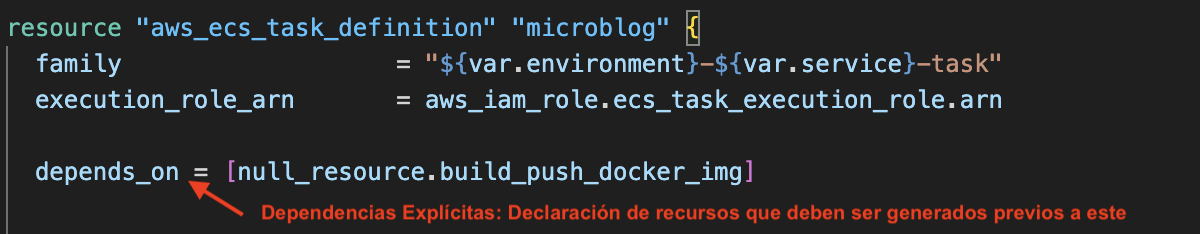
* **Implícitas:** A través de referencias directas entre recursos usando atributos de salida (**outputs**).



*Figura 18. Dependencias implícitas en Terraform*

*Fuente: Elaboración Propia*

* **Explícitas:** Mediante la directiva **depends\_on**, cuando la dependencia no es detectable automáticamente.



*Figura 19. Dependencias explícitas en Terraform*

*Fuente: Elaboración Propia*

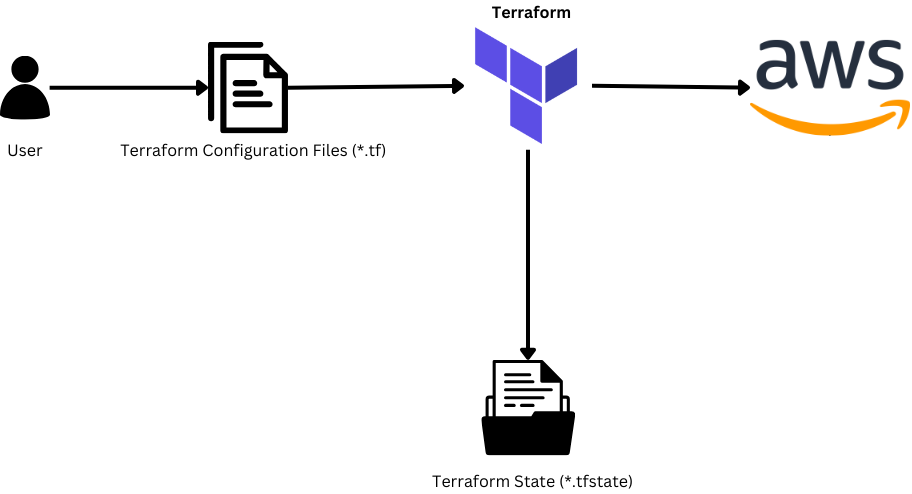
Terraform y el manejo de estado

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** para el control de acceso, asegura que el estado de la infraestructura permanezca **seguro, accesible y protegido contra pérdidas accidentales**.

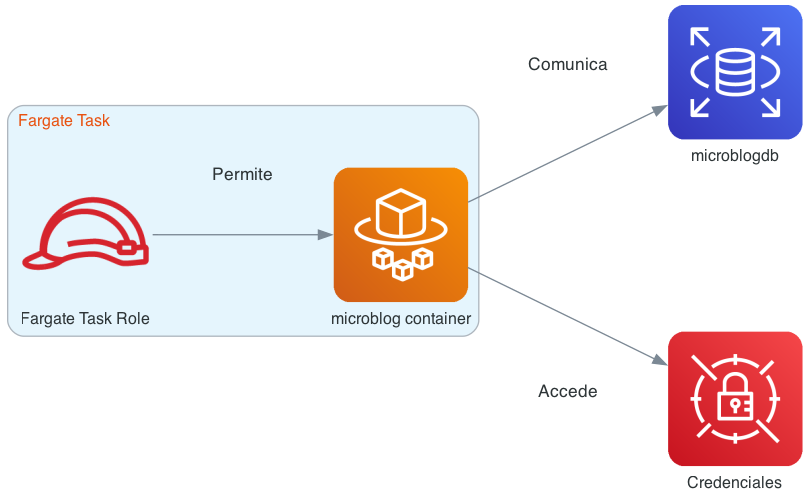


*Figura 20. Uso del remote tfstate file en Terraform*

*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, que es un servicio de claves seguras que las guarda en formato key/value. Esta implementación en particular permite inyectar 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 del contenedor.

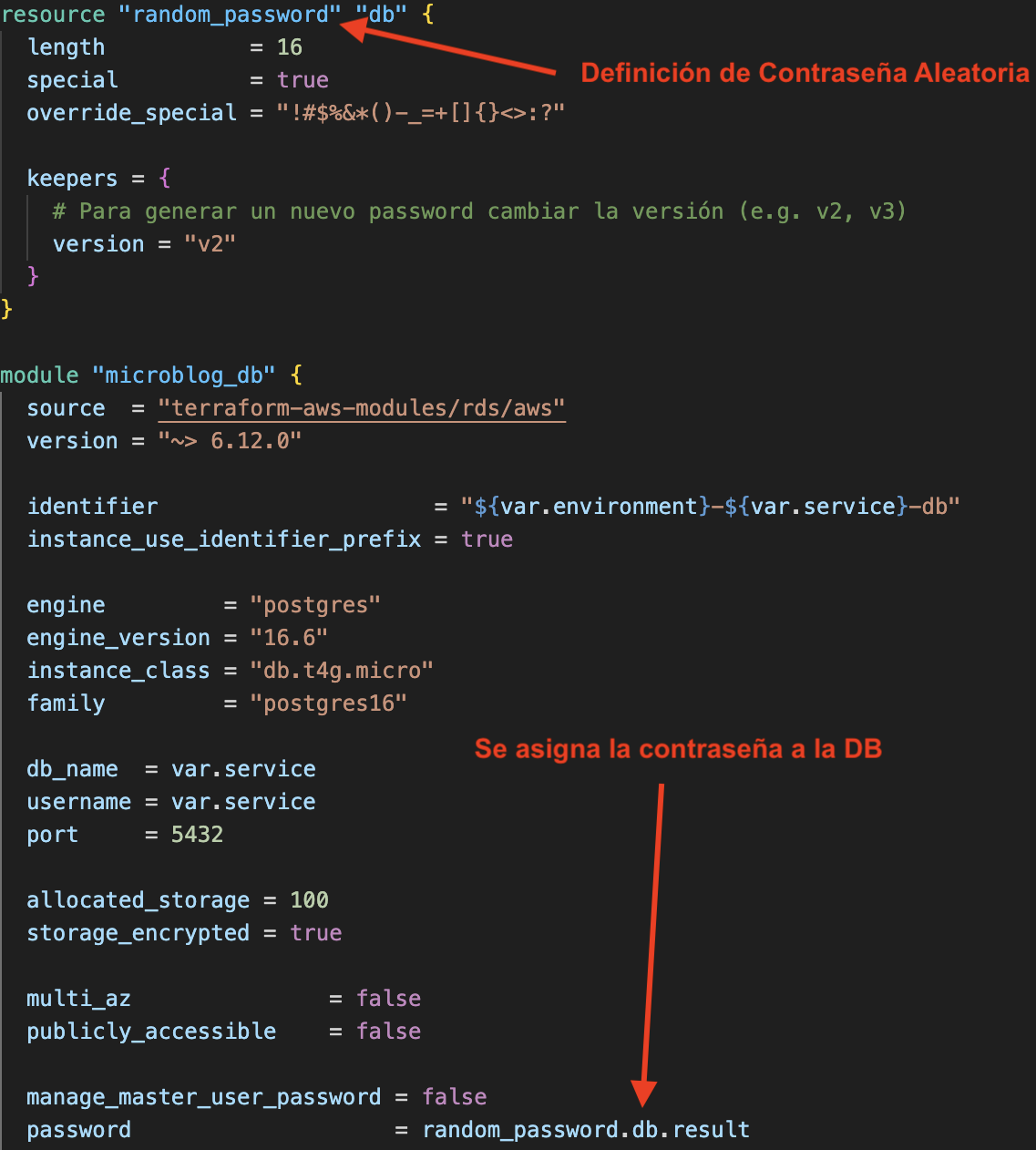


*Figura 21. Gestión de credenciales en la aplicación*

*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.



*Figura 22. Generación aleatoria de la contraseña de la DB en Terraform*

*Fuente: Elaboración Propia*

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



*Figura 23. Guardado de secretos en Secrets Manager via Terraform*

*Fuente: Elaboración Propia*

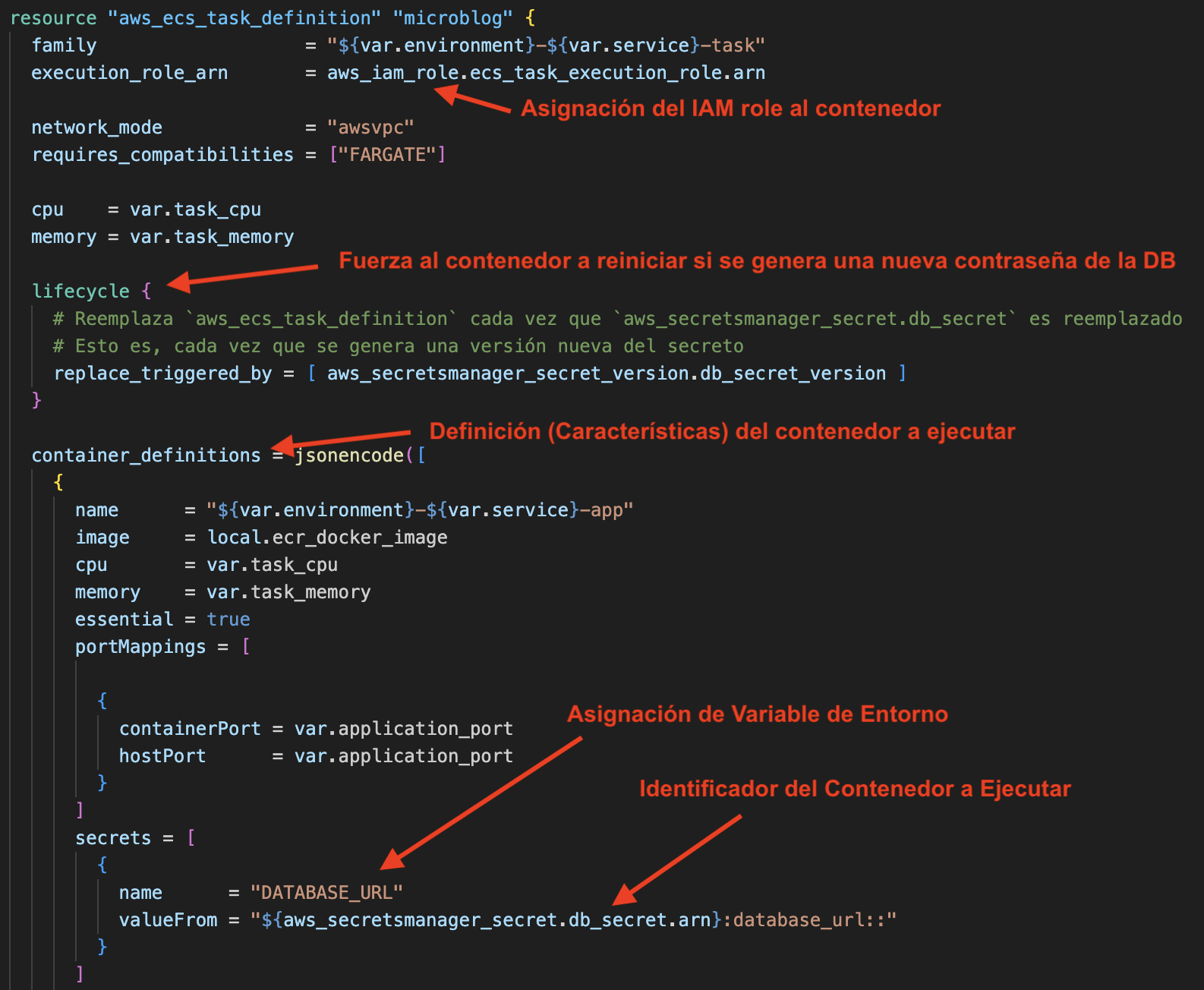
1. Siguiente, se define un IAM Role que es un tipo de ACL (Access Control List) específico para el contenedor a ejecutar que incluye una regla que le otorga **permisos de lectura limitados** específicamente al secreto requerido.



*Figura 24. Creación de IAM role y asignación de permisos en Terraform*

*Fuente: Elaboración Propia*

1. 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.



*Figura 25. 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 privilegio14** 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**.

1. 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**.

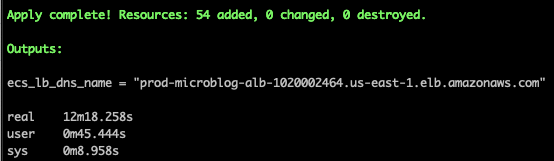
En primer lugar, porque con exactitud podemos saber exactamente los recursos se van a desplegar dada la utilidad del comando **terraform plan**, que nos muestra la cantidad y el tipo de recursos de infraestructura:



*Figura 26. Output del plan de 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.

****

*Figura 27. Tiempos de Creación de la Infraestructura via Terraform*

*Fuente: Elaboración Propia*

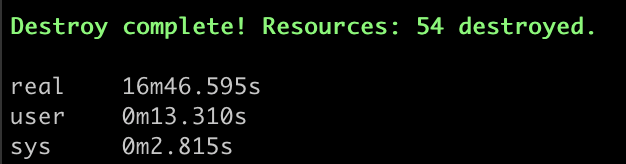
Para **validar** que la infraestructura creada es funcional, basta con localizar el output **ecs\_lb\_dns\_name**, que es la URL de ingreso a la aplicación.



*Figura 28. 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.



*Figura 29. 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** auto-generadaen 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.

Tabla 3. Comparación entre despliegues manuales y automatizados. Elaboración propia.

|  |  |  |
| --- | --- | --- |
| **Caracteristicas** | **Despliegue Manual** | **Despliegue con IaC** |
| **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 AWS15**.

1. **Gestión de Accesos, Secretos y Credenciales:**
   1. 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.
   2. Proporciona un sistema de rotación de credenciales que minimiza el riesgo de exposición y reduce la intervención humana.
   3. Integración con la aplicación de microblog mediante roles de IAM, evitando el uso de credenciales estáticas para el acceso a los recursos.
   4. Implementación del **principio de privilegio mínimo14** mediante roles de IAM específicos limitados para el servicio de microblog.
2. **Seguridad en la red:**
   1. Arquitectura de red segmentada utilizando subredes públicas y privadas.
   2. Segregación de accesos basados en grupos de seguridad, limitando el acceso a los recursos necesarios
   3. Acceso a internet restringido sólo a recursos que lo requieren explícitamente.
3. **Gestión Segura de la Infraestructura:**
   1. Estado de Terraform (tfstate) almacenado en S3 con encriptación en reposo.
   2. Versionado del estado para mantener un historial de cambios y facilitar reversiones (rollbacks).
   3. 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.

1. Conclusiones

Impacto en el Desarrollo de la Infraestructura

La implementación de la Infraestructura como Código via Terraform ha demostrado tener un impacto transformador en el ciclo de desarrollo y despliegue de la infraestructura.

Por un lado, la automatización del proceso de despliegue ha **reducido drásticamente el tiempo de provisión de infraestructura**. Esta mejora no solo representa un ahorro de tiempo significativo, sino que también **elimina tareas manuales repetitivas**, **permitiendo a los equipos técnicos enfocarse en actividades de mayor valor agregado**, **como el diseño de infraestructura siguiendo las mejores prácticas**.

Además, la **consistencia** garantizada entre los distintos ambientes reduce los errores de configuración que suelen surgir en despliegues manuales.

Eficiencia Operativa

La **capacidad de replicar ambientes** **completos** en cuestión de minutos no solo acelera el proceso de desarrollo, sino que también **democratiza el acceso a la infraestructura** al reducir la **dependencia de conocimientos específicos de AWS**. Esta estandarización **permite que equipos con diferentes niveles de experiencia puedan participar en el proceso de manera efectiva y segura**.

Quizás uno de los beneficios más significativos se ha observa en la **colaboración y el trabajo en equipo**: La **implementación de control de versiones** para la infraestructura permite rastrear cambios, mientras que el código actúa como documentación viva y actualizada. La posibilidad de realizar revisiones de código en los cambios de infraestructura, junto con la capacidad de trabajar en paralelo en diferentes componentes, ha llevado a una mejora sustancial en la calidad y la velocidad del desarrollo.

Por último, la **estructura de trabajo ágil en formato de sprints** facilitó la planificación y **seguimiento del progreso**, con **checkpoints regulares** que permitieron validar cada componente antes de avanzar al siguiente. La posibilidad de realizar revisiones de código en los cambios de infraestructura, junto con la capacidad de trabajar en paralelo en diferentes componentes, ha llevado a una mejora sustancial en la calidad y la velocidad del desarrollo.

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. 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.** Actualmente, la infraestructura mantiene un número fijo de contenedores, lo cual podría resultar insuficiente en momentos de alta demanda o excesivo en períodos de baja actividad. Un sistema de auto-escalado permitiría ajustar automáticamente la cantidad de contenedores en función de la carga de trabajo, optimizando tanto el rendimiento como los costos operativos.

**Por último**, la integración con sistemas de **Integración Continua y Despliegue Continuo (CI/CD)**, como **GitHub Actions16**, representaría un avance significativo. Esta integración automatizaría completamente el proceso de despliegue de infraestructura, permitiendo validaciones automáticas de código, pruebas de seguridad y despliegues controlados, todo dentro de un pipeline unificado y consistente.

Estas mejoras propuestas no solo aumentarían la robustez de la solución, sino que también la acercarían más a los estándares de la industria para **entornos de producción de gran escala**.

Reflexión Final

Todos los cambios mencionados en el Desarrollo de la infraestructura y la Eficiencia Operativa contribuyen a establecer una **cultura DevOps17** más madura, donde los cambios de infraestructura se gestionan con el mismo rigor y disciplina que los cambios en el software. Esta transformación cultural es fundamental para mantener la agilidad y calidad en el desarrollo de software moderno.

1. Referencias
2. The NIST Definition of Cloud Computing. (2011). NIST SP 800-145. https://csrc.nist.gov/pubs/sp/800/145/final
3. Andreas Wittig and Michael Wittig. (2023). Amazon Web Services in Action (3rd ed.). Manning Publications.

https://www.manning.com/books/amazon-web-services-in-action-third-edition

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

https://www.oreilly.com/library/view/infrastructure-as-code/9781098150341/

1. Amazon Web Services. (2021). Our Origins.

https://aws.amazon.com/es/about-aws/our-origins/

1. Amazon Web Services. (2023). The history and future roadmap of the AWS CloudFormation Registry. https://aws.amazon.com/es/blogs/devops/cloudformation-coverage/
2. Hashicorp. (2021). The Story of Hashicorp Terraform with Mitchell Hashimoto. https://www.hashicorp.com/en/resources/the-story-of-hashicorp-terraform-with-mitchell-hashimoto
3. 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

1. 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
2. B. W. Kernighan and M. D. McIlroy. 1979. UNIX Time-Sharing System: UNIX Programmer's Manual (7th ed.). Vol. 1. Bell Telephone Laboratories Incorporated, Murray Hill, NJ.
3. What is a Container? (2021). Docker

https://www.docker.com/resources/what-container/

1. Visual Studio Marketplace. Hashicorp Terraform Plugin

https://marketplace.visualstudio.com/items?itemName=HashiCorp.terraform

1. Visual Studio Marketplace. Python Plugin

https://marketplace.visualstudio.com/items?itemName=ms-python.python

1. Welcome to ChartDB (2025). ChartDB

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

1. Prepare for least-privilege permissions (2025). AWS

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

1. AWS Well-Architected Framework

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

1. Understanding Github Actions (2025). Github

https://docs.github.com/en/actions/about-github-actions/understanding-github-actions

1. Rouan Wilsenach (2015). Dev Ops Culture. Martin Fowler Blog

https://martinfowler.com/bliki/DevOpsCulture.html