Mario Robres Royuela

AWS-AUTODEPLOY

Una herramienta para la automatización de despliegues de infraestructuras en Amazon Web Services con GitHub Actions y Terraform

TRABAJO DE FIN DE GRADO

Grado de Ingeniería Informática

dirigido por David Isern Alarcón

UNIVERSITAT ROVIRA I VIRGILI

Tarragona

2024

Summary

AWS-Autodeploy is an innovative tool designed to automate the deployment of infrastructures on Amazon Web Services (AWS) using GitHub Actions and Terraform. The main goal of this project is to simplify and streamline cloud migration, making it accessible to users of all experience levels. AWS-Autodeploy features an intuitive and minimalist graphical interface that allows users to deploy and manage multiple services and architectures simultaneously, eliminating the need for extensive coding or navigating between multiple screens.

This tool significantly enhances the productivity of professionals and offers an affordable solution for companies lacking dedicated cloud engineers. Additionally, it serves as an ideal entry point for beginners in Cloud technologies. AWS-Autodeploy includes robust and customizable functionalities, along with a modular architecture that facilitates the integration and expansion of new AWS services and complementary technologies like Ansible. The combination of innovative design, cutting-edge technologies, and multiple functionalities positions AWS-Autodeploy as a highly attractive tool for both professional and educational environments

Resum

AWS-Autodeploy és una eina innovadora dissenyada per automatitzar el desplegament d'infraestructures a Amazon Web Services (AWS) utilitzant GitHub Actions i Terraform. L'objectiu principal d'aquest projecte és simplificar i agilitzar la migració al núvol i fer-lo accessible per a usuaris de tots els nivells d'experiència. AWS-Autodeploy incorpora una interfície gràfica intuïtiva i minimalista que permet als usuaris implementar i administrar múltiples serveis i arquitectures simultàniament, eliminant la necessitat de codificació extensiva o de navegar entre múltiples pantalles.

Aquesta eina millora significativament la productivitat dels professionals i ofereix una solució assequible per a empreses que no tenen enginyers dedicats al núvol. A més, es posiciona com a punt d'entrada ideal per a principiants en tecnologies *Cloud*. AWS-Autodeploy inclou funcionalitats fermes i personalitzables, juntament amb una arquitectura modular que facilita la integració i expansió de nous serveis d'AWS i tecnologires complementaris com Ansible. La combinació d'un disseny innovador, l'ús de tecnologies capdavanteres i les seves múltiples funcionalitats posicionen AWS-Autodeploy com una eina molt atractiva tant per a l'àmbit professional com educatiu.

Resumen

AWS-Autodeploy es una herramienta innovadora diseñada para automatizar el despliegue de infraestructuras en Amazon Web Services (AWS) utilizando GitHub Actions y Terraform. El objetivo principal de este proyecto es simplificar y agilizar la migración a la nube, haciéndola accesible para usuarios de todos los niveles de experiencia. AWS-Autodeploy incorpora una interfaz gráfica intuitiva y minimalista que permite a los usuarios implementar y administrar múltiples servicios y arquitecturas simultáneamente, eliminando la necesidad de codificación extensiva o de navegar entre múltiples pantallas.

Esta herramienta mejora significativamente la productividad de los profesionales y ofrece una solución asequible para empresas que carecen de ingenieros dedicados a la nube. Además, se posiciona como un punto de entrada ideal para principiantes en tecnologías *Cloud*. AWS-Autodeploy incluye funcionalidades robustas y personalizables, junto con una arquitectura modular que facilita la integración y expansión de nuevos servicios de AWS y tecnologías complementarias como Ansible. La combinación de un diseño innovador, el uso de tecnologías punteras y sus múltiples funcionalidades posicionan a AWS-Autodeploy como una herramienta muy atractiva tanto para el ámbito profesional como educativo.

Índice general

Índice de tablas

Índice de figuras

1. Introducción

Hoy en día, el *Cloud Computing* se ha convertido en una parte fundamental del mundo tecnológico, cambiando la forma en que las organizaciones, empresas y personas gestionan sus recursos informáticos. Básicamente, el término *Cloud Computing* se refiere a ofrecer servicios informáticos a través de Internet, lo que permite acceder a una amplia gama de recursos como almacenamiento, servidores, bases de datos y redes sin necesidad de poseer la infraestructura física donde se ejecutan. Esto proporciona una infraestructura flexible y escalable que puede adaptarse a las necesidades cambiantes de sus clientes.

Este modelo se basa en la idea fundamental de deslocalizar los recursos informáticos, permitiendo a los usuarios acceder a ellos desde cualquier lugar y en cualquier momento, siempre y cuando tengan conexión a Internet. Además, al estar distribuidos por el mundo, los usuarios no conocen la ubicación física real de la infraestructura donde se ejecutan sus recursos.

Con tantos beneficios y ventajas, es comprensible que en la actualidad muchas empresas estén contemplando migrar sus operaciones al entorno de la nube. La migración al *Cloud* supone, simplemente, el traslado de los recursos informáticos, como datos, aplicaciones y servicios, desde los sistemas locales o tradicionales a plataformas en la nube, directamente en su infraestructura o a través de servicios.

Sin embargo, la migración al Cloud tiene una alta complejidad y requiere de profesionales con una gran cantidad de conocimientos técnicos y prácticos, los *Cloud Engineer*. Es aquí donde entra en juego AWS-Autodeploy, la herramienta desarrollada como Trabajo de Fin de Grado diseñada para facilitar la migración al Cloud tanto para principiantes, profesionales experimentados o empresas.

- Para los principiantes, AWS-Autodeploy ofrece una introducción amigable al mundo del *Cloud Computing*. Uno de sus mayores beneficios es que no se necesita experiencia en programación ni una infraestructura previa. Puede ser utilizada en cualquier dispositivo, ya sea móvil o ordenador, y en cualquier sistema operativo, lo que permite a los usuarios iniciar su viaje hacia el *Cloud Computing* de manera rápida y sin barreras técnicas.
- Para los profesionales, AWS-Autodeploy mejora significativamente la productividad al permitirles lanzar uno o múltiples servicios, así como arquitecturas completas, con varias instancias de varios servicios de manera simultánea. Esto se logra sin la necesidad de navegar entre diferentes pantallas o escribir código de infraestructura (*IaC*, por sus siglas en inglés). Con esta capacidad de simplificar la árdua tarea de lanzar instancias, AWS-Autodeploy libera tiempo para que los profesionales se centren en las tareas estratégicas y de personalización de los servicios lanzados, en vez de gastar tiempo en los despliegues.
- Para las empresas, que no deseen o no tengan un *Cloud Engineer*, AWS-Autodeploy ofrece una solución eficaz y accesible. Al ser una herramienta que cualquier persona con conocimientos básicos de informática puede utilizar, elimina la necesidad de contratar personal adicional o externalizar la migración a proveedores costosos.

En resumen, AWS-Autodeploy se presenta como una solución versátil y poderosa en el ámbito del *Cloud Computing*, allanando el camino para personas menos experimentadas y ofreciendo un herramienta potente y extensible para aquellos con un perfil más profesional.

1.1. Objetivos del proyecto

El objetivo principal de este proyecto es desarrollar y poner en práctica AWS-Autodeploy como una herramienta efectiva para facilitar la migración al entorno de la nube, específicamente en el contexto de Amazon Web Services (AWS). Así mismo, para conseguir una correcta implementación de AWS-Autodeploy, se especifican los siguientes objetivos:

- Diseño e implementación de una herramienta intuitiva y fácil de usar, para permitir a usuarios con diferentes niveles de experiencia usar la herramienta. Tanto en el propio repositorio, definiendo un ciclo de vida entendedor, como en la página web, con una interfaz simple y minimalista.

- Desarrollo de funcionalidades robustas, parametrizables y versátiles, para permitir a los usuarios lanzar uno o múltiples servicios, con una o múltiples instancias de cada uno de ellos, en AWS de manera simultánea.

- Establecer una arquitectura modular y flexible para AWS-Autodeploy, que facilite la incorporación de nuevos servicios de AWS. Los módulos, clases y herramientas deben estar bien diferenciadas, con una arquitectura estructurada y con posibilidad de extenderla.

- Probar y validar la eficacia de AWS-Autodeploy, asegurando que la herramienta cumpla con los estándares de calidad y fiabilidad. Tanto para servicios aislados, como en arquitecturas complejas. Esto permitirá valorar y entender el potencial de la herramienta.

1.2. Objetivos formativos

En el aspecto formativo del alumno, los objetivos de la construcción y creación de AWS-Autodeploy puedan ofrecer son:

- Adquirir experiencia en las tres facetas principales que debe tener un *Cloud Engineer*:
    - En la faceta de DevOps, desarrollar habilidades en la creación de flujos de trabajo mediante el uso de herramientas como GitHub Actions y la gestión de repositorios para optimizar el desarrollo y despliegue de aplicaciones en la nube.
    - En la faceta de arquitecto, diseñar plantillas de servicios y definir arquitecturas que puedan implementarse de manera efectiva en entornos de producción.
    - En la faceta de desarrollador, dominar el uso de Infraestructura como Código (*IaC*) mediante tecnologías como Terraform, así como la creación y gestión de la herramienta misma, incluyendo el desarrollo de *scripts*, su flexibilidad y la gestión de credenciales de forma segura.
- Contribuir al mundo con una herramienta *Open Source*, con el propósito de solucionar problemas y necesidades del sector. El código y la documentación serán públicos, se emplean tecnologías de código abierto y se fomenta la colaboración entre usuarios para mejorar continuamente la herramienta.
- Desarrollar una herramienta agradable a principiantes. Para ello se construirá una página web destinada a ayudar a nuevos usuarios a descubrir y utilizar el *Cloud Computing* en sus vidas. Además, se contempla esta plataforma como una herramienta potencialmente útil en entornos educativos, facilitando el aprendizaje y comprensión del *Cloud Computing*.

2. Descripción General

Para comprender la herramienta AWS-Autodeploy, es esencial establecer una base sólida en los principios fundamentales del *Cloud Computing*, así como en las tecnologías utilizadas como GitHub, GitHub Actions, *IaC*, Terraform y Ruby. En esta sección, se explican los conceptos básicos de estas tecnologías, su funcionamiento y las ventajas que aportan al proyecto, así como otras alternativas.

2.1. Conceptos Básicos del Cloud Computing

2.1.1. Desarrollo Cloud vs Tradicional

En el **desarrollo tradicional**, las empresas crean aplicaciones que se ejecutarán en la infraestructura local de la empresa (en sus servidores). Esto conlleva una **gran inversión inicial** tanto económica como temporal, si la empresa no dispone de los medios *hardware* y *software* necesarios. Por ejemplo, la adquisición de servidores físicos, su instalación y puesta a punto, pueden llevar semanas o incluso meses, dependiendo de las restricciones e inconvenientes. Además, la escalabilidad en este entorno es **limitada**. Por ejemplo, si una empresa experimenta un gran aumento en número de usuarios en su aplicación (aumento de demanda), es complicado y costoso para la empresa, agregar o actualizar los servidores para que puedan manejar esta carga adicional. Esta falta de flexibilidad puede resultar en la pérdida de oportunidades o malas experiencias de uso para los usuarios.

En términos de **seguridad**, las empresas deben implementar y mantener sus **propias medidas** de seguridad. Esto incluye, entre otros, la protección contra ataques cibernéticos, la gestión de accesos, el cifrado de datos y la realización de copias de seguridad periódicas.

En cuanto al proceso de desarrollo, los equipos deben contar con profesionales altamente capacitados en programación y administración de sistemas. Cada empresa debe **configurar** y **gestionar** su propia infraestructura. Por ejemplo, deberán gestionar los requisitos del sistema operativo, la carga de trabajo de la aplicación, la actualización de componentes… Así como el propio **mantenimiento** de la infraestructura, como climatización, reparación de elementos dañados y la gestión de la infraestructura eléctrica y de red. Estos procesos consumen tiempo y recursos que podrían destinarse a otras actividades, provocando a la larga una menor eficiencia y rentabilidad. El **rendimiento** en la infraestructura local puede ser **inconsistente** debido a las limitaciones físicas y técnicas de los servidores disponibles. Las empresas pueden experimentar problemas de latencia, tiempos de respuesta lentos y otros inconvenientes que afectan la experiencia del usuario, especialmente en momentos de alta demanda.

En el **desarrollo *cloud*** [2][3]**,** las empresas crean aplicaciones que se ejecutarán en infraestructuras remotas en la nube proporcionadas por proveedores *cloud*, como Amazon Web Services (AWS), Microsoft Azure o Google Cloud Platform (GCP). Este enfoque elimina la necesidad de una gran inversión inicial, así como una **reducción** en el tiempo que se tarda en poner en marcha la aplicación. Por ejemplo, en vez de perder semanas o meses en tener la infraestructura lista, las empresas pueden **aprovisionar** de manera instantánea servidores virtuales. El tiempo y recursos que se ahorran, se pueden dedicar a mejorar la aplicación. Además, la **escalabilidad** en este entorno es casi ilimitada. Este permite a las empresas adaptarse a cargas de usuarios variantes, con picos de mucha demanda y otros de muy baja, teniendo habilitados en cada momento, el número de servidores virtuales necesarios.

En términos de **seguridad**, los proveedores cloud **invierten** significativamente en medidas de seguridad avanzadas, incluyendo protección contra ataques DDoS, cifrado de datos en tránsito y en reposo, y sistemas avanzados de detección y prevención de intrusiones. Además, estos proveedores suelen cumplir con estrictos **estándares** y **certificaciones** de seguridad, lo que garantiza un alto nivel de protección para las aplicaciones y datos de las empresas.

En cuanto al proceso de desarrollo [4], los equipos pueden enfocarse más en la lógica de la aplicación en lugar de preocuparse por la gestión, configuración y mantenimiento de la infraestructura. Los propios **proveedores *cloud*** se encargan de administrar la infraestructura. Asimismo, también ofrecen una gran gama de servicios que simplifican el desarrollo e implementación de la aplicación. Por ejemplo, AWS ofrece servicios para ejecutar aplicaciones en contenedores, para la creación de bases de datos, etc. El **rendimiento** en el entorno *cloud* suele ser **superior** debido a la capacidad de los proveedores para optimizar la distribución de cargas y utilizar tecnologías avanzadas para minimizar la latencia y maximizar la velocidad de procesamiento. Esto asegura que las aplicaciones funcionen de manera eficiente, incluso durante los **picos** de **demanda**, proporcionando una experiencia de usuario más fluida y rápida.

Comparativa Cloud vs Tradicional		
Característica	**Tradicional**	**Cloud**
Conocimiento Programación	Alto	Bajo
Control/flexibilidad	Alto/Bajo	Medio/Alto
Infraestructura	Local	Nube
Costos iniciales	Altos	Bajos
Costos a largo plazo	Variables	Variables
Escalabilidad	Baja	Alta
Rendimiento	Inconsistente	Dinámico
Seguridad	Propia	Proveedor
Tiempo de implementación	Largo	Corto

Tabla 1. Tabla comparativa entre desarrollo *Cloud* y Tradicional.

2.1.2. Arquitecturas

En este apartado se hará una comparativa entre las arquitecturas de *software* más populares en el desarrollo de aplicaciones. En esta estructura, se definen como se dividen las responsabilidades entre los diferentes módulos o componentes del sistema, como se comunicarán entre sí y qué se hará para conseguir cumplir con los requisitos funcionales y no funcionales del sistema [1][6].

2.1.2.1. Arquitectura Monolítica

En la ingeniería del *software*, el término "monolito" hace referencia a una única unidad que no puede ser dividida o separada en partes más pequeñas, es decir, una unidad indivisible. En la **arquitectura monolítica**, esta unidad se refiere a la aplicación completa, donde todos sus componentes, interfaz gráfica, bases de datos y lógica de aplicación, se integran como una única unidad o paquete (un único ejecutable en una única plataforma).

Esta metodología fue la **precursora** de todas las demás y la más utilizada en los primeros días de la informática. Sin embargo, este enfoque tan sencillo y popular, conlleva una serie de limitaciones. Todos los **componentes** están **relacionados entre sí** (acoplados) y es complicado modificar o actualizar los componentes de forma individual. Por otro lado, cuando el código está listo, se **despliega** en su **totalidad** y se comprueban todas sus partes simultáneamente.

2.1.2.2. Arquitectura SOA (orientada a servidores)

Tras unos años, surgió la **arquitectura orientada a servidores** (***Server-Oriented Architecture***) como una evolución de la arquitectura monolítica para sobrepasar sus limitaciones y mejorar otros aspectos del desarrollo. En esta arquitectura, la aplicación se divide en diferentes módulos o servicios relacionados, cada uno de los cuales se ejecuta en su propio servidor, mejorando la escalabilidad y modularidad en comparación a la arquitectura anterior.

El mayor beneficio de esta metodología es la **distribución de la carga** de trabajo en **múltiples servidores**. Por un lado, agrega redundancia como fiabilidad en caso de fallos. Si un servidor falla, el impacto es menor ya que el resto pueden seguir trabajando. Por otro lado, esto permite una mejor utilización de los recursos ya que cada servidor estará dedicado y optimizado para una tarea específica.

Sin embargo, este enfoque trae otro tipo de limitaciones, y la principal es la **comunicación** entre los diferentes **servicios** y **servidores**. De igual manera, también deberá **validar** todos los ***inputs*** que se mandan entre los diferentes módulos durante estas comunicaciones. Cuanto más compleja sea la aplicación y más módulos contenga, más carga tendrá este servicio, provocando cuellos de botella y tiempos de respuesta más lentos.

2.1.2.3. Arquitectura de Microservicios

La **arquitectura de microservicios** es una especie de arquitectura orientada a servidores, con protocolos más ligeros (*lightweights*) y completamente opuesta a la arquitectura monolítica. En esta arquitectura, la aplicación se divide en **múltiples componentes** (servicios) independientes, cada uno de ellos con una tarea específica y recursos únicos, que serán unidos mediante APIs.

Dividir la aplicación en módulos pequeños e independientes, tiene como beneficio una mejor escalabilidad y flexibilidad. Cada microservicio puede ser desarrollado, desplegado y escalado de forma **independiente**, lo que permite y facilita a los equipos trabajar de forma **paralela** y sin necesidad de estar sincronizados de manera continua. Además, esta separación mejora la resistencia a fallos y el tiempo de recuperación, ya que es posible **aislar** y **solucionar** el servicio que ha fallado sin que afecte al funcionamiento del resto.

2.1.2.4. Arquitectura Serverless

La aparición del *Cloud* trajo con ella un nuevo tipo de arquitectura orientada a servidores, la **arquitectura *serverless***. Esta arquitectura intenta aprovechar al máximo las cualidades y ventajas que ofrece el *Cloud Computing*. Igualmente, las aplicaciones seguirán ejecutándose en servidores físicos o virtuales, pero esta gestión tiene una capa de abstracción. Por un lado, los servidores serán gestionados por el proveedor *cloud*. Por otro lado, los desarrolladores se centran más en la lógica de la aplicación, ya que no deben preocuparse por la gestión y el tipo de infraestructura donde se ejecutarán las aplicaciones.

La no gestión de la infraestructura donde se ejecutará la aplicación, ofrece también una **escalabilidad automática**. Los servidores escalan de manera automática en función a la carga de trabajo. Además, sólo se **paga** por los recursos y por el tiempo de uso de los mismos que realmente se usen, provocando una reducción **significativa** de los **costos**.

El funcionamiento de esta arquitectura *serverless* se basa en el concepto de **funciones** o **piezas** de código que se ejecutan en respuesta a **eventos** específicos. Cuando se produce un evento, como una petición HTTP entrante, el proveedor cloud se encarga de aprovisionar los **recursos necesarios**, como pueden ser servidores, memoria o red, para que la aplicación pueda llevar a cabo la función escrita para ese evento. Una vez completada la ejecución, estos recursos se **liberan** de manera automática.

A modo de resumen, a continuación se muestra una tabla resumen con las cuatro arquitecturas:

Arquitecturas				
Característica	**Monolítica**	**Server-oriented**	**Microservicios**	**Serverless**
Escalabilidad	Limitada	Mejorada	Alta	Máxima
Complejidad	Baja	Media	Alta	Alta
Implementación	Fácil	Media	Media	Difícil
Flexibilidad	Baja	Media	Alta	Alta
Costos	Variables	Variables	Variables	Basados en uso

Tabla 2. Tabla comparativa entre arquitecturas de *software*.

2.1.3. Modelos de servicio

En este apartado se hará una introducción a los modelos de servicio en la nube, para entender cómo se gestionan y despliegan los recursos tecnológicos en el entorno *cloud*. Estos modelos permiten elegir diferentes niveles de control y responsabilidad sobre la infraestructura y aplicaciones, optimizando y mejorando el rendimiento y su despliegue [7].

2.1.3.1. Modelo IaaS (Infraestructura como Servicio)

IaaS proporciona acceso a recursos básicos de computación como servidores, almacenamiento y redes. Con *IaaS*, se pueden crear y gestionar aplicaciones y sistemas propios. Los usuarios tienen un control significativo sobre estos recursos, permitiendo configurarlos y personalizarlos según sus necesidades específicas. Sin embargo, implica la responsabilidad total de gestionar, mantener y asegurar todos los componentes, lo que requiere un alto nivel de conocimiento técnico. Ejemplos de *IaaS* son Amazon Web Services (AWS) y Microsoft Azure.

2.1.3.2. Modelo PaaS (Plataforma como Servicio)

PaaS ofrece un entorno completo para desarrollar y desplegar aplicaciones en la nube, incluyendo herramientas de desarrollo, bases de datos y sistemas operativos, sin necesidad de gestionar la infraestructura donde se ejecutan. Esto permite a los desarrolladores concentrarse en la creación y gestión de aplicaciones, mientras que el proveedor del servicio se encarga de la infraestructura. Ejemplos de *PaaS* son Google App Engine y Heroku.

2.1.3.3. Modelo SaaS (Software como Servicio).

SaaS proporciona aplicaciones completas que son gestionadas y mantenidas por el proveedor del servicio. Los usuarios acceden a estas aplicaciones a través de Internet, generalmente mediante suscripción mensual o anual. La principal ventaja es la simplicidad y rapidez de implementación, ya que se puede comenzar a utilizar el *software* de inmediato sin configuraciones complejas. Ejemplos de *SaaS* son Google Workspace y Salesforce.

A modo de resumen, a continuación se muestra una tabla resumen con los tres modelos:

Modelos de Servicio			
Característica	**IaaS**	**PaaS**	**SaaS**
Nivel de abstracción	Bajo	Medio	Alto
Control	Alto	Medio	Bajo
Responsabilidad	Total: servidores, almacenamiento, redes…	Parcial: desarrollo y gestión aplicación	Nula: proveedor se encarga de todo
Ideal para	Expertos en adminstración de sistemas	Equipos de desarrollo	Empresas con soluciones fáciles
Casos de uso	Alojar servidores	Desarrollar y probar aplicaciones	Gestión de CRM, correo electrónico…
Costo	Pago por uso de recursos	Suscripción mensual o anual	Suscripción mensual/anual por usuario

Tabla 3. Tabla comparativa entre los modelos de servicios de *Cloud*.

2.1.4. Tipos de Cloud

En este apartado se hará una introducción a los diferentes tipos de nube. Cada tipo de nube tiene sus propias características y ventajas, para poder adaptarse a las diversas necesidades y escenarios en los que se utilizan [5].

2.1.4.1. Nube Privada

La nube **privada** se refiere a una infraestructura de *cloud computing* dedicada exclusivamente a una sola organización. Esta infraestructura puede ser gestionada internamente o por un tercero y puede estar ubicada en las instalaciones de la empresa o en un centro de datos externo.

El principal beneficio de la nube privada es el **control total** sobre los recursos y datos, lo que mejora la seguridad y el cumplimiento normativo. Además, permite una mayor **personalización** para satisfacer las necesidades específicas de la empresa. Sin embargo, este enfoque conlleva mayores costos iniciales y de mantenimiento, ya que la organización es la responsable de la completa gestión de la infraestructura.

2.1.4.2. Nube Pública

La nube **pública** es un modelo en el que los servicios de computación son ofrecidos por proveedores externos a través de Internet. Estos proveedores, como Amazon Web Services (AWS), Microsoft Azure y Google Cloud Platform, poseen y operan la infraestructura física, mientras que los usuarios acceden a los recursos según sus necesidades a través de Internet.

La nube pública ofrece una **escalabilidad** casi ilimitada y un modelo de **pago** por uso, lo que reduce los costos iniciales. Además, elimina la necesidad de gestionar la infraestructura. No obstante, puede presentar desafíos en términos de seguridad y cumplimiento, ya que los datos y aplicaciones residen en servidores compartidos y las empresas quedan expuestas a las políticas que implementen los proveedores.

2.1.4.3. Nube Híbrida

La nube **híbrida** combina elementos de nubes privadas y públicas, permitiendo que los datos y aplicaciones se compartan entre ambas. Esto proporciona a las empresas la **flexibilidad** de utilizar la nube pública para tareas menos sensibles y la nube privada para datos y aplicaciones **críticas**.

El beneficio clave de la nube híbrida es su capacidad para ofrecer un **equilibrio** entre escalabilidad y control. Las empresas pueden aprovechar la nube pública para cargas de trabajo fluctuantes mientras mantienen un control estricto sobre los recursos críticos en la nube privada. Sin embargo, la gestión de una infraestructura híbrida puede ser compleja, ya que se debe masterizar ambos ámbitos.

2.1.4.4. Nube Edge

En la nube ***edge*** [9], el procesamiento de datos se realiza lo más **cerca** posible del lugar donde se generan los datos, en lugar de en un gran centro de datos centralizado. La principal ventaja del *edge computing* es la reducción de **latencia**, mientras que sus desventajas son la **distribución** y **gestión** de múltiples puntos de procesamiento.

A modo de resumen, a continuación se muestra una tabla resumen con los cuatro tipos [8]:

Tipos de Cloud				
Característica	**Nube Privada**	**Nube Pública**	**Nube Híbrida**	**Edge Computing**
Control	Total	Limitado	Mixto	Localizado
Seguridad	Propia	Proveedor	Mixta	Mixta
Escalabilidad	Limitada	Alta	Alta	Limitada
Costos	Altos	Basados en uso	Variables	Altos
Flexibilidad	Alta	Media	Alta	Media
Latencia	Alta	Variable	Variable	Baja
Gestión	Compleja	Sencilla	Compleja	Compleja

Tabla 4. Tabla comparativa entre los tipos de *Cloud*.

2.2. GitHub

2.2.1. GitHub y Git

Antes de entrar en los detalles acerca de los conceptos de GitHub, es conveniente entender la diferencia entre Git y GitHub, puesto que son términos que se confunden a menudo.

Git [10] es un sistema de **versiones distribuido** (VCS por las siglas en inglés, *Version Control System*). En esencia, automatiza el proceso de **seguimiento** y **gestión** de versiones del código fuente de un proyecto. Cada modificación realizada en el código es registrada por Git, permitiendo a los desarrolladores tener un **historial** completo de todos los cambios realizados a lo largo del tiempo. Este historial detallado facilita la identificación de quién realizó cada cambio, así como la posibilidad de retroceder a versiones anteriores en cualquier momento.

GitHub es una **plataforma** que se basa en Git y actúa como un servicio de alojamiento en la nube para repositorios Git. GitHub no sólo permite almacenar y gestionar código de manera eficiente, sino que también ofrece una serie de **características adicionales** que enriquecen el proceso de desarrollo y colaboración en proyectos. Entre las funcionalidades más destacadas de GitHub se encuentran los **repositorios** públicos y privados, que facilitan el almacenamiento seguro del código; los ***pull requests***, que permiten revisar y discutir cambios propuestos antes de integrarlos en la rama principal del proyecto; y las herramientas de gestión de proyectos como las ***Issues***, que ayudan a organizar tareas y seguir el progreso de manera estructurada.

Además, GitHub incluye **acciones** de integración y despliegue continuos (GitHub Actions), que **automatizan** flujos de trabajo y despliegues en respuesta a eventos en el repositorio. También ofrece un espacio para la **documentación** en forma de wikis y archivos de documentación.

2.2.2. Alternativas a GitHub

Si bien GitHub es una de las plataformas más populares basadas en Git, existen **otras alternativas** que pueden adaptarse a diferentes necesidades y preferencias:

- **GitLab** [11] es una plataforma de desarrollo **colaborativo** similar a GitHub. Destaca por incluir características como la **integración continua** y el **despliegue continuo** (CI/CD) y la opción de **autoalojamiento**.
- **Bitbucket** [12] es otra alternativa a GitHub que también se basa en Git desarrollada por Atlassian. Destaca por **integrarse** perfectamente con otras herramientas de Atlassian como Jira y Confluence.

Aunque GitLab y Bitbucket también ofrecen capacidades de CI/CD, se descartaron por varias razones. GitLab, a pesar de ser una herramienta muy potente, es demasiada **compleja** de configurar y mantener, especialmente en entornos de autoalojamiento, como el que se usaría para realizar el proyecto. Bitbucket, por otro lado, destaca casi exclusivamente por integrarse con otras herramientas de Atlassian, herramientas las cuáles **no se usan** en este proyecto, lo que deja a Bitbucket con menos soporte y una comunidad más pequeña en comparación con GitHub

Por estos motivos, en este proyecto se ha seleccionado GitHub como la tecnología principal. Además, GitHub ofrece la funcionalidad de **GitHub Actions,** funcionalidad **esencial** para el proyecto. Con las GitHub Actions, la creación de *Issues* y la asignación de *labels*, AWS-Autodeploy podrá **desplegar** los servicios y arquitecturas en AWS.

2.2.3. Repositorios de GitHub

Los **repositorios** [13] son los elementos más **básicos** de GitHub. Representan un espacio donde los desarrolladores pueden almacenar, compartir y gestionar el código de sus proyectos. Es como una especie de **carpeta** que contiene todos los archivos, carpetas, y la historia de cambios (*commits*) del proyecto.

Los repositorios en GitHub son **públicos** por defecto, lo que significa que cualquier persona puede ver su contenido, pero también se pueden configurar como **privados** para restringir el acceso. Algunas **características** clave de los repositorios de GitHub incluyen:

- Gestión de **versiones**: GitHub utiliza el control de versiones Git para gestionar los cambios en el código. Cada vez que se realiza una modificación en los archivos del repositorio, se crea un *commit* que registra los cambios realizados, lo que facilita la reversión a versiones anteriores si es necesario.
- **Colaboración**: GitHub permite que varios desarrolladores trabajen juntos en un mismo proyecto. Los colaboradores pueden clonar (copiar) el repositorio en su máquina local, realizar cambios y enviarlos de vuelta al repositorio a través de *push* y *pull requests*.
- ***Issues***: Las *Issues* son una herramienta para rastrear tareas, errores o mejoras en un proyecto. Se pueden usar para discutir problemas específicos, asignar tareas a diferentes miembros del equipo, y realizar un seguimiento del progreso [14].
- ***Labels***: Las *labels* (etiquetas) se pueden aplicar a las *Issues* para categorizarlas o clasificarlas de alguna manera. Por ejemplo, se pueden utilizar *labels* para identificar problemas relacionados con una determinada función o para marcar problemas como urgentes o de baja prioridad.
- **Historial**: GitHub mantiene un historial detallado de todos los cambios realizados en un repositorio a lo largo del tiempo. Esto permite a los desarrolladores revisar versiones anteriores del código, comparar cambios y revertir a versiones anteriores si es necesario.

2.2.4. GitHub Workflows y GitHub Actions

Los GitHub Workflows [17] son **secuencias automatizadas** de GitHub Actions (*actions*) que se desencadenan en respuesta a **eventos** específicos en un repositorio de GitHub. Estos eventos pueden ser acciones como la creación de una nueva solicitud de extracción (*pull request*), la actualización de una *Issue*, el envío de una confirmación (*commit*) o cualquier otro evento compatible con GitHub.

En cuanto a la **configuración** de un *workflow*, ésta se realiza mediante un archivo YAML dentro del directorio '.github/workflows' en el repositorio. Esta metodología está impuesta por GitHub y es imprescindible para poder usar GitHub Workflow. En el archivo en cuestión, se define la configuración del *workflow* y una serie de *actions* y pasos (*steps*) que se ejecutarán en respuesta a estos eventos.

Es importante definir la diferencia entre ***steps*** y ***actions*** para poder entender su uso:

- Los ***steps*** son acciones individuales que componen el *workflow*. Cada *step* representa una tarea **específica** que se ejecuta como parte del proceso automatizado definido en el *workflow*. Estos *steps* se definen dentro del archivo YAML que configura el *workflow* (en '.github/workflows') y se ejecutan **secuencialmente** en respuesta a un evento específico. Por ejemplo, un *workflow* puede incluir *steps* para clonar el repositorio, compilar el código, ejecutar pruebas o desplegar un servicio.
- Por otro lado, las ***actions*** [15] son componentes más amplios que pueden **contener** uno o más *steps*. Son unidades de trabajo reutilizables que encapsulan un **conjunto** de pasos relacionados. Las GitHub Actions se definen también en archivos YAML y se almacenan en el directorio del repositorio. Una *action* puede abarcar una serie de tareas **complejas** y puede incluir cualquier número de *steps* necesarios para completar esa acción específica. Pueden ser acciones **predefinidas** proporcionadas por GitHub o la comunidad, o acciones **personalizadas** creadas por los propios equipos de desarrollo para adaptarse a las necesidades específicas de su proyecto. Por ejemplo, una *action* predefinida podría ser una acción para compilar el código en varios entornos, mientras que una *action* personalizada podría ser una acción para desplegar la aplicación en un servidor de pruebas.

2.2.4.1. Visualización de un workflow y Github Actions

Una vez el repositorio tiene configurado el *workflow*, en el momento en que se produzca un evento, GitHub iniciará **automáticamente** la ejecución del mismo.

- Durante la **ejecución**, GitHub ofrece en la pestaña de **'Actions'** una interfaz gráfica.
- Si se **accede** a uno de estos *workflows*, se puede ver la lista de **GitHub Actions** que se han realizado, junto con los ***steps*** que las componen.
- Además, para cada *step* se ofrece una serie de **detalles**, en los que se encuentran su **estado** (éxito, fallo o en espera) y los **registros de salida** (*logs*), lo que facilita la depuración y el seguimiento del progreso del *workflow*.

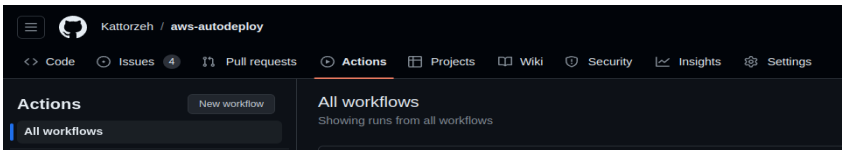

Figura 1. Pestaña 'Actions' de GitHub.

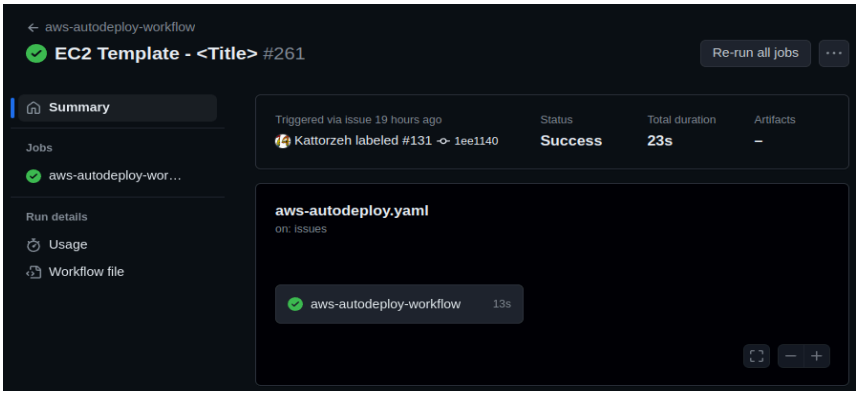

Figura 2. Ejemplo de un *workflow* con una GitHub Action.

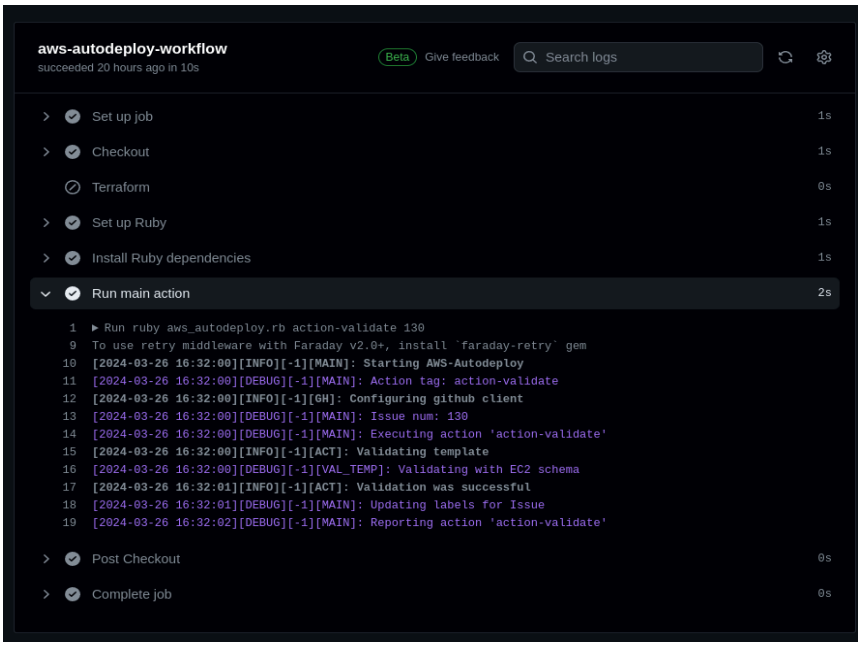

Figura 3. Ejemplo de una GitHub Action con ocho *steps*, con los detalles de uno de ellos.

2.3. Infraestructura como Código (IaC)

La **Infraestructura como Código** (*IaC*) [18] es una práctica que permite **gestionar** y **aprovisionar** infraestructura a través de código en lugar de mediante procesos manuales. Este enfoque utiliza archivos de configuración legibles por humanos que pueden ser versionados y gestionados de manera similar al *software*. *IaC* se utiliza en la **creación**, **modificación** y **destrucción** de entornos de infraestructura de manera eficiente y consistente. Entre las ventajas más destacadas de *IaC* se encuentran:

- La **automatización** de despliegues que reduce la intervención manual.
- La **consistencia**, que garantiza que los entornos de desarrollo, prueba y producción sean idénticos, evitando problemas de configuración e integridad.
- El **versionado**, que permite llevar un registro de cambios, facilitando la auditoría, recuperar versiones anteriores y la colaboración entre equipos.
- El **despliegue rápido**, que acelera el proceso de despliegue de aplicaciones al eliminar tareas repetitivas y manuales.
- La **escalabilidad**, que facilita la ampliación de la infraestructura mediante la reutilización de configuraciones y plantillas, permitiendo escalar según las necesidades del negocio.

El principal objetivo de *IaC* es mejorar la **eficiencia** operativa y la **calidad** de la infraestructura gracias a gestionarla en forma de **código**. Asimismo, busca reducir el tiempo y los costos, aumentar la consistencia y facilitar la colaboración entre equipos de desarrollo y operaciones.

2.3.1. Terraform

Terraform es una herramienta de código abierto (*open source*) desarrollada por HashiCorp que permite a los usuarios **definir** y **aprovisionar** infraestructuras de manera segura y eficiente. Utiliza su **propio** lenguaje de configuración declarativo, HCL (HashiCorp Configuration Language), para describir la infraestructura deseada. Terraform destaca por varias razones:

- **Compatibilidad** con múltiples proveedores *cloud*: facilita la gestión de infraestructura en entornos híbridos o multi-nube, permitiendo una integración con diversos proveedores.
- Estado **declarativo**: permite describir el estado deseado de la infraestructura, y Terraform se encarga de implementar los cambios necesarios para alcanzar ese estado.
- **Modularidad**: facilita la reutilización de configuraciones mediante módulos, promoviendo las buenas prácticas y la eficiencia.
- **Amplia** comunidad y ecosistema: una extensa comunidad y un ecosistema de módulos y extensiones permiten que diversas empresas y desarrolladores creen y compartan sus propios módulos de Terraform, facilitando el acceso a soluciones preconstruidas.
- Funcionalidad de **planificación**: permite ver el impacto de las modificaciones propuestas antes de aplicarlas, lo cual es crucial para evitar errores y garantizar despliegues exitosos, ya que se puede revisar y validar los cambios antes de ejecutarlos.

Para el proyecto, se ha seleccionado Terraform debido a su completa **compatibilidad** con AWS, lo que facilita la gestión de servicios y recursos en esta plataforma. Además, su **facilidad de uso**, gracias a su lenguaje declarativo y su capacidad de planificación. Finalmente, al tratarse de una tecnología muy **popular**, cuenta con un gran **soporte** comunitario para resolver problemas.

2.4. Ruby como lenguaje de programación

Ruby es un lenguaje de programación **dinámico**, **orientado** a **objetos**, conocido por su **simplicidad** y **elegancia**. La elección de Ruby para este proyecto se fundamenta en varias razones alineadas con los objetivos y necesidades del proyecto "AWS-Autodeploy". A continuación, se detallan las principales razones para esta elección [19]:

- **Sencillez** y **flexibilidad**: La capacidad de Ruby para manejar estructuras de texto complejas y su integración con herramientas como *ERB* (Embedded Ruby) permite la **creación** de plantillas que pueden adaptarse a diferentes configuraciones y requisitos.
- Interacción con **servicios externos** y **APIs**: En este proyecto, se utilizan gemas como ‘aws-sdk’ para interactuar con los servicios de AWS, lo que simplifica la validación y gestión de recursos en la nube. Además, el uso de Open3 para la ejecución de comandos en subprocesos permite capturar la salida estándar, errores y estados de salida.
- **Compatibilidad** y **configuración**: Ruby tiene una comunidad activa y un ecosistema de gemas y bibliotecas bien mantenido. Además, la configuración del entorno de ejecución específico de Ruby elimina las inconsistencias y problemas de compatibilidad.

La elección de Ruby para este proyecto no solo está justificada por sus características técnicas y beneficios, sino también por cómo estas características se alinean con las necesidades específicas del proyecto. Ruby proporciona una combinación de simplicidad, flexibilidad, y potencia, que facilitará el desarrollo de la herramienta AWS-Autodeploy.

3. Requisitos

3.1. Guión del usuario para GitHub y la interfaz gráfica

Crea servicios y arquitecturas utilizando la herramienta AWS-Autodeploy. Hay dos modalidades para usar la herramienta: directamente en el repositorio de GitHub o a través de la interfaz gráfica ofrecida en la página web de la herramienta. Para ambas, debe primero satisfacer los requisitos de la herramienta. Posee una cuenta de GitHub, con acceso a ella y con credenciales (*token*) suficientes para la herramienta. Posee una cuenta de AWS, con acceso a ella y con credenciales (*Access Key* y *Secret Access Key*) suficientes para la herramienta. Para finalizar, inserta los secretos necesarios en el repositorio.

Para la modalidad de Github, puede crear nuevas *Issues* en el repositorio. Le aparecen las plantillas preconfiguradas y tiene acceso a ellas. Selecciona la que le interesa y completa los campos con sus datos. Puede agregar *labels* a la *Issue*. Conoce y entiende el ciclo de vida de la herramienta. Para realizar una acción, inserta nuevas *labels* a la *Issue*. Al insertar una acción, se inicia una ejecución automatizada por un *workflow* de la herramienta. La herramienta le ofrece una comunicación clara sobre el estado de la acción solicitada. En caso de error, recibe información sobre cómo solucionarlo, así como dispone y usa las acciones de recuperación. En caso de éxito, avanza por las distintas acciones y estados de la herramienta.

Para la modalidad de interfaz gráfica, escribe en el campo de texto correcto el nombre de su repositorio público, su nombre de usuario y su *token* de GitHub. Así mismo, le aparecen las distintas plantillas disponibles para la herramienta, identificadas con una imagen. Tiene acceso a las plantillas y puede hacer clic sobre ellas. Al clicar se le abre una nueva vista. En esta vista, puede rellenar los campos de la plantilla con sus datos. Se le muestran las acciones de la herramienta como opciones con un diseño gráfico que ayuda a su comprensión. Puede seleccionar las distintas acciones del ciclo de vida de la herramienta, con las opciones visuales que le aparecen en la misma vista. Al seleccionar una acción, en el repositorio de GitHub se inicia una ejecución automatizada del *workflow* de la herramienta. Se le muestra de manera visual el éxito o fracaso de la acción en una nueva ventana, donde aparecen nuevas acciones a realizar. Puede solucionar los errores mediante el uso de las herramientas de recuperación. Puede seguir con las siguientes acciones en caso de éxito. En caso de querer modificar un valor de la plantilla, puede regresar al formulario inicial.

3.2. Requisitos funcionales

Los requisitos funcionales de la herramienta AWS-Autodeploy son los siguientes:

1. El usuario puede usar las plantillas preconfiguradas en las *Issues*.
    a. Estas plantillas deben estar disponibles en el repositorio donde se aloja AWS-Autodeploy.
    b. Al crear una nueva *Issue*, debe aparecer un listado con todas las plantillas preconfiguradas.
    c. Al seleccionar una plantilla, en la *Issue* deben aparecer los campos preconfigurados de la plantilla.
2. El usuario puede crear nuevas plantillas para la herramienta.
    a. AWS-Autodeploy debe soportar la creación de nuevas plantillas.
    b. Se puede agregar una nueva plantilla, siempre y cuando se sigan los pasos estipulados en el apartado correspondiente: las plantillas se guardarán en el directorio ‘ISSUE_TEMPLATE’ en un archivo con extensión ‘.md’ con los parámetros en formato correcto.

3. El usuario puede crear nuevos servicios para la herramienta.
    a. AWS-Autodeploy debe soportar agregar nuevos servicios del proveedor AWS.
    b. Se puede agregar un nuevo servicio siempre y cuando se sigan los pasos estipulados en el apartado correspondiente: la implementación del servicio se guardará en el directorio 'TERRAFORM', agregando las validaciones y las plantillas correspondientes.
4. El usuario puede crear *Issues*.
    a. El repositorio debe otorgar permisos suficientes al usuario para que pueda crear *Issues*.
    b. El usuario debe tener acceso al espacio donde se crean las *Issues*.
5. El usuario puede agregar *labels* a las *Issues*.
    a. El repositorio debe otorgar permisos suficientes al usuario para que pueda agregar *labels* a las *Issues*.
    b. AWS-Autodeploy debe tener el listado de *labels* necesarios para poder llevar a cabo su ciclo de vida.
    c. El usuario debe tener acceso al espacio donde se agregan las *labels*.
6. El usuario puede cambiar de estado en el ciclo de vida de la herramienta mediante agregar *labels* específicas.
    a. AWS-Autodeploy debe iniciar un *workflow* y GitHub Actions necesarias para realizar el cambio de estado pertinente.
    b. AWS-Autodeploy debe ofrecer una documentación clara y detallada sobre su ciclo de vida.
    c. En el repositorio, existen las *labels* necesarias para realizar las acciones de cambio de estado.
7. El usuario puede crear servicios individuales de AWS.
    a. Tras ejecutar correctamente la herramienta AWS-Autodeploy, debe estar disponible el servicio correspondiente que se ha configurado previamente.
    b. AWS-Autodeploy ofrece soluciones en caso de contratiempos para poder recuperarse de ellos.
    c. AWS-Autodeploy ofrece una interacción correcta con el usuario detallando su estado actual y procesos realizados.
    d. AWS-Autodeploy puede ofrecer plantillas del servicio para facilitar su despliegue.
    e. El usuario puede configurar a su gusto el servicio, siempre y cuando se adecue a las validaciones del mismo.
8. El usuario puede crear múltiples servicios simultáneamente (arquitecturas) de AWS.
    a. Tras ejecutar correctamente la herramienta AWS-Autodeploy, debe estar disponible el conjunto de servicios correspondientes que se han configurado previamente.
    b. AWS-Autodeploy ofrece soluciones en caso de contratiempos para poder recuperarse de ellos.
    c. AWS-Autodeploy ofrece una interacción correcta con el usuario detallando su estado actual y procesos realizados.
    d. El usuario puede agregar a AWS-Autodeploy la plantilla de la arquitectura para agilizar su proceso.
    e. El usuario puede configurar a su gusto los servicios que conforman la arquitectura, siempre y cuando se adecue a las validaciones de cada uno de los servicios desplegados

3.3. Requisitos no funcionales

Los requisitos no funcionales de la herramienta AWS-Autodeploy son los siguientes:

Requisitos de seguridad

- **Uso de credenciales de AWS**: La herramienta debe utilizar las credenciales de acceso de la cuenta de AWS del usuario para interactuar con los servicios de AWS de manera segura y autenticada.
- **Uso de credenciales de GitHub**: La herramienta debe utilizar las credenciales de acceso de la cuenta de GitHub del usuario para autenticar y autorizar las acciones que se realicen en los repositorios de GitHub.
- **Configuración de secretos del repositorio**: El repositorio debe tener configurados varios secretos de repositorio para almacenar de manera segura las credenciales y otros datos sensibles necesarios para la ejecución de la herramienta.

Requisitos de diseño y construcción

- **Lenguaje de programación**: La lógica de la herramienta se debe implementar con el lenguaje de programación Ruby para asegurar un código claro y mantenible.
- **Gestión de infraestructura**: Se debe usar Terraform para gestionar la infraestructura como código (*IaC*), lo que permite una configuración repetible y fácil de mantener.
- **Parámetros de implementación**: La implementación debe estar parametrizada para permitir la creación de múltiples servicios con múltiples instancias, proporcionando flexibilidad y escalabilidad.
- **Flujos de trabajo automatizados**: Se deben usar los flujos de trabajo y las GitHub Actions que ofrece GitHub para automatizar las tareas necesarias para llevar a cabo los despliegues configurados en las *Issues*.
- **Directrices de GitHub**: Se deben seguir las directrices de GitHub para implementar su flujo de trabajo, asegurando así que las prácticas de desarrollo sean consistentes y de alta calidad.
- **Entorno independiente**: La herramienta debe ejecutarse en un entorno independiente al dispositivo o sistema operativo del usuario, lo que garantiza que el código sea portable y consistente en diferentes entornos.

Requisitos de usabilidad

- **Repositorio público**: La herramienta debe estar alojada en un repositorio público del usuario, facilitando el acceso y la colaboración con otros desarrolladores.
- **Plantillas preconfiguradas en *Issues***: Debe ofrecer plantillas preconfiguradas en las *Issues* para facilitar la introducción de los parámetros necesarios, mejorando así la experiencia del usuario y reduciendo errores de configuración.
- **Flujo de vida detallado**: Debe ofrecer un flujo de vida detallado y fácil de seguir, proporcionando a los usuarios una guía clara de los estados y transiciones del sistema.
- ***Labels* para el cambio de estado**: Debe ofrecer un conjunto de *labels* para cambiar de estado. Estos *labels* deben iniciar un flujo de trabajo automáticamente, asegurando que las transiciones de estado sean consistentes y automatizadas.
- **Directrices de GitHub para plantillas preconfiguradas**: Se deben seguir las directrices de GitHub para implementar las plantillas preconfiguradas, asegurando así que las plantillas sean fáciles de usar y estén bien integradas con las prácticas de GitHub.

4. Análisis de los requisitos

4.1. Diagrama de secuencia de los casos de uso

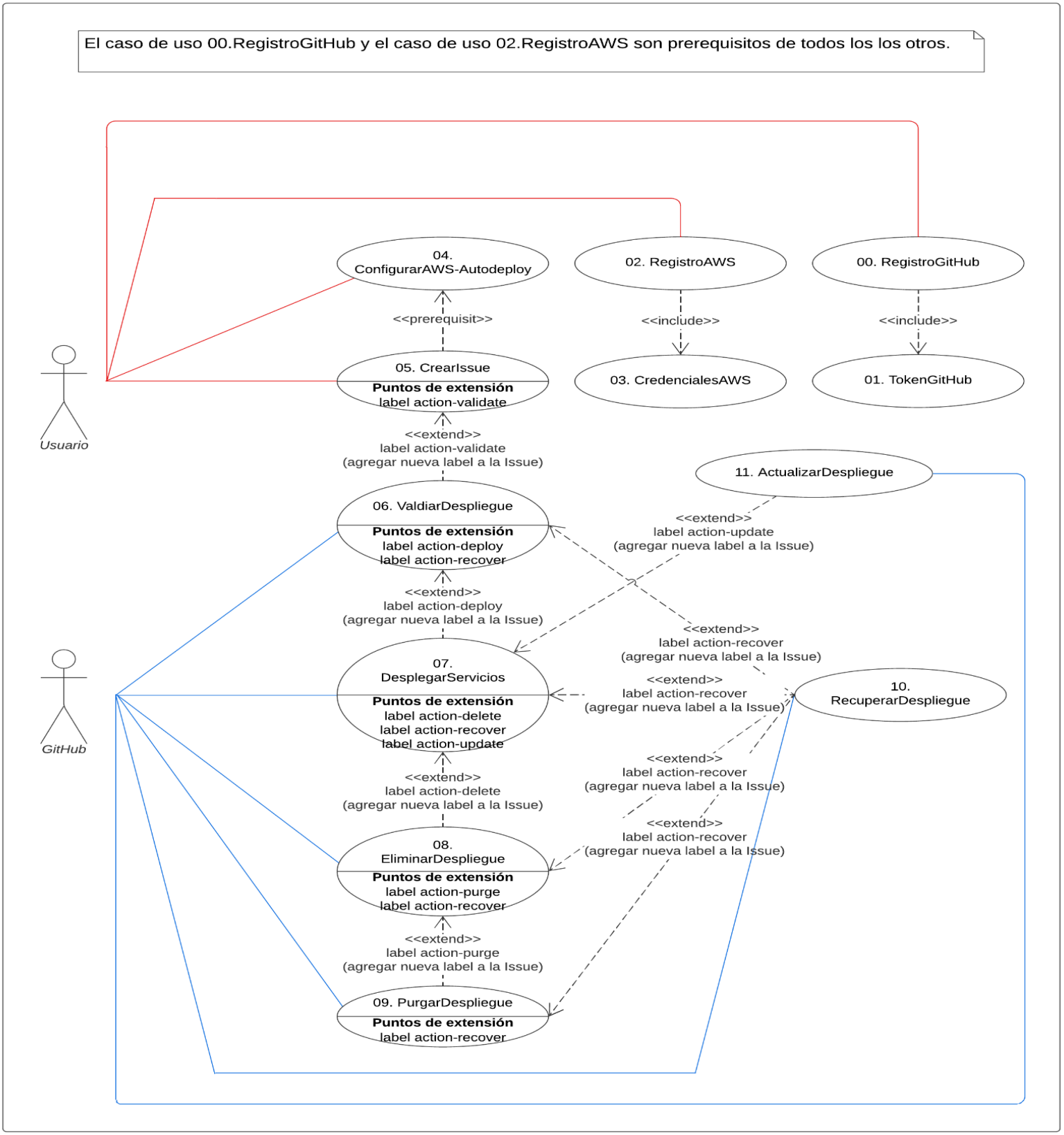

Figura 4. Diagrama de casos de uso de AWS-Autodeploy [20].

4.2. Especificación textual de los casos de uso

Caso de uso 00. RegistroGitHub	
Resumen de la funcionalidad	El **Usuario** inicia sesión en la plataforma GitHub con sus datos.
Parámetros de entrada	Ninguno.
Parámetros de salida	Ninguno.
Usuarios	**Usuario.**
Precondición	Ninguno.
Postcondición	El **Usuario** se ha identificado y tiene acceso a las funciones correspondientes.

Tabla 5. Tabla caso de uso '00. RegistroGitHub'.

Proceso normal principal:

1. La plataforma GitHub solicita los datos de inicio de sesión al **Usuario**.
2. El **Usuario** completa los campos requeridos (usuario, contraseña).
3. La plataforma GitHub ofrece al **Usuario** acceso a sus funcionalidades.

Alternativas de proceso y excepciones:

1a. El **Usuario** no tiene una cuenta en la plataforma GitHub.
  1a1. La plataforma GitHub pide los datos de registro al **Usuario**.
  1a2. El **Usuario** completa los campos (usuario, contraseña, email y preferencias).
  1a3. La plataforma valida los datos proporcionados por el **Usuario**.
    1a3a1. En caso de error, la plataforma vuelve al paso 1a.
  1a4. La plataforma registra al **Usuario** y regresa al paso 3 del proceso normal principal.

Caso de uso 01. TokenGitHub	
Resumen de la funcionalidad	El **Usuario** obtiene un *token* de GitHub con las credenciales necesarias.
Parámetros de entrada	Ninguno.
Parámetros de salida	*Token* de acceso.
Usuarios	**Usuario.**
Precondición	El **Usuario** debe haber completado el registro en GitHub (caso de uso 00.RegistroGitHub) y tener una cuenta válida.
Postcondición	El **Usuario** obtiene un token de acceso válido con las credenciales.

Tabla 6. Tabla caso de uso '01. TokenGitHub'.

Proceso normal principal:

1. El **Usuario** accede al menú de configuración de la plataforma GitHub.

2. El **Usuario** accede a la sección de ajustes de desarrollador.
3. El **Usuario** genera un nuevo *token* para su cuenta.
4. El **Usuario** asigna los permisos necesarios para el *token*.

Alternativas de proceso y excepciones: ninguna.

Caso de uso 02. RegistroAWS	
Resumen de la funcionalidad	El **Usuario** inicia sesión en AWS con sus datos.
Parámetros de entrada	Ninguno.
Parámetros de salida	*Token* de acceso.
Usuarios	**Usuario.**
Precondición	Ninguno.
Postcondición	El **Usuario** obtiene un *token* de acceso válido y con las credenciales necesarias.

Tabla 7. Tabla caso de uso '02. RegistroAWS'.

Proceso normal principal:

1. La plataforma AWS solicita los datos de inicio de sesión al **Usuario**.
2. El **Usuario** completa los campos requeridos (usuario, contraseña).
3. La plataforma AWS ofrece al **Usuario** acceso a sus funcionalidades, como crear recursos en la nube, gestionar servicios, entre otras.

Alternativas de proceso y excepciones: ninguna.

1a. El **Usuario** no tiene una cuenta en AWS.

1a1. AWS pide los datos de registro al **Usuario**.

1a2. El **Usuario** completa los campos (nombre de la cuenta, contraseña, email, tarjeta de crédito y preferencias).

1a3. La plataforma valida los datos proporcionados por el **Usuario**.

1a3a1. En caso de error, la plataforma vuelve al paso 1a.

1a4. La plataforma registra al **Usuario** y regresa al paso 3 del proceso normal principal.

Caso de uso 03. CredencialesAWS	
Resumen de la funcionalidad	El **Usuario** obtiene unas credenciales de acceso AWS (*Access Key* y *Secret Access Key*).
Parámetros de entrada	Ninguno.
Parámetros de salida	Credenciales de acceso AWS (*Access Key* y *Secret Access Key*).
Usuarios	**Usuario.**
Precondición	El **Usuario** debe haber completado el registro en AWS (caso de uso 02.RegistroAWS) y tener una cuenta válida.

Postcondición	El **Usuario** obtiene una clave de acceso (*Access Key*) y una clave de acceso secreta (*Secret Access Key*) válidas y necesarias para interactuar con los servicios de AWS.

Tabla 8. Tabla caso de uso '03. CredencialesAWS'.

Proceso normal principal:

1. El **Usuario** accede al panel de administración de identidad y acceso (*IAM*) AWS.
2. El **Usuario** crea un nuevo usuario que será el que obtenga las credenciales.
3. El **Usuario** otorga permisos necesarios (o de administrador) al usuario que ha creado en el paso 2.
4. El **Usuario** genera unas credenciales de acceso para el usuario del paso 2.

Alternativas de proceso y excepciones: ninguna.

Caso de uso 04. ConfigurarAWS-Autodeploy	
Resumen de la funcionalidad	El **Usuario** configura el repositorio de AWS-Autodeploy cumpliendo con los requisitos del mismo.
Parámetros de entrada	Ninguno.
Parámetros de salida	Ninguno.
Usuarios	**Usuario.**
Precondición	El **Usuario** debe haber completado el registro en GitHub (caso de uso 00.RegistroGitHub) y tener un token válido (caso de uso 01.TokenGitHub). El **Usuario** debe haber completado el registro en AWS (caso de uso 02.RegistroAWS) y tener unas credenciales válidas (caso de uso 03.CredencialesAWS).
Postcondición	El **Usuario** cuenta con el repositorio de la herramienta AWS-Autodeploy en su perfil, con una configuración correcta y listo para usarse.

Tabla 9. Tabla caso de uso '04. ConfigurarAWS-Autodeploy'.

Proceso normal principal:

1. El **Usuario** descarga el repositorio AWS-Autodeploy.
2. El **Usuario** agrega las *labels* necesarias para el correcto funcionamiento de la herramienta.
3. El **Usuario** agrega los secretos del repositorio necesarios.

3.1. Para las credenciales de GitHub y AWS, utiliza las credenciales obtenidas en los casos de uso 01.*TokenGitHub* y 03.*CredencialesAWS*.

4. El **Usuario** modifica los permisos del repositorio para seguir con las prácticas recomendadas de la herramienta.

Alternativas de proceso y excepciones:

5. El **Usuario** utiliza la interfaz gráfica de AWS-Autodeploy.
6. El **Usuario** debe configurar la interfaz gráfica de AWS-Autodeploy.
    a. El **Usuario** inicia sesión en la interfaz gráfica con sus datos (nombre de usuario, nombre del repositorio y *token* de GitHub).

Caso de uso 05. CrearIssue	
Resumen de la funcionalidad	El **Usuario** crea una nueva *Issue* en el repositorio donde se aloja AWS-Autodeploy utilizando una plantilla preconfigurada.
Parámetros de entrada	Ninguno.
Parámetros de salida	Una *Issue* con las *labels* “state-pending” y “action-validate”.
Usuarios	**Usuario.**
Precondición	Ninguno.
Postcondición	Se ha creado una nueva *Issue* en el repositorio de AWS-Autodeploy, con las *labels* “state-pending” y “action-validate” y con los campos necesarios completados.

Tabla 10. Tabla caso de uso ‘05. CrearIssue’.

Proceso normal principal:

1. El **Usuario** accede al repositorio de AWS-Autodeploy.
2. El **Usuario** selecciona la opción para crear una nueva *Issue*.
3. El **Usuario** elige una plantilla preconfigurada para la nueva *Issue*.
4. El **Usuario** completa los campos requeridos en la plantilla, proporcionando la información necesaria según sea necesario y cumpliendo con los protocolos.
5. El **Usuario** comprueba que la *Issue* tenga las *labels* “state-pending” y “action-validate”.
    a. Si las *labels* no están presentes,el **Usuario** las agrega manualmente.
6. El **Usuario** envía la nueva *Issue*, con las *labels* y los campos completados.

Alternativas de proceso y excepciones:

1a. El **Usuario** utiliza la interfaz gráfica de AWS-Autodeploy.
    1a1. El **Usuario** navega por la ventana gráfica donde aparecen todas las plantillas configuradas en su repositorio de GitHub.
    1a2. El **Usuario** selecciona la plantilla que desee.
    1a3. El **Usuario** completa los campos de la plantilla, proporcionando la información necesaria según sea necesario y cumpliendo con los protocolos.
    1a5. El **Usuario** pulsa el botón ‘VALIDATE’ y se crea la *Issue* con sus *labels* automáticamente.
    1a4. El **Usuario** es redirigido a una nueva ventana, donde aparece el estado de la ejecución, así como nuevas acciones disponibles para la *Issue*.
2a. El **Usuario** no utiliza una plantilla preconfigurada para crear la nueva *Issue*.
    2a1. El **Usuario** selecciona crear una nueva *Issue* en blanco.
    2a2. El **Usuario** agrega los campos necesarios para configurar el servicio o arquitectura.
    2a3. Se regresa al paso 4 del proceso normal principal.

Caso de uso 06. ValidarDespliegue	
Resumen de la funcionalidad	El actor **GitHub** inicia un *workflow* automatizado para validar los parámetros de la *Issue*.
Parámetros de entrada	Una Issue con la *label* “action-validate”.

Parámetros de salida	Una *Issue* con la *label* "state-validated".
Usuarios	**GitHub.**
Precondición	Ninguno.
Postcondición	Se ha iniciado y ejecutado el *workflow* de validación de los parámetros de la *Issue*.

Tabla 11. Tabla caso de uso '06. ValidarDespliegue'.

Proceso normal principal:
1. **GitHub** detecta una *Issue* con la *label* "action-validate".
2. **GitHub** ejecuta satisfactoriamente el *workflow* de validación.
    a. Se inicia el *workflow* con la GitHub Action configurada correspondientemente.
    b. El *workflow* verifica los parámetros de la *Issue*.
    c. El *workflow* devuelve un estado de satisfacción.
3. **GitHub** cambia la *label* de la *Issue* a "state-validated".

Alternativas de proceso y excepciones:

2a. **GitHub** ejecuta con error el *workflow* de validación.
2a1. Se inicia el *workflow* con la GitHub Action configurada correspondientemente.
2a2. El *workflow* verifica los parámetros de la *Issue*.
2a3. El *workflow* devuelve un estado de error.
3a. **GitHub** cambia la *label* de la *Issue* a "state-failed-validate".

Caso de uso 07. DesplegarServicios	
Resumen de la funcionalidad	El actor **GitHub** inicia un *workflow* automatizado para desplegar los servicios configurados en la *Issue*.
Parámetros de entrada	Una Issue con la *label* "action-deploy".
Parámetros de salida	Una *Issue* con las *labels* "state-deploying" y "state-running".
Usuarios	**GitHub.**
Precondición	Ninguno.
Postcondición	Se ha iniciado y ejecutado el *workflow* del despliegue de servicios configurados en la *Issue*.

Tabla 12. Tabla caso de uso '07. DesplegarServicios'.

Proceso normal principal:
1. **GitHub** detecta una *Issue* con la *label* "action-deploy".
2. **GitHub** ejecuta satisfactoriamente el *workflow* de despliegue.
    a. Se inicia el *workflow* con la GitHub Action configurada correspondientemente.
    b. El *workflow* prepara el entorno Terraform.
    c. El *workflow* ejecuta los comandos Terraform para desplegar los servicios.
    d. El *workflow* devuelve un estado de satisfacción.
3. **GitHub** cambia la *label* de la *Issue* a "state-deploying" y "state-running".

Alternativas de proceso y excepciones:
2a. **GitHub** ejecuta con error el *workflow* del despliegue.
2a1. Se inicia el *workflow* con la GitHub Action configurada correspondientemente.
2a2. El *workflow* prepara el entorno Terraform.
2a3. El *workflow* ejecuta los comandos Terraform para desplegar los servicios.
2a4. El *workflow* devuelve un estado de error.
3a. **GitHub** cambia la *label* de la *Issue* a "state-failed-deploying".

Caso de uso 08. EliminarDespliegue	
Resumen de la funcionalidad	El actor **GitHub** inicia un *workflow* automatizado para eliminar (terminar) los servicios desplegados por AWS-Autodeploy.
Parámetros de entrada	Una Issue con la *label* "action-delete".
Parámetros de salida	Una *Issue* con las *labels* "state-undeploying" y "state-deleted".
Usuarios	**GitHub.**
Precondición	Ninguno.
Postcondición	Se ha iniciado y ejecutado el *workflow* de finalización (terminación) de servicios desplegados por la *Issue*.

Tabla 13. Tabla caso de uso '08. EliminarDespliegue'.

Proceso normal principal:
1. **GitHub** detecta una *Issue* con la *label* "action-delete".
2. **GitHub** ejecuta satisfactoriamente el *workflow* de finalizar los servicios.
    a. Se inicia el *workflow* con la GitHub Action configurada correspondientemente.
    b. El *workflow* prepara el entorno Terraform.
    c. El *workflow* ejecuta los comandos Terraform para finalizar los servicios.
    d. El *workflow* devuelve un estado de satisfacción.
3. **GitHub** cambia la *label* de la *Issue* a "state-undeploying" y "state-deleted".

Alternativas de proceso y excepciones:
2a. **GitHub** ejecuta con error el *workflow* de finalizar los servicios.
2a1. Se inicia el *workflow* con la GitHub Action configurada correspondientemente.
2a2. El *workflow* prepara el entorno Terraform.
2a3. El *workflow* ejecuta los comandos Terraform para finalizar los servicios.
2a4. El *workflow* devuelve un estado de error.
3a. **GitHub** cambia la *label* de la *Issue* a "state-failed-undeploying".

Caso de uso 09. PurgarDespliegue	
Resumen de la funcionalidad	El actor **GitHub** inicia un *workflow* automatizado para eliminar los ficheros de configuración creados por AWS-Autodeploy.
Parámetros de entrada	Una Issue con la *label* "action-purge".
Parámetros de salida	Una *Issue* con la *label* "state-purged".

Usuarios	**GitHub.**
Precondición	Ninguno.
Postcondición	Se han eliminado los ficheros de configuración del repositorio.

Tabla 14. Tabla caso de uso '09. PurgarDespliegue'.

Proceso normal principal:

1. **GitHub** detecta una *Issue* con la *label* "action-purge".
2. **GitHub** ejecuta satisfactoriamente el *workflow* para eliminar los ficheros de configuración.
    a. Se inicia el *workflow* con la GitHub Action configurada correspondientemente.
    b. El *workflow* borra (elimina) los ficheros de configuración del repositorio.
    c. El *workflow* devuelve un estado de satisfacción.
3. **GitHub** cambia la *label* de la *Issue* a "state-purged".

Alternativas de proceso y excepciones:

2a. **GitHub** ejecuta con error el *workflow* para eliminar los ficheros de configuración.
2a1. Se inicia el *workflow* con la GitHub Action configurada correspondientemente.
2a2. El *workflow* no borra (no elimina) los ficheros de configuración del repositorio.
2a3. El *workflow* devuelve un estado de error.
3a. **GitHub** cambia la *label* de la *Issue* a "state-failed-purge".

Caso de uso 10. RecuperarDespliegue	
Resumen de la funcionalidad	El actor **GitHub** inicia un *workflow* automatizado para recuperar (solucionar) la *Issue* de un estado fallido a un estado válido..
Parámetros de entrada	Una Issue con las *labels* "action-recover" y "state-failed-*".
Parámetros de salida	Una *Issue* con la *label* "state-*".
Usuarios	**GitHub.**
Precondición	Ninguno.
Postcondición	Se ha recuperado el estado de la *Issue*.

Tabla 15. Tabla caso de uso '10. RecuperarDespliegue'.

Proceso normal principal:

1. **GitHub** detecta una *Issue* con la *label* "action-recover".
2. **GitHub** ejecuta satisfactoriamente el *workflow* para recuperar y guiar nuevamente a la *Issue* a un estado válido dentro de su ciclo de vida.
    a. Se inicia el *workflow* con la GitHub Action configurada correspondientemente.
    b. El *workflow* obtiene cuál es el estado fallido de la *Issue*.
    c. El *workflow* ejecuta la acción pertinente para solventar y revertir el estado.
    d. El *workflow* devuelve un estado de satisfacción.
3. **GitHub** cambia la *label* de la *Issue* a "state-*", dependiendo del estado en el que se encuentre la *Issue*.

Alternativas de proceso y excepciones:

2a. **GitHub** ejecuta con error el *workflow* para recuperar y guiar nuevamente a la *Issue* a un estado válido dentro de su ciclo de vida.

2a1. Se inicia el *workflow* con la GitHub Action configurada correspondientemente.

2a2. El *workflow* obtiene cuál es el estado fallido de la *Issue*.

2a3. El *workflow* ejecuta la acción pertinente para solventar y revertir el estado, pero lo hace sin éxito.

2a4. El *workflow* devuelve un estado de error.

3a. **GitHub** cambia la *label* de la *Issue* a "state-failed-*", dependiendo del estado fallido en el que se encuentre la *Issue*.

Caso de uso 11. ActualizarDespliegue	
Resumen de la funcionalidad	El actor **GitHub** inicia un *workflow* automatizado para actualizar los servicios desplegados por AWS-Autodeploy.
Parámetros de entrada	Una Issue con la *label* "action-update".
Parámetros de salida	Una *Issue* con las *labels* "state-deploying" y "state-running"..
Usuarios	**GitHub.**
Precondición	Ninguno.
Postcondición	Se han actualizado los servicios según se han configurado en la *Issue*.

Tabla 16. Tabla caso de uso '11. ActualizarDespliegue'.

Proceso normal principal:

1. **GitHub** detecta una *Issue* con la *label* "action-update".
2. **GitHub** ejecuta satisfactoriamente el *workflow* para actualizar los servicios desplegados.
   a. Se inicia el *workflow* con la GitHub Action configurada correspondientemente.
   b. El *workflow* valida los nuevos parámetros de la *Issue*.
   c. El *workflow* prepara el entorno Terraform.
   d. El *workflow* ejecuta los comandos Terraform para actualizar los servicios.
   e. El *workflow* devuelve un estado de satisfacción.
3. **GitHub** cambia la *label* de la *Issue* a "state-deploying" y "state-running".

Alternativas de proceso y excepciones:

2a. **GitHub** ejecuta con error el *workflow* para actualizar los servicios desplegados.

2a1. Se inicia el *workflow* con la GitHub Action configurada correspondientemente.

2a2. El *workflow* valida los nuevos parámetros de la *Issue*.

2a2a. El *workflow* detecta un error en los parámetros, se salta al paso 3a.

2a3. El *workflow* prepara el entorno Terraform.

2a4. El *workflow* ejecuta con error los comandos Terraform para actualizar los servicios.

2a5. El *workflow* devuelve un estado de error.

3a. **GitHub** cambia la *label* de la *Issue* a "state-failed-*", dependiendo del estado fallido en el que se encuentre la *Issue* (validación o despliegue).

5. Diseño

AWS-Autodeploy es una herramienta que facilita el despliegue de servicios y arquitecturas en el *Cloud* de AWS. Para utilizarla, el usuario deberá tener una cuenta en GitHub y en AWS, así como configurar correctamente el repositorio de AWS-Autodeploy en su GitHub (teniendo especial cuidado en la sección de los secretos).

El verdadero potencial de AWS-Autodeploy se encuentra en la sencilla e intuitiva manera en la que permite a los usuarios crear y desplegar múltiples servicios o arquitecturas en AWS, con múltiples instancias de cada uno. Todo ello sin la necesidad de escribir código o sin tener que pasar por multitudes de páginas web distintas.

Para poder implementar esta herramienta, se han seguido una serie de directrices que se encuentran definidas en este apartado de Diseño.

5.1. Arquitectura

Como se explica en el apartado "6. Implementación", AWS-Autodeploy se sustenta bajo tres tecnologías principales, las GitHub Actions, Ruby y Terraform, como muestra la imagen de su arquitectura.

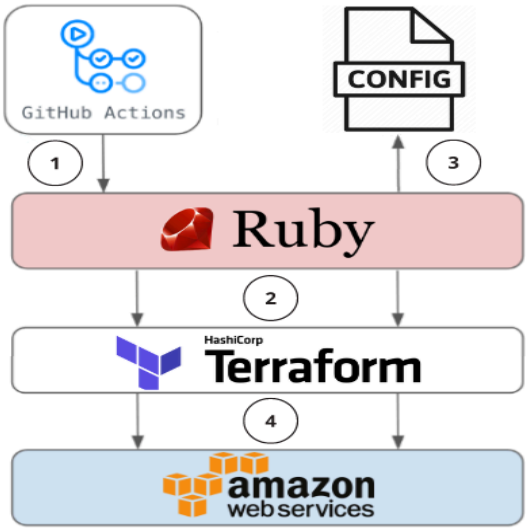

Figura 5. Arquitectura de AWS-Autodeploy.

En la herramienta, se ha definido un *workflow* con una **GitHub Action** con una serie de *steps*, que serán activados con cuando se crea o se modifica una *Issue* en el repositorio. Estas acciones, se ejecutarán de manera automática en un contenedor que ofrece GitHub. Se encargarán de preparar las configuraciones y el entorno para que los scripts de Ruby y los ficheros de Terraform puedan ejecutarse.

En el último *step* de la GitHub Action, se inicia la ejecución del *script* principal de **Ruby**. Los ficheros Ruby contienen la implementación de la lógica de AWS-Autodeploy, es decir, son los encargados de ejecutar las distintas acciones que se definen en el ciclo de vida de la herramienta. Según el tipo de *label* que contenga la *Issue*, se ejecutará una u otra acción implementada en Ruby. Además, se han creado diferentes herramientas internas de AWS-Autodeploy para facilitar su implementación.

Aparte de estos *scripts*, AWS-Autodeploy también genera diversos **ficheros de configuración** que se subirán al repositorio. Estos ficheros contienen, como indica su nombre, las configuraciones necesarias para poder desplegar los servicios y arquitecturas, así como el estado en el que se encuentra la infraestructura, incluyendo qué servicios se han creado o cuáles se han eliminado.

Estos ficheros serán utilizados por algunas acciones de los *scripts* de Ruby. En concreto, por las acciones 'deploy', 'delete', 'recover', 'update' y 'purge', ya que son las encargadas de ejecutar el código **Terraform**. El código Terraform es el que permite configurar los servicios y arquitecturas de manera dinámica con los valores que el usuario desee, así como desplegarlos o eliminarlos.

Finalmente, el resultado del uso de la herramienta AWS-Autodeploy queda reflejado en el **proveedor AWS**, donde el usuario puede ver en ejecución los servicios que ha desplegado con la herramienta.

5.2. Ciclo de Vida

El ciclo de vida de AWS-Autodeploy inicia con la creación de una *Issue* en GitHub, la cuál debe estar etiquetada con el estado **'PENDING'** y la acción **'action-validate'** para su detección por parte de AWS-Autodeploy.

Las transiciones entre estados son **automáticas** y son provocadas al agregar nuevas etiquetas (***labels***) de acciones (denominadas 'action-') sobre la *Issue*. Agregar estas ***labels*** desencadena un *workflow* de GitHub que contendrá GitHub Actions.

Por ejemplo, al insertar la *label* **'action-validate'**, se iniciará un *workflow* que contendrá una GitHub Action configurada para validar los parámetros de la *Issue*. Si su ejecución es correcta, la *Issue* avanzará al estado **'VALIDATED'**, y cambiará sus *labels* para indicarlo. En caso de que la ejecución acabe con un error, la *Issue* entrará en un estado de error **'FAILED_VALIDATE'**.

Los estados de error ('FAILED_****') representan situaciones en las que una acción no se ha ejecutado correctamente. Estos estados requieren una intervención específica por parte del usuario. Esto puede incluir la revisión de los parámetros introducidos o la revisión de la infraestructura en el proveedor *Cloud*. Para facilitar su solución, en la propia *Issue* se proporciona información sobre qué ha fallado. Asimismo, la acción **'action-recover'** se utiliza para intentar remediar la situación y llevar la *Issue* de vuelta a un estado válido dentro del ciclo de vida de la herramienta.

Finalmente, el ciclo de vida de AWS-Autodeploy concluye cuando la *Issue* es eliminada de GitHub. Una *Issue* en AWS-Autodeploy, y en consecuencia, su ciclo de vida, no se considera **terminado** hasta que no se han purgado (eliminando) los ficheros de configuración que se han subido al repositorio durante el despliegue. En ese momento, cuando se recibe el estado **'PURGED'**, la *Issue* puede ser concluida o eliminada.

Para entender mejor el funcionamiento de AWS-Autodeploy, a continuación se define un diagrama representando su ciclo de vida:

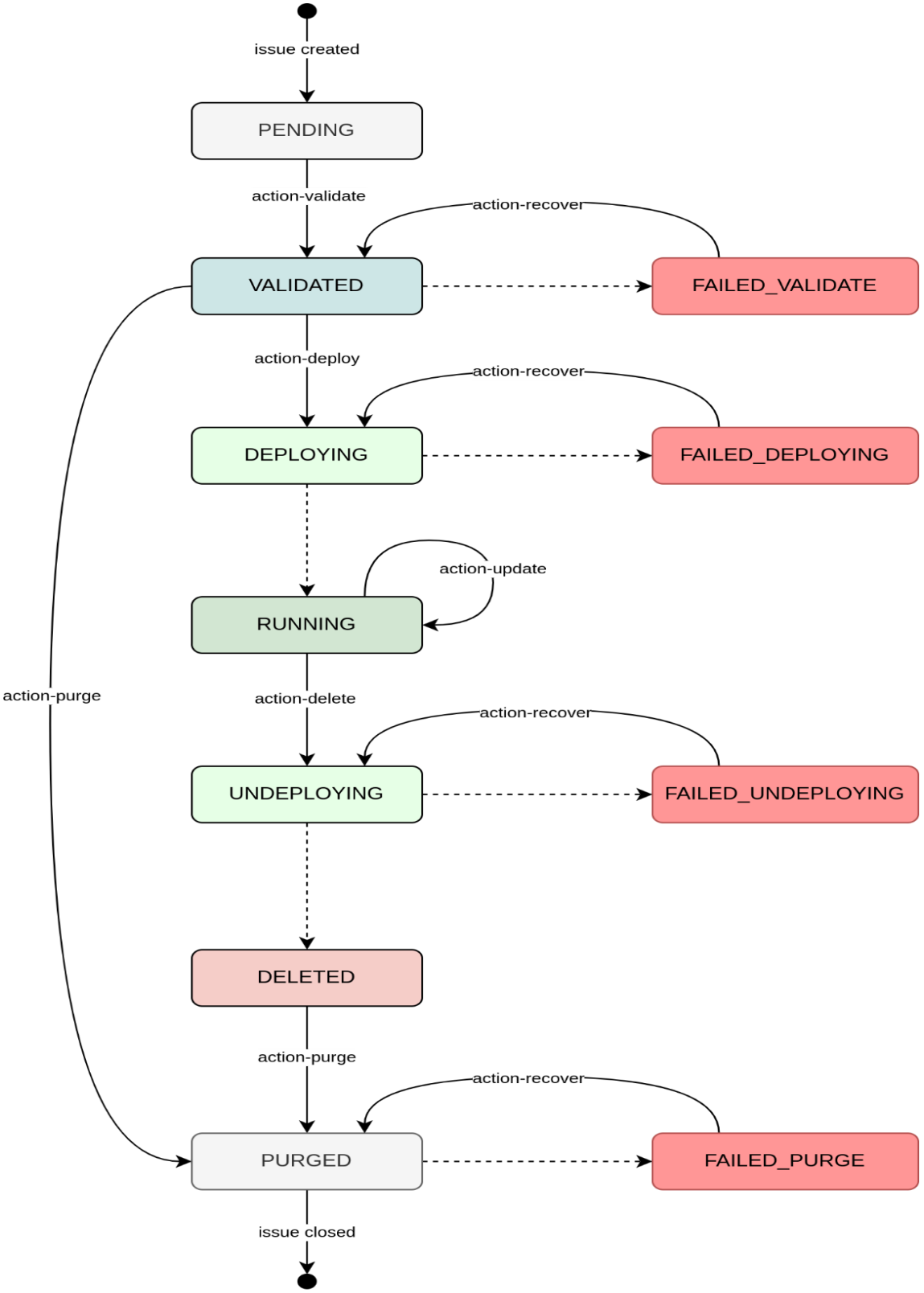

Figura 6. Diagrama del ciclo de vida AWS-Autodeploy.

5.2.1. *Estados de AWS-Autodeploy*

La tabla a continuación describe los estados de AWS-Autodeploy. Se ha utilizado los mismos colores que se encuentran en las *labels* del repositorio de GitHub:

Estado	*Descripción*
PENDING	Estado inicial de AWS-Autodeploy. Seguirá en este estado hasta que un usuario realice un acción de validación.
VALIDATED	La plantilla de la *Issue* (y sus valores) ha sido validada correctamente y está lista para ser desplegada.
DEPLOYING	La plantilla de la *Issue* está siendo desplegada en el proveedor Cloud.
RUNNING	El despliegue ha sido satisfactorio y los servicios y arquitecturas están corriendo.
UNDEPLOYING	Los servicios y arquitecturas desplegados están siendo retirados (*undeployed*).
DELETED	Los servicios y arquitecturas desplegados han sido terminados (finalizados) en el proveedor Cloud.
PURGED	Los ficheros de configuración utilizados por AWS-Autodeploy han sido eliminados.
FAILED_VALIDATE	Ha ocurrido un error mientras se validaba la plantilla de la *Issue*.
FAILED_DEPLOYING	Ha ocurrido un error mientras se desplegaba los servicios y arquitecturas en el proveedor Cloud.
FAILED_UNDEPLOYING	Ha ocurrido un error mientras se retiraba (*undeployed*) los servicios y arquitecturas desplegados.
FAILED_PURGE	Ha ocurrido un error mientras se eliminaban los ficheros de configuración de AWS-Autodeploy.

Tabla 17. Tabla de los estados de AWS-Autodeploy.

5.2.2. *Acciones de AWS-Autodeploy*

La tabla a continuación describe las acciones que se pueden realizar en AWS-Autodeploy. Se ha utilizado los mismos colores que se encuentran en las labels del repositorio de GitHub:

Acción	*Descripción*
validate	Comprueba si la plantilla y los datos introducidos por el usuario en la *Issue* son válidos.
deploy	Despliega la infraestructura (servicios o arquitectura) detallados en la plantilla de la *Issue*.
update	Actualiza la configuración de servicios o arquitecturas ya desplegados en el proveedor Cloud.
delete	Retira los servicios y arquitecturas desplegados (*undeployed*) y los termina (finaliza).
purge	Elimina los ficheros de configuración creados por AWS-Autodeploy del repositorio.
recover	Intenta recuperar (regresar a un estado válido) la herramienta AWS-Autodeploy cuando se encuentra en un estado de error.

Tabla 18. Tabla de las acciones de AWS-Autodeploy.

5.3. ¿Dónde se ejecuta AWS-Autodeploy?

AWS-Autodeploy es una herramienta que facilita el despliegue automático de servicios en AWS (Amazon Web Services). Este proceso se lleva a cabo mediante la integración de GitHub Actions, una plataforma de CI/CD (Integración Continua y Despliegue Continuo) que permite automatizar flujos de trabajo directamente desde un repositorio de GitHub. A continuación, se explica detalladamente dónde y cómo se ejecuta AWS-Autodeploy.

5.3.1. Runners

GitHub Actions se ejecuta en **contenedores** o **máquinas virtuales** proporcionadas por GitHub, conocidos como ***runners*** [21]. Estos *runners* pueden ser de dos tipos principales:

- *Runners* **hospedados** por GitHub: Son máquinas virtuales proporcionadas y gestionadas por GitHub que ejecutan los flujos de trabajo de GitHub Actions. Estas máquinas están preconfiguradas con un entorno estándar que incluye herramientas y bibliotecas comunes, facilitando la configuración y el despliegue. Cada vez que se activa un flujo de trabajo, GitHub asigna una máquina virtual nueva y aislada para ejecutar las acciones definidas. Una vez completada la ejecución, la máquina virtual se descarta, asegurando así un entorno limpio para cada ejecución. Los *runners* hospedados por GitHub soportan múltiples sistemas operativos, incluidos Ubuntu, Windows y macOS.
- *Runners* **auto-hospedados**: Son máquinas que pueden estar en la infraestructura propia del usuario o en cualquier proveedor de nube, y que el usuario configura y mantiene. Estos *runners* permiten una mayor personalización y control sobre el entorno de ejecución. Los usuarios pueden instalar cualquier software adicional necesario y ajustar los recursos según sus necesidades específicas. Los *runners* auto-hospedados pueden ser útiles cuando se requieren recursos específicos o configuraciones personalizadas no disponibles en los *runners* hospedados por GitHub.

5.3.2. Requisitos y Configuración de los Runners

Para que los *runners* puedan ejecutar los flujos de trabajo de GitHub Actions, necesitan **cumplir** ciertos requisitos y configuraciones:

- **Requisitos de Sistema**: Dependiendo del tipo de *runner* (hospedado por GitHub o auto-hospedado), puede ser necesario contar con un sistema operativo compatible (Ubuntu, Windows, macOS) y *hardware* que cumpla con los requisitos mínimos de recursos (CPU, memoria, almacenamiento).
- **Conectividad**: Los *runners* deben tener acceso a Internet para descargar las acciones y dependencias necesarias desde GitHub y otros repositorios públicos o privados.
- **Permisos y Acceso**: Los *runners* deben estar configurados con las credenciales adecuadas para acceder a los repositorios y servicios que utilizarán durante la ejecución de los flujos de trabajo.

5.3.3. Instalación de software en los Runners hosteados por GitHub

GitHub proporciona *runners* hospedados que vienen con **varias** herramientas y bibliotecas preinstaladas para facilitar el proceso de CI/CD. Estos *runners* se configuran automáticamente y están disponibles en tres sistemas operativos principales: Ubuntu, Windows, y macOS. A continuación, se detalla el *software* preinstalado en cada uno de estos sistemas:

- **Lenguajes de Programación**: Node.js, Python, Ruby, Java, Go, .NET, PHP y Perl.
- **Herramientas de Construcción y Pruebas**: Maven, Gradle, npm, pip, pytest, JUnit, Ant y Make.
- **Herramientas de Automatización y Orquestación**: Docker, Kubernetes (kubectl), Terraform, Ansible y Packer.
- **Clientes de Servicios Cloud**: AWS CLI, Azure CLI, Google Cloud SDK y Heroku CLI.
- **Otras Utilidades**: Git, cURL, jq, wget, zip/unzip y tar.

El *software* preinstalado puede variar dependiendo del sistema operativo del *runner*. GitHub mantiene actualizadas las imágenes de los runners hospedados para asegurar que las versiones del *software* sean recientes y estén libres de vulnerabilidades conocidas. Las imágenes se actualizan regularmente y se publican en los repositorios de GitHub correspondientes.

5.3.4. Seguridad y Aislamiento en los Runners hosteados por GitHub

Los *runners* hospedados por GitHub proporcionan un entorno **aislado** y **temporal** para cada ejecución, asegurando que no haya interferencias entre diferentes ejecuciones de flujos de trabajo. Esto garantiza que cada ejecución comience en un entorno **limpio** y **predefinido**. La infraestructura de GitHub maneja automáticamente la creación y destrucción de estos entornos, minimizando el riesgo de contaminación o persistencia de datos entre ejecuciones.

5.3.5. Recomendaciones

AWS-Autodeploy es una herramienta gratuita que sigue la filosofía de código abierto (*open source*). Sin embargo, hay que tener en cuenta que el *runner* donde se ejecuta, puede no ser gratuito. Para garantizar una mejor experiencia con AWS-Autodeploy, se definen a continuación una serie de recomendaciones [22]:

- **Controlar el acceso** a la creación de *Issues* y *labels*: Limitar quién puede crear *Issues* o agregar *labels* en el repositorio. De esta manera, usuarios no autorizados no realizarán despliegues no deseados. Se pueden usar las configuraciones de permisos de GitHub para restringir estas acciones, agrupando a los usuarios en colaboradores, propietarios o externos.

- **Monitorear costos**: Tanto en GitHub como en AWS. Es recomendable configurar límites de cuotas en AWS para controlar el uso de recursos de los servicios o usar herramientas como CloudWatch para este propósito. Para GitHub, se puede usar la funcionalidad "Insights" o consultar en "Billing and plans" los recursos gastados.
- **Visibilidad del repositorio**: Si el repositorio es público, se pueden utilizar las acciones de GitHub de manera gratuita dentro de ciertos límites. En cambio, si es privado, GitHub ofrece un límite gratuito de minutos de ejecución para GitHub Actions, pero más allá de este límite, se incurre en costos adicionales.
- **Especificaciones del contenedor**: Los contenedores utilizados en GitHub Actions pueden tener diferentes especificaciones en términos de CPU, memoria y almacenamiento. A mayor demanda de recursos, mayores serán los costos asociados.
- **Sistema operativo**: GitHub ofrece distintos precios dependiendo del sistema operativo del contenedor. Windows es el sistema operativo más barato, debido a su menor costo de licencia y demanda. Seguidamente se encuentra macOS y como el más caro, es Linux, por su extensa popularidad.

5.4. Estructura Repositorio

La herramienta AWS-Autodeploy está alojada en un repositorio de GitHub. Ese repositorio tiene la siguiente estructura:

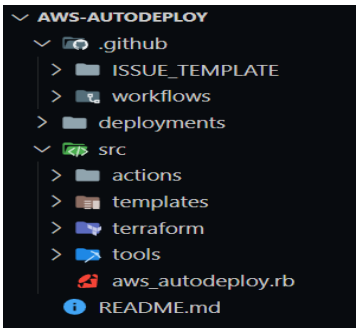

Figura 7. Repositorio de AWS-Autodeploy.

Como se puede observar, en la estructura se diferencian 3 carpetas principales: **'.github'**, **'deployments'** y **'src'**. Cada una de estas carpetas tiene un uso y propósito distinto, siendo cada una de ellas fundamentales e indispensables para la ejecución de AWS-Autodeploy.

5.4.1. Carpeta .github

La carpeta **'.github'**, como se ha mencionado en apartados anteriores, es impuesta por el propio GitHub para poder utilizar y ejecutar sus ***workflows***. Así pues, dentro de esta carpeta se encuentran definidos e implementados estos *workflows*. En este caso sólo existe 1 *workflow*, llamado 'aws-autodeploy.yaml'.

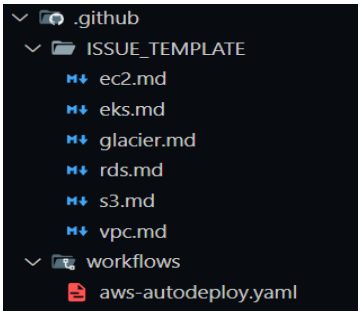

Figura 8. Contenido del directorio '.github' de AWS-Autodeploy.

Así mismo, AWS-Autodeploy usa **plantillas** predefinidas para las *Issues*. Para que estas plantillas aparezcan al crear una *Issue*, es indispensable que se sitúen dentro de la carpeta '.github', concretamente en el directorio 'ISSUE_TEMPLATE'. Actualmente la herramienta AWS-Autodeploy cuenta con plantillas para instanciar los siguientes servicios de AWS:

- Servicio EC2 (*Elastic Compute Cloud*) [27].
- Servicio EKS (*Elastic Kubernetes Service*) [30].
- Servicio Glacier (*Simple Storage Service (S3) Glacier*) [32].
- Servicio RDS (*Relational Database Service*) [29].
- Servicio S3 (*Simple Storage Service*) [28].
- Servicio VPC (*Virtual Private Cloud*) [31].

5.4.2. Carpeta Deployments

La carpeta **'deployments'** contiene los archivos de configuración de los distintos despliegues de infraestructura que se han realizado con la herramienta AWS-Autodeploy. Por consiguiente, hasta que no se realice un primer despliegue, el directorio estará vacío. En la imagen mostrada, se han configurado algunos despliegues para mostrar cómo quedaría el directorio tras unos despliegues. Cada despliegue estará identificado por el identificador que se le ha asignado a la *Issue* que se ha creado para ello. Por ejemplo, el directorio '385', contiene los archivos de configuración creados al lanzar la *Issue* con identificador '385'.

Figura 9. Contenido del directorio 'deployments' de AWS-Autodeploy.

Estos **archivos** de **configuración** son los que se crean con Terraform:

- '**provider.tf**': Este archivo contiene la configuración del proveedor de infraestructura en la nube que se utilizará para el despliegue. Define el proveedor y utiliza los secretos configurados en el repositorio para autenticarse. Esto incluye credenciales de acceso y región, entre otros parámetros necesarios para la comunicación con AWS.
  - Motivo de no inclusión: Este archivo contiene información sensible como claves de acceso de AWS, por lo que no se sube para evitar problemas de seguridad.
- '**directorio .terraform**': Este directorio es creado por Terraform y contiene archivos de configuración y estado que Terraform utiliza internamente. Incluye la caché del estado y la configuración del *backend*, entre otros archivos necesarios para que Terraform funcione correctamente.
  - Motivo de no inclusión: Para reducir el tamaño del repositorio y porque los archivos en este directorio se pueden regenerar fácilmente.
- '**plantillas servicio.tf**': Contienen el código de Terraform necesario para el despliegue de servicios específicos en AWS. Definen los recursos de AWS a desplegar, como instancias EC2, VPCs, S3, etc.
  - Motivo de no inclusión: Estos archivos se han eliminado para simplificar la implementación de la acción 'UPDATE' y que no hubieran colisiones al crear nuevas plantillas actualizadas.

- '**plant.txt**': Este archivo es la salida del comando *terraform plan*, que muestra una vista previa de los cambios que se realizarán en la infraestructura cuando se aplique la configuración. Incluye detalles sobre los recursos que se crearán, modificarán o destruirán.
    - Motivo de no inclusión: Similar a los archivos de plantillas, este archivo se elimina para mantener el repositorio ligero. Además, una vez revisado el plan, no es necesario para las etapas posteriores del despliegue.
- '**.terraform.lock.hcl**': Archivo de bloqueo que asegura que Terraform utilice las versiones exactas de los proveedores y módulos que se resolvieron durante la inicialización. Define versiones específicas de los proveedores y módulos, asegurando que las futuras ejecuciones sean consistentes.
    - Motivo de inclusión: Este archivo se sube al repositorio para asegurar que cualquier persona que clona el repositorio utilice las mismas versiones de proveedores y módulos, garantizando así que puedan reproducir el entorno.
- '**terraform.tfstate**': Es el archivo más importante, ya que almacena el estado de la infraestructura desplegada. Terraform utiliza este archivo para realizar un seguimiento de los recursos que ha creado y su estado actual. Incluye detalles de todos los recursos gestionados por Terraform, sus IDs y configuraciones actuales, permitiendo a Terraform realizar actualizaciones en la infraestructura.
    - Motivo de inclusión: Este archivo es crucial para cualquier operación de Terraform posterior, ya que sin él, Terraform no podría gestionar la infraestructura correctamente.

5.4.3. Carpeta src

La carpeta **'src'** es la carpeta principal donde se encuentran los *scripts* usados para la implementación de AWS-Autodeploy.

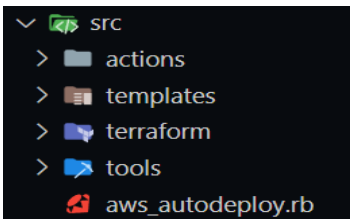

Figura 10. Contenido del directorio 'src' de AWS-Autodeploy.

El ***script*** 'aws_autodeploy.rb' constituye la lógica principal de la herramienta AWS-Autodeploy.

- Este *script* es el encargado de gestionar las acciones, mandar su ejecución y reportar los resultados nuevamente a la *Issue*. Además, se encarga de configurar y usar las herramientas propias de AWS-Autodeploy.

El directorio **'actions'** incluye los *scripts* de las diferentes acciones que gestiona AWS-Autodeploy.

- Cada acción parte de una clase padre llamada Action (**'action.rb'**), la cual define los métodos esenciales que todas las acciones deben implementar.
- El resto de *scripts* (****_action.rb*) implementan las acciones que se pueden activar mediante *labels* en una *Issue* de GitHub.
- Las acciones, a pesar de deber implementar una serie de métodos obligatoriamente, son independientes entre sí, y cada una puede utilizar las herramientas y *scripts* que necesite.

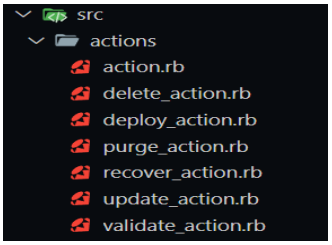

Figura 11. Contenido del directorio 'actions' de AWS-Autodeploy.

El directorio **'templates'** tiene las distintas acciones que se aplican a las plantillas predefinidas en el directorio '.github'.

- Actualmente la herramienta de AWS-Autodeploy sólo cuenta con el *script* de validación sobre los parámetros introducidos en la plantilla de la *Issue*, llamado 'validate_template.rb'

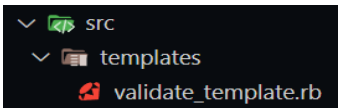

Figura 12. Contenido del directorio 'templates' de AWS-Autodeploy.

El directorio **'terraform'** contiene los *scripts* y archivos de configuración necesarios para que AWS-Autodeploy pueda usar Terraform. Estos archivos son fundamentales para gestionar infraestructuras como código de manera eficiente y automatizada.

- El *script* principal (**'terraform.rb'**) contiene la implementación de los métodos y funcionalidades utilizados por Terraform, como 'init', 'plan', 'apply' o 'destroy', esenciales para el ciclo de vida de la infraestructura definida por código.
- El archivo **'aws_provider'** configura un proveedor específico de Terraform utilizando las credenciales del usuario obtenidas en los secretos del repositorio.
- El resto de archivos en este directorio conformarán las **plantillas** de **configuración** para los servicios y arquitecturas que serán desplegados. Estas plantillas contendrán los valores y configuraciones específicos que el usuario haya configurado para cada despliegue.

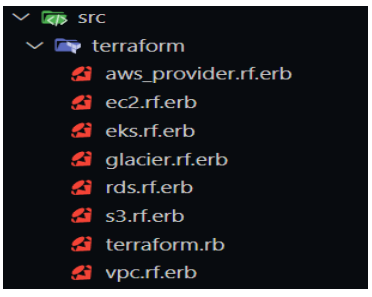

Figura 13. Contenido del directorio 'terraform' de AWS-Autodeploy.

El directorio **'tools'** alberga los *scripts* de las diversas herramientas que se han creado para facilitar la implementación de AWS-Autodeploy.

- La herramienta **Log** ('log.rb') simplifica la gestión de registros con un diseño flexible y eficiente en Ruby. Permite personalizar el formato del registro incluyendo fechas y niveles de severidad, y proporciona métodos dinámicos para registrar diferentes niveles de información como errores, información o depuración, facilitando así el seguimiento del código en los *outputs* de los *workflows*.

- La herramienta **GitHub Client** (‘github_client.rb’) simplifica las interacciones con la API de GitHub [16] para gestionar repositorios e *Issues* de manera eficiente desde Ruby. Actualmente, está diseñada para obtener información sobre las *Issues* de un repositorio, para realizar confirmaciones (*commits*) de código de manera automática dentro de un *workflow* de GitHub y para eliminar el directorio de despliegue de la *Issue*.
- La herramienta **Parameterizer** (‘issue_params.rb’) simplifica la extracción y organización de los parámetros introducidos por el usuario en las plantillas de la *Issue*. Su objetivo es dar un formato específico a estos parámetros, para que el resto de métodos puedan usarlos adecuadamente.
- La herramienta **FileManager** (‘file_manager.rb’) simplifica la gestión de archivos en entornos Ruby, permitiendo crear directorios, guardar archivos y cambiar temporalmente de directorio de manera eficiente.Sus funciones facilitarán las implementación del resto de métodos de AWS-Autodeploy.

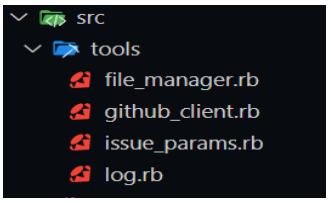

Figura 14. Contenido del directorio ‘terraform’ de AWS-Autodeploy.

5.5. Plantillas de AWS-Autodeploy

Los servicios y arquitecturas que se despliegan con AWS-Autodeploy están definidos en plantillas preconfiguradas almacenadas en GitHub. Aunque su uso no es obligatorio, su utilización facilita enormemente una configuración correcta.

Las plantillas de **servicios** contienen todos los parámetros que pueden ser configurados para cada servicio. En la parte superior, a través de un comentario, se proporciona un ejemplo de cómo completar los campos y posibles consideraciones a tener en cuenta, como restricciones o campos incompatibles, entre otros. Las plantillas de servicios, en general, no son modificables. Es decir, no se recomienda agregar más campos de configuración, principalmente porque utilizan plantillas de Terraform predefinidas que no incluyen dichos campos.

Las plantillas de **arquitecturas** incluyen los parámetros necesarios para desplegar la arquitectura correspondiente. Estas plantillas no ofrecen asistencia sobre cómo completar los campos, por lo que es una buena práctica desplegar primero algunos servicios antes de una arquitectura. Las plantillas de arquitecturas son modificables dependiendo de los servicios configurados. Es decir, se pueden añadir los parámetros de nuevos servicios si se desea ampliar la arquitectura correspondiente.

En adición, el usuario podrá crear sus propias plantillas:

Crear plantillas de **servicios** es un proceso complejo debido al trabajo. Para elaborar una plantilla, el usuario primero debe configurar el nuevo servicio en el código base, establecer validaciones para sus parámetros y, finalmente, crear un archivo de configuración Terraform que utilice variables para los parámetros definidos en la plantilla. Por esta razón, no se recomienda la creación de plantillas para nuevos usuarios poco experimentados.

Por otro lado, las plantillas de **arquitectura** son más fáciles de crear. La herramienta internamente recoge las plantillas de configuración Terraform de cada servicio y las despliega

conjuntamente en el proveedor configurado. Para crear una nueva arquitectura que no utilice nuevos servicios, el usuario debe seguir el ejemplo de otras plantillas de arquitectura: especificar los campos necesarios para cada servicio. Si se desea agregar una arquitectura con un servicio no existente, primero deberá crear la plantilla de dicho servicio y luego, agregar los parámetros necesarios de ese servicio en la plantilla de la arquitectura.

5.6. Interfaz gráfica

La interfaz gráfica para AWS-Autodeploy tiene como objetivo facilitar y acercar el uso de la herramienta a personas que no están familiarizadas con el entorno de GitHub. Así pues, para poder lograr su objetivo, la interfaz debe ser minimalista, simple e intuitiva. Para implementarla, se han tomado una serie de decisiones de diseño detalladas en este apartado.

5.6.1. Arquitectura

La interfaz gráfica sigue un modelo de implementación donde toda la lógica se ejecuta en el lado del cliente. Esto significa que no hay un servidor *backend*; todos los cálculos y operaciones se realizan en el navegador del cliente. Este modelo trae consigo las siguientes ventajas:

- **Reducción de Costos**: No se necesitan servidores dedicados, lo que reduce costos de infraestructura (incluyendo la adquisición y el mantenimiento). Asimismo, no hay necesidad de pagar por servicios de alojamiento, bases de datos, o servidores en la nube, lo que se traduce en un ahorro significativo a largo plazo.
- **Simplicidad de Implementación**: Al no requerir un *backend*, la arquitectura es más simple y fácil de mantener. Además, con toda la lógica ejecutándose en el cliente, hay menos dependencias de servicios externos y tecnologías de *backend*, lo que facilita el desarrollo y la resolución de problemas.
- **Mejor Escalabilidad (horizontal)**: Dado que cada usuario ejecuta la lógica de la aplicación en su propio navegador, la aplicación escala de manera natural con el aumento de usuarios sin requerir servidores adicionales.
- **Mayor Seguridad**: Los datos sensibles pueden manejarse directamente en el navegador del usuario sin tener que ser enviados a un servidor. Desde la seguridad a la hora de almacenar los datos en el servidor, como en la encriptación y cifrado a la hora de enviar estos datos hacia el servidor

Debido a la ausencia de un servidor *backend*, la aplicación no utiliza una base de datos tradicional. Para poder lograr que la aplicación sea funcional sin tener una base de datos, se ha definido el siguiente diseño:

- **GitHub como** "**Base de Datos**": Se utilizará GitHub para almacenar y gestionar datos. Cuando se necesitan datos sobre las plantillas para ofrecerlas al cliente, se hará una petición mediante la API de GitHub al repositorio del cliente. De esta manera, al actualizar o rellenar los campos de una issue, todo se persistirá mediante llamadas a la API que creen o actualicen las *Issues* en GitHub.
- **Datos Mínimos Requeridos:** Los usuarios deberán proporcionar únicamente su nombre de usuario, nombre del repositorio y un *token* de acceso para realizar las llamadas a la API de GitHub correctamente, ya que estas requieren permisos específicos.
- **Almacenamiento en el Navegador**: Se utiliza *localStorage* para guardar temporalmente la información en el navegador del usuario. Esto asegura que los datos persistan entre sesiones de la aplicación hasta que el usuario los elimine o actualice.

Este modelo trae consigo las siguientes ventajas adicionales:

- **Persistencia Local**: Los datos guardados en el navegador permanecen disponibles incluso después de cerrar la aplicación, mejorando la experiencia del usuario.
- **Rendimiento Mejorado**: Al manejar los datos localmente, se reducen las latencias asociadas a las consultas de bases de datos remotas. Del mismo modo, al procesar y almacenar datos en el cliente, se reduce la carga en servidores externos.
- **Facilidad de Uso**: Los usuarios pueden empezar a utilizar la aplicación rápidamente sin necesidad de configuraciones complejas o instalaciones adicionales.

5.6.2. React y sus componentes

La interfaz gráfica de la aplicación se implementa utilizando React, una **biblioteca** de JavaScript ampliamente utilizada para construir interfaces de usuario. React se basa en **componentes**, que son bloques reutilizables de código que representan diferentes partes de la interfaz. Cada componente en React es una pieza **autónoma** de la interfaz de usuario que gestiona su propio estado y lógica. De esta manera, los desarrolladores pueden construir interfaces complejas a partir de componentes más pequeños y manejables [23].
Un **componente** en React puede ser tan simple como un botón o tan complejo como una vista completa. Los componentes pueden recibir datos a través de "props" (propiedades) y gestionar su propio estado interno usando "state". Además, React utiliza un Virtual DOM para actualizar la interfaz de manera **eficiente**, minimizando las manipulaciones directas del DOM real y mejorando así el rendimiento.

Para gestionar la **navegación** entre las distintas ventanas de la aplicación, se utiliza R***eact Router*** [24]. *React Router* es una biblioteca que permite manejar la navegación y las rutas dentro de una aplicación React. A través de React Router, los desarrolladores pueden definir qué componentes deben renderizarse en función de la URL actual. Esta configuración de rutas se suele hacer en un archivo centralizado ("router.jsx"). En este fichero "**router.jsx**", se definen **todas** las rutas de la aplicación utilizando componentes de *React Router* (como *Route* o *Link*).

5.6.3. Funcionalidad

El **primer componente "Login"** (con el **path** /): En este componente se solicita el nombre de usuario, el nombre del repositorio y el *token* de acceso del usuario. Estos datos son necesarios para interactuar con la API de GitHub.

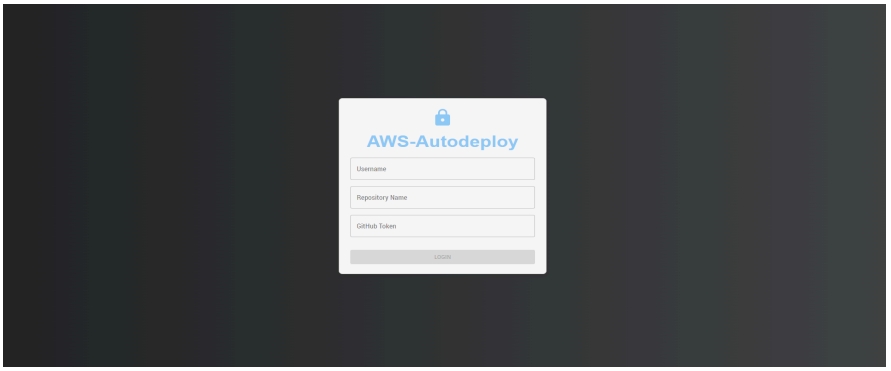

Figura 15. Contenido del componente "Login" de la interfaz gráfica de AWS-Autodeploy.

El **segundo componente "Templates"** (con el **path /home**): En esta ventana se muestran las plantillas disponibles en el repositorio, formando un *grid* de tres por tres. El usuario puede acceder a cualquier plantilla haciendo clic sobre ella.

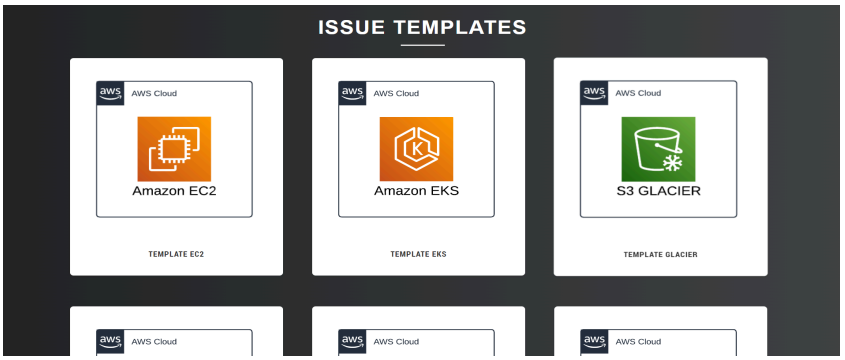

Figura 16. Contenido del componente "Templates" de la interfaz gráfica de AWS-Autodeploy.

El **tercer componente "Issue"** (con el **path /issue/:templateName**): En este formulario aparecen todos los campos de la plantilla a completar, similar a un formulario tradicional. Una vez que el usuario haya completado los campos necesarios, puede validar y enviar la información. Es en este momento cuando se crea una nueva *Issue* en GitHub de forma **automática**, asignándole como nombre la **fecha actual**. Esto simplifica el proceso para el usuario, quien no necesita preocuparse por los detalles técnicos de la implementación en GitHub.

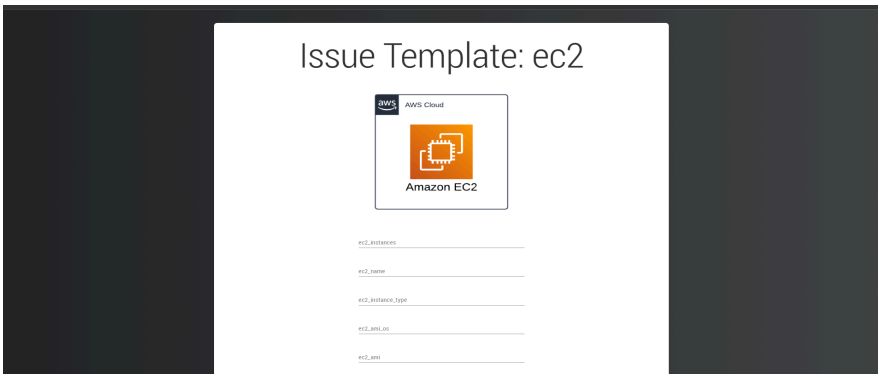

Figura 17. Contenido del componente de *Issues* de la interfaz gráfica de AWS-Autodeploy.

El **cuarto componente "Actions"** (con el **path /issue/:templateName/:issueNumber**): En esta ventana se muestra el estado de ejecución de la última acción realizada sobre la *Issue*. Se puede observar el éxito o fracaso de la ejecución y ejecutar nuevas acciones mediante los botones en la parte inferior. Los comentarios sobre la *Issue* se actualizan con una tasa de refresco configurable.

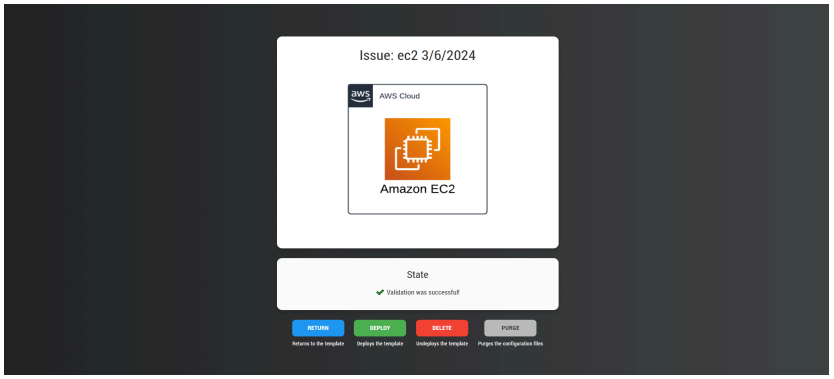

Figura 18. Contenido del componente de *Issues* de la interfaz gráfica de AWS-Autodeploy.

6. Implementación

AWS-Autodeploy está implementado bajo tres principales tecnologías, como son GitHub, Ruby y Terraform, cada una de ellas desempeñando un papel fundamental en el correcto funcionamiento de la herramienta. GitHub proporciona el entorno, acciones y *workflows* necesarios para poder automatizar los despliegues. Ruby ofrece un lenguaje de programación versátil, con el cual poder implementar la lógica de la herramienta, así como la integración con diversas bibliotecas para simplificar el desarrollo. Finalmente, Terraform es esencial para poder gestionar los despliegues, permitiendo manipular la infraestructura desde código (*IaC*).

6.1. Workflow

Como se ha mencionado con anterioridad, AWS-Autodeploy funciona con un ***workflow*** de GitHub, que se encuentra definido bajo el directorio 'workflow' dentro de '.github'. El *workflow* actúa como un *script* automatizado, que especifica las acciones a ejecutar en respuesta a eventos específicos dentro del repositorio. En este apartado, se detalla la configuración y apartados que conforman el *workflow* principal de AWS-Autodeploy.

6.1.1. Configuración del workflow

En primer lugar, el *workflow* se debe configurar para indicar sobre qué eventos que sucedan en el repositorio debe actuar. En este caso particular, el evento especificado es cuando exista una *Issue* con *labels*.

```
name:
aws-autodeploy-workflow
on:
  issues:
    types:
      - labeled
```

Figura 19. Código de la configuración del *workflow*.

Esto significa que cuando a un *Issue* se le asigne una *label*, el *workflow* definido bajo el nombre 'aws-autodeploy-workflow' será activado, permitiendo la ejecución de las *actions* y *steps* definidos en en los siguientes apartados.

6.1.2. GitHub Actions y Steps del workflow

Los siguientes apartados de un *workflow* son la definición e implementación de sus GitHub Actions, con sus *steps* y configuraciones. El *workflow* de **'aws-autodeploy-workflow'** cuenta con una única GitHub Action, la cual tiene un total de cinco *steps*. Así pues, por cada *Issue* con *label* creada o modificada en el repositorio, se lanzará una única GitHub Action.

Cada *step* dentro de esta *action*, está diseñado para un tarea distinta. A continuación, se detallan los cinco *steps* que conforman esta *action*. Para una mejor comprensión y legibilidad, se han omitido fragmentos del código (indicados con '[...]'), dejando únicamente las partes más relevantes.

Checkout: este primer *step* asegura que el código del repositorio esté disponible para su procesamiento, realizando la operación de *checkout* en el entorno de ejecución (el contenedor de GitHUb). Es decir, obtiene una copia del código fuente del repositorio y la copia en el entorno donde se ejecutará el *workflow*. Este paso es esencial para que el resto de *steps* tengan acceso al código y puedan realizar las acciones necesarias correctamente.

Terraform (condicional): configura Terraform, pero sólo en los casos indicados para mejorar el rendimiento. Estos casos son las acciones 'deploy', 'delete', 'recover' y 'update', ya que son las únicas que ejecutan código Terraform. Si este paso no se realiza, las variables de entorno y la configuración de Terraform no estarán preparadas. Esto podría llevar a errores como no poder ejecutar código Terraform, errores en la manipulación de la infraestructura de AWS o no detectar los cambios en la infraestructura que se hayan podido producir.

Set up Ruby: Prepara el entorno de ejecución con la versión específica de Ruby necesaria para ejecutar el código del proyecto. Aquí se puede especificar una versión concreta de Ruby. Esto es importante porque diferentes versiones de Ruby pueden tener diferentes características y comportamientos, y establecer una versión particular para todos los usuarios, garantiza que no haya problemas de configuración.

Install Ruby dependencies: Instala las dependencias de Ruby requeridas para el correcto funcionamiento de la herramienta. Una gema en Ruby es una biblioteca o un conjunto de funcionalidades que se pueden incluir en un proyecto para extender su funcionalidad. Al instalar estas dependencias, se asegura que el código tenga acceso a las funcionalidades necesarias para interactuar con servicios externos como los clientes de AWS, clientes de GitHub o métodos especiales con salidas de tres *outputs*.

Run main action: Ejecuta la acción principal del despliegue automático, gestionada por el *script* principal 'aws_autodeploy.rb'. En este *step* se le pasan a este *script* principal parámetros relevantes como el nombre de la *label*, el número de la *Issue*, la lista de servicios disponibles para AWS-Autodeploy, entre otros. Así mismo, en este *step* también se envían, como variables de entorno, los secretos del repositorio de GitHub que el usuario ha configurado previamente.

```yaml
jobs:
  aws-autodeploy-workflow:
    if: startsWith(github.event.label.name, 'action-')
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v3
        [...]

# Setup terraform (only needed for action-deploy and action-delete)
      - name: Terraform
        uses: hashicorp/setup-terraform@v3
        [...]
      # Setup Ruby
      - name: Set up Ruby
        uses: ruby/setup-ruby@v1
        [...]
      # Setup gems
      - name: Install Ruby dependencies
        run: |
          gem install aws-sdk-ec2 aws-sdk-rds [...]
      # run aws_autodeploy.rb
      - name: Run main action
        env:
          GH_TOKEN: ${{ secrets.GH_TOKEN }}
          ACCESS_KEY_ID: ${{ secrets.ACCESS_KEY_ID }}
          SECRET_ACCESS_KEY: ${{ secrets.SECRET_ACCESS_KEY }}
          [...]
        run: |
          ruby aws_autodeploy.rb [...] "ec2,s3,rds"
```

Figura 20. Fragmento reducido del código de la GitHub Action con sus *steps*.

6.2. Plantillas

Una plantilla es un **modelo predefinido** que sirve como base para crear instancias o configuraciones específicas de forma rápida y consistente. Estas plantillas están diseñadas para simplificar y estandarizar el proceso de configuración de los servicios y arquitecturas disponibles en la herramienta, ofreciendo los conjuntos de parámetros que el usuario deberá rellenar.

Cuando se crea una nueva *Issue* en el repositorio de GitHub, aparece un **listado** con todas las plantillas disponibles. Esto es posible gracias a que están implementadas bajo el directorio 'ISSUE TEMPLATES'. Una vez el usuario escoge una plantilla del listado, se le proporciona un **formulario** estructurado con los campos que deberá rellenar con la información necesaria que él desee. Algunos ejemplos de estos campos son el número de instancias, el tipo de instancia, el sistema operativo, la configuración de red, los *tags*, entre otros.

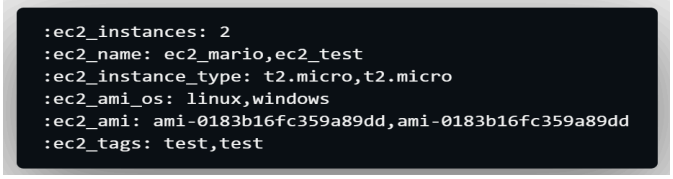

Figura 21. Campos a rellenar en la plantilla del servicio EC2.

En el ejemplo proporcionado, la plantilla "EC2 Instance" está diseñada para crear **instancias EC2** en AWS. Los campos a rellenar están delimitados por dos puntos. A continuación del campo, el usuario introduce valores para cada campo con el fin de configurar a su gusto el servicio. En caso de que el usuario quiera lanzar más de una instancia del servicio, deberá asignar tantos valores como números de instancias desee en cada campo, separados por comas.

6.3. Script Principal

El *script* principal **'aws_autodploy.rb'** se divide en varias partes distintas, cada una desempeñando un papel fundamental en el funcionamiento de AWS-Autodeploy.

6.3.1. Configuraciones iniciales

En la primera sección, se establecen las configuraciones iniciales de las herramientas y entornos necesarios para que las acciones de AWS-Autodeploy puedan ejecutarse de manera efectiva y llevar a cabo sus tareas.

En primer lugar, se comienza configurando la **herramienta** de **registro** 'Log'. Una acción que se repite por norma general en el resto de *scripts*, ya que cada *script* debe definir su propio Log. En el caso del *script* principal, el Log se identifica como 'MAIN'.

A continuación, se procede a configurar un **cliente** de **GitHub** utilizando la biblioteca 'octokit'. Como se explica en el apartado de la herramienta GitHub Client, inicialmente no se planeaba utilizar herramientas externas para esta configuración, pero debido a la complejidad y su bajo rendimiento, se llevó a cabo la decisión de adoptar estas librerías, las cuales simplifican significativamente el desarrollo. Como muestra de ello, la configuración de un cliente en este *script* se logra con tan solo una línea de código.

Posteriormente, se **obtiene** la **configuración** de la *Issue*. En este apartado, se recopila información necesaria para el despliegue, como el número de la *Issue* y el nombre del repositorio en un formato específico ('UsuarioGitHub/NombreRepositorio'). Asimismo, se extrae el contenido de la *Issue*, esta vez sí, utilizando la herramienta propia de cliente de GitHub de AWS-Autodeploy.

6.3.2. Tratar los parámetros

Por último, antes de proceder con la ejecución de las acciones programadas, el *script* principal se encarga de obtener los parámetros de las plantillas completadas por el usuario.

AWS-Autodeploy está diseñado para admitir la creación de **múltiples servicios** con **múltiples instancias** de cada uno, lo que requiere una estructura flexible para manejar estos datos de manera eficiente. Para lograr esta flexibilidad, los parámetros se organizan como un **conjunto de mapas**. En otras palabras, el conjunto total de parámetros de cada servicio se estructura como un mapa, y cada mapa, que corresponde a un servicio específico, se guarda dentro de un conjunto global de parámetros. Es importante destacar que cada servicio tiene asignado una **posición específica** en el conjunto donde se almacenará su mapa de parámetros. Estas posiciones están definidas por el conjunto de servicios que se utilizarán en el despliegue actual.

Por ejemplo, si en el despliegue actual solo se usarán los servicios EC2 y S3, y estos servicios están definidos en ese mismo orden en el conjunto de servicios, el mapa de parámetros de EC2 se guardará en la posición 0, mientras que el de S3 se guardará en la posición 1.

A su vez, cada mapa de servicio contiene como clave los diferentes campos de la plantilla, y cada uno de estos campos (*keys*) contiene como valor un arreglo (*array*) con las distintas opciones proporcionadas por el usuario para ese campo en particular.

Para configurar los parámetros de esta manera, el proceso se divide en **tres fases** fundamentales:

6.3.2.1. Fase 1: obtener los parámetros

En la **primera**, se toma el cuerpo de la *Issue* previamente obtenido y se transforma en un mapa. En este mapa, cada clave (*key*) representa un campo de la plantilla, y su valor es un arreglo (*array*) que contiene los valores introducidos por el usuario.

```
Mapa Inicial con todos los parámetros (key: campo_plantilla, value:array_values)
{:ec2_instances=>["2"], :ec2_name=>["ec2_1", "ec2_2"], :ec2_ami_os=>["linux", "windows"],
:s3_instances=>["1"], :s3_name=>["my-bucket"], :s3_tags=>["test"], :vpc_instances=>["1"],
:vpc_name=>["vpc-autodeployed"], :vpc_cidr_block=>["10.0.0.0/16"]}
```

Figura 22. Mapa inicial con los parámetros de la plantilla.

6.3.2.2. Fase 2: obtener los servicios

En la **segunda**, se identifican los servicios que se van a utilizar para el despliegue. Como se ha mencionado previamente, el conjunto de servicios disponibles para la herramienta AWS-Autodeploy se pasa como parámetro desde el *workflow*, con cada servicio separado por comas (,). Durante esta fase, se emplean las claves (*keys*) del mapa generado previamente para compararlas con el conjunto completo de servicios disponibles. La herramienta procede entonces a crear un nuevo conjunto que contiene exclusivamente los servicios que serán utilizados para el despliegue. De esta manera, se omiten aquellos servicios que no se utilizarán. Este conjunto de servicios será el que indique en qué posición deberán guardar su mapa.

```
Conjunto de Servicios disponibles en AWS-Autodpeloy -> [ec2,s3,rds,vpc,eks,glacier...]
Conjunto de Servicios que se usarán en el Despliegue -> [ec2,s3,vpc]
```

Figura 23. Conjunto de servicios disponibles en AWS-Autodeploy y conjunto de servicios a utilizar en el despliegue.

6.3.2.3. Fase 3: obtener el conjunto de parámetros con mapas

En la **tercera**, se ordenan las parejas clave-valor del mapa inicial según las posiciones de los servicios obtenidos en la etapa anterior, creando un mapa para cada servicio. Es decir, en esta fase, se agrupan todos los parámetros de un mismo servicio en un mismo mapa, y dicho mapa se inserta en el conjunto total de parámetros en la posición que le indique el conjunto de servicios disponibles.

```
Conjunto de Paramétros con Mapas por cada Servicio
{
    {:ec2_instances=>["2"], :ec2_name=>["ec2_1", "ec2_2"], :ec2_ami_os=>["linux", "windows"]},
    {:s3_instances=>["1"], :s3_name=>["my-bucket"], :s3_tags=>["test"]}
    {:vpc_instances=>["1"], :vpc_name=>["vpc-autodeployed"], :vpc_cidr_block=>["10.0.0.0/16"]}
}
```

Figura 24. Conjunto con varios mapas, uno por cada servicio, ordenados según el conjunto de servicios anterior.

6.3.3. Ejecución de las acciones

En la segunda sección del script principal **'aws_autodeploy.rb'**, se gestiona la ejecución de las acciones definidas para el despliegue automático en AWS. Como se explica en el siguiente apartado llamado 'Acciones', las acciones de AWS-Autodeploy siguen un modelo de herencia. Cada una de ellas implementa, a su manera, los métodos *execute* y *report*, que heredan de la clase padre 'Action'. Estos métodos son los que llamará el *script* principal.

En primer lugar, el *script* se encarga de instanciar la acción correspondiente basándose en la *label* de la *Issue* que recibe como argumento desde el *workflow*. A continuación, una vez que se ha instanciado la acción, se procede a ejecutarla invocando su método ***execute***. Este método requiere de varios parámetros, comunes (los mismos) para todas las acciones. En concreto, estos parámetros son:

- El número de la *Issue*. Este valor se recibe como argumento desde el *workflow*.
- El conjunto de mapas con los parámetros de los servicios con los valores introducidos por el usuario. Este dato se obtiene en la sección anterior.
- El *array* con los servicios que se utilizarán para el despliegue. Este dato se obtiene en la sección anterior.

Después de ejecutar la acción, se genera un informe sobre los resultados de dicha ejecución. Esto se consigue mediante el método ***report***, que devuelve la información sobre cómo ha ido la ejecución. El informe generado puede contener diferentes tipos de mensajes, cada uno precedido por un símbolo específico que indica el resultado de la ejecución:

- ✔ **Éxito**: Si la ejecución de la acción ha sido exitosa, se incluirá un mensaje de satisfacción precedido por el símbolo ✔.
- ✘ **Error**: En caso de que haya habido algún error durante la ejecución de la acción, el informe contendrá un mensaje precedido por el símbolo ✘. Además, se proporcionará un informe detallado con los errores que han ocurrido.
- ⚠ **Advertencia**: Si parte de la acción se ha ejecutado correctamente pero ha habido algún problema con otra parte de la acción, el informe incluirá un mensaje precedido por el símbolo ⚠. Esto indica que la acción ha continuado a pesar del problema, y se proporcionará información sobre lo que se ha ejecutado correctamente y lo que no.

Por ejemplo, un caso posible con este último tipo de mensaje es cuando se despliegan los servicios correctamente en AWS, pero no se puede realizar un *commit* de los archivos de configuración en el repositorio. En este caso, el informe contendrá los valores de los servicios desplegados por la herramienta, así como un mensaje con el error que se ha producido.

Finalmente, la última parte de esta sección del *script* principal es la **actualización** de las *labels* en la *Issue*. Este proceso se realiza en dos partes. En la primera, se recorren todas las *labels* que actualmente tiene la *Issue* actualmente y se eliminan una a una. Una vez hecho esto, en la segunda fase, se recorre el *array* devuelto por el método *execute*, el cual contiene todas las *labels* que deben ser incluidas en la *Issue*, y se las asignan. Ambas fases utilizan el cliente GitHub de la biblioteca la biblioteca 'octokit' para interactuar con la *Issue*, utilizando los métodos ***remove_label*** y ***add_labels_to_an_issue***.

6.4. Acciones

En el contexto de AWS-Autodeploy, se ha empleado el patrón de herencia, una práctica común en la programación orientada a objetos, para estructurar y modularizar la lógica de las acciones que pueden realizar despliegues automáticos en AWS.

La clase principal **'Action'**, sirve como una abstracción general que define la estructura y el comportamiento básico que deben tener todas las acciones. Esta clase dos métodos abstractos, ***execute*** y ***report***, que deben ser implementados por las clases hijas para realizar acciones específicas y generar informes sobre los resultados de dichas acciones, respectivamente.

El patrón de **herencia** permite la reutilización de código y su cohesión al definir una estructura común en la clase base. Las clases hijas, como *ValidateAction* o cualquier otra acción específica, podrán especializar su comportamiento heredando esta estructura. Además, facilita la extensibilidad del código, ya que nuevas acciones pueden agregarse simplemente creando nuevas clases que hereden de 'Action' y proporcionen sus propias implementaciones de *execute* y *report*. Esto facilita la incorporación de nuevas funcionalidades al sistema sin afectar el código existente y fomenta una arquitectura modular y mantenible.

Otro aspecto que aborda la clase base 'Action' es la implementación del método de clase ***self.for(action)***, llamado desde el *script* principal. Este método utiliza una estructura de control ***case*** para determinar qué clase de acción se debe instanciar según el tipo de acción proporcionado por parámetro. Asimismo, ofrece una forma flexible y extensible de crear instancias de diferentes tipos de acciones sin la necesidad de conocer los detalles de implementación de cada clase de acción. Facilita la adición de nuevas clases de acción al sistema simplemente agregando nuevos casos a la estructura de control case y proporcionando la implementación correspondiente para cada tipo de acción.

6.4.1. validate_action

La acción **'ValidateAction'** es la encargada de validar los parámetros introducidos por el usuario en las plantillas de las *Issue*. Mediante el uso de un *script* especializado en validaciones, esta acción verifica que los valores proporcionados cumplen con los patrones requeridos y, por lo tanto, son adecuados para el despliegue.

El método ***execute*** es el corazón de la acción 'ValidateAction'. En este método, se crea una instancia de **'ValidateTemplate'** para realizar las validaciones necesarias utilizando los parámetros ordenados y los servicios proporcionados. Los errores y advertencias resultantes se asignan a las variables de instancia '@errors' y **'**@warnings', respectivamente.

Si no se encuentran errores, se establece el atributo '@success' en *True*, indicando que la validación fue exitosa. En caso contrario, se establece '@success' en *False*, y se procede a generar un estado de fallo de validación. Finalmente, el método devuelve un array que contiene el **estado** resultante, ya sea 'state-validated' si la validación fue exitosa o 'state-failed-validate' si ocurrieron errores.

El método ***report*** proporciona un informe sobre los resultados de la validación realizada. Si la validación fue exitosa (**'@success'** es ***True***), se formatean las advertencias (si las hay) para que sean legibles y se devuelve un mensaje indicando que la validación fue exitosa, junto con las advertencias formateadas. En caso de que la validación haya fallado (**'@success'** es ***False***), se formatean los errores para que sean legibles y se devuelve un mensaje indicando que la validación falló, junto con los errores formateados.

6.4.2. deploy_action

La acción **'DeployAction'** es la encargada de instanciar y lanzar los servicios y arquitecturas de AWS-Autodeploy en el proveedor AWS. Para ello, utiliza herramientas como Terraform y un cliente de GitHub, y debe tener un control absoluto sobre qué servicios y arquitecturas se han creado en cada momento para no dejar al usuario con instancias colgadas en AWS sin control.

Similar a la acción anterior, pero en este caso más notable debido a que se ejecutan varios métodos y pueden suceder varios errores, la acción 'DeployAction' utiliza un ***array*** para ir guardando los posibles errores que puedan surgir en las diferentes ejecuciones.

Los métodos principales de la acción se encuentran bajo el bloque ***begin***. Estos son una serie de comandos que Terraform necesita ejecutar para poder instanciar servicios y arquitecturas en AWS. Como se explica en el apartado 'Terraform', se opta por llamar a estos métodos de manera similar a la sintaxis de Terraform, con el objetivo de mejorar la legibilidad del código y mantener coherencia con las convenciones establecidas. Posteriormente, se verifica si el **estado** de Terraform indica éxito o fallo. Este estado ('tf_state') indica si los servicios están disponibles o no en AWS. Si a partir de aquí sucediera cualquier otro error, en el método ***report*** se devolvería una advertencia (⚠), ya que los servicios sí que han podido ser desplegados.

Finalmente, se realiza un ***commit*** de los archivos de configuración de Terraform utilizando el cliente propio de GitHub de AWS-Autodeploy. A continuación, se establece el atributo '@success' en base a si el *array* '@errors' está vacío o no. Se devuelve un *array* que indica el estado resultante del despliegue, con 'state-deploying' y 'state-running' en caso afirmativo, y con 'state-failed-deploying' en caso negativo.

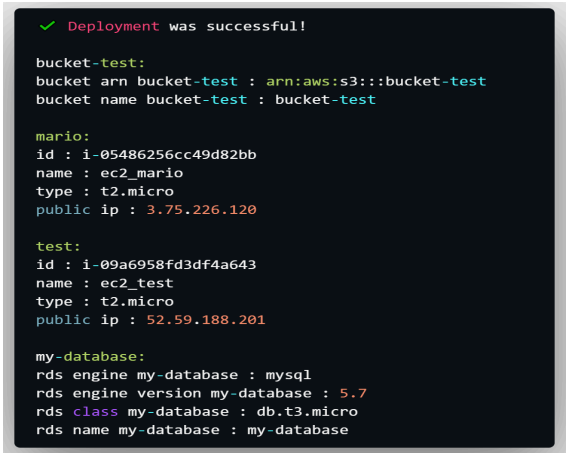

Figura 25. Ejemplo de salida de *outputs* de Terraform ordenados por servicios e instancias.

El método ***report*** genera un informe detallado sobre el resultado del despliegue. Los datos de salida utilizados son los que proporciona Terraform mediante sus ***outputs*** y permiten obtener información como IPs, ARN, entre otros, de los servicios y arquitecturas desplegados. Al dividir esta información (estos *outputs*) por servicios y por instancias de servicios, se ofrece una mejor presentación de los datos que facilita la comprensión al usuario. En este método, se definen tres posibles estados:

- Si **'@success'** y **'@tf_state'** son ***True***: Se indica que tanto los servicios y arquitecturas han sido desplegados correctamente en AWS, como que los archivos de configuración de Terraform han podido ser subidos al repositorio en su correspondiente directorio dentro de 'deployments' mediante un *commit*.
- Si **'@success'** es ***False*** pero **'@tf_state'** es ***True***: En este caso, se indica que los servicios y arquitecturas han sido desplegados correctamente en AWS, pero ha ocurrido un error al subir los archivos de configuración. Esta situación se muestra como una advertencia (⚠).
- Si **'@tf_state'** es ***False***: En consecuencia, **'@success'** también será ***False***. Esto indica que ha habido un error al desplegar los servicios y se ha abortado la acción DeployAction.

6.4.3. delete_aciton

La acción **'DeleteAction'** es la encargada de eliminar (terminar) las instancias de los servicios y arquitecturas previamente desplegados con la acción anterior. Para realizarlo, utiliza los parámetros que recibe de la *Issue* y los ficheros de configuración de Terraform, generados y subidos al repositorio en la acción anterior.

El método ***execute*** se encarga de realizar la eliminación de la infraestructura asociada a la *Issue* especificada. Dentro de un bloque *begin*, se llama al método **'provider'** de Terraform para configurar el proveedor de Terraform correspondiente a la *Issue*.

Es importante destacar que la configuración del proveedor de Terraform, en este caso AWS, no se incluye como archivo de configuración y, por lo tanto, no se sube al repositorio. Esto se hace deliberadamente para evitar **exponer públicamente** las **credenciales de acceso**, protegiendo así la seguridad de la infraestructura.

Una vez que el proveedor está configurado correctamente, se procede a ejecutar el método **'destroy'** de Terraform. Este paso desencadena la eliminación de la infraestructura según las especificaciones proporcionadas en los archivos de configuración. El éxito de esta operación se refleja en el atributo **'@success'**. Finalmente, se devuelve un *array* que indica el estado resultante de la eliminación, siendo 'state-undeploying' y 'state-deleted' en caso afirmativo, y 'state-failed-undeploying' en caso negativo.

La implementación del método ***report*** es muy sencilla. Si la eliminación fue exitosa ('@success' es *True*), simplemente devuelve un mensaje sencillo. En caso opuesto ('@success' es *False*), se formatean los errores para mejorar su legibilidad y se devuelven.

6.4.4. purge_action

La acción **'PurgeAction'** es la encargada de borrar (eliminar) definitivamente los ficheros de configuración asociados a una *Issue* específica. Este proceso mantiene la limpieza y la integridad del repositorio, eliminando archivos obsoletos o no deseados.

El método ***execute*** consta de una única invocación, al método **'delete_dir'** del cliente GitHub propio de AWS-Autodeploy. Este método, al que se le pasa como argumento el número de la *Issue*, será el encargado de borrar todos los ficheros de configuración relacionados con la *Issue*.

Si la eliminación de los archivos de configuración se completa sin errores, el atributo **'@success'** se establece en base a si el *array* **'@errors'** está vacío o no. Finalmente, se devuelve un array que indica el estado resultante de la eliminación, con 'state-purged' en caso afirmativo, y con 'state-failed-purge' en caso negativo.

El método ***report*** proporciona un breve informe sobre el resultado de la eliminación de los archivos de configuración. Si la operación fue exitosa (**'@success'** es ***True***), devuelve un mensaje indicando el éxito de la purga. En caso contrario (**'@success'** es ***False***), formatea los errores para una mejor legibilidad y los presenta junto con un mensaje de error.

6.4.5. recover_action

La acción **'RecoverAction'** es la encargada de recuperar una *Issue* que ha fallado y ha entrado en un estado de fallida. Esta acción intentará reconducir a al *Issue* nuevamente a un estado válido dentro del ciclo de vida de AWS-Autodeploy. Este proceso permite ofrece una manera efectiva para tratar los posibles errores que pueda tener la *Issue*.

En esta fase inicial del método ***execute***, la acción 'RecoverAction' realiza una serie de configuraciones para determinar en qué estado se encuentra la Issue. Estas acciones son:

- Utiliza el cliente de GitHub de la biblioteca Octokit para obtener el **conjunto de *labels*** de la *Issue*. Este conjunto puede contener *labels* de acción (action-), de estado válido (state-) o de estado de fallo (state-failed-*).
- Compara el conjunto de *labels* obtenido previamente con el conjunto predefinido por la acción llamado '**state_to_action'**. En este mapa, se establece para cada estado de fallo cuál es la acción que se debe ejecutar para salir de ese estado y regresar al ciclo de vida.
- Recorre cada etiqueta del conjunto de etiquetas hasta encontrar la etiqueta correspondiente al estado de fallo (**state-failed-***). Una vez encontrada, realiza la conversión y obtiene la acción que va a ejecutar, la cual se guarda en '**action_tag'**.

En la fase final del método ***execute***, se llevan a cabo las operaciones necesarias para recuperar el estado de la *Issue* y resolver cualquier posible fallo que haya ocurrido. Como norma general, la mayoría de los fallos con la herramienta AWS-Autodeploy pueden atribuirse a errores en los parámetros introducidos por el usuario, como valores inválidos sintácticamente o duplicados en AWS. Por consiguiente, las soluciones que el usuario puede intentar son revisar su infraestructura, revisar y actualizar los parámetros de la plantilla de la *Issue*… Teniendo en cuenta este flujo de trabajo, lo que primero hace 'RecoverAction', es realizar una serie de operaciones similares a las que se ejecutan en el *script* principal 'aws_autodeploy.rb'. Estas operaciones son necesarias para obtener los nuevos parámetros de la *Issue*, los cuales han sido potencialmente actualizados con algunas modificaciones por el usuario. Estos nuevos parámetros se almacenan en la variable '**new_ordered_params'**.

Una vez obtenidos los nuevos parámetros, se procede a ejecutar la acción indicada por 'action_tag'. Esto se logra mediante la llamada al método **Action.for(action_tag)**, que devuelve una instancia de la acción correspondiente. Inicialmente, se invoca el método *execute* de esta acción pasándole el número de la *Issue*, los nuevos parámetros y los servicios a usar en el despliegue. Posteriormente, se invoca el método *report* de la misma acción. El estado resultante de la ejecución de la acción se almacena en '**action_state'**. Esta variable contiene el conjunto de *labels* que deben asignarse a la *Issue*, reflejando el resultado de la acción ejecutada. Si 'action_state' contiene algún fallo, se establece '@success' en *False*. De lo contrario, se considera que la recuperación fue exitosa y se establece '@success' en *True*. Además, también se almacena la salida del método *report* para su posterior uso en 'RecoverAction'.

El método ***report***, nuevamente con una implementación sencilla, se divide en dos posibles mensajes. Si todo ha ido bien (‘@success’ es *True*), se reporta el informe obtenido previamente al invocar el método *report* de la acción especificado por ‘action_tag’. En caso de que se haya producido algún error (‘@success’ es *False*), se usa el informe de esa acción fallida, agregándole los errores producidos durante la ejecución de **‘RecoverAction’**.

6.4.6. update_action

La acción **‘UpdateAction’** es la encargada de administrar las actualizaciones en la infraestructura desplegada por AWS-Autodeploy. Esta acción está diseñada principalmente para permitir actualizaciones, como modificaciones de parámetros y configuraciones, en los servicios desplegados. Pese a ello, la acción ha sido implementada para poder soportar la creación de nuevos servicios como la eliminación de servicios. Para ello, en el método *execute* deberá realizar tanto la ActionValidate como la DeployAction.

En la fase inicial del método ***execute***, la acción ‘UpdateAction’ se inicia obteniendo los nuevos parámetros introducidos por el usuario. Posteriormente, manda ejecutar la acción ‘**ValidateAction’** para validar los posibles nuevos servicios o nuevas modificaciones que haya podido introducir el usuario. A continuación, se verifica si la ejecución ha sido satisfactoria comprobando si ‘action_state’ contiene el valor ‘state-validating’.

En la siguiente fase, se prepara el entorno para ejecutar la acción **'DeployAction'**. Esto incluye recrear nuevamente el proveedor de Terraform y configurar el entorno de manera adecuada. Una vez que todo está listo, se utiliza el método ‘**Action.for(action_tag)’** para poder ejecutar los métodos *execute* y *report* de 'DeployAction'. Durante esta ejecución, se obtiene un conjunto de *labels* que reflejan el resultado de la acción. Este conjunto se almacena como respuesta para el método *execute* de 'UpdateAction', ya que indican el estado de la acción. Además, se guarda el resultado del método *report* de 'DeployAction' para su posterior uso en el informe de 'UpdateAction'.

El método ***report*** presenta un informe simple sobre el resultado de la actualización. Si la actualización se realizó con éxito, se devuelve el informe generado por la acción ‘DeployAction’. En caso de fallo, se muestran los errores encontrados durante la ejecución de la acción 'UpdateAction’, ‘DeployAction’ y/o ‘ValidateAction’.

6.5. Terraform

Terraform es una herramienta de infraestructura como código (*IaC*) que permite definir y provisionar la infraestructura de manera automatizada. En el contexto de AWS-Autodeploy, la implementación de Terraform se realiza con dos tipos principales de archivos:

- **‘terraform.rb’**: Este archivo constituye el núcleo de la implementación de Terraform en el AWS-Autodeploy. Aquí se encuentran los definidos e implementados métodos esenciales como *init*, *apply*, *destroy*, entre otros. Es importante destacar que la implementación de estos métodos se ajusta específicamente a las necesidades y peculiaridades de AWS-Autodeploy, no siguiendo necesariamente la implementación estándar de Terraform.
- **Ficheros de plantillas (*.rf.erb)**: Estos archivos, con extensión ‘.rf.erb’, son plantillas en formato de texto que combinan código Ruby con texto estático. Utilizan marcadores de posición (‘<%= variable %>’), para indicar dónde se deben introducir los datos proporcionados por el usuario durante el despliegue. Estas plantillas permiten una configuración dinámica, facilitando la personalización y adaptación de la infraestructura según los requisitos específicos de cada despliegue.

6.5.1. Método provider

El método ***provider*** de 'terraform.rb' es el encargado de preparar el proveedor que se usará para el despliegue con Terraform. En el caso de AWS-Autodeploy, este proveedor es AWS. Para poder lograrlo, se encarga de configurar una plantilla especial, llamada **'aws_provider.rf.erb'**. Esta configuración es crucial para permitir a Terraform acceso a los recursos de la infraestructura en AWS y que así pueda gestionarlos correctamente.

El método inicia su ejecución leyendo el contenido de la plantilla especial llamada 'aws_provider.rf.erb', la cual contiene la **estructura** y **configuración inicial** requerida para el proveedor de Terraform destinado a interactuar con los servicios de AWS. Esta información la guarda localmente en una variable.

Una vez que se ha obtenido la información de la plantilla, el método procede a obtener las **variables necesarias** para la configuración. Estas variables no se pasan como parámetro, sino que han sido configuradas desde el *workflow* inicial como **variables** de **entorno**. Dichas variables contienen los **secretos** del **repositorio** de GitHub, incluyendo las credenciales de acceso y la región deseada por el usuario en AWS. Con las variables de entorno aseguradas, el método procede a completar los campos relevantes de la plantilla con los valores correspondientes de las variables de entorno.

Posteriormente, el archivo de configuración del proveedor es almacenado en el **directorio** correspondiente a la ***Issue*** dentro de la carpeta 'deployments'. Aunque este archivo no será subido al repositorio, su ubicación temporal en el directorio de la *Issue* es fundamental para el funcionamiento adecuado de los demás métodos de Terraform. Como se verá en los siguientes métodos, los métodos de Terraform cambian su **directorio activo** al de la *Issue* dentro de 'deployments'. Por lo tanto, es esencial que el archivo con la configuración del proveedor se encuentre en esa ubicación para permitir a estos métodos de Terraform acceder a esta información.

6.5.2. Método prepare

El método ***prepare*** de 'terraform.rb' es el encargado de preparar los archivos de configuración necesarios para el despliegue utilizando Terraform. Básicamente, este método se encarga de tomar los parámetros proporcionados por el usuario y utilizarlos para generar los archivos de configuración requeridos para cada servicio a desplegar.

En primer lugar, el método realiza una llamada al método ***provider*** de Terraform, el cual se encarga de preparar el proveedor específico para su uso con Terraform. Una vez configurado el proveedor, el método procede a generar los archivos de configuración para cada servicio que será desplegado.

Aunque la implementación de este código puede parecer compleja debido a la presencia de bucles anidados, en realidad sigue un proceso simple pero necesario para manejar y desplegar **múltiples servicios** con sus **múltiples instancias** de manera simultánea. El método recorre cada **servicio disponible** en AWS-Autodeploy, con el fin de encontrar la plantilla **'.rf.erb'** correspondiente. Una vez localizada, lee esta plantilla para obtener los datos necesarios y los almacena temporalmente para su posterior uso en la generación de archivos de configuración. Posteriormente, el método procede a leer el primer parámetro proporcionado para cada servicio, que indica el **número de instancias** que se deben crear para dicho servicio. Este parámetro es común en todos los servicios y se encuentra siempre en la primera posición de su mapa correspondiente. Gracias a esta información, el método sabe cuántas instancias de cada servicio se deben configurar, lo que es crucial para el proceso de despliegue.

Cada instancia de servicio se personaliza utilizando los parámetros específicos proporcionados por el usuario. Estos parámetros se encuentran en forma de ***arrays*** dentro del mapa de cada servicio. Para cada instancia, el método selecciona una posición diferente de estos *arrays*, asegurando así que cada una obtenga su configuración correcta.

Finalmente, una vez que se ha completado la configuración de una instancia, el método guarda el archivo de configuración correspondiente dentro del directorio 'deployments', específicamente en el directorio de la *Issue* pertinente. Este proceso se repite hasta que se hayan configurado todas las instancias para todos los servicios involucrados, momento en el que el método de preparación concluye su ejecución.

6.5.3. Método init y plan

El método ***init_plan*** de 'terraform.rb' es el encargado de ejecutar los métodos estándares de Terraform **'init'** y **'plan'**, respectivamente.

En primer lugar, el comando ***terraform init*** inicializa el entorno de Terraform, descargando los proveedores necesarios y configurando el estado de trabajo. Esto garantiza que Terraform esté correctamente preparado para administrar la infraestructura según las especificaciones proporcionadas en los archivos de configuración.

Por otro lado, el comando ***terraform plan*** realiza una validación exhaustiva de la configuración definida en los archivos de Terraform. Durante esta fase, Terraform analiza la infraestructura existente y compara el estado actual con la definición deseada. Esto permite detectar cualquier discrepancia o posible conflicto que pueda surgir al aplicar los cambios propuestos. Al finalizar, Terraform genera un plan detallado de las acciones que se realizarán durante el despliegue, sin realizar cambios reales en la infraestructura.

Una vez que el directorio de trabajo ha sido cambiado, se ejecutan los comandos ***terraform init*** y ***terraform plan*** utilizando el método **'capture3'** de la biblioteca Open3. Esta biblioteca estándar de Ruby proporciona una interfaz para ejecutar comandos en un subproceso y manejar las entradas y salidas de esos subprocesos. El método 'capture3' de Open3 se utiliza para ejecutar estos comandos y capturar tres valores:

- ***stdout*:** Esta variable contiene la salida estándar del comando ejecutado. En ella se incluye cualquier información o mensajes generados durante la ejecución del comando, como mensajes de progreso o información relevante para el usuario.
- ***stderr*:** Aquí se almacena la salida de error del comando. Esta salida registra mensajes de advertencia o errores que puedan surgir durante la ejecución del comando. Es importante capturar esta salida para identificar y abordar cualquier problema que pueda surgir durante el proceso.
- ***status*:** Indica el estado de salida del comando. Un valor de 0 indica que el comando se ejecutó correctamente, mientras que cualquier otro valor indica un error durante la ejecución. Este valor es esencial para determinar si el comando se ejecutó con éxito o si ocurrieron problemas que requieren atención.

Después de ejecutar los comandos, se lleva a cabo una verificación del estado de su ejecución utilizando la variable **'status'** obtenida con 'capture3'. En el caso de que el estado sea afirmativo (valor 0), se considera que la ejecución fue exitosa y se procede según lo planificado. Sin embargo, si el estado indica un fallo en la ejecución, se registra un mensaje de error utilizando el sistema de registro de logs y se devuelve una lista de errores.

6.5.4. Método apply

El método ***apply*** de 'terraform.rb' es el encargado de llevar a cabo la ejecución del despliegue de la infraestructura definida en los archivos de configuración, utilizando Terraform para interactuar con el proveedor especificado (AWS). Esto se logra ejecutando el método estándar de Terraform ***terraform apply***. Durante la ejecución de este método, Terraform se encarga de comparar el estado actual de la infraestructura con la definición deseada en los archivos de configuración. Luego, identifica y aplica las adiciones, modificaciones y eliminaciones necesarias para asegurar que la infraestructura coincida con la definición deseada. Además, ***terraform apply*** también proporciona retroalimentación detallada sobre los cambios aplicados, como la creación de nuevos recursos, la actualización de configuraciones existentes o la eliminación de recursos obsoletos. Esta información facilita la revisión y validación de las acciones realizadas. En el caso de AWS-Autodeploy, esta información se pasa a la acción DeployAction, que con su método *report* será la encargada de dar formato y elegibilidad a estos datos.

Primero, se realiza nuevamente la inicialización utilizando ***terraform init***. La decisión de ejecutar esta acción antes de cada operación de Terraform ('*plan*', '*apply*', '*destroy*') se basa en garantizar que el entorno de Terraform esté adecuadamente configurado y actualizado. Esto es esencial para mantener la coherencia entre la configuración definida en los archivos de Terraform y el estado real de la infraestructura. Sin esta inicialización adecuada, podrían surgir **discrepancias** entre la configuración y el estado real de la infraestructura, lo que podría resultar en despliegues incorrectos o inesperados.

En el contexto de AWS-Autodeploy, donde el código del repositorio se ejecuta en un **entorno** de **contenedor** de GitHub, esta práctica cobra aún más importancia. La ejecución en un entorno de contenedor puede introducir cierta complejidad adicional, por lo que garantizar que Terraform esté correctamente inicializado antes de cada operación ayuda a prevenir problemas y asegura un despliegue estable y confiable en AWS-Autodeploy.
Luego, se ejecuta el comando ***terraform apply***, que aplica los cambios a la infraestructura según los ficheros de configuración. Al igual que en el método ***init_plan***, se utiliza el método 'capture3' de Open3 para ejecutar el comando y capturar su salida estándar, salida de error y estado de salida.

Es importante destacar que ***terraform apply*** realiza cambios reales en la infraestructura del proveedor de servicios en la nube, a diferencia de ***terraform plan***, que sólo simula los cambios y muestra qué acciones se realizarían sin ejecutarlas realmente. Por lo tanto, ***terraform apply*** es una operación crítica que debe ejecutarse con precaución, ya que puede afectar directamente la infraestructura en producción.

Después de aplicar los cambios con éxito, el método ejecuta ***terraform-bin output*** para obtener los *outputs* generados por Terraform. Estos *outputs* representan información relevante sobre la infraestructura desplegada, como direcciones IPs, IDs de recursos, o cualquier otro dato definido en los archivos de configuración. Estos *outputs* se capturan en formato JSON y se devuelven como parte de la respuesta del método ***apply***, para que puedan ser procesados y utilizados por la acción 'DeployAction' en su método *report*.

6.5.5. Método destroy

El método ***destroy*** de 'terraform.rb' es el encargado de eliminar (terminar) las instancias de los servicios y arquitecturas creados en métodos anteriores con AWS-Autodeploy. Para lograr esto, el método utiliza el comando estándar de Terraform, ***terraform destroy***.

La implementación de este método sigue la misma estructura que los métodos anteriores. Primero, se ejecuta ***terraform init*** para preparar el entorno y asegurarse de que Terraform esté correctamente configurado. Una vez completada la inicialización, se procede a ejecutar el comando ***destroy***. Ambos comandos utilizan el método 'capture3' de la biblioteca Open3, que permite ejecutar comandos en un subproceso y capturar su salida estándar, salida de error y estado de salida. Además, se utiliza el método **'change_dir_temp'** de la herramienta propia de AWS-Autodeploy, FileManager, para cambiar temporalmente al directorio de la *Issue* correspondiente.

En caso de que la ejecución de alguno de los comandos falle, se registrará un mensaje de error utilizando el registro de logs y se devolverá una lista de errores para su posterior manejo. En caso de que no falle ninguno, se devuelve la *array* de errores vacía.

6.5.6. Plantillas de Terraform

Las plantillas de los servicios en AWS-Autodeploy permiten procesar y generar contenido dinámico utilizando la sintaxis de Ruby en un documento de texto plano. Estas plantillas son archivos con extensión **'.rf.erb'**. En estas plantillas, se utilizan marcadores de posición delimitados por '<%= %>' para definir los valores que serán reemplazados dinámicamente durante el proceso de despliegue.

El uso de estas plantillas ofrece diversas ventajas. En primer lugar, permiten parametrizar la configuración, lo que facilita la **reutilización** y la **escalabilidad** de los archivos de configuración. Esto significa que los mismos archivos de plantilla pueden utilizarse para desplegar **múltiples instancias** de un servicio con diferentes configuraciones. Además, al utilizar la sintaxis de Ruby, se pueden incorporar **lógicas** más **complejas** en las plantillas, como iteraciones o condiciones, lo que aumenta la flexibilidad y el poder de expresión.

En estas plantillas, se debe definir una estructura estándar de Terraform. Las plantillas de Terraform siguen una estructura común que define los recursos a desplegar, sus atributos y cualquier salida (*outputs*) necesaria para acceder a información después del despliegue.

```
resource "aws_db_instance" "<%= rds_name %>" {
  engine          = "<%= rds_engine %>"
  engine_version = "<%= rds_engine_version %>"
  instance_class = "<%= rds_instance_class %>"
  allocated_storage = "<%= rds_allocated_storage %>"
  storage_type       = "<%= rds_storage_type %>"
  identifier     = "<%= rds_identifier %>"
  username       = "<%= rds_username %>"
  password       = "<%= rds_password %>"
  skip_final_snapshot  = true

  tags = {
    Name = "<%= rds_name %>"
    Tags = "<%= rds_tags %>"
  }
}
```

Figura 26. Código de una plantilla para crear una instancia de RDS en Terraform.

En Terraform, se utilizan **bloques de recursos** que se definen con la palabra clave 'resource', seguida del tipo de servicio que se desea configurar. En la implementación particular en AWS-Autodeploy, los **nombres** de los **marcadores** son cruciales. Estos nombres sirven para que el código de otros *scripts*, como el script de *apply* en 'terraform.rb', pueda identificar estos marcadores y asignarles el valor correspondiente durante el proceso de despliegue. Por lo tanto, es importante definir nombres descriptivos y significativos para los marcadores, de modo que sea claro qué información se espera que se inserte en cada uno de ellos. En este caso, tienen el mismo nombre que los campos que se deben rellenar en las plantillas de los servicios.

6.5.7. Ficheros de configuración de Terraform

Para que AWS-Autodeploy pueda ejecutar sus métodos de Terraform de manera efectiva, es necesario subir ciertos archivos de configuración al repositorio. Uno de los archivos importantes en este proceso es el fichero **'terraform.tfstate'**. Este fichero contiene el estado actual de la infraestructura gestionada por Terraform y se utiliza para mantener un registro de los recursos desplegados, como las instancias, subredes, grupos de seguridad, entre otros.

Figura 27. Código de 'terraform.tfstate' tras un despliegue de dos instancias EC2.

El ejemplo mostrado contiene el contenido de 'terraform.tfstate' tras un despliegue de dos instancias EC2 en AWS. En este archivo, cada instancia EC2 es tratada como un recurso y se representa como un objeto dentro del *array* **'resources'**. Para cada recurso desplegado, se **registran** una serie de atributos y configuraciones que describen su estado actual. Estos atributos incluyen información importante como el AMI utilizada, la disponibilidad de la zona, el tipo de instancia, la dirección IP pública, la ID de la instancia y otros detalles relacionados con la configuración y el estado de la instancia.

Además del *array* 'resources', el archivo 'terraform.tfstate' también puede contener otro *array* llamado **'outputs'**. Este *array* almacena las salidas (*outputs*) definidas en plantilla de Terraform. El método *apply* de 'terraform.rb', obtiene la información de los *outputs* de este *array*.

6.6. Herramientas

La mayoría de las herramientas desarrolladas para AWS-Autodeploy están diseñadas e implementadas **específicamente** para integrarse con esta plataforma, lo que puede limitar su portabilidad a otros repositorios o proyectos. Por lo general, trasladar estas herramientas a un entorno diferente requerirá de ajustes en su implementación, y no de un proceso como copiar y pegar su código. Asimismo, las herramientas han sido desarrolladas con un cierto grado de **parametrización** y **flexibilidad**, lo que permite su **adaptación** a diferentes contextos dentro del propio proyecto de AWS-Autodeploy. Es decir, muchos de sus métodos y funciones pueden ser utilizados de manera independiente en distintas partes del proyecto.

Además, la lógica subyacente de estas herramientas es **extrapolable** a otros proyectos. Aunque inicialmente pueden parecer cajas negras, es decir, componentes opacos que no son fácilmente comprensibles o utilizables fuera de su contexto original, al examinar y comprender las implementaciones específicas de estas herramientas, se pueden identificar patrones y principios generales que pueden ser aplicados en otros proyectos de manera más amplia y flexible. Es por ello, que en los siguientes apartados se explica la lógica que hay detrás de cada una.

6.6.1. Log

La herramienta **Log** en AWS-Autodeploy se encarga de **registrar** mensajes durante la ejecución de las acciones en AWS-Autodeploy. Su implementación principal se centra en mostrar estos registros en la salida del workflow, para su visualización desde la página del repositorio en GitHub. Estos mensajes permiten tanto identificar, seguir y solucionar errores como comprender el progreso de AWS-Autodeploy .

La implementación del Log cuenta la peculiaridad de tener varios tipos o niveles de registro, cada uno con un propósito específico. Estos tipos de registros, a parte de salir indicados en el mensaje de un Log, ofrecen un estilo visual diferente para cada uno de ellos.

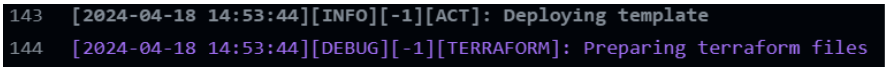

Figura 28. Ejemplo de salida de un mensaje de información y debug del Log.

Los mensajes de **información** se utilizan para marcar el inicio o fin de acciones importantes, como la configuración de una herramienta o la ejecución de un método de Terraform. Por otro lado, los mensajes de depuración (***debug***) ofrecen información más detallada que puede ser útil para comprender el estado o resultados de ciertas funciones, aunque son menos relevantes para la ejecución general del flujo. Los mensajes de advertencia (**warning**) y errores (**error**) se utilizan para señalar fallos o problemas durante la ejecución, y suelen ir acompañados de excepciones o capturas de errores en el código. Por último, los estados **fatal** y **unknown** son estados críticos que indican situaciones graves o desconocidas, aunque en el caso de AWS-Autodeploy se ha optado por no utilizarlos, dejándolos disponibles para futuras expansiones de la herramienta.

Cada *script* en AWS-Autodeploy define y utiliza su propio registro de logs, identificando el componente o script con un nombre específico en la variable **'LOG_COMP'**. En puntos críticos del código, se insertan mensajes de log según la naturaleza y relevancia de la información a registrar.

6.6.2. GitHub Client

La herramienta **GitHub Cliente** en AWS-Autodeploy se encarga de configurar el cliente GitHUb para que pueda **interactuar** y **obtener** información relevante de él. Inicialmente, se planeaba que esta herramienta contuviera todas las configuraciones del cliente, pero debido a su alta complejidad y baja eficiencia, no resultaba rentable implementar todos los métodos. Por lo tanto, se optó por incluir únicamente los **tres** métodos **principales**.

El primer método es ***commit***. En este método, se llevan a cabo las acciones necesarias para subir archivos al repositorio de GitHub. Hay dos formas de implementarlo: una es pasándole los archivos a subir como parámetros y la otra es proporcionándole el directorio donde se encuentran los archivos y que agregue y suba todos los documentos de ese directorio. Se optó por la segunda opción, ya que el método de **cambio de directorio** ya está configurado con la herramienta **FileManager**.

Como se mencionó en el apartado de Terraform, durante la ejecución de las acciones de AWS-Autodeploy, existen ciertos archivos que se generan temporalmente y que no deben ser incluidos en el repositorio de GitHub. Estos archivos incluyen:

- **plan.txt**: Este archivo contiene información sensible, como las credenciales de acceso, de manera encriptada, y no debe ser compartido en el repositorio público. Por lo tanto, se elimina antes de realizar el commit para evitar la exposición de información confidencial.

- **provider.tf**: Similar a plan.txt, provider.tf también contiene credenciales, pero en este caso, no están encriptadas. Por razones de seguridad, es importante evitar subir este archivo al repositorio público, ya que podría exponer las credenciales de acceso. Actualmente, existen bots que están continuamente escaneando este tipo de archivos en los repositorios de GitHub para estafar y hackear a la gente.
- Directorio **.terraform**: Este directorio contiene archivos generados por Terraform durante el proceso de ejecución, como el estado del proyecto y los archivos de configuración. Sin embargo, estos archivos no son necesarios en el repositorio de GitHub y pueden ocupar una cantidad significativa de espacio. Por lo tanto, se elimina para reducir el tamaño del repositorio y evitar la inclusión de información redundante.
- **plantillas (*.rf)** de terraform: Estos ficheros, con las configuraciones utilizadas para desplegar los servicios y arquitecturas, tampoco se suben al repositorio. La razón principal es facilitar la implementación de los métodos de las acciones 'RecoverAction' y 'UpdateAction', que en caso de que el usuario cambie cualquier configuración, se deberían borrar.

El primer método, ***get_issue***, se utiliza para obtener el cuerpo de una *Issue* en GitHub. Para lograrlo, primero se configura el cliente de GitHub y se construye la URL de la *Issue* con el identificador proporcionado como parámetro. Luego, se realiza una solicitud HTTP GET a esta URL, incluyendo la autenticación mediante un **token de acceso**. Una vez que se recibe la respuesta, se verifica si es exitosa y, en caso contrario, se registra un error y se genera una excepción. Finalmente, se formatea la respuesta JSON y se devuelve como resultado.

El último y tercer método es ***delete_dir***, que se encarga de eliminar el directorio asociado a una *Issue* específica en el repositorio de GitHub. Este método se utiliza en la acción **'PurgeAction'** de AWS-Autodeploy para limpiar y eliminar los archivos de configuración de una *Issue* después de que se hayan completado todas las acciones relacionadas.

Este método ha sido implementado de manera específica utilizando ***git rm -r*** en lugar de simplemente ***rm -r*** del sistema operativo. Esta elección se ha hecho debido a que ***git rm -r*** elimina los archivos y directorios del índice de Git, asegurando que los cambios se reflejen correctamente en el historial de versiones del repositorio. Por otro lado, ***rm -r*** eliminaría los archivos y directorios del sistema de archivos sin tener en cuenta el control de versiones de Git, lo que podría resultar en inconsistencias en el registro de cambios.

6.6.3. Parameterizer

La herramienta **Parameterizer** en AWS-Autodeploy se encarga de dar formato específico a los diferentes **parámetros** obtenidos de la *Issue*, así como de gestionar los distintos **servicios** que contempla y permite AWS-Autodeploy. Implementa **tres** métodos principales, los cuales siguen la estructura y el formato descritos en el apartado "6.3.2 Tratar los parámetros".

El método ***get_params*** se encarga de analizar el cuerpo de la *Issue* y extraer los parámetros necesarios para su procesamiento. Utilizando una expresión regular, identifica las líneas que contienen los **campos** y **valores** delimitados por ":". Posteriormente, separa los valores para cada campo, que están separados por comas, y los almacena en un conjunto asociado a su respectiva clave. De esta manera, se obtiene la primera estructura de datos que representa los parámetros necesarios para la configuración de los diferentes servicios sin ordenar.

El método ***get_services*** obtiene, a partir de las claves del conjunto obtenido en el método anterior, qué servicios de la totalidad de AWS-Autodeploy (pasados como parámetros) se utilizarán para el despliegue. Luego, retorna esta información formando un nuevo *array* que contiene únicamente los servicios a utilizar.

El método ***get_order_params*** finaliza el formateo de los parámetros de la *Issue*, transformando el conjunto obtenido en el método inicial. Este método reorganiza los parámetros para que cada servicio tenga su propio mapa en un conjunto de mapas. Utiliza el *array* de servicios obtenido anteriormente para iterar sobre las claves del conjunto inicial y colocar los pares clave-valor en el mapa correspondiente según el servicio al que pertenecen.

6.6.4. FileManager

La herramienta **FileManager** en AWS-Autodeploy se encarga de gestionar y manipular las diferentes acciones y procesos relacionados con los **ficheros** y **directorios** dentro del repositorio de GitHub. Dado que AWS-Autodeploy se ejecuta en un contenedor, la gestión de estos métodos es crucial. A más a más, en esta herramienta se definen algunas **rutas** globales relativas y algunos **nombres** de archivos relacionados con Terraform y AWS Provider.

El método ***change_dir_temp*** es utilizado para cambiar **temporalmente** al directorio especificado como parámetro dentro de un bloque de código. Esto se consigue con el valor ***yield.*** Para usarlo, todo el código que se encuentre dentro de él (entre su *do* y *end*), se ejecutará en el directorio correspondiente. Este método se utiliza en los métodos ***init_plan*** y ***apply*** de 'terraform.rb'.

El método ***save_file*** permite guardar un archivo con el **nombre** y los **datos** especificados dentro del directorio de la ***Issue*** correspondiente. Estos tres valores mencionados, son los que recibe como parámetro. Este proceso es utilizado por el método ***prepare*** de 'terraform.rb' para almacenar las plantillas configuradas con los servicios y arquitecturas asociados,en el directorio 'deployments/*Issue*'.

Por último, los métodos ***create_dir*** y ***delete_dir*** gestionan el ciclo de vida del directorio asociado al identificador de la *Issue*. Estos métodos utilizan la biblioteca FileUtils para crear y eliminar directorios de manera segura y eficiente, utilizando los métodos **'mkdir_p'** y **'rm_rf'**, respectivamente.

6.7. Validaciones

Las validaciones de los parámetros introducidos por el usuario en las plantillas de la *Issue*, son un aspecto fundamental dentro de AWS-Autodeploy. El *script* de validaciones **'valdiate_tempalte.rb'** se encuentra bajo el directorio 'templates'. En él, están definidas todas las validaciones y métodos de validación de todos los parámetros posibles para los servicios y arquitecturas de AWS-Autodeploy. Este paso previo de validación, facilita la vida tanto al **usuario** como a la propia **plataforma**. Al primero, le permite en tiempo de despliegue de una *Issue*, conocer si los parámetros que ha introducido son válidos o no. A más a más, se le ofrece mensajes con posibles soluciones y/o que le indican dónde se ha cometido el error. Respecto a la herramienta, una correcta validación previa a la ejecución de los métodos de Terraform, reduce significativamente los posibles errores durante el despliegue.

6.7.1. Método principal

El método principal **'validate_actions.rb'**, el cual es el que llamará ValidateAction, consta de varios bucles anidados así como de varias expresiones condicionales, para evaluar y aplicar las validaciones pertinentes a cada parámetro.

Además, este método sigue el principio de que cada servicio tenga su propia plantilla de validaciones y no una única global para todos. En consiguiente, para validar cada parámetro,

primero se obtiene el servicio correspondiente y se accede a su plantilla de validaciones específica, ya que será la que contenga los métodos necesarios para validar los parámetros asociados a ese servicio.

El método inicia obteniendo el **servicio** del conjunto de servicios disponible. Este servicio indica que es el responsable del mapa de parámetros que se validará. Usando esta información, carga en **'validation**' la plantilla de validaciones del servicio correspondiente. A continuación, el código itera sobre cada llave (*key*) en el mapa del servicio para obtener los valores (*values*) que ha configurado el usuario para ese campo de la plantilla. Como en AWS-Autodeploy cada llave puede contener como valor un *array*, es necesario iterar nuevamente este conjunto de datos. En la primera validación, se intenta utilizar una expresión regular específica (**'regex'**) sobre ese campo en particular. Si se trata de un campo especial, donde no son aplicables las validaciones mediante expresiones regulares ('regex'), el programa saltará a la segunda validación. Esta segunda validación consiste en una lista de opciones (**'options'**). Esta lista contiene una serie de validaciones especiales que no se pueden implementar en expresiones regulares. En general, estas validaciones son aquellas que hacen uso de la API de AWS y sus clientes para validar los valores con sus bases de datos.

6.7.2. Primera validación ('regex')

Cada **servicio** define su propio conjunto de expresiones regulares específicas para validar los campos asociados, pero la implementación es común en todas ellas. Usan la sintaxis clave-valor, donde la clave es el campo (la *key* del mapa del servicio), y el valor es la expresión regular a aplicar.

Para la **implementación** de estas expresiones regulares, AWS-Autodeploy se basa en los formatos y restricciones que se especifican para cada campo en la documentación oficial de AWS. Por ejemplo, la documentación de AWS especifica que el *CIDR block* en un EKS debe seguir el formato 'x.x.x.x/y', mientras que el nombre de una VPC sólo puede contener letras minúsculas, mayúsculas, guiones y guiones bajos. Al utilizar estas validaciones **propias** de AWS y no unas inventadas, permite que los métodos de Terraform (que también realizan sus propias validaciones), fallen mucho menos, ya que se reducen mucho las discrepancias.

6.7.3. Segunda validación ('options')

Las validaciones específicas ('options') están diseñadas para usar la API de AWS. La API de AWS es una fuente confiable y actualizada de recursos, que proporciona, entre otros métodos y funcionalidades, validaciones específicas con datos precisos.

Para poder interactuar con la API de AWS, se debe definir y configurar un cliente de AWS. Por consiguiente, se requiere de las credenciales de acceso del usuario, así como en la región deseada. Aunque en AWS-Autodeploy sólo se realicen consultas para validar parámetros, esta API ofrece métodos para interactuar con los recursos de la infraestructura del usuario. Por ese motivo se requiere de las credenciales de acceso. Por otro lado, el parámetro de la región es importante porque especifica en qué región de AWS se realizarán las operaciones. Cada región puede tener sus propias características, disponibilidad de servicios y precios, cosa que puede alterar algunas validaciones. Para configurar estos clientes se usa la biblioteca específica de cada servicio **'aws-sdk-service'**. Esta biblioteca no sólo permite crear estos clientes, sino que también proporciona métodos específicos para realizar validaciones en la API de AWS.

En estas implementaciones en Ruby, cada método de validación recorre los valores proporcionados para su respectivo recurso, como los tipos de instancia EC2, los IDs de AMI o los motores de base de datos RDS. Para cada valor, se realiza una llamada a la API de AWS correspondiente utilizando métodos específicos de la biblioteca 'aws-sdk-service'. Por ejemplo, ***describe_instance_type_offerings*** que valida que los tipos de instancias existan en AWS.

6.7.4. Valores por defecto

En esta sección, se establecen valores por **defecto** para aquellos campos que estén definidos en la plantilla de la *Issue* pero no contengan un valor. Es decir, si un campo como ":ec2_name:" está presente en la plantilla pero al usuario se le olvidó darle un valor, se le asignará automáticamente uno de los valores por defecto definidos en este apartado. Sin embargo, si el usuario no incluye ese campo, los valores por defecto no se aplicarán. Es importante tener en cuenta que esta funcionalidad sólo está habilitada para servicios que impliquen una única instancia. En arquitecturas más complejas o servicios con múltiples instancias, estos valores por defecto no serán asignados.

6.8. Interfaz Gráfica

En el componente **Login** se implementa una ventana de registro donde el usuario introduce sus datos. Para poder implementarlo, primero se realiza un formulario con los campos deseados. Este formulario tiene la peculiaridad de que sólo se habilita el envío de sus datos (sólo se permite hacer *login*), si se han rellenado todos los campos. En caso contrario, esa opción queda inhabilitada. Finalmente, lo que hace este componente es guardar estos campos en *localStorage*, como se ha mencionado en el apartado de diseño.

En el componente **Templates** se implementa una cuadrícula de tres por tres con las plantillas que obtiene del repositorio donde se aloja AWS-Autodeploy. Para implementarlo, se deben obtener estos datos. Dichos datos se obtienen gracias a una llamada a la API de GitHub, que se recogen en el componente usando un *useEffect*. Una vez se tienen los datos de las plantillas, se pueden obtener su imágenes (descargadas con anterioridad en la aplicación gráfica) y los nombres a representar. Finalmente, se debe representar formando esta cuadrícula y con el objetivo de que sea responsivo al cambio de dimensión de ventana.

En el componente **Issue** se implementa el formulario donde se rellenan los campos de la plantilla para crear la *Issue*. Para implementarlo, en primer lugar hay que obtener los datos de esta plantilla, de igual manera que se hace en el componente anterior. Una vez se tienen los campos a completar, se estructura el formulario. Cuando el usuario completa los campos, se recogen tanto los campos como sus valores y se crea una nueva *Issue*, con nombre de la fecha actual y con las *labels* "state-pending" y "action-validate".

En el componente **Actions** se implementa la lógica para transmitir al usuario el estado actual en el que se encuentra AWS-Autodeploy. Además, se ofrece un nuevo conjunto de acciones que el usuario puede ejecutar. Para implementarlo, se debe recoger el último comentario de la *Issue* de GitHub. Esta petición a la API se hará regularmente, el tiempo entre cada llamada dependerá del que se configure en este componente. Se recoge el último comentario, ya que es el que indica si la ejecución ha sido satisfactorio o no. Finalmente, se implementan cuatro botones con nuevas acciones a ejecutar sobre la *Issue*.

La **navegación** entre ventanas se hace gracias a ***React Router***. Para implementarlo, hay que definir en el fichero "router.jsx" todas las rutas de la aplicación. De este modo, en cualquier componente, simplemente usando *useNavigate* con la función *navigate("path")*, se podrá navegar a la nueva ventana.

Para implementar las **llamadas** a la **API de GitHub**, se ha utilizado la misma biblioteca que se usa en el cliente de GitHub del código fuente de AWS-Autodeploy. Esta biblioteca es "octokit" y se implementa del mismo modo que en los apartados superiores. Se prepara la petición con los datos necesarios (nombre del repositorio, nombre de usuario, valores específicos de la petición) y se agrega también el *token* de usuario para que la biblioteca tenga permisos.

7. Conclusiones

El Trabajo de Fin de Grado (TFG) titulado "**AWS-Autodeploy**" se centra en el desarrollo y la implementación de una herramienta para automatizar el despliegue de infraestructuras en Amazon Web Services (AWS). Utilizando GitHub Actions y Terraform, esta herramienta tiene como objetivo simplificar y agilizar la migración a la nube, haciéndola accesible para usuarios de todos los niveles de experiencia. Su funcionalidad principal se integra directamente en el repositorio de GitHub donde se aloja. Además, AWS-Autodeploy incorpora una interfaz gráfica intuitiva y minimalista, permitiendo a los usuarios implementar y administrar el ciclo de vida de la herramienta, de manera más gráfica y sin entrar en detalle en su implementación.

Los **objetivos** planteados al inicio del proyecto se han cumplido satisfactoriamente.

- Entre estos objetivos, se encontraba la necesidad de crear una herramienta que no sólo facilitara el despliegue de infraestructuras en AWS, sino que también ofreciera una experiencia de usuario mejorada mediante una interfaz gráfica amigable.
- Asimismo, era otro de los objetivos desarrollar una arquitectura modular que permitiera la integración y expansión de nuevos servicios de AWS, crear plantillas para estos servicios e implementar sus respectivas validaciones. La herramienta no solo ha sido diseñada e implementada con éxito, sino que también ha demostrado ser robusta y personalizable.
- Además, se ha conseguido implementar la herramienta GitHub Actions, lo que ha permitido gestionar el ciclo de vida de una herramienta compleja de manera simplificada, utilizando *labels* e *Issues* para su manejo y gestión en sus diferentes estados.

En el **ámbito personal**, este proyecto ha sido una experiencia enriquecedora. Inicié el TFG con el objetivo de ampliar y fortalecer mis conocimientos sobre el *Cloud Computing*. Con esta meta en mente, diseñé el proyecto AWS-Autodeploy, con la intención de abordar y mejorar las tres áreas principales de un *Cloud Engineer*: **desarrollo**, **diseño** y **DevOps**.

- Esta última era la rama en la que menos experiencia tenía debido a la falta de proyectos o lugares donde practicar. Principalmente, mi experiencia se basaba en la realización de tests automáticos a la hora de integrar. Sin embargo, gracias a este proyecto, pude profundizar en esta práctica y explotarla al máximo para comprender todo su potencial. Usar los *runners*, gestionar el ciclo de vida de una herramienta compleja simplemente añadiendo *labels* a las *Issues*, entender cómo se usa la API de GitHub, fueron algunos de los aspectos fundamentales que aprendí. En definitiva, implementar la lógica de AWS-Autodeploy con GitHub Actions no ha sido un proceso sencillo, pero al lograrlo, adquirí un conocimiento y experiencia que no habría podido obtener con ningún otro proyecto.
- En cuanto a las otras áreas, pude aplicar los conocimientos que ya tenía para ofrecer a AWS-Autodeploy una funcionalidad atractiva, tanto para un usuario experimentado como para un usuario principiante. Comprender a fondo algunos servicios de AWS, identificar qué parámetros son obligatorios y cuáles opcionales, las validaciones requeridas para cada uno y cómo combinarlos entre sí, ha sido en general, una experiencia muy positiva.

De cara al **futuro**, hay varias áreas en las que AWS-Autodeploy puede mejorar y expandirse:

- En primer lugar, AWS-Autodeploy podría ampliarse agregando **compatibilidad** con nuevos proveedores de servicios como Microsoft Azure o Google Cloud Platform (GCP). Cada uno de estos proveedores tiene sus propias APIs, definiciones de servicios y validaciones distintas. En este TFG no se optó por incluirlos debido a la falta de experiencia con estos proveedores. Entenderlos y adaptarlos a la herramienta habría supuesto un aumento significativo en la complejidad y en el número de horas, ya que para poder integrarlos se requiere un dominio profundo sobre ellos.
- En segundo lugar, AWS-Autodeploy podría mejorar adoptando **nuevas tecnologías**, principalmente de código abierto (*open source*) para mantener la filosofía del proyecto. Una de ellas podría ser Ansible. Ansible permite configurar e instalar *software* en una infraestructura ya desplegada. Esta funcionalidad encajaría perfectamente en el ámbito empresarial, ya que cada empresa podría configurar sus inventarios de Ansible y, una vez desplegada la infraestructura con AWS-Autodeploy, añadir un nuevo paso (estado) a la herramienta para instalar el *software* necesario, convirtiendo así AWS-Autodeploy en una solución de aprovisionamiento muy útil tanto para trabajadores como para clientes.
- En tercer y último lugar, el futuro de AWS-Autodeploy se orienta tanto al mundo **laboral** como al **educativo**. En el primero, agregando nuevas compatibilidades y funcionalidades descritas anteriormente, y aprovechando el potencial de AWS-Autodeploy, podría convertirse en una herramienta fácilmente utilizada en empresas para gestionar el despliegue de infraestructura. En el ámbito educativo, AWS-Autodeploy podría implementarse en institutos o universidades. En el primero, se podría usar la herramienta sin entrar en detalles sobre su funcionamiento, simplemente como una solución cerrada que permite desplegar servicios en AWS, facilitando la comprensión de su funcionamiento y características. En el segundo, podría aportar un gran valor si se decide profundizar en la implementación, agregando nuevas funcionalidades, servicios y plantillas, haciendo la plataforma moldeable para cada uso.

Bibliografía

[1] Robert C. Martin (2017) *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. First Edition. Prentice Hall.

[2] Thomas Erl, Z.M.R.P. (2013) *Cloud Computing: Concepts, Technology & Architecture*. First edition. Prentice Hall.

[3] Rajkumar Buyya, C.V.S.T.S. (2013) *Mastering Cloud Computing: Foundations and Applications Programming*. First Edition. Morgan Kaufmann.

[4] *What Is Cloud Computing? | Strategies and Importance | Gartner* (5 May, 2022). Available at: https://www.gartner.com/en/topics/cloud (Accessed: May 16, 2024).

[5] *Types of cloud computing* (25 July, 2022). Available at: https://www.redhat.com/en/topics/cloud-computing/public-cloud-vs-private-cloud-and-hybrid-cloud (Accessed: May 16, 2024).

[6] *What is cloud architecture? Benefits & Components | Google Cloud* (no date). Available at: https://cloud.google.com/learn/what-is-cloud-architecture (Accessed: May 18, 2024).

[7] *Types of Cloud Computing - SaaS vs PaaS vs IaaS - AWS* (no date). Available at: https://aws.amazon.com/types-of-cloud-computing (Accessed: May 25, 2024).

[8] *Public Cloud vs Private Cloud vs Hybrid Cloud | Microsoft Azure* (May 24, 2024). Available at: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-are-private-public-hybrid-clouds (Accessed: May 21, 2024).

[9] *What is Edge Cloud? | VMware* (August 29, 2023). Available at: https://www.vmware.com/topics/glossary/content/edge-cloud.html (Accessed: May 21, 2024).

[10] *¿Qué es Git? - Azure DevOps | Microsoft Learn* (October 5, 2023). Available at: https://learn.microsoft.com/es-es/devops/develop/git/what-is-git (Accessed: May 10, 2024).

[11] *What is version control? | GitLab* (February 4, 2022). Available at: https://about.gitlab.com/topics/version-control/ (Accessed: May 10, 2024).

[12] *Tips for scripting tasks with Bitbucket Pipelines* (no date). Available at: https://www.atlassian.com/devops/continuous-delivery-tutorials/scripting-tasks-bitbucket-pipelines (Accessed: May 10, 2024).

[13] *Repositories documentation - GitHub Docs* (Sep 10, 2021). Available at: https://docs.github.com/en/repositories (Accessed: March 3, 2024).

[14] *GitHub Issues documentation - GitHub Docs* (May 25, 2021) Available at: https://docs.github.com/en/issues (Accessed: March 3, 2024).

[15] *GitHub Actions documentation - GitHub Docs* (Sep 27, 2020). Available at: https://docs.github.com/en/actions (Accessed: March 4, 2024).

[16] *GitHub REST API documentation - GitHub Docs* (November 28, 2022). Available at: https://docs.github.com/en/rest?apiVersion=2022-11-28 (Accessed: March 20, 2024).

[17] *Using workflows - GitHub Docs* (May 31, 2019). Available at: https://docs.github.com/en/actions/using-workflows (Accessed: March 4, 2024).

[18] *What is Infrastructure as Code with Terraform? | Terraform | HashiCorp Developer* (no date). Available at: https://developer.hashicorp.com/terraform/tutorials/aws-get-started/infrastructure-as-code (Accessed: March 10, 2024).

[19] *The Benefits and Applications of Ruby On Rails Programming Language* (January 19, 2022). Available at: https://programmers.io/blog/the-benefits-and-applications-of-using-ruby-as-a-programming-language (Accessed: May 9, 2024).
[20] *UML Use Case Diagram Tutorial | Lucidchart* (no date). Available at: https://www.lucidchart.com/pages/uml-use-case-diagram (Accessed: April 20, 2024).
[21] *About GitHub-hosted runners - GitHub Docs* (Sep 20, 2023). Available at: https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners/about-github-hosted-runners (Accessed: March 4, 2024).
[22] *Github Actions Security Best Practices - Salesforce Engineering Blog* (October 19, 2021). Available at: https://engineering.salesforce.com/github-actions-security-best-practices-b8f9df5c75f5 (Accessed: March 4, 2024).
[23] *React calls Components and Hooks – React* (no date). Available at: https://react.dev/reference/rules/react-calls-components-and-hooks (Accessed: May 5, 2024).
[24] *Main Concepts v6.23.1 | React Router* (no date). Available at: https://reactrouter.com/en/main/start/concepts (Accessed: May 5, 2024).
[25] *Managing access keys for IAM users - AWS Identity and Access Management* (no date). Available at: https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_access-keys.html (Accessed: April 14, 2024).
[26] *AWS account requirements - AWS Setup* (no date). Available at: https://docs.aws.amazon.com/SetUp/latest/UserGuide/setup-acctrequirements.html (Accessed: April 14, 2024).
[27] *Amazon Elastic Compute Cloud Documentation* (no date). Available at: https://docs.aws.amazon.com/ec2/ (Accessed: March 30, 2024).
[28] *Amazon Simple Storage Service Documentation* (no date). Available at: https://docs.aws.amazon.com/s3/ (Accessed: April 6, 2024).
[29] *Amazon RDS and Aurora Documentation* (no date). Available at: https://docs.aws.amazon.com/rds/ (Accessed: April 7, 2024).
[30] *Amazon Elastic Kubernetes Service Documentation* (no date). Available at: https://docs.aws.amazon.com/eks/ (Accessed: April 9, 2024).
[31] *Amazon Virtual Private Cloud Documentation* (no date). Available at: https://docs.aws.amazon.com/vpc/ (Accessed: April 9, 2024).
[32] *What Is Amazon S3 Glacier? - Amazon S3 Glacier* (no date). Available at: https://docs.aws.amazon.com/amazonglacier/latest/dev/introduction.html (Accessed: April 9, 2024).
[33] *Docs overview | hashicorp/aws | Terraform | Terraform Registry* (June 9, 2017). Available at: https://registry.terraform.io/providers/hashicorp/aws/latest/docs (Accessed: March 28, 2024).

Código proyecto: https://github.com/Kattorzeh/aws-autodeploy.git
Código interfaz gráfica: https://github.com/Kattorzeh/ui-aws-autodeploy.git

Anexos

A. Manual de Usuario: Uso de AWS-Autodeploy

AWS-Autodeploy define un ciclo de vida que, en ocasiones, puede ser complejo de entender debido a su nivel de abstracción. Este apartado tiene el fin de aclarar su funcionamiento, proporcionando una guía detallada sobre todos los pasos necesarios para su configuración y ofreciendo una ayuda sobre cómo utilizar la herramienta de manera efectiva en cada momento.

a. Requisitos

Para utilizar AWS-Autodeploy, es necesario tener una cuenta tanto en GitHub como en AWS y con una credenciales de acceso a ellas.

i. AWS

En primer lugar, para obtener las credenciales de AWS, hay que registrarse en AWS, o iniciar sesión con una cuenta existente, y acceder en la consola gráfica de AWS. En el caso de registrar una nueva cuenta en AWS, es importante mencionar que AWS requiere de una tarjeta de crédito válida, de información personal y de contacto, como nombre, dirección o número de teléfono y de aceptar los términos y condiciones de AWS, que incluyen el acuerdo de servicio y la política de privacidad. Asimismo, en el proceso inicial, se aplicará un cargo de validación a la tarjeta de crédito del usuario por un importe de 0$. Este cargo es temporal y se utiliza para verificar la autenticidad y validez de la tarjeta de crédito proporcionada durante el registro de la cuenta en AWS [26].

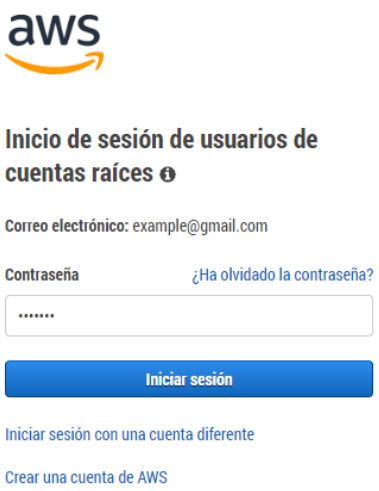

Figura 29. Ejemplo de inicio de sesión en la consola gráfica de AWS.

Una vez registrado, para obtener su clave de acceso (*Access Key*) y su clave de acceso secreta (*Secret Access Key*), es necesario acceder al servicio IAM (*Identity and Access Management*). Una vez dentro, debe ir a la pestaña de Users (Access Management -> Users) y crear un nuevo usuario en AWS, seleccionando el botón “Create Users” y siguiendo su proceso [25].

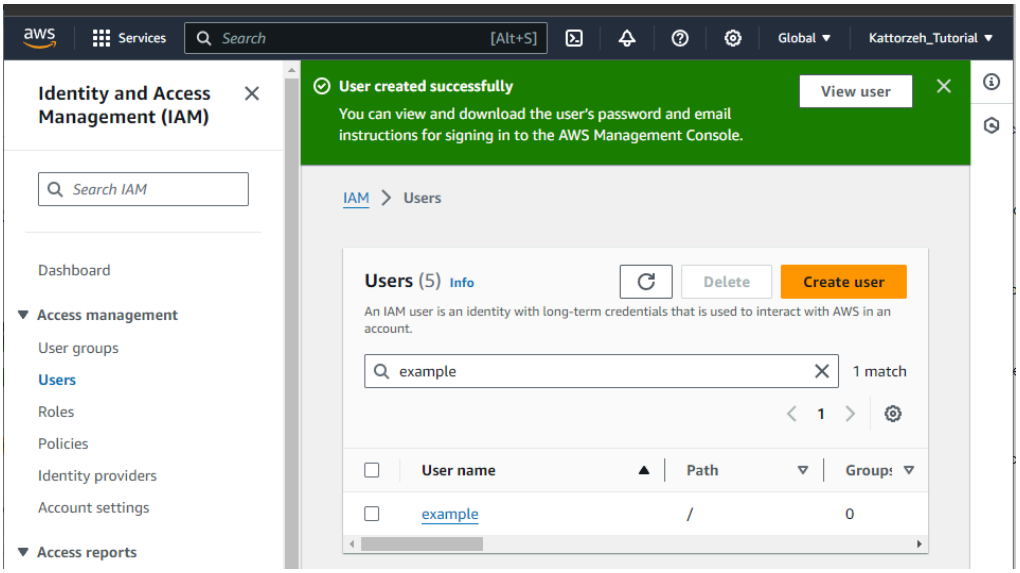

Figura 30. Pestaña de Users en el servicio AWS con un usuario "example" creado.

Una vez creado, se debe asignar una serie de permisos para que este usuario pueda crear, modificar y eliminar servicios de AWS. Para ello, se debe acceder a la información detallada de dicho usuario, pulsando sobre su nombre. En este nuevo menú, aparecen más opciones para configurar el usuario. Seleccione la opción "Add permissions", en la pestaña de "Permissions".

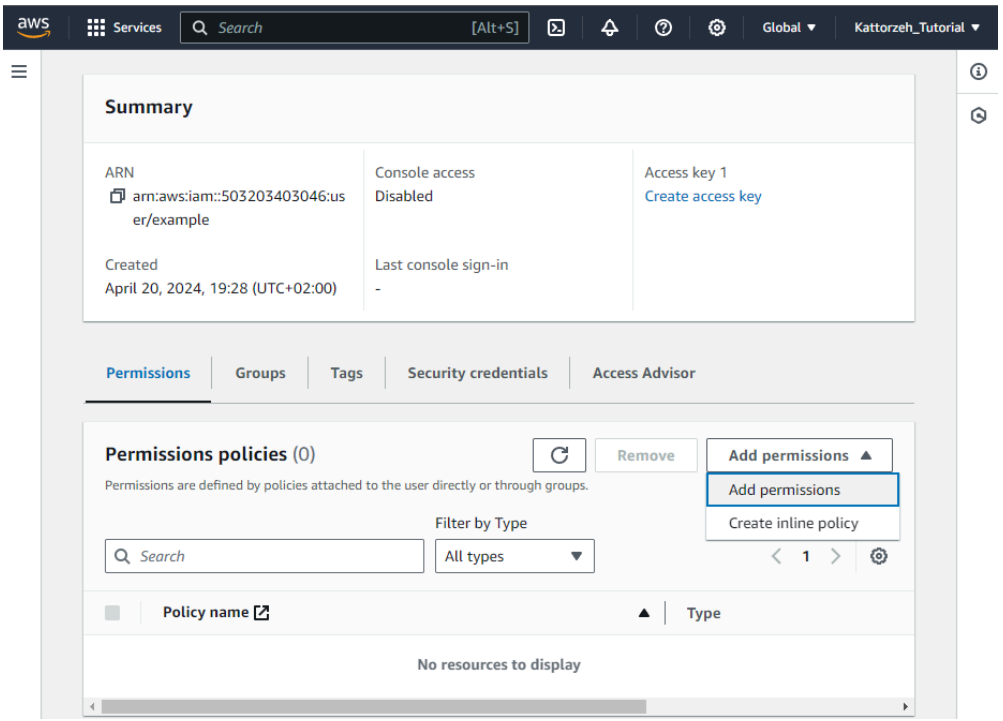

Figura 31. Pestaña de información del usuario "example", con la opción "Add permissions" resaltada.

A continuación, se debe asignar los permisos necesarios. Seleccione la opción de "Attach policies directly". Ahí, se le debe otorgar los permisos de "AdministratorAccess", que ofrecen autorización completa para realizar cualquier operación en AWS, incluyendo la creación, modificación y destrucción de cualquier servicio.

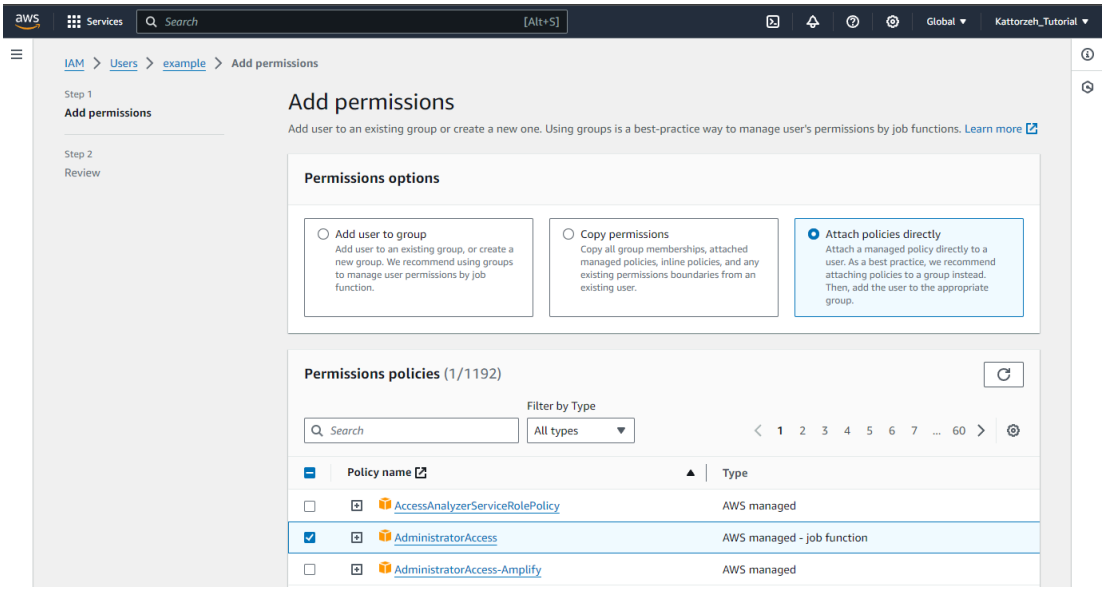

Figura 32. Pestaña Momento en el que se le asigna los permisos de "AdministratorAcces" al usuario "example" .

A partir de este momento, en el menú con la información detallada, debería aparecer esa política. Seguidamente, hay que obtener las credenciales de acceso. Para ello, se debe entrar en la pestaña de "Security Credentials", buscar la opción de " Access Key" y crear una nueva *access key*. Del menú con diferentes opciones, seleccione "Command Line Interface (CLI)".

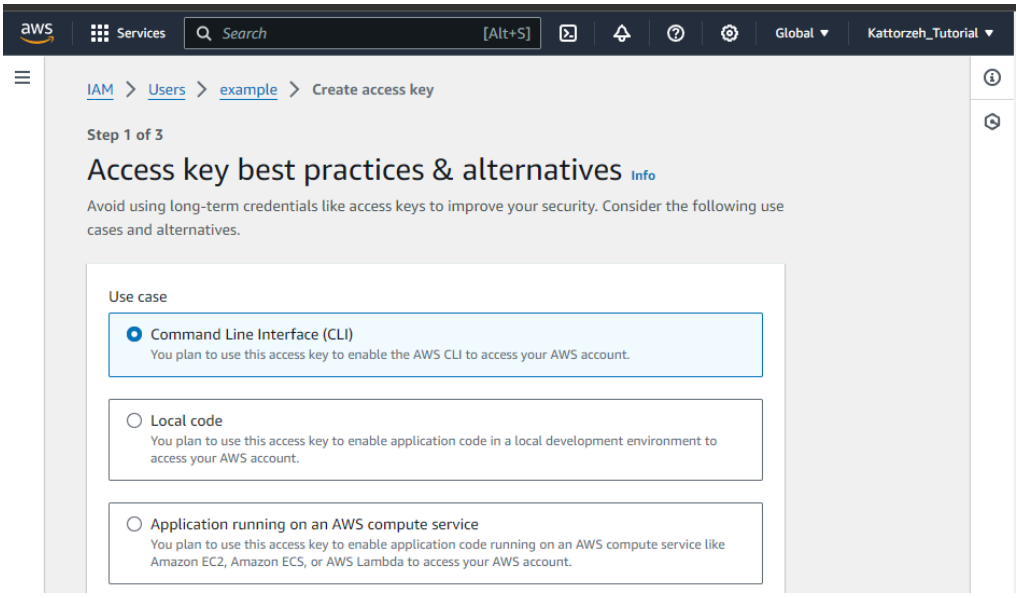

Figura 33. Diferentes opciones para crear una *access key* en AWS.

En este momento, se mostrarán por única vez la *Access Key* y la *Secret Access Key* que se han creado para el usuario. Es fundamental guardarlas en un lugar seguro. Además, AWS ofrece la opción de exportarlas en formato CSV. Es importante destacar que estas credenciales no deben compartirse con nadie, ya que proporcionan acceso completo a los servicios y recursos de AWS y su divulgación puede comprometer la seguridad de la cuenta.

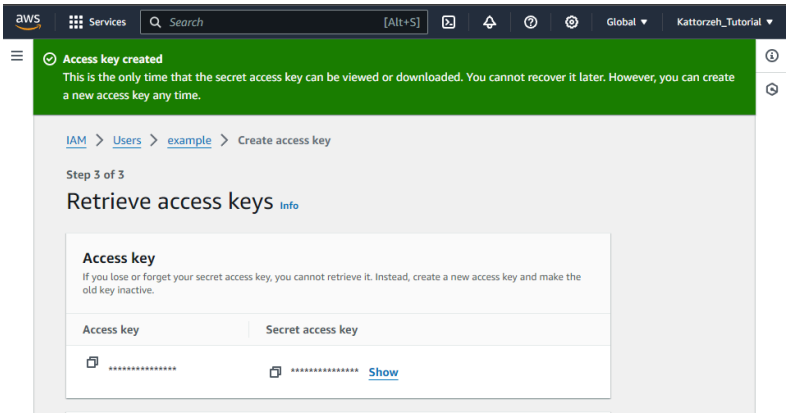

Figura 34. *Access Key* y *Secret Access Key* del usuario "example". Por motivos de seguridad, dichas credenciales han sido ocultadas.

ii. GitHub

En segundo lugar, para obtener el *token* con las credenciales de GitHub, hay que registrarse en GitHub, o iniciar sesión con una cuenta existente. En el caso de registrar una nueva cuenta en GitHub, es importante mencionar que GitHub requiere de un correo electrónico válido, un nombre de usuario único y aceptar sus términos de servicio.

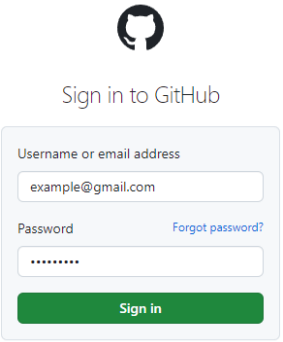

Figura 35. Ejemplo de inicio de sesión en la plataforma de GitHub.

Una vez dentro de la plataforma, para obtener el token con todos los permisos necesarios, hay que navegar a la sección "Settings", ciclando sobre el avatar de su perfil (a través del menú desplegable de su perfil).

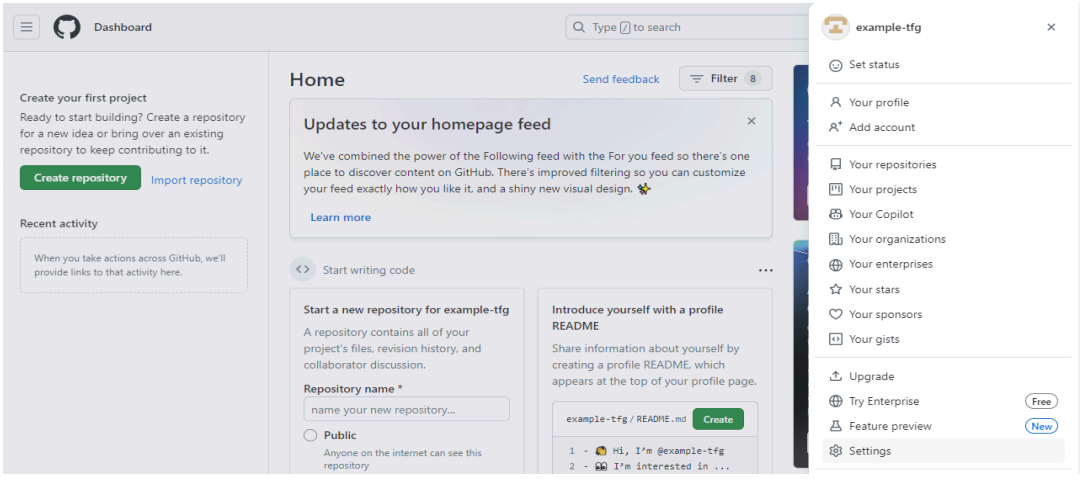

Figura 36. Menú desplegable obtenido al clicar sobre el avatar del usuario de GitHub.

La opción para crear un *token* está en "Developer Settings", que se encuentra en la parte inferior de "Settings". La tercera opción "Personal access token", en particular el método "Tokens (classic)", permite crear estos *tokens*.

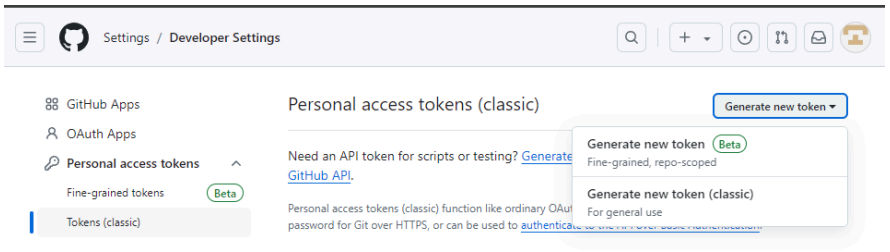

Figura 37. Opción para crear un *token*, dentro de "Developers Settings" en GitHub.

En este apartado, se debe configurar qué servicios se le asignan al *token*. Para facilitar el proceso y no dejarse alguno, se recomienda asignarle todos los posibles, así como una fecha de caducidad para proteger el *token* en caso de compartirlo públicamente.

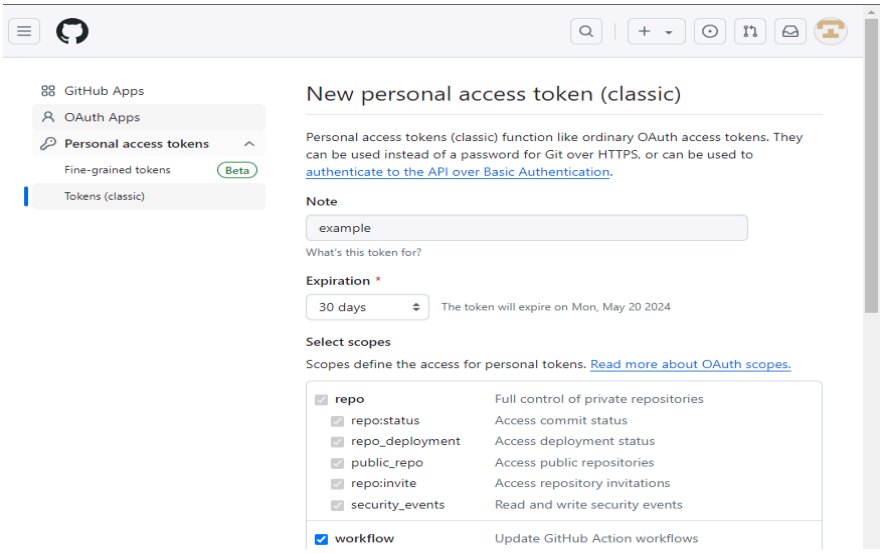

Figura 38. Configuración de los permisos para *token* de GitHub.

A continuación, se le mostrará por única vez el *token* de su usuario GitHub. Es fundamental guardarlas en un lugar seguro y con acceso a él, ya que se requerirá para la configuración de AWS-Autodeploy.

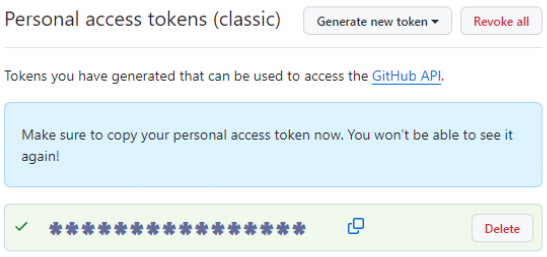

Figura 39. Configuración de los permisos para *token* de GitHub. Por motivos de seguridad, dicho *token* ha sido ocultado.

b. Secretos de repositorio

Para crear un secreto de repositorio en GitHub, hay que acceder en primer lugar al repositorio. Una vez dentro, una de las pestañas disponibles es la de "Settings". Ahí dentro, se encuentra la opción "Secrets and variables". De todas ellas, seleccione "Actions". Es aquí donde se deberán crear los diferentes secretos, usando el botón verde "New Repository Secret".

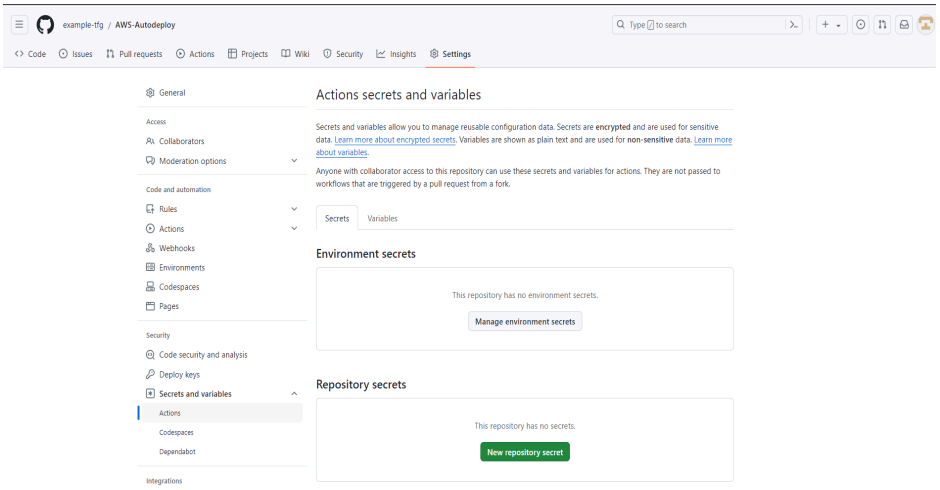

Figura 40. Menú donde se configuran los secretos de un repositorio de GitHub.

Los primeros secretos a configurar son los de AWS. Es importante utilizar la misma nomenclatura que la que se especifica a continuación.

Secretos de *Amazon Web Services*	
Nombre del secreto	**Valor del Secreto**
ACCESS_KEY_ID	Contiene el *Access Key* de AWS del usuario.
SECRET_ACCESS_KEY	Contiene el *Secret Access Key* de AWS del usuario.
REGION	Contiene la región de AWS que desea el usuario.

Tabla 19. Tabla de los secretos de repositorio para AWS.

Los siguientes secretos a configurar son los de GitHub. Es importante utilizar la misma nomenclatura que la que se especifica a continuación.

Secretos de GitHub	
Nombre del secreto	**Valor del Secreto**
GH_EMAIL	Contiene el correo electrónico del usuario de GitHub.
GH_TOKEN	Contiene el *token* con las credenciales del usuario de GitHub.
GH_USER	Contiene el nombre de usuario del usuario de GitHub.

Tabla 20. Tabla de los secretos de repositorio para GitHub.

c. Primeros pasos

Una vez se tengan las credenciales de acceso tanto para AWS como para GitHub, como los secretos de repositorio configurados, el usuario puede iniciar la ejecución de AWS-Autodeploy.

Como se menciona en el apartado "5.5 Plantillas", se recomienda comenzar desplegando servicios individuales en lugar de arquitecturas complejas. Para iniciar el despliegue de un servicio en AWS-Autodeploy, es necesario abrir una nueva *Issue* en el repositorio. Para ello, hay que escoger la opción "Issues" en la parte superior y seleccionar el botón "New Issue".

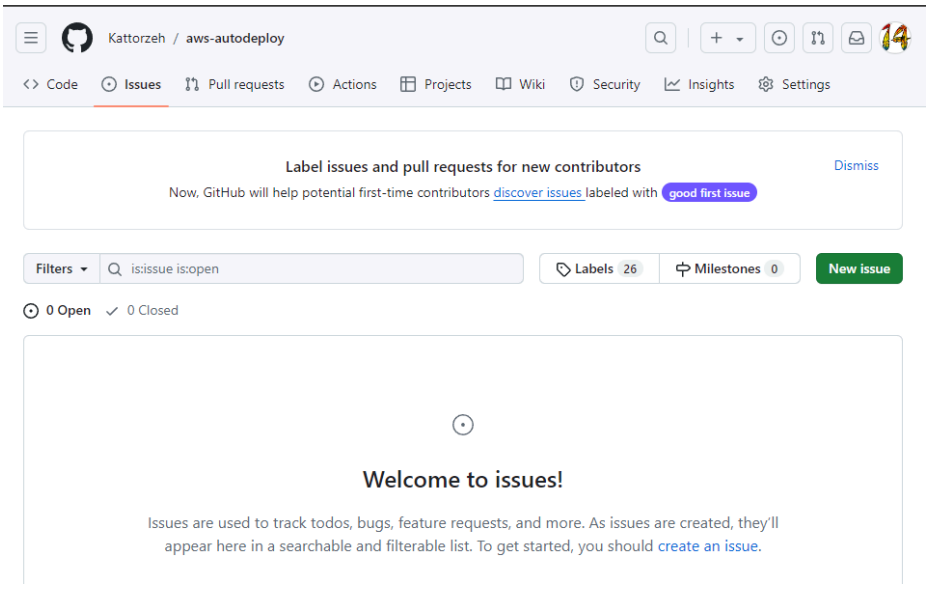

Figura 41. Menú donde se abren las *Issues* en el repositorio de GitHub.

Cuando se pulse sobre el botón "New Issue", aparecerán un listado con las plantillas de servicios y arquitecturas que haya configuradas en AWS-Autodeploy. Seleccione la deseada. Para hacer un ejemplo sencillo, se usará la plantilla de EC2 de AWS.

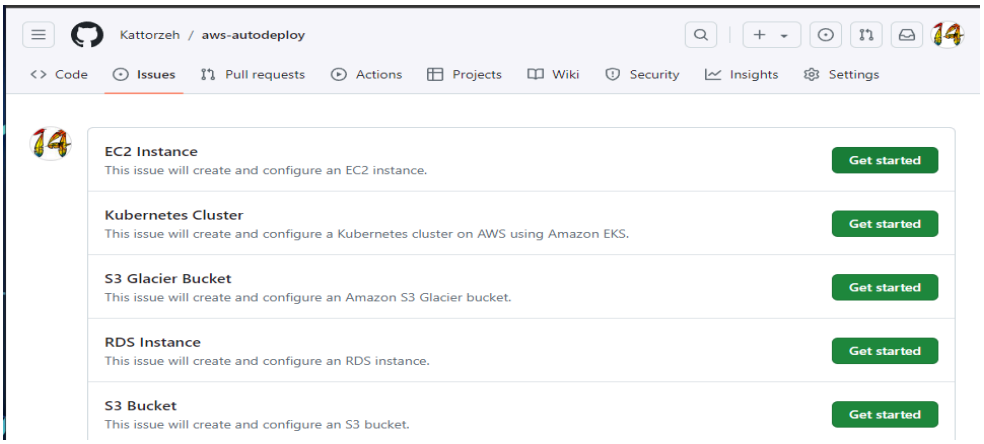

Figura 42. Listado con algunas de las plantillas configuradas en AWS-Autodeploy.

A continuación, se abrirá el formulario que el usuario debe rellenar para poder instanciar el servicio EC2 con la configuración deseada. En este caso, se muestra una configuración para desplegar dos instancias EC2, de tamaño *t2.micro* y con una AMI con sistema operativo Linux.

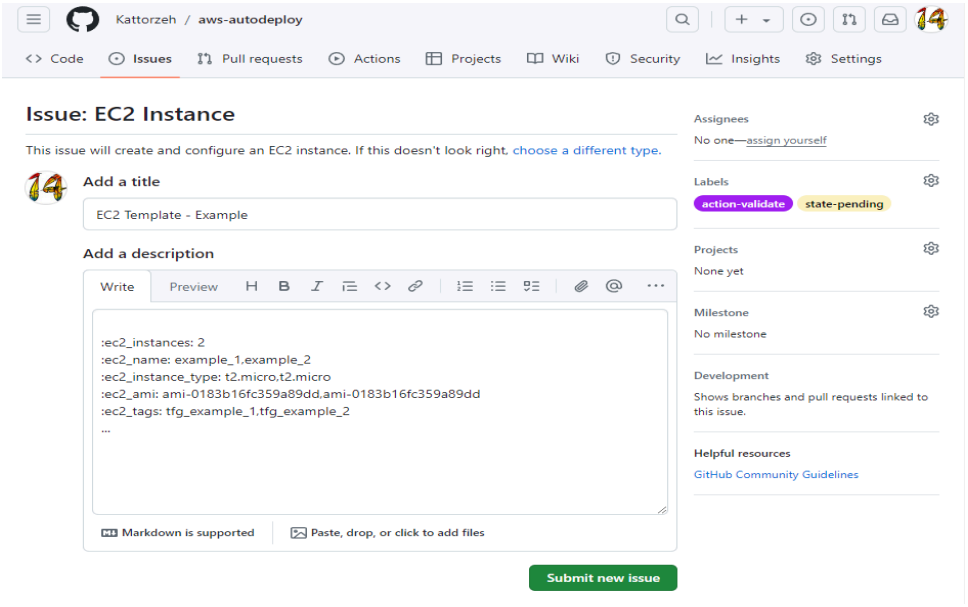

Figura 43. Configuración de instancias EC2 en una plantilla de AWS-Autodeploy.

Como se puede observar en la parte derecha, a esta *Issue* se le ha asignado por defecto dos *labels*. La primera "action-validate" iniciará la ejecución del *workflow* que permitirá a la GitHub Action validar los parámetros del formulario. La segunda *label* "state-pending" permite saber a la herramienta en qué estado de su ciclo de vida se encuentra la *Issue*. En este caso, se encuentra al inicio del todo, en el estado "PENDING". Al pulsar sobre "Submit new issue", se abrirá (creará) la *Issue* en el repositorio y dará inicio a la ejecución de su *workflow*. Esta ejecución, puede verse desde la pestaña "Actions".

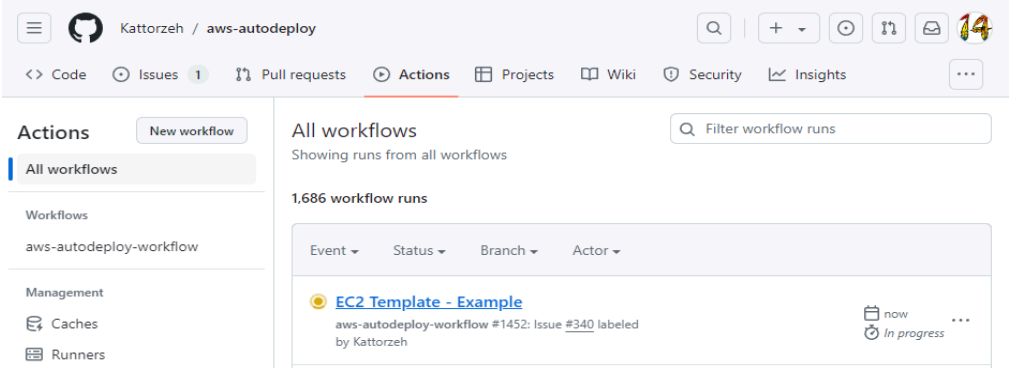

Figura 44. Pestaña de "Actions" donde se puede observar la ejecución del *workflow* iniciado por la plantilla EC2 de AWS-Autodeploy.

Si se pulsa sobre el *workflow* se pueden ver los detalles específicos de la GitHub Action, separados por sus diferentes *steps*. Para cada uno de ellos, se han programado una serie de registros que permiten seguir la ejecución de la acción correspondiente.

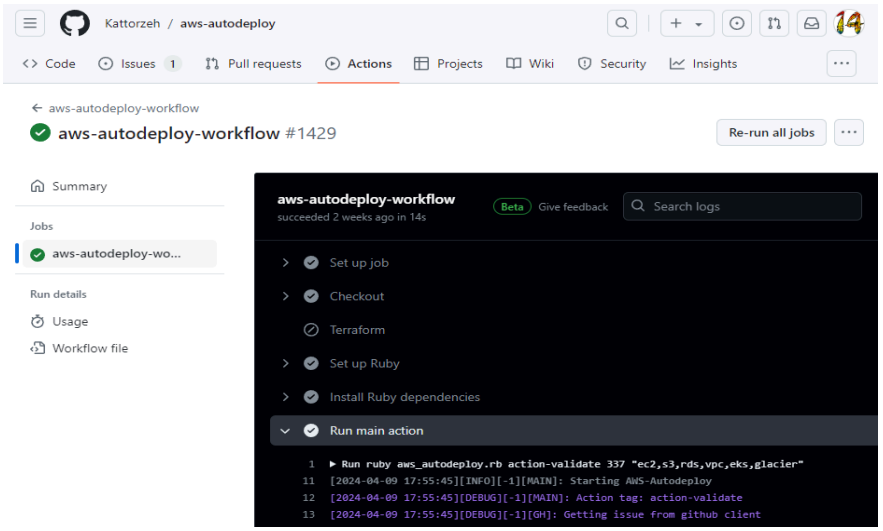

Figura 45. Menú con los *steps* de la GitHub Action creada para validar los parámetros de la plantilla.

Si la acción se ha ejecutado con éxito o no, a parte de en los registros de la GitHub Action, quedará reflejado en la propia *Issue*. En caso de éxito, se retirarán las dos *labels* anteriores y se le agregará la nueva *label* de "state-validated", así como un mensaje en la propia *Issue* indicando que la acción se ha ejecutado correctamente.

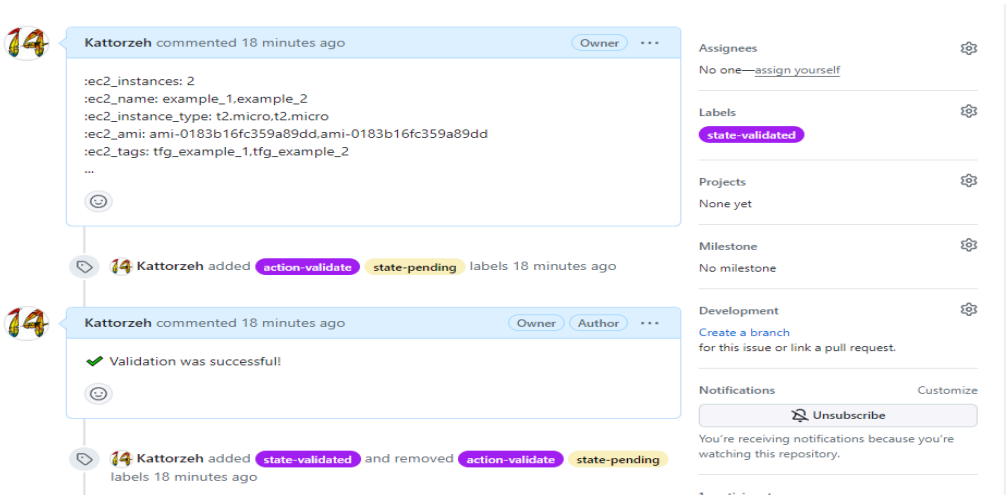

Figura 46. *Issue* de AWS-Autodeploy en el estado "VALIDATED".

En caso de que se haya producido un error, principalmente por la introducción de datos erróneos en la plantilla, se le asignará el *label* "state-failed-validate". Para ayudar al usuario a poder subsanar estos problemas, se ofrece un mensaje en la *Issue* indicando qué parámetros han fallado así como qué valores son válidos para ellos.

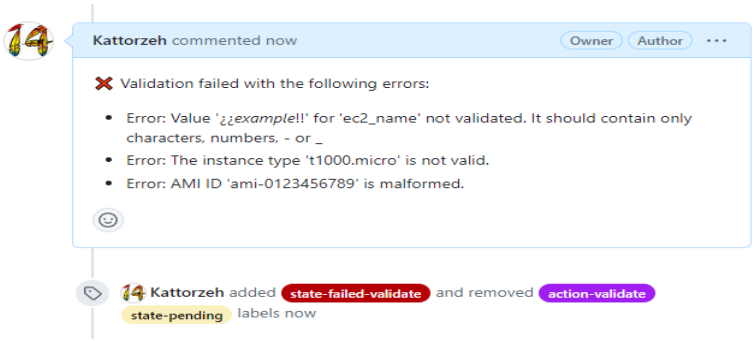

Figura 47. *Issue* de AWS-Autodeploy en el estado "FAILED_VALIDATE".

El siguiente paso a realizar, es indicar a la herramienta qué acción se desea ejecutar para avanzar en el ciclo de vida de AWS-Autodeploy. Como se muestra en el diagrama de su ciclo de vida, en el caso de estar en el estado "VALIDATED" se le deberá asignar la acción "action-deploy" para avanzar a los estados "DEPLOYING" y "RUNNING". En el caso de estar en el estado de fallida "FAILED_VALIDATE", se le deberá asignar la acción "action-recover" para lograr estar en el estado "VALIDATED".

Para asignar nuevas *labels*, se debe pulsar sobre el icono con forma de engranaje que aparece en la derecha de las *labels*. Para el correcto funcionamiento, es indispensable no **borrar** o **eliminar** ninguna *label* que ya contenga la *Issue*. Sólo se deben **agregar** nuevas *labels*.

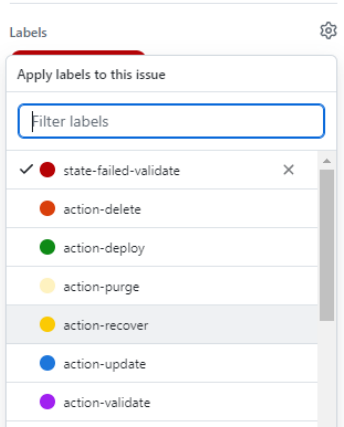

Figura 48. Menú donde se agregan nuevas *labels* a la *Issue*.

Con estos pasos, el usuario debe ser capaz de moverse por el ciclo de vida de AWS-Autodeploy, así como poder lanzar varios servicios con la herramienta. El usuario sólo debe tener en cuenta qué acción debe asignar a la *Issue* en cada momento, para no romper el ciclo de vida La idea es sencilla y fácil de aplicar.

Una vez se le ha asignado una nueva *label* a la *Issue*, un *workflow* con su GitHub Action se ejecutará para llevar a cabo dicha acción. Esto devolverá un comentario en la propia *Issue* indicando el estado de la ejecución, al igual que se le actualizará al *label* de estado para indicarlo.

Con estas indicaciones y después de unos despliegues de servicios exitosos, se recomienda pasar a utilizar las plantillas de arquitectura, del mismo modo que configurar varias instancias simultáneamente. Esta opción es la más potente y diferencial de AWS-Autodeploy.

B. Agregar nuevos servicios a AWS-Autodeploy

Para crear un nuevo servicio y poder usarlo en una nueva plantilla, hay que realizar una serie de pasos. Se recomiendo implementar este nuevo código con el uso de un *IDE* para reducir los errores tipográficos. Para simplificar su proceso, en este apartado se aporta una pequeña **guía** de los pasos a seguir.

El primer paso para crear un nuevo servicio es crear su **plantilla de configuración** que usará Terraform para su despliegue. Para crearlas se pueden seguir las plantillas de ejemplo que ofrece Terraform en su documentación oficial [33].

Los únicos **requisitos** para esta plantilla son:

- La plantilla se debe crear en un nuevo fichero dentro del repositorio "terraform"
- La plantilla debe tener un nombre corto que identifique fácilmente el servicio que configura dicha plantilla
- El fichero de la plantilla debe tener la extensión ".rb.tf".

Una vez se tiene la plantilla obtenida de Terraform que cumpla con los requisitos, se modificar los atributos para poder hacerlos **parametrizables** y configurables de manera **dinámica**:

- Para cada uno de ellos, se debe asignar un nombre que lo identifique.
- El atributo "*service_instances*", pese a ser obligatorio, no se debe incluir en las plantillas.
- Se deben sustituir los atributos por el nombre identificativo insertándolos dentro de las marcas "<%= =>" para su correcta detección.

Con la plantilla creada, es momento de definir e implementar las **validaciones** para cada uno de estos atributos. En el directorio "templates" se encuentra el fichero "validate_template.rb" el cuál contiene todas las validaciones de AWS-Autodeploy:

- En el método *get_validations_for_service* se deben definir el conjunto de validaciones con expresiones regulares que se le aplicarán a los atributos.
- El valor que identifique a estas validaciones debe ser el mismo que se le puso al servicio en el fichero anterior
- Para definir validaciones más complejas, se implementarán "*case service*" en *validate*.
- Se pueden incluir parámetros que se aplicarán por defecto en caso de que el usuario no introduzca ningún valor, en el método *default_value_for_parameter*.
- El atributo "*service_instances*" es obligatorio, así que también es obligatorio implementar una validación para él.

Cuando se tienen creadas tanto la plantilla de configuración de Terraform, como la validación de sus atributos, el nuevo servicio ha quedado **configurado**. Pese a ello, aún no se podrá usar ya que antes hay que **agregarlo** en la lista de servicios disponibles para AWS-Autodeploy:

```
# run aws_autodeploy.rb
- name: Run main action
  [...]
  run: |
    ruby aws_autodeploy.rb ${{ github.event.label.name}} ${{ github.event.issue.number }}
"ec2,s3,rds,vpc,eks,glacier"
```

Figura 49. Fragmento del *workflow* principal donde se definen los servicios disponibles para AWS-Autodeploy.

Finalmente, se puede crear una plantilla del nuevo servicio creado para agilizar el proceso de desplegarlo. Para ello, habrá que crear en el directorio 'ISSUE_TEMPLATE' una plantilla con extensión '.md' y los campos configurables, según se haya creado la plantilla de Terraform.

C. Cómo usar AWS-Autodeploy con su interfaz gráfica

En primer lugar, el usuario que **accede** a la interfaz gráfica de AWS-Autodeploy debe registrarse. Para ello, deberá escribir su nombre de usuario, nombre del repositorio donde se aloja AWS-Autodeploy y su *token* de acceso de GitHub.

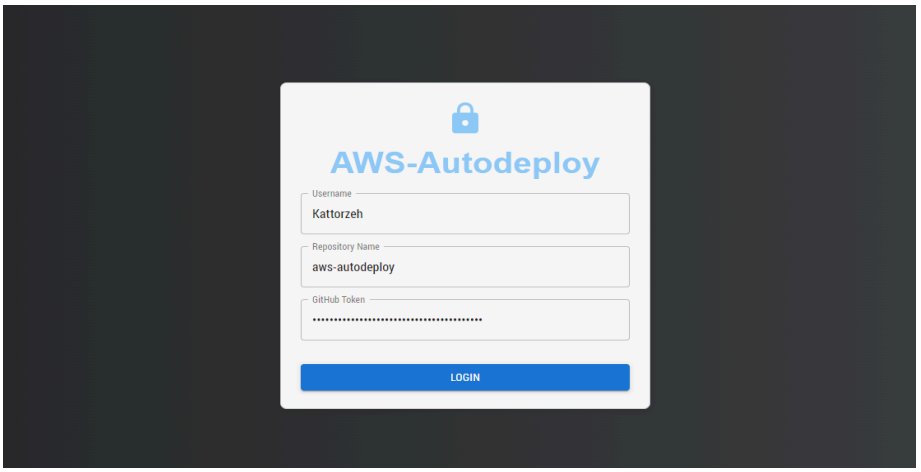

Figura 50. Imagen del registro en la interfaz gráfica de AWS-Autodeploy.

Una vez dentro, podrá **navegar** por las distintas plantillas, visualmente representadas con el logo del servicio de AWS y su nombre. El usuario deberá seleccionar la plantilla del servicio que desee desplegar.

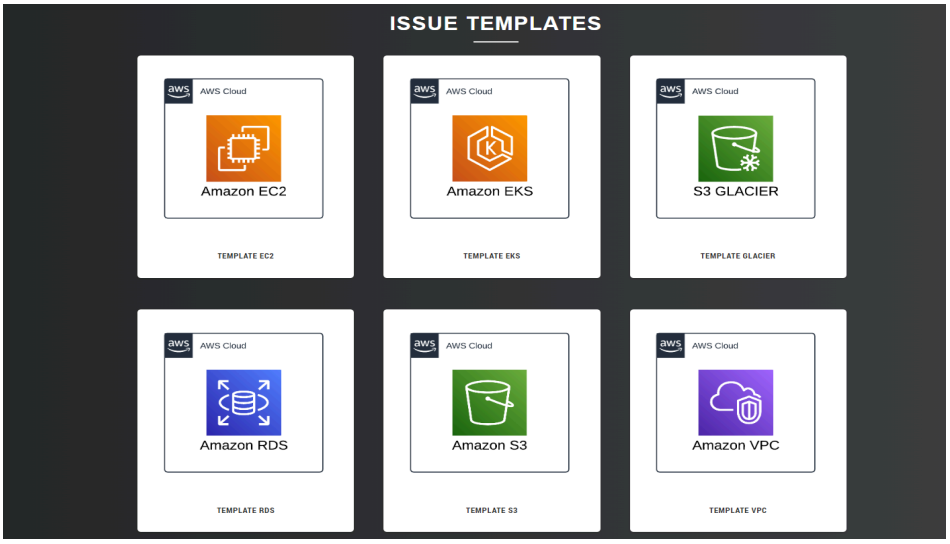

Figura 51. Imagen de las plantillas en la interfaz gráfica de AWS-Autodeploy.

Cuando se **selecciona** la plantilla, el usuario puede introducir los datos que desee en los campos del servicio en cuestión. Estos campos son obtenidos de la plantilla respectivamente, haciendo que el usuario no se deba preocupar por buscarlos o introducirlos manualmente.

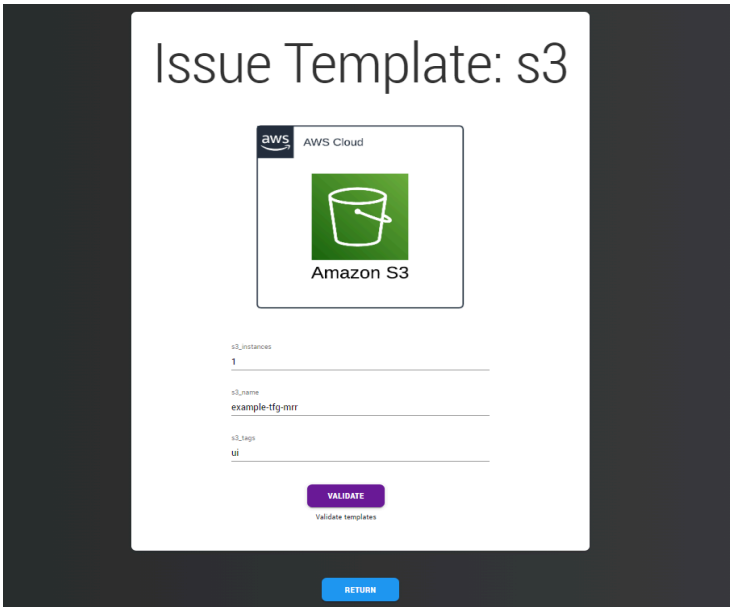

Figura 52. Imagen de campos de servicio en la interfaz gráfica de AWS-Autodeploy.

Una vez **completados**, si se pulsa sobre el botón "VALIDATE", la aplicación internamente se encargará de crear la *Issue* pertinente, con las *labels* "state-pending" y "action-validate". De este modo, el usuario no debe preocuparse de su implementación, de agregar un nombre a la *Issue* (se le asignará la fecha actual) o de comprobar si las *labels* se han agregado correctamente. Asimismo, si se pulsa sobre el botón "RETURN", se regresa a la cuadrícula donde aparecían las plantillas.

Al clicar sobre "VALIDATE", se **iniciará** la GitHub Action para validar los parámetros. Al usuario se le redirige a una nueva ventana, donde se le mostrará el resultado de esa ejecución. Adicionalmente, se le ofrecen tres nuevas acciones (deploy, delete y purge) para poder seguir avanzando en el ciclo de vida de la aplicación.

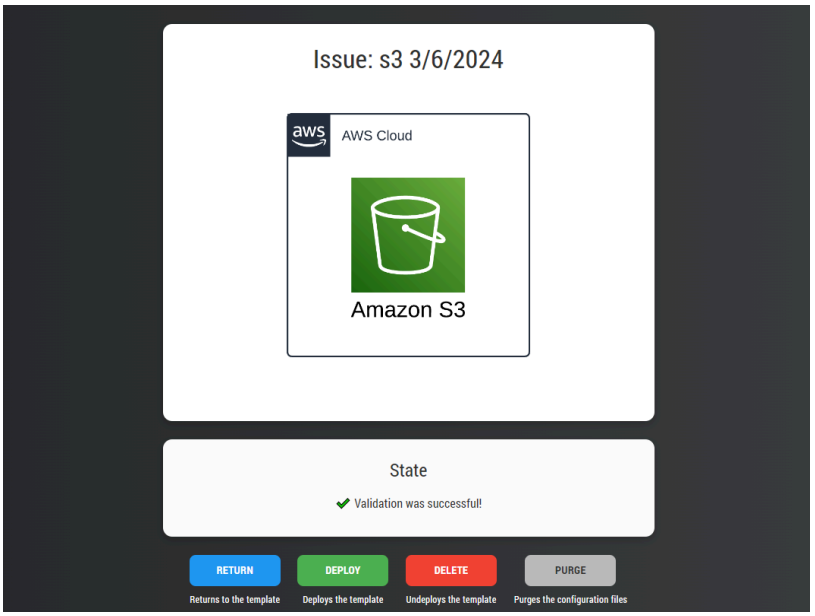

Figura 53. Imagen del resultado en la interfaz gráfica de AWS-Autodeploy.

En caso de que la **ejecución** falle, o bien porque el usuario desee actualizar la *Issue* (acción update), se puede regresar al formulario con los datos mediante el botón "RETURN". Al regresar, aparecerá el formulario con los datos introducidos previamente. En caso de que se trate de un error, el usuario puede modificarlos y luego intentar realizar la acción pertinente correctamente mediante el botón "RECOVER". Del mismo modo, si desea actualizar alguno de los campos cuando el servicio se encuentra desplegado, puede utilizar el botón "UPDATE". Como se puede observar, la acción "VALIDATE" ya no se encuentra disponible. Esto es debido a que dicha acción se oculta cuando la *Issue* ya ha sido creada y AWS-Autodeploy ya se encuentra en algún estado de su ejecución.

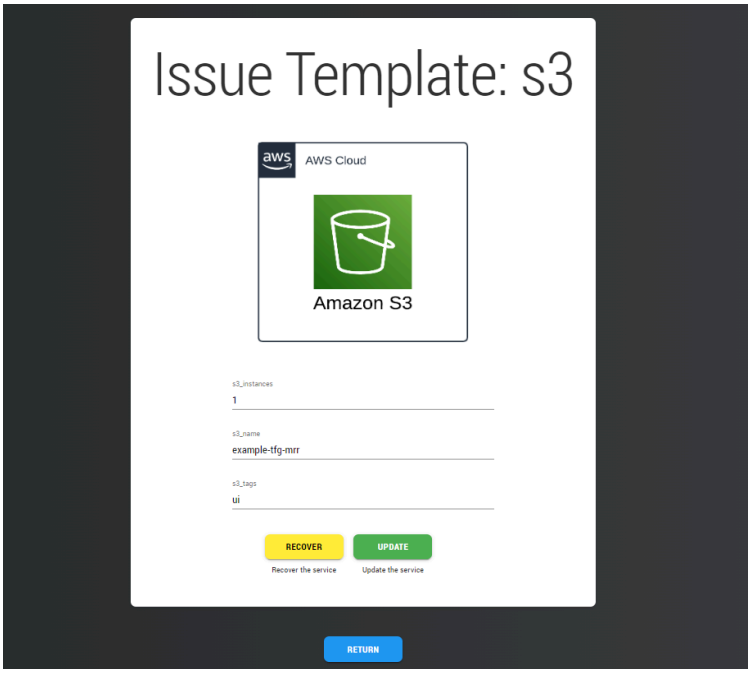

Figura 54. Imagen del formulario en la interfaz gráfica de AWS-Autodeploy.

UNIVERSITAT ROVIRA i VIRGILI