

UNIVERSIDAD DE SANTIAGO DE
COMPOSTELA



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA

Ciclo completo de CI/CD con Dagger y Kubernetes

Autor/a:

Daniel Vieites Torres

Tutores:

Juan Carlos Pichel Campos

Francisco Maseda Muiño

Grado en Ingeniería Informática

2025

Trabajo de Fin de Grado presentado en la Escuela Técnica Superior de
Ingeniería de la Universidad de Santiago de Compostela para la obtención do
Grado en Ingeniería Informática

Resumen

Ante la creciente complejidad de los flujos DevOps, resulta conveniente identificar herramientas de *Continuous Integration/Continuous Delivery* (CI/CD) que unifiquen los *pipelines* y reduzcan los costes de mantenimiento. Este trabajo investiga si Dagger, un motor programable basado en contenedores, puede simplificar y acelerar la integración y entrega continuas frente a los métodos convencionales predominantes.

Se construyó un *monorepo* con una aplicación de prueba (*frontend*: Vue + Typescript; *backend*: Typescript; MongoDB) y se diseñaron e implementaron dos módulos de Dagger para los ciclos de CI y CD, empleando el SDK que proporciona Dagger para el lenguaje de programación Go. Además, se desplegó la aplicación en tres *clusters* de KinD, gestionados por ArgoCD bajo filosofía GitOps, con la infraestructura definida mediante Helm y Kubernetes.

La evaluación cuantitativa incluyó una prueba en la que se ejecutó la función **endtoend** del módulo de CI. Esta mostró una reducción del tiempo de ejecución del 40 % cuando los parámetros de entrada cambiaban entre ejecuciones, y del 94 % cuando los parámetros permanecían sin cambios. Esto es gracias al sistema de gestión de caché nativo de Dagger. El análisis cualitativo destacó ventajas en cuanto al mantenimiento, la portabilidad (ejecución sobre *runtime* OCI), la reproducibilidad y la escalabilidad, frente a una curva de aprendizaje moderada.

En conclusión, Dagger se posiciona como una alternativa sólida para estandarizar y modernizar los ciclos de vida del *software*, facilitando la integración continua y reduciendo la dependencia de *scripts* heterogéneos. Se sugiere como mejora futura la validación del funcionamiento de los módulos en *clusters* gestionados en la nube.

Índice general

1. Introducción	1
1.1. Objetivos generales	1
1.2. Relación de la documentación	2
2. Estado del arte y fundamentos teóricos	4
2.1. CI/CD	4
2.2. Ecosistema de herramientas	6
2.3. Dagger	9
3. Diseño y arquitectura del sistema	14
3.1. Estructura general	14
3.2. zoo	15
3.3. helm-repository	20
3.4. state	20
4. Implementación del <i>pipeline</i> con Dagger	22
4.1. CI	22
4.2. CD	25
5. Pruebas	28
5.1. Entorno de pruebas	28
5.2. Prueba 1	29
5.3. Prueba 2	31
6. Conclusiones e posibles ampliaciones	34
A. Manuales técnicos	35
A.1. Descripción de tecnologías	35
A.2. Figuras	40
A.3. Listings	46
A.4. Gestión de secretos	53
A.5. Promoción de entornos	57
A.6. Desarrollo de la Chart de la aplicación	59

B. Manuales de usuario	62
B.1. zoo	62
Bibliografía	66

Índice de figuras

2.1. <i>pipelines</i> con Dagger sobre un <i>runtime</i> compatible con Docker.[26]	9
2.2. Uso de la API de GraphQL para Dagger.[33]	11
3.1. Comunicación entre paquetes de la aplicación. Imagen creada con excalidraw.com	16
5.1. Tiempos de ejecución, con y sin cambios en el código de la aplicación entre ejecuciones.	30
A.1. Diagrama de la organización de GitHub. Imagen creada con excalidraw.com	35
A.2. Proceso de integración continua.[4]. Referenciado en 2.1.	40
A.3. Proceso de despliegue continuo.[5]. Referenciado en 2.1.	40
A.4. Arquitectura de Kubernetes.[19]. Referenciado en 2.2.	41
A.5. Arquitectura de Helm.[25]. Referenciado en 2.2.	41
A.6. Clusters y comunicación con el repositorio de estado. Imagen creada con excalidraw.com. Referenciado en 3.2.	42
A.7. Encriptado y desencriptado de secretos. Imagen creada con excalidraw.com. Referenciado en A.4, 3.2 y 3.4.	42
A.8. Descripción del proceso de promoción de entornos. Imagen creada con excalidraw.com. Referenciado en 3.2 y A.5.	43
A.9. Diagrama de organización de las Charts de la aplicación. Imagen creada con excalidraw.com. Referenciado en 3.3.	43
A.10. Diagrama de la estructura del repositorio de estado. Imagen creada con excalidraw.com. Referenciado en 3.4	44
A.11. Diagrama del diseño del módulo de CI con Dagger. Imagen creada con excalidraw.com. Referenciado en 4.1.	45

Índice de cuadros

5.1. Diferencias entre Dagger frente a métodos convencionales	33
B.1. Software y versiones utilizadas durante el desarrollo	65

Índice de Listings

2.1. Comando para crear un cluster con KinD	9
2.2. Aplicar la configuración de ArgoCD con kubectl	9
2.3. Comando para lanzar el backend del proyecto	12
4.1. Funciones del módulo de Dagger de CI	23
4.2. Encadenamiento de funciones del módulo de CI	24
5.1. Borrado de caché de Dagger y Docker	29
5.2. Testeo integral de la aplicación con el módulo de CI	29
A.1. Construir y correr una imagen de Docker	36
A.2. Despliegue con Docker Compose	37
A.3. Makefile para compilación de un programa en C	46
A.4. Extracto de justfile utilizado en el proyecto	46
A.5. Extracto de Dockerfile utilizado en el proyecto	47
A.6. docker-compose.yaml usado en el proyecto	48
A.7. Código de Dagger con CUE	49
A.8. Ejemplo del SDK de Go de Dagger.	49
A.9. Script de creación de los clusters	50
A.10. Configuración del cluster de dev	51
A.11. Configuración de Argo en dev	51
A.12. Definición de la Chart umbrella de la aplicación	52
A.13. Archivo de configuración de helmfile	52
A.14. Archivo de valores de zoo-backend en dev	53
A.15. Archivo de configuración de SOPS	54
A.16. Archivo 'kustomization.yaml'	55
A.17. Generador de los secretos	55
A.18. Valores que pueblan la Chart de ArgoCD	56
A.19. Workflow de CI/CD	58
A.20. Generación de un archivo comprimido de la Chart	59
A.21. Workflow de publicación de la Chart en GH Pages	60
B.1. Estructura principal del repositorio “zoo”	62
B.2. Archivo de secretos .env	64
B.3. Archivo de secretos .secrets.yaml, para pruebas con act	65

Capítulo 1

Introducción

1.1. Objetivos generales

Objetivos principales

En este trabajo se pretende demostrar y evaluar la viabilidad, eficiencia y flexibilidad de Dagger[1] como motor de CI/CD (*Continuous Integration/Continuous Delivery*)[3, 6], con el fin de estandarizar y modernizar los ciclos de vida del desarrollo de software.

Se realizará el mismo flujo de trabajo, tanto utilizando Dagger como sin usarlo, y se compararán los resultados en cuanto a rendimiento. Se evaluarán las ventajas que tiene su uso frente a métodos convencionales, entre las que destacarán su portabilidad, al correr sobre un *runtime* de Docker; y su capacidad programática, ya que se pueden crear *pipelines* implementando funciones en el lenguaje conocido para el desarrollador. En vez de coordinar *scripts* creados a mano en diferentes entornos, el programador es capaz de componer acciones reusables, utilizando un lenguaje de programación y una API (*Application Programming Interface*) REST a su disposición.

Se van a proporcionar ejemplos de módulos creados con Dagger, los cuales estarán especialmente diseñados para cumplir los ciclos tanto de CI como de CD de una aplicación *dummy*. De esta manera se podrá comprobar que este mismo proceso se puede llevar a cabo para cualquier aplicación, y de una manera sencilla.

Lo que sigue son las preguntas a las que este trabajo busca responder:

- ¿Qué grado de complejidad tiene desarrollar módulos de Dagger para la gestión de un ciclo de CI/CD de una aplicación?
- ¿Vale realmente la pena aprender a utilizar esta herramienta?
- ¿Es capaz de aumentar la velocidad de desarrollo de una aplicación?
- ¿Es fácilmente integrable en cualquier tipo de aplicación?

- ¿Qué puntos débiles corrige Dagger frente al uso de otros métodos convencionales?
- ¿Qué desafíos, limitaciones o desventajas se encuentran al trabajar con Dagger?

Objetivos secundarios

Para lograr los objetivos principales es necesario llevar a cabo varios pasos:

- Creación de un *monorepo*[7] en GitHub.
- Diseño y creación de una aplicación de prueba. Esta consistirá en una página web de gestión de un zoo, la cual realizará peticiones a una API REST que estará conectada a una base de datos.
- Implementación de un *pipeline* CI/CD con y sin Dagger.
- Entorno orquestado por Kubernetes[8], configurado a través de una Chart de Helm[9].
- Análisis comparativo de las ventajas que ofrece Dagger frente a métodos convencionales.

1.2. Relación de la documentación

- Capítulo 1: Introducción.

Este capítulo, en el cual se describen la finalidad del proyecto, las tecnologías a utilizar, de manera breve; y la estructura, a grandes rasgos, del trabajo en sí.

- Capítulo 2: Estado del arte y fundamentos teóricos.

En el segundo capítulo se detallan los conceptos más importantes de CI/CD. Además, se estudia la evolución de las herramientas DevOps[16], incluyendo Dagger como una de las últimas y más innovadoras herramientas en este sector, y se compara con las otras tecnologías.

- Capítulo 3: Diseño y arquitectura del sistema.

Aquí se describe la organización de repositorio, así como las tecnologías utilizadas para implementar cada uno de las piezas de software del trabajo. Por lo tanto, se habla de la aplicación de prueba y de los módulos de Dagger. También se explica cómo se ha organizado la infraestructura de despliegue.

- Capítulo 4: Implementación del *pipeline* con Dagger.

Aquí se detallan los pasos que se han dado para crear los *pipelines* con Dagger, utilizando el SDK para definirlos como código. Este es el núcleo del proyecto.

- Capítulo 5: Pruebas y resultados.

En este capítulo se presentan las pruebas que se han llevado a cabo. Se habla de las dificultades que se han tenido, así como de las ventajas que ofrece Dagger frente a otras tecnologías, aportando comparaciones cuantitativas y cualitativas.

- Capítulo 6: Conclusiones y líneas futuras.

Finalmente, se resumen los hechos que se han obtenido, se valora el resultado final del uso de Dagger y se indica si ha cumplido con las expectativas. Además, se añaden puntos de mejora o extensiones del proyecto.

Capítulo 2

Estado del arte y fundamentos teóricos

Antes de empezar a escribir código, se deben entender los conceptos fundamentales que permitirán llevar a cabo este trabajo.

Dagger busca mejorar el ciclo completo de CI/CD de una aplicación. Por lo tanto, es fundamental definir los conceptos de *Continuous Integration* y el *Continuous Delivery*. Una vez se comprenda a qué se refieren esos términos, se podrán entender los métodos y tecnologías convencionales que permiten implementar dichos procesos. Será entonces cuando se pueda introducir Dagger, una herramienta innovadora para realizar *pipelines*.

2.1. CI/CD

CI/CD son las siglas de *Continuous Integration/Continuous Delivery*, o en casos más específicos, este último también se puede conocer como *Continuous Deployment*.

Se trata de un conjunto de pasos automatizados, utilizados en el desarrollo de software para llevar el código desde su implementación inicial hasta el despliegue de la aplicación. Estos pasos incluyen:

- Integración de cambios en el código.
- Compilación de la aplicación con los cambios realizados.
- Realización de pruebas.
- Creación y publicación de imágenes de Docker y paquetes NPM.
- Despliegue de la aplicación.

Continuous Integration

Se basa en la integración de código de manera constante, día a día, en un repositorio compartido por programadores. Cada uno de los programadores realiza cambios en el código y lo integra en el repositorio. Una vez se realizan cambios, estos deben pasar una serie de pruebas antes de poder ser incorporados de forma definitiva en el código fuente de la aplicación (Fig. A.2).

Desde hace años se utilizan sistemas de control de versiones para gestionar el código de cualquier proyecto. Este tipo de herramientas permite a un equipo controlar el estado del código en cada momento, siendo capaces de conocer el historial de los cambios realizados, saber quién ha hecho cada cambio y tener la capacidad de revertir alguna modificación en el caso de ser necesario. La herramienta de control de versiones más utilizada hoy en día, y la que se utiliza en este proyecto, es Git[12].

La integración de código en un repositorio no se trata simplemente de modificar una porción de un archivo y subirlo. El código debe ser probado antes de integrarlo completamente en el núcleo de la aplicación. Durante el proceso de integración continua, cada vez que se modifica algo de código, se debe:

- Construir la aplicación.
- Pasar pruebas de funcionalidad.
- Pasar el *linting* del propio código, es decir, verificar que el código cumpla ciertos estándares de estilo.
- Reportar cualquier error en el caso de que exista.

Todo lo anterior se debe realizar de manera automatizada, con el fin de integrar el código modificado en el repositorio lo más rápido posible, evitando en la medida de lo posible la intervención humana.

Continuous Delivery

Tras haber construido la aplicación durante el proceso de integración continua, toca desplegar la aplicación. El despliegue automático de nuevas versiones de una aplicación que han pasado el ciclo de CI se conoce como “despliegue continuo” (figura A.3).

Esto, evidentemente, tiene como requisito que la aplicación que se está construyendo tenga el despliegue como uno de los pasos en su ciclo de vida, lo cual no tiene por qué ser así. En este trabajo sí que ocurre, ya que la aplicación *dummy* que se construye es una página web, junto con una API y una base de datos.

Es necesario que exista relación entre los desarrolladores y los encargados de desplegar la aplicación. Sin embargo, hoy en día encontramos en muchas empresas un conjunto de metodologías conocidas como DevOps[16], lo cual implica que ciertos integrantes de un equipo deben tener conocimiento tanto del desarrollo de la aplicación como del despliegue de la misma.

Esta transición a la cultura DevOps permite a los equipos desplegar sus aplicaciones más fácilmente. Además, incluye la necesidad de que el despliegue sea una parte muy importante en el proceso de desarrollo.

Al igual que en la integración continua, en este ciclo también es necesario automatizar el proceso despliegue de una aplicación. Esto siempre va a disminuir la posibilidad de error humano.

Con el despliegue continuo podemos tener *feedback* más rápido por parte del usuario, lo que permitirá mejorar y corregir errores más rápidamente. Además, se despliegan con más frecuencia cambios realizados en la aplicación, por lo que los errores en producción son menos probables y, en el caso de que los haya, más fáciles de corregir. Esto es gracias también a llevar un historial de los cambios mediante una herramienta de control de versiones como Git.

GitOps & ArgoCD

Basándose en la filosofía DevOps, mencionada antes, que abarca tanto el ciclo de CI como el de CD, existen un conjunto de prácticas en las que se utiliza Git como fuente de verdad para la gestión de la infraestructura y las aplicaciones. Esto es conocido como GitOps[23]. Utilizar Git como la única fuente de información permite gestionar de manera consistente la infraestructura de las aplicaciones. Los cambios en un repositorio hacen que se ejecuten *workflows*, o secuencias de acciones de CI/CD que implementan los cambios en el entorno correspondiente.

La herramienta que realiza estas prácticas de GitOps, y que se utiliza en este trabajo, es ArgoCD[24]. Argo es una herramienta que, como su nombre indica, facilita el despliegue continuo. Supervisa repositorios de Git para aplicar los cambios que se realizan en ellos automáticamente en el *cluster* (A.1).

2.2. Ecosistema de herramientas

Un *pipeline* moderno se compone de diferentes tipos de herramientas, cada una con sus características y finalidades. Se pueden agrupar en los ciclos que se han indicado anteriormente, CI y CD. El grupo de herramientas de CI facilitan la construcción y empaquetado de la aplicación que se va a construir, mientras que las de CD permiten la aplicación empaquetada previamente.

Herramientas de construcción y empaquetado

Como cabe esperar, los pasos mencionados en 2.1, que forman parte de la integración continua, van a depender del tipo de aplicación que se esté construyendo, y de las tecnologías que se estén utilizando. Además, esta secuencia de acciones pueden incluir unos pocos comandos en trabajos o proyectos sencillos, o necesitar varios *scripts* complejos en el caso de aplicaciones más avanzadas. Por

lo tanto, es necesario tener una herramienta de construcción que permita realizar los pasos mencionados anteriormente, sin la necesidad de memorizar cada uno de los comandos o *scripts* que hay que ejecutar.

Make

Para ello existe `make`[13], una aplicación de línea de comandos que permite definir bloques de comandos o reglas, aportando a cada bloque un nombre u objetivo que se pretende obtener ejecutando dicha regla. Se suele crear un archivo llamado `Makefile` para definir todas las reglas que se precisen.

Un ejemplo muy típico de compilación de un programa escrito en C sería el que se puede observar en el Listing A.3.

Just

En este trabajo se utiliza una herramienta de construcción más moderna y polivalente llamada `just`[14]. Este software tiene la misma finalidad que `make`, ejecutar comandos específicos de un proyecto. Pero este incluye muchas más funcionalidades (A.1).

En A.4 se puede ver que se hace uso de comandos y sintaxis propios de sistemas UNIX (por ejemplo, Bash), por lo que se trata de un requisito de software contar con un entorno UNIX o compatible (Linux, macOS, WSL, etc.) para su correcta ejecución.

Docker

Docker[17] es una herramienta que permite empaquetar aplicaciones, creando imágenes con las dependencias necesarias para que la aplicación se lance sin problemas. Las imágenes generadas se pueden ejecutar, creando contenedores, que son entornos completamente aislados del contexto de la máquina en la que han levantado. Estos contenedores son muy ligeros en cuanto a espacio y uso de recursos, ya que almacenan únicamente lo necesario para correr el software que queremos desplegar (A.1).

La mayor ventaja que proporciona el empaquetado de aplicaciones con Docker, es la portabilidad. Aunque hay que tener en cuenta la arquitectura de la máquina, las imágenes se pueden lanzar en cualquier entorno con Docker instalado, lo cual evita el conocido problema de: “En mi máquina funciona”.

Plataformas de despliegue

Docker Compose

Con Docker se es capaz de gestionar varios servicios desplegados en distintos contenedores. Pero existe una herramienta que apareció poco después y que fa-

cilita esta tarea, llamada “Docker Compose”[18]. Esta permite simular entornos con múltiples contenedores para desarrollar localmente (A.1).

Docker Compose no es una plataforma de producción, se utiliza únicamente con el fin de desarrollar localmente, y es muy útil en el caso de querer hacer pruebas rápidas de una aplicación sencilla. Se puede tomar como un precursor conceptual a la orquestación más compleja que realiza Kubernetes.

Kubernetes, Helm & KinD

El proyecto Kubernetes nació un año después que Docker. Es una herramienta de software que permite orquestar contenedores. Permite gestionar el ciclo de vida de las aplicaciones en contenedores que viven en un *cluster*. Entre sus características principales destacan:

- Escalado automático.

Aumenta o disminuye automáticamente el número de contenedores en ejecución. Esto va a depender de la cantidad de réplicas de una misma aplicación que se hayan indicado en su configuración. Kubernetes siempre va a intentar mantener el estado del *cluster* cumpliendo los parámetros que se indicaron en las plantillas de configuración de cada uno de los servicios.

- Autorreparación.

Si un contenedor falla, se reinicia o se reemplaza por otra instancia del mismo servicio, garantizando la continuidad de este.

- Descubrimiento de servicios y balanceo de carga.

Se exponen los contenedores entre ellos y/o a Internet. Además, permite distribuir el tráfico de red, evitando así sobrecargas.

Más información en A.1.

Complementando a Kubernetes tenemos Helm (figura A.5), que se trata de un gestor de paquetes para Kubernetes. Su propósito es ayudar a instalar y administrar el ciclo de vida de las aplicaciones de Kubernetes. Además, las Helm Charts son formatos de archivos YAML que te permiten definir los objetos de Kubernetes de una manera dinámica. Esto te permite definir aplicaciones mucho más complejas .

Para desplegar una aplicación de Kubernetes es necesario tener un *cluster* preparado para soportar este tipo de recursos. Un *cluster* de Kubernetes es un conjunto de máquinas o nodos que trabajan juntos con el fin de ejecutar y gestionar aplicaciones que corren dentro de contenedores. Para crear los *clusters* en los que se despliega la aplicación se utiliza KinD[21].

KinD (*Kubernetes in Docker*) permite crear *clusters* de Kubernetes de manera local. Se utiliza esta herramienta debido a su sencillo uso. Simplemente utilizando el comando que se muestra en el Listing 2.1 se puede construir un *cluster* en el que desplegar cualquier aplicación de Kubernetes.

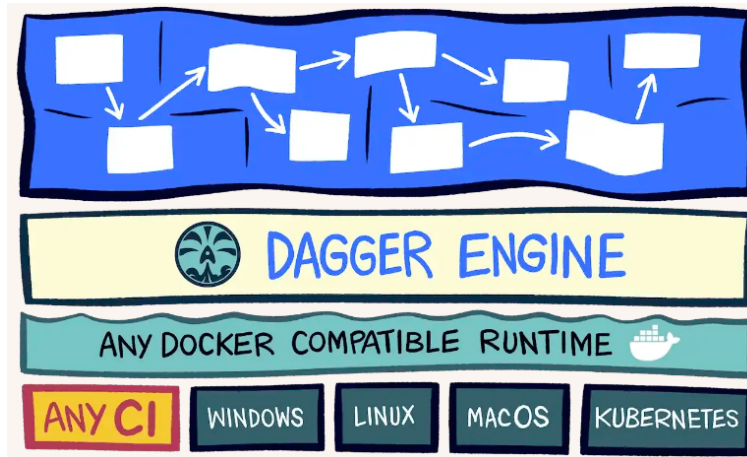


Figura 2.1: *pipelines* con Dagger sobre un *runtime* compatible con Docker.[26]

```
1 | kind create cluster
```

Listing 2.1: Comando para crear un cluster con KinD

Para gestionar los propios recursos de Kubernetes que existen dentro del *cluster* se utiliza la herramienta `kubectl`[22]. Se trata de un CLI oficial de Kubernetes que permite comunicarse con el controlador principal del *cluster*, utilizando la API de Kubernetes. Un ejemplo de uso se pueden observar en el Listing 2.2

```
1 | kubectl apply -f "argo/argo_dev.yaml" --context kind-dev
```

Listing 2.2: Aplicar la configuración de ArgoCD con `kubectl`

El uso de estas dos herramientas se ve más en detalle en la sección 3.2.

2.3. Dagger

Dagger es el pilar fundamental de este trabajo. Se trata de un kit de desarrollo de software que permite a los desarrolladores crear *pipelines* CI/CD, y ejecutarlos en cualquier sitio.

La principal idea de los creadores de Dagger siempre ha sido poder crear *pipelines* portables, que no sea necesario implementarlos de nuevo cada cierto tiempo debido a cambios en el entorno de desarrollo o de pruebas. Esta portabilidad se consigue permitiendo a los desarrolladores utilizar cualquier *runtime* de OCI (*Open Container Initiative*[27]) para ejecutar las funciones que forman parte del SDK[29] de Dagger, así como las que definen los propios desarrolladores. Además, desde el principio se ha evitado el uso de archivos YAML, que está siendo el lenguaje de configuración más utilizado por parte de la mayoría de aplicaciones.

CUE

Dagger comenzó utilizando un lenguaje de configuración muy potente llamado CUE[28]. Este se podría ver como una extensión de JSON, pero con más funcionalidades. Todo lo escrito en JSON se puede traducir a CUE, pero no al revés. En el Listing A.7 se puede ver un ejemplo de código de Dagger en el que se utiliza CUE para lanzar un “plan”. En él se descarga la imagen de Docker de Alpine y se almacena en un registro levantado en la máquina local.

Esta es la primera vez que se pueden ejecutar *pipelines* definidas de manera programática, ejecutadas sobre un *runtime* de Docker, y de manera relativamente funcional. Además, se puede ejecutar toda la secuencia de acciones de manera local, permitiendo realizar pruebas y buscar errores sin necesidad de hacer uso de otras herramientas de CI como GitHub Actions.

Además, Dagger hace un uso exhaustivo de la caché. Todas las acciones son cacheadas automáticamente. Esto es una funcionalidad muy útil, ya que va a hacer que un *pipeline* se ejecute hasta un 90 % más rápido, como ha ocurrido en pruebas realizadas en este trabajo, que se comentan más adelante. Si se pone como ejemplo la ejecución de tests sobre una aplicación, la primera vez que se lanza el *pipeline*, este tiene que ejecutarse completamente:

1. Instalar las dependencias.
2. Compilar la aplicación.
3. Correr los tests.

Dependiendo del tipo de aplicación y, evidentemente, de la conexión a Internet, esto puede tardar desde unos segundos hasta varios minutos. Pero con Dagger solo ocurre una vez. La primera vez. Gracias a la caché, todas las acciones repetitivas, como la instalación de dependencias o la compilación de la aplicación, se almacenan en la caché, ahorrando así mucho tiempo a la hora de realizar pruebas de cualquier tipo. Mientras tanto, con otras herramientas se tendría que esperar siempre la misma cantidad de tiempo para cada una de las veces que se quiere lanzar el *pipeline*. Y, además, muchas veces ni siquiera sería de manera local, habría que depender de aplicaciones que pueden fallar o no estar disponibles en algún momento.

CI/CD como código

Dagger a querido ir más allá pensando que los desarrolladores deberían ser capaces de crear sus *pipelines* de la misma manera que crean sus aplicaciones, escribiendo código. Es así como el SDK de Go[31] para Dagger, el cual se utiliza en este trabajo. Go es un lenguaje de programación con muchos casos de uso. Desde la creación de servicios de red y en la nube, hasta aplicaciones CLI y desarrollo web. Es conocido también por su simplicidad en cuanto a sintaxis y por tener una librería estándar muy completa, aportando muchas de las herramientas

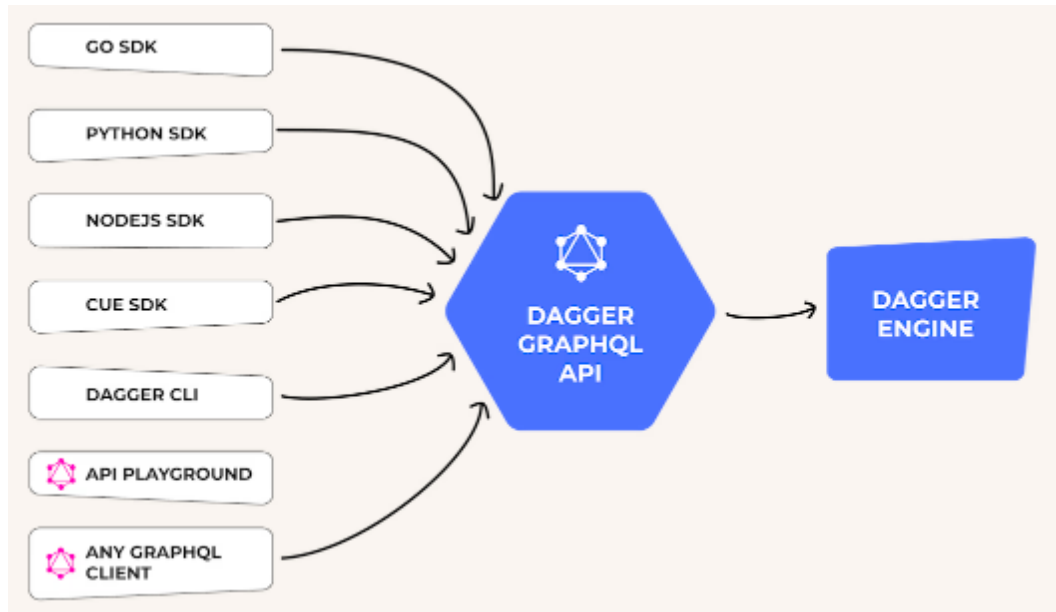


Figura 2.2: Uso de la API de GraphQL para Dagger.[33]

necesarias para realizar proyectos comunes. Además, facilita la implementación de programación concurrente, gracias a las *goroutines*, similares a los hilos de ejecución de un sistema operativo, pero mucho más ligeros. En el Listing A.8 se puede observar un ejemplo de código utilizando el SDK de Go.

Todo lo anterior hace de Go una elección excelente para empezar la lista de lenguajes de programación sobre los que el equipo de Dagger implementaría su propio SDK, que hasta el día de hoy incluyen: Java, PHP, Node y Python.

GraphQL

Pero, ¿cómo es capaz el equipo de Dagger de desarrollar SDKs específicos para cada lenguaje tan rápidamente? Gracias al uso de GraphQL[32], un lenguaje para manipulación y consulta de datos.

Se puede ver su funcionamiento en la Figura 2.2. El SDK de cada uno de los lenguajes funcionan como traductores del código que escribes en dicho lenguaje a sentencias que entiende el motor de Dagger. Esto es a través de la API de GraphQL de Dagger. El motor de Dagger es el que se encarga de ejecutar las instrucciones en un entorno controlado. De esta manera, no son los propios SDKs los que corren los programas dependiendo del lenguaje, sino que funcionan como clientes de la API para traducir la secuencia de acciones y ejecutarse en un mismo entorno.

Dagger *functions*

Entonces, tras varias mejoras y nuevas versiones, el equipo de Dagger implementó las “funciones de Dagger”. Estas funciones son el componente principal de Dagger hoy en día. Cada una de las operaciones principales de Dagger se pueden llamar a través de una función, utilizando una API. Además, estas se pueden encadenar, generando *pipelines* dinámicas en una sola llamada. De esta manera, se puede decir finalmente que gracias a Dagger podemos programar nuestros ciclos CI/CD, en el lenguaje que queramos, dentro de los que están disponibles.

Daggerverse

La existencia de las funciones de Dagger permite la creación de módulos, un conjunto de funciones que toman una entrada y producen una salida en concreto. Los módulos creados por la comunidad se pueden encontrar en el Daggerverse[34]. Este es el lugar en el que se comparten módulos de Dagger, los cuales se pueden reutilizar para añadir funcionalidades a otro módulo, sin necesidad de volver a crear una lógica de programación ya implementada por otra persona o equipo de desarrolladores.

CLI

Otra funcionalidad que tiene Dagger es su CLI[35] (*Command-Line Interface*). A través de ella puedes llamar a funciones de módulos de Dagger, tanto de de tu sistema de archivos local como directamente de un repositorio de Git.

```
1 | dagger call --sec-env=file:///.../.env backend service up --ports
   ↪ 3000:3000
```

Listing 2.3: Comando para lanzar el backend del proyecto

De esta manera (Listing 2.3) es como se ejecutan las funciones que se definen en los módulos creados en este trabajo. Toda la documentación al respecto se puede encontrar en B

Resumen

Dagger es una herramienta revolucionaria que redefine la manera de crear *pipelines* de CI/CD, permitiendo implementarlos como código y evitar que los desarrolladores tengan que lidiar con archivos de configuración estáticos como YAML. Ofrece SDKs en una variedad de lenguajes de programación, tales como Go, Python o Node, que actúan como clientes del motor central de Dagger.

Una de sus ventajas más significativas es su portabilidad, la cual se consigue al ejecutar todas las operaciones sobre un *runtime* de Docker. De esta manera se garantiza que cualquier *pipeline* definido con Dagger funcione de igual manera sin importar la máquina en la que se ejecuta.

Otro de sus pilares es la gestión que hace de la caché. Dagger cachea cada una de las acciones ejecutadas. Gracias a esto, tras la primera ejecución de un *pipeline*, las siguientes ejecuciones son significativamente más rápidas. Así se reducen los tiempos de espera, permitiendo a los desarrolladores trabajar más rápido.

Finalmente, la evolución hacia las “Dagger *functions*” y la creación de módulos permite un gran nivel de reutilización. Estos módulos, que pueden ser compartidos a través del Daggraphverse, junto con su CLI para invocarlos, proponen una manera muy poderosa de crear *pipelines* para construir, probar y desplegar aplicaciones.

Capítulo 3

Diseño y arquitectura del sistema

3.1. Estructura general

El código del trabajo y todo lo que abarca se encuentra almacenando en GitHub. Se han creado los repositorios necesarios en una misma organización de GitHub.

Entre los repositorios creados se pueden encontrar:

- **zoo.**

Este es el repositorio principal. Se trata de un *monorepo*, en el que se encuentra implementado todo el código necesario dentro del presente trabajo de investigación. Dentro del repositorio se encuentran:

- La aplicación de prueba sobre la que se apoya el proyecto, y que da nombre al repositorio, debido a que se trata de una aplicación de gestión de un zoo.
- Los módulos de Dagger para realizar los ciclos de CI y CD.
- Otros archivos, como *scripts* y archivos de configuración.

- **helm-repository.**

Este repositorio alberga las Charts de Helm que definen la estructura necesaria para desplegar la aplicación de prueba.

- **state.**

Se trata del repositorio que, siguiendo el enfoque GitOps, almacena los valores que poblarán los recursos de Kubernetes según el entorno en el que se despliegue la aplicación. Además, en este repositorio también existe una rama de despliegue, de la cual ArgoCD lee los manifiestos de los recursos que debe desplegar para cada uno de los entornos.

En la figura A.1 se muestra un diagrama de la disposición de los repositorios y la relación entre ellos.

3.2. zoo

Como se ha comentado anteriormente, el repositorio *zoo* está estructurado como un *monorepo*. Un *monorepo* es un repositorio con diferentes proyectos, los cuales se encuentran interrelacionados de una manera bien definida. A lo largo de esta sección se justifica la elección de este tipo de estructura, a la vez que se explica cómo están implementados las diferentes piezas de software.

Aplicación de prueba

La primera razón para escoger este tipo de estructura es el hecho de querer crear una aplicación relativamente pequeña, una página web que consta de un *frontend* y un *backend* (se hará referencia a estos como “paquetes” a partir de ahora). Por lo tanto, se hace más sencillo gestionar estos dos paquetes si se ubican en un único repositorio.

Otra ventaja de utilizar un *monorepo* tiene que ver con el software utilizado para crear los paquetes de la aplicación de prueba. Ambos se implementan utilizando Node.js, en lenguaje Typescript[36]. Los paquetes tienen dependencias propias, y se puede dar el caso de que ambos utilicen una o varias dependencias iguales. Usar un *monorepo* permite tener esas dependencias en un mismo lugar, evitando su duplicado. Con esto se consigue reducir el tiempo de construcción de los paquetes.

Sin embargo, es necesaria una herramienta que permita manejar los paquetes de manera independiente. Alguno de los motivos para tener esta preferencia pueden ser: que haya dos equipos de desarrolladores, uno para cada paquete; o que se quiera publicar versiones, hacer tests, u otro tipo de tarea sobre cada paquete por separado. La herramienta que se utiliza en este trabajo se llama Lerna[37]. Este software está específicamente diseñado para gestionar *monorepos* de proyectos de Node.js. Entre las ventajas que proporciona se encuentran:

- Gestión de tareas locales.
- Cacheo local de salidas de comandos, con posibilidad de que dicha caché sea compartida entre entornos, por ejemplo, con agentes de CI.
- Detección de paquetes afectados por cambios en el código.
- Análisis de la estructura del proyecto.

Por los beneficios anteriormente comentados, y más, es por lo que se ha elegido esta herramienta para gestionar el *monorepo*.

En cuanto a las tecnologías que se utilizan en la aplicación, ya se ha mencionado Typescript como lenguaje principal. Este lenguaje permite tener un sistema tipado, lo cual puede ser útil para detectar muchos errores comunes mediante el análisis estático en tiempo de construcción. Esto reduce las posibilidades de errores en tiempo de ejecución.

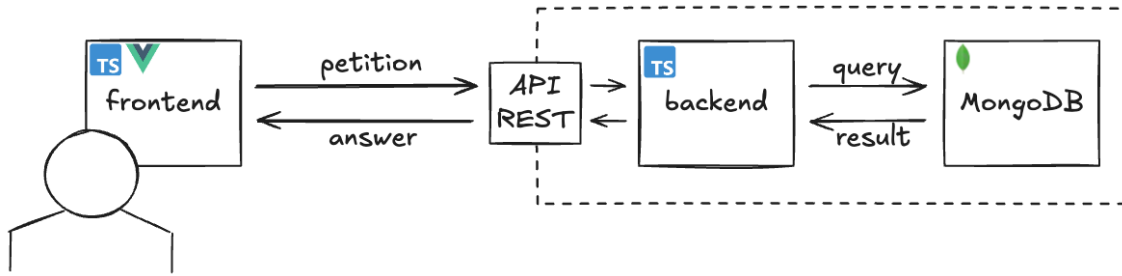


Figura 3.1: Comunicación entre paquetes de la aplicación. Imagen creada con excalidraw.com

El *backend* está completamente desarrollado utilizando dicho lenguaje. Su funcionalidad es proporcionar una API REST que el *frontend* pueda utilizar para realizar cambios en la base de datos. Se usa MongoDB[38] como base de datos debido a que es fácil de gestionar y porque solo se almacena información sobre animales, sin ningún tipo de relación entre ellos, en una única tabla o documento.

El *frontend* se ha implementado utilizando Vue.js[39], un *framework* que permite construir interfaces web mediante componentes reactivos. Se ha escogido este frente a otras opciones debido a su facilidad de uso sin conocimiento previo. Tiene una API intuitiva, por lo que no tiene una gran curva de aprendizaje. Además, el propio *framework* está construido utilizando Typescript, por lo que tiene compatibilidad de primera clase con este lenguaje.

En la figura 3.1 se puede ver un diagrama que muestra cómo es la comunicación entre los paquetes de la aplicación, y con la base de datos, junto con las tecnologías que se utiliza en cada uno de ellos.

Módulos de Dagger

Se integran también en el repositorio los módulos de Dagger de CI y de CD. Estos módulos se incluyen en el *monorepo* para facilitar la referencia a los paquetes que constituyen la aplicación de prueba. Además, resulta lógico que se ubiquen en el mismo lugar una aplicación y las herramientas que permiten su evolución, como son cualquier tipo de *software* que realice las funciones de CI y de CD.

Ambos módulos se realizan utilizando el SDK del lenguaje Go que proporciona Dagger. Se ha escogido este lenguaje debido al conocimiento previo que ya se tenía de este. Además, es el lenguaje en el que está implementado el propio Dagger.

Uno de los módulos se encarga del ciclo de CI, es decir, de realizar los tests de la aplicación, del *linting* o análisis del código en sí, y de la publicación de imágenes de Docker y paquetes NPM. Está organizado de manera que se pueden gestionar cada uno de los paquetes de la aplicación de manera independiente. Esto también es posible gracias al uso de Lerna, que ya se ha comentado anteriormente.

El segundo de los módulos, el de CD, realiza la tarea de publicación de los recursos de Kubernetes, los cuales son posteriormente obtenidos por ArgoCD para su despliegue completo. Esto se consigue haciendo uso de los repositorios `helm-repository` y `state`, en los cuales se almacenan las Charts de Helm y los valores que pueblan dichas Charts, respectivamente.

Se detalla más profundamente la implementación de los módulos de Dagger en el capítulo 4.

Creación y configuración de los *clusters*

La fase final del ciclo de una aplicación es el despliegue. En este trabajo se levantan tres *clusters* de KinD de manera local. Estos son los lugares en los que se despliega la aplicación. Generalmente se tienen diferentes *clusters* con el fin de probar la aplicación en entornos distintos antes de desplegarla en el principal, que sería el de producción. El hecho de crearlos todos localmente hace que sean más sencillas las pruebas relacionadas con el despliegue. En equipos de desarrollo reales, los entornos de producción se encuentran en la nube. Sin embargo, sí que se pueden llegar a tener entornos locales para realizar pruebas de la aplicación.

Los *clusters* se crean con el *script* que se muestra en el Listing A.9 (se han puesto comentarios en vez de código algunas partes para reducir su tamaño, a modo de pseudocódigo). Este *script* está escrito para funcionar en sistemas UNIX, en Bash, por lo que es un requisito utilizar un sistema operativo como MacOS o una distribución de Linux para probar el *script*. No funcionará en un sistema operativo Windows. A modo de resumen del *script*:

- Se crean tres *clusters* con su configuración específica. Se puede ver el archivo de configuración del *cluster* de `dev` en el Listing A.10.
- Se instala en cada *cluster* un controlador de Ingress, lo cual permitirá acceder a la aplicación a través de una URL customizada desde el exterior.
- Se incluye como Secret el valor de la clave de cifrado que permite descifrar los secretos. El proceso de creación de cifrado y descifrado se explica en 3.2.
- Se instala la Chart de Helm de Argo. A esta Chart se le pasan una serie de valores, de los cuales se habla en 3.2. Más información sobre cómo acceder a la interfaz de Argo en



Important!

! CITAR MANUAL DE USUARIO

- Se aplica la configuración específica de Argo para el entorno que se está creando. Se puede ver la configuración para Argo en el entorno de `dev` en el Listing A.11. En dicha configuración se indica el repositorio, la ruta desde la raíz donde se encuentran los recursos y la rama de donde Argo debe obtener los archivos que definen los recursos que se van a desplegar.
- Se añade un banner en la parte superior de la interfaz para distinguir cada uno de los *clusters*.
- Se muestra la contraseña del usuario `admin`, para poder hacer *log in* a través de la interfaz de Argo.

En la figura A.6 se puede observar cómo se sincroniza ArgoCD, en cada uno de los *clusters*, con el repositorio de estado. Argo reacciona cada vez que se realizan cambios en el directorio que le corresponde, dentro de la rama de despliegue del repositorio `state`. En ese momento, obtiene de nuevo todos los archivos de definición de los recursos y se sincroniza con el estado deseado.

Gestión de secretos

La aplicación de prueba consta de una base de datos, la cual tiene usuario y contraseña. Este es un ejemplo de datos que es necesario almacenar como un secreto o Secret de Kubernetes. Debido a que se está utilizando un método *pull*[11] de despliegue de la aplicación; es decir, la herramienta que realiza el despliegue, ArgoCD, “tira” (hace *pull*) del repositorio que le indicamos como objetivo; es necesario almacenar en el repositorio los secretos encriptados previamente. Esto es una buena práctica para evitar que se filtren sin querer los datos al exterior, aunque el repositorio sea privado.

Existe un apartado completamente dedicado a como se realiza la gestión de secretos en A.4.

Función de cada *cluster* y promoción de entornos

A continuación se explica para qué se utiliza cada *cluster* y cuál es el proceso de despliegue de la aplicación en cada uno de los entornos.

- **dev.**

Se trata del *cluster* de desarrollo. En este se despliega la aplicación en el momento en el que se añade una nueva funcionalidad, ya sea en el *frontend* o en el *backend*. Esto implica, en términos de GitHub:

1. Crear una *Pull Request* (PR) en la que se implementa la nueva funcionalidad. Esta debe ser lo más reducida posible, cumpliendo con la filosofía de CI. Esto implica que la intención del equipo de desarrollo debe ser desplegar nuevas funcionalidades o correcciones de errores en

producción en el menor tiempo posible. Se puede conseguir esto planeando PRs cortas en cuanto a tiempo de desarrollo, evitando que el equipo tenga demasiado trabajo en progreso y asignando los recursos necesarios para que cada PR se lleve a cabo lo más rápidamente[40].

2. Implementar la funcionalidad o realizar la corrección pertinente. A medida que se implementa, se puede, y es una buena práctica, ejecutar localmente el ciclo de CI para asegurarnos de que se pasan las pruebas y el *linting* del código. Esta es una de las ventajas principales de utilizar Dagger. El desarrollador puede comprobar de manera local si el código actualizado es capaz de pasar el *pipeline* de CI, lo cual evita tener errores inesperados a la hora de integrar el código en la rama principal.
3. Revisar que la tarea que correspondía hacer en dicha PR se ha realizado correctamente.
4. Integrar la funcionalidad o corrección en la rama principal del repositorio.

Tras haber terminado todos los pasos anteriores, se ejecuta un *workflow* de GitHub que realiza todo el ciclo de CI y CD, independientemente del entorno en el que se vaya a desplegar. El *workflow* es el que se ve en el Listing A.19. Para más información sobre los pasos que son necesarios para la promoción de entornos ver A.5.

Lo que se consigue con el *workflow* mencionado es ejecutar los *pipelines* de CI y de CD para la versión actual de la aplicación, lo cual permite desplegarla completamente en el entorno que le corresponde. El entorno en el que se despliega depende del evento que produce la ejecución del *workflow*, que puede ser: una integración de código en la rama principal, la creación de una *prerelease* o la creación de una *release*. Los eventos anteriores hacen que se despliegue en **dev**, **pre** y **pro**, respectivamente.

■ pre

Este es el entorno de pre-producción. Este tipo de entornos están diseñados para simular el entorno de producción real, y funciona como prueba final previa a la publicación de una aplicación. En el caso de este trabajo, como ya se ha comentado, todos los entornos son idénticos, pero en equipos y entornos reales, cada uno de ellos tiene características distintas.

■ pro

Finalmente, el entorno de producción. Aquí es donde se despliega la aplicación de manera abierta a los usuarios. Como se lleva insistiendo a lo largo de este capítulo, lo normal es que estos entornos se encuentren en la nube. Para mayor facilidad de pruebas y debido a que no es la finalidad de este trabajo, se ha decidido crear todos los *clusters* de forma local.

En la figura A.8 se muestra cómo el *workflow* escucha los diferentes eventos que hacen que se ejecute el ciclo de CI/CD. Dependiendo del evento que ocurre, se va a utilizar una *tag* diferente y se va a desplegar en el entorno correspondiente.

3.3. helm-repository

Es una práctica habitual tener un repositorio con todas las Charts de Helm que utiliza el equipo de desarrollo. Esto permite tener un lugar centralizado para que los desarrolladores compartan y los usuarios encuentren aplicaciones para desplegar en Kubernetes.

Este repositorio tiene como finalidad lo que se acaba de comentar, funcionar como el lugar en el que se almacena la Chart de Helm de la aplicación de prueba.

La figura A.9 muestra que se ha creado una Chart *umbrella* llamada **zoo**. Una Chart de este tipo funciona como una agrupación de más **subcharts** que están interrelacionadas. Normalmente se utiliza este método para gestionar despliegues de aplicaciones complejas en Kubernetes. La definición de la Chart **zoo** se puede ver en el Listing A.12. En esta se indican las dependencias (*subcharts*) que agrupa la Chart. Entre estas se encuentran las Charts propias de la aplicación, **zoo-frontend** y **zoo-backend**. Además, se puede ver que se utiliza una Chart ya definida en un repositorio conocido, perteneciente a Bitnami[46]. De este repositorio se obtiene la Chart de MongoDB. De esta manera, se utiliza una Chart ya implementada y desarrollada por expertos, proporcionando muchas opciones de configuración y ofreciendo mucha más seguridad de lo que se tendría en el caso de crear una Chart de Mongo propia para la aplicación.

También se puede observar en la figura A.9 los recursos que se crean para cada una de las *subcharts* de la aplicación. Para más detalle en cuanto a su funcionamiento, ver A.1.

Se detalla el proceso de desarrollo de la Chart de la aplicación en A.6.

3.4. state

El repositorio de estado funciona como única fuente de verdad. Aquí se definen los valores que pueblan la Chart de la aplicación. Se utiliza una herramienta llamada **helmfile**[47]. Esta permite integrar Charts de Helm y valores, aunque estos se encuentren en lugares diferentes. En la figura A.10 se muestra la estructura del repositorio de estado, incluyendo la rama principal y la rama de despliegue.

Se puede ver el archivo de configuración de **helmfile** en el Listing A.13. Los archivos del apartado de **environments** indican el *namespace* y la versión de la Chart que se va a utilizar. Todos los valores que se indican en el apartado de **releases** dependen del entorno que se ha elegido para desplegar. Los valores de

las líneas 19-22 son aquellos que no dependen del entorno en el que se quiere desplegar, mientras que los de las líneas 23-26 sí que dependen del entorno, y se almacenan en un directorio dedicado a cada uno de ellos. En estos últimos archivos de valores dependientes del entorno se indica, por ejemplo, la *tag* a utilizar de las imágenes de Docker.

Un ejemplo de archivo de valores se puede observar en el Listing A.14. Como se ha comentado, el valor de la *tag* se ve indicado en la línea 3, con el formato que se menciona en A.5.

En la sección

**Important!**

| CITAR SECCIÓN DE DAGGER CD

se explica cómo se especifica el entorno y, por lo tanto, los valores que se van a utilizar para poblar la Chart de la aplicación.

Este repositorio también es el lugar en el que se almacenan los archivos que definen los recursos que se van a desplegar en cada entorno. En la figura A.7 se puede ver la estructura que tiene uno de los directorios (**dev**) de la rama de despliegue del repositorio *state*. La estructura de dicho directorio en la rama **deploy** es la misma para los directorios correspondientes a los demás entornos.

Capítulo 4

Implementación del *pipeline* con Dagger

Este capítulo describe la implementación del *pipeline* con Dagger. Se parte de una versión inicial sin la herramienta, y se llega a un ciclo completo, plenamente integrado con esta herramienta. El objetivo es demostrar que Dagger aporta mayor flexibilidad a los desarrolladores para construir *pipelines* de CI y CD, sin depender de tener un entorno de desarrollo ni de una configuración específica para ejecutarlos.

4.1. CI

Tras realizar una primera aproximación de la aplicación de prueba, el siguiente paso consiste en realizar el ciclo de CI. En este ciclo se realizan los tests de la aplicación, el *linting* del código y la publicación de imágenes de Docker y de paquetes NPM. Al principio se comenzó a implementar el ciclo completo sin el uso de Dagger. Se utilizó un `justfile` con los comandos necesarios para realizar tanto los tests como el *linting* de los paquetes de la aplicación, y se crearon dos *scripts*: uno para la publicación de imágenes y otro para la publicación de paquetes NPM. Además, se crea un Dockerfile para poder realizar las pruebas *end-to-end* del *frontend* con Cypress[48].

La desventaja con respecto al Dagger es evidente: *scripts* y comandos que “funcionan en mi máquina”. Siguiendo esta práctica, sería necesario que todo el equipo de desarrollo utilizara el mismo entorno de desarrollo, y aún así, no se garantizaría el funcionamiento del ciclo de CI, debido a posibles configuraciones que cada desarrollador haya hecho a su sistema que pueda diferir de las configuraciones de los demás. Con este método, los desarrolladores no tienen margen de maniobra en cuanto al sistema que deben utilizar.

Ahora se explica el diseño e implementación con Dagger del mismo ciclo de CI.

Una vez el ciclo de CI sin Dagger se ha realizado al completo, se pasó a traducir los scripts y comandos a un módulo de CI con Dagger, utilizando el SDK de Go.

En la figura A.11 se puede comprobar la estructura de este módulo. Se ha optado por implementar un objeto principal **Ci**, el cual tiene la capacidad de hacer uso de cualquiera de los otros dos objetos customizados, uno específicamente diseñado para gestionar el *frontend* y otro para el *backend*. Este diseño concuerda con la propia estructura y filosofía del *monorepo*: elementos separados que realizan tareas diferentes, pero que se interrelacionan desde un punto central, en este caso con la herramienta Lerna para el *monorepo*, ya mencionada anteriormente; y el objeto **Ci** en el módulo.

Por lo tanto, la idea del diseño del módulo es la misma que la del *monorepo*, tener un elemento principal desde el que hacer uso de cualquiera de los otros tipos customizados[49], que se corresponden con los paquetes de la aplicación.

En Dagger se hace uso del “encadenamiento”. Esto permite llamar a las funciones de los objetos que devuelven las propias funciones. Así, se puede llamar a la función del objeto principal del módulo que devuelve el objeto **Backend**, y se tendrá acceso a las funciones que están definidas para este paquete.

Para entender cómo funciona, se explicará un ejemplo sencillo con pseudocódigo del propio módulo. En el Listing 4.1 se puede ver un ejemplo de dos funciones del módulo de CI (se han escrito comentarios de lo que se hace en el código para reducir su tamaño). La primera función corresponde con el objeto principal del módulo. Este construye la imagen base y posteriormente crea un objeto **Backend**, el cual devuelve como referencia al final de la función. El hecho de que este objeto sea devuelto por la función, permite acceder a las funciones que este tiene definidas.

Una función definida para **Backend** es **Test**, que aparece justo después. En esta se utiliza la imagen base que se ha pasado como parámetro en la construcción del objeto. Se ejecuta dentro del contenedor que define dicha imagen el comando que permite correr los tests del paquete *backend* de la aplicación. Como se puede comprobar, se utiliza Lerna para ejecutar el *script test*, definido en el *package.json* del paquete correspondiente, indicado mediante el parámetro *scope*.

```

1 // Ci (main.go)
2
3 type Ci struct {
4     // +required
5     SecEnv *dagger.Secret
6     secrets secrets
7 }
8
9 func (m *Ci) Backend(
10 ctx context.Context,
11 // +defaultPath="/"
12 src *dagger.Directory,
```

```

13 ) (*Backend, error) {
14     base, _ := m.Init(ctx, src)
15
16     // Se obtienen las claves y los valores de los secretos para
17     // el *backend*.
18
19     return &Backend{
20         Name:      "backend",
21
22         // Imagen base, con todo lo necesario para ejecutar los
23         // comandos que se precise.
24         Base:      base,
25
26         // Claves y valores de los secretos obtenidos antes.
27         Secrets: SecMap{Keys: keys, Values: values},
28     }, nil
29 }
30
31 // Backend (backend.go)
32
33 func (m *Backend) Test(ctx context.Context) (string, error) {
34     return m.Base.
35         WithExec([]string{"lerna", "run", "test", "--scope", "
36         ↪ @vieites-tfg/zoo-backend"}).
37         Stdout(ctx)
38 }

```

Listing 4.1: Funciones del módulo de Dagger de CI

La manera en la que se ejecuta a través de un comando la cadena de funciones anterior es como aparece en el Listing 4.2. El comando se realiza en el directorio de trabajo del módulo de CI. Se hace una llamada a Dagger, se pasan los valores requeridos por el módulo, los cuales se indican en la definición del objeto principal; y finalmente se empieza a llamar a las funciones que tiene cada uno de los objetos. A las funciones se las llama en formato “kebab-case”, aunque en el propio código estén en “camelCase”. Por lo tanto, primero se realiza la llamada a la función **backend** y, posteriormente, sabiendo que esta función devuelve un tipo **Backend**, se llama a la función **test** que este tiene definida.

```

1 | dagger call --sec-env=file:///../../.env backend test

```

Listing 4.2: Encadenamiento de funciones del módulo de CI

De esta manera se puede lanzar la función que se prefiera de cualquiera de los paquetes. También, esta estructura y diseño del módulo facilita la escalabilidad de la aplicación, permitiendo añadir fácilmente más paquetes en el caso de ser necesario. Lo único que habría que hacer sería crear un objeto nuevo con funciones para las tareas que se quieran realizar en este, y añadir una función en el objeto principal del módulo que devuelva el nuevo objeto creado para el paquete.

Junto con la ejecución de los tests y del *linter*, se añade también el módulo la posibilidad de levantar por separado cada uno de los paquetes. Esto quiere decir

que se permite acceder de manera local a los servicios de los paquetes, la página web por un lado y la API por el otro.

Lo primero que se hace es conseguir levantar la aplicación entera de la manera más sencilla posible. En este caso, se utiliza un **Dockerfile** con varios pasos, en los cuales se construye y se prepara la imagen para la ejecución de cada uno de los paquetes. Esto se consigue con construcciones *multi-stage*, un ejemplo de esto se ha mencionado ya en 2.2.

Con el **Dockerfile** anterior y con la construcción de un **docker-compose.yaml** se consigue levantar la aplicación de manera local. El uso de este método conlleva crear las imágenes cada vez que se quiere desplegar. Con el uso de Dagger, que se explica más adelante, se verá que muchos pasos se cachean y, por lo tanto, se ahorra tiempo de despliegue.

Tras conseguir esto, se traduce de nuevo la lógica al módulo de Dagger. Se crea en cada uno de los objetos del módulo una función **service**, que permiten levantar los paquetes por separado y acceder a ellos de manera local. Gracias a estas funciones es posible realizar los tests *end-to-end* de la aplicación, ejecutando con una sola función en el objeto principal:

- Tests y *linting* de cada uno de los paquetes.
- Levantar los servicios de los paquetes, permitiendo su comunicación de manera interna.
- Correr los tests de la aplicación con Cypress.

Todo lo anterior integrado en el módulo y ejecutado con una única llamada a una función del objeto principal de la aplicación.

Aquí es donde realmente se ve la potencia de Dagger. Todo se programa en un lenguaje conocido para el desarrollador, con sintaxis que no depende del sistema en el que se está desarrollando y sin la aparición de problemas relacionados con el entorno de ejecución, ya que siempre va a funcionar, independientemente del sistema. Esto es gracias a ejecutar todo sobre un *runtime* de OCI, como Docker. Para realizar todo lo anterior sin Dagger, habría que: lanzar los comandos de Lerna de manera local, levantar los servicios con Docker Compose y correr los tests de Cypress con un **Dockerfile** específico para la causa. Y aún haciendo todo esto, no se tendría la flexibilidad de realizar pruebas y obtener *logs* que se tiene con Dagger, mediante el cual se es capaz de acceder al contenedor en cualquier momento de la ejecución del *pipeline*.

4.2. CD

El ciclo de CD involucra: la construcción de los recursos de Kubernetes, el encriptado de los secretos y la publicación de todos los manifiestos necesarios en el repositorio de estado.

Para comenzar con este ciclo, se crearon un par de *scripts*, uno que permitía construir las imágenes de Docker y su publicación y otro que facilitaba la publicación de los paquetes NPM. Esto tiene las desventajas ya comentadas, son *scripts* que se ejecutan de manera local y que, por lo tanto, dependen del entorno en el que se ejecutan. En el caso de cambiar de entorno de desarrollo, sería necesario crear nuevos *scripts* o acomodarlos al nuevo entorno. Esto solo trae problemas innecesarios a los desarrolladores, que se tienen que centrar en solucionar estos inconvenientes, externos a la finalidad del desarrollo de la aplicación en sí.

Tras crear los script anteriores, se pasó a la construcción de las Charts de Helm de la aplicación. Se diseñó la estructura de las Charts y los componentes que iban a tener cada uno de los servicios de la aplicación. Entonces se definieron todos los recursos con Helm y se realizaron pruebas hasta que finalmente todo funcionaba como se esperaba.

La construcción de las Charts era necesario realizarlo tras ser capaz de publicar las imágenes de Docker al Container Registry. Esto es porque Kubernetes utiliza imágenes construidas que obtiene de cualquier registro que se le indique, no se le puede pasar un *Dockerfile* para que construya la imagen.

Todo el proceso de creación de las Charts se ha realizado en local, utilizando comandos de Helm. Una vez comprobada la funcionalidad de esta, se procedió a la separación de responsabilidades: un repositorio para las Charts por un lado (**helm-repository**), y otro repositorio con los valores que pueblan las Charts por otro lado (**state**). Esto facilita el desarrollo y mantenimiento de las Charts. Además, se tiene una única fuente de verdad para obtener los valores.

Con los distintos repositorios creados, es necesario utilizar la herramienta **helmfile** para integrar Charts y valores. El archivo de configuración de esta herramienta se incluye en el repositorio de estado, manteniendo así la norma de tener toda la gestión de valores en un mismo lugar.

Así, ya es posible desplegar la aplicación utilizando Kubernetes. Lo siguiente es hacer uso de Dagger para permitir realizar el despliegue. Se comienza utilizando un método *push* de despliegue. En el propio módulo de Dagger que se creó inicialmente, se utiliza **helmfile** para desplegar la aplicación en un *cluster* de KinD. No existe ArgoCD, ni varios entornos. Por ahora momento es sencillo y directo, se crea un *cluster* con un módulo de KinD para Dagger, y se ejecuta el comando de **helmfile** necesario para desplegar la aplicación.

A continuación se decide emplear el método *pull*, ya comentado, donde existe una herramienta que es la que se encarga de obtener los recursos que se van a desplegar. Esa herramienta es ArgoCD. El módulo de CD de Dagger pasa de desplegar la aplicación a ser el encargado de publicar en la rama **deploy** del repositorio **state** los recursos que se van a desplegar en cada entorno. Esto implica

1. Realizar el *template* de los recursos con **helmfile**, es decir, mostrar el renderizado de estos pero sin desplegarlos.

2. Obtener los secretos y encriptarlos.
3. Crear los archivos que dice a ArgoCD cómo desencriptar los secretos,
4. Publicar todo en la rama de despliegue del repositorio de estado.

Tras tener en funcionamiento el módulo, se procede a crear el *workflow* de GitHub, cuyo funcionamiento se explica en A.5. En resumen, se encarga de ejecutar los módulos de CI y CD, ejecutando los tests *end-to-end* previamente a la publicación de las nuevas imágenes de Docker y actualizando el repositorio de estado con la definición de los recursos en el entorno que proceda, utilizando las nuevas imágenes creadas.

De esta manera, se integra todo lo que se ha creado:

- Los módulos de Dagger de CI y de CD.
- Las Charts de Helm y valores.
- Los *clusters* de KinD con los entornos levantados, con ArgoCD instalado en cada uno de ellos.
- Un *workflow* de GitHub que enlaza todo lo anterior.

Capítulo 5

Pruebas

En este capítulo se realizan pruebas con el fin de medir, cuantitativa y cualitativamente, las ventajas y/o desventajas que tiene el uso de Dagger para la gestión de un ciclo completo de CI/CD.

Se proponen dos pruebas diferentes:

- Ejecución de los tests *end-to-end* con Dagger.
- Construcción y ejecución de tests de los paquetes de la aplicación. Tanto con Dagger como sin él.

En cada uno de los apartados dedicados a las diferentes pruebas se indica lo que implica realizarlos.

5.1. Entorno de pruebas

Las pruebas se realizan en un ordenador portátil con las siguientes características de *software* y *hardware*:

- *PC*: MacBook Air, 13-inch, 2024
- *Chip*: Apple M3
- *RAM*: 16 GB
- *Sistema Operativo*: MacOS Sequoia 15.5
- *Red*: Por cable, aproximadamente 1 GB simétrico.

Las versiones de los binarios utilizados se pueden encontrar en



Important!

! CITAR MANUAL DE USUARIO

5.2. Prueba 1

Lo que se pretende comprobar con esta prueba es ver cómo afecta el almacenamiento de caché que lleva a cabo Dagger tras varias ejecuciones de una función. Se realizarán medidas de tiempo bajo ciertas condiciones, con el fin de obtener diferentes valores y observar cuánto tiempo permite ahorrar Dagger gracias a su gestión de la caché. Se ejecuta varias veces la función `endtoend` implementada en el módulo de CI. Dicha función realiza las siguientes acciones:

- Ejecuta los tests y el *linting* del paquete del *backend* de la aplicación.
- Realiza el *linting* del *frontend*.
- Levanta los servicios de ambos paquetes.
- Ejecuta los tests del *frontend* con Cypress.

En definitiva, con la función que se va a utilizar en esta prueba, se realiza un testeo integral de toda la aplicación.

Antes realizar esta prueba se borra la caché que pudieran tener almacenada tanto de Dagger como de Docker. Para esto se utilizan los comandos del Listing 5.1.

```
1 dagger core engine local-cache prune
2
3 docker system prune -af
```

Listing 5.1: Borrado de caché de Dagger y Docker

Para probar la función, simplemente es necesario ejecutar el comando que se muestra en el Listing 5.2.

```
1 dagger call --sec-env=file:///../../.env endtoend
```

Listing 5.2: Testeo integral de la aplicación con el módulo de CI

Se lanza el comando anterior cuatro veces, realizando un cambio en el código de la aplicación entre cada una de las ejecuciones. Esto tiene como finalidad evitar que se haga un uso total de la caché, debido a que Dagger comprueba los parámetros de entrada que se pasan a cada función (el código fuente, en este caso) para decidir si va a utilizar la caché o no. Si los parámetros de entrada cambian, no hace uso de la caché de dicha función, y la ejecuta. En el caso de que no cambien los parámetros de entrada, se hace uso de la salida almacenada en la caché para esa función.

En el cuadro 5.1 se pueden observar los tiempos que se obtienen tras realizar las ejecuciones del comando anterior. Los resultados en azul muestran los tiempos obtenidos haciendo cambios en el código de la aplicación entre cada una de las ejecuciones. Se observa que, tras la primera vez, se reduce el tiempo de ejecución un 40 %. A partir de ahí, el tiempo se mantiene constante, y se continuaría con este tiempo en consecuentes ejecuciones, como muestra la línea azul punteada.

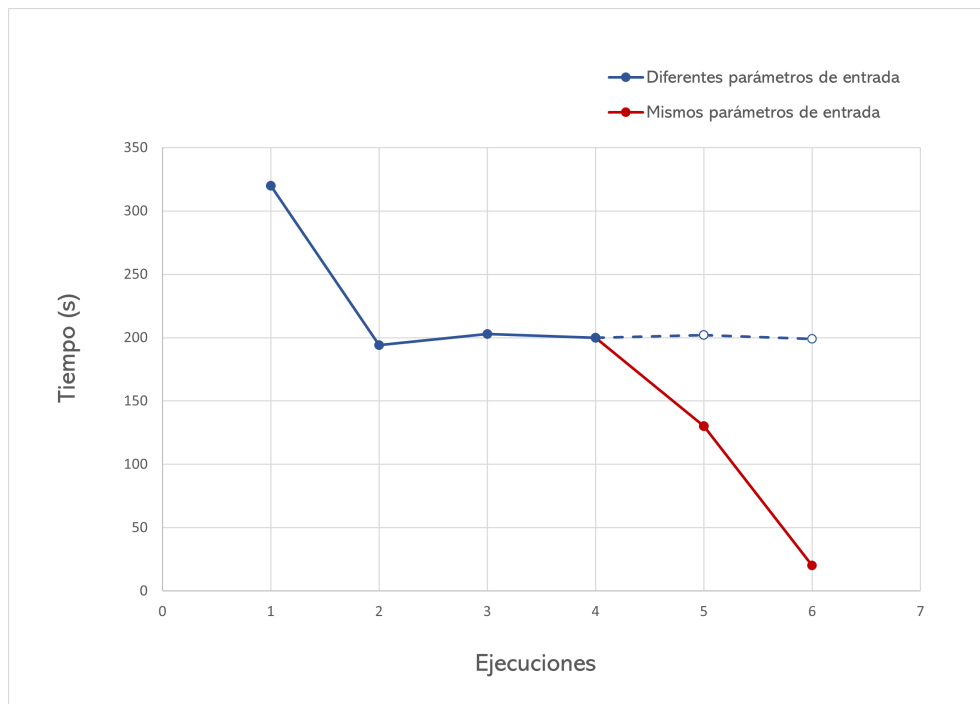


Figura 5.1: Tiempos de ejecución, con y sin cambios en el código de la aplicación entre ejecuciones.

En rojo se muestran los tiempos obtenidos posteriormente, esta vez sin realizar cambios en el código y, por lo tanto, manteniendo los parámetros de entrada de la función iguales a como estaban en la última ejecución. Esto tiene como resultado una caída drástica del tiempo de ejecución, debido a lo comentado previamente. Dagger comprueba que la entrada de la función para la ejecución actual es la misma que la anterior, la cual tiene almacenada en caché. Por lo tanto, no ejecuta la función, y toma como salida la que ya había obtenido anteriormente. Así, se obtiene una reducción con respecto al tiempo inicial de un 94 %.

Por lo tanto, se puede observar la buena gestión que Dagger hace de la caché. Es capaz de reducir un tercio el tiempo de ejecución de la misma función, con cambios en el código entre ejecuciones. En el caso de realizar pruebas sin hacer cambios en los parámetros de entrada, el tiempo de ejecución se reduce un 94 %. Este último valor de 20 segundos no es realista para el caso de uso de estos módulos, donde la entrada de las funciones (el código fuente) debería cambiar constantemente. Sin embargo, puede ser muy útil conocer esta capacidad de Dagger para aplicarlo en otras situaciones.

5.3. Prueba 2

La finalidad de esta prueba es comparar cómo se gestionan acciones básicas en el desarrollo de *software*, como son la construcción de la aplicación y la ejecución de tests, tanto utilizando el módulo de Dagger como sin usarlo. Por lo tanto, se busca un resultado cualitativo, en el que lo importante reside en la facilidad del desarrollador para implementar y mantener lo necesario para llevar a cabo dichas acciones.

El módulo de Dagger de CI es el encargado de realizar las acciones comentadas. Se proporciona una función de construcción y de ejecución de tests para cada uno de los paquetes. El ciclo de construcción y testeo de la aplicación está creado íntegramente utilizando un único lenguaje de programación, lo cual ofrece simplicidad tanto a la hora de la implementación como del mantenimiento. Se puede utilizar lógica de programación para adaptar el *pipeline* a distintos escenarios. El desarrollador también puede reutilizar código de manera muy sencilla, mediante la creación de funciones, a las cuales se las puede llamar desde cualquier sitio. El código está organizado y se previene la duplicación de lógica de programación.

La ejecución del módulo se puede hacer de manera local gracias a que Dagger lanza sus funciones sobre un *runtime* de OCI, como Docker. Dagger también facilita la reproducibilidad gracias a que construye grafos para definir dependencias entre funciones, donde cada nodo del grafo define su entrada y su salida. Además, lo anterior también permite gestionar de mejor manera la caché, haciendo que las ejecuciones sean más rápidas, como ya se ha visto en la primera prueba de este capítulo.

En el caso de que no se use Dagger, el desarrollador tendría que optar por métodos convencionales. Comenzaría por crear un *Dockerfile* que permitiera realizar la construcción de los paquetes de la aplicación. Para realizar los tests del *frontend*, haría falta otro *Dockerfile* específico para utilizar Cypress. Estos tests implicarían también implementar un archivo de Docker Compose, el cual permite levantar todos los servicios al mismo tiempo con posibilidad de comunicación entre ellos, necesario para realizar los tests *end-to-end* del *frontend*.

Además, extender la lógica de programación implicaría crear *scripts*, los cuales habría que acomodar a un nuevo entorno de desarrollo en el caso de que este cambie en algún momento. Esto añade más carga al desarrollador, que tiene que mantener dichos *scripts* a lo largo de los diferentes entornos en los que se quieran ejecutar. Crear diferentes *scripts* hace la reutilización de código más complicada, y puede llevar a la duplicación de lógica de programación. Finalmente, también se encuentra el problema de la reproducibilidad, siendo esto más complicado de conseguir debido a las pequeñas diferencias que pueda haber en la configuración del entorno de desarrollo de cada programador.

Sin embargo, la realidad es que las tecnologías y herramientas que se utilizan de manera convencional son más conocidas que Dagger. Por lo tanto, inicialmente puede ser lógico encaminarse por el uso de estas tecnologías, ya que permite ini-

ciar la implementación de este tipo de ciclos más rápidamente. Además, aunque la API de Dagger es intuitiva si ya conoces las otras herramientas convencionales, su amplitud introduce una pequeña curva de aprendizaje. Pero, como se ha comentado, el mantenimiento a la larga es más sencillo con Dagger, debido a que todo se encuentra en un mismo lugar e implementado con un mismo lenguaje de programación, como Go, Python o Typescript (entre otros). Y, lo más importante, Dagger tiene una gran capacidad de portabilidad gracias a que ejecuta todas las funciones en entornos aislados, como los contenedores de Docker. Esto es una ventaja frente a los *scripts* que es necesario mantener en diferentes entornos en el caso de que no se utilice Dagger.

En el cuadro 5.1 se proporciona una tabla a modo de resumen, con las diferencias que se encuentran entre las dos opciones de implementación que se han comentado.

<i>Aspecto</i>	<i>Con Dagger</i>	<i>Con métodos convencionales</i>
Configuración inicial	Requiere un mayor esfuerzo para aprender su API	Más rápido de implementar al utilizar herramientas conocidas
Mantenimiento	Centralizado en un solo lenguaje; facilita la reutilización y evita duplicación	Requiere mantener múltiples archivos, lo cual es propenso a inconsistencias
Reproducibilidad	Alta, gracias a la ejecución en entornos aislados	Menor, debido a las posibles diferencias entre entornos locales de desarrollo
Flexibilidad	Alta, permitiendo aplicar lógica de programación a todo el <i>pipeline</i>	Limitada por la necesidad de creación y adaptación de diferentes <i>scripts</i>
Caché	Optimizada de forma automática	Manual, más difícil de mantener y optimizar
Portabilidad	Elevada, todo se ejecuta en contenedores	Menor, requiere adaptar <i>scripts</i> o configuraciones según el entorno
Escalabilidad	Adecuado para cualquier proyecto, destacando en aquellos más grandes	Difícil de escalar conforme crece la lógica
Curva de aprendizaje	Moderada, API extensa pero intuitiva si se tiene experiencia previa con contenedores	Baja, herramientas ampliamente conocidas
Madurez del ecosistema	En crecimiento activo	Herramientas maduras con amplia documentación y soporte

Cuadro 5.1: Diferencias entre Dagger frente a métodos convencionales

Capítulo 6

Conclusións e posibles ampliacións

O traballo describe o grao de cumprimento dos obxectivos. Posibles vías de mellora.

Apéndice A

Manuales técnicos

Todo el código fuente se encuentra en la siguiente organización de GitHub: vieites-tfg. En este se puede encontrar tres repositorios y un registro de contenedores de GitHub, donde se publican las imágenes y los paquetes NPM de la aplicación.

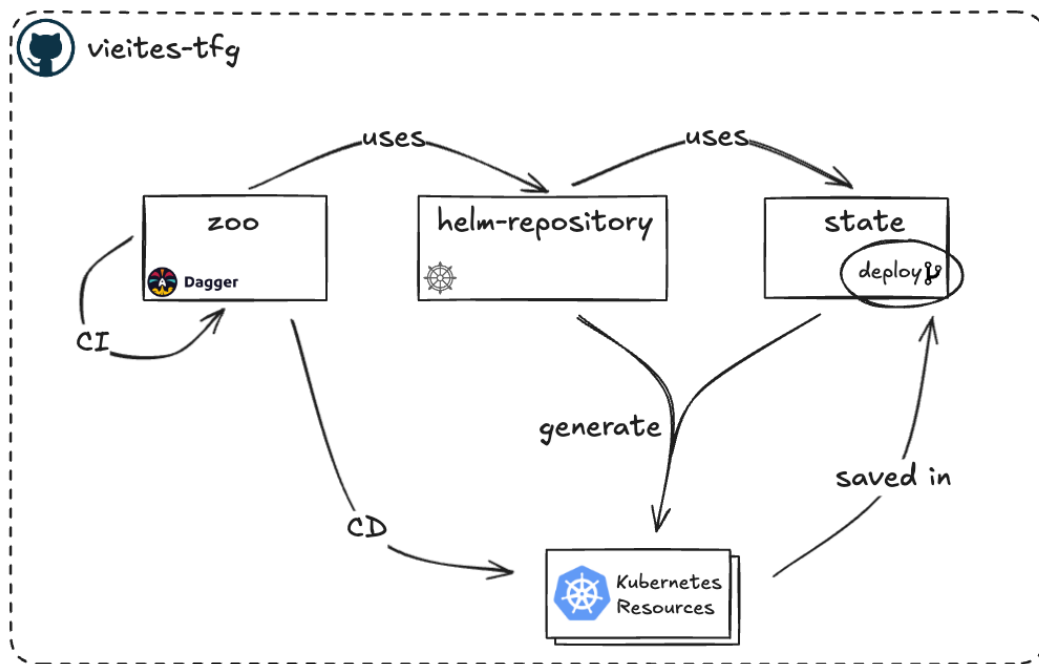


Figura A.1: Diagrama de la organización de GitHub. Imagen creada con excalidraw.com

A.1. Descripción de tecnologías

- Just.

Herramienta con la misma funcionalidad que Make, pero con más funcionalidades, entre las cuales destacan:

- Poder pasar parámetros a las “recetas” (las “reglas” en `make`).
- Crear alias para las recetas.
- Cargar archivos `.env`.
- Poder definir recetas como scripts en el lenguaje que se prefiera, simplemente añadiendo el *shebang*[15] correspondiente.
- Ser capaz de ser invocado desde cualquier subdirectorio.

Como se puede comprobar en el Listing A.4, el archivo de configuración de `just`, en este caso nombrado habitualmente `justfile`, tiene una estructura similar a la de un `Makefile`. La diferencia principal es que los nombres de las recetas no hacen referencia a un archivo objetivo que se supone que se debe crear al ejecutar el bloque de comandos, sino que se trata simplemente del nombre de la receta.

■ Docker.

Docker permite empaquetar aplicaciones, creando imágenes con las dependencias necesarias para que la aplicación se lance sin problemas. Las imágenes generadas se pueden ejecutar, creando contenedores, que son entornos completamente aislados del contexto de la máquina en la que han levantado.

El proceso más habitual a la hora de construir una imagen de Docker es definir un `Dockerfile` como el del Listing A.5. En este se indica, paso a paso, todo el proceso de instalación de dependencias y compilación del código fuente, necesario para lanzar la aplicación. En el `Dockerfile` mencionado, se puede observar que, además, se hace uso de *multi-stage builds*, distintos estados de la construcción. Esto permite construir imágenes de Docker más pequeñas y optimizadas separando el proceso de construcción en distintas fases.

```
1 | docker build -t my-image .
2 |
3 | docker run --rm -d -p 8080:80 my-image
4 |
```

Listing A.1: Construir y correr una imagen de Docker

El comando de la línea 1 construye la imagen que se define en el `Dockerfile` del directorio de trabajo actual (`.`), con el nombre `my-image`. Con el siguiente comando se ejecuta la imagen. Las *flags* indican:

- `--rm`

Se eliminará el contenedor creado al finalizar su ejecución.

- -d

El contenedor correrá en *background*.

- -p 8080:80

Se mapea el puerto 8080 de la máquina local al puerto 80 del contenedor.

■ Docker Compose.

Con Docker se es capaz de gestionar varios servicios desplegados en distintos contenedores. Pero existe una herramienta que apareció poco después y que facilita esta tarea, llamada “Docker Compose”[18]. Esta permite simular entornos con múltiples contenedores para desarrollar localmente (A.1).

En el archivo que se muestra en el Listing A.6, se puede observar cómo se configura el despliegue de tres servicios diferentes. Cada uno de los servicios se construye a partir de una imagen de Docker. Las imágenes correspondientes al frontend y al backend de la aplicación (*zoo-frontend* y *zoo-backend*, respectivamente) se generan y almacenan en un registro de GitHub al finalizar el ciclo de CI. Una vez publicadas, se pueden descargar indicando en el campo *image* el registro en el que están almacenadas junto con su nombre, como se puede ver en las líneas 4 y 16.

Simplemente, utilizando el comando del Listing A.2:

- Se levantan los tres servicios.
- Se les pasarán las variables de entorno indicadas.
- Se podrá acceder a ellos a través de los puertos establecidos, siendo el primer número el puerto local y el segundo el puerto del contenedor (*<local>:<contenedor>*).
- Se compartirán los volúmenes mencionados.

```
1 | docker compose up
2 |
```

Listing A.2: Despliegue con Docker Compose

Los valores de las variables de entorno, los indicados como *\${variable}*, se obtienen de un archivo *.env*, el cual debe estar presente en el mismo directorio que el archivo de configuración. En otro caso, es posible indicar la ruta al archivo mediante el campo *env_file*, dentro de cada uno de los servicios configurados.

■ Kubernetes.

Se trata de una herramienta de software que permite orquestar contenedores. Permite gestionar el ciclo de vida de las aplicaciones en contenedores que viven en un *cluster*.

En la figura A.4, se puede observar la estructura de un *cluster* de Kubernetes, el cual se compone de dos tipos principales de servidores (“nodos”):

- El *Control Plane*.
Toma todas las decisiones. Se encarga de que todo el sistema funcione como debe.
- Los nodos de trabajo.
Donde realmente se ejecutan las aplicaciones. Reciben órdenes del *Control Plane*. Puede y suele haber más de un nodo de trabajo en un *cluster*.

Kubernetes te permite definir diferentes elementos en archivos YAML. Estos archivos describen el estado que deseamos que tenga el sistema en todo momento. Kubernetes se encarga de procesar estos archivos e intentar hacer que el estado real del sistema sea igual al estado deseado.

Entre los elementos que se pueden construir se encuentran los siguientes:

- Pod.
Es la unidad más pequeña que se puede crear. Puede tener uno o más contenedores, pero lo normal es que tenga solo uno. Su función es encapsular y ejecutar la aplicación que le corresponda, que se indica mediante una imagen de Docker.
- Deployment.
Se trata de un controlador de Pods. Normalmente se utiliza este tipo de elementos en vez de crear Pods directamente. Esto es porque le puedes indicar la cantidad de Pods (réplicas) que deseas que haya en todo momento en el sistema, y el *deployment* lo hace por ti.
- Service.
Debido a que los Pods son efímeros, es decir, se crean y se destruyen constantemente, cambiando así su dirección IP; es necesario tener un elemento que funcione como punto fijo de acceso a un Pod. Para eso sirve un Service. Estos proporcionan un nombre y una IP únicos y fijos para los Pods.
- Ingress.
Es un elemento más avanzado que un Service. Permite gestionar el acceso desde Internet, dirigiendo las peticiones hacia los servicios correctos dentro del *cluster*.
- ConfigMap.
Este elemento está diseñado para almacenar valores no sensibles. Se definen en formato clave-valor. Permite separar la configuración de una aplicación de su código.

- Secret.

Es muy similar a un ConfigMap, pero está diseñado para almacenar y gestionar información sensible. Su función es guardar datos que no se deberían mostrar a simple vista en la configuración de una aplicación, como contraseñas o *tokens* de autenticación.

- ArgoCD.

Herramienta diseñada específicamente para Kubernetes. Automatiza el despliegue de las aplicaciones, supervisando repositorios de Git para aplicar los cambios que se realizan en ellos automáticamente en el *cluster*. Su uso permite no tener que realizar actualizaciones manuales en entornos de producción, y hace que el entorno siempre esté sincronizado con el código definido en el repositorio. Esto se consigue gracias al proceso de reconciliación que realiza Argo cada vez que detecta cambios en el repositorio que se le ha indicado.

A.2. Figuras

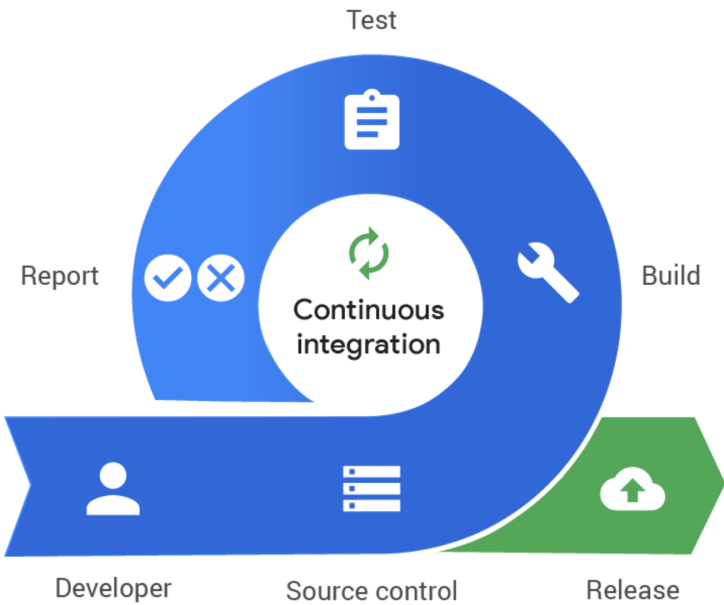


Figura A.2: Proceso de integración continua.[4]. Referenciado en 2.1.



Figura A.3: Proceso de despliegue continuo.[5]. Referenciado en 2.1.

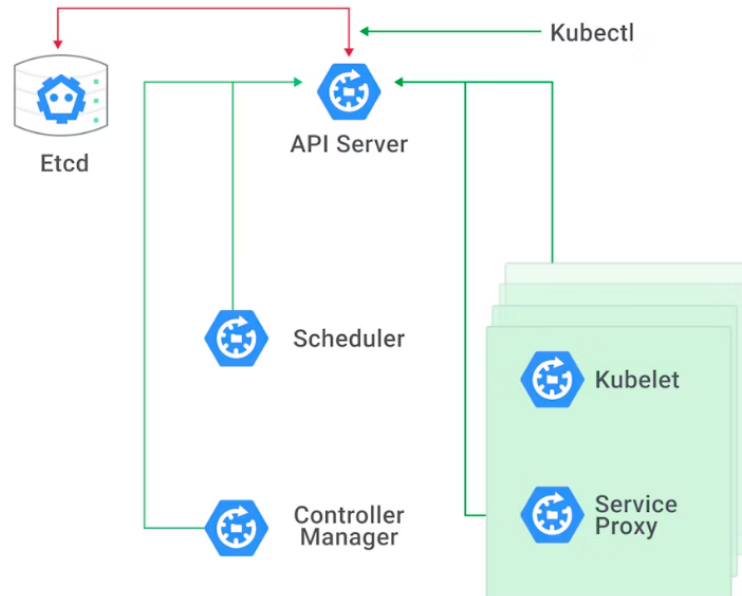


Figura A.4: Arquitectura de Kubernetes.[19]. Referenciado en 2.2.

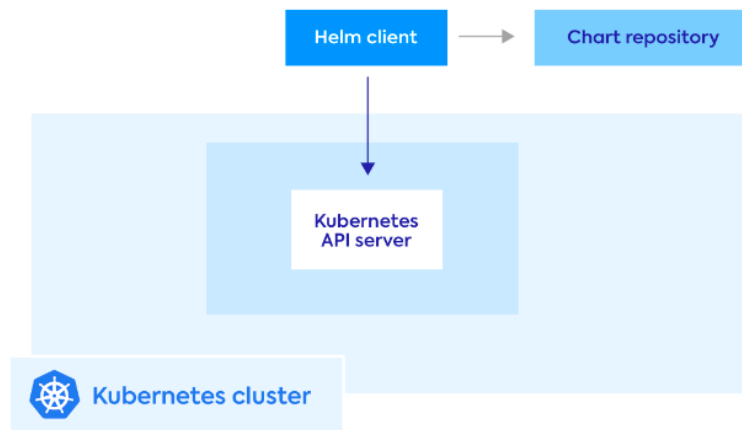


Figura A.5: Arquitectura de Helm.[25]. Referenciado en 2.2.

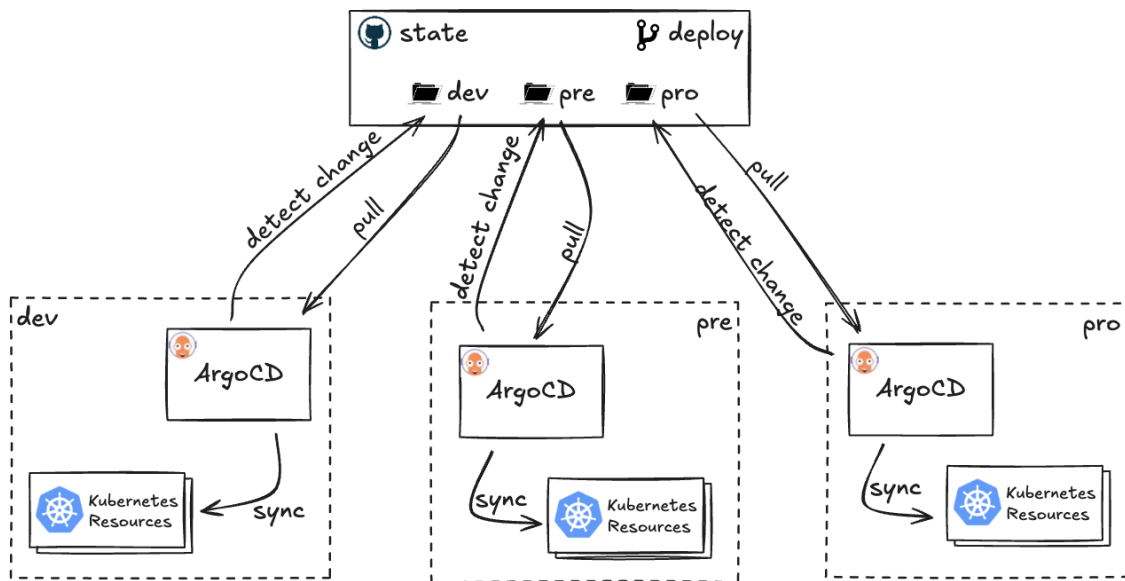


Figura A.6: Clusters y comunicación con el repositorio de estado. Imagen creada con excalidraw.com. Referenciado en 3.2.

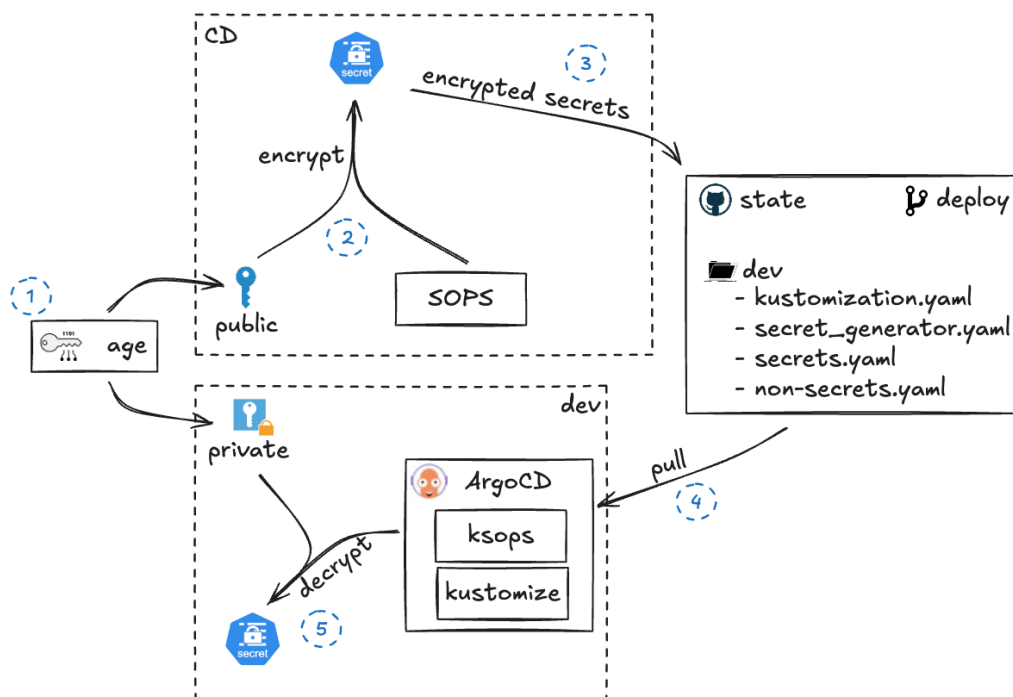


Figura A.7: Encriptado y desencriptado de secretos. Imagen creada con excalidraw.com. Referenciado en A.4, 3.2 y 3.4.

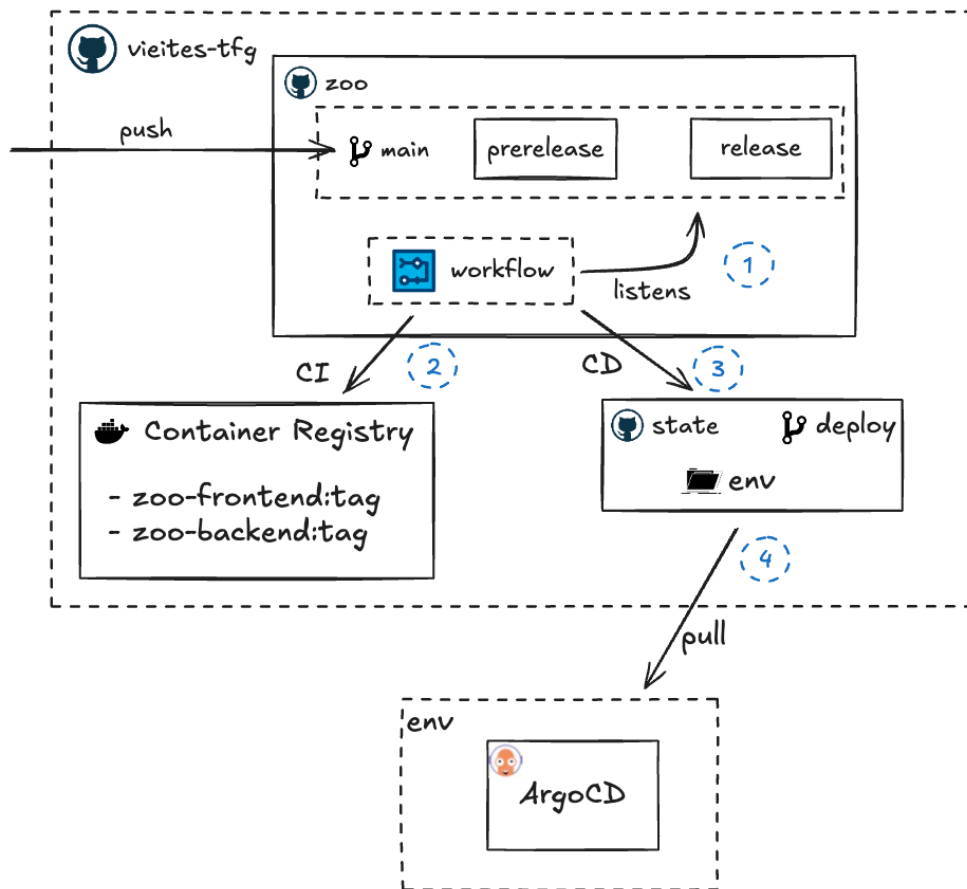


Figura A.8: Descripción del proceso de promoción de entornos. Imagen creada con excalidraw.com. Referenciado en 3.2 y A.5.

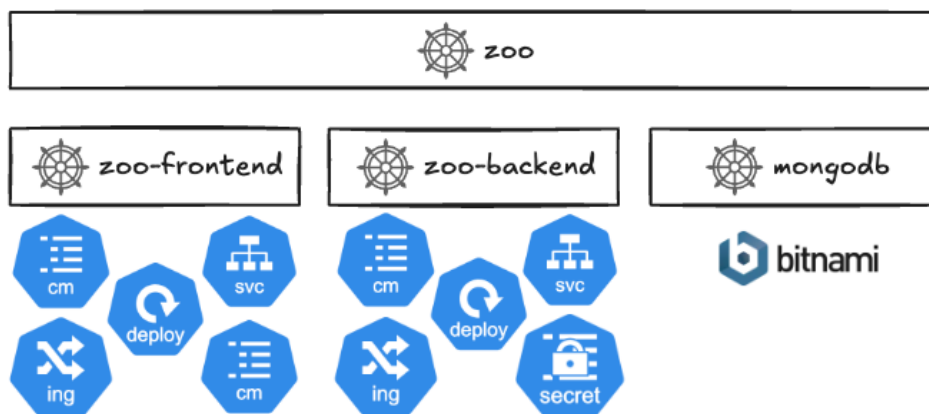


Figura A.9: Diagrama de organización de las Charts de la aplicación. Imagen creada con excalidraw.com. Referenciado en 3.3.

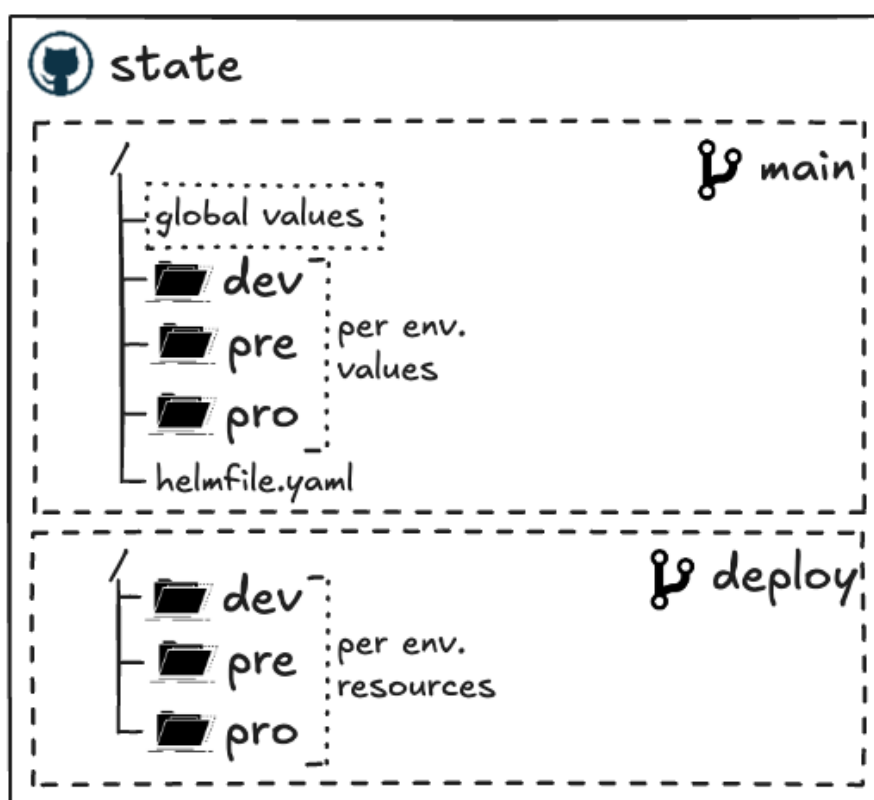


Figura A.10: Diagrama de la estructura del repositorio de estado. Imagen creada con excalidraw.com. Referenciado en 3.4

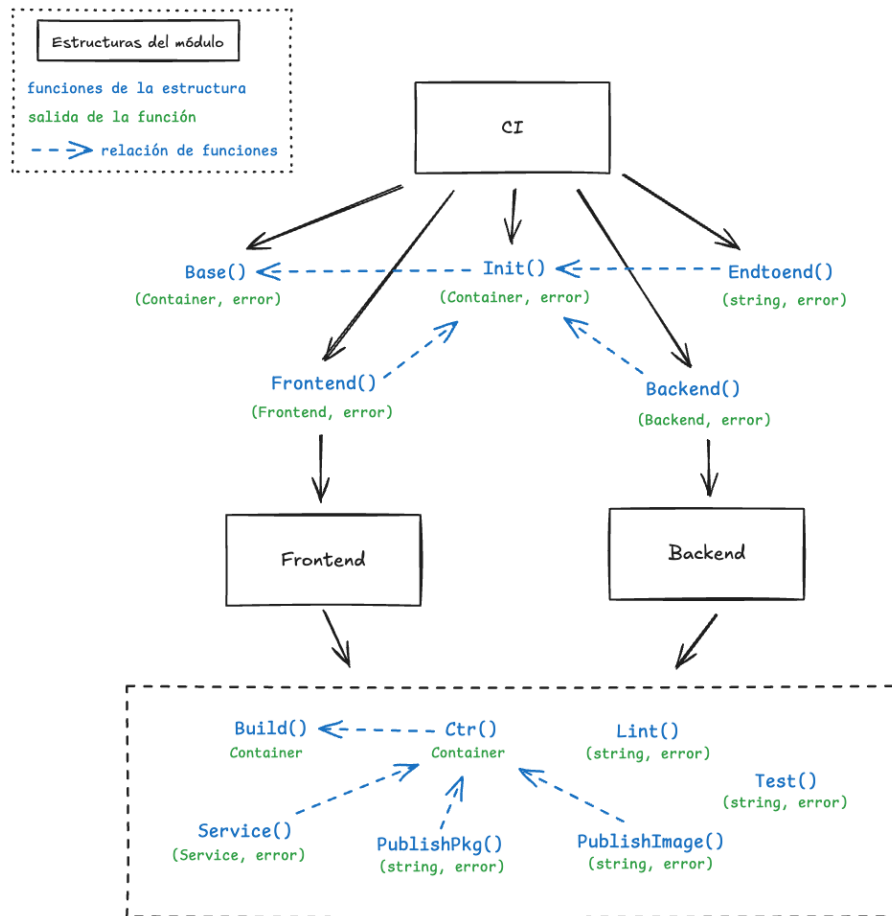


Figura A.11: Diagrama del diseño del módulo de CI con Dagger. Imagen creada con excalidraw.com. Referenciado en 4.1.

A.3. Listings

Referenciado en 2.2.

```

1 # Compiler
2 CC = gcc
3
4 # Compiler options
5 CFLAGS = -Wall -g
6
7 # Final executable name
8 TARGET = my_program
9
10 # The object files (.o) needed by the program
11 # Make infers automatically that .o depends on the corresponding
12 ↪ .c
13 OBJS = main.o hello.o
14
15 # --- Rules ---
16
17 # The first rule is the one executed by default with "make"
18 # It declares that to create the TARGET, it needs the OBJS
19 $(TARGET): $(OBJS)
20     $(CC) $(CFLAGS) -o $(TARGET) $(OBJS)
21
22 # ".PHONY" tells Make that "clean" is not a file
23 .PHONY: clean
24 clean:
25     rm -f $(TARGET) $(OBJS)

```

Listing A.3: Makefile para compilación de un programa en C

Referenciado en 2.2.

```

1 # --- ALIASES ---
2 # Defines shortcuts (aliases) for longer commands.
3 alias dv := down_vol
4
5 # --- DEFAULT RECIPE ---
6 # This is the recipe that runs if you just type 'just' in the
7 # terminal.
8 # By default, it invokes the 'just -l' recipe, which lists all
9 # available recipes.
10 # The '_' prefix indicates that it is a helper recipe, not
11 # intended to be called directly by the user.
12 _default:
13     just -l
14
15 # --- INTERNAL (PRIVATE) RECIPES ---
16 _build_zoo_base:
17     #!/usr/bin/env bash
18     if [[ "$(docker images -f reference=zoo-base | wc -l | xargs)"
19 ↪     != "2" ]]

```

```

19     then
20         docker build --target base -t zoo-base .
21     fi
22
23 # Accepts two parameters: 'entrypoint' and 'command'.
24 _run entrypoint command:
25     # '@' at the beginning of a command line prevents 'just' from
26     # printing the command before executing it.
27     @just _build_zoo_base
28     docker run --rm -w /app -v $PWD:/app --env-file .env --
29         ↪ entrypoint={{entrypoint}} zoo-base {{command}}
30
31 # --- PUBLIC RECIPES ---
32 init:
33     @just _run "yarn" "install"
34
35 down_vol:
36     docker compose down -v

```

Listing A.4: Extracto de justfile utilizado en el proyecto

Referenciado en 2.2.

```

1 # Base
2
3 FROM node:20 AS base
4
5 WORKDIR /app
6
7 COPY package.json lerna.json yarn.lock* ./
8
9 COPY packages packages/
10
11 RUN yarn install
12
13 RUN yarn global add lerna@8.2.1
14
15 RUN yarn global add @vercel/ncc
16
17 # Frontend build stage
18
19 FROM base AS frontend-build
20
21 WORKDIR /app
22
23 RUN lerna run --scope @vieites-tfg/zoo-frontend build
24
25 # Frontend
26
27 FROM nginx:alpine AS frontend
28
29 WORKDIR /usr/share/nginx/html
30

```

```

31 COPY --from=frontend-build /app/packages/frontend/dist .
32
33 EXPOSE 80
34
35 CMD ["nginx", "-g", "daemon off;"]

```

Listing A.5: Extracto de Dockerfile utilizado en el proyecto

Referenciado en 2.2.

```

1 services:
2   zoo-frontend:
3     image: ghcr.io/vieites-tfg/zoo-frontend
4     container_name: zoo-frontend
5     hostname: zoo-frontend
6     ports:
7       - "8080:80"
8     depends_on:
9       - zoo-backend
10    environment:
11      NODE_ENV: production
12      YARN_CACHE_FOLDER: .cache
13
14   zoo-backend:
15     image: ghcr.io/vieites-tfg/zoo-backend
16     container_name: zoo-backend
17     hostname: zoo-backend
18     ports:
19       - "3000:3000"
20     depends_on:
21       - mongodb
22     environment:
23       NODE_ENV: production
24       YARN_CACHE_FOLDER: .cache
25       MONGODB_URI: "mongodb://${MONGO_ROOT}:${MONGO_ROOT_PASS}
↪ @mongodb:${MONGO_PORT:-27017}/${MONGO_DATABASE}?authSource=
↪ admin"
26
27   mongodb:
28     image: mongo:7.0
29     container_name: zoo-mongo
30     hostname: mongodb
31     environment:
32       - MONGO_INITDB_DATABASE=${MONGO_DATABASE}
33       - MONGO_INITDB_ROOT_USERNAME=${MONGO_ROOT}
34       - MONGO_INITDB_ROOT_PASSWORD=${MONGO_ROOT_PASS}
35     ports:
36       - "${MONGO_PORT_HOST:-27017}:${MONGO_PORT:-27017}"
37     volumes:
38       - ./mongo-init:/docker-entrypoint-initdb.d/
39       - mongo_data:/data/db
40
41 volumes:

```

```
42 | mongo_data:
```

Listing A.6: docker-compose.yaml usado en el proyecto

Referenciado en 2.3.

```
1 | package main
2 |
3 | import (
4 |     "dagger.io/dagger"
5 |     "universe.dagger.io/docker"
6 | )
7 |
8 | dagger.#Plan & {
9 |     actions: {
10 |         pull: docker.#Pull & {
11 |             source: "alpine"
12 |         }
13 |         push: docker.#Push & {
14 |             image: pull.output
15 |             dest:  "localhost:5042/alpine"
16 |         }
17 |     }
18 | }
```

Listing A.7: Código de Dagger con CUE

Referenciado en 2.3

```
1 | package main
2 |
3 | import (
4 |     "context"
5 |     "dagger/dagger/internal/dagger"
6 |     "fmt"
7 |     "strconv"
8 | )
9 |
10 | type Backend struct {
11 |     Name string
12 |     Base *dagger.Container
13 |     Secrets SecMap
14 | }
15 |
16 | func (m *Backend) Build(ctx context.Context) *dagger.Container {
17 |     build := m.Base.
18 |         WithWorkdir("/app").
19 |         WithExec([]string{"lerna", "run", "--scope", "@vieites-tfg/
↪ zoo-backend", "build"}).
20 |         WithExec([]string{"ncc", "build", "./packages/backend/dist/
↪ index.js", "-o", "./dist"})
21 |
22 |     return build
23 | }
```



```

24
25 func (m *Backend) Ctr(ctx context.Context) *dagger.Container {
26     build := m.Build(ctx)
27
28     compiled := build.File("/app/dist/index.js")
29     pkgJson := build.File("/app/packages/backend/package.json")
30
31     dag := dagger.Connect()
32     back := dag.
33         Container().From("node:20-alpine").
34         WithExposedPort(3000).
35         WithEnvVariable("NODE_ENV", "production").
36         WithEnvVariable("YARN_CACHE_FOLDER", "/.yarn/cache").
37         WithMountedCache("/.yarn/cache", dag.CacheVolume("yarn-cache"
38         ↪)).
39         WithWorkdir("/app").
40         WithFile("/app/package.json", pkgJson).
41         WithFile("/app/index.js", compiled).
42         WithExec([]string{"yarn", "install", "--production"}).
43         WithEntrypoint([]string{"node", "index.js"})
44     return back
45 }

```

Listing A.8: Ejemplo del SDK de Go de Dagger.

Referenciado en 3.2.

```

1 # Variables globales
2 # ---
3
4 for ENV in "${ENVS[@]"; do
5     case "${ENV}" in
6         dev)
7             BANNER_TEXT="We are in DEV";;
8             # ... otros entornos
9     esac
10
11     CONTEXT="kind-${ENV}"
12
13     kind create cluster --config "${CLUSTER_DIR}/kind_${ENV}.yaml"
14
15     kubectl apply -f "${INGRESS_MANIFEST}" --context "${CONTEXT}"
16     # Se espera a que se construya
17
18     kubectl create namespace argocd || true
19
20     cat "${SOPS_DIR}/age.agekey" |
21     kubectl create secret generic sops-age -n argocd \
22     --context ${CONTEXT} --from-file=keys.txt=/dev/stdin
23
24     # Se descarga el repositorio de la Chart
25     helm install argocd argo/argo-cd -n argocd \

```

```
26     -f argo/values.yaml \
27     # ... otros *flags*
28
29     # Se espera a que se instale la Chart de Argo
30
31     kubectl apply -f "${ARGO_DIR}/argo_${ENV}.yaml" \
32     --context "${CONTEXT}"
33     # Se espera a que se apliquen los cambios
34
35     # Se aplica el banner a la interfaz de Argo
36
37     # Se obtiene la clave inicial del usuario "admin" y se muestra
38     # por pantalla
39     PASSWORDS+="${current_pass}"
40 done
41
42 printf "${PASSWORDS}"
```

Listing A.9: Script de creación de los clusters

Referenciado en 3.2.

```
1 kind: Cluster
2 apiVersion: kind.x-k8s.io/v1alpha4
3 name: dev
4 networking:
5   apiServerPort: 6443
6 nodes:
7 - role: control-plane
8   kubeadmConfigPatches:
9   - |
10     kind: InitConfiguration
11     nodeRegistration:
12       kubeletExtraArgs:
13         node-labels: "ingress-ready=true"
14   extraPortMappings:
15   - containerPort: 80
16     hostPort: 8080
17     protocol: TCP
18   - containerPort: 443
19     hostPort: 8443
20     protocol: TCP
```

Listing A.10: Configuración del cluster de dev

Referenciado en 3.2.

```
1 apiVersion: argoproj.io/v1alpha1
2 kind: Application
3 metadata:
4   name: app-dev
5   namespace: argocd
6 spec:
7   project: default
```

```

8   source:
9     repoURL: 'https://github.com/vieites-tfg/state.git'
10    path: dev
11    targetRevision: deploy
12  destination:
13    server: 'https://kubernetes.default.svc'
14    namespace: dev
15  syncPolicy:
16    automated:
17      prune: true
18      selfHeal: true
19    syncOptions:
20      - CreateNamespace=true

```

Listing A.11: Configuración de Argo en dev

Referenciado en 3.3

```

1  apiVersion: v2
2  name: zoo
3  description: Umbrella chart to deploy frontend, backend and mongo
4  version: 0.0.7
5  appVersion: "0.0.0"
6  dependencies:
7    - name: zoo-frontend
8      version: 0.0.0
9      repository: file://charts/zoo-frontend
10   - name: zoo-backend
11     version: 0.0.0
12     repository: file://charts/zoo-backend
13   - name: mongodb
14     repository: https://charts.bitnami.com/bitnami
15     version: 15.0.0
16   condition: mongo.internal.enabled

```

Listing A.12: Definición de la Chart umbrella de la aplicación

Referenciado en 3.4.

```

1  repositories:
2    - name: helm-repository
3      url: https://raw.githubusercontent.com/vieites-tfg/
4      ↪ helm-repository/gh-pages/
5  environments:
6    dev:
7      values:
8        - ./dev.yaml
9      # ... otros entornos
10
11  ---
12
13  releases:
14    - name: zoo-{{ .Environment.Name }}

```

```

15 namespace: {{ .Environment.Name }}
16 chart: helm-repository/zoo
17 version: {{ .Values.version }}
18 values:
19   - zoo-frontend.yaml
20   - zoo-backend.yaml
21   - mongodb.yaml
22   - global.yaml
23   - {{ .Environment.Name }}/zoo-frontend.yaml
24   - {{ .Environment.Name }}/zoo-backend.yaml
25   - {{ .Environment.Name }}/mongodb.yaml
26   - {{ .Environment.Name }}/global.yaml
27   - global:
28     ghcrSecret:
29       enabled: true
30       password: {{ requiredEnv "CR_PAT" | quote }}
31   - zoo-backend:
32     mongo:
33       root:
34         user: {{ requiredEnv "MONGO_ROOT" | quote }}
35         password: {{ requiredEnv "MONGO_ROOT_PASS" | quote
↪ }}
36   - mongodb:
37     auth:
38       rootUser: {{ requiredEnv "MONGO_ROOT" | quote }}
39       rootPassword: {{ requiredEnv "MONGO_ROOT_PASS" |
↪ quote }}

```

Listing A.13: Archivo de configuración de helmfile

Referenciado en 3.4.

```

1 zoo-backend:
2   image:
3     tag: "69ab8a1e"
4   mongo:
5     service:
6       name: "zoo-dev-mongodb"
7   ingress:
8     hostTemplate: "api-zoo-dev.example.com"

```

Listing A.14: Archivo de valores de zoo-backend en dev

A.4. Gestión de secretos

Para encriptar los secretos se utilizan dos herramientas:

- age[41].

age es una herramienta creada con Go que permite encriptar y desencriptar archivos. Una vez instalada, es necesario crear unas claves privada y pública. Estas se proporcionan en el manual de usuario



! CITAR MANUALES DE USUARIO

para poder probar la aplicación.

- SOPS[42].

SOPS (*Secrets OPerationS*) permite editar archivos encriptados, pero con la capacidad de realizar la encriptación de campos específicos de tipos de archivos como YAML o JSON. Esta herramienta tiene compatibilidad con **age**, por lo que es un recurso excelente para encriptar recursos de Kubernetes, ya que estos se definen mediante archivos en formato YAML.

El archivo de configuración de SOPS es como el que se muestra en el Listing A.15. En él se indican: el formato de los nombres de los archivos que tiene que encriptar (línea 2), los campos que *no* tiene que encriptar (línea 3) y la herramienta de encriptado junto con la clave pública que le sirve para realizar la encriptación (línea 4).

```

1 creation_rules:
2   - path_regex: ".*\\.ya?ml$"
3     unencrypted_regex: "^(apiVersion|metadata|kind|type)$"
4     age: age15peyc7 #...
```

Listing A.15: Archivo de configuración de SOPS

Con las claves pública y privada, y la configuración de SOPS, se es capaz de encriptar los Secrets de Kubernetes durante el ciclo de CD, en el módulo de Dagger. Los archivos encriptados se suben a la rama de despliegue del repositorio de estado, junto con los demás recursos que definen la aplicación.

Para desencriptar los secretos, ArgoCD necesita información: la clave privada creada con **age** y las herramientas necesarias para gestionar archivos encriptados con SOPS. La clave se le proporciona durante uno de los pasos de la creación de los *clusters* 3.2. Sin embargo, es necesario decirle a Argo cómo utilizarla.

Para ello es necesario el uso de otras dos herramientas:

- kustomize[43].

kustomize permite modificar valores de definiciones de recursos de Kubernetes sin necesidad de realizar cambios directamente en el archivo original, o bien crear recursos completamente nuevos a partir de otros. Las customizaciones se definen igual que cualquier otro recurso de Kubernetes, como se muestra en el Listing A.16. En dicho archivo, el cual se debe llamar *kustomization.yaml*, se indican: **resources**, que son los archivos que se van a incluir tal y como estén definidos; y **generators**, archivos que muestren cómo construir recursos a partir de otros. El generador que se utiliza en este trabajo se muestra en el Listing A.17. Estos archivos los lee ArgoCD,

los interpreta, y así sabe cómo comportarse y las herramientas que tiene que utilizar cuando encuentra los archivos que definen los recursos, como `secrets.yaml` y `non-secrets.yaml`.

```

1  apiVersion: kustomize.config.k8s.io/v1beta1
2  kind: Kustomization
3  resources:
4    - non-secrets.yaml
5  generators:
6    - secret-generator.yaml

```

Listing A.16: Archivo 'kustomization.yaml'

- `ksops`[44].

`ksops` (kustomize-SOPS) es un *plugin* de `kustomize` para gestionar recursos encriptados con SOPS. Se utiliza, sobre todo, para desencriptar Secrets o ConfigMaps de Kubernetes encriptados con SOPS. En el generador A.17 se ve cómo se le indica a ArgoCD que debe utilizar el *plugin* `ksops` para desencriptar el archivo con el nombre `secrets.yaml`.

```

1  apiVersion: viaduct.ai/v1
2  kind: ksops
3  metadata:
4    name: secret-generator
5    annotations:
6      config.kubernetes.io/function: |
7        exec:
8          path: ksops
9  files:
10   - secrets.yaml

```

Listing A.17: Generador de los secretos

Lo único que falta es instalar dentro de Argo estas herramientas, `kustomize` y `ksops`, e indicarle dónde se encuentra la clave privada que usará para desencriptar los secretos.

Esto se consigue con los valores que muestran en la línea 26 del Listing A.9. El archivo `values.yaml` tiene el contenido que se muestra en el Listing A.18. Con estos valores se consigue:

- Indicar las *flags* que tiene que utilizar Argo a la hora de ejecutar comandos con `kustomize` (líneas 1-4).
- Crear variables de entorno (líneas 6-11) que guardan información sobre (de arriba a abajo, respectivamente) el directorio raíz de archivos de configuración del sistema y la ruta en la que se puede encontrar la clave privada de `age`.
- Construir dos volúmenes (líneas 13-18), uno para almacenar los binarios de `kustomize` y `ksops`, y otro para la clave privada que se ha introducido en el *cluster* en las líneas 20-22 del Listing A.9.

- Instalar los binarios de `kustomize` y `ksops` en el volumen `custom-tools` creado previamente (líneas 20-21).
- Permitir a Argo acceder a los binarios y a la clave, montando los volúmenes que contienen estos elementos dentro del servidor principal de Argo (líneas 33-41).

```

1 configs:
2   cm:
3     kustomize.buildOptions: "--enable-alpha-plugins --enable-exec
  ↪ "
4     ui.bannerpermanent: "true"
5
6 repoServer:
7   env:
8     - name: XDG_CONFIG_HOME
9       value: /.config
10    - name: SOPS_AGE_KEY_FILE
11      value: /.config/sops/age/keys.txt
12
13 volumes:
14   - name: custom-tools
15     emptyDir: {}
16   - name: sops-age
17     secret:
18       secretName: sops-age
19
20 initContainers:
21   - name: install-ksops
22     image: viaductoss/ksops:v4
23     command: ["/bin/sh", "-c"]
24     args:
25       - echo "Installing KSOPS and Kustomize...";
26         mv ksops /custom-tools/;
27         mv kustomize /custom-tools/;
28         echo "Done.";
29     volumeMounts:
30       - mountPath: /custom-tools
31         name: custom-tools
32
33 volumeMounts:
34   - mountPath: /usr/local/bin/kustomize
35     name: custom-tools
36     subPath: kustomize
37   - mountPath: /usr/local/bin/ksops
38     name: custom-tools
39     subPath: ksops
40   - name: sops-age
41     mountPath: /.config/sops/age

```

Listing A.18: Valores que pueblan la Chart de ArgoCD

Los archivos: `kustomization.yaml`, `secret.generator.yaml`, `secrets.yaml` y `non-secrets.yaml`; todos ellos son los ficheros que se disponen en la rama de despliegue `deploy` del repositorio `state`. Por lo tanto, son los archivos que ArgoCD obtiene y utiliza para desplegar toda la aplicación en los distintos entornos.

En la figura A.7 se puede ver el ciclo completo de encriptado y desencriptado de secretos

A.5. Promoción de entornos

Referenciado en 3.2 y 3.4.

Estos son los pasos que se realizan en el *workflow* que realiza el despliegue al entorno correspondiente:

1. Se clonan los repositorios necesarios (líneas 12-15).
2. Se instala Dagger (líneas 17-18).
3. Se determina el entorno y la *tag* que se le pondrá a la imagen de Docker de los paquetes de la aplicación (*backend* y *frontend*) (líneas 20-29).
 - En `dev` la *tag* se trata de los ocho primeros caracteres del último *commit* que se ha realizado. De esta manera se sabe a ciencia cierta el código que conforma la aplicación en dicha imagen, y facilita la detección de errores. El entorno es `dev` siempre que el evento que haya disparado el *workflow* sea un *push* de la PR a la rama principal, en este caso `main`.
 - En `pre` se publica la imagen y los recursos siempre que el evento que lanza el *workflow* sea la creación de una *prerelease*. La *tag* que se utiliza es la que se le pone al nombre de la *prerelease*. Esta debería tener formato SemVer[45] con una coletilla “**snapshot**” (e.j. `1.2.3-snapshot`). Se utiliza esta coletilla con el fin de dar a entender que dicha imagen es una copia o “captura” de lo que sería la versión final de la imagen, la que se publicaría en el entorno de producción.
 - En `pro` se despliega la aplicación cuando el evento que lanza el *workflow* no es una *prerelease*, sino una *release*. Al igual que en `pre`, la *tag* sigue el formato SemVer, pero en este caso sin la coletilla que se usaba antes (e.j. `1.2.3`).
4. Se recrean los archivos necesarios con las variables almacenadas en GitHub (líneas 31-39). Se proporcionan estos archivos y datos en los manuales de usuario



Important!

! CITAR MANUALES DE USUARIO

para su testeo en local.

5. Se ejecuta el ciclo de CI para ambos paquetes de la aplicación y se publica cada una de las imágenes, indicando la *tag* que se ha determinado previamente (líneas 51-57). Además, es necesario actualizar los valores de las *tags* en el repositorio de estado, el cual tiene un campo específico para indicar este dato (líneas 45-49). Más información en 3.4.
6. Se ejecuta el ciclo de CD, aportando los parámetros necesarios, entre los que se encuentran los archivos que se han construido previamente (líneas 59-69).

Finalmente, la instancia de ArgoCD instalada en el entorno se sincroniza con el repositorio, obtiene los recursos de Kubernetes que se han almacenado en este y despliega la aplicación.

```

1 on:
2   push:
3     branches: [ main ]
4   release:
5     types: [ published ]
6   workflow_dispatch:
7
8 jobs:
9   cicd:
10    runs-on: ubuntu-24.04
11    steps:
12      - % Clona el repositorio "zoo" en la ruta "/zoo"
13
14      - % Clona el repositorio "state" en la ruta "/state"
15        % utilizando el token STATE_REPO
16
17      - name: Install Dagger
18        uses: dagger/dagger-for-github@8.0.0
19
20      - name: Determine environment
21        id: env_tag
22        run: |
23          % Determina el entorno en el que se despliega,
24          % teniendo en cuenta el *trigger* que ha lanzado
25          % el workflow:
26          %   - *push*      -> main
27          %   - *release* -> *release* o *pre-release*
28
29          echo "environment=${{envi}}" >> "$GITHUB_OUTPUT"
30          echo "tag=${{tag}}" >> "$GITHUB_OUTPUT"
31
32      - name: Recreate needed files
33        run: |
34          % Recrea el archivo .env para tenerlo disponible
35          % en "zoo"
36          echo "CR_PAT=${{ secrets.CR_PAT }}" >> ./env

```

```

37      % .... se incluyen todas las variables
38
39      % Almacena la clave privada de "age"
40      % Crea el archivo de configuracion de SOPS
41
42      - name: Run Dagger CI module
43      run: |
44          tag=${{ steps.env_tag.outputs.tag }}
45
46          update_state () {
47              % Actualiza el valor en "state" de la *tag* de
48              % la imagen para que se despliegue la que se
49              % acaba de publicar
50          }
51
52          dagger call --sec-env=file://.env backend \
53              publish-image --tag "${tag}"
54          update_state "zoo-backend" "${tag}"
55
56          dagger call --sec-env=file://.env frontend \
57              publish-image --tag "${tag}"
58          update_state "zoo-frontend" "${tag}"
59
60      - name: Run Dagger CD module
61      run: |
62          dagger call -m "./dagger/cd" \
63              --socket=/var/run/docker.sock \
64              --kind-svc=tcp://localhost:3000 \
65              --config-file=file://cluster/kind_local.yaml \
66              deploy \
67              --sec-env=file://.env \
68              --env=${{ steps.env_tag.outputs.environment }} \
69              --age-key=file://sops/age.agekey \
70              --sops-config=file://sops/.sops.yaml

```

Listing A.19: Workflow de CI/CD

En A.8 se muestra el ciclo de eventos que se realizan previa y posteriormente a la ejecución del *workflow*.

A.6. Desarrollo de la Chart de la aplicación

Referenciado en 3.3.

El proceso de desarrollo de la Chart comienza con la propia creación de esta. Tras realizar pruebas de construcción de la aplicación, se construye un archivo comprimido con el comando que se muestra en el Listing A.20. De esta manera se obtiene un archivo *.tgz* con toda la definición de la Chart de la aplicación. Posteriormente, se almacena el archivo comprimido en un directorio *temp*.

```
1 | helm package zoo
```

Listing A.20: Generación de un archivo comprimido de la Chart

Por último, se publica la Chart a través de GitHub Pages. Esto consiste gracias a un *workflow* que existe en el repositorio, el cual se ve en el Listing A.21 (se han comentado las acciones que se realizan en cada paso para reducir su tamaño). El *workflow* se lanza únicamente cuando existe un directorio `temp` con archivos en su interior (líneas 10-11). Este realiza los comandos necesarios para actualizar el archivo `index.yaml`, el cual define las diferentes versiones de la Chart y la URL pública donde se pueden descargar. Además, actualiza las ramas principal y `gh-pages`, siendo esta última en la que se puede encontrar el archivo índice ya mencionado.

```
1 | name: Release Helm Charts
2
3 | concurrency: release-helm
4
5 | on:
6 |   workflow_dispatch:
7 |     push:
8 |       branches:
9 |         - main
10 |      paths:
11 |        - 'temp/**'
12
13 | permissions:
14 |   contents: write
15
16 | jobs:
17 |   release:
18 |     runs-on: ubuntu-latest
19 |     steps:
20 |       - % Clona la rama principal del repositorio en "src/"
21 |
22 |       - % Clona la rama "gh-pages" del repositorio en "dest/"
23 |
24 |       - name: Install Helm
25 |         uses: azure/setup-helm@v4.3.0
26 |
27 |       - name: Update New Files and push to main branch
28 |         shell: bash
29 |         working-directory: src
30 |         run: |
31 |           % Se genera o actualiza el "index.yaml" con la URL
32 |           % donde se van a alojar las diferentes versione de la
33 |           % Chart
34 |
35 |           % Se guarda el .tgz del directorio "temp/" con la Chart
36 |           % en la rama de "gh-pages", el "index.yaml" en la raiz
37 |           % de la misma rama y se suben los cambios la rama
```

```
38         % principal
39         env:
40             GITHUB_TOKEN: ${ secrets.GITHUB_TOKEN }
41
42     - name: Push New Files
43       shell: bash
44       working-directory: dest
45       run: |
46           % Se guardan los cambios en la rama "gh-pages",
47           % realizando finalmente la publicacion de la version
48           % de la Chart
49       env:
50           GITHUB_TOKEN: ${ secrets.GITHUB_TOKEN }
```

Listing A.21: Workflow de publicación de la Chart en GH Pages

Apéndice B

Manuales de usuario

Este trabajo se compone de tres repositorios bien diferenciados y con su propia funcionalidad. Todos ellos se pueden encontrar en esta organización, `vieites-tfg`.

- **zoo**: Es el repositorio principal del trabajo. En él se encuentra todo el código fuente, desde la aplicación de prueba hasta la implementación de los módulos de Dagger. Es el único necesario para probar las diferentes funcionalidades.
- **helm-repository**: Alberga las Chart de Helm que describen el despliegue de la aplicación de prueba en Kubernetes.
- **state**: En él se almacenan los valores de las Charts de Helm correspondientes a cada uno de los entornos en los que se puede desplegar la aplicación. Funciona como única fuente de verdad. También es el lugar de donde ArgoCD obtiene los recursos de Kubernetes, con el fin de ser desplegados.

Cada uno de los repositorios tiene su propio **README**. Consultar los demás para más información.

B.1. zoo

Estructura general del repositorio

```
1 | argo
2 | |- argo_dev.yaml
3 | |- argo_pre.yaml
4 | |- argo_pro.yaml
5 | |- values.yaml
6 | cluster
7 | |- kind_dev.yaml
8 | |- kind_local.yaml
9 | |- kind_pre.yaml
10 | |- kind_pro.yaml
11 | dagger
```

```
12 |- cd
13 |- ci
14 docker-compose.yml
15 Dockerfile
16 example.env
17 mongo-init
18 |- init-zoo.js
19 packages
20 |- backend
21 |- frontend
22 scripts
23 |- create_envs.sh
24 |- image.sh
25 |- push_package.sh
26 sops
27 |- .sops.yaml
28 |- age.agekey
```

Listing B.1: Estructura principal del repositorio “zoo”

Este repositorio funciona como un monorepo, lo cual indica que todo el código fuente se encuentra en este único lugar.

Aplicación de prueba

Consta de un *frontend* y un *backend*, ambos creados con Typescript, y utilizando Vue para el *frontend*. Se gestiona utilizando una herramienta de gestión de *monorepos* llamada Lerna. En la raíz del repositorio se pueden encontrar archivos de configuración de Lerna y Node.js. El código correspondiente al *frontend* y al *backend* se incluyen en el directorio **packages**. El *backend* se conecta a una base de datos de animales de un zoo, y proporciona una API REST que el *frontend* consume, con el fin de poder realizar acciones CRUD sobre la base de datos.

Dagger

En el directorio **dagger** se encuentran las implementaciones de los módulos correspondientes a los flujos de CI y CD. Estos se pueden ejecutar localmente teniendo Dagger y Docker instalados. Se utilizan en el *workflow* de GitHub encargado de realizar todo el flujo de testeo, publicación y despliegue.

Otros

También se pueden encontrar varios ejecutables en el directorio de **scripts**, de los cuales **create_envs.sh** es el más interesante. Este permite levantar los *clusters* en local para probar el despliegue de la aplicación en los tres entornos posibles: **dev**, **pre** y **pro**.

En todos los *clusters* se instala ArgoCD, aplicación creada específicamente para Kubernetes y que utiliza el método *pull*, siguiendo la filosofía GitOps, leyendo

los recursos a desplegar del repositorio de estado (**state**) mencionado anteriormente. Este tiene una rama **deploy**, en la que se suben los archivos necesarios para que Argo lea y despliegue la aplicación.

Se configuran los *clusters* y las propias instancias de Argo de manera diferente para cada uno de los clusters. Estas configuraciones se pueden encontrar en los directorios **cluster** y **argo**.

Además, este trabajo tiene una memoria asociada, en la cual se pueden encontrar unas claves privada y pública, junto con otros *tokens*, necesarias para poder probar la implementación. Estas claves y *tokens* tienen varios propósitos:

- Claves privada y pública: Se utilizan para encriptar y desencriptar los secretos (**Secrets**) de Kubernetes.
- **tokens**: Un **CR_PAT**, que permite acceder al Container Registry de GitHub, donde se encuentran las imágenes de Docker de la aplicación; y un **STATE_REPO**, necesario para poder actualizar el repositorio de estado y actualizar los recursos que se van a desplegar.

Prueba mínima

Aquí se describen los requisitos de *software* y pasos a seguir para probar de la manera más simple la aplicación. Con esta prueba se comprobará que se pueden obtener los recursos de Kubernetes y levantar la aplicación en los diferentes entornos, pudiendo visualizar todos estos recursos en ArgoCD.

Los elementos que van a influir en esta prueba serán: los *clusters* junto con sus configuraciones y las de ArgoCD, y el repositorio de estado, en el cual ya debería haber recursos preparados para desplegar.

Requisitos software

A continuación, en el cuadro B.1 se indica el software junto con las versiones utilizadas para el desarrollo del proyecto.

Es necesario tener en cuenta que *no* se ha probado en un sistema operativo Windows, por lo que no se asegura su funcionamiento en este. Sí se ha probado en MacOS y distribuciones Linux.

```

1 # Se ha dividido en dos STATE_REPO, pero es un solo string
2 MONGO_DATABASE=zoo
3 MONGO_ROOT=carer
4 MONGO_ROOT_PASS=carerpass
5 CR_PAT=ghp_vEImTvw0ByxaS1FFvYSuNhaRGF2QZf0gARoA
6 STATE_REPO=github_pat_11A00YJPI0kFqDJhfeNXe0_mqIU7LFD5b3 # sigue
7 aPCjva40Jc1FoU4VazLelsjhuyCbsxTKGDLECGEMzuyySO

```

Listing B.2: Archivo de secretos .env

<i>Software</i>	<i>Version</i>
Git	2.48.1
Just	1.39.0
Docker	27.5.1
Kubectl	1.33
Kind	0.27.0
Helm	3.17.3

Cuadro B.1: Software y versiones utilizadas durante el desarrollo

```

1 # Se han dividido en dos STATE_REPO, age y SOPS_PRIVATE_KEY,
2 # pero son un solo string
3 MONGO_DATABASE: zoo
4 MONGO_ROOT: carer
5 MONGO_ROOT_PASS: carerpass
6 CR_PAT: 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
7 STATE_REPO: github_pat_11A00YJPI0kFqDJhfeNXe0_mqIU7LFD5b # sigue
8 3aPCjva40Jc1FoU4VazLelsjhuyyCbsxTKGDLECUGEMzuyySO
9 SOPS_CONFIG_FILE: |
10     creation_rules:
11         - path_regex: ".*\\\\\\.ya?ml$"
12           unencrypted_regex: "^(apiVersion|metadata|kind|type)$"
13           age: age15peyc7pedj8gjwnarat6s3u87wy4j5xtf7t96v # sigue
14           uj74m3l9xq5ys0r4sag
15 SOPS_PRIVATE_KEY: AGE-SECRET-KEY-1CTS4S4QNNZ9N9YXXM288LS # sigue
16 E9VKPJ220E57ZHC4558WMZ8LG2QWKQFFER8C

```

Listing B.3: Archivo de secretos .secrets.yaml, para pruebas con act

Bibliografía

- [1] Dagger.io. “Dagger Documentation — Dagger.” Dagger.io, 2022, <https://docs.dagger.io>
- [2] Dagger.io. “Dagger — Blog.” Dagger.io, 2025, <https://dagger.io/blog/>. Accedido el 15 de junio de 2025
- [3] Fowler, Martin. “Continuous Integration.” Martinfowler.com, 18 Jan. 2024, <https://martinfowler.com/articles/continuousIntegration.html>.
- [4] PagerDuty, Inc. “What Is Continuous Integration?” PagerDuty, 20 Nov. 2020, <https://www.pagerduty.com/resources/devops/learn/what-is-continuous-integration>.
- [5] Amazon Web Services, Inc. “¿Qué Es La Entrega Continua? – Amazon Web Services.” Amazon Web Services, Inc., 2024, <https://aws.amazon.com/es/devops/continuous-delivery>.
- [6] Fowler, Martin. “Bliki: ContinuousDelivery.” Martinfowler.com, 2013, <https://martinfowler.com/bliki/ContinuousDelivery.html>.
- [7] Nx. “Monorepo Explained.” Monorepo.tools, 2025, <https://monorepo.tools>
- [8] Los autores de Kubernetes. “Orquestación de Contenedores Para Producción.” Kubernetes, 2025, <https://kubernetes.io/es>.
- [9] Helm. “Helm.” Helm.sh, 2019, <https://helm.sh>.
- [10] Atlassian. “¿Qué Es DevOps?” Atlassian, <https://www.atlassian.com/es/devops>.
- [11] Nasser, Mohammed. “Push vs. Pull-Based Deployments.” DEV Community, 25 Nov. 2024, <https://dev.to/mohamednasser018/push-vs-pull-based-deployments-4m78>. Accedido el 14 de junio de 2025.
- [12] Git. “Git.” Git-Scm.com, 2024, <https://git-scm.com>.
- [13] Wikipedia Contributors. “Make (Software).” Wikipedia, 10 July 2021, en [https://en.wikipedia.org/wiki/Make_\(software\)](https://en.wikipedia.org/wiki/Make_(software)).

- [14] casey. “GitHub - Casey/Just: Just a Command Runner.” GitHub, 2025, <https://github.com/casey/just>. Accedido el 14 de junio de 2025.
- [15] Wikipedia Contributors. “Shebang (Unix).” Wikipedia, 13 Aug. 2021, [https://en.wikipedia.org/wiki/Shebang_\(Unix\)](https://en.wikipedia.org/wiki/Shebang_(Unix)).
- [16] Wikipedia Contributors. “DevOps.” Wikipedia, Wikimedia, 1 Dec. 2019, <https://en.wikipedia.org/wiki/DevOps>.
- [17] Wikipedia Contributors. “Docker (Software).” Wikipedia, 16 Nov. 2019, [https://en.wikipedia.org/wiki/Docker_\(software\)](https://en.wikipedia.org/wiki/Docker_(software)).
- [18] “Overview of Docker Compose.” Docker Documentation, 10 Feb. 2020, <https://docs.docker.com/compose>.
- [19] Kong. “What Is Kubernetes? Examples and Use Cases.” Kong Inc., 2024, <https://konghq.com/blog/learning-center/what-is-kubernetes>.
- [20] The Kubernetes Authors. “Cluster Architecture.” Kubernetes, 2025, <https://kubernetes.io/docs/concepts/architecture>.
- [21] The Kubernetes Authors. “Kind.” Kind.sigs.k8s.io, 2025, <https://kind.sigs.k8s.io>.
- [22] The Kubernetes Authors. “Command Line Tool (Kubectl).” Kubernetes, 2025, <https://kubernetes.io/docs/reference/kubectl>.
- [23] GitLab. “¿Qué Es GitOps?” Gitlab.com, GitLab, 9 Feb. 2022, <https://about.gitlab.com/es/topics/gitops>. Accedido el 20 de junio de 2025.
- [24] Argo Project Authors. “Argo CD.” Github.io, 2025, <https://argoproj.github.io/cd>. Accedido el 20 de junio de 2025.
- [25] Flexera. “Kubernetes Helm: K8s Application Deployment Made Simple.” Spot.io, 12 Sept. 2024, <https://spot.io/resources/kubernetes-architecture/kubernetes-helm-k8s-application-deployment-made-simple>. Accedido el 20 de junio de 2025.
- [26] Dagger. “Introducing Dagger: A New Way to Create CI/CD Pipelines — Dagger.” Dagger.io, 2022, <https://dagger.io/blog/public-launch-announcement>. Accedido el 21 de junio de 2025.
- [27] Wikipedia Contributors. “Open Container Initiative.” Wikipedia, Wikimedia Foundation, 12 Nov. 2024, https://en.wikipedia.org/wiki/Open_Container_Initiative.

- [28] CUE. “The CUE Language Specification.” CUE, 16 Apr. 2025, <https://cuelang.org/docs/reference/spec>. Accedido el 21 de junio de 2025.
- [29] Wikipedia Contributors. “Conjunto de Herramientas de Desarrollo de Software.” Wikipedia, 15 Aug. 2006, https://es.wikipedia.org/wiki/Kit_de_desarrollo_de_software. Accedido el 21 de junio de 2025.
- [30] GitHub. “Features · GitHub Actions.” GitHub, 2025, <https://github.com/features/actions>.
- [31] “The Go Programming Language.” Go.dev, 2025, <https://go.dev>. Accedido el 21 junio de 2025.
- [32] The GraphQL Foundation. “GraphQL: A Query Language for APIs.” GraphQL.org, 2012, <https://graphql.org>.
- [33] Dagger. “Introducing the Dagger GraphQL API — Dagger.” Dagger.io, 2022, <https://dagger.io/blog/graphql>. Accedido el 21 de junio de 2025.
- [34] Dagger. “Daggriverse.” Daggriverse.dev, 2025, <https://daggriverse.dev>. Accedido el 21 de junio de 2025.
- [35] Dagger. “ContentKeeper Content Filtering.” Dagger.io, 2025, <https://docs.dagger.io/reference/cli>. Accedido el 21 de junio de 2025.
- [36] Microsoft. “TypeScript - JavaScript That Scales.” Typescriptlang.org, 2025, <https://www.typescriptlang.org>.
- [37] Nx. “Lerna · a Tool for Managing JavaScript Projects with Multiple Packages.” Lerna.js.org, 2025, <https://lerna.js.org>.
- [38] MongoDB. “MongoDB.” MongoDB, 2024, <https://www.mongodb.com>.
- [39] You, Evan. “Vue.js.” Vuejs.org, 2014, <https://vuejs.org>.
- [40] LinearB, Inc. “Cycle Time Breakdown: Tactics for Reducing PR Review Time — LinearB Blog.” Linearb.io, 4 Aug. 2021, <https://linearb.io/blog/reducing-pr-review-time>. Accedido el 10 de julio de 2025.
- [41] Valsorda, Filippo. “FiloSottile/Age.” GitHub, 21 Apr. 2022, <https://github.com/FiloSottile/age>.
- [42] getsops. “Getsops/Sops.” GitHub, 9 Jan. 2024, <https://github.com/getsops/sops>.

- [43] Kubernetes SIGs. “Kustomize - Kubernetes Native Configuration Management.” Kustomize.io, 2025, <https://kustomize.io>.
- [44] viaduct-ai. “GitHub - Viaduct-Ai/Kustomize-Sops: KSOPS - a Flexible Kustomize Plugin for SOPS Encrypted Resources.” GitHub, 28 Jan. 2025, <https://github.com/viaduct-ai/kustomize-sops>. Accedido el 11 de julio de 2025.
- [45] Preston-Werner, Tom. “Versionado Semántico 2.0.0.” Semantic Versioning, 2024, <https://semver.org/lang/es>.
- [46] Broadcom. “Applications - Bitnami.com.” Bitnami.com, 2024, <https://bitnami.com/stacks?stack=helm>. Accedido el 12 de julio de 2025.
- [47] Helmfile Authors. “Helmfile.” Readthedocs.io, 2025, <https://helmfile.readthedocs.io/en/latest>. Accedido el 12 de julio de 2025.
- [48] Cypress. “JavaScript End to End Testing Framework.” JavaScript End to End Testing Framework — Cypress.io, 2025, <https://www.cypress.io>.
- [49] Dagger. “Custom Types — Dagger.” Dagger.io, 2022, <https://docs.dagger.io/api/custom-types>. Accedido el 13 de julio de 2025.