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

Este trabajo aborda la gestión completa de un ciclo CI/CD (*Continuous Integration/Continuous Delivery*) con Dagger. Se desarrolla una aplicación de prueba, que consta de un frontend y un backend, junto con la infraestructura como código, y se comparan dos *pipelines*: uno implementado con Dagger y otro sin él. Los resultados demuestran que Dagger mejora el flujo de trabajo debido a la gestión que realiza de la caché, ejecutando todo el ciclo de CI/CD un 80 % más rápido que el *pipeline* sin Dagger. Como resultado, este trabajo propone un conjunto de módulos de Dagger que ofrece un enfoque práctico sobre cómo utilizar Dagger para acelerar el desarrollo despliegue de cualquier aplicación, minimizando el tiempo de espera y pasos manuales.

Índice general

Índice de figuras

Índice de cuadros

Índice de Listings

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 lunguaje conocido para el desarrollador. En vez de coordinar *scripts* creados a mano en diferentes entornos, el programador es capaz de componer acciones reusbles, 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 desafios, 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.

  Todo el código principal se almacenará en un mismo repositorio. De esta manera se evitarán complicaciones debido a la gestión de dependencias de cada una de las partes de la aplicación. Permitirá controlar todo el código fuente de una manera más sencilla al tratarse de un proyecto relativamente pequeño.

- Diseño y creación de una aplicación *dummy*.

  Es necesario una aplicación sobre la que realizar los ciclos de CI/CD. Esta consistirá en una página web (frontend) de gestión de un zoo, la cual realizará peticiones a una API REST (backend) que estará conectada a una base de datos.

- Implementación de un *pipeline* CI/CD.

  Se creará un módulo de Dagger para cada uno de los procesos (CI y CD). El módulo de CI permitirá desde compilar la aplicación, hasta publicar las imágenes de Docker y los paquetes NPM tanto del frontend como del backend. Por otro lado, el módulo de CD será el encargado de utilizar esas imágenes que se han publicado previamente y desplegar la aplicación en el entorno correspondiente.

- Entorno orquestado.

  El ciclo de CD termina en un entorno orquestado por Kubernetes[8], configurado a través de una *chart* de Helm[9]. La fase final del *pipeline* de Dagger no despliega la aplicación, sino que actualiza la configuración en un repositorio de estado. El despliegue sigue un modelo GitOps[23] gestionado por ArgoCD[24], el cual se encarga de detectar los cambios en el repositorio de estado y actualizar los *clusters* en consecuencia.

- Análisis comparativo

  Finalmente, se realiza un análisis 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, un 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.

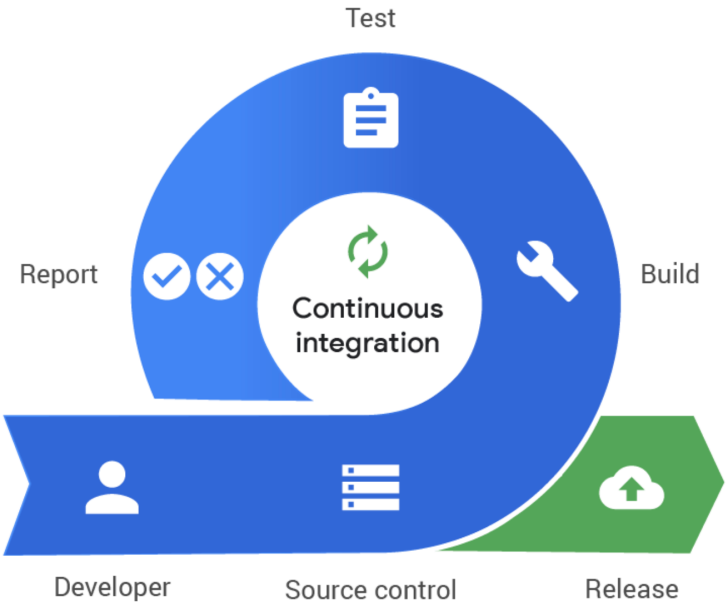

Figura 2.1: Proceso de integración continua.[4]

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. 2.1).

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 intregrarlo 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.

Figura 2.2: Proceso de despliegue continuo.[5]

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".

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. 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. Argo es una herramienta que, como su nombre indica, facilida la entrega continua. Está diseñada específicamente para Kubernetes. Automatiza el despliegue de las aplicaciones, superevisando 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.

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.

```
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
       ↪ .c
12 OBJS = main.o hello.o
13
14 # --- Rules ---
15
16 # The first rule is the one executed by default with "make"
17 # It declares that to create the TARGET, it needs the OBJS
18 $(TARGET): $(OBJS)
19   $(CC) $(CFLAGS) -o $(TARGET) $(OBJS)
20
21 # ".PHONY" tells Make that "clean" is not a file
22 .PHONY: clean
23 clean:
24   rm -f $(TARGET) $(OBJS)
```

Listing 2.1: Makefile para compilación de un programa en C

Un ejemplo muy típico de compilación de un programa escrito en `C` sería el que se puede observar en el Listing 2.1.

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, entre las cuales destacan:

- Poder pasar parámetros a las "recetas" (las "reglas" en `make`).
- Crear aliases 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.

```
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
```

```
# available recipes.
# The '_' prefix indicates that it is a helper recipe, not
# intended to be called directly by the user.
_default:
  just -l

# --- INTERNAL (PRIVATE) RECIPES ---
_build_zoo_base:
  #!/usr/bin/env bash
  if [[ "$(docker images -f reference=zoo-base | wc -l | xargs)"
   ↪ != "2" ]]
  then
    docker build --target base -t zoo-base .
  fi

# Accepts two parameters: 'entrypoint' and 'command'.
_run entrypoint command:
  # '@' at the beginning of a command line prevents 'just' from
  # printing the command before executing it.
  @just _build_zoo_base
  docker run --rm -w /app -v $PWD:/app --env-file .env --
   ↪ entrypoint={{entrypoint}} zoo-base {{command}}

# --- PUBLIC RECIPES ---
init:
  @just _run "yarn" "install"

down_vol:
  docker compose down -v
```

Listing 2.2: Extracto de justfile utilizado en el proyecto

Como se puede comprobar en el Listing 2.2, 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. Como se puede comprobar, 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.

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".

```
# Base

FROM node:20 AS base

WORKDIR /app

COPY package.json lerna.json yarn.lock* ./

COPY packages packages/

RUN yarn install

RUN yarn global add lerna@8.2.1

RUN yarn global add @vercel/ncc

# Frontend build stage

FROM base AS frontend-build

WORKDIR /app

RUN lerna run --scope @vieites-tfg/zoo-frontend build

# Frontend

FROM nginx:alpine AS frontend

WORKDIR /usr/share/nginx/html

COPY --from=frontend-build /app/packages/frontend/dist .

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

Listing 2.3: Extracto de Dockerfile utilizado en el proyecto

El proceso más habitual a la hora de construir una imagen de Docker es definir un `Dockerfile` como el del Listing 2.3. 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.

Una vez definido el `Dockerfile`, se puede construir y ejecutar la imagen con la secuencia de comandos del Listing 2.4

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

Listing 2.4: 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.

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 facilita esta tarea, llamada "Docker Compose"[18]. Esta permite simular entornos con múltiples contenedores para desarrollar localmente.

```
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 2.5: docker–compose.yaml usado en el proyecto

En el archivo que se muestra en el Listing 2.5, 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 2.6:

- 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
```

Listing 2.6: 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 directo 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.

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 Kubernetes2.2.

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.

Un *cluster* de Kubernetes 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.

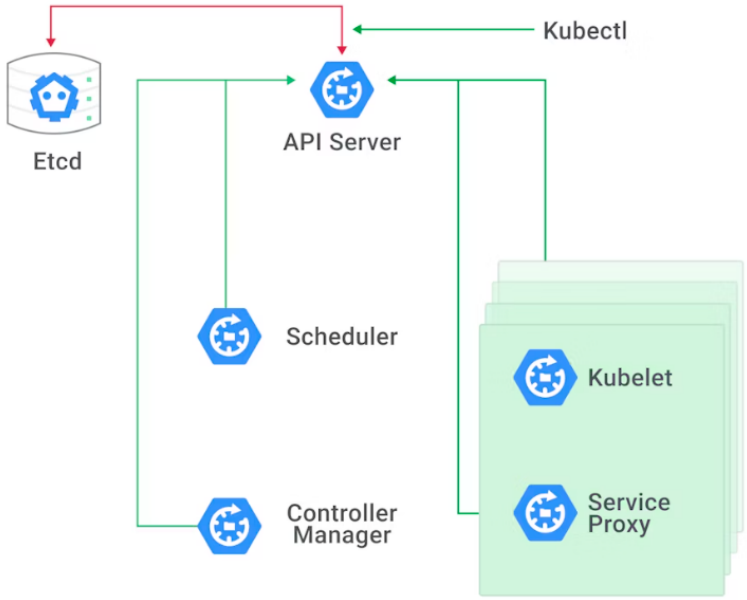

Figura 2.3: Arquitectura de Kubernetes.[19]

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*.

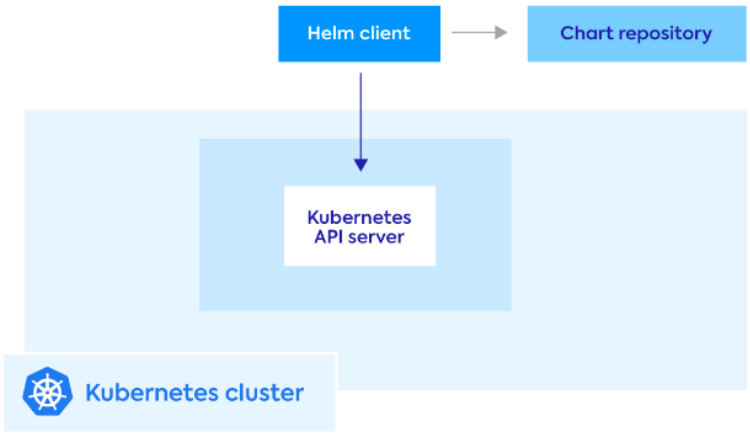

Figura 2.4: Arquitectura de Helm.[25]

- 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.

Complementando a Kubernetes tenemos Helm, 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.

```
1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    name: {{ .Release.Name }}-{{ .Chart.Name }}-ingress
5  spec:
6    rules:
7      - host: {{ tpl .Values.ingress.hostTemplate . }}
8        http:
9          paths:
10           - path: /
11             pathType: Prefix
12             backend:
13               service:
14                 name: {{ .Release.Name }}-{{ .Chart.Name }}-svc
15                 port:
```

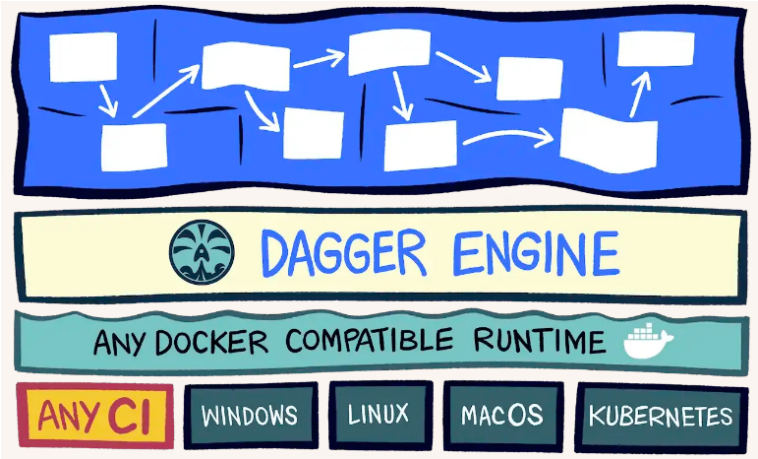

Figura 2.5: *pipelines* con Dagger sobre un *runtime* compatible con Docker.[26]

```
16                    number: {{ .Values.service.port }}
```

Listing 2.7: Ejemplo de objeto Ingress utilizado en este proyecto.

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 2.8 se puede construir un *cluster* en el que desplegar cualquier aplicación de Kubernetes.

```
1 kind create cluster
```

Listing 2.8: 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 2.9

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

Listing 2.9: Aplicar la configuración de ArgoCD con kubectl

El uso de estas dos herramientas se ve más en detalle en la sección 3.1.

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 2.10 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.

```
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 2.10: Código de Dagger con CUE

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 serda 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 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 2.11 se puede observar un ejemplo de código[33] utilizando el SDK de Go.

```
1  package main
2
3  import (
4    "context"
5    "fmt"
6    "os"
7
8    "dagger.io/dagger"
9  )
10
11 func main() {
12   err := doCi()
13   if err != nil {
```

```
14      fmt.Println(err)
15    }
16  }
17
18  func doCi() error {
19    ctx := context.Background()
20
21    // create a Dagger client
22    client, err := dagger.Connect(ctx, dagger.WithLogOutput(os.
        ↪ Stdout))
23    if err != nil {
24      return err
25    }
26    defer client.Close()
27
28    src := client.Host().Directory(".") // get the projects source
        ↪ directory
29
30    yarn := client.Container().From("yarnpkg/node-yarn"). //
        ↪ initialize new container from yarn image
31    WithDirectory("/src", src).WithWorkdir("/src"). // mount source
        ↪  directory to /src
32    WithExec([]string{"yarn", "test"}) // execute yarn test command
33
34    // get test output
35    test, err := yarn.Stdout(ctx)
36    if err != nil {
37      return err
38    }
39    // print output to console
40    fmt.Println(test)
41
42    // execute build command and get build output
43    build, err := yarn.WithExec([]string{"yarn", "build"}).Stdout(
        ↪ ctx)
44    if err != nil {
45      return err
46    }
47    // print output to console
48    fmt.Println(build)
49
50    return nil
51  }
```

Listing 2.11: Primeros ejemplos del SDK de Go de Dagger.

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.

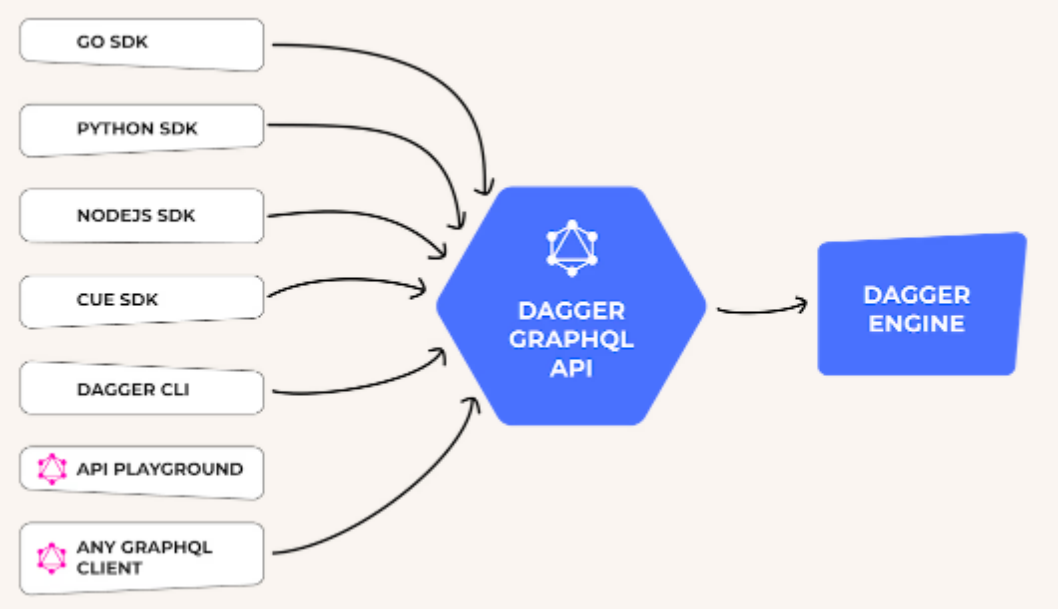

Figura 2.6: Uso de la API de GraphQL para Dagger.[34]

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 2.6. 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.

```
1 // Builds the frontend package, generating only one executable
```

```
// file and returns the container.
func (m *Frontend) Build(ctx context.Context) *dagger.Container {
  build := m.Base.
    WithWorkdir("/app").
    WithExec([]string{"lerna", "run", "--scope", "@vieites-tfg/
    ↪ zoo-frontend", "build"})

  return build
}

// Based on the build stage, gets the executable file and creates
// a ready to run container with Nginx and the port 80 exported.
func (m *Frontend) Ctr(ctx context.Context) *dagger.Container {
  build := m.Build(ctx)

  dist := build.Directory("/app/packages/frontend/dist")

  ctr := dag.
    Container().
    From("nginx:alpine").
    WithWorkdir("/usr/share/nginx/html").
    WithDirectory(".", dist).
    WithExposedPort(80).
    WithEntrypoint([]string{"nginx", "-g", "daemon off;"})

  return ctr
}
```

Listing 2.12: Extracto de funciones de Dagger de este trabajo

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[35]. Este es el lugar en el que se comparten módulos de Dagger, los cuales se pueden utilizar en el caso de que alguno sea necesario para el *pipeline* que se quiera crear.

CLI

Otra funcionalidad que tiene Dagger es su CLI[36] (*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.

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

Listing 2.13: Comando para lanzar el backend del proyecto

De esta manera (Listing 2.13) 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

Important!

CITAR APÉNDICE

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 Daggerverse, 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 para la realización del trabajo. 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 en el que se almacenan los valores que poblarán los recursos de Kubernetes, dependiendo del entorno en los 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 3.1 se muestra un diagrama de la disposición de los repositorios y la relación entre ellos.

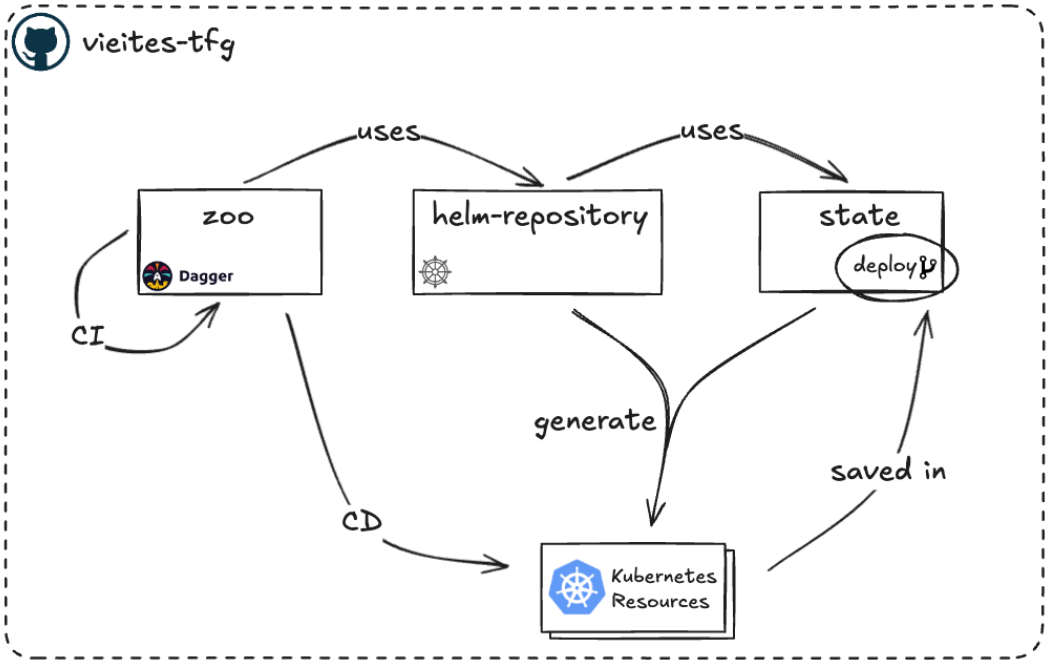

Figura 3.1: Diagrama del la organización de GitHub. Imagen creada con excalidraw.com

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 viven juntos 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[37]. 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[38]. 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.

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[39] 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[40], 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 apendizaje. Además, el propio *framework* está construido utilizando Typescript, por lo que tiene compatibilidad de primera clase con este lenguaje.

En la figura 3.2 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, tiene sentido que vivan

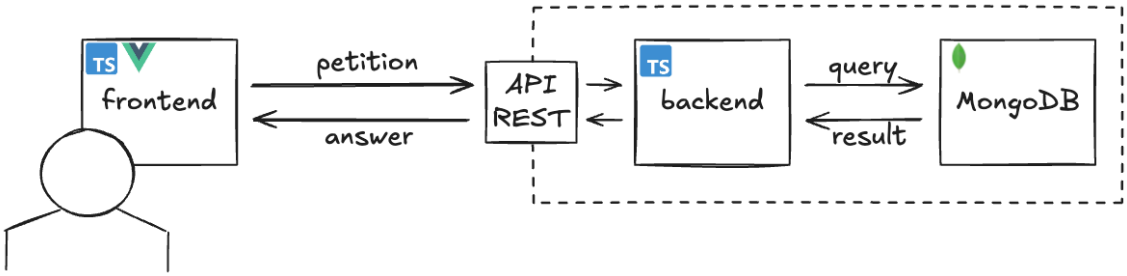

Figura 3.2: Comunicación entre paquetes de la aplicación. Imagen creada con excalidraw.com

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 3.1 (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. Estos son los pasos que se siguen para crear cada uno de los *clusters*:

```
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 3.1: Script de creación de los clusters

1. Se indica el banner que va a tener cada uno de los *clusters* (líneas 5-9). El banner se muestra en la parte superior de la interfaz de Argo.

2. Se indica el contexto actual (línea 11), lo cual identifica cada *cluster*. Esto es necesario a la hora de ejecutar comandos con `kubectl`, para asegurarse de que se está ejecutando cada comando sobre el *cluster* que se precisa.

3. Se crea el *cluster* con su configuración específica (línea 13). Se puede ver el archivo de configuración del *cluster* de `dev` en el Listing 3.2. Los archivos de los demás entornos son iguales a este, solo difieren en los puertos que se utilizan.

```
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 3.2: Configuración del cluster de dev

4. Se instala el controlador de Ingress (líneas 15-16), lo cual permitirá acceder a la aplicación a través de una URL customizada desde el exterior.

5. Se crea el *namespace* en el que se va a instalar la instancia de ArgoCD (línea 18).

6. Se crea un secreto en el que se almacena el valor de la clave de cifrado de los secretos de Kubernetes (líneas 20-22). El proceso de creación de cifrado y descifrado se explica en 3.1.

7. Se instala la Chart de Helm de Argo y se espera a que esté disponible (líneas 24-29). A la Chart se le pasan una serie de valores, de los cuales se habla en 3.1. Una vez instalado, se podrá acceder a ArgoCD de manera local. Para ello será necesario crear un "túnel" de un puerto libre local al puerto 443 del servicio principal de Argo. Esto se explica también en el

> **Important!**
> CITAR MANUAL DE USUARIO

8. Se aplica la configuración específica de Argo para el entorno que se está creando (líneas 31-33). Se puede ver la configuración para Argo en el entorno de `dev` en el Listing 3.3. Lo más importante de esta configuración se encuentra en las líneas 9-11, en las cuales se indica el repositorio (`repoURL`), la ruta desde la raíz (el directorio `dev`) y la rama (`targetRevision`) de donde Argo debe obtener los archivos que definen los recursos que se van a desplegar.

```
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 3.3: Configuración de Argo en dev

9. Se modifica el ConfigMap de Argo para que muestre el banner en la interfaz (línea 35).

10. Se comprueba la contraseña inicial que tiene el usuario `admin` y se muestra por pantalla (líneas 37-39). Estas son las credenciales que hay que utilizar para poder hacer *log in* en la interfaz de Argo. Por lo tanto, se muestran directamente por pantalla a medida que se van creando los *clusters* y tras la creación de todos ellos (línea 42).

En la figura 3.3 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.

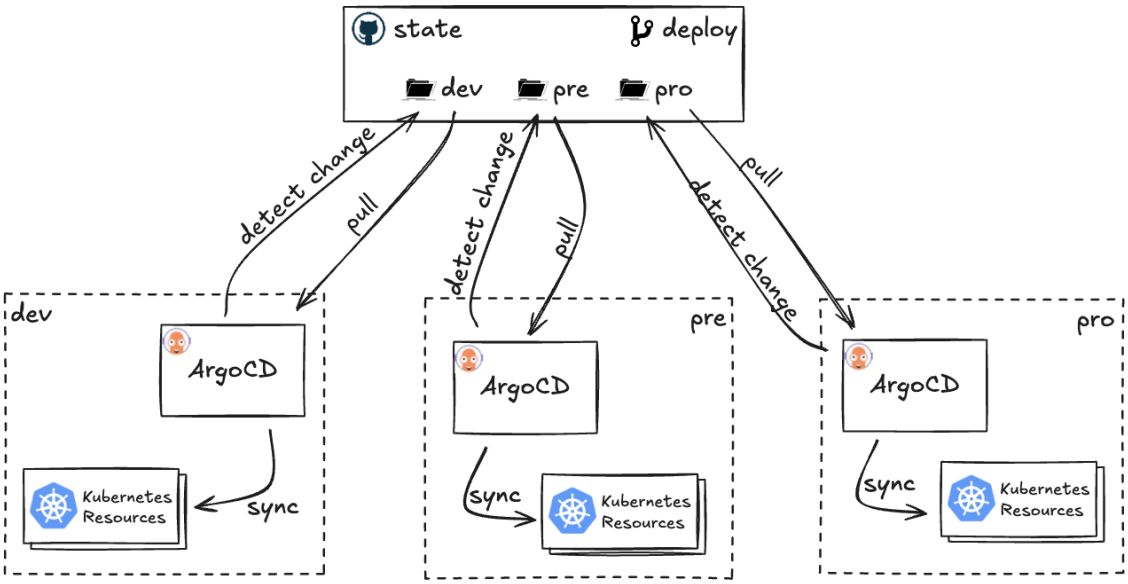

Figura 3.3: Clusters y comunicación con el repositorio de estado. Imagen creada con excalidraw.com

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.

Para encriptar los secretos se utilizan dos herramientas:

- `age`[42].

  `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

  > **Important!**
  > CITAR MANUALES DE USUARIO

  para poder probar la aplicación.

- SOPS[43].

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 3.4. 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 3.4: 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* 3.1. Sin embargo, es necesario decirle a Argo cómo utilizarla.

Para ello es necesario el uso de otras dos herramientas:

- `kustomize`[44].

  `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 3.5. 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 muestran cómo construir recursos a partir de otros. El generador que se utiliza en este trabajo se muestra en el Listing 3.6. 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 3.5: Archivo 'kustomization.yaml'

- `ksops`[45].

  `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 3.6 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 3.6: 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 3.1. El archivo `values.yaml` tiene el contenido que se muestra en el Listing 3.7. 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 3.1.
- 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).

```yaml
configs:
  cm:
    kustomize.buildOptions: "--enable-alpha-plugins --enable-exec
    ↪ "
    ui.bannerpermanent: "true"

repoServer:
  env:
    - name: XDG_CONFIG_HOME
      value: /.config
    - name: SOPS_AGE_KEY_FILE
      value: /.config/sops/age/keys.txt

  volumes:
    - name: custom-tools
      emptyDir: {}
    - name: sops-age
      secret:
        secretName: sops-age

  initContainers:
    - name: install-ksops
      image: viaductoss/ksops:v4
      command: ["/bin/sh", "-c"]
      args:
        - echo "Installing KSOPS and Kustomize...";
          mv ksops /custom-tools/;
          mv kustomize /custom-tools/;
          echo "Done.";
      volumeMounts:
        - mountPath: /custom-tools
          name: custom-tools

  volumeMounts:
    - mountPath: /usr/local/bin/kustomize
      name: custom-tools
      subPath: kustomize
    - mountPath: /usr/local/bin/ksops
      name: custom-tools
      subPath: ksops
    - name: sops-age
      mountPath: /.config/sops/age
```

Listing 3.7: 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 3.4 se puede ver el ciclo completo de encriptado y desencriptado de secretos

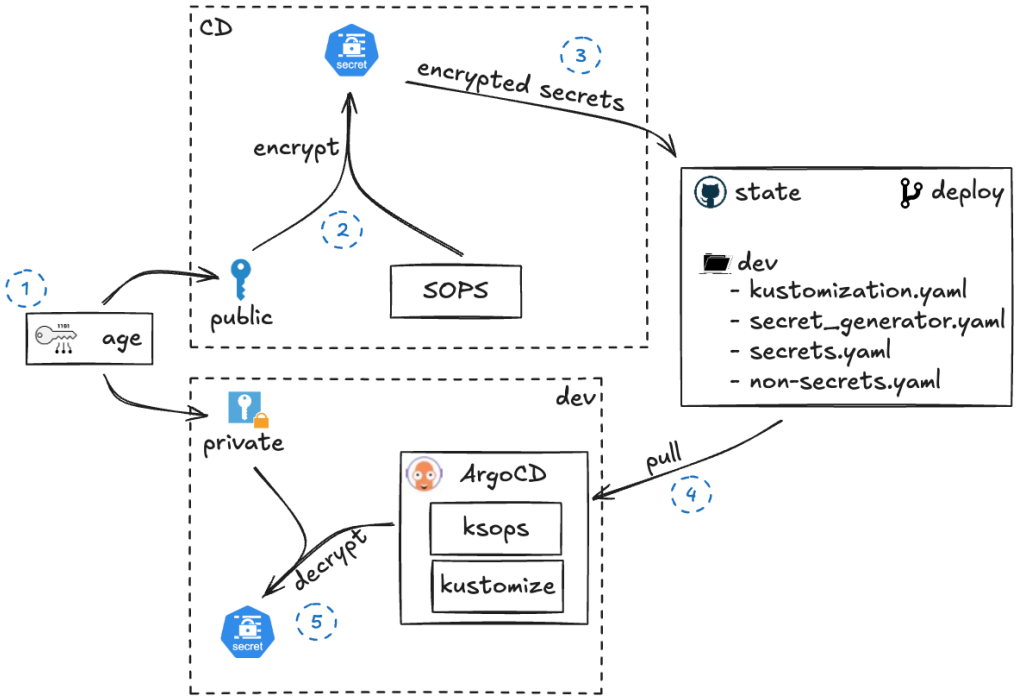

Figura 3.4: Clusters y comunicación con el repositorio de estado. Imagen creada con excalidraw.com

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 siempre 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[41].
  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 3.8. En este se realizan los siguientes pasos:

```
on:
  push:
    branches: [ main ]
  release:
    types: [ published ]
  workflow_dispatch:

jobs:
  cicd:
    runs-on: ubuntu-24.04
    steps:
      - % Clona el repositorio "zoo" en la ruta "/zoo"

      - % Clona el repositorio "state" en la ruta "/state"
        % utilizando el token STATE_REPO

      - name: Install Dagger
        uses: dagger/dagger-for-github@8.0.0

      - name: Determine environment
        id: env_tag
        run: |
          % Determina el entorno en el que se despliega,
          % teniendo en cuenta el *trigger* que ha lanzado
          % el workflow:
          %   - *push*    -> main
          %   - *release* -> *release* o *pre-release*

          echo "environment=${envi}" >> "$GITHUB_OUTPUT"
          echo "tag=${tag}" >> "$GITHUB_OUTPUT"

      - name: Recreate needed files
        run: |
          % Recrea el archivo .env para tenerlo disponible
          % en "zoo"
          echo "CR_PAT=${{ secrets.CR_PAT }}" >> ./.env
          % .... 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 3.8: Workflow de CI/CD

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). Esta *tag* es diferente para cada entorno. En el entorno de `dev` 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`.
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 3.1.
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.

- `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 de manera pública. 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.

  Diferencias con respecto al entorno de `dev`:

  - Se publica la imagen y los recursos en este entorno 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[46] 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.

  Los pasos que se realizan en el *workflow* son los mismos, solo cambian los elementos que se acaban de indicar.

- `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.

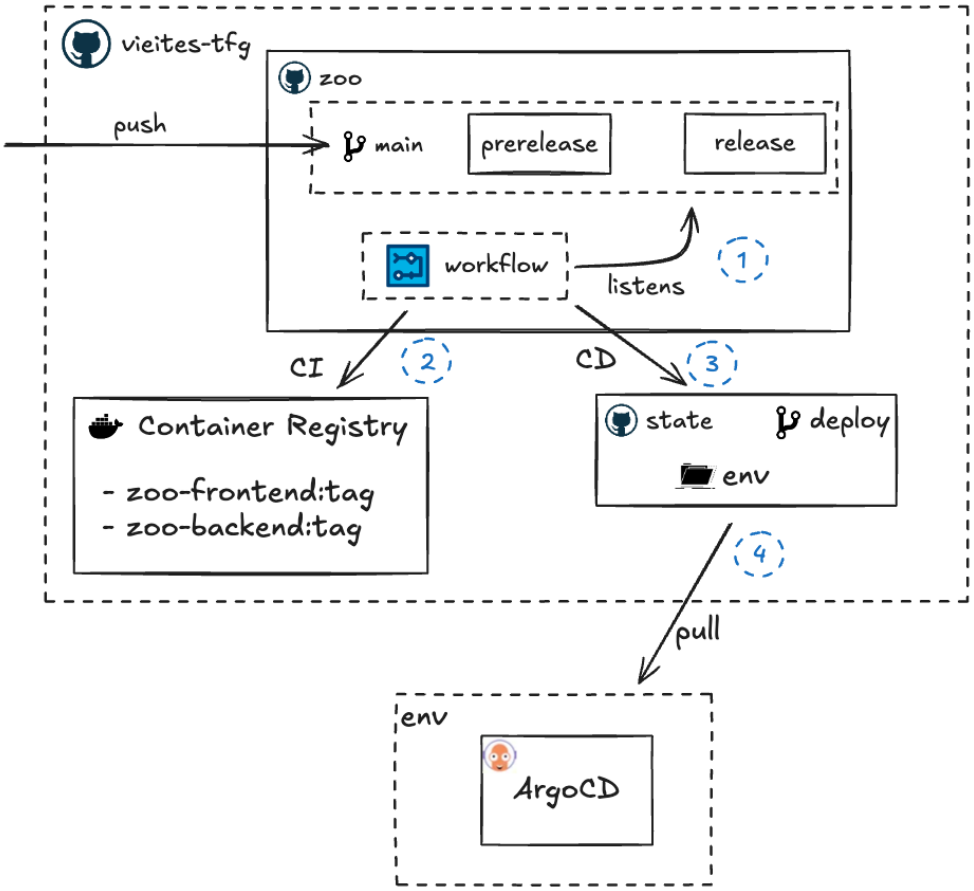

Figura 3.5: Descripción del proceso de promoción de entornos. Imagen creada con excalidraw.com

Para facilidad de pruebas y debido a que no es la finalidad de este trabajo, se ha decidido crear todos los *clusters* de forma local.

Cambios en este entorno con respecto a los anteriores:

- Se despliega la aplicación en él 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`).

En la figura 3.5 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 entorno correspondiente (`env`).

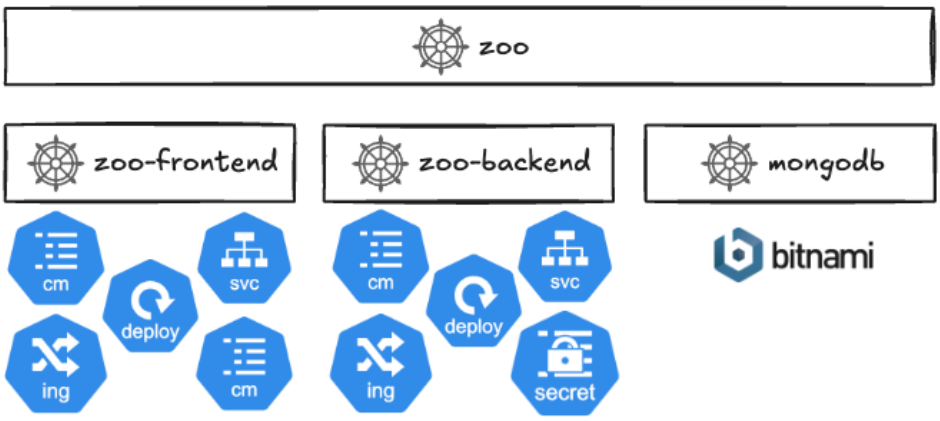

Figura 3.6: Diagrama de organización de las Charts de la aplicación. Imagen creada con excalidraw.com

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 3.6 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 3.9. 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 como Bitnami[47]. 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 garantizando mucha más seguridad de lo que se tendría en el caso de crear una Chart de Mongo propia para la aplicación.

```
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 3.9: Definición de la Chart umbrella de la aplicación

También se puede observar en la figura 3.6 los recursos que se crean para cada una de las *subcharts* de la aplicación. Para más detalle en cuanto a su funcionamiento, ver la sección 2.2.

El proceso de desarrolo 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 3.10. 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 pachage zoo
```

Listing 3.10: 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 3.11 (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 3.11: Workflow de publicación de la Chart en GH Pages

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`[48]. Esta permite integrar Charts de Helm y valores, aunque estos se encuentren en lugares diferentes. En la figura 3.7 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 3.12. 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 23-27 sí que dependen del entorno, y se

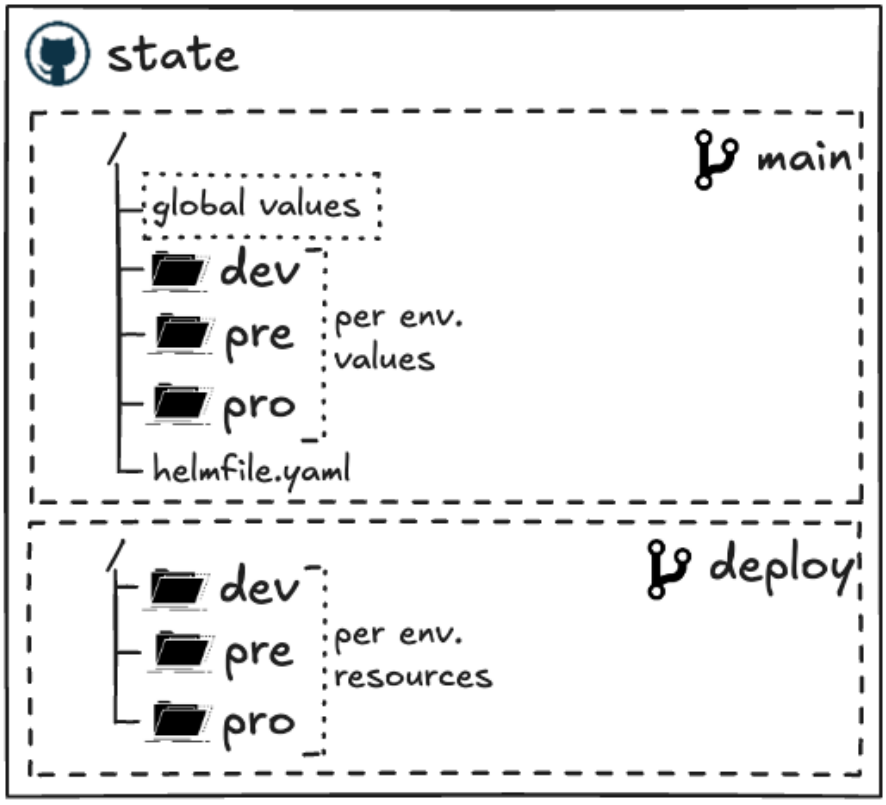

Figura 3.7: Diagrama de la estructura del repositorio de estado. Imagen creada con excalidraw.com

almacenan en un directorio dedicado a cada uno de ellos. En estos últimos valores se indica, por ejemplo, la *tag* a utilizar de las imágenes de Docker.

```
  repositories:
  - name: helm-repository
    url: https://raw.githubusercontent.com/vieites-tfg/
    ↪ helm-repository/gh-pages/

environments:
  dev:
    values:
      - ./dev.yaml
  # ... otros entornos

---

releases:
  - name: zoo-{{ .Environment.Name }}
    namespace: {{ .Environment.Name }}
    chart: helm-repository/zoo
    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 3.12: Archivo de configuración de helmfile

Un ejemplo de archivo de valores se puede observar en el Listing 3.13. Como se ha comentado, el valor de la *tag* se ve indicado en la línea 3, con el formato que se ha mencionado en la sección 3.1.

```
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 3.13: Archivo de valores de zoo–backend en dev

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 3.4 se

puede ver la estructura que tiene la rama de despliegue del repositorio *state*. Esa misma estructura es la que tienen los directorios correspondientes a todos los entornos en la rama `deploy`.

Capítulo 4

Implementación del *pipeline* con Dagger

Capítulo 5

Exemplos (eliminar capítulo na versión final)

5.1. Un exemplo de sección

Esta é *letra cursiva*, esta é **letra negrilla**, esta é letra subrallada, e esta é `letra curier`. Letra tiny, scriptsize, small, large, Large, LARGE e moitas más. Exemplo de fórmula: $a = \int_o^\infty f(t)dt$. E agora unha ecuación aparte:

$$S = \sum_{i=0}^{N-1} a_i^2. \tag{5.1}$$

As ecuaciones se poden referenciar: ecuación (5.1).

5.1.1. Un exemplo de subsección

O texto vai aquí.

5.1.2. Otro exemplo de subsección

O texto vai aquí.

Un exemplo de subsubsección

O texto vai aquí.

Un exemplo de subsubsección

O texto vai aquí.

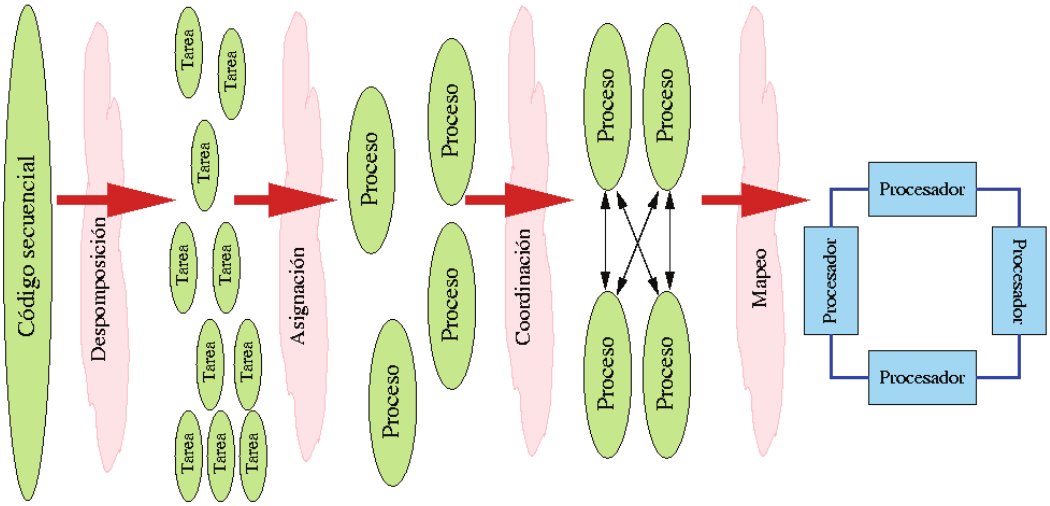

Figura 5.1: Esta é a figura de tal e cal.

Izquierda	Derecha	Centrado
ll	r	cccc
llll	rrr	c

Cuadro 5.1: Esta é a táboa de tal e cal.

Un exemplo de subsubsección

O texto vai aquí.

5.2. Exemplos de figuras e cadros

A figura número 5.1.
O cadro (taboa) número 5.1.

5.3. Exemplos de referencias á bibliografía

5.4. Exemplos de enumeracións

Con puntos:

- Un.
- Dous.
- Tres.

Con números:

1. Catro.
2. Cinco.
3. Seis.

Exemplo de texto verbatim:

```
O texto        verbatim
    se visualiza tal
           como se escribe
```

Exemplo de código C:

```c
#include <math.h>
main()
{  int i, j, a[10];
   for(i=0;i<=10;i++) a[i]=i; // comentario 1
   if(a[1]==0) j=1; /* comentario 2 */
   else j=2;
}
```

Listing 5.1:

Exemplo de código Java:

```java
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Display the string
    ↪ .
    }
}
```

Listing 5.2:

Capítulo 6

Conclusións e posibles ampliacións

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

Apéndice A

Manuais técnicos

En función do tipo de Traballo e metodoloxía empregada, o contido poderase dividir en varios documentos. En todo caso, neles incluirase toda a información precisa para aquelas persoas que se vaian encargar do desenvolvemento e/ou modificación do Sistema (por exemplo código fonte, recursos necesarios, operacións necesarias para modificacións e probas, posibles problemas, etc.). O código fonte poderase entregar en soporte informático en formatos PDF ou postscript.

Apéndice B

Manuais de usuario

Incluirán toda a información precisa para aquelas persoas que utilicen o Sistema: instalación, utilización, configuración, mensaxes de erro, etc. A documentación do usuario debe ser autocontida, é dicir, para o seu entendemento o usuario final non debe precisar da lectura doutro manual técnico.

Apéndice C

Licenza

Se se quere pór unha licenza (GNU GPL, Creative Commons, etc), o texto da licenza vai aquí.

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)*.

[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)*.

[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)*.

[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 Porject 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. "Examples/Go/Yarn-Build/Ci.go at Main · Dagger/Examples." GitHub, 2023, *https://github.com/dagger/examples/blob/main/go/yarn-build/ci.go*. Accedido el 21 de junio de 2025.

[34] Dagger. "Introducing the Dagger GraphQL API — Dagger." Dagger.io, 2022, *https://dagger.io/blog/graphql*. Accedido el 21 de junio de 2025.

[35] Dagger. "Daggerverse." Daggerverse.dev, 2025, *https://daggerverse.dev*. Accedido el 21 de junio de 2025.

[36] Dagger. "ContentKeeper Content Filtering." Dagger.io, 2025, *https://docs.dagger.io/reference/cli*. Accedido el 21 de junio de 2025

[37] Microsoft. "TypeScript - JavaScript That Scales." Typescriptlang.org, 2025, *https://www.typescriptlang.org*.

[38] Nx. "Lerna · a Tool for Managing JavaScript Projects with Multiple Packages." Lerna.js.org, 2025, *https://lerna.js.org*.

[39] MongoDB. "MongoDB." MongoDB, 2024, *https://www.mongodb.com*.

[40] You, Evan. "Vue.js." Vuejs.org, 2014, *https://vuejs.org*.

[41] 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.

[42] Valsorda, Filippo. "FiloSottile/Age." GitHub, 21 Apr. 2022, *https://github.com/FiloSottile/age*.

[43] getsops. "Getsops/Sops." GitHub, 9 Jan. 2024, *https://github.com/getsops/sops*.

[44] Kubernetes SIGs. "Kustomize - Kubernetes Native Configuration Management." Kustomize.io, 2025, *https://kustomize.io*.

[45] 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.

[46] Preston-Werner, Tom. "Versionado Semántico 2.0.0." Semantic Versioning, 2024, *https://semver.org/lang/es*.

[47] Broadcom. "Applications - Bitnami.com." Bitnami.com, 2024, *https://bitnami.com/stacks?stack=helm*. Accedido el 12 de julio de 2025.

[48] Helmfile Authors. "Helmfile." Readthedocs.io, 2025, *https://helmfile.readthedocs.io/en/latest*. Accedido el 12 de julio de 2025.