

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

1. Introducción	1
1.1. Objetivos generales	1
1.2. Relación de la documentación	3
2. Estado del arte y fundamentos teóricos	5
2.1. CI/CD	5
2.2. Ecosistema de herramientas	8
2.3. Dagger	17
3. Deseño	23
4. Probas	25
5. Exemplos (eliminar capítulo na versión final)	27
5.1. Un exemplo de sección	27
5.1.1. Un exemplo de subsección	27
5.1.2. Otro exemplo de subsección	27
5.2. Exemplos de figuras e cadros	28
5.3. Exemplos de referencias á bibliografía	28
5.4. Exemplos de enumeracións	28
6. Conclusións e posibles ampliacións	31
A. Manuais técnicos	33
B. Manuais de usuario	35
C. Licenza	37
Bibliografía	39

Índice de figuras

2.1. Proceso de integración continua.[4]	6
2.2. Proceso de despliegue continuo.[5]	7
2.3. Arquitectura de Kubernetes.[19]	15
2.4. Arquitectura de Helm.[23]	16
2.5. <i>pipelines</i> con Dagger sobre un <i>runtime</i> compatible con Docker.[24]	17
2.6. Uso de la API de GraphQL para Dagger.[31]	20
5.1. Esta é a figura de tal e cal.	28

Índice de cuadros

5.1. Esta é a táboa de tal e cal.	28
---	----

Índice de Listings

2.1. Makefile para compilación de un programa en C	8
2.2. Extracto de justfile utilizado en el proyecto	9
2.3. Extracto de Dockerfile utilizado en el proyecto	11
2.4. Construir y correr una imagen de Docker	11
2.5. docker-compose.yaml usado en el proyecto	12
2.6. Despliegue con Docker Compose	13
2.7. Ejemplo de objeto Ingress utilizado en este proyecto.	16
2.8. Código de Dagger con CUE	17
2.9. Primeros ejemplos del SDK de Go de Dagger.	19
2.10. Extracto de funciones de Dagger de este trabajo	21
2.11. Comando para lanzar el backend del proyecto	22
5.1.	29
5.2.	29

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.

No solo se va a utilizar Dagger como complemento del ciclo de desarrollo de una aplicación, sino que se va a comparar con la no utilización de este. Se evaluarán las ventajas que tiene su uso frente a métodos convencionales, entre las que destacarán su portabilidad, al poder crear *pipelines* de manera programática, implementando funciones que corren dentro de contenedores, permitiendo al programador tener el control total del entorno en el que se ejecuta la aplicación. En vez de intentar unir *scripts* creados a mano en diferentes entornos, el programador es capaz de componer acciones reusables, utilizando un lenguaje de programación y una API 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.

Estas son las preguntas se resuelven a lo largo del presente trabajo:

- ¿Cómo de viable es 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 resuelve Dagger frente al uso de otros métodos convencionales?

ELIMINAR: infierno del YAML, testeo local, portabilidad, lenguaje real para crear abstracciones

- ¿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.

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 (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 *pipeline* de CD termina con el despliegue de la aplicación sobre Kubernetes[8], utilizando Helm[9]. Esto permite levantar la aplicación en el entorno que le corresponda, según el estado en el que se encuentra cada versión.

- 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 describen las tecnologías utilizadas para implementar la aplicación *dummy*. También se explica cómo se ha organizado la infraestructura de despliegue.

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

Aquí se indican 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.

Lo que se intenta mejorar utilizando Dagger es 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 método innovador 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.

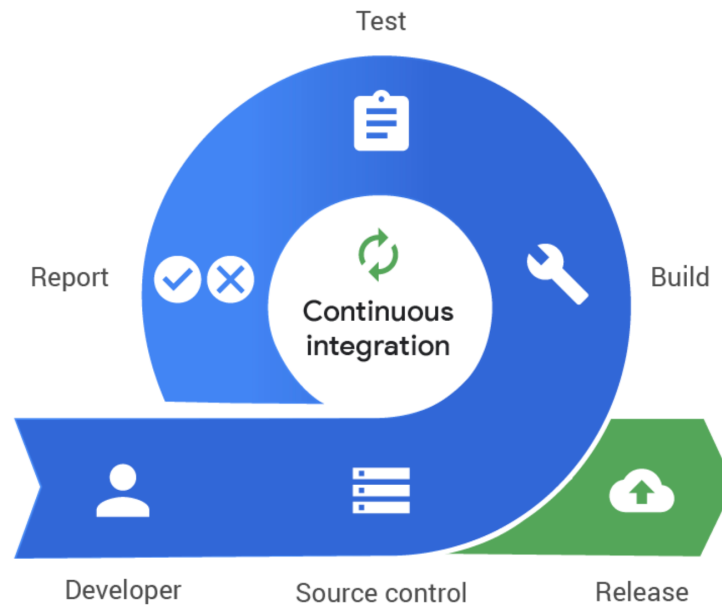


Figura 2.1: Proceso de integración continua.[4]

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 para que se incluyan definitivamente en el código fuente de la aplicación (Fig. 2.1).

Desde hace años se utiliza un sistema 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 incluirlo 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 a través de un análisis del propio código (*linting*).
- 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.

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 el rol DevOps[16] en muchas empresas, lo cual implica que la persona que ocupa este rol debe 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[21]. 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[22]. Argo es una herramienta que, como su nombre indica, facilita 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.

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 nueva 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 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.

```

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" ]]
20     then
21         docker build --target base -t zoo-base .
22     fi
23
24 # Accepts two parameters: 'entrypoint' and 'command'.
25 _run entrypoint command:
26     # '@' at the beginning of a command line prevents 'just' from
27     # printing the command before executing it.
28     @just _build_zoo_base
29     docker run --rm -w /app -v $PWD:/app --env-file .env --
30     ↪ entrypoint={{entrypoint}} zoo-base {{command}}
31
32 # --- PUBLIC RECIPES ---
33 init:
34     @just _run "yarn" "install"
35
36 down_vol:
37     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.

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”.

```
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 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 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.

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

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.

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

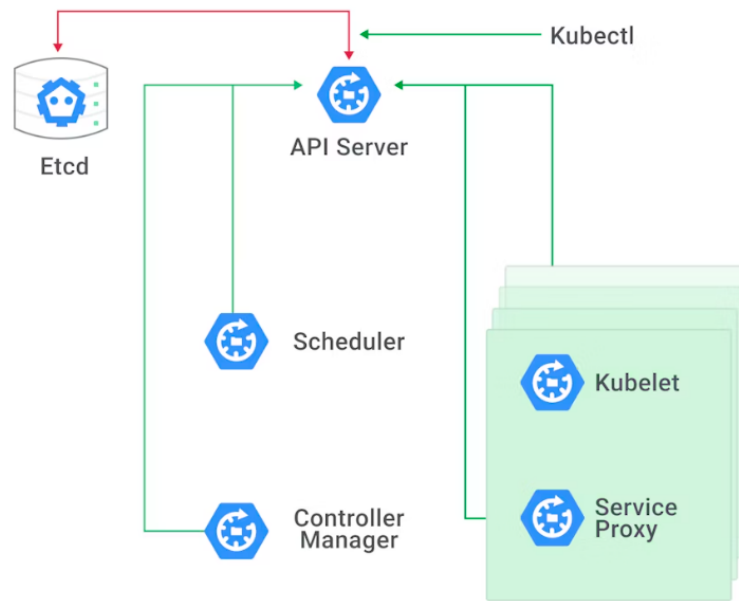


Figura 2.3: Arquitectura de Kubernetes.[19]

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.

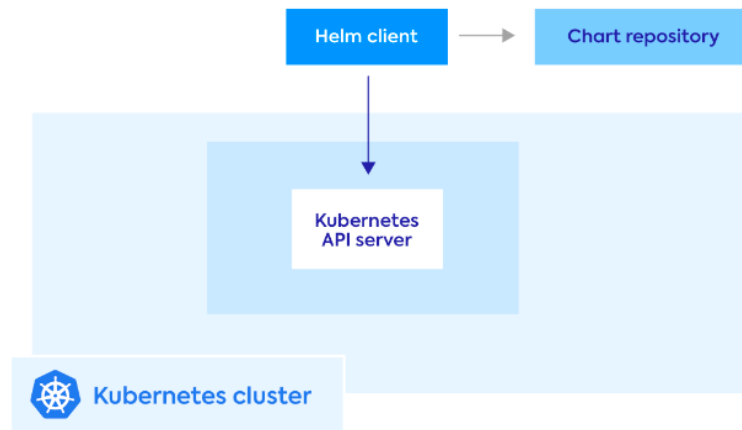


Figura 2.4: Arquitectura de Helm.[23]

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

```

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

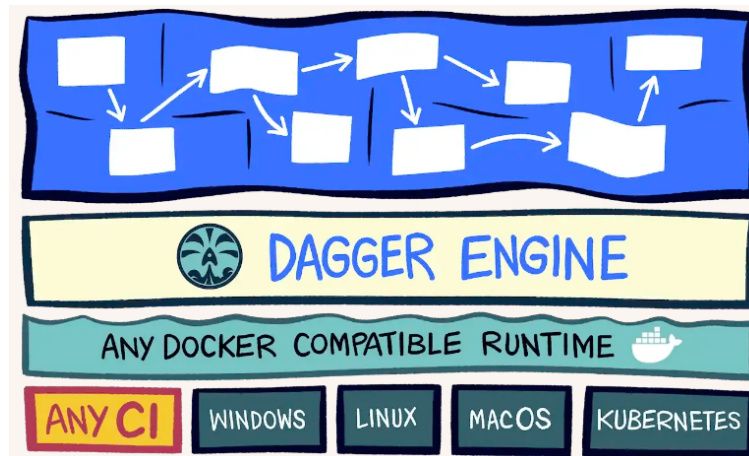


Figura 2.5: *pipelines* con Dagger sobre un *runtime* compatible con Docker.[24]

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 utilizando cualquier *runtime* de Docker para ejecutar las funciones que forman parte del SDK[26] 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[25]. 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 2.8 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 2.8: 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 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[28] 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 2.9 se puede observar un ejemplo de código[30] 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.
23         ↪ Stdout))
24     if err != nil {
25         return err
26     }
27     defer client.Close()
28
29     src := client.Host().Directory(".") // get the projects source
30     ↪ directory
31
32     yarn := client.Container().From("yarnpkg/node-yarn"). //
33     ↪ initialize new container from yarn image
34     WithDirectory("/src", src).WithWorkdir("/src"). // mount source
35     ↪ directory to /src
36     WithExec([]string{"yarn", "test"}) // execute yarn test command
37
38     // get test output
39     test, err := yarn.Stdout(ctx)
40     if err != nil {
41         return err
42     }
43     // print output to console
```

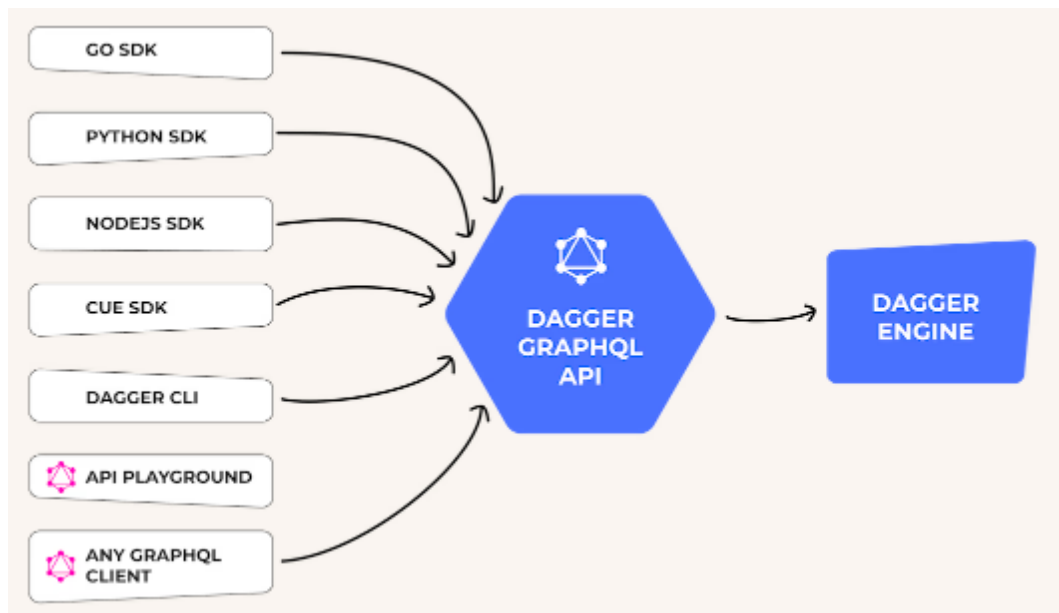


Figura 2.6: Uso de la API de GraphQL para Dagger.[31]

```

40  fmt.Println(test)
41
42  // execute build command and get build output
43  build, err := yarn.WithExec([]string{"yarn", "build"}).Stdout(
44    ↪ ctx)
45  if err != nil {
46    return err
47  }
48  // print output to console
49  fmt.Println(build)
50
51  return nil
52 }

```

Listing 2.9: 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.

GraphQL

Pero, ¿cómo es capaz el equipo de Dagger de desarrollar SDKs específicos para cada lenguaje tan rápidamente? Gracias al uso de GraphQL[29], 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
2 // file and returns the container.
3 func (m *Frontend) Build(ctx context.Context) *dagger.Container {
4     build := m.Base.
5         WithWorkdir("/app").
6         WithExec([]string{"lerna", "run", "--scope", "@vieites-tfg/"
7             ↪ zoo-frontend", "build"})
8     return build
9 }
10
11 // Based on the build stage, gets the executable file and creates
12 // a ready to run container with Nginx and the port 80 exported.
13 func (m *Frontend) Ctr(ctx context.Context) *dagger.Container {
14     build := m.Build(ctx)
15
16     dist := build.Directory("/app/packages/frontend/dist")
17
18     ctr := dag.
19         Container().
20         From("nginx:alpine").
21         WithWorkdir("/usr/share/nginx/html").
22         WithDirectory(".", dist).
23         WithExposedPort(80).
24         WithEntrypoint([]string{"nginx", "-g", "daemon off;"})
25
26     return ctr
27 }

```

Listing 2.10: 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 Daggervse[32]. 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[33] (*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 2.11: Comando para lanzar el backend del proyecto

De esta manera (Listing 2.11) 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 **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 Daggervse, junto con su CLI para invocarlos, proponen una manera muy poderosa de crear *pipelines* para construir, probar y desplegar aplicaciones.

Capítulo 3

Deseño

Debe describirse como se realiza o Sistema, a división deste en diferentes compoñentes e a comunicación entre eles. Así mesmo, determinarase o equipamento hardware e software necesario, xustificando a súa elección no caso de que non fose un requisito previo. Debe achegarse a un nivel suficiente de detalle que permita comprender a totalidade da estrutura do produto desenvolvido, utilizando no posible representacións gráficas.

Capítulo 4

Probas

Plan de probas (con evidencias) que verifica a funcionalidade e correctitude global do sistema, e se leva a cabo.

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: $a = \int_0^\infty f(t)dt$. E agora unha ecuación aparte:

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

As ecuaciones se poden referenciar: ecuación (5.1).

5.1.1. Un exemplo de subsección

O texto vai aquí.

5.1.2. Outro exemplo de subsección

O texto vai aquí.

Un exemplo de subsubsección

O texto vai aquí.

Un exemplo de subsubsección

O texto vai aquí.

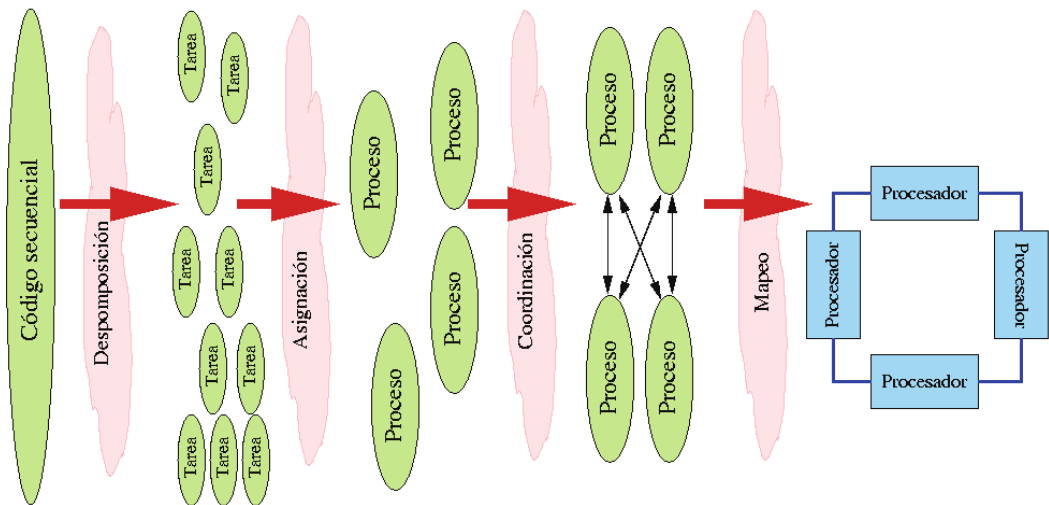


Figura 5.1: Esta é a figura de tal e cal.

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:

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

Exemplo de código C:

```
1 #include <math.h>
2 main()
3 {   int i, j, a[10];
4     for(i=0;i<=10;i++) a[i]=i; // comentario 1
5     if(a[1]==0) j=1; /* comentario 2 */
6     else j=2;
7 }
```

Listing 5.1:

Exemplo de código Java:

```
1 class HelloWorldApp {
2     public static void main(String[] args) {
3         System.out.println("Hello World!"); // Display the string
4     }
5 }
```

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 incluírase 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_\(software\)](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. “Kind.” Kind.sigs.k8s.io, 2025, <https://kind.sigs.k8s.io>.
- [21] GitLab. “¿Qué Es GitOps?” Gitlab.com, GitLab, 9 Feb. 2022, <https://about.gitlab.com/es/topics/gitops>. Accedido el 20 de junio de 2025.
- [22] Argo Project Authors. “Argo CD.” Github.io, 2025, <https://argoproj.github.io/cd>. Accedido el 20 de junio de 2025.
- [23] 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.
- [24] 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.
- [25] CUE. “The CUE Language Specification.” CUE, 16 Apr. 2025, <https://cuelang.org/docs/reference/spec>. Accedido el 21 de junio de 2025.
- [26] 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.
- [27] GitHub. “Features · GitHub Actions.” GitHub, 2025, <https://github.com/features/actions>.

- [28] “The Go Programming Language.” Go.dev, 2025, <https://go.dev>. Accedido el 21 junio de 2025.
- [29] The GraphQL Foundation. “GraphQL: A Query Language for APIs.” GraphQL.org, 2012, <https://graphql.org>.
- [30] 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.
- [31] Dagger. “Introducing the Dagger GraphQL API — Dagger.” Dagger.io, 2022, <https://dagger.io/blog/graphql>. Accedido el 21 de junio de 2025.
- [32] Dagger. “Daggerverse.” Daggerverse.dev, 2025, <https://daggerverse.dev>. Accedido el 21 de junio de 2025.
- [33] Dagger. “ContentKeeper Content Filtering.” Dagger.io, 2025, <https://docs.dagger.io/reference/cli>. Accedido el 21 de junio de 2025.