UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

Desarrollo de artefactos Helm para computación intensiva sobre Kubernetes

Trabajo Fin de Máster

Máster Universitario en Computación en la Nube y de Altas Prestaciones / Cloud and High-Performance Computing

AUTOR/A: Gálvez Ruiz, Alejandro

Tutor/a: Blanquer Espert, Ignacio

Cotutor/a: Calatrava Arroyo, Amanda

CURSO ACADÉMICO: 2023/2024

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

Desarrollo de artefactos Helm para computación intensiva sobre Kubernetes

Septiembre de 2024

Autor: Alejandro Gálvez Ruiz

Tutores: Ignacio Blanquer Espert
Amanda Calatrava Arroyo

Resumen

El entrenamiento de redes neuronales para la inteligencia artificial, o la realización de simulaciones científicas; son ejemplos de tareas más que comunes hoy en día. Estos problemas tan complejos requieren de grandes recursos de procesamiento, lo que conocemos como Computación Intensiva.

Crear una arquitectura orientada a resolver este tipo de problema no es ni mucho menos una tarea sencilla, y en muchos casos requiere de esfuerzo económico y técnico.
No obstante, el crecimiento exponencial de la virtualización en la última década nos ha abierto un abanico mucho más amplio de posibilidades para el despliegue de todo tipo de entornos. Estos avances nos han permitido no solo una mayor flexibilidad en la configuración y gestión de infraestructuras, sino también una significativa reducción de costos operativos y la optimización de recursos, haciendo accesibles tecnologías de alto rendimiento que anteriormente estaban reservadas a grandes centros de investigación o empresas con vastos recursos.

Por lo tanto, este proyecto tiene como objetivo la creación de un conjunto de especificaciones de infraestructuras virtuales para la computación intensiva. Específicamente, se hará uso de artefactos de Kubernetes y *charts* de Helm para crear dos soluciones: una cola *batch* mediante SLURM con un *backend* MPI, y un servidor de Jupyter con un cluster IPython.

Abstract

Training neural networks for artificial intelligence, or performing scientific simulations; are common tasks nowadays. These complex problems require significant processing resources, what we know as Intensive Computation.

To create an architecture designed to solve this type of problem is not a simple task, and in many cases, it requires both a financial and a technical effort.
However, the exponential growth of virtualization over the past decade has opened up a much broader range of possibilities for deploying all kinds of environments. These advancements have allowed not only a greater flexibility in the configuration and management of infrastructures, but also a significant reduction in operational costs and resource optimization, making high-performance technologies, which were previously reserved for large research centers or companies with vast resources, accessible

Therefore, this project aims to create a set of virtual infrastructure specifications for Intensive Computation. Specifically, we will use Kubernetes artifacts and Helm charts to develop two solutions: a batch queue with SLURM and a MPI backend, and a Jupyter server with an IPython cluster.

Índice general

Índice de figuras

Índice de tablas

Listado de acrónimos

CP	Control Program
VLAN	Virtual Local Area Network
VPN	Virtual Private Network
RAID	Redundant Array of Independent Disks
NFS	Network File System
MPI	Message Passing Interface
SLURM	Simple Linux Utility for Resources Management
API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
YAML	YAML Ain't Markup Language
JSON	JavaScript Object Notation
SSH	Secure Shell
CLI	Command Line Interface
MUNGE	MUNGEUid 'N' Gid Emporium (MUNGE)Uid 'N' Gid Emporium
HPL	High-Performance Linpack
FLOPS	Floating Point Operations Per Second
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security
CI/CD	Continuous Integration / Continuous Delivery

Capítulo 1

Introducción

A lo largo de este capítulo se encontrará una entrada en contexto sobre el problema a resolver. De este modo, se hará un repaso sobre las motivaciones y objetivos de este trabajo, así como de la metodología llevada a cabo para resolverlos.

1.1 Motivación

Al inicio de la década de los 60, era común que grandes organizaciones contaran con unidades centrales o *mainframes*. Estas computadoras estaban diseñadas específicamente para el procesamiento de datos masivo, por lo que era común que los científicos (los principales usuarios de estos sistemas) enviaran sus trabajos mediante una técnica conocida como *batch processing* [26]. Esto consistía en la ejecución de programas sin ningún tipo de interacción por parte del usuario.

Por aquel entonces, existían diversos tipos de *mainframes*, cada uno con sus propias características y peculiaridades. Sin embargo, todos contaban con una limitación muy importante: únicamente era posible ejecutar un trabajo de un único usuario simultáneamente.

Esta restricción representaba una gran desventaja. En primer lugar porque implicaba que los diferentes usuarios de estos sistemas debían turnarse para utilizar el *mainframe*, lo que también representaba un problema de seguridad. En segundo lugar porque estos aparatos tan costosos, debido a sus características específicas, ofrecían muy poca flexibilidad

El primer avance llegó con la creación del Multics, un sistema operativo que permitía su uso simultáneo por varios usuarios. Sin embargo, IBM fue más allá y revolucionó aquella idea con un sistema que permitía la coexistencia de varios sistemas operativos. Es decir, el *mainframe* estaría ahora virtualmente dividido en pequeñas maquinas con su propio sistema operativo corriendo encima de lo que llamaron Control Program (CP), o lo que actualmente conocemos como Hipervisor [26].

Esta arquitectura fue el primer paso hacia la **virtualización**, es decir, la ejecución de varios entornos virtuales sobre una misma máquina física. Esto conlleva un gran número de ventajas.

- *Flexibilidad.* Es posible crear sistemas mucho más personalizados para cada solución específica.

- *Reducción de tiempo y costos.* Es posible levantar distintos entornos virtuales sin depender de aspectos físicos como el tiempo de montaje de un sistema y todos los gastos económicos que esto conlleva.

- *Seguridad.* Es posible crear entornos aislados entre sí, de modo que, si existe un problema en uno de ellos, el resto siga funcionando sin inconvenientes.

- *Migración.* Es posible crear copias de estos entornos, o *scripts* que los repliquen, de manera que su despliegue o migración sea mucho más rápida y sencilla.

Desde que IBM introdujo en el mercado su CP, la virtualización ha evolucionado enormemente hasta convertirse en una pieza fundamental dentro de la industria. De este modo, podemos encontrarla en recursos como la red, en forma de Virtual Local Area Network (VLAN) o Virtual Private Network (VPN); el almacenamiento, como en la misma paginación de memoria o en los Redundant Array of Independent Disks (RAID); o directamente en plataformas, como los hipervisores ya mencionados o los contenedores [3].

La contenerización se refiere a la creación de entornos aislados dentro de un mismo sistema operativo. Esto permite generar pequeñas unidades ejecutables llamados contenedores que incluyen únicamente los servicios y dependencias necesarios.

La principal diferencia entre estos contenedores y las máquinas virtuales radica en que, mientras estas últimas virtualizan todo el *hardware* subyacente, los contenedores se ejecutan directamente sobre el sistema operativo anfitrión (generalmente Linux). Esto nos permite desarrollar piezas de *software* mucho más ligeras y fácilmente transportables.

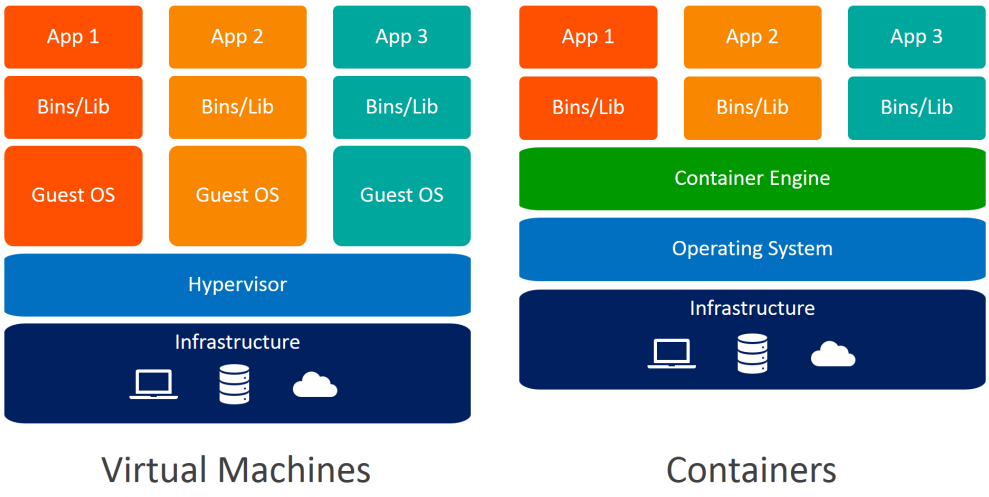


Figura 1.1: Contenedores y Máquinas Virtuales comparados. Fuente: [32].

A lo largo de este trabajo, se pretende profundizar en esta última tecnología, aprovechando sus múltiples ventajas para desarrollar soluciones orientadas a la computación intensiva. En particular, se busca facilitar su uso mediante la creación de entornos bajo demanda que puedan ser utilizados tanto para formación y pruebas, como en producción. De esta manera, proponemos

una forma simplificada de crear clústeres de cómputo intensivo utilizando recursos virtualizados, de modo que cualquier usuario pueda configurarlos y gestionarlos de manera sencilla y efectiva.

1.2 Objetivos

El objetivo principal de este proyecto es la construcción con Kubernetes [28] y Helm [14] de dos especificaciones de infraestructuras virtuales para computación intensiva: un sistema de colas *batch* mediante SLURM [48] con Message Passing Interface (MPI) como *backend*, y un servidor de Jupyter [42] con clúster IPython [23].

Para alcanzar dicho objetivo, se han definido los siguientes objetivos específicos:

- Ampliar los conocimientos obtenidos durante el máster sobre las herramientas de virtualización de Docker [9] y Kubernetes.
- Aprender a gestionar el despliegue de soluciones en Kubernetes con Helm.
- Definir una arquitectura orientada a la computación intensiva. Identificar por tanto cada caso de uso y adaptar cada sistema en función de sus necesidades.
- Aprovechar la ventaja que nos ofrecen este tipo de herramientas de virtualización para automatizar lo máximo posible el despliegue de los sistemas ideados.
- Virtualizar el servicio Network File System (NFS) a usar por los nodos de ambos sistemas.
- Implementar y validar varias soluciones para la ejecución de trabajos de computación intensiva

1.3 Metodología

1.3.1 Metodología ágil

Ágil es un conjunto de métodos y metodologías que comparten una serie de valores y principios aplicables a muchas áreas dentro de la ingeniería del *software* como la gestión de proyectos o el diseño y arquitectura *software* [54].

Tradicionalmente, las empresas hacían uso de metodologías en cascada a la hora de gestionar sus proyectos. Este consistía en un proceso mucho más lineal donde se recogían unos requisitos, se realizaba un diseño y se implementaba un *software*. Sin embargo, con el paso del tiempo, el proceso de creación de *software* se volvió demasiado complejo, las necesidades del cliente cambiaban, y la metodología en cascada quedaba anticuada [2].

Es entonces en 2001, cuando un grupo de expertos se reunieron para buscar una solución a este problema y nació el término «ágil». Todos coincidieron en que las empresas estaban tan centradas en planificar y documentar sus sistemas que habían perdido el foco en lo realmente importante, complacer a sus clientes. Su reacción fue la creación del Manifiesto Ágil, un

compendio de 12 principios que agrupan 4 valores que cambiaron por completo el modo en que se gestionan los proyectos [2].

- *Individuos e iteraciones por encima de procesos y herramientas.* Es más importante contar con un buen grupo de trabajo con los conocimientos técnicos adecuados, facilidad de adaptación y la capacidad de trabajar en equipo que las herramientas y los procesos.

- *Software funcionando por encima de la documentación extensiva.* Si bien la documentación es una parte importante del proceso de creación de software, estos deben ser cortos y limitarse a lo fundamental.

- *Colaboración con el cliente por encima de la negociación contractual.* Más que un ambiente de enfrentamiento en el cual las partes buscan su beneficio propio, se busca la participación constante del cliente, desde el comienzo hasta la culminación de proyecto, y su interacción con el equipo de desarrollo.

- *Respuesta ante el cambio por encima de seguir un plan.* Dada la naturaleza cambiante de la tecnología y la sociedad moderna, se busca la creación de planes flexibles para poder adaptarse a los cambios que puedan surgir.

1.3.2 Metodología ágil frente a metodologías tradicionales

Como se puede observar en la comparativa realizada en la Tabla 1.1 , las metodologías tradicionales ofrecen una forma de trabajo mucho más estricta y con menor implicación entre los miembros del grupo de trabajo.

Parámetro	**Ágil**	**Tradicional**
Adaptabilidad	Más adaptable y flexible para adaptarse a las necesidades que vayan surgiendo	Planificación más estricta y predecible
Requisitos	Requisitos pueden cambiar a lo largo del proyecto	Requisitos no cambian durante el proyecto
Implicación del cliente	Alta	Baja
Tamaño de grupos	Pequeños	Grandes
Comunicación	Comunicación informal en encuentros cara a cara	Comunicación formal a través de escritos
Acercamiento	Iterativo	Lineal
Modelo de desarrollo	Entrega de evolutivos	Ciclo de vida
Riesgo	Mayor	Menor

Tabla 1.1: Comparativa metodologías ágiles y tradicionales

Esto puede suponer en muchos aspectos un riesgo algo menor, pues siempre se tiene en mente cual es el objetivo final del proyecto y cuales son los pasos precisos para alcanzarlos. Sin embargo, también requiere de un conocimiento de la materia en la que se va a trabajar mucho mayor, con el fin de realizar todas las estimaciones con la máxima exactitud, además de saber con precisión cuales son los límites de tu proyecto.

1.3.3 Metodología usada en este proyecto

Debido a que el alumno va a trabajar con una serie de tecnologías algo desconocidas, se considera que es difícil crear una planificación estricta, ya que es posible que vayan surgiendo complicaciones a lo largo del proyecto que supongan cambios en el mismo. Es por lo tanto, y siguiendo con los puntos expuestos en anteriores secciones, por lo que se ha optado por seguir una metodología ágil.

De este modo, la primera tarea que se realizó fue definir el alcance del proyecto, que se puede encontrar en la Sección 1.2 recientemente descrita.

A continuación se creó un *backlog* o lista de tareas a realizar que fue especificándose a medida que avanzaba el proyecto. A estas tareas se les asignó un nivel de esfuerzo medido en días hábiles[1], y se fueron organizando en ciclos de trabajos llamados *sprint*.

Sprint 1. Familiarización con los entornos

Si bien el alumno disponía de ciertos conocimientos sobre algunas de las herramientas con las que se iba a trabajar, también desconocía completamente alguna de ellas como Helm. Por lo tanto, se decidió dedicar un *sprint* específico a la familiarización con esta herramienta.

El objetivo final de este *sprint* es desplegar varias soluciones utilizando Helm, lo que permitirá al alumno generar un documento informativo sobre la herramienta y sus ventajas, para luego aplicarlas en el desarrollo del proyecto. Las tareas que conforman este *sprint* son:

- *Introducción a Helm.* Esfuerzo estimado: 2 días.
- *Instalación y configuración de Helm.* Esfuerzo estimado: 1 día.
- *Despliegue de unas soluciones básicas.* Esfuerzo estimado: 2 días.
- *Profundización en la configuración de Helm.* Esfuerzo estimado: 3 días.
- *Despliegue de soluciones más avanzadas.* Esfuerzo estimado: 3 días.
- *Preparación de un informe final sobre la herramienta.* Esfuerzo estimado: 3 días.

Esfuerzo estimado total: 14 días.

Sprint 2. Creación de las imágenes

Tras obtener una base sólida sobre las herramientas que se utilizarán en el proyecto, se procederá a la creación de las imágenes correspondientes a los nodos que se desplegarán en cada solución.

El objetivo final de este *sprint* es disponer de todas las imágenes necesarias para cada arquitectura. Para ello, es esencial analizar detalladamente cada caso de uso, entender qué servicios requiere cada herramienta de computación intensiva, e idear los nodos que compondrán cada solución. Las tareas que conforman este *sprint* son:

[1] Se considera día hábil cualquier día de Lunes a Viernes, independientemente de si es festivo.

- *Análisis de cada caso de uso.* 2 días.
- *Diseño de los nodos necesarios para cada solución.* Esfuerzo estimado: 3 días.
- *Creación de las imágenes.* Esfuerzo estimado: 5 días.
- *Construcción y pruebas.* Esfuerzo estimado: 2 días.
- *Documentación.* Esfuerzo estimado: 2 días.

Esfuerzo estimado total: 14 días.

Sprint 3. Migración a Kubernetes

Una vez se han construido y comprobado el funcionamiento de las imágenes del *sprint* anterior, el siguiente paso es migrar la solución a Kubernetes.

El objetivo final de este *sprint* es crear una arquitectura en Kubernetes que utilice las imágenes ya disponibles. Al finalizar este *sprint*, se contará con una versión preliminar de ambas soluciones. Las tareas que conforman este *sprint* son:

- *Diseño de cada arquitectura en Kubernetes.* Esfuerzo estimado: 3 días.
- *Creación de la arquitectura ideada.* Esfuerzo estimado: 9 días.
- *Pruebas y validación.* Esfuerzo estimado: 4 días.
- *Documentación.* Esfuerzo estimado: 5 días.

Esfuerzo estimado total: 21 días.

Sprint 4. Integración con Helm

Tras haber alcanzado una solución estable con Kubernetes, se pondrá en práctica lo aprendido en el *Sprint 1*.

El objetivo final de este *sprint* es integrar las soluciones desarrolladas en el *sprint* anterior con la herramienta de Helm, aprovechando sus ventajas para facilitar y automatizar el despliegue de los clústeres. Las tareas que conforman este *sprint* son:

- *Comprobación de elementos automatizables.* Esfuerzo estimado: 4 días.
- *Gestión de retos ante la actualización de valores "en caliente".* Esfuerzo estimado: 4 días.
- *Despliegue y pruebas.* Esfuerzo estimado: 3 días.
- *Documentación.* Esfuerzo estimado: 3 días

Esfuerzo estimado total: 14 días.

Sprint 5. Adición de nuevas funcionalidades

Después de haber alcanzado una solución estable y automatizar su despliegue con Helm, se procederá a añadir nuevas funcionalidades que aumenten la complejidad de los clústeres.

El objetivo final de este *sprint* es explotar las funcionalidades adicionales que ofrece cada plataforma, tales como la incorporación de bases de datos para gestionar la contabilidad de las tareas o la inclusión de múltiples usuarios. Las tareas que conforman este *sprint* son:

- *Estudio de posibles añadidos a cada solución.* Esfuerzo estimado: 6 días.
- *Implementación de las funcionalidades propuestas.* Esfuerzo estimado: 8 días
- *Despliegue y pruebas.* Esfuerzo estimado: 4 días.
- *Documentación.* Esfuerzo estimado: 3 días.

Esfuerzo estimado total: 21 días.

Sprint 6. Adición de NFS contenerizado

Al finalizar el *sprint* anterior, y dado que se logró completar cada *sprint* en el tiempo estimado, se propuso la expansión del proyecto para incluir la posibilidad de desplegar un servidor NFS contenerizado.

El objetivo final de este *sprint* es ofrecer la posibilidad de utilizar un nodo NFS local en cada uno de los clústeres. De este modo, se facilita aún más el despliegue de una arquitectura de computación intensiva que, en dicho caso, no requerirá de mayor configuración extra. Las tareas que conforman este *sprint* son:

- *Recolección de información sobre el funcionamiento de NFS.* Esfuerzo estimado: 3 días.
- *Creación de la imagen.* Esfuerzo estimado: 4 días.
- *Implementación y adaptación a cada solución.* Esfuerzo estimado: 3 días.
- *Pruebas y validación.* Esfuerzo estimado: 2 días.
- *Documentación.* Esfuerzo estimado: 2 días.

Esfuerzo estimado total: 14 días.

Sprint 7. Realización de Benchmarks

Tras alcanzar el estado final de los clústeres, se realizarán una serie de pruebas demandantes sobre los mismos para conocer el rendimiento de los sistemas.

El objetivo final de este *sprint* es contar con una serie de documentos en forma de gráficos y tablas que muestren como se comportan los sistemas frente a una serie de problemas complejos que hayamos planteado. Las tareas que conforman este *sprint* son las siguientes.

- *Elección de los benchmarks más adecuados para cada sistema.* Esfuerzo estimado: 1 día.
- *Preparación de los entornos para poder realizar cada benchrmark.* Esfuerzo estimado: 3 días.
- *Realización de los benchmarks.* Esfuerzo estimado: 3 días.
- *Documentar y estudiar los resultados obtenidos.* Esfuerzo estimado: 3 días.

Esfuerzo estimado total: 10 días.

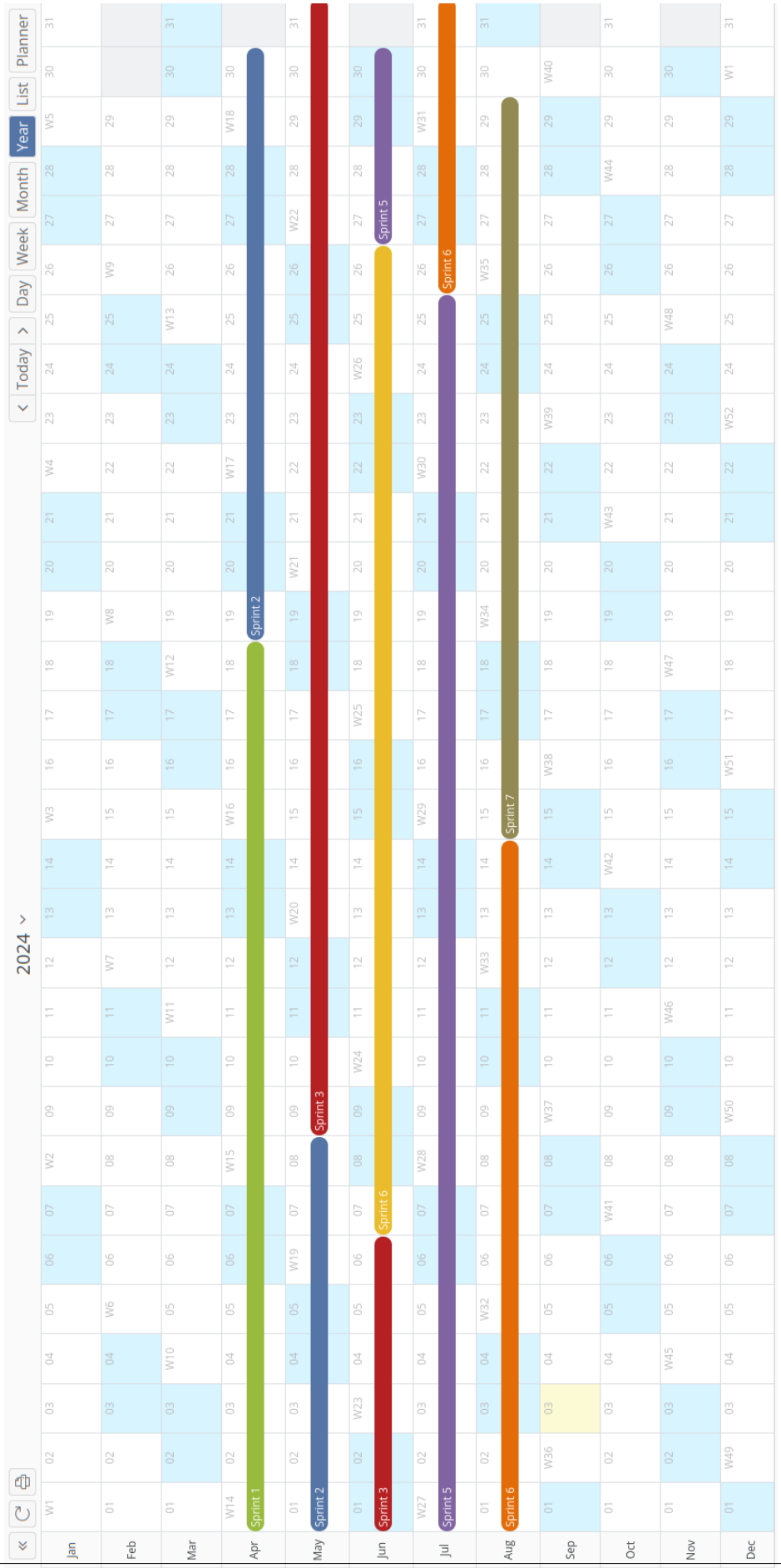


Figura 1.2: Diagrama final con los *sprints* mencionados. Cada *sprint* estaba estimado en días *hábiles*.

Capítulo 2

Estado del Arte

A lo largo de este capítulo se hará un repaso de las tecnologías que utilizaremos durante este proyecto, explicando brevemente su importancia y funcionamiento. Haremos una distinción entre aquellas herramientas para la virtualización y aquellas para la computación intensiva.

2.1 Herramientas para la virtualización

2.1.1 Docker

En la sección 1.1 se comentó brevemente el concepto de contenerización. Docker es una plataforma que proporciona las herramientas necesarias para gestionar contenedores a través de un entorno amigable.

El funcionamiento de esta plataforma es sencillo. Los desarrolladores pueden crear imágenes, que son paquetes con una serie de herramientas, archivos y dependencias necesarias. A partir de estas imágenes es posible crear tantos contenedores como se desee, pero todos partirán de la misma configuración especificada en la imagen. Es decir, el concepto es similar al de un proceso que es un programa en ejecución.

Lo cierto es que la analogía de programa/proceso aquí es más que acertada, puesto que la realidad es que estos contenedores no son más que procesos ejecutados dentro de una máquina *host* (que puede ser nuestro propio ordenador o un servidor enorme). Estos procesos se encuentran aislados gracias a ciertas características que nos ofrece el *Kernel* de Linux.

Pongamos un ejemplo. Imaginemos que queremos desplegar un servidor web creado en Python [60]. Para ello, podríamos crear una imagen que tenga instalado dicho lenguaje, que contenga las librerías necesarias para nuestro servidor (por ejemplo, Flask [59]) y, finalmente, incluir el código del servidor desarrollado.
Una vez tengamos la imagen, podríamos crear un contenedor a partir de ella y lanzarlo en nuestro servidor. E incluso, si la demanda del servidor aumentara significativamente, podríamos lanzar otro contenedor para satisfacer todas las peticiones.

Pero, ¿Cómo se crean estas imágenes? Lo cierto es que lo más probable es que alguien ya haya abordado este caso de uso en alguna ocasión y su imagen se encuentre disponible en algún

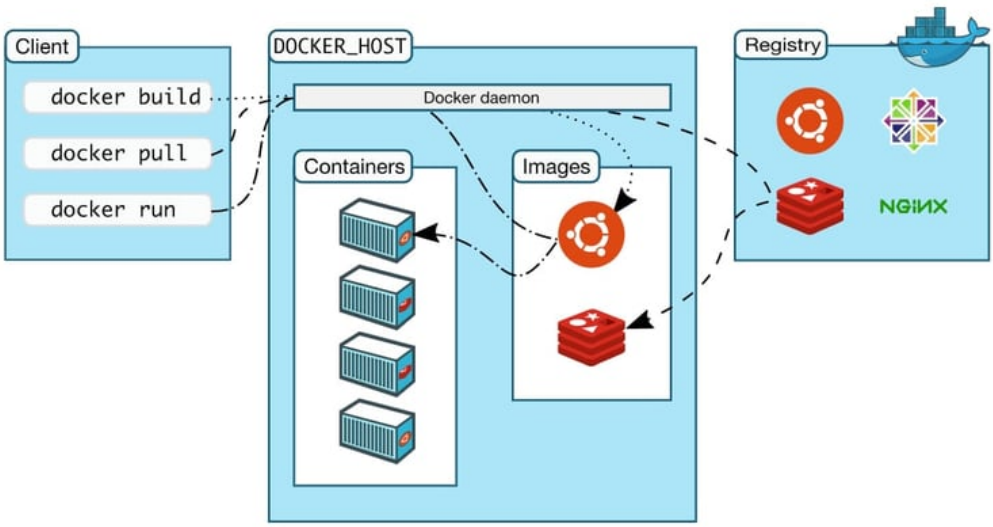


Figura 2.1: Funcionamiento de Docker. Fuente: [44].

repositorio de imágenes como Docker Hub [38] disponible para uso. Pero en la mayoría de los casos, el *modus operandi* es crear una nueva a través de un Dockerfile.

Un Dockerfile es un archivo con las instrucciones necesarias para la creación de una imagen. En el Listado 2.1 se puede comprobar parte de la sintaxis utilizada para la creación de una imagen descrita en el ejemplo anterior. Es posible consultar la funcionalidad de cada cláusula en el siguiente enlace: [10].

```
FROM ubuntu:latest

RUN apt-get update && apt-get install -y \
    python3 \
    python3-pip

RUN pip3 install flask

WORKDIR /app

COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt

COPY app.py app.py

EXPOSE 8080

CMD ["python3", "app.py"]
```

Listing 2.1: Dockerfile para un servidor en Python

Además de Docker, existen otras herramientas para gestionar contenedores. La más común de todas es Podman [40], que ofrece algunas ventajas como ser algo más ligero debido a la ausencia

de un *daemon* intermedio, ser más seguro al permitir lanzar contenedores sin permisos *root*, o permitir crear *pods* (concepto que comentaremos en la siguiente sección).

La razón por la que se ha utilizado Docker a pesar de todas estas ventajas, es el peso que tiene dentro de la industria. Podman apenas nació en 2019, por lo que aunque es una tecnología interesante y a tener en cuenta, esta no cuenta aún con la extensa documentación y comunidad de usuarios que tiene Docker, el cual es aún un estándar dentro de la industria.

2.1.2 Kubernetes

Docker es una excelente herramienta para gestionar contenedores en un solo host, y podemos incluso utilizar Docker Compose [8] para facilitar la gestión de múltiples instancias. Sin embargo, este enfoque presenta varias limitaciones, principalmente porque todos los contenedores dependen de un único punto de fallo: la máquina host donde se ejecutan.

Es aquí donde Kubernetes (K8s) ofrece una ventaja significativa, ya que no solo permite orquestar y gestionar múltiples contenedores, sino que permite hacerlo dentro de varios *hosts* de forma centralizada. Dentro de este entorno los contenedores pueden comunicarse entre sí y se distribuyan eficientemente las cargas de trabajo, eliminando la necesidad de gestionar manualmente la asignación de recursos y la distribución del tráfico, a la vez que mantenemos una alta disponibilidad.

Kubernetes nació de la mano de Google. Se trata de una plataforma de código abierto, y actualmente es la herramienta de orquestación de contenedores más extendida. Si bien existen otras alternativas como Docker Swarm [55] o Nomad [35], ninguna ha conseguido realizarle competencia real a Kubernetes, que ya es un estándar *de facto* dentro la industria y en la computación científica.

Arquitectura

Kubernetes consigue gestionar todos los aspectos anteriormente descritos mediante la construcción de un *Kubernetes cluster* con dos tipos de nodos: *Worker Nodes* y *Control Plane*.

- *Control Plane*. Este componente es el cerebro y corazón del clúster. Se encarga de exponer la Application Programming Interface (API) de Kubernetes, asignar los recursos a cada *Worker Node*, gestionar que estos tengan el estado deseado, etc.
- *Worker Nodes*. Son el músculo del clúster. Se encargan de ejecutar los contenedores y gestionar las conexiones entre estos.

En la Figura 2.2 se puede observar una versión esquematizada del clúster descrito. Como se puede ver, cada tipo de nodo está modularmente dividido en varios componentes,se puede ver, cada tipo de nodo está modularmente dividido en varios componentes que en que en conjunto realizan las acciones detalladas anteriormente. No obstante, si se desea obtener una información más detallada sobre estos componentes es posible hacerlo en el siguiente enlace: [29].

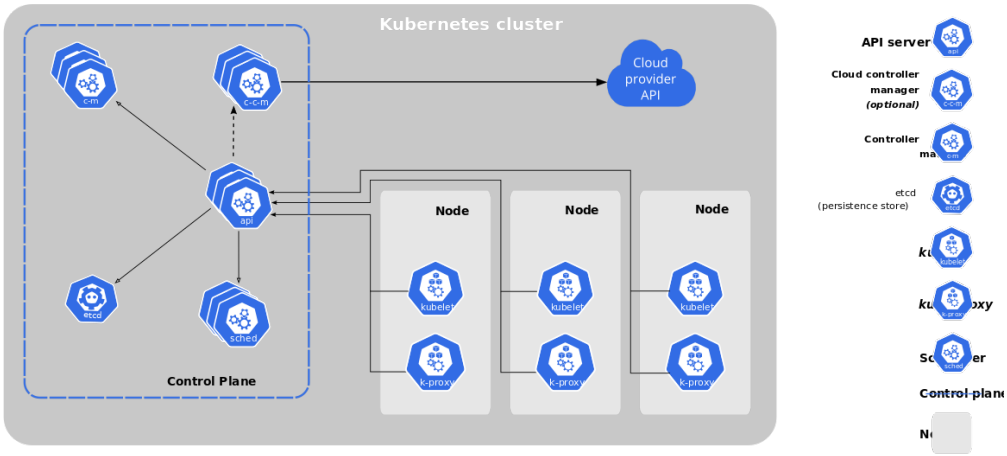


Figura 2.2: Componentes de un *Kubernetes Cluster*. Fuente: [29].

Pods

Mientras que en Docker trabajábamos con contenedores, en Kubernetes la unidad mínima de ejecución pasa a ser el Pod [41]. Un Pod es un conjunto de contenedores que comparten ciertos recursos como el almacenamiento o la red.

Aunque este concepto pueda parecer que añade una complejidad innecesaria, aún es posible ejecutar un único contenedor en un Pod (y, de hecho, es el patrón más común). Por otro lado, también se ofrece una solución óptima para aquellos casos de uso donde varios contenedores necesitaban compartir recursos.
Por ejemplo, imaginemos nuestro anterior ejemplo del servidor web. Es posible que este servicio ofrezca una serie de contenido a sus consumidores. Podríamos, entonces, añadir un nuevo contenedor que se encargue de gestionar este contenido y almacenarlo en un volumen. Debido a que el almacenamiento dentro de un Pod es compartido entre contenedores, nuestro servidor web podría acceder a dicho contenido sin tener que gestionarlo directamente. De esta manera desacoplamos dos tareas diferentes en dos contenedores distintos. Este concepto se conoce como *Sidecar Container* [47], y aunque es un caso de uso más avanzado, es interesante tenerlo en cuenta.

Por otro lado, Kubernetes ofrece más recursos para trabajar con los Pods. Por ejemplo ofrece Deployments [7], para desplegar varias copias de un mismo Pod; o Jobs [25], pensados para Pods que se ejecutan una (o un determinado número de veces) y luego terminan.

Services y otros mecanismos de red

En Kubernetes todos los Pods llevan asignados una dirección IP, y estos pueden comunicarse entre si por defecto. Sin embargo, estas direcciones son efímeras, lo que significa que si por alguna razón el Pod se reinicia, su IP muy posiblemente habrá cambiado.

Para solventar esto, se utiliza un recurso llamado Service [46]. Este recurso expone una serie de *endpoints* que reenvían el tráfico a una serie de Pods en función de la política descrita. Debido a que el ciclo de vida de este recurso es independiente de el del Pod, podemos estar seguros de que la comunicación se realizará sin problemas.

Así, retomando una vez más nuestro ejemplo, podríamos querer añadir un *frontend* que ofrezca una interfaz web más amigable para acceder al servidor web. Para comunicar estos dos Pods con IPs efímeras, podríamos colocar un Service delante del Pod del servidor web con una dirección que será siempre conocida. De esta manera nuestro *frontend* enviará cualquier comunicación al Service, que reenviará el mensaje al servidor.

Además del Service, Kubernetes ofrece muchos otros recursos relacionados con el tráfico de red. Uno de ellos es Ingress [20], un recurso encargado de permitir el acceso externo a los Pods mediante Hypertext Transfer Protocol (HTTP).

Definición de recursos y puesta en marcha

Existen varias maneras de crear recursos en Kubernetes. Quizás la más común es el uso de ficheros descriptivos escritos en YAML como el del Listado 2.2. Es posible comprobar los distintos parámetros de cada recurso en el siguiente enlace [21] .

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: python-web-server-pod
  labels:
    app: python-web
spec:
  containers:
  - name: python-web-server
    image: my-python-web-server-image
    ports:
    - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: python-web-service
spec:
  selector:
    app: python-web
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
  type: ClusterIP
```

Listing 2.2: Archivo YAML para desplegar un Service y un Pod con la imagen creada en el Listado 2.1

Una vez creado nuestros archivos YAML, podremos desplegarlos usando la herramienta de línea de comandos Kubectl [27].

```
kubectl apply -f mi-web-pod.yaml
kubectl delete -f mi-web-pod.yaml
kubectl replacce -f mi-web-pod.yaml
```

Listing 2.3: Ejemplo de comandos de Kubectl para gestionar el ciclo de vida de los recursos definidos en un fichero.

2.1.3 Helm

Como podemos suponer a partir del apartado anterior, gestionar recursos mediante Kubectl puede llegar a ser una tarea complicada cuando trabajamos en proyectos más grandes.
Es en este contexto donde nace Helm, una herramienta para la gestión de paquetes en Kubernetes.

Helm organiza los proyectos en *charts* [15], que son colecciones de manifiestos de Kubernetes (que Helm denomina *templates*) junto con una serie de archivos de configuración. Cabe recalcar la importancia de estos archivos, pues nos brindan la posibilidad de introducir valores en nuestros *templates* al realizar un despliegue de nuestro *chart*.

Por ejemplo, imaginemos que en un momento dado queremos cambiar el puerto de nuestro servidor web que vimos en el Listado 2.2. En este caso, la solución es muy sencilla puesto que podríamos simplemente cambiarlo manualmente. Pero, ¿qué sucedería si nuestro proyecto fuera mucho más grande? Es posible que dicho valor aparezca repetido en otros manifiestos y/o que haya más de un miembro trabajando en el proyecto y no conozca con exactitud todos los archivos donde se especificó dicho puerto.

Para solventar esto, Helm nos permite crear archivos `values.yaml`[1] donde incluir estos valores, como por ejemplo el del Listado 2.4.

```
server:
  port: 8080
```

Listing 2.4: Archivo `values.yaml` con el valor del puerto de nuestro servidor.

Posteriormente, Helm incluirá automáticamente dicho valor allá donde le hagamos referencia en nuestro *template*, que ahora lucirá como se muestra en el Listado 2.5.

```
apiVersion: v1
kind: Pod
metadata:
  name: python-web-server-pod
  labels:
    app: python-web
spec:
  containers:
  - name: python-web-server
    image: my-python-web-server-image
    ports:
    - containerPort: {{ .Values.server.port }}
---
apiVersion: v1
kind: Service
```

[1] Existen otros mecanismos para introducir estos valores, por ejemplo al lanzar el *chart* mediante línea de comandos.

```
  metadata:
    name: python-web-service
  spec:
    selector:
      app: python-web
    ports:
    - protocol: TCP
      port: 80
      targetPort: {{ .Values.server.port }}
    type: ClusterIP
```

Listing 2.5: *Template* para el despliegue de nuestro servidor web.

Para una mejor inserción de valores, Helm pone a nuestra disposición una serie de funciones, como operadores matemáticos, insertores de archivos o conversores de tipos de datos; y estructuras de control de flujo, como bucles o condicionales. Para obtener más información sobre como crear *templates*, se recomienda acceder al siguiente enlace: [56].

La creación y puesta en marcha de un *chart* se hace mediante línea de comandos. En el siguiente enlace [16] es posible comprobar todas sus posibilidades, aunque incluiremos las más comunes en el Listado 2.6.

```
helm create <nombre>
helm install <nombre> <chart>
helm uninstall <nombre>
helm list
helm status <release>
```

Listing 2.6: Comandos más comunes en Helm.

2.2 Herramientas para la computación intensiva

2.2.1 Ipython y Jupyter

Ipython surgió como un entorno interactivo para Python que ofrecía una serie de características para trabajar más eficientemente. Con el tiempo, la plataforma fue creciendo exponencialmente hasta el punto en que se decidió dividirla en diferentes proyectos. De esta manera, todas aquellas partes que ya funcionaban independientemente del lenguaje se movieron a nuevos proyectos bajo el dominio de Jupyter [62].

Así, Jupyter es un gran abanico de proyectos que intentan ofrecer *software* y herramientas para una computación más interactiva. Entre todos estos proyectos debemos destacar Jupyter Notebook, una plataforma que nos permite crear Notebooks, o documentos que combinan código, texto plano, gráficos o visualizaciones 3D entre otras cosas, y que son fácilmente compartibles entre usuarios [42]. Por otro lado, JupyterHub es una plataforma que extiende las funcionalidades de Jupyter Notebook para múltiples usuarios. JupyterHub permite la creación y gestión de entornos de Notebooks en un servidor compartido, lo que resulta muy útil para implementar servidores para clases, laboratorios de investigación o equipos de trabajo colaborativos [43].

Como mencionamos anteriormente, IPython fue dividido en muchos proyectos una vez alcanzó una dimensión considerable. En la Figura 2.3 se puede revisar el estado de estos proyectos. Como se puede comprobar, IPython ha quedado relegado a un *Kernel* más de todos los que ofrece la plataforma. Un *Kernel*[2] es, a grandes rasgos, el proceso encargado de la ejecución del código.

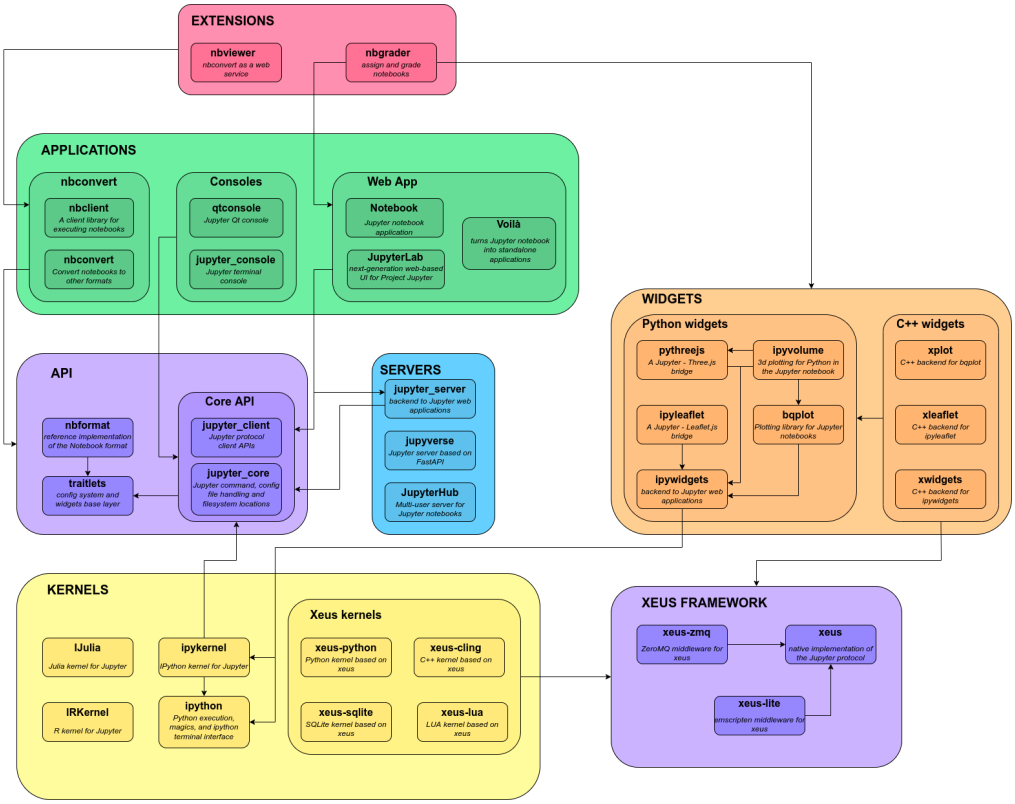


Figura 2.3: Proyectos de Jupyter. Fuente: [4].

Todas las interfaces, ya sea el mismo Jupyter Notebook o una simple consola, hacen uso de este *Kernel* para ejecutar el código del usuario. Esta comunicación entre *Frontend* y *Kernel* se realiza mediante mensajes JavaScript Object Notation (JSON) enviados sobre *sockets* de ZeroMQ [3].

Esta independencia entre *Frontend* y *Kernel* ofrece muchas posibilidades, como conectar más de un *Frontend* simultáneamente a un mismo *Kernel*, o facilitar el desarrollo de los mismos.

[2]No hay que confundir el concepto de Kernel del sistema operativo con el de Kernel de IPython. Son conceptos distintos.

[3]Su protocolo es descrito en el siguiente enlace: `https://jupyter-client.readthedocs.io/en/latest/messaging.html#messaging`

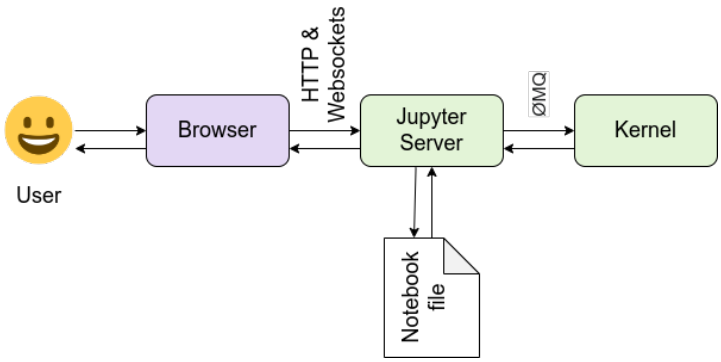


Figura 2.4: Cómo se produce la comunicación en Jupyter Notebook. Fuente: [4].

IPyParallel

Por último, IPython ofrece un *framework* para la computación paralela denominado IPyParallel [37] que consiste principalmente de cuatro componentes:

- *IPython Controller*. Proceso encargado de distribuir las tareas y recopilar los resultados.
- *IPython Engine*. Proceso encargado de recibir peticiones de código para luego proceder a su ejecución y retorno de resultados.
- *IPython Hub*. Proceso que mantiene un registro de las conexiones entre los componentes, los clientes, sus peticiones y resultados.
- *IPython Schedulers*. Proceso encargado de decidir cómo y cuándo se asignan las tareas a los Engine.

2.2.2 SLURM

SLURM es un sistema de gestión de trabajos para clústeres muy utilizado en Computación de Alto Rendimiento.

En esta herramienta, los distintos usuarios del clúster envían solicitudes para ejecutar sus trabajos. SLURM se encarga de gestionar estas solicitudes, asignándoles una prioridad y colocándolas en una cola de trabajos. Una vez dichos trabajos se encuentran a la cabeza de la cola, SLURM les asigna los recursos necesarios (nodos de cómputo) y los ejecuta.
Así, este *software* ofrece a sus usuarios un *framework* para iniciar, ejecutar y monitorear una serie de trabajos en sus nodos [49].

Para conseguir esto, SLURM se compone de principalmente dos demonios.

- *Slurmctld*. Se encuentra en el nodo principal y es el encargado de la gestión de los recursos y los trabajos.
- *Slurmd*. Se encuentra en cada nodo de cómputo y se encarga de la ejecución de los trabajos de manera similar a como lo haría un *shell* remoto.

Opcionalmente, existen otros demonios como *slurmdbd*, para almacenar información en una base de datos; o *slurmrestd* para el uso de SLURM a través de una API REST.

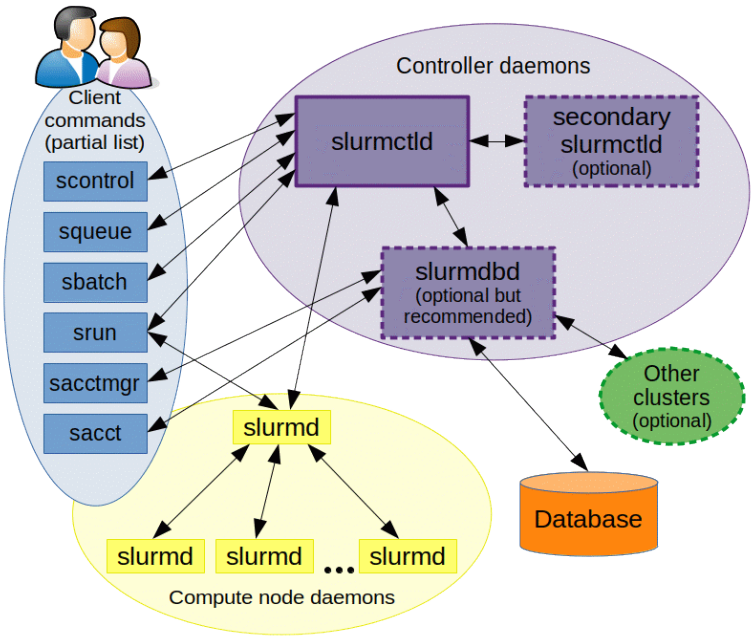


Figura 2.5: Componentes de SLURM y como interactúan. Fuente: [49].

SLURM trabaja con cuatro tipo de entidades [49].

- *Nodo.* Es el recurso de cómputo.
- *Partición.* Es un conjunto lógico de nodos.
- *Trabajo.* Asignación de recursos a un usuario durante un tiempo específico.
- *Pasos del trabajo.* Grupo de tareas de un trabajo.

En la Figura 2.6 se muestra un ejemplo del uso de las cuatro entidades. Como se puede observar, encontramos dos particiones: una con doce nodos y otra con seis. En la primera, se ha creado un trabajo con una única tarea, a la que se le ha asignado seis nodos. En la segunda, se ha creado otro trabajo pero con dos tareas, cada uno con dos nodos asignados.

2.2.3 OpenMPI

MPI es un estándar para la especificación de librerías para el intercambio de mensajes muy utilizado en la computación paralela [30]. Es especialmente útil en sistemas de memoria distribuida, donde cada proceso mantiene su propia memoria aislada. Por lo tanto, cuando varios procesos deben colaborar, estos se coordinan mediante el intercambio de mensajes.
Esto contrasta con otras herramientas de computación paralela como OpenMP [17], que está diseñado para sistemas de memoria compartida y se enfoca en aprovechar las capacidades de los procesadores para mejorar la eficiencia de los programas.

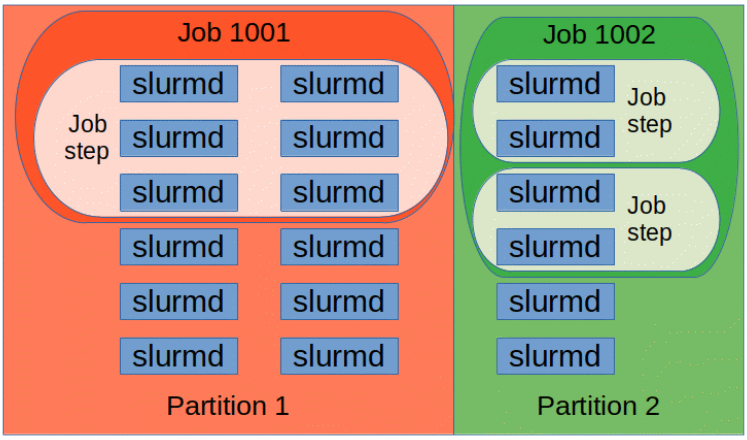


Figura 2.6: Ejemplo de uso de entidades en SLURM. Fuente: [49].

Volviendo a MPI, existen varias implementaciones de este estándar, algunas privadas como aquellas de Intel o IBM, y otras libres y de código abierto como OpenMPI [36] y MPICH [31]. No existen diferencias demasiado grandes entre estas dos últimas implementaciones, puesto que todas parten de la misma especificación ya comentada. Por lo tanto, para este proyecto se ha optado por usar OpenMPI debido a que tenemos más experiencia con dicha especificación.

2.2.4 NFS

Un NFS es un servicio que permite a diferentes *hosts* montar sistemas de archivos remotamente y tratarlos como si fueran locales. De esta manera, diferentes nodos pueden compartir un mismo sistema de archivos y acceder al mismo contenido. Por lo tanto, se puede entender su importancia en clústeres de computación intensiva formados por varios nodos.

A lo largo de su evolución, NFS ha visto diferentes versiones. Sus inicios al público fueron con NFSv2, que hacía uso de User Datagram Protocol (UDP) y era adecuado para almacenar archivos de tamaño reducido. NFSv3, llegó para solventar los problemas de rendimiento que suponían que las escrituras en el servidor fueran asíncronas, además añadió soporte para Transmission Control Protocol (TCP) y ficheros mucho más grandes.
Finalmente, unos años después llegó NFSv4, que ha recibido diferentes mejoras a lo largo de los años. A partir de ahora, NFS haría uso de un puerto bien conocido (2049) lo que facilitaría las comunicaciones y la implementación de reglas de *firewall*. Por otro lado, incluye la capacidad de realizar un control de acceso a nivel de usuario, mientras que hasta entonces solo era posible a nivel de equipo. Sin emabrgo, esto supone una mayor complejidad en la configuración y puesta en marcha.

En la práctica, tanto NFSv3 como NFSv4 siguen siendo utilizados y soportados debido a sus respectivas ventajas y compatibilidades.

Capítulo 3

Desarrollo

En este capítulo se describirá el desarrollo llevado a cabo para alcanzar el objetivo descrito en la Introducción de este proyecto. Para ello, inicialmente se expondrán las arquitecturas ideadas, y finalmente nos pondremos manos a la obra con su creación desde la construcción de las imágenes hasta su materialización con Kubernetes y Helm.

Todos los archivos mencionados en esta sección pueden ser accedidos desde el repositorio del proyecto en `https://github.com/algaru01/TFM`. Dentro del repositorio existen dos ramas, cada una específica de cada solución.

3.1 Arquitectura ideada

Durante las secciones 2.2.1 y 2.2.2 se abordó cuales eran los componentes de cada servicio. A lo largo de esta sección, se realizará una descripción general de la arquitectura que se construirá a partir de dichos componentes para cada una de las dos soluciones que se busca en este proyecto.

Estas arquitecturas estarán ideadas desde el punto de vista de cada herramienta, más adelante se explicará la arquitectura equivalente desplegada con Kubernetes.

3.1.1 SLURM

En la Figura 3.1 se puede observar la arquitectura que se ha ideado para el clúster de SLURM con *backend* MPI. Constará de los siguientes elementos.

- *Frontend*. Nodo encargado de correr el demonio *slurmctld* que gestiona los trabajos. Los usuarios podrán conectarse mediante Secure Shell (SSH) a este nodo para ejecutar trabajos o comprobar sus estados mediante el Command Line Interface (CLI) de SLURM.

- *Workers*. Nodos encargados de correr el demonio *slurmd* que ejecuta los trabajos. Este nodo deberá contar contar con la biblioteca OpenMPI.

- *Database*. Nodo encargado de correr el demonio *slurmdbd* para la contabilidad de los trabajos. Para ello contará también de una base de datos MySQL [34], pues es la opción

más recomendada por los propios desarrolladores [1].
Si bien este servicio es opcional, se ha optado por incluirlo debido a que es recomendado para llevar un registro de los trabajos que ejecuta cada usuario.

- *NFS.* Nodo encargado de ofrecer los directorios compartidos entre los nodos *Worker* y *Frontend*. Necesario para el lanzamiento de tareas y recepción de sus resultados.

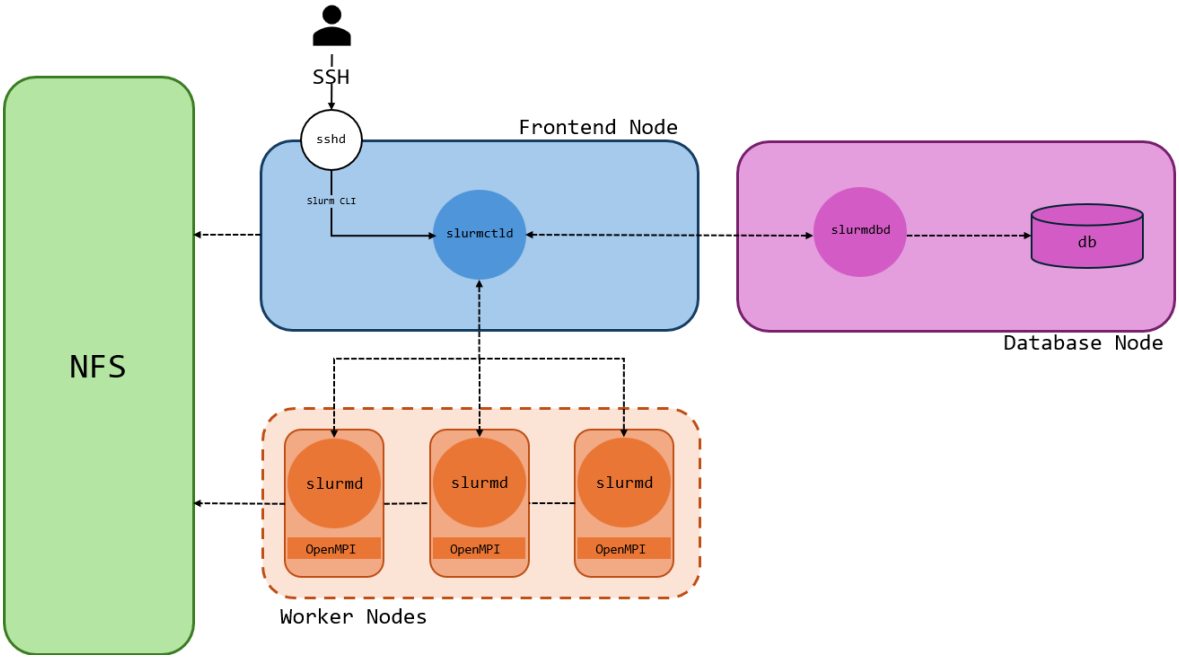


Figura 3.1: Arquitectura ideada para el clúster de SLURM.

3.1.2 Jupyter

En la Figura 3.2 se puede observar la arquitectura que se ha ideado para el servidor de Jupyter con un clúster de IPython. Constará de los siguientes elementos.

- *Frontend.* Nodo encargado de lanzar el servidor de Jupyter a través del cual accederán los usuarios. Además correrá el servicio `IPController` encargado de gestionar las tareas de computación paralela a través de IPython.

- *Workers.* Nodos encargados de ejecutar el servicio `IPEngine` para la recepción de tareas por parte del IPController y su posterior ejecución paralela junto con sus nodos homólogos. Este nodo deberá contar con una serie de librerías Python necesarias por los usuarios.

- *NFS.* Nodo encargado de ofrecer los directorios compartidos entre el *Frontend* y los *Workers*. Necesario para la configuración y puesta en marcha del clúster de IPython.

Como se puede comprobar, en este caso no existe un nodo de base de datos independiente. Esto es debido a que IPython no ofrece directamente ningún servicio de registro de tareas como si hace SLURM. Sin embargo, el propio servicio IPController si que permite usar una pequeña base de datos local donde almacenar las peticiones de tareas y sus resultados [24].

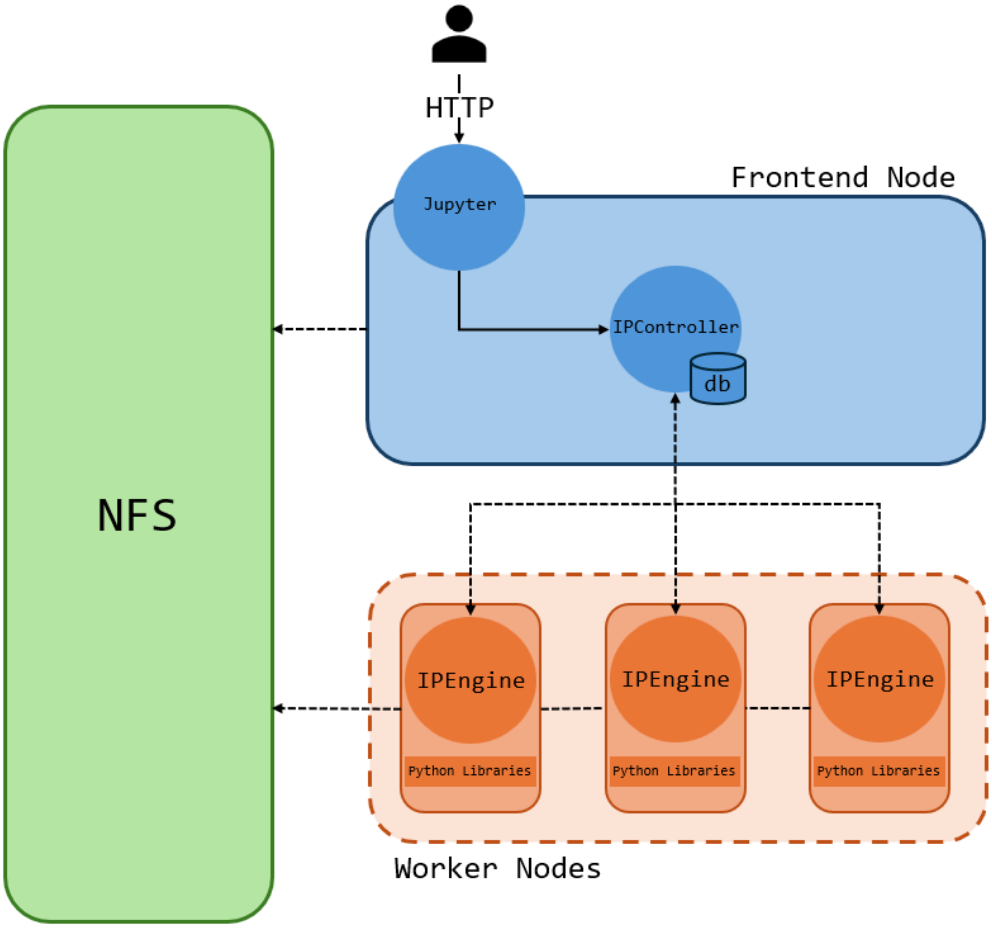


Figura 3.2: Arquitectura ideada para el servidor Jupyter con clúster de IPython.

3.2 Creación de imágenes

A lo largo de esta sección se detallarán los pasos a seguir para la creación de las imágenes para ambas soluciones. Durante este proceso, se ha intentado seguir todo lo posible las buenas prácticas que Docker recomienda en su página web [5].

Por ejemplo, si bien cada nodo especificado en la Sección 3.1 tendrá su propia imagen, la mayoría de estas parten de muchas herramientas en común. Por lo tanto para las dos soluciones se ha creado previamente una imagen con todos aquellos elementos que comparten y que servirá como base para posteriormente especificar cada nodo con otra nueva imagen.

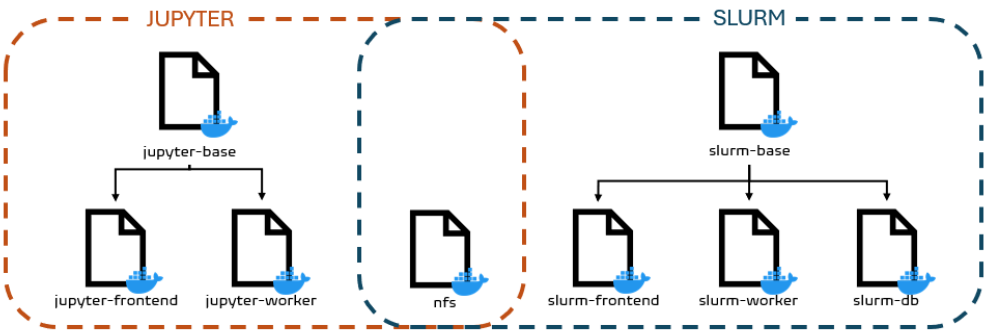


Figura 3.3: Imágenes a crear. Cada solución parte de una imagen base común, y ambas comparten la imagen del NFS.

3.2.1 NFS

Debido a que el nodo NFS es igual para ambas soluciones, se ha optado por describir la creación de su imagen por separado.

Para ello, partiremos de una imagen Debian donde instalaremos las herramientas necesarias para crear un servidor NFS, crearemos el directorio a compartir, y lo expondremos incluyéndolo en el archivo `/etc/exports`.

Por último, incluiremos un *script* de inicio donde se lanzarán todos los servicios necesarios. El servicio creado usará la versión 4, de este modo reducimos el número de servicios que hay que desplegar.

- `rpc.nfsd`. Servicio encargado de manejar las solicitudes de los usuarios.
- `rpc.mountd`. Servicio encargado de manejar las solicitudes de montaje.

3.2.2 SLURM

Los archivos Dockerfile se encuentran en su repositorio, en `SLURM/docker/dockerfile`.

Para la instalación de los nodos de SLURM se ha seguido la guía de instalación ofrecida en su documentación [50].

En primer lugar crearemos la imagen base de SLURM. Esta parte de una imagen Debian que instala las herramientas necesarias para montar un directorio NFS y lanzar un servidor SSH, permitiendo que los usuarios se conecten.

A continuación, se instala MUNGE [33], un *software* necesario para la autenticación. Esta herramienta utiliza una clave que debe ser la misma en todos los nodos involucrados en la comunicación. Dado que las demás imágenes partirán de esta misma, bastará con crear la clave aquí únicamente.

Posteriormente se procederá a instalar SLURM. Existen varios métodos para ello, pero en nuestro caso haremos uso de los *Debian Packages*, pues es la opción recomendada por sus desarrolladores.

SLURM ofrece una serie de *plugins* muy útiles. Su instalación se realiza automáticamente si en el momento de instalar SLURM contamos con los paquetes necesarios. Así, en nuestro caso haremos uso de los siguientes:

- `MUNGE`. SLURM requiere de instalar obligatoriamente un *software* de autenticación. Como alternativas ofrece esta herramienta ya mencionada anteriormente, o la opción menos recomendada `auth/Slurm`.
- `task/cgroup`. Un *plugin* muy útil para limitar los recursos que se pueden asignar a una tarea.
- `accounting_storage/as_mysql`. Requerido para llevar un registro de los trabajos ejecutados en nuestro sistema en una base de datos MySQL.

Del mismo modo, es necesario instalar previamente la especificación de MPI a usar, en este caso OpenMPI.

Finalmente se instala SLURM siguiendo sus instrucciones de instalación, y se incluye el archivo de configuración `slurm.conf` que deberá ser compartido por todos los nodos. Este fichero es crucial para que todos los nodos puedan comunicarse entre sí, pues en él se especifican, entre otras cosas, los nombres de cada uno de ellos. Posteriormente hablaremos más en profundidad sobre dicho archivo.

En este punto, cada nodo utilizará esta imagen recién creada para instalar unos paquetes en función de su finalidad:

- *Frontend*. `slurm-smd`, `slurm-smd-slurmctld` y `slurm-smd-client`.
- *Workers*. `slurm-smd`, `slurm-smd-slurmd` y `slurm-smd-client`.
- *DB*. `slurm-smd`, `slurm-smd-slurmdbd`. Además este nodo instalará un cliente de MySQL para acceder al servidor que especifiquemos en la variable de entorno `MYSQL_ADDRESS`.

Por último contarán con un *script* de inicio para la puesta en marcha de sus servicios. Para una mayor seguridad, se ha hecho uso de `gosu` [13]. Este *software* realiza la misma función que otros comandos como `sudo` o `su` utilizados para ejecutar comandos como otro usuario, en lugar del típico `root`. Sin embargo, `gosu` está optimizado para funcionar correctamente dentro de contenedores.

3.2.3 Jupyter

Los archivos Dockerfile pueden ser consultados en su repositorio, en `Jupyter/docker/dockerfile`.

La instalación de Jupyter y el clúster de IPython es mucho más sencilla. En primer lugar crearemos la imagen base, que, al igual que la imagen de SLURM, incluirá las herramientas para montar un directorio de un NFS y para permitir conexiones SSH de los usuarios.

A continuación instalaremos Python junto con IPyParallel y otras librerías[1] comunes mediante un archivo `requirements.txt`. Sin embargo, el contenido de este archivo se podría modificar en un futuro para satisfacer las demandas de los usuarios.

Por último crearemos el usuario `jupyter` con los permisos justos para ejecutar los servicios, y creamos una variable de entorno llamada `IPCONTROLLER_JSON_DIR` para especificar el directorio donde se almacenarán una serie de archivos JSON generados en tiempo real por el `IPController`.

Estos archivos son fundamentales para la comunicación entre el `IPEngine` y el `IPController`, puesto que contienen la información necesaria para su comunicación, como la IP, los puertos o las claves.
Una puesta en marcha manual requeriría de copiar dicho fichero desde el *Frontend* a cada *Worker*. Gracias al uso de un NFS, podemos especificar un directorio común en estos archivos de configuración para automatizar todo este proceso [52].

Una vez tenemos la imagen base, cada nodo especifica su función realizando los pasos necesarios.

- *Frontend*. Instalará el servidor Jupyter que expondrá a los usuarios y establecerá mediante la variable de entorno `CONFIG_DIR` la ubicación del archivo de configuración del `IPController` y de JupyterHub. Finalmente ejecutará un *script* de inicio para poner en marcha los servicios pertinentes.

- *Workers*. Establecerá mediante la variable de entorno `CONFIG_DIR` la ubicación del archivo de configuración del `IPEngine` y ejecutará un *script* de inicio para poner en marcha los servicios pertinentes.

3.3 Kubernetes y Helm

Una vez construidas las imágenes que usarán nuestros nodos, es hora de materializar la arquitectura ideada en la Sección 3.1 con Kubernetes y Helm. Para ello se ha creado un *chart* para cada uno de los sistemas, que contienen los ficheros necesarios para automatizar el despliegue de los mismos.

[1] El uso de `-break-system-packages`, puede parecer muy agresivo pero es realmente inofensivo. Por defecto, Debian intenta proteger su sistema de mezclar paquetes `apt` con `pip`, por lo que te "obliga" a realizar estas instalaciones en entornos vitrtuales (`venv`). Debido a que un contenedor es ya de por sí un entorno virtual, esto es inútil, así que usaremos esta *flag* para evitarlo.

3.3.1 SLURM

El *chart* de SLURM puede ser accedido en su repositorio, en `SLURM/k8s/`.

En la Figura 3.4 puede observarse una representación esquemática de la arquitectura que se va a construir desde el punto de vista de Kubernetes. Así, podemos observar de manera general los distintos objetos que se utilizarán, algunos de ellos realmente opcionales (en rojo). Como se puede ver, los nodos *Frontend*, *Worker*, y opcionalmente el *DB*, montarán volúmenes a partir de recursos del NFS y de varios archivos de configuración

En cualquier caso, en las siguientes secciones especificaremos cada uno de los nodos mostrados.

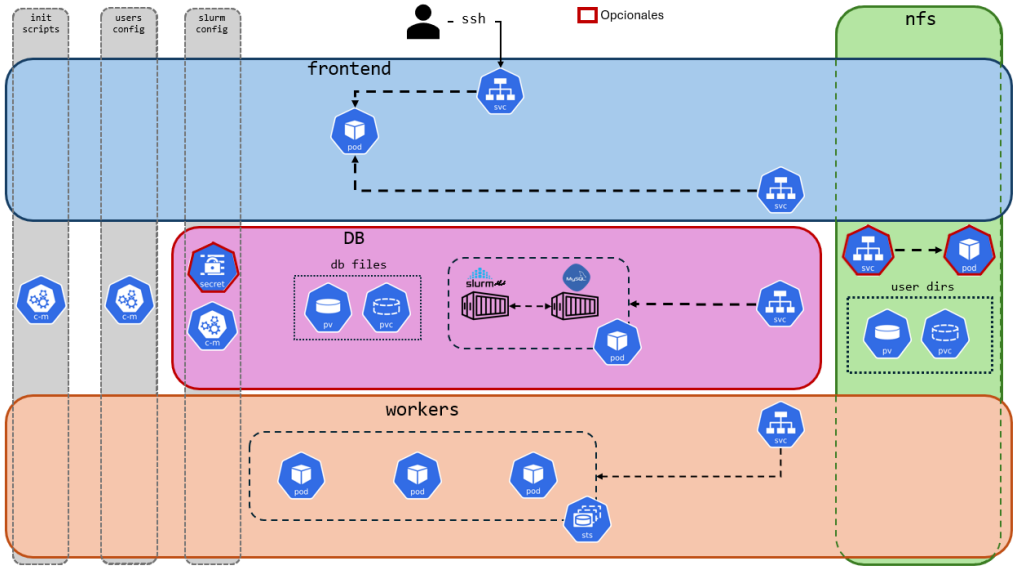


Figura 3.4: Arquitectura del clúster SLURM a construir en Kubernetes.

NFS

Como ya se comentó en secciones anteriores, el servidor NFS es fundamental para el funcionamiento del sistema. En este caso, su función principal será la de ofrecer los directorios compartidos para los distintos usuarios que vayan a hacer uso del clúster de colas SLURM.

Para ello, se crearán tantos `PersistentVolume` y `PersistentVolumeClaim` como usuarios se hayan especificado en el archivo `values.yaml` de Helm.

Un `PersistentVolume` [39] en Kubernetes representa un volumen en una máquina que almacena información de manera persistente. Por otro lado, un `PersistentVolumeClaim` es una petición para usar un `PersistentVolume` específico. Posteriormente los `Pod` pueden hacer uso de estas solicitudes para montar los volúmenes necesarios.
Debido a que el ciclo de vida de los `PersistentVolume` es independiente del ciclo de vida de los `Pod`, esta configuración permite conservar datos a través de diferentes ciclos de vida de un `Pod`.

Para nuestra solución ideada, ofrecemos dos posibilidades para incluir el servicio NFS:

- **Remoto**. La forma más clásica donde hacemos uso de un servidor NFS ya existente. Para ello, se define mediante valores de Helm la IP del servidor (`nfs.server`) y la ruta base (`nfs.basePath`) donde se encontrarán los directorios necesarios.

- **Local**. Mediante el valor `nfs.useLocal` puedes seleccionar la opción de crear localmente un contenedor con la imagen NFS que mencionamos anteriormente. Esta opción es posiblemente más rápida y requiere menos configuración externa, pero al montar el servicio en un contenedor, hace que su contenido se pierda una vez este se elimine. Además, su uso requiere otorgar ciertos permisos al contenedor, como `SYS_ADMIN` y `SETCAP`, por lo tanto su aplicación esta más orientado al *testing*.

En cualquiera de los casos, como se mencionó anteriormente, se creará un `PersistentVolume` para cada usuario especificado, el cual buscará directorios `/user_home_dir_<user_index>` en el servidor proporcionado. Por lo tanto, el servicio NFS deberá tener estos directorios disponibles en su volumen exportado.

En el caso de que se haya decidido usar un NFS local, se creará también los siguientes elementos:

- `Pod`. Encargado de correr el contenedor con la imagen NFS creada previamente. Como se puede observar, este `Pod` cuenta con un `initContainer`[2], que estará encargado de crear los directorios `/user_home_dir_<user_index>` para el NFS automáticamente.

- `Service`. Servicio encargado de exponer el puerto del NFS (2049) al resto de nodos.

Archivos de configuración

Durante la sección 3.2.2 se mencionó que todos los nodos debían compartir un archivo `slurm.conf` para el correcto funcionamiento del sistema. Si bien este mismo archivo ya se incluyó en cada imagen creada, a continuación vamos a remplazarlo mediante el uso de `ConfigMap`.

Un `ConfigMap` [6] en Kubernetes permite almacenar información no encriptada en forma de pares clave-valor. Estos datos pueden ser posteriormente inyectados dentro de un `Pod`, ofreciendo una manera eficiente de gestionar la configuración del sistema.
De este modo, podemos desacoplar este tipo de datos (como es `slurm.conf`) del proceso de creación de imágenes. De manera que nuestra solución se vuelve mucho más portable.

El primero de estos archivos lo encontramos en el `template` llamado `slurm-config.yaml`, y es, como acabamos de mencionar, el `slurm.conf`. Este archivo contiene toda la configuración necesaria para a puesta en marcha de SLURM. Entre sus parámetros más importantes destacan los siguientes:

- `SlurmctldHost`. Establece el nombre de *host* del nodo que ejecuta el demonio `slurmctld`.

- `SlurmctldPort` y `SlurmdPort`. Definen los puertos en los que cada demonio estará escuchando.

- `AccountigStorageX`. Serie de parámetros responsables de gestionar la contabilidad del sistema, como el nombre del *host* que ejecute `slurmdbd` en `AccountingStorageHost`.

[2]Los `initContainer` [22] son contenedores que se ejecutan antes que aquellos otros especificados en `containers`. Cabe recordar que estos contenedores, al encontrarse en el mismo `Pod`, comparten recursos entre sí, por lo que son ideales para realizar alguna tarea de configuración previa.

- `NodeName` y `NodeHostname`. Especifican los nodos del clúster y sus nombres de *host*. Todos parten de un nombre base y un índice que índica el número de réplica.

El resto de parámetros se pueden ajustar en función de los requisitos que tenga nuestro clúster, pero si se ha hecho especial énfasis en estos valores, es porque son fundamentales para el funcionamiento del sistema. Cambiar alguno de ellos sin conocimiento previo puede hace que el clúster deje de funcionar. Es por ello que estos parámetros son establecidos automáticamente mediante `values` de Helm, asegurando que cualquier modificación se refleje en todos los aspectos del sistema donde también es necesario realizar cambios.

Asimismo, otros *plugins* de SLURM como el de `task/cgroup` y `accounting_storage/as_mysql` tienen sus propios archivos de configuración (que podrán encontrarse en el mismo `template`). Este último merece una mención especial, ya que requiere la inclusión de una contraseña, para lo cual se ha utilizado otro objeto de Kubernetes: los `Secret` [45]. Estos elementos son similares a los `ConfigMap` mencionados anteriormente, pero están diseñados para manejar información sensible.

Los `ConfigMap` también han tenido gran protagonismo en la gestión de usuarios para el clúster. Dentro del `template users.yaml`, pueden encontrarse cuatro ficheros imprescindibles para gestionar usuarios en Linux [58]:

- `/etc/passwd`. Contiene información sobre los usuarios del sistema.
- `/etc/shadow`. Contiene las contraseñas de los usuarios del sistema.
- `/etc/group`. Contiene información sobre los grupos de usuarios del sistema.
- `/etc/gshadow`. Contiene las contraseñas de los grupos de usuarios del sistema.

Dentro de estos ficheros se incorporarán automáticamente los usuarios que se hayan especificado como `values` de Helm, y al desplegar el clúster estos serán compartidos por todos los nodos.

Gestionar estos ficheros manualmente es peligroso y no recomendado. Además, fijar estos archivos desde un `ConfigMap` puede suponer problemas si en un futuro se modifican las imágenes y estas incluyen nuevos usuarios. Sin embargo, cuando se evaluaron las diferentes posibilidades para incluir usuarios en el clúster, esta solución surgió como la más óptima.

Por último, los *scripts* de inicio también han sido redefinidos mediante `ConfigMap` en `startup.yaml`, con el fin de tener algo más de control sobre el lanzamiento de los servicios.

Base de datos

Como se explicó en la Sección 2.2.2, SLURM ofrece la posibilidad de incluir un demonio de base de datos para la contabilidad del sistema. En nuestro caso, hemos configurado las `templates` para que mediante el valor de Helm `db.enabled`, sea posible hacer uso o no de una de esta opción. El despliegue de la misma trae consigo algunas ventajas como se comentó, pero también requiere de algún tiempo extra para su puesta en marcha.

Su despliegue viene descrito en `db.yaml` y consta de tres elementos.

En primer lugar, el par `PersistentVolume` y `PersistentVolumeClaim`, ya descrito anteriormente, para la persistencia de la base de datos.

En segundo lugar, un `Pod` que, en primera instancia, ejecutará un `initContainer` para establecer los permisos del archivo de ejecución `slurmdbd.conf` necesarios[3]. A continuación se lanzarán dos contenedores siguiendo el patrón *sidecar* ya mencionado en la Sección 2.1.2.

- `slurmdbd`. Contenedor con la imagen creada anteriormente y que despliega de demonio de base de datos de SLURM.
- `mysql`. Contenedor que despliega una base de datos MySQL.

Al principio, durante el desarrollo del proyecto, incluimos los dos servicios en una misma imagen. Sin embargo, al final nos dimos cuenta que desacoplar estos servicios en dos contenedores distintos nos proporcionaba muchas ventajas. Por ejemplo, nos ayuda a aligerar el peso y la carga de trabajo de cada contenedor, o permite el desarrollo de cada uno de los servicios independientemente.

Además, como ya se mencionó durante la Sección 3.1.1, el contenedor `slurdmdbd` hace uso de una variable de entorno para seleccionar la base de datos a la que conectarse. En este caso, como ambos contenedores se encuentran en el mismo `Pod`, basta con especificar la IP `127.0.0.1`. Sin embargo, en el caso de que en un futuro quisiéramos hacer uso de otra base de datos (por ejemplo con algún servicio en la nube), bastaría con cambiar dicho valor.

Otro aspecto que podríamos tener en cuenta de cara al futuro, es sustituir el contenedor de `mysql` por un `Deployment` que nos ofrezca algo más de redundancia. En este caso, no se ha realizado porque la carga de trabajo a la que es sometida esta base de datos no es demasiado grande y no se ha visto necesidad en ningún momento.

En cualquier caso, ambos contenedores hacen uso del `Secret` de la contraseña de la base de datos que ya mencionamos en la sección anterior a través de una variable de entorno.

Por último, se lanzará un `Service` para recibir las comunicaciones con el demonio `slurmdbd` en los puertos donde escuche.

Frontend

El *Frontend* es una parte fundamental del sistema. Se encarga de coordinar los *Workers*, y será el punto de entrada a los usuarios que se conecten mediante SSH para usar el sistema.

Este se ha modelado mediante un `Pod` que, en primer lugar, ejecuta una serie de `initContainer`.

- `wait-for-nfs`. En caso de que se haya especificado un NFS local con `nfs.useLocal`, este contenedor esperará a que dicho servicio esté disponible.
- `wait-for-db`. En caso de que se haya activado la contabilidad mediante una base de datos con `db.enabled`, este contenedor esperará a que dicho servicio esté disponible.

[3]Desde la versión 1.9.6 de Kubernetes, los volúmenes montados a partir de un `ConfigMap` son de solo lectura y no es posible cambiar directamente sus permisos. Por lo tanto, la solución alternativa es lanzar un `initContainer` que cree una copia de dicho fichero en un volumen compartido con el contenedor definido en `container`. Esta información es omitida en la documentación, pero ha sido activamente debatida aquí: `https://github.com/kubernetes/kubernetes/issues/62099#issuecomment-378809922`.

- `change-perm-users-dir`. En Kubernetes, los volúmenes que montas heredan los permisos que tenían en su lugar de montaje. Por lo tanto, si se especifican usuarios y no se usa un NFS local, este contenedor se encargará de que cada directorio `/home` tenga los permisos adecuados.
  Esto se gestiona automáticamente al lanzar un NFS local, por eso no es necesario en dicho caso.

A continuación lanzará el contenedor con la imagen creada anteriormente. Este monta los archivos de configuración y volúmenes de usuarios que ya se mencionaron en secciones anteriores.

Por último se crean dos `Service`, uno para el acceso al nodo mediante SSH, y otro para las comunicaciones con el demonio `slurmctld`. Nota como el nombre de este último `Service` es directamente el nombre de *host* de nuestro *Frontend*. Esto es debido a que cuando los *Workers* quieran comunicarse con el *Frontend*, buscarán directamente su nombre en el archivo `Slurm.conf`. Por lo tanto, es necesario que este `Service` tenga su mismo nombre de *host* para poder ser accedido por los *Workers*[4].

Workers

Los *Workers* son responsables de ejecutar los trabajos en paralelo, por lo que es esencial que puedan ser múltiples y cuenten con la capacidad de escalar y desescalar según sea necesario. Por lo tanto, su modelado debe realizarse mediante un `Deployment`, como se explicó en la Sección 2.1.2, o mediante un `StatefulSet` [53].

Ambos objetos permiten lanzar varios `Pod` con las mismas especificaciones. La diferencia entre ellos radica en que los `StatefulSet` mantienen una identidad persistente para cada uno de los `Pod` que generan. Es decir, mientras que los `Deployment` asignan una cadena de texto aleatoria a los `Pod`, los `StatefulSet` los numeran secuencialmente, permitiéndonos conocer de antemano cuál será su nombre. Además, este recurso garantiza que, en caso de desescalado, los `Pod` se eliminarán en orden descendente.

Como se mencionó en la Sección 3.3.1, el archivo `Slurm.conf` debe contener exactamente los nombres de los *hosts* que conformarán el clúster. Por lo tanto, dadas las ventajas que ofrece el `StatefulSet`, queda claro que es el recurso ideal para nuestro sistema.

Este objeto creará tantos `Pod` como el valor establecido en `workers.numReplicas`. En cuanto a la especificación de cada `Pod` dentro del `StatefulSet`, no hay nada nuevo que no se haya mencionado previamente. En primer lugar, se utiliza un `initContainer` para esperar a que el *Frontend* esté disponible. Una vez que esté listo, se crearán los contenedores utilizando las imágenes previamente generadas, los cuales montarán los mismos `ConfigMap` utilizados en el *Frontend*.

Finalmente, se creará un `Service` para gestionar las comunicaciones con el *Worker* en el puerto especificado.

[4]Se podría pensar que bastaría con poner el nombre del `Service` en `Slurm.conf`. Sin embargo el demonio `slurmctld` fallará si ve que el nombre del *host* donde es lanzado no concuerda con el especificado en `Slurm.conf`.

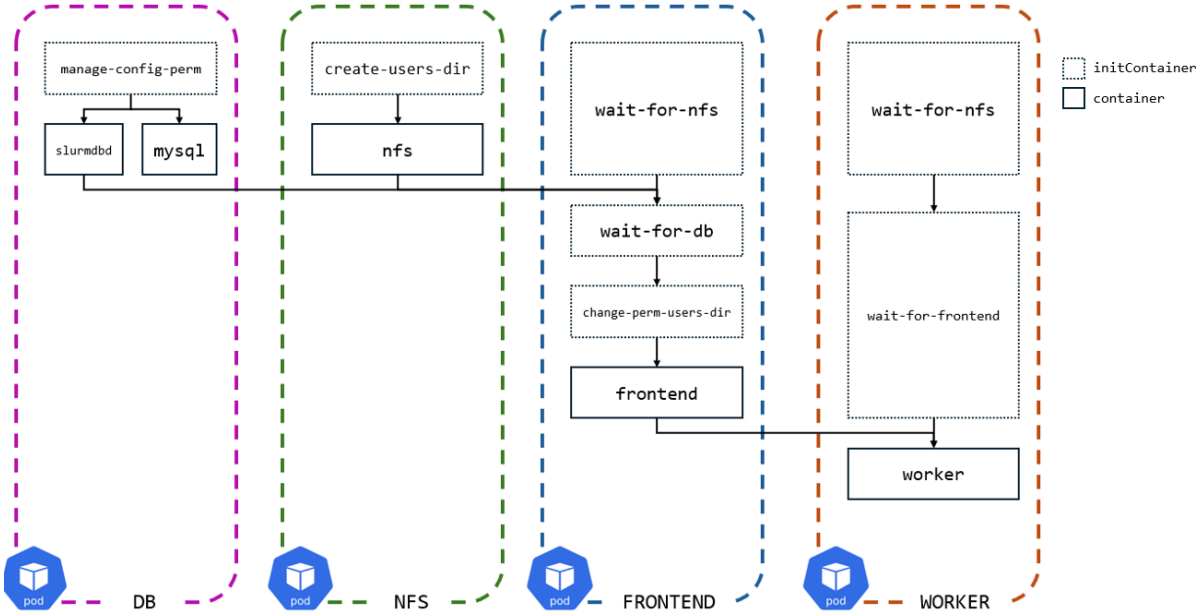


Figura 3.5: Orden de inicialización de cada `Pod` del clúster de SLURM.

3.3.2 Jupyter

El *chart* de Jupyter puede ser accedido en su repositorio, en `JUPYTER/k8s/`.

En la Figura 3.6 puede observarse una representación esquemática de la arquitectura que se va a construir desde el punto de vista de Kubernetes. Así, podemos observar de manera general los distintos objetos que se utilizarán, algunos de ellos realmente opcionales. En este caso la arquitectura es algo más sencilla debido a que la base de datos que mencionábamos que usa el `IPController` se lanza localmente dentro del contenedor con SQLite [51], y por lo tanto no necesitamos de un nodo que la gestione. Como se puede ver, los nodos *Frontend* y *Worker* siguen montando volúmenes a partir de recursos del NFS y de varios archivos de configuración

En cualquier caso, en las siguientes secciones especificaremos cada uno de los nodos mostrados.

NFS

De la misma manera que ya se mencionó en la solución de SLURM, el servidor NFS se encarga de ofrecer los directorios compartidos para los distintos usuarios que vayan a usar el clúster.

Sin embargo, en este caso tienen una finalidad adicional. Debido a que tanto el `IPController` como los `IPEngine` deben compartir un archivo JSON con la información necesaria para la comunicación, este servicio exportará a través de un `PersistentVolume` (y su respectivo `PersistentVolumeClaim`) un directorio común para ambos servicios. De este modo, los nodos especificarán este mismo directorio en sus archivos de configuración y la puesta en marcha será automática

En cualquier otro caso, la creación de este NFS es similar al ya mencionado para la anterior solución. Se permite el uso tanto de un NFS remoto como local. En este último caso, se

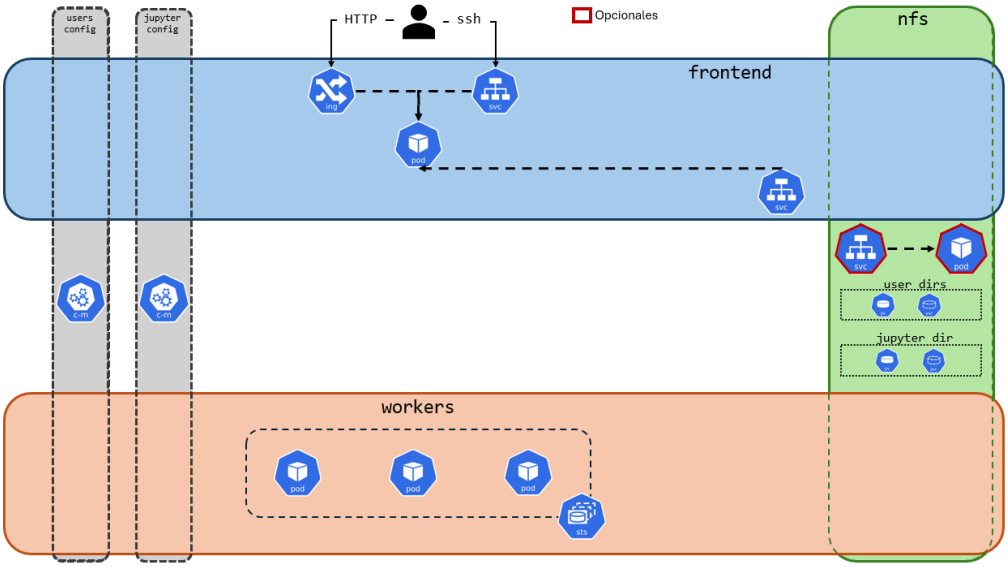


Figura 3.6: Arquitectura del clúster Jupyter a construir en Kubernetes.

crearía un `Pod` con la imagen descrita en secciones anteriores, junto con un `Service` encargado de exponer el puerto del servicio.

Archivos de configuración

Tanto Jupyter como los servicios de IPyParallel, ofrecen archivos de configuración con una gran variedad de opciones. Para automatizar la personalización de estos servicios, se han utilizado `ConfigMap` y los `values` de Helm. A continuación, se describen algunos de estos archivos de configuración:

- `ipcontroller_config.py`. Archivo que configura aspectos el servicio `IPController`, como sus archivos de *log*, sus tiempos para conexión, puertos y direcciones IP usados para la comunicación, o localización del fichero JSON compartido con los `IPEngine`.
- `ipengine_config.py`. Archivo que configura aspectos del servicio `IPEngine`. En este caso no ofrece mucha personalización, pero es importante para especificar la localización del archivo JSON que tanta veces hemos mencionado.
- `jupyterhub_config.py`. Archivo que configura el servidor de Jupyter. En él se especifica, entre otras cosas, los usuarios que podrán acceder al sistema.

Por otro lado, y del mismo modo que se hizo en SLURM, se han hecho uso de `ConfigMap` para la gestión de usuarios dentro del clúster. En la Sección 3.3.1 es posible recordar cómo se ha llevado ha cabo.

Frontend

El *Frontend* de esta solución se encarga de exponer el servidor de Jupyter a través de HTTP, y actúa como controlador para el resto de *Workers* al trabajar paralelamente con IPyParallel.

Esta se ha modelado medianta un `Pod` que ejecutará en primer lugar una serie de `initContainer`.

- `wait-for-nfs`. En caso de que se haya especificado un NFS local con `nfs.useLocal`, este contenedor esperará que dicho servicio esté disponible.
- `change-perm-jupyter`. Como ya se explicó, debido a que los volúmenes montados heredan los permisos que tenían en su lugar de montaje, haremos uso de este contenedor para asegurarnos que el directorio compartido para los archivos JSON tengan los permisos adecuados.
- `change-perm-users-dirs`. Por la misma razón expuesta anteriormente, en caso de que el NFS no sea local, haremos lo mismo con los directorios de los usuarios del clúster.

Finalmente, se lanzará el contenedor con la imagen previamente creada, que montará los volúmenes y `ConfigMap` necesarios para la configuración de los servicios en los directorios especificados mediante las variables de entorno establecidas al crear la imagen.

Además, se creará un `Service` para exponer cada uno de los puertos utilizados por los servicios y, por último, un `Ingress` para permitir el acceso HTTP al servicio de Jupyter. Opcionalmente, también hemos creado un `Service` para permitir el acceso SSH al *Frontend*, para el caso en que se prefiera a trabajar directamente sobre CLI.

Workers

Los *Workers* se encargarán de ejecutar paralelamente el código enviado por el `IPController` del *Frontend*. Por lo tanto, es importante que deban ser varios y que cuenten con capacidad de escalar o desescalar según se estime. Al hablar de los *Workers* en SLURM, mencionamos la diferencia entre `Deployment` y `StatefulSet`, y como estos últimos se ajustaban más a dicho caso de uso. En este caso, sin embargo, no necesitamos que los *Workers* mantengan ninguna identidad. Es por ello que nos bastará con usar un `Deployment` para su creación.

Este `Deployment` creará tantos `Pod` como réplicas se hayan establecido en `workers.numReplicas`. Y los `Pod` esperarán mediante `initContainer` a que *Frontend* y NFS estén disponibles. Acabada este espera, ejecutarán un contenedor con la imagen ya mencionada, y con los `ConfigMap` y volúmenes necesarios para la puesta en marcha de sus servicios.

Como se puede apreciar, en este caso no es necesario ningún `Service`. Esto es debido a como se producen las comunicaciones entre `IPController` y `IPEngine`.

El `IPController` publica en su archivo JSON los puertos donde trabaja, y serán los distintos `IPEngine` los encargados de iniciar la comunicación con él. El `IPController` abrirá entonces una serie de *sockets* de ZeroMQ donde publicará las tareas. De este modo, los *Workers* no necesitan exponer ningún puerto, puesto que el *Frontend* nunca intentará comunicarse con ellos directamente.

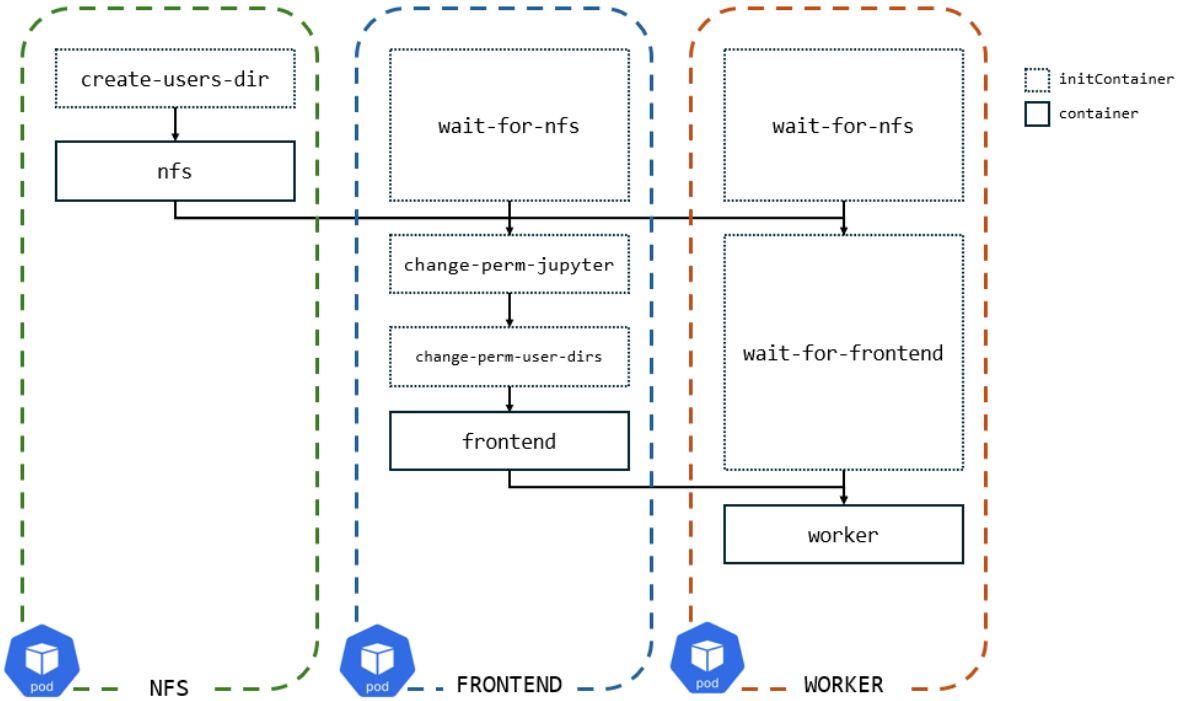


Figura 3.7: Orden de inicialización de cada `Pod` del clúster de Jupyter.

Capítulo 4

Validación

En este capítulo se realizará una validación de los sistemas construidos. En primer lugar se explicará como realizar la puesta en marcha de los clústeres y se mostrará un ejemplo muy básico. Finalmente se realizará un benchmark con algo más de complejidad para intentar llevar los sistemas al límite y estudiar su rendimiento.

Todas las ejecuciones que se van a mostrar a continuación, han sido realizadas sobre el entorno de Minikube [61]. Esta herramienta te permite ejecutar un clúster de Kubernetes (véase Sección 2.1.2) localmente sobre tu propia máquina personal.

Mi ordenador personal cuenta con un procesador Intel Core i7-9750H CPU @ 2.60GHz con 6 núcleos físicos con *multithreading* (lo que equivalen a 12 núcleos lógicos), y 15GiB de memoria. Desplegaré Minikube usando Docker como driver en este ordenador, asignándole 10 núcleos lógicos y 10GiB de memoria.

4.1 SLURM

4.1.1 Puesta en marcha

Para la puesta en marcha y desinstalación del clúster, se ha proporcionado un `Makefile` para una mejor automatización. De este modo, bastará con la ejecución de los comandos mostrados en el Listado 4.1.

```
make install
make uninstall
make template # valida los templates de helm
```

Listing 4.1: Comandos para lanzar el clúster de SLURM.

El tiempo de instalación puede variar de unos pocos segundos hasta varios minutos en función de los valores usados. Pero tras la instalación, se mostrará en pantalla un texto con información útil para los encargados de gestionar el clúster, como información sobre cómo añadir usuarios en el sistema o como conectarse al mismo mediante SSH.

Una vez todos los `Pod` hayan iniciado, la salida del comando `kubectl get pod` se verá como se muestra en el Listado 4.2.

```
NAME                        READY   STATUS    RESTARTS   AGE
slurm-cluster-db            1/1     Running   0          9m10s
slurm-cluster-frontend      1/1     Running   0          9m10s
slurm-cluster-lin-0         1/1     Running   0          9m7s
slurm-cluster-lin-1         1/1     Running   0          2m25s
slurm-cluster-lin-2         1/1     Running   0          78s
slurm-cluster-nfs           1/1     Running   0          9m10s
```

Listing 4.2: Salida del comando `kubectl get pod` para un cluster con 3 *workers*, un NFS local y un nodo con base de datos.

Una vez dentro del *Frontend*, podemos comprobar que nuestra partición está lista con el comando `sinfo`, tal y como se muestra en el Listado 4.3.

```
test@slurm-cluster-frontend-hn:~$ sinfo
PARTITION    AVAIL  NODES  STATE NODELIST
mypartition*    up      3   idle slurm-cluster-lin-[0-2]
```

Listing 4.3: Salida del comando `sinfo` simplificado.

Podemos probar a lanzar un trabajo con MPI a todos los nodos. El código, muy simple, puede encontrarse en el Listado 4.4, y deberá compilarse previamente.

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    printf("Hola Mundo desde el procesador %s,
            rango %d de %d procesos\n",
           processor_name, world_rank, world_size);

    MPI_Finalize();

    return 0;
}
```

Listing 4.4: Código en C a ejecutar por los *Workers*.

Preparamos un *script* que lance tres tareas, una por nodo, y guarde el resultado en el archivo `/home/test/test_job_mpi.out`.

```
#!/bin/bash
#SBATCH --job-name=test_job_mpi
#SBATCH --output=/home/test/test_job_mpi.out
#SBATCH --ntasks=3 --ntasks-per-node=1
#SBATCH --time=00:01:00

srun /home/test/hello
```

Listing 4.5: *Script* para el lanzamiento de la tarea. Utilizaremos el directorio `home` del usuario `test` para recoger el resultado de la tarea.

Por último lanzamos el *script* con `sbatch` y monitoreamos el trabajo en la cola con `squeue`. Una vez listo el resultado, estará disponible donde indicamos en el *script*.

```
>> mpicc /home/test/hello.c -o /home/test/hello
>> sbatch job.sh
Submitted batch job 10
>> squeue
JOBID PARTITION     NAME     USER ST       TIME  NODES
    10 mypartiti test_job     test  R       0:03      3
>> squeue
JOBID PARTITION     NAME     USER ST       TIME  NODES
>> cat test_job_mpi.out
Hola Mundo desde el procesador slurm-cluster-lin-2, rango 2 de 3
Hola Mundo desde el procesador slurm-cluster-lin-0, rango 0 de 3
Hola Mundo desde el procesador slurm-cluster-lin-1, rango 1 de 3
```

Listing 4.6: Lanzamiento del trabajo. Primeramente se compila el código en un directorio común, a continuación lanzamos el *script* y lo monitoreamos con `squeue`. Por último mostramos el resultado.

En el caso en que lanzáramos también la base de datos, podremos comprobar los trabajos lanzados con `sacct`.

```
test@slurm-cluster-frontend-hn:~$ sacct
JobID           JobName  Partition AllocCPUS      State
------------ ---------- ---------- ---------- ----------
6              hostname mypartiti+      2       COMPLETED
6.0            hostname                 2       COMPLETED
7              hostname mypartiti+      6       COMPLETED
7.0            hostname                 6       COMPLETED
10          test_job_+ mypartiti+      3       COMPLETED
10.batch         batch                 1       COMPLETED
10.0             hello                 3       COMPLETED
```

Listing 4.7: Salida del comando `sacct`.

Por último, es posible escalar y desescalar el clúster manualmente con el comando: `helm upgrade <chart_name> <dir> -set workers.numReplicas=<int>`. Automáticamente el *Frontend* detectará los cambios pertinentes y los aplicará automáticamente sin que los usuarios noten ninguna interrupción.

4.1.2 High-Performance Linpack (HPL)

El Top500 [57] es una lista que clasifica anualmente los 500 supercomputadores más potentes del mundo. Para ello, se utiliza un *benchmark* de la familia LINPACK, que se encarga de resolver sistemas de ecuaciones lineales densos. Este *software* no busca un número mágico que determine la potencia general del sistema, ya que sería imposible de obtener, sino que refleja la capacidad de un sistema para resolver este tipo de problemas, que son extremadamente comunes en la ingeniería. Estas operaciones se realizan en coma flotante, por lo que sus resultados se miden en Floating Point Operations Per Second (FLOPS).

Dentro de todos los *benchmarks* de LINPACK, el más utilizado es HPL [19], ya que está específicamente diseñado para ser probado en sistemas paralelos con múltiples nodos y permite probar diferentes tamaños `n` en las matrices.

Este *benchmark* requiere la instalación de ciertos paquetes adicionales (además de tener instalada una implementación de MPI). Por lo tanto, siguiendo las instrucciones de este enlace [12], se ha creado una nueva versión de la imagen base de Debian con el *software* instalado (puede accederse en `SLURM/docker/dockerfile/benchmark`).

Así, es posible activar un valor de Helm (`benchmark`) para que el clúster que despleguemos haga uso de estas imágenes con el *benchmark* instalado, de manera que en cualquier momento sea posible realizar este *test*.

Para la ejecución de este *benchmark* es necesario aportar una serie de valores para ajustar el algoritmo a la arquitectura disponible. A contiuación se explican los más importantes [18].

El tamaño del problema: `N`

`N` representa la cantidad de filas y columnas que tendrá la matriz a resolver. Por lo tanto es un valor crucial que influirá directamente en el resultado del *benchmark*.

Podríamos calcular teóricamente el tamaño de `N` con la siguiente fórmula, donde `MemoryUse` es el porcentaje de memoria que queremos destinar.

$$N = \sqrt{\frac{MemoryUse * MemorySizeinBytes}{\texttt{sizeof(double)}}}$$

Lo ideal sería usar entre un 80 o 90 porciento, por lo que en nuestro sistema podríamos usar teóricamente:

$$10GiB = 10 * 2^{30} bytes$$
$$N = \sqrt{\frac{0,8 * 10 * 2^{30}}{8}} = 32,768$$

Sin embargo, debido a que todo esto se está realizando sobre el entorno de Minikube corriendo en mi propio ordenador personal, en la práctica, usar un tamaño mayor a 19000 lo vuelve muy inestable y propenso a errores por parte de Minikube. Lo más probable es que en mi caso, el mayor porcentaje de uso que pueda llegar a probar esté alrededor del 50 porciento.

El tamaño de bloque: `NB`

`NB` representa el tamaño en que se va a dividir la matriz para ser repartida entre todos los nodos. Un `NB` más pequeño favorece el balanceo de carga entre nodos, por lo que es recomendable en sistemas con muchos nodos. Sin embargo, si usamos un dato demasiado pequeño, podemos limitar el rendimiento del sistema al no utilizar al máximo el tamaño de la memoria caché.

En mi caso, mi ordenador cuenta con la siguiente memoria caché.

- L1d Cache (datos): 192 KiB (sumando 6 instancias de 32 KiB cada una)
- L1i Cache (instrucciones): 192 KiB (sumando 6 instancias de 32 KiB cada una)
- L2 Cache: 1.5 MiB (sumando 6 instancias de 256 KiB cada una)
- L3 Cache: 12 MiB (compartida entre todos los núcleos)

Debido a que este clúster estará repartido entre todos los procesadores de mi ordenador, podríamos intentar seleccionar un tamaño de bloque que quepa dentro de mi caché de nivel 2, puesto que cada procesador físico tiene una.

$$NB^2 * 8 \leq 262144 \ \ bytes$$
$$NB^2 \leq 32768$$
$$NB \leq \sqrt{32768} \approx 181$$

Para comprobar cual es finalmente el `NB` ideal, se ha hecho una comparación con varios tipos de bloque: 64, 96, 128, 181, 192 y 256. La Figura 4.1 muestra los resultados obtenidos en un clúster con 5 *workers*.

Como se puede observar, los tamaños de bloques más grandes, dan mejores resultados. Aunque se ha comprobado que estos son también más propensas a generar errores.

Malla de procesos `PxQ`

Este parámetro determina la disposición de los procesos en una malla bidireccional. `P` es el número de procesos en las filas, mientras que `Q` en las columnas.

Este valor normalmente depende de las conexiones físicas que se han hecho entre los nodos. Por lo tanto en nuestro caso estudiaremos con 5 *workers* y 2 CPUs cada uno, cual de las dos formaciones nos ofrece mejores resultados: `2x5` o `1x10`

Como se puede observar en la Figura 4.2, independientemente del `NB` o `N`, desplegar los *workers* en diferentes columnas nos ofrece mucho mejor resultado.

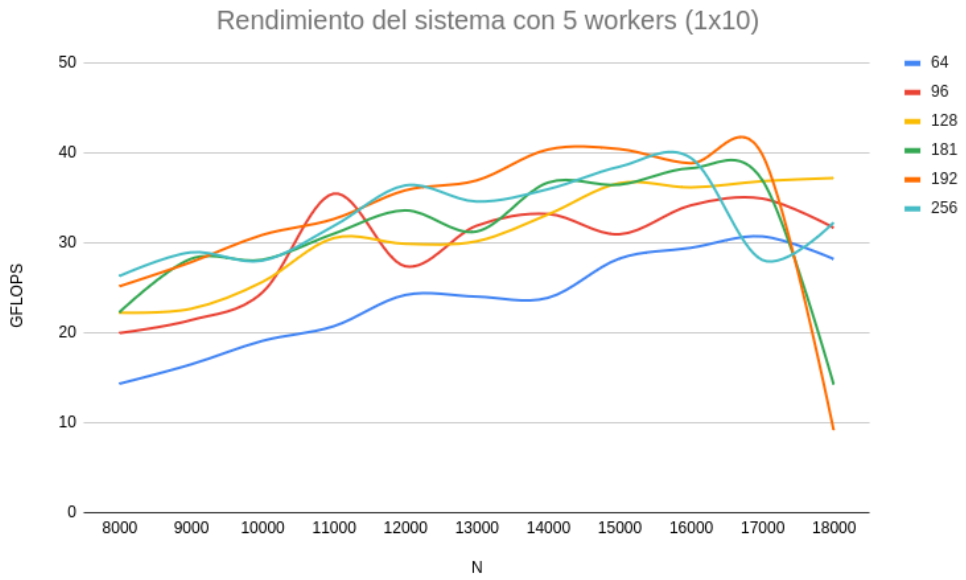
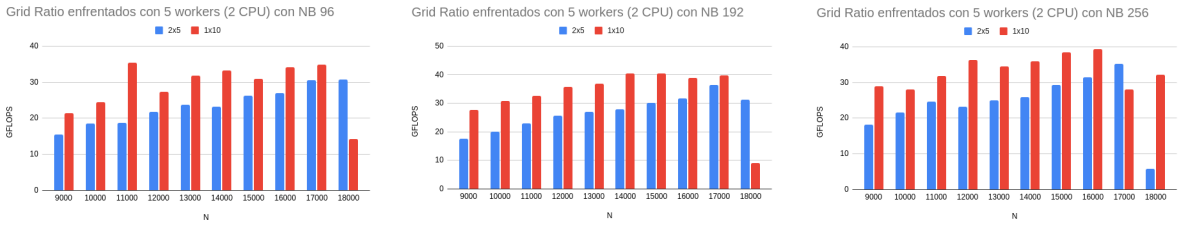


Figura 4.1: Comparación de diferentes `NB` a lo largo de diferentes tamaños de problema. Los tamaños más grandes dan mejores resultados, pero en mi caso generan errores cuando el tamaño del problema es muy grande.



(a) Tamaño de bloque 96. **(b)** Tamaño de bloque 192. **(c)** Tamaño de bloque 256.

Figura 4.2: Comparación de diferentes *grid ratio* para diferentes `NB`. En mi caso, las formaciones en fila (`rojo`) dan mucho mejor resultado que las cuadradas (`azul`)

.

Resultados finales

Con los datos obtenidos, se ha realizado un *benchmark* para ver cuales son los FLOPS máximos que podemos alcanzar, teniendo como resultado 50,41 GFLOPS.

Desgraciadamente no entramos dentro del Top 500 con nuestro clúster. Sin embargo, hay que tener en cuenta que este se ha encontrado sujeto a muchas condiciones, entre ellas el hecho de correr en un servicio contenerizado sobre un portátil. Estos sistemas no son del todo estables, puesto que sus procesadores tienen modos *Turbo* donde aumentan temporalmente su frecuencia, y la disminuyen cuando la temperatura aumenta demasiado.

4.2 Jupyter

4.2.1 Puesta en marcha

De la misma manera que se hizo en la Sección 4.1.1, se ha proporcionado un `Makefile` para la puesta en marcha y desinstalación del clúster. Alternativamente, también se pueden usar directamente los comandos Helm expuestos en la Sección 2.1.3.

El tiempo de instalación puede variar en función de los valores usados, y tras su instalación se mostrará en pantalla una pequeña información sobre como conectarse al sistema.

Una vez el `chart` haya sido instalado, la salida del comando `kubectl get pod` se verá como se muestra en el Listado 4.8

```
NAME                                READY   STATUS    RESTARTS   AGE
jupyter-cluster-frontend            1/1     Running   0          31s
jupyter-cluster-nfs                 1/1     Running   0          31s
jupyter-cluster-worker-cfdh747      1/1     Running   0          28s
jupyter-cluster-worker-cfdg131      1/1     Running   0          27s
jupyter-cluster-worker-cfde589      1/1     Running   0          27s
```

Listing 4.8: Salida del comando `kubectl get pod` para un cluster de IPython con 3 *workers* y un NFS local.

Para acceder al sistema deberemos conocer la IP del `Ingress` a través del cual accederemos al servidor web de Jupyter [1]. En este caso, hemos establecido el puerto para acceder a este servicio en `80`. Por lo tanto únicamente será necesario acceder a dicha web desde cualquier navegador, como se muestra en la Figura 4.3.

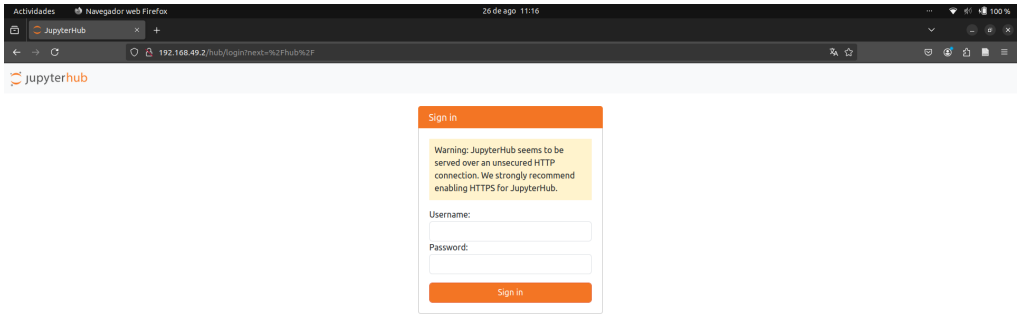


Figura 4.3: Pantalla de inicio de sesión al acceder al servidor de Jupyter.

[1] Es posible conocer dicha IP a través del comando `kubectl get ingress jupyter-cluster-frontend-ingress -o jsonpath='.status.loadBalancer.ingress[0].ip'`

Para iniciar sesión, podemos usar cualquiera de los usuarios establecidos en el archivo `values.yaml`. Automáticamente, JupyterHub creará un entorno para dicho usuario, como se puede observar en la Figura 4.4. Este entorno tendrá de directorio de trabajo la carpeta `/home/<usuario>`,y te permitirá crear varios tipos de archivos como Notebooks para los *kernel* instalados, o incluso abrir un terminal.

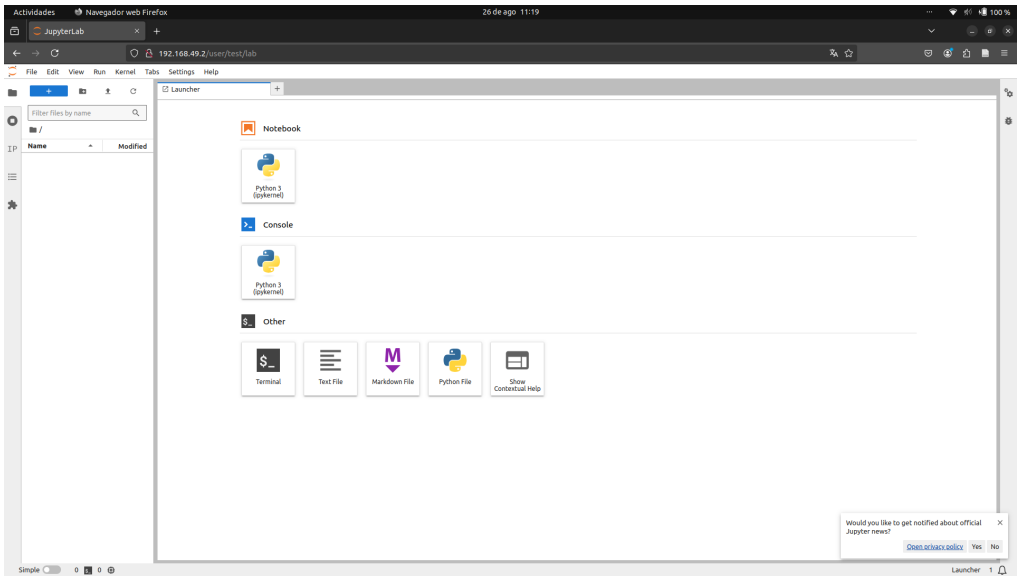


Figura 4.4: Entorno creado para un usuario `test` recién inicado sesión.

En cuanto a IPyParallel, lo ideal es hacer uso del clúster montado. Para ello deberemos crear un objeto `Client` a partir del fichero JSON que se encuentra en `/home/jupyter/jupyter_shared/ipcontroller-client.json`. A partir de este momento, podemos utilizar el clúster para paralelizar la ejecución de código, tal y como se muestra en la Figura 4.5 [2].

Alternativamente, también es posible crear clústeres locales desde nuestro entorno de JupyterLab a través de la pestaña con el icono `IP`. Sin embargo, hay que tener en cuenta que estos clústeres se crean localmente en el *Frontend*, por lo que no se recomienda abusar de ellos para no saturar dicho nodo.

Por último, es posible escalar y desescalar el clúster manualmente con el comando: `helm upgrade <chart_name> <dir> -set workers.numReplicas=<int>`. Automáticamente el *Frontend* detectará los cambios pertinentes y los aplicará automáticamente sin que los usuarios noten ninguna interrupción.

[2]Este trabajo no pretende ahondar en la API de IPyParallel, para obtener más información sobre todo lo que puede hacer con esta herramienta, es recomendable dirigirse a su documentación: [37].

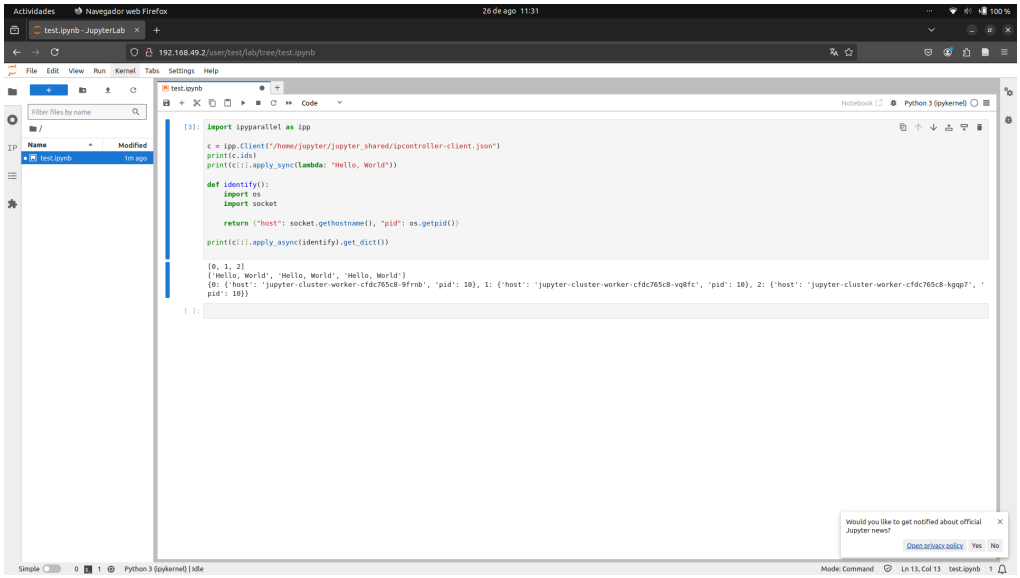


Figura 4.5: Prueba de ejecución múltiple de código en Python haciendo uso de nuestro clúster.

4.2.2 Benchmark: El Fractal de Mandelbrot

El conjunto de Mandelbrot es un fractal famoso que se forma iterando una función matemática simple en el plano complejo. Para determinar si un punto en el plano pertenece al conjunto, se aplica la función repetidamente sobre dicho punto. Si, al iterar la función, el valor se mantiene dentro de un rango específico después de muchas iteraciones, entonces el punto pertenece al conjunto de Mandelbrot. Si el valor se aleja demasiado (crece sin límite) en un número finito de iteraciones, el punto no pertenece al conjunto [11].

Calcular el conjunto de Mandelbrot no es tarea sencilla, sobre todo con una alta precisión y a gran escala. Es por ello que nos aprovecharemos del clúster que acabamos de construir para realizar paralelamente un cálculo del fractal de Mandelbrot en ASCII (véase Listado 4.9).

```
def mandel(width, height, first, last):
    minX = -2.0
    maxX = 1.0
    aspectRatio = 2

    chars = " .,-:;i+hHM$*#@ "

    yScale = (maxX - minX)*(float(height)/width)*aspectRatio

    fractal = ""
    for y in range(first, last):
        line = ""
        for x in range(width):
            c = complex(minX+x*(maxX-minX)/width,
                        y*yScale/height-yScale/2)
            z = c
            for char in chars:
                if abs(z) > 2:
```

```
                    break
                z = z * z + c
            line += char
        fractal += line + "\n"

    return fractal
```

Listing 4.9: Algoritmo que genera el fractal a partir de las medidas dadas.

Para ello, hemos dividido la ejecución de `mandel` en `n` tareas, y hemos hecho uso de `map_sync` para dividir dichas tareas entre todos nodos *Workers*. Es posible comprobar más a fondo el código resultante en el repositorio del proyecto, en `JUPYTER/test/mandel.py`.

Por lo tanto, realizamos diversas ejecuciones variando tanto el tamaño del problema como el número de *workers* con el fin de comprobar que, independientemente del tamaño del problema, aumentar el número de *workers* mejora el rendimiento del algoritmo.

En la Figura 4.6 se muestran los tiempos de ejecución obtenidos. Como se puede observar, aumentar el número de *workers* mejora notablemente el rendimiento del algoritmo, sin importar el tamaño del problema. Además, esta mejora es más significativa a medida que aumenta el tamaño del problema. Sin embargo, en el entorno en el que estamos trabajando, la ganancia en rendimiento se vuelve prácticamente imperceptible a partir de 20 *workers*.

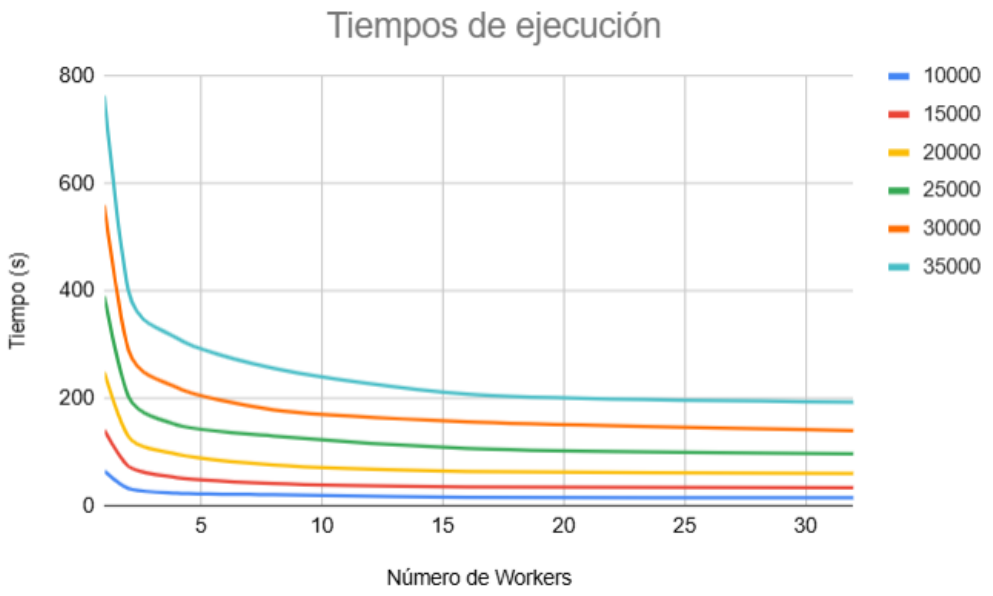


Figura 4.6: Tiempos de ejecución del algoritmo con distintos *Workers* y tamaño del problema.

A continuación mostraremos también cuales han sido el *Speedup* y Eficiencia resultantes de nuestro algoritmo paralelizado.

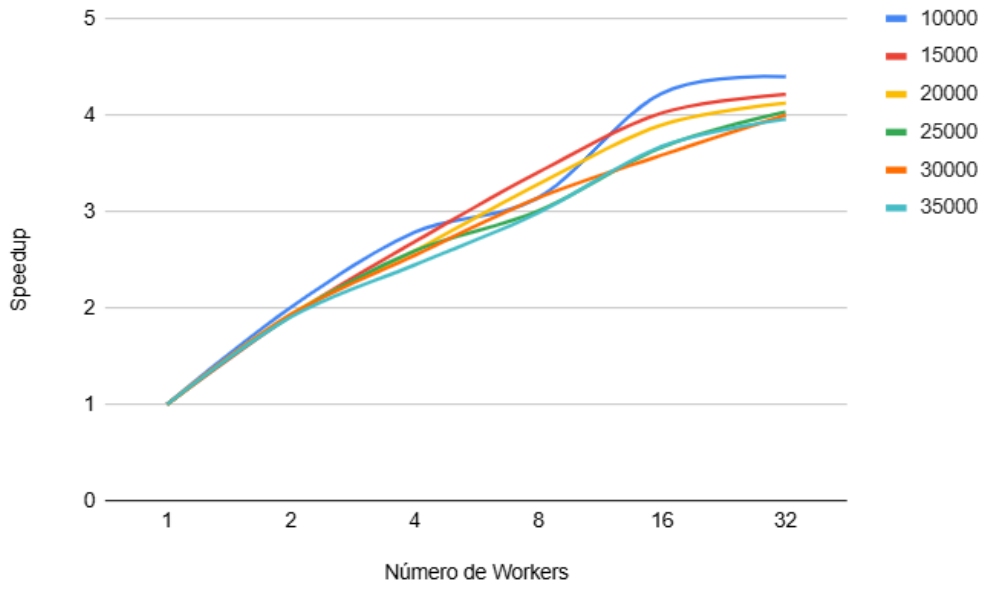
El *Speedup* mide la mejora en tiempo de ejecución del algoritmo paralelizado en comparación con su versión secuencial. Se calcula utilizando la siguiente fórmula.

$$Speedup = T_{serie}/T_{paralelo}$$

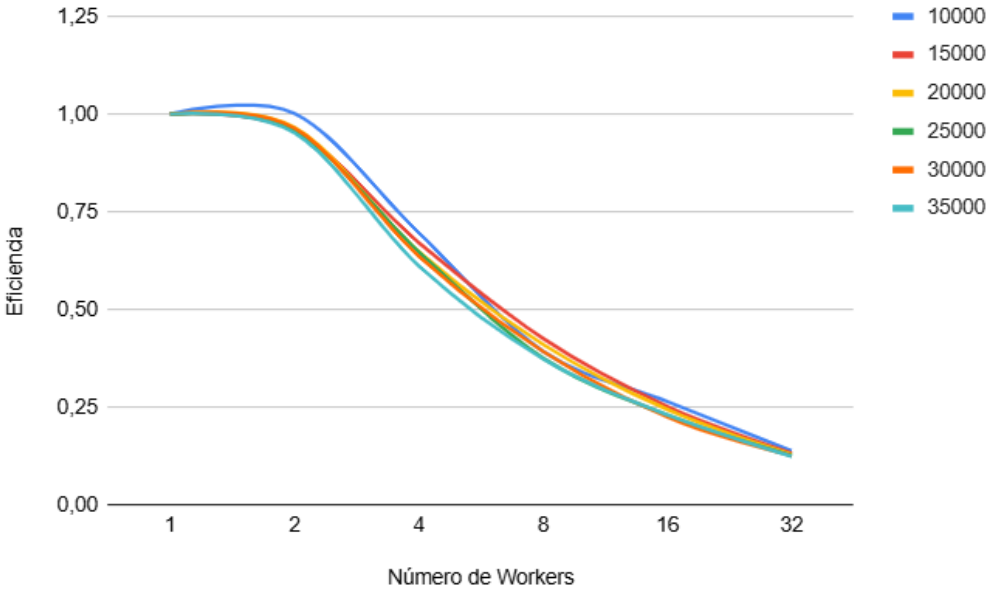
Por otro lado, la Eficiencia refleja qué tan bien se están aprovechando los recursos de procesamiento. Se obtiene comparando el *Speedup* con el número de nodos de procesamiento utilizados, tal como se muestra a continuación.

$$Eficiencia = Speedup/N_{nodos}$$

En la Figura 4.7 se presentan los resultados del *Speedup* y la Eficiencia obtenidos.



(a) *Speedup* del algoritmo paralelizado.



(b) Eficiencia del algoritmo paralelizado.

Figura 4.7: Eficiencia y *Speedup* obtenidos con los tiempos de la Figura 4.6.

Capítulo 5

Conclusiones

En este capitulo se van a presentar las conclusiones sobre el trabajo realizado. En primer lugar se hará una visión general del proyecto y a continuación se presentará los objetivos cumplidos y su relación dentro del marco de estudios. Finalmente se incluirá una sección sobre posible trabajos futuros sobre el proyecto.

La virtualización es una tecnología poderosa que permite desplegar una amplia gama de entornos en cualquier tipo de computadora, incluidas las de uso personal, como en este proyecto. Gracias a ella, hemos podido desarrollar dos soluciones de computación intensiva que pueden ser fácilmente implementadas en otros sistemas, independientemente de su capacidad, facilitando así el trabajo de una comunidad de usuarios que, aunque no esté familiarizada con la creación de entornos de cómputo avanzados, depende de ellos de manera constante.

Este logro ha sido posible gracias a Kubernetes y Helm, dos herramientas que se complementan perfectamente para el despliegue de entornos virtuales complejos, adaptables a configuraciones específicas.

En general, el resultado del proyecto se puede considerar satisfactorio. La planificación inicial fue precisa, lo que nos permitió completar cada sprint dentro del tiempo estimado y añadir funcionalidades adicionales, como la integración de un servidor NFS contenerizado.

De este modo, se han logrado alcanzar todos los objetivos establecidos en la Sección 1.2.

5.1 Cumplimiento de los objetivos

A continuación se realizará una revisión de los objetivos que nos marcamos al inicio del proyecto, desarrollando los *Sprints* en los que se ha abordado cada objetivo, y en que secciones de esta documentación son abordados.

Aprender a gestionar el despliegue de soluciones en Kubernetes con Helm

Durante el *Sprint* 1, se realizó un análisis intensivo sobre la herramienta Helm a través del despliegue de varias soluciones de distinta complejidad. Finalmente, este conocimiento se aplicó en el *Sprint* 4, donde desplegamos las arquitecturas construidas con Kubernetes en Helm.

Los resultados obtenidos, se pueden comprobar en la Sección 3.3, donde se explicaba la construcción de las soluciones con Kubernetes. Al hablar de estas, mencionábamos la importancia que tenia Helm en este sentido, pues nos permitirá facilitar el despliegue de todos los objetos y facilitar su personalización a través de los `values`.

Definir una arquitectura orientada a la computación intensiva

Durante el *Sprint* 2, fue necesario realizar un análisis a fondo sobre las herramientas de computación intensiva a utilizar durante el proyecto, con el fin de entender su funcionamiento, y así poder configurar los nodos que formarán parte de cada clúster. Finalmente, en el *Sprint 3*, estas imágenes y conocimientos fueron materializados en los objetos de Kubernetes que crearían nuestras arquitecturas de cómputo intensivo.

De este modo, la Sección 3.1 muestra las arquitecturas que finalmente se han definido, tras tener en cuenta todo lo llevado a cabo en los *Sprint* mencionados.

Automatizar lo máximo posible el despliegue de los sistemas ideados

Como ya se ha mencionado varias veces durante este documento, Helm es una herramienta muy importante para la automatización. Por lo tanto, el *Sprint* 4 ha sido clave para alcanzar este objetivo. Gracias a la capacidad de introducir valores en los manifiestos de Kubernetes, hemos sido capaces de automatizar y personalizar el despliegue de estas soluciones.

A lo largo de la Sección 3.3 se han ido explicando varios de los `values` que hemos especificado en nuestro proyecto para ayudarnos a automatizar el despliegue de cada sistema. Finalmente, en la Sección 4 se ha mostrado lo fácil que es desplegar estos sistemas, donde con solo un comando podemos disponer de sistemas que hace años requerían de configuraciones mucho más complejas.

Virtualizar el servicio NFS a usar por los nodos

Tras haber realizado con éxito cada *Sprint* en su tiempo estimado, se propuso añadir un nuevo objetivo para contenerizar este servicio. Por lo tanto el *Sprint* 6 de este proyecto estuvo dedicado completamente a la inclusión de este servicio que, sorprendentemente, ha dado muy buenos resultados y es una pieza clave a la hora de facilitar la creación de este tipo de entornos de cómputo intensivo.

Finalmente, a lo largo de la Sección 3, hemos abordado la contenerización de este servicio, tanto de la creación de su imagen como su materialización con Kubernetes. Finalmente, con Helm hemos brindado la oportunidad a los usuarios de utilizar o no este servicio simplemente con la especificación de un `value`.

Implementar y validar varias soluciones para la ejecución de trabajos de computación intensiva

Tras desplegar con éxito las dos soluciones planteadas, se realizó un último *Sprint* encargado de realizar una validación de estos clústeres. Por lo tanto, se propuso por un lado realizar el *benchmark* HPL, puesto que es muy utilizado en clústeres reales; y por otro lado, la paralización de un algoritmo de cierta complejidad como es el cálculo del conjunto de Mandelbrot.

Los resultados de ambas pruebas fueron descritos recientemente en la Sección 4.

Ampliar los conocimientos sobre las herramientas de virtualización de Docker y Kubernetes

Por último, todos los *sprints* llevados a cabo en el proyecto han sido fundamentales para mejorar nuestro conocimiento en dos herramientas de virtualización vistas durante el Máster: Docker y Kubernetes.

Este era uno de los objetivos principales al proponer el proyecto, y se ha cumplido con creces. A través del diseño de arquitecturas en Kubernetes y la resolución de los problemas que surgieron, el alumno ha consolidado y ampliado la base adquirida sobre estas herramientas durante el curso.

5.2 Relación del proyecto con los estudios cursados

Los conocimientos obtenidos en el Máster Universitario en Computación en la Nube y de Altas Prestaciones han sido esenciales para el desarrollo de este proyecto. A continuación se describen algunas asignaturas que han sido relevantes para la consecución de este proyecto.

- *Plataformas de Gestión de Contenedores.* Este proyecto nació directamente de esta asignatura, pues a lo largo de la misma se abordaron dos de las herramientas de virtualización claves para el proyecto, Docker y Kubernetes. La visión inicial de este trabajo fue ahondar algo más en estas herramientas, con el fin de que el alumno ganara más experiencia con las mismas.

- *Configuración, Administración y Utilización de Clusters de Computadores.* Durante esta asignatura, se adquirió experiencia práctica en la creación y gestión de clusters de computadores, utilizando máquinas virtuales para configurar clusters con diversas finalidades. A lo largo del curso, se abordaron desafíos similares a los enfrentados en este proyecto, como la implementación del servicio NFS. Aunque en este caso el *modus operandi* es diferente al trabajar con Kubernetes, la experiencia adquirida fue valiosa para comprender el funcionamiento de estos servicios.

- *Conceptos de la computación en grid y cloud* y *Tecnología de la programación paralela.* Estas dos asignaturas han sido, en general, una buena base para entender como trabajar con tecnologías de computación intensiva. Algunas de ellas usadas durante este proyecto, ya sea para construir las infraestructuras o para realizar las pruebas.

5.3 Trabajos Futuros

A pesar de haber logrado los objetivos mencionados anteriormente, la falta de experiencia con las herramientas o el tiempo disponible para llevar a cabo el proyecto hace que aún existan varios aspectos en los que los sistemas pueden mejorar.

- *Seguridad.* Aunque el sistema cuenta con mecanismos de monitoreo y auditoría, la falta de conocimiento en este ámbito ha limitado la implementación de medidas de seguridad. Como resultado, algunos nodos, como el frontend, podrían ser vulnerables a ataques que ralenticen o incluso detengan el funcionamiento del sistema. Un posible futuro añadido sería limitar el tráfico dentro de los clústeres o incluir cifrado Transport Layer Security (TLS) en la solución con JupyterHub.

- *Sistema de usuarios.* Como se discutió en la Sección 3, los usuarios del sistema están sincronizados entre los nodos mediante la compartición de archivos pertinentes. Sin embargo, este enfoque presenta ciertos problemas ya mencionados. Por lo tanto, una mejora futura sería implementar un sistema de gestión de usuarios más robusto.

- *Despliegue y pruebas más exigentes.* Hasta ahora, el despliegue del sistema se ha realizado en un ordenador personal, lo que ha limitado las pruebas que se pueden realizar. Aprovechando los conocimientos adquiridos durante el Máster, podríamos intentar desplegar los sistemas en un entorno Cloud. Esto no solo brindaría varias ventajas, sino que también facilitaría aún más su uso por parte de los usuarios, que podrían desentenderse completamente de adquirir un computador con potencia.

- *Continuous Integration / Continuous Delivery (CI/CD).* En línea con el punto anterior, podríamos automatizar la construcción, pruebas y despliegue de nuestros sistemas utilizando los principios de CI/CD, vistos vagamente durante el Máster.

Bibliografía

[1] *Accounting and Resource Limits - Slurm Workload Manager*. https://slurm.schedmd.com/accounting.html[Accessed: 17/08/2024] (vid. pág. 24).

[2] Luz Estela Valencia Ayala y Eliécer Herrera Uribe. "Del manifiesto ágil sus valores y principios". En: (2007) (vid. págs. 3, 4).

[3] Dmytro Ageyev et al. "Classification of existing virtualization methods used in telecommunication networks". En: *2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT)*. 2018, págs. 83-86. DOI: 10.1109/DESSERT.2018.8409104 (vid. pág. 2).

[4] *Architecture - Jupyter Documentation*. https://docs.jupyter.org/en/latest/projects/architecture/content-architecture.html[Accessed: 16/08/2024] (vid. págs. 18, 19).

[5] *Building best practices | Docker Docs*. https://docs.docker.com/build/building/best-practices/[Accessed: 20/08/2024] (vid. pág. 25).

[6] *ConfigMaps | Kubernetes*. https://kubernetes.io/docs/concepts/configuration/configmap/[Accessed: 22/08/2024] (vid. pág. 30).

[7] *Deployments | Kubernetes*. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/[Accessed: 13/09/2024] (vid. pág. 14).

[8] *Docker Compose overview | Docker Docs*. https://docs.docker.com/compose/[Accessed: 22/08/2024] (vid. pág. 13).

[9] *Docker: Accelerated Container Application Development*. https://www.docker.com/ [Accessed: 13/08/2024] (vid. pág. 3).

[10] *Dockerfile reference*. https://docs.docker.com/reference/dockerfile/ [Accessed: 14/08/2024] (vid. pág. 12).

[11] *El conjunto de Mandelbrot*. https://complex-analysis.com/contenido/conjunto_de_mandelbrot.html[Accessed: 26/08/2024] (vid. pág. 47).

[12] Mathieu Gaillard. *How to compile HPL LINPACK on Ubuntu 22.04*. https://www.mgaillard.fr/2022/08/27/benchmark-with-hpl.html[Accessed: 28/08/2024] (vid. pág. 42).

[13] *gosu*. https://github.com/tianon/gosu[Accessed: 20/08/2024] (vid. pág. 27).

[14] *HELM: The package manager for Kubernetes*. https://helm.sh/ [Accessed: 13/08/2024] (vid. pág. 3).

[15] *Helm/Charts*. https://helm.sh/docs/topics/charts/[Accessed: 16/08/2024] (vid. pág. 16).

[16] *Helm/Docs*. https://helm.sh/docs/helm/[Accessed: 16/08/2024] (vid. pág. 17).

[17] *Home - OpenMP*. https://www.openmp.org/[Accessed: 18/08/2024] (vid. pág. 20).

[18] *HPL Frequently Asked Questions*. https://www.netlib.org/benchmark/hpl/faqs.html[Accessed: 29/08/2024] (vid. pág. 42).

[19] *HPL-A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers*. https://www.netlib.org/benchmark/hpl/index.html[Accessed: 28/08/2024] (vid. pág. 42).

[20] *Ingress*. https://kubernetes.io/docs/concepts/services-networking/ingress/[Accessed: 15/08/2024] (vid. pág. 15).

[21] *Ingress*. https://kubernetes.io/docs/reference/kubernetes-api/workload-resources/[Accessed: 15/08/2024] (vid. pág. 15).

[22] *Init Contianer | Kubernetes*. https://kubernetes.io/docs/concepts/workloads/pods/init-containers/[Accessed: 22/08/2024] (vid. pág. 30).

[23] *IPython: Interactive Computing*. https://ipython.org/ [Accessed: 13/08/2024] (vid. pág. 3).

[24] *IPython's Task Database*. https://ipyparallel.readthedocs.io/en/stable/reference/db.html[Accessed: 18/08/2024] (vid. pág. 24).

[25] *Jobs | Kubernetes*. https://kubernetes.io/docs/concepts/workloads/controllers/job/[Accessed: 13/09/2024] (vid. pág. 14).

[26] Nectarios Koziris. "Fifty years of evolution in virtualization technologies: from the first IBM machines to modern hyperconverged infrastructures". En: *Proceedings of the 19th Panhellenic Conference on Informatics*. PCI '15. Athens, Greece: Association for Computing Machinery, 2015, págs. 3-4. ISBN: 9781450335515. DOI: 10.1145/2801948.2802039 (vid. pág. 1).

[27] *kubectl*. https://kubernetes.io/docs/reference/kubectl/kubectl/[Accessed: 15/08/2024] (vid. pág. 15).

[28] *Kubernetes*. https://kubernetes.io/es/ [Accessed: 13/08/2024] (vid. pág. 3).

[29] *Kubernetes Components*. https://kubernetes.io/docs/concepts/overview/components/ [Accessed: 15/08/2024] (vid. págs. 13, 14).

[30] *MPI: A Message-Passing Interface Standard*. Ver. 4.1. Message Passing Interface Forum, 2021 (vid. pág. 20).

[31] *MPICH | High-Performance Portable MPI*. https://www.mpich.org/[Accessed: 18/08/2024] (vid. pág. 21).

[32] Chrissy Kidd Muhammad Raza. *Virtual Machines (VMs) vs Containers: What's The Difference?* https://www.bmc.com/blogs/containers-vs-virtual-machines/ [Accessed: 13/08/2024]. 2020 (vid. pág. 2).

[33] *MUNGE by dun*. https://dun.github.io/munge/[Accessed: 20/08/2024] (vid. pág. 26).

[34] *MySQL*. https://www.mysql.com/[Accessed: 17/08/2024] (vid. pág. 23).

[35] *Nomad by HashiCorp*. https://www.nomadproject.io/[Accessed: 16/08/2024] (vid. pág. 13).

[36] *Open MPI: Open Source High Performance Computing*. https://www.open-mpi.org/[Accessed: 18/08/2024] (vid. pág. 21).

[37] *Overview and getting started - ipyparallel 8.9.0.dev documentation*. https://ipyparallel.readthedocs.io/en/latest/tutorial/intro.html[Accessed: 16/08/2024] (vid. págs. 19, 46).

[38] *Overview of Docker Hub*. https://docs.docker.com/docker-hub/ [Accessed: 14/08/2024] (vid. pág. 12).

[39] *Persistent Volumes | Kubernetes*. https://kubernetes.io/docs/concepts/storage/persistent-volumes/[Accessed: 22/08/2024] (vid. pág. 29).

[40] *Podman*. https://podman.io/ [Accessed: 14/08/2024] (vid. pág. 12).

[41] *Pods*. https://kubernetes.io/docs/concepts/workloads/pods/ [Accessed: 15/08/2024] (vid. pág. 14).

[42] *Project Jupyter Documentation*. https://docs.jupyter.org/en/latest/ [Accessed: 13/08/2024] (vid. págs. 3, 17).

[43] *Project Jupyter|JupyterHub*. https://jupyter.org/hub[Accessed: 29/08/2024] (vid. pág. 17).

[44] Scott Raynovich. *How Does A Docker Container Work?* https://www.sdxcentral.com/cloud/containers/definitions/what-is-docker-container/how-does-a-docker-container-work-explanation/ [Accessed: 14/08/2024]. 2019 (vid. pág. 12).

[45] *Secrets | Kubernetes*. https://kubernetes.io/docs/concepts/configuration/secret/[Accessed: 22/08/2024] (vid. pág. 31).

[46] *Services*. https://kubernetes.io/docs/concepts/services-networking/service/[Accessed: 15/08/2024] (vid. pág. 14).

[47] *Sidecar Containers*. https://kubernetes.io/docs/concepts/workloads/pods/sidecar-containers/ [Accessed: 15/08/2024] (vid. pág. 14).

[48] *Slurm Workload Manager - Documentation*. https://slurm.schedmd.com/documentation.html [Accessed: 13/08/2024] (vid. pág. 3).

[49] *Slurm Workload Manager - Overview*. https://slurm.schedmd.com/overview.html[Accessed: 17/08/2024] (vid. págs. 19-21).

[50] *Slurm Workload Manager - Quick Start Administrator Guide*. https://slurm.schedmd.com/quickstart_admin.html[Accessed: 17/08/2024] (vid. pág. 26).

[51] *SQLite Home Page*. https://www.sqlite.org/index.html[Accessed: 03/09/2024] (vid. pág. 34).

[52] *Starting the IPython controller and engines - IPython*. https://ipython.org/ipython-doc/2/parallel/parallel_process.html[Accessed: 20/08/2024] (vid. pág. 28).

[53] *StatefulSets | Kubernetes*. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/[Accessed: 23/08/2024] (vid. pág. 33).

[54] Andrew Stellman y Jennifer Greene. *Learning Agile*. O'Reilly Media, Inc., 2014 (vid. pág. 3).

[55] *Swarm mode overview - Docker Docs*. https://docs.docker.com/engine/swarm/[Accessed: 16/08/2024] (vid. pág. 13).

[56] *The Chart Template Developer's Guide*. https://helm.sh/docs/chart_template_guide/[Accessed: 16/08/2024] (vid. pág. 17).

[57] *Top500*. https://top500.org/[Accessed: 28/08/2024] (vid. pág. 42).

[58] *Users and Groups*. https://wiki.archlinux.org/title/Users_and_groups[Accessed: 22/08/2024] (vid. pág. 31).

[59] *Welcome to Flask — Flask Documentation (3.0.x)*. https://flask.palletsprojects.com/en/3.0.x/ [Accessed: 14/08/2024] (vid. pág. 11).

[60] *Welcome to Python.org*. https://www.python.org/ [Accessed: 14/08/2024] (vid. pág. 11).

[61] *Welcome!|Minikube*. https://minikube.sigs.k8s.io/docs/[Accessed: 27/08/2024] (vid. pág. 39).

[62] Karlijn Willems. *IPython Or Jupyter?* https://www.datacamp.com/blog/ipython-or-jupyter [Accessed: 16/08/2024]. 2017 (vid. pág. 17).

Apéndice A

Objetivos de Desarrollo Sostenible

A.1 Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS)

En 2015, la Organización de las Naciones Unidas aprobó una serie de objetivos globales con el fin de alcanzar un futuro más sostenible y próspero. Este plan, conocido como la Agenda 2030, establece 17 Objetivos de Desarrollo Sostenible (ODS) para promover el crecimiento económico, abordar las necesidades sociales y proteger el medio ambiente.

La Universitat Politècnica de València, comprometida con estos valores, insta a sus alumnos a reflexionar sobre la relación de sus Trabajos de Fin de Estudios con estos objetivos. Por lo tanto, a continuación se presenta un análisis de los ODS y su vínculo con el proyecto desarrollado en este Trabajo de Fin de Máster.

El ODS más relacionado con nuestro proyecto es el 9: "Industria, Innovación e Infraestructura". Esto es debido a que estamos ofreciendo una manera más accesible de crear infraestructura para el cómputo intensivo. Esto no solo impulsa la innovación tecnológica, sino que también promueve un crecimiento inclusivo, ya que cualquier institución, independientemente de su tamaño o presupuesto, puede implementar soluciones de cómputo intensivo para la investigación científica, la educación o el desarrollo tecnológico.

Por otro lado, durante este proyecto hemos demostrado la facilidad para desplegar infraestructuras de computación intensiva en cualquier tipo de entorno. Específicamente, una de las soluciones desarrolladas consistió en crear un servidor de JupyterHub con un clúster de IPython. Como ya se mencionó anteriormente, este tipo de entornos puede ser útil no solo para laboratorios de investigación, sino también para clases educativas. De esta manera, ofrecemos una opción más económica para que profesionales de la educación puedan hacer uso de herramientas de trabajo sin necesidad de grandes inversiones en infraestructura física. Esto democratiza el acceso a la computación y fomenta la innovación tanto en el ámbito académico como en el profesional, lo que está estrechamente relacionado con el ODS 4: "Educación de Calidad".

Por último, la computación intensiva se aplica en una amplia variedad de ámbitos que tienen un impacto significativo en el desarrollo sostenible. Gracias a esta tecnología, es posible realizar simulaciones científicas complejas que contribuyen, entre otras cosas, al tratamiento de enfermedades, la búsqueda de fuentes de energía más limpias y renovables, y la mitigación del cambio climático. Facilitar el acceso a estas herramientas a la comunidad científica está relacionado con los ODS 3, 7 y 13, que abarcan la salud y el bienestar, la energía asequible y no contaminante, y la acción por el clima, respectivamente.

Sin embargo, es importante tener en cuenta que estas herramientas son extremadamente consumidoras de energía. Un uso ineficaz o desmedido de las mismas puede tener un impacto negativo en el medio ambiente, aumentando el consumo de recursos y las emisiones de carbono.

Objetivos de Desarrollo Sostenible	**Alto**	**Medio**	**Bajo**	**No procede**
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**			X	
ODS 4. **Educación de calidad.**		X		
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**			X	
ODS 8. **Trabajo decente y crecimiento económico.**		X		
ODS 9. **Industria, innovación e infraestructuras.**	X			
ODS 10. **Reducción de las desigualdades.**				X
ODS 11. **Ciudades y comunidades sostenibles.**				X
ODS 12. **Producción y consumo responsables.**				X
ODS 13. **Acción por el clima.**			X	
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

Tabla A.1: Relación del trabajo realizado con los Objetivos de Desarrollo Sostenible (ODS).