UNIVERSIDAD EUROPEA DE MADRID

ESCUELA DE ARQUITECTURA, INGENIERÍA Y DISEÑO

MÁSTER UNIVERSITARIO EN ANÁLISIS DE DATOS MASIVOS

TRABAJO FIN DE MÁSTER

ARQUITECTURA ESCALABLE DE MICROSERVICIOS PARA PROBLEMAS NUMÉRICOS EN ENTORNOS WEB

JOSÉ ANTONIO MARTÍNEZ LÓPEZ

Dirigido por

Yudith Coromoto Cardinale

CURSO 2024 - 2025

TÍTULO: ARQUITECTURA ESCALABLE DE MICROSERVICIOS PARA PROBLEMAS NUMÉRICOS EN ENTORNOS WEB

AUTOR: JOSÉ ANTONIO MARTÍNEZ LÓPEZ

TITULACIÓN: MÁSTER UNIVERSITARIO EN ANÁLISIS DE DATOS MASIVOS

DIRECTOR DEL PROYECTO: Yudith Coromoto Cardinale

FECHA: Septiembre de 2025

RESUMEN

En diversos campos de la ingeniería, se requiere la resolución de problemas usando diversos métodos numéricos complejos, que requieren de gran potencial computacional. En este contexto, contar con una herramienta *online* que permita resolver tales problemas de ingeniería basado en los métodos numéricos más frecuentes, representa un apoyo en las actividades de ingeniería. Por ejemplo, en las primeras fases de diseño preliminar de soluciones o para usarlo en plataformas educativas para formar a estudiantes. Sin embargo, para poder ofrecer un servicio así, sin que el usuario final tenga una experiencia negativa causada por las demoras que pueden conllevar la resolución de algunos de estos problemas, es necesario disponer de una arquitectura particular que sea capaz de resolver múltiples problemas de manera distribuida y en tiempo cercano al tiempo real.

Las tecnologías del Big Data ofrecen herramientas para el desarrollo de plataformas distribuidas escalables y seguras que además sean accesibles de manera sencilla. Tal es el caso de las tecnologías de contenedores, bases de datos NoSQL y microservicios. Todas estas tecnologías han resultado útiles en el desarrollo de este proyecto para crear una arquitectura que pueda satisfacer las necesidades de la aplicación descrita anteriormente.

En ese sentido, en este trabajo se construye una arquitectura usando contenedores Docker con los distintos servicios que son necesarios. Los servicios que se *dockerizan* son la API, los trabajadores (que son los procesos que resuelven los problemas), la base de datos y el almacén de objetos. Todos estos contenedores están orquestados con Kubernetes. El objetivo principal de la orquestación es poder escalar el número de trabajadores. La arquitectura propuesta, y la implementación de un primer prototipo, demuestran la factibilidad y la utilidad de este tipo de herramientas en las primeras fases de diseño preliminar o para usarlo en plataformas educativas.

Palabras clave: Kubernetes, Docker, Python, Redis, Microservicios, NoSQL.

ABSTRACT

There are numerous fields of knowledge, such as Aerospace Engineering, in which great computational power is required for some of the most frequent problems. There are several applications to being able to solve these problems online such as in preliminary design or for educational purposes, such as teaching students. However, to be able to use a service like this it is necessary for users to not have a negative experience due to slow responses to their problems. It is necessary to have an architecture capable of solving these kinds of problems in a distributed manner and near real time.

The technologies associated with Big Data provide tools that enable the development of distributed scalable platform that are safe and easily accessible. Such is the case with containers, NoSQL database and microservices which are part of the Big Data technologies. All these were useful when carrying out this project and creating an architecture that can satisfy all the requirements listed above.

The architecture that is built in this project is one that uses Docker with necessary services. The services that are dockerized are the API, the workers, the database and the object storage. All of these containers are orchestrated with Kubernetes. The main objective of orchestration is to be able to scale the number of workers.

Keywords: Kubernetes, Docker, Python, Redis.

Índice general

1. INTRODUCCIÓN **7**
1.1. Planteamiento del problema . . . 7
1.2. Objetivos del proyecto . . . 8
1.2.1. Objetivos específicos . . . 8
1.2.2. Beneficios del proyecto . . . 8
1.3. Resultados obtenidos . . . 8

2. Marco Teórico y Tecnológico **10**
2.1. Marco Teórico . . . 10
2.2. Marco Tecnológico . . . 10

3. DESARROLLO DEL PROYECTO **13**
3.1. Metodología . . . 13
3.2. Diseño de la Solución: Análisis del Problema . . . 13
3.3. Diseño de la Solución: Partición del Problema . . . 14
3.4. Diseño de la Solución: Selección de tecnologías . . . 15
3.5. Primer Prototipo . . . 16
3.5.1. Paquete *api* . . . 16
3.5.2. Paquete *methods* . . . 19
3.5.3. Paquete *task_manager* . . . 20
3.5.4. Paquete *tests* . . . 22
3.6. Dockerización de *Worker* y *Publisher* . . . 23
3.7. Segundo Prototipo . . . 25
3.8. Adaptación de *Workers* a *Kubernetes* . . . 26
3.9. Pruebas Finales . . . 26
3.10. Proceso de Desarrollo . . . 28
3.11. Presupuesto . . . 32
3.12. Viabilidad . . . 32
3.13. Resultados del proyecto . . . 33

4. DISCUSIÓN **34**

5. CONCLUSIONES **36**
5.1. Conclusiones del trabajo . . . 36
5.2. Conclusiones personales . . . 36

6. FUTURAS LÍNEAS DE TRABAJO **37**

Índice de Figuras

3.1. Diagrama general de la arquitectura. 14
3.2. Foto de la estructura del proyecto. 17
3.3. Estado del clúster al iniciarse 27
3.4. Estado del clúster con 40 usuarios 27
3.5. Interfaz de Locust con cuarenta usuarios 28
3.6. Uso de Swagger para hacer pruebas. 29
3.7. Uso de Coverage. 30
3.8. Uso de Docker para desarrollo. 30

Índice de Tablas

3.1. Costes del proyecto 33

Capítulo 1. INTRODUCCIÓN

Los métodos numéricos complejos, que requieren de gran potencial computacional, pueden ser usados en diversos campos de la ingeniería. Para tener un apoyo en las actividades de ingeniería sería útil contar con una herramienta para la resolución de los métodos numéricos más frecuentes. Dicha herramienta sería útil en las primeras fases de diseño preliminar de soluciones, o para usarlo en plataformas educativas para formar a estudiantes. Disponer de una herramienta así requiere una arquitectura especial que pueda resolver múltiples problemas de manera distribuida y en tiempo cercano al tiempo real.

1.1 Planteamiento del problema

En este caso, el principal problema que se quiere abordar es la falta de una herramienta para usuarios (desarrolladores o estudiantes del área de ingeniería) que les permita usar desde un *Jupyter notebook*, o similar, métodos numéricos relacionados con su campo de estudio, en el área de ingeniería . Por ejemplo, la resolución de matrices mediante métodos numéricos es útil para resolver problemas relacionados con las cerchas [3], así como otros métodos numéricos más simples, como la búsqueda de raíces, son útiles en muchos otros contextos de la ingeniería. Para poder facilitar la interacción con estos recursos de resolución de problemas numéricos, es necesario que haya una interfaz accesible bien documentada, que los usuarios (desarrolladores, docentes, alumnos, etc.) puedan entender con facilidad. En el área docente, usando estos *notebooks* con los profesores en diversas clases se puede agilizar tanto la compresión de conceptos de los métodos numéricos, como de la asignatura en sí. En el área de desarrollo, se pueden usar en las primeras fases de diseño de soluciones ingenieriles. Esto debe ser una experiencia rápida y positiva, para que los usuarios usen esta nueva herramienta y no les obstruya en el flujo de desarrollo o aprendizaje.

Para poder crear una herramienta con las capacidades descritas anteriormente se opta en primer lugar por crear una API para invocar los distintos métodos numéricos. Desde la API se generan distintas tareas asociadas a la resolución de los problemas. Estos problemas son almacenados en una base de datos NoSQL. Después, los trabajadores, que son los procesos que realmente resuelven los problemas, recogen los problemas que han sido puestos en la base de datos. El paradigma descrito anteriormente, con respecto al proceso de publicar los problemas a la base de datos y después leerlos, es conocido como el paradigma publicador-suscriptor. Finalmente, cuando los trabajadores hayan resuelto los problemas, los resultados se almacenan en un almacenamiento de objetos similar a S3.

Todos los pasos descritos anteriormente tienen que operar bajo unas tecnologías que permitan dar al usuario una experiencia positiva. El primero de estos es el uso de contenedores. Los contenedores son un elemento fundamental para la arquitectura necesaria para una experiencia positiva. Estos permiten tener cada elemento aislado y así poder asegurar la resistencia del sistema ante incidencias. Esto se puede conseguir mediante el uso de orquestadores de contenedores. Estos orquestadores permiten la alta disponibilidad, ya que pueden regenerar contenedores que hayan sufrido algún fallo [12]. Además, usándolos se puede conseguir que

la capacidad escale según la demanda [12].

En este proyecto se propone una arquitectura que permita la resolución de estos problemas y el almacenaje de sus resultados. Esto se logra usando una arquitectura de microservicios usando contenedores Docker. Para poder desacoplar estos servicios se usa el modelo publicador-subcriptor. Finalmente, para almacenar estos resultados se usa un almacenamiento por objetos compatible con Docker. El segmento de los trabajadores es escalable usando Kubernetes. La mayor parte del código es código Python.

1.2 Objetivos del proyecto

El objetivo principal de este proyecto es desarrollar una arquitectura basada en microservicios, contenedores y Kubernetes que sirva de apoyo en el desarrollo de soluciones ingenieriles y al proceso de enseñanza-aprendizaje de métodos numéricos y que sea escalable y distribuida.

1.2.1. Objetivos específicos

Los siguientes son los objetivos específicos que se quieren completar en este proyecto:

- Crear la API para recibir *requests*.
- Dockerizar la API.
- Crear *workers* para resolver los problemas numéricos .
- Dockerizar *workers* para facilitar la escalabilidad .
- Usar el modelo publicador-subscriptor para desacoplar los servicios .
- Habilitar escalabilidad según el uso de los *workers*.
- Implementar herramientas de monitorización de los contenedores.
- Desplegar una solución de almacenamiento mediante objetos.

1.2.2. Beneficios del proyecto

El valor de este proyecto radica en las facilidades que una infraestructura de microservicios puede dar en una aplicación de este estilo. Por ejemplo, al ser todo *dockerizado* será fácil de trasladar de un entorno *cloud* a otro, o despegarlo *on premise* [12]. Esto puede facilitar el ahorro en costes y reducir el *vendor lock in*. Por otro lado, este tipo de soluciones aportan tolerancia a errores en el caso de que un contenedor finalice de manera errónea fácilmente se podría recuperar.

1.3 Resultados obtenidos

Se consigue desplegar una solución basada en Docker y Kubernetes que es fácilmente escalable. Se crea una arquitectura de publicador-suscriptor usando Celery y Redis. Minio es usado como almacén de resultados. Tanto la API como los trabajadores son *dockerizados*

con éxito.

Capítulo 2. Marco Teórico y Tecnológico

En este capítulo, se describen los aspectos más relevantes abordados en este trabajo y las principales herramientas usadas para el desarrollo de la solución propuesta.

2.1 Marco Teórico

Tanto los contenedores como sus orquestadores vienen a facilitar el uso de microservicios. Los microservicios son definidos como: 'un componente desplegable de manera independiente de alcance limitado, los cuales tienen capacidad de operar entre ellos a través de la comunicación basada en mensajes' [2]. Las principales características de los microservicios son que los servicios deben de ser accesibles a través de la red, el software está optimizado para conseguir un objetivo; y que el tamaño del servicio es un factor importante [2]. Hay varios modelos para conseguir diseñar microservicios; entre ellos, el *SEED(S)* [2]. Estos modelos se crean para tratar de mitigar algunos de los problemas de los microservicios como: la parálisis por análisis, el exceso de componentes y la retroalimentación lenta [2].

En el caso de los contenedores, esta es una tecnología que es similar a la de las máquinas virtuales [11]. Una ventaja de los contenedores frente a las máquinas virtuales es que no requieren un sistema operativo al completo para poder funcionar [11]. Esto se debe a que los contenedores comparten su sistema operativo con la máquina huésped [11]. Estos contenedores cuando son usados a gran escala tienen que ser gestionados mediante orquestadores.

En el caso de los orquestadores, estos son un sistema que puede desplegar aplicaciones y responder a los cambios de manera dinámica [12]. Algunas acciones que puede tomar también es escalar ,debido a la demanda [12]. Esta tecnología permite abstraerse de las diferencias que hay entre las distintas nubes [12].

En la definición de los microservicios, proporcionada anteriormente, se menciona la capacidad de comunicarse entre los componentes mediante mensajes. Para conseguir esto hay varios modelos; en este caso se usa un modelo publicador-consumidor. En este modelo, el productor pone los mensajes en colas de tareas y recoge los resultados [10]. Los consumidores realizan los trabajos en las colas de tareas [10]. Los consumidores realizan esto sin saber nada sobre los publicadores [10].

2.2 Marco Tecnológico

Para el desarrollo de la arquitectura propuesta se han empleado diversas herramientas para el desarrollo de todos sus componentes.

En este proyecto se usa Python como lenguaje de programación. Python es considerado un lenguaje lento comparado con otros lenguajes compilados como pueden ser C++ o Rust [9]. En este proyecto, el rendimiento no es una prioridad pero sí una ventaja, ya que puede ayudar a reducir la latencia. Hay varias librerías de Python que pueden compilar el código fuente de

Python a otros lenguajes. Esto ayuda a aumentar la eficiencia energética del lenguaje [15]. De entre las librerías que permiten compilar a otros lenguajes destacan Numba y Cython. Numba traduce el código Python a código máquina [9]. Según la propia documentación de la librería, esto permite que alcance velocidades similares a las de C++ o Fortran [9]. Numba tiene dos métodos para compilar; uno de ellos permite el uso de objetos y el otro no [9]. Otra manera que se suele usar para conseguir más velocidad es el paralelismo; y Numba puede ser usada con facilidad en paralelo [9]. En el caso de Cython, el código es compilado a C [9]. Cython también permite deshabilitar el *Global Interpreter Lock* [9]. Sin embargo, esto es menos relevante, ya que en las últimas versiones de Python esto se puede deshabilitar [6]. Estos aspectos son importantes para el proyecto, ya que el rendimiento es clave para que las latencias sean bajas.

Para poder obtener una arquitectura de microservicios, es clave contenerizar las aplicaciones. Existen varios motores para contenedores, de los cuales los más populares son Docker y Podman. Podman por defecto no opera como *root*, pero Docker por defecto sí funciona como *root* [8]. Sí es verdad que Docker tiene una implementación *rootless*, pero esta es menos madura que la de Podman [4]. Podman limita en mayor medida el alcance que puede tener un ataque a, únicamente, al contenedor que haya sido comprometido [4]. Las diferencias también afectan a cómo funcionan los contenedores, ya que Docker funciona con un servicio; mientras que Podman no lo necesita [4]. Podman, al no disponer de un servicio, es más difícil que pueda orquestarlos y monitorizarlos [4]. El uso del servicio por parte de Docker hace más fácil interactuar con los contenedores [4]. Pero estas diferencias no implican que sean incompatibles, ya que Podman da soporte a la *Open Container Initiative* [4]. Sin embargo, Podman no está siendo adaptado por muchos equipos, pero el ritmo de adopción está acelerándose [4]. Esto también se puede deber a que la mayoría de *pipelines* de integración y despliegue continuos están pensados para Docker [4]. Adicionalmente, Podman requiere que se use un *kernel* de Linux moderno, ya que necesita usar características modernas de él.

Es inevitable tener que mencionar a Kubernetes cuando se habla de orquestadores de contenedores. Sin embargo, es verdad que existen otras alternativas como Docker Swarm o Marathon; además, de aquellas que son nativas a sus respectivas nubes [7]. En el caso de Kubernetes, los *Pods* son la unidad básica y estos son gestionados por *Deployments* [7]. El *Kube-scheduler* es el componente responsable de seleccionar a un nodo para cada *Pod* ,dependiendo de sus requerimientos [7]. Este proceso se realiza en tres pasos principales: organizar, filtrar y puntuar [7]. Aunque solo hayamos mencionado la planificación, Kubernetes también se ocupa del despliegue y la gestión de los contenedores [7]. Kubernetes sí puede llegar a tener problemas con microservicios complejos que comparten el mismo nodo en el *cluster* [7].

Redis es una base de datos no relacional que vive en la memoria [1]. Aunque viva en la memoria, sí tiene capacidad para escribir al disco mediante dos métodos [1]. Redis funciona con pares de claves y valores [1]. Redis usa cinco tipos de datos como valores: cadenas de caracteres, listas, sets, hashes y zsets [1]. Además, tiene capacidad para replicación en el modo maestro esclavo [1].

La cola de tareas está implementada por la librería Celery [10]. Se suele usar para estas

aplicaciones a RabbitMQ para las tareas y Redis para almacenar los resultados [10].

Capítulo 3. DESARROLLO DEL PROYECTO

En este capítulo se describen las distintas fases del proyecto y su ejecución.

3.1 Metodología

Para el desarrollo de este proyecto se plantea una metodología *ad-hoc* que consistió en las siguientes fases:

- Diseño de la solución.
- Primer prototipo.
- Dockerización de *Worker* y *Publisher*.
- Segundo prototipo.
- Adaptación de *Workers* a *Kubernetes*.
- Pruebas Finales.

La fase de diseño de la solución consistió principalmente en cuatro pasos: revisión literaria, análisis del problema, partición del problema y selección de tecnologías. En los pasos de diseño de la solución, el objetivo principal es familiarizarse con el contexto del problema a resolver, decidir en qué microservicios se puede partir la aplicación y qué tecnologías son necesarias para poder hacerlo. Esto va alineado con los principales objetivos del proyecto, especificados en el Capítulo 1. Parte de la revisión literaria se resume en el Capítulo 1 y Capítulo 2. Después de estos cuatro pasos, se hace el primer prototipo. En este prototipo funciona la API y los trabajadores; solo estaría el *broker* en un contenedor Docker. Tras hacer las pruebas, se pasa a Dockerizar tanto la API como los *workers*. En el segundo prototipo están en Docker todos los elementos, incluyendo también el almacenaje final de resultados. Estos componentes están todos integrados dentro de un Docker Compose. En esta fase, también se realizan pruebas de la capacidad de recoger los resultados. Por último, se añade escalabilidad mediante Kubernetes a la capa de los trabajadores. A continuación, se describen detalladamente las actividades realizadas en cada una de las fases de la metodología abordada. Se describe en detalle la solución y sus componentes. Se divide en los principales pasos enumerados en la sección de metodología. En ellas, se explican tanto partes del código como la arquitectura a bajo y alto nivel.

3.2 Diseño de la Solución: Análisis del Problema

Para conseguir tener una herramienta de apoyo al desarrollo de soluciones ingenieriles y a la enseñanza de métodos numéricos y sus usos son necesarios varios elementos. En primer lugar, hace falta una API para recibir peticiones de los usuarios. En este caso, los usuarios serán desarrolladores, estudiantes o profesores. Esta herramienta debe intentar resolver las consultas con la mínima latencia posible.

En este caso, la API tendrá tres *endpoints* principales para poder solicitar la resolución de tres distintos tipos de problemas. Esta API está desarrollada en Python, usando el *framework*

de FastAPI. Después, hace falta una base de datos para almacenar estas solicitudes. En este caso, al usar Celery para publicar y leer las tareas en la cola es necesario atenerse a las recomendaciones de los desarrolladores de la herramienta. Los desarrolladores recomiendan usar la base de datos Redis [13]. También, hacen falta *workers* que estén *dockerizados* y escalables para poder resolver las solicitudes. Desde el inicio de la aplicación hay un *worker* iniciado. Sin embargo, mediante la escalabilidad el número de *workers* podrá aumentar si el uso de un pod excede ciertos niveles. La escalabilidad de estos *workers* será facilitada usando Kubernetes. Por último, hace falta un lugar en el cual almacenar estos resultados, que en este caso será un almacenamiento de tipo S3. En este caso, el *object store*, para almacenar los resultados, será Minio.

3.3 Diseño de la Solución: Partición del Problema

En la Figura 3.1 se muestran los principales elementos de la arquitectura: la API escrita con FastApi, la base de datos Redis, los *workers* de Celery y el *object store*. Cada uno de estos elementos es un microservicio necesario para poder conseguir una arquitectura capaz de resolver estos problemas de manera acorde a los objetivos establecidos.

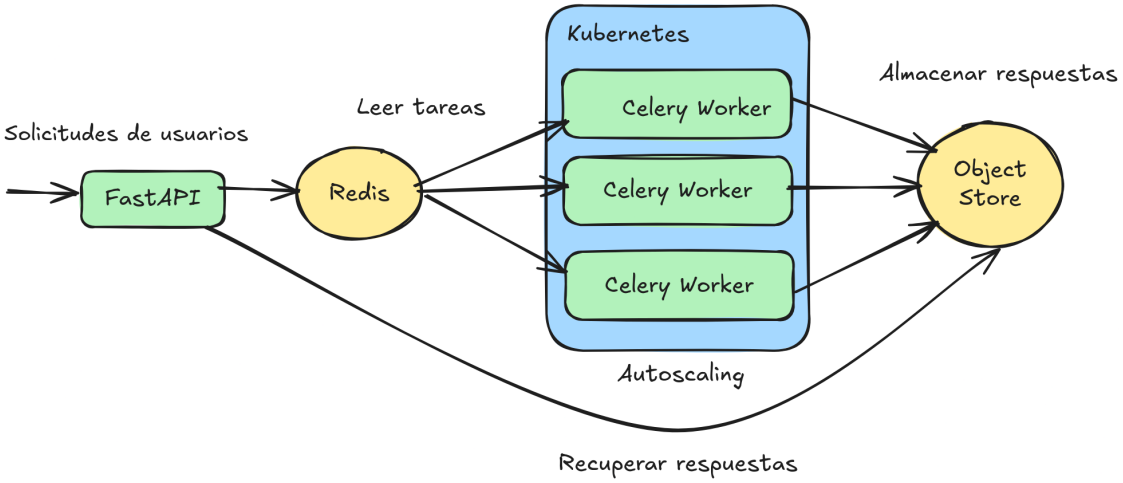

Figura 3.1. Diagrama general de la arquitectura.

En Figura 3.1 se puede observar el flujo de una petición de un usuario, que es el siguiente:

1. El usuario manda una *request* a un *endpoint*.
2. En el *endpoint* se valida la información para iniciar el problema.
3. La tarea es publicada por FastAPI a la base de datos Redis.
4. Se devuelve un identificador de la tarea.
5. Un trabajador de Celery está suscrito a la cola, y lee y resuelve el problema.
6. El trabajador de Celery escribe los resultados en el *object store* en Minio.

7. Desde FastApi, usando el identificador de la tarea, se puede recuperar el estado de la tarea y/o los resultados.

Los pasos enumerados anteriormente se cubren los principales eventos que ocurren para procesar una solicitud para resolver un problema.

3.4 Diseño de la Solución: Selección de tecnologías

Para el desarrollo del sistema web propuesto capaz de resolver problemas matemáticos de manera distribuida y con poca latencia, es necesario contar con varios componentes tecnológicos, tales como lenguaje de programación, contenedores, orquestador de contenedores, base de datos y sistema de almacenaje.

El lenguaje de programación que se ha empleado es Python, ya que es el que se ha usado durante el transcurso del máster. Python ofrece distintas librerías necesarias para generar la API y para gestionar las tareas. La API es un componente fundamental, ya que a través de ella se generan las tareas. Hay numerosos *frameworks* de APIs dentro del ecosistema de Python; en este proyecto se usa FastAPI. FastAPI tiene documentación por defecto con Swagger y además, tiene Pydantic para validaciones [14]. Tener validaciones es muy importante, ya que se usan datos proporcionados por los usuarios. Como *framework* para hacer pruebas se utiliza Pytest. Hacer pruebas es una práctica hoy en día muy generalizada. Ayuda a saber qué efectos tienen los cambios y prevenir *bugs*. Por último, se usa la librería Celery para gestionar las colas. A la hora de implementar la lógica de los métodos numéricos fueron necesarias dos librerías adicionales: Sympy y Numpy. Numpy es usada también en la asignatura de Procesamiento de Datos.

Con respecto a los contenedores y su orquestador, hay varias opciones para cada uno de ellos. En el caso de los contenedores, hay dos principales: Podman y Docker. En este caso, las ventajas de seguridad de Podman no justifican su uso, ya que no se va a desplegar en producción. Además, el uso de Docker, tanto durante el transcurso del máster como en entornos profesionales, está más generalizado, y esto ayuda a encontrar recursos de ayuda en caso de que sean necesarios. Con respecto a los orquestadores, hay dos que son más familiares como Docker Swarm y Kubernetes [12]. En este caso, tampoco hay duda de que Kubernetes tienen una adopción mucho más alta en los entornos profesionales. Además, también ha sido empleado en el máster a través de MiniKube para hacer pruebas con Kubernetes en local. La capacidad de hacer estas pruebas es útil durante el desarrollo del proyecto.

Las bases de datos que se usan durante el proyecto vienen en gran medida dictadas por las recomendaciones de la librería Celery. En este caso, se recomiendan dos *brokers* RabbitMQ y Redis [13]. En este caso, Redis también se ha usado durante el máster.

Con respecto a los sistemas de almacenaje hay varias opciones. Una de las primeras es simplemente almacenar los resultados en el *filesystem*. Sin embargo, esto no es compatible con el paradigma de la escalabilidad, ya que los datos quedan atados a un servidor. En el caso de usar un contenedor como Minio, se conserva la capacidad de escalar y no se accede a los ficheros mediante un *path*, sino mediante una llamada a una API. MiniO es un *object*

store del mismo modo que S3 en AWS, que se estudió durante el transcurso del máster.

Durante el proceso de desarrollo, se usaron también diversas herramientas de apoyo. Durante todo el proceso, se usa el ordenador de sobremesa personal en el cual el sistema operativo es EndevourOS. Este sistema es un sistema operativo basado en ArchLinux. En él se tiene instalado el editor de texto NeoVim, que es el que se usa durante el proceso de desarrollo.

Además, hay varias herramientas de ayuda al desarrollo, como lo son las siguientes librerías de Python: ruff, uv , mypy y coverage. Estas cuatro librerías son de gran ayuda durante el proceso de desarrollo en distintos aspectos. Uv es el gestor de paquetes que se usa durante el proyecto. Ruff es un *linter* que es de gran ayuda para poder evitar errores mientras se escribe código. Mypy es un comprobador de tipos. Coverage es útil a la hora de comprobar qué partes del código están cubiertas por pruebas. Se entra en detalle en su uso de estas dos últimas más adelante. Por supuesto, aunque no es una librería Python, el software Git es muy útil, ya que permite llevar un control de versiones y da tranquilidad a la hora de desarrollar el código.

3.5 Primer Prototipo

El código que está escrito en Python está dividido en varias partes, para así poder facilitar su comprensión y mantenimiento. En la Figura 3.2 se pueden observar los principales elementos que hay en este proyecto.

Se observan varias carpetas, que son paquetes, las cuales contienen ficheros **__init__.py**. Esto facilita llamar funciones u objetos que estén definidas dentro. Además, permite dividir el código en segmentos que estén relacionados. En este caso, los paquetes son los siguientes:

- Api.
- Methods.
- Task_manager.
- Tests.

Cada uno de estos contienen uno o más módulos con extensión *.py*. A continuación, se explican los elementos que hay en cada uno de los módulos y qué contiene cada uno.

3.5.1. Paquete *api*

Este primer paquete se compone por dos módulos. En el llamado *api.py* se define la API en FastApi y sus *endpoints*. Hay tres *endpoints* definidos para entregar los problemas a los métodos numéricos implementados. La definición de los *endpoints* es similar en todos los casos. En el *Code Snipet* 3.1 podemos ver sus características principales en el caso del método de la bisección.

```
1 @api.post("/submit_bisection_method")
2 async def submit_bisection_method(problem:
    Problem_Bisection_Method):
```

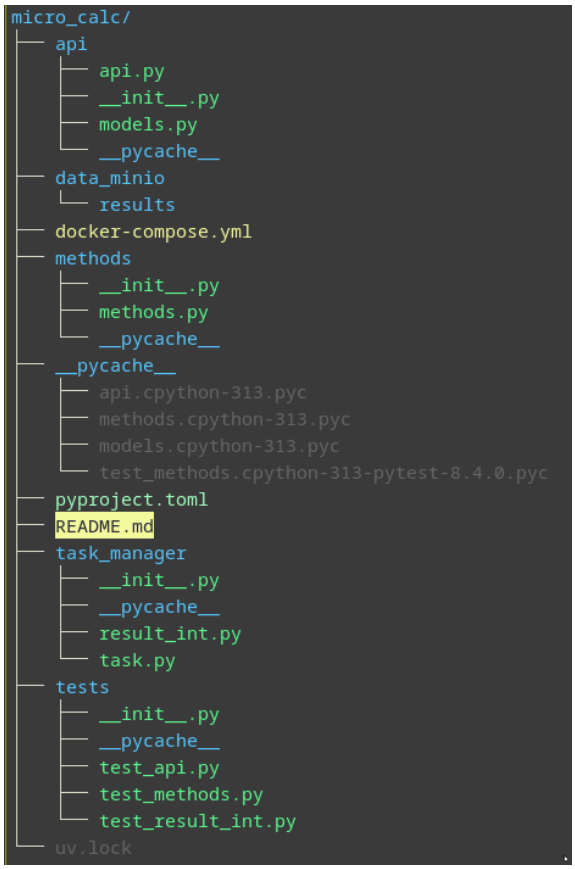

Figura 3.2. Foto de la estructura del proyecto.

```
3     solver = solve_bisection_method.delay(problem.to_dict())
4     return {"problem_id": solver.id}
```

Code Snippet 3.1: Muestra de *endpoint* de la API

En la primera línea del *Code Snipet* 3.1 se puede ver cuál es el *endpoint* de esta función. En la siguiente línea se puede observar una anotación de tipo para la entrada a esta función. En la tercera línea se envía el problema a resolver a Redis mediante Celery. También, es importante destacar el método *to_dict*, ya que al tener que ser serializados los elementos del problema esto es más fácil si todos los componentes del objeto *Problem_Bisection_Method* se envían como parte de un diccionario.

En este primer paquete, hay varias funciones como la que se muestra en el *Code Snipet* 3.1, cada una para un método numérico distinto. Además, en este paquete también entran los *endpoints* que se usan para comprobar el estado de un problema o recoger los resultados. Para poder hacer estas funcionalidades, es necesario tener algún tipo de identificador de estas tareas. Este identificador lo retorna Celery cuando se añade a la cola. Este identificador es lo que se retorna en la última fila del *Code Snipet* 3.1.

En este paquete hay un módulo más llamado *models.py*. En este módulo se valida lo que ha introducido el usuario. Para hacerlo se usa Pydantic. En este caso, se pueden incluir varios elementos como parte de la verificación, como veremos en el *Code Snipet* 3.2. Para cada tipo

de método numérico hay definida una clase, que hace estas validaciones; el código completo para cada una de ellas está en el Anexo 1.

```
1  class Problem_Bisection_Method(BaseModel):
2      endpoint_a: float
3      endpoint_b: float
4      tolerance: float
5      max_it: int
6      equation: str
7
8      def to_dict(self):
9          return {
10             "endpoint_a": self.endpoint_a,
11             "endpoint_b": self.endpoint_b,
12             "tolerance": self.tolerance,
13             "max_it": self.max_it,
14             "equation": self.equation,
15         }
16
17     @model_validator(mode="after")
18     def int_value_theorem_val(self) -> Self:
19         endpoint_a = self.endpoint_a
20         endpoint_b = self.endpoint_b
21         eq = self.equation
22         x = sy.Symbol("x")
23         func = sy.lambdify(x, eq)
24         eq_at_a = func(endpoint_a)
25         eq_at_b = func(endpoint_b)
26         if eq_at_a * eq_at_b >= 0:
27             raise ValueError(
28                 "Error in Intermediate Value Theorem
   validation of endpoints."
29             )
30         return self
```

Code Snippet 3.2: Muestra de clase para validaciones

En la declaración de la clase se ponen *type hints* para cada uno de los elementos de este problema desde la línea 2 a la 6 en el *Code Snippet* 3.2. Un beneficio de Pydantic es que estos tipos se comprueban solos. Con esto, nos quitamos de algunos errores que pueden ocurrir aguas abajo. Probablemente, en el desarrollo de este programa, este sea de los puntos más importantes, ya que es aquí donde se filtran las entradas del usuario y donde es más fácil parar los errores antes de que sucedan.

Para poder hacer esto, se pueden agregar validaciones personalizadas como el *int_value_theorem_val*. Esta validación está más relacionada con la lógica de los métodos numéricos, pero puede ser

útil para prevenir algunos errores más adelante en la fase de resolución del problema.

También, hay que destacar que el método mencionado anteriormente, *to_dict*, para convertir en diccionario los elementos de la clase se encuentra implementado en el *Code Snippet* 3.2 . Esto se debe a que este método debe de estar accesible para cualquier objeto de tipo problema.

Este módulo es probablemente el que, de cara a un uso por usuarios reales, debe reforzarse más para evitar más errores. Uno de los aspectos en los que es más difícil reforzarlo, pero también es más útil, es en la validación de la conversión del *string*, de entrada a una ecuación mediante Sympy. Además, en este punto al ser usada la aplicación, se pueden encontrar errores y problemas habituales, derivados de las entradas de los usuarios y este es el lugar apropiado para detectar esos problemas y tratar de solventarlos.

3.5.2. Paquete *methods*

El paquete *methods* contiene un único módulo con el mismo nombre. Este módulo contiene la parte central de la lógica para resolver los problemas con distintos métodos. En caso del *Code Snippet* 3.3 es el *bisection_method*. El resto de los métodos implementados se pueden encontrar en el Anexo 1. En las primeras líneas, desde la uno hasta la siete, en el *Code Snippet* 3.3, se ponen *type_hints* para todos los elementos, que son necesarios para resolver el problema, al igual que los tipos que se pueden retornar. En la línea nueve se crea una tabla para guardar los distintos componentes. Esto se hace para aprovechar el uso de Numpy.

```
1  def bisection_method(
2      endpoint_a: float,
3      endpoint_b: float,
4      tolerance: float,
5      max_it: int,
6      equation: Callable,
7  ) -> tuple[float, npt.NDArray[np.float64]] | str:
8      """Implements the Bisection Method to find roots."""
9      table = np.empty([max_it, 5])
10     eq_at_endpoint_a = equation(endpoint_a)
11     for iteration in range(1, max_it):
12         midpoint = endpoint_a + ((endpoint_b - endpoint_a) /
       2)
13         eq_at_midpoint = equation(midpoint)
14         table[iteration - 1, :5] = [
15             iteration,
16             endpoint_a,
17             endpoint_b,
18             midpoint,
19             eq_at_midpoint,
20         ]
```

```
21          if eq_at_midpoint == 0 or (endpoint_b - endpoint_a) /
    2 < tolerance:
22              table = np.delete(table, slice(iteration, max_it),
     0)
23              return midpoint, table
24          if eq_at_endpoint_a * eq_at_midpoint > 0:
25              endpoint_a = midpoint
26          else:
27              endpoint_b = midpoint
28      return "Procedure failed."
```

Code Snippet 3.3: Método numérico de muestra

Cada método tiene una lógica distinta, pero todos comparten dos puntos de salida del bucle. Se puede salir porque se completa con éxito o con un error, al no conseguir resolver el problema después de las iteraciones especificadas. En el caso del *Code Snippet* 3.3 la salida con éxito se produce en la línea 23 y sin éxito en la línea 28.

También, hay que comentar que el punto en el cual las librerías, como Numba, deberían de emplearse, es principalmente en este módulo, para conseguir unas ejecuciones más rápidas.

3.5.3. Paquete *task_manager*

El paquete *task_manager* consta de dos módulos: *task.py* y *result_int.py*. El primero de estos módulos es el encargado de ejecutar las tareas que se han puesto en la pila anteriormente. Para hacer esto, es necesario que, usando el diccionario que se recibe desde la API, se utilice este diccionario como argumento para las funciones definidas en el módulo *methods* del paquete con el mismo nombre. Esto se hace en la línea seis del *Code Snipet* 3.4 .

Para poder hacerlo, es necesario primero transformar la ecuación en una función de Sympy. Esta transformación se realiza de la línea 3 a 5 en el *Code Snippet* 3.4. Es aquí donde podemos notar la falta de validaciones adicionales en el objeto *Problem_Bisection_Method*, ya que si la transformación sale mal la tarea fallará.

Por último, usando la función *write_results* los resultados se escriben a Minio.

Cada uno de los métodos implementados dispone se una función *solve* de estilo similar a la que sale en el *Code Snipet* 3.4. Las funciones del resto de métodos se pueden encontrar en el Anexo 1.

```
1 @celery.task(name="solve_bisection_method", bind=True)
2 def solve_bisection_method(self, problem):
3     x = sy.Symbol("x")
4     eq = problem["equation"]
5     func = sy.lambdify(x, eq)
6     result = bisection_method(
7         problem["endpoint_a"],
```

```
        problem["endpoint_b"],
        problem["tolerance"],
        problem["max_it"],
        func,
    )
    write_results(client, problem, result, self)
```

Code Snippet 3.4: Muestra código que genera la tarea en Celery

El segundo módulo de este paquete contiene la lógica para poder comunicarse con el almacén de objetos, que en este caso es Minio. La función principal de este paquete está reflejada en el *Code Snipet* 3.5 y es la que se usa en el módulo *task.py* para escribir la solución.

```
def write_results(
    client: Minio,
    problem: dict,
    result: Union[tuple, str],
    task: Task,
):
    task_id = task.request.id
    try:
        with tempfile.TemporaryDirectory() as temporary_dir:
            if type(result) is tuple:
                table_name = "".join(["table_", task_id, ".csv"])
                table_path = Path(temporary_dir) / table_name
                np.savetxt(table_path, result[1])
                put_file_in_bucket(client, str(table_path), "results", table_name)
                problem["result"] = result[0]
            else:
                problem["result"] = result
            statement_name = "".join(["statement_", task_id, ".csv"])
            statement_path = Path(temporary_dir) / statement_name
            with open(statement_path, "w") as statement_file:
                json.dump(problem, statement_file)
            put_file_in_bucket(client, str(statement_path), "results", statement_name)
    except (MaxRetryError, S3Error) as e:
        if isinstance(e, MaxRetryError):
            logger.error("MinIO unreachable. Skipping result upload.")
        else:
            logger.error("MinIO internal error.")
```

```
28          task.update_state(state=states.FAILURE, meta="Failed
    to save to Minio.")
29          raise Ignore()
```

Code Snippet 3.5: Codigo para gestionar la escritura a Minio

En esta función nuevamente se usan *type_hints*. Los *type_hints* se encuentran desde la línea 2 a la 5 en el *Code Snipet* 3.5. Aparte de esto, se encapsula gran parte de la lógica en un bloque de *type-except*. Esto se debe a que gran parte de los problemas pueden surgir a raíz de errores en Minio, como podrían ser problemas de conexión.

Se usa también un gestor de contextos para crear un directorio temporal. Esto es para evitar que en el contenedor donde se despliega esto acumule archivos. Este gestor de contextos permite crear un directorio temporal,en el cual al salir del contexto se borra automáticamente con todos sus contenidos. Este directorio temporal se crea en la línea 9 del *Code Snippet* 3.5.

Esta función sirve para escribir los resultados, tanto en el caso en el que el problema se haya resuelto con éxito, como si no se ha alcanzado solución. La función que se usa para escribir es *put_file_in_bucket* la cual también está declarada en este mismo módulo. Es en el caso en el que se haya encontrado una solución, en la cual hay que escribir un fichero a Minio. Este fichero se relaciona con el problema mediante el identificador de la tarea. Este identificador pasa a formar parte del nombre del fichero que se guarda en Minio.

En el caso en el cual no se puede guardar a Minio, porque no se puede guardar en la línea 28 del *Code Snippet* 3.5, se actualiza el estado del problema en la base de datos de Redis. Los errores mediante los cuales se entran en el *except* se encuentran mediante el proceso de desarrollo.

3.5.4. Paquete *tests*

Este paquete no entra en la cadena para poder resolver solicitudes del usuario. Sin embargo, es clave para intentar reducir la cantidad de *bugs* en el código y facilitar el mantenimiento del mismo. Para cada uno de los paquetes anteriores, hay un módulo para crear *tests*. En estos *tests* se comprueban varios aspectos de la funcionalidad del código.

En el *Code Snipet* 3.6 se puede observar uno de los *tests* para el módulo de *methods*. En este caso, es bastante sencillo, ya que solo se requiere un *assert*. Hay bastantes más *tests* en este módulo, el resto se pueden encontrar en el Anexo 1.

```
1 def test_bisection_method_fail():
2     """Test if Bisection Method solves problem fails due to
    lack of iterations."""
3     x = sy.Symbol("x")
4     eq = x**3 + 4 * x**2 - 10
5     func = sy.lambdify(x, eq)
6     result = bisection_method(1, 2, 0.0001, 10, func)
```

```
7     assert result == "Procedure failed."
```

Code Snippet 3.6: *Test* para comprobar que método numérico falla correctamente

Este caso es de los más sencillos, ya que el paquete Pytest facilita mucho estas pruebas. Sin embargo, en otras ocasiones, para hacer un unit test que sea del todo aislado es necesario usar librerías como Pytest Mock. Esto hace que sea un poco más complicado crear los *tests*.

```
1  def test_table_results_success(mocker):
2      task_id = "054c9a22-f2ae-4b4b-8e11-1bc312d0e995"
3      object_name = "".join(["table_", task_id, ".csv"])
4      mock_client = mocker.Mock(spec=Minio)
5      mocker.patch("api.api.client", mock_client)
6      mock_get_file_in_bucket = mocker.patch("api.api.
       get_file_in_bucket")
7      mock_client.stat_object.return_value = Object("results",
       object_name)
8      response = client.get(f"/table_results/{task_id}")
9      assert response.status_code == 200
10     mock_get_file_in_bucket.assert_called_once()
11     mock_client.stat_object.assert_called_once()
```

Code Snippet 3.7: *Test* de un *endpoint* de FastAPI

Con la librería Mock, lo que se pretende hacer es simular alguna de las funcionalidades que tiene algún objeto o función. Las pruebas que se pueden hacer son, por ejemplo, cuántas veces estos objetos han sido llamados y con qué argumentos. En el caso del *Code Snippet* 3.7 en las líneas 10 y 11 se prueba cuántas veces estos objetos han sido llamados.

3.6 Dockerización de *Worker* y *Publisher*

Para poder generar contenedores Docker para la API y para los *Worker* de Celery, es necesario primero generar las imágenes. Estas imágenes son las plantillas para los contenedores [11]. Las imágenes se definen en ficheros llamados Dockerfile [11]. En estos, se especifica la imagen base y los elementos que se quieren añadir durante la construcción de la misma. Además, del comando a ejecutar durante la ejecución del contenedor. Esto se puede observar en el *Code Snippet* 3.8 .

```
1 FROM python:3.13
2 WORKDIR /code
3 COPY ./requirements.txt /code/requirements.txt
4 RUN pip install --no-cache-dir --upgrade -r /code/requirements
     .txt
5 COPY . /code
6 EXPOSE 8080
7 CMD ["fastapi", "run", "api/api.py", "--port", "8080"]
```

Code Snippet 3.8: Dockerfile para la imagen de FastAPI

En el *Code Snippet* 3.8 se puede observar cómo se hace lo siguiente:

1. Crea un directorio.
2. Se instala mediante pip las dependencias.
3. Se copia los contenidos del directorio *build* a la imagen.
4. Se expone el puerto 8080.
5. Se especifica el comando a ejecutar en el contenedor.

Para poder generar el fichero de dependencias, es necesario ejecutar un comando desde Uv, ya que Uv no lo usa de manera nativa.

También, se debe destacar la importancia del fichero dockerignore a la hora de copiar datos a la imagen. En las primeras ocasiones, en las cuales se trata de generar la imagen, esta imagen es de un tamaño muy grande. Es mejor tratar que las imágenes sean lo más pequeñas posibles, por eso, se incluye en el dockerignore varios directorios que no son necesarios, entre ellos están los enumerados en el *Code Snipet* 3.9.

```
.git
.mypy_cache
.pytest_cache
data_minio
.ropeproject
.ruff_cache
.venv
__pycache__
tests
.coverage
.gitignore
.python-version
Dockerfile
README.md
docker-compose.yml
pyproject.toml
uv.lock
```

Code Snippet 3.9: Archivo dockerignore

De los listados en el *Code Snipet* 3.9, se destacan varios y por qué han sido suprimidos. En el caso del .venv, este se suprime, ya que en la imagen docker en sí se van a instalar ya las dependencias y Python, por lo tanto, no hace falta copiarlo. Este directorio es ,probablemente, de los más grandes. También, se suprime el directorio de *test*, ya que no hacen falta para la ejecución del contenedor. Aparte de eso, se suprimen otros ficheros que también contienen las dependencias, ya que solamente es necesario el requirements.txt para instalar con Pip las dependencias en la imagen.

En el caso de la dockerización del *worker* de Celery, el dockerfile es muy parecido. La principal diferencia se encuentra en el comando que se ejecuta al iniciar el contenedor, como se puede observar en el *Code Snippet* 3.10 .

```
CMD ["celery", "-A", "task_manager.task", "worker"]
```

Code Snippet 3.10: Muestra del Dockerimage de los trabajadores de Celery

3.7 Segundo Prototipo

En esta fase del desarrollo, tras haber dockerizado todos los servicios, el objetivo principal es conseguir que todos funcionen de manera correcta dentro de un docker compose. Para conseguir esto, hay que hacer varios ajustes en el código fuente en los casos en los que se inician las conexiones a Minio y a la base de datos Redis. Dentro de una red en docker se puede referenciar un contenedor usando su nombre, ya que los DNS se pueden resolver internamente. Este cambio se realiza en este punto, para que a la hora de desarrollar no haya que poner la IP del contenedor, y que sea suficiente con el nombre del contenedor. Hay un pequeño problema con los nombres de los contenedores en Minio, ya que los nombres con barra bajas no son válidos.

```
services:
  cache:
    image: redis:latest
    container_name: redis-micro-calc
    restart: unless-stopped
  object_store:
    image: quay.io/minio/minio:latest
    container_name: minio-micro-calc
    environment:
      MINIO_ROOT_USER: minioadmin
      MINIO_ROOT_PASSWORD: minioadmin
    volumes:
      - ./data_minio:/data
    command: server /data --console-address ":9001"
    ports:
      - "9001:9001"
    restart: unless-stopped
  api:
    image: api-micro-calc:0.1
    container_name: api-micro-calc
    ports:
      - '8080:8080'
    restart: unless-stopped
  worker:
    image: worker-micro-calc:0.1
```

```
26      container_name: worker-micro-calc
27      restart: unless-stopped
```

Code Snippet 3.11: Docker compose

En el *Code Snipet* 3.11 , que contiene el docker compose, constan todos los servicios que son necesarios para este proyecto. Se puede observar cómo puntos más relevantes, que tanto la base de datos de Redis como el contenedor de Minio, no exponen todos sus puertos al *host*. Esto no es necesario, ya que los contenedores están dentro de una misma red de docker, y en esa red se pueden conectar unos a otros. Solo es necesario exponer aquellos puertos que tengan que ser accedidos desde el *host*, como es el caso del puerto de Minio por el cual se accede a la consola web y el de la API.

3.8 Adaptación de *Workers* a *Kubernetes*

Para poder desarrollar en Kubernetes es necesario disponer de algún mecanismo para poder desplegar los servicios en un software que simule ser un clúster de Kubernetes. Para poder hacerlo se usa KIND, que es muy similar a MiniKube. KIND son las siglas que corresponden a Kubernetes en Docker en inglés. La definición del clúster, al igual que del resto de componentes, se realiza mediante ficheros .yaml. En la definición del clúster se despliegan dos nodos.

Después, en otros ficheros se definen los pods para el resto de servicios: API, Redis, Minio y *workers*. En los ficheros de los servicios que no tienen que escalar se incluye un número fijo de pods. Sin embargo, también se ha creado un fichero para configurar la escalabilidad de los *workers*.

3.9 Pruebas Finales

Para poder comprobar si realmente los *workers* escalan es necesario someter al clúster a una carga elevada. Si la carga pasa cierto límite se activa la replicación de un pod *worker*. Para conseguir generar esta carga no basta con que un usuario mande peticiones. Es necesario simular un gran número de usuarios. Para conseguir hacer esto se puede usar una librería como Locust de Python. Esta librería levanta también una interfaz web para poder controlar las peticiones y la cantidad de ellas que fallan.

En la Figura 3.3 se muestra el estado del clúster nada más arrancar. En ella, se puede ver que hay cuatro pods, además de la carga en el *worker*.

En la Figura 3.4 se observa el estado del clúster con una carga más significativa de cuarenta usuarios. Se ha probado antes con diez, pero esos no son suficientes para desencadenar la generación de un nuevo pod *worker*. En Figura 3.4 se puede observar que el uso de CPU del pod *worker* ya ha superado el cincuenta por ciento. En ella, también se pueden observar ya dos pods de tipo *worker*. En la Figura 3.5 se muestra como hay cuarenta usuarios, además de la cantidad de peticiones que se han hecho, cuántas han tenido éxito y cuántas han fallado.

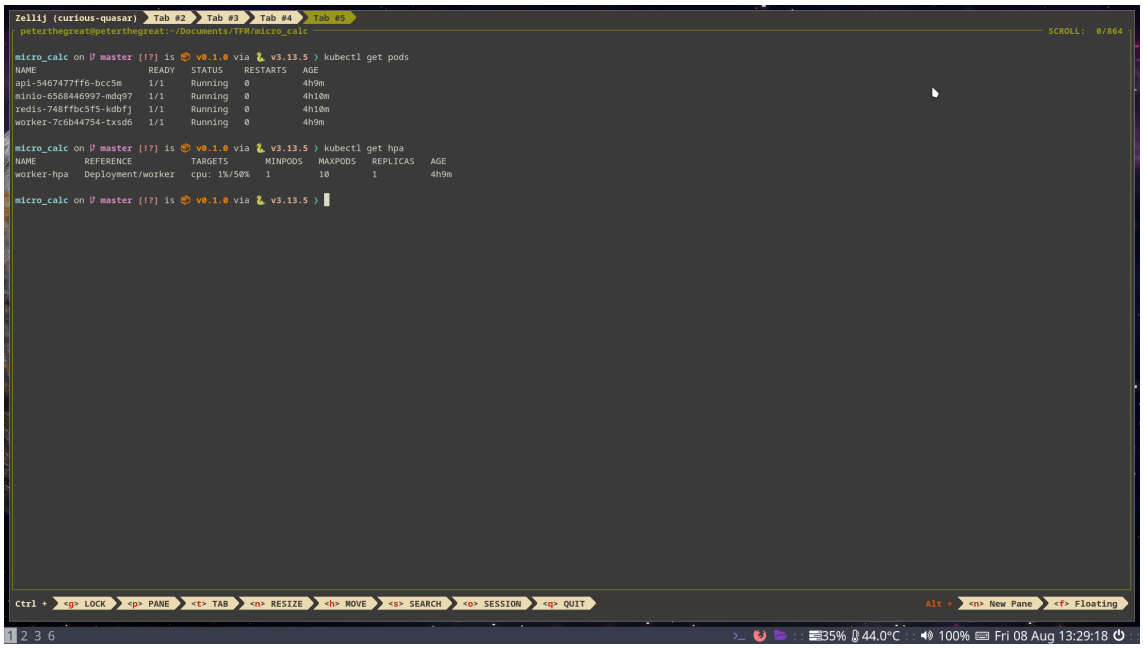

Figura 3.3. Estado del clúster al iniciarse

Figura 3.4. Estado del clúster con 40 usuarios

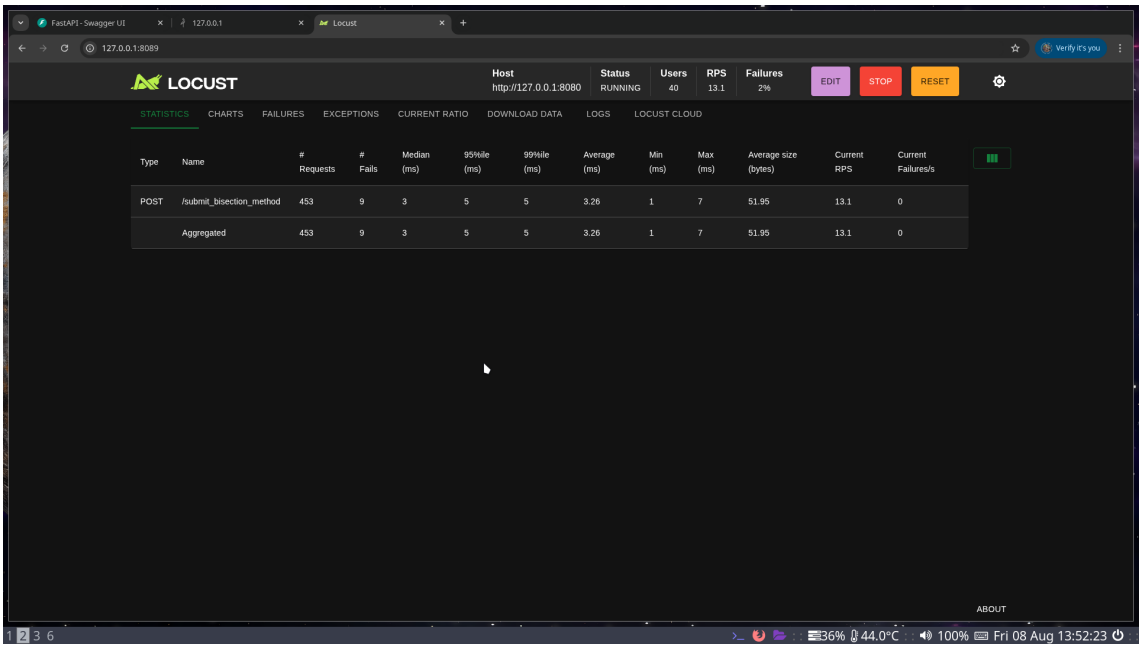

Figura 3.5. Interfaz de Locust con cuarenta usuarios

3.10 Proceso de Desarrollo

Hay varios elementos enumerados en la sección de recursos requeridos que no se pueden entender sin verlos en el contexto del proceso de desarrollo. En esta sección se explican algunos de los elementos citados anteriormente en el contexto del proceso de desarrollo.

El proceso de desarrollo se realiza principalmente en **Neovim**, que es un editor de texto que funciona en la terminal. En este proceso, primero se trata de desarrollar algún *feature* para después crear los *test* usando la herramienta Pytest. Hasta este punto del desarrollo se ha usado la herramienta Git para tener algo de trazabilidad sobre los cambios realizados. En este caso, sólo se usa una rama *main* pero en otros casos es mejor crear ramas para cada uno de los *features* y luego unir estas a la rama *main*.

Para poder hacer pruebas más rápidamente durante el proceso de desarrollo, se aprovecha de la interfaz gráfica que tiene FastApi. Esta interfaz hace de documentación de la API. Este componente que se llama Swagger se muestra en la Figura 3.6. Sin embargo, desde esta interfaz se pueden lanzar *requests*. Esta ha sido una herramienta que ayuda a iterar rápidamente. De no haberla tenido, se puede usar curl o Postman, pero al tenerla integrada con la documentación de la API podía encontrar problemas rápidamente.

Durante el proceso de desarrollo, hace falta poder buscar los errores de manera ágil y poder solventarlos. Para esto hay varias herramientas. La primera de ellas es un *linter*. En este caso, se usa Ruff. Es un *linter* para Python desarrollado en Rust por Astral. Esto ayuda a detectar algunos de los errores más básicos. Además, es posible tener el *linter* activado dentro de Neovim. Por otro lado, también se pueden usar comprobadores de tipo para detectar incompatibilidades entre tipos. Para poder usar estos comprobadores es necesaria la colocación de *type hints*. De ahí, que se trate a lo largo de todo el código de ponerlos. En este

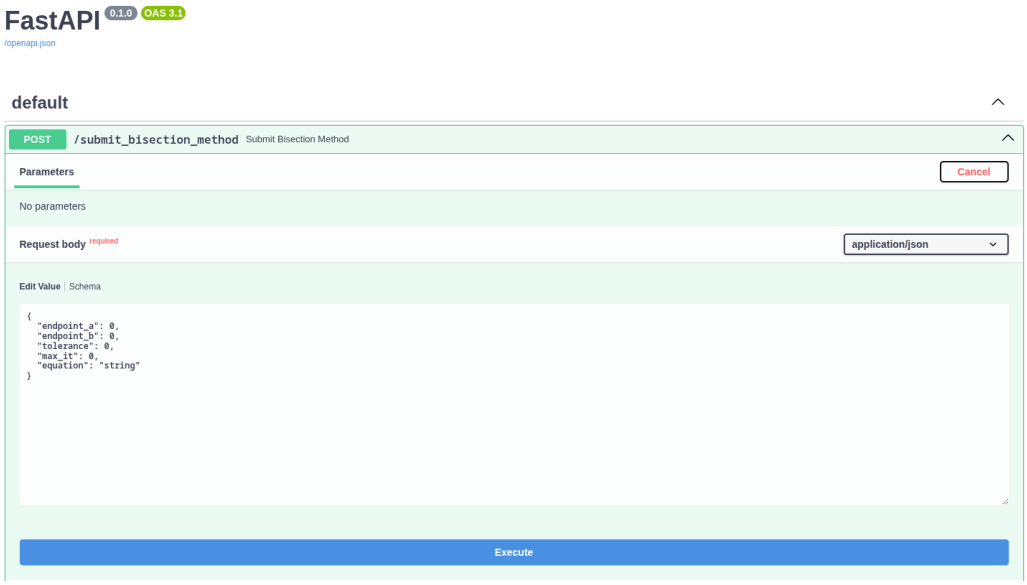

Figura 3.6. Uso de Swagger para hacer pruebas.

caso, el comprobador de tipos que se empleó fue mypy pero hay muchos otros. De hecho, recientemente, Astral ha sacado uno nuevo que también está implementado en Rust [5]. Por último, para tratar de mantener uniformidad en el formato del código, el formateador Black ha sido empleado.

En el caso del *testing* no sólo se usan librerías como Pytest para generar los test. Es también necesario verificar que con los *test* se está verificando el estado de la mayor parte del código. Para poder hacer esto, hay librerías como *coverage* que sirven para saber qué módulos están menos cubiertos por *tests*. Sin embargo, es importante no confiarse en exceso al ver cifras elevadas de *coverage* en los informes, ya que puede estar todo cubierto; pero si los *test* no son buenos no detectaremos posibles problemas. En la Figura 3.7, primero ejecutamos Coverage para que recopile la información. Después, se nos muestra la información sobre qué módulos están cubiertos por pruebas unitarias. En este caso, hay algunos módulos que están cubiertos por completo, mientras otros como task.py están a la mitad. Esto puede valer para identificar en qué zonas hay que centrar la creación de *tests*.

Para poder realizar pruebas, es necesario que tanto el contenedor de Redis como Minio estén funcionando. En este caso, se ha optado por usar Docker en modo *rootless*. En la Figura 3.8, se inicia Docker en modo *rootless* y se comprueba la existencia de los dos contenedores que se usaron en la fase inicial de desarrollo. De manera posterior, se usan también los contenedores en los cuales se haya dockerizado tanto la API como los *workers*.

Por último, en el proceso de desarrollo también es importante gestionar las dependencias. En este caso, se ha optado por usar Uv para gestionarlas. Uv es un paquete, desarrollado también por Astral, para poder substituir a Pip y a Venv y alguna otra librería adicional. En este caso, al igual que en los anteriores, está desarrollado en Rust. En este caso, lo más relevante es que puede instalar las dependencias rápidamente y también tiene un registro detallado de las dependencias de cada uno. Este registro está dentro del fichero llamado uv.lock. En él

```
micro_calc on  master is  v0.1.0 via  v3.13.5 ❯ uv run coverage run -m pytest
========================================= test session starts ==========================================
platform linux -- Python 3.13.5, pytest-8.4.0, pluggy-1.6.0
rootdir: /home/peterthegreat/Documents/TFM/micro_calc
configfile: pyproject.toml
plugins: mock-3.14.1, anyio-4.9.0
collected 18 items

tests/test_api.py .......                                                                      [ 38%]
tests/test_methods.py ......                                                                   [ 72%]
tests/test_result_int.py .....                                                                 [100%]

========================================== 18 passed in 1.25s ==========================================

micro_calc on  master is  v0.1.0 via  v3.13.5 ❯ uv run coverage report
Name                          Stmts   Miss  Cover
-------------------------------------------------
api/__init__.py                   0      0   100%
api/api.py                       47      3    94%
api/models.py                    50      2    96%
methods/__init__.py               0      0   100%
methods/methods.py               42      0   100%
task_manager/__init__.py          0      0   100%
task_manager/result_int.py       47      6    87%
task_manager/task.py             30     15    50%
tests/__init__.py                 0      0   100%
tests/test_api.py                60      0   100%
tests/test_methods.py            39      0   100%
tests/test_result_int.py         93     16    83%
-------------------------------------------------
TOTAL                           408     42    90%
```

Figura 3.7. Uso de Coverage.

```
micro_calc on  master is  v0.1.0 via  v3.13.5 took 2s ❯ systemctl --user start docker

micro_calc on  master is  v0.1.0 via  v3.13.5 ❯ docker container ls -a
CONTAINER ID   IMAGE                        COMMAND                  CREATED       STATUS                   PORTS                                       NAMES
965cfcbc6c67   quay.io/minio/minio:latest   "/usr/bin/docker-ent…"   3 weeks ago   Exited (0) 11 hours ago                                              minio_micro_calc
fa68baf1c73b   redis:latest                 "docker-entrypoint.s…"   3 weeks ago   Up 11 seconds            0.0.0.0:6379->6379/tcp, [::]:6379->6379/tcp   redis_micro_calc
```

Figura 3.8. Uso de Docker para desarrollo.

se contienen las dependencias de cada uno de los paquetes usados en el proyecto. Esto es en contraste a los contenidos del fichero pyproject.toml el cual contiene los paquetes de los que depende el proyecto a más alto nivel. Además, al igual que Pip, genera un entorno virtual para evitar contaminar el entorno del ordenador *host* con paquetes no deseados.

```
requires-python = ">=3.13"
dependencies = [
    "black>=25.1.0",
    "celery>=5.5.3",
    "celery-stubs>=0.1.3",
    "coverage>=7.9.1",
    "fastapi[standard]>=0.115.12",
    "isort>=6.0.1",
    "locust>=2.38.0",
    "minio>=7.2.15",
    "mypy>=1.16.0",
    "numpy>=2.2.6",
    "pytest>=8.4.0",
```

```
    "pytest-mock>=3.14.1",
    "python-lsp-server[all]>=1.12.2",
    "redis>=6.2.0",
    "ruff>=0.11.12",
```

Code Snippet 3.12: Muestra de pyproject.toml

En el *Code Snipet* 3.12 pueden observar lo que se especifica en el fichero pyproject.toml. En él se puede observar todas las dependencias de alto nivel que hay, además de la versión de Python. En contraste con el *Code Snipet* 3.13, que pertence al fichero uv.lock, que contiene las dependencias que tiene un paquete como Black.

```
[[package]]
name = "black"
version = "25.1.0"
source = { registry = "https://pypi.org/simple" }
dependencies = [
    { name = "click" },
    { name = "mypy-extensions" },
    { name = "packaging" },
    { name = "pathspec" },
    { name = "platformdirs" },
]
sdist = { url = "https://files.pythonhosted.org/packages
   /94/49/26
   a7b0f3f35da4b5a65f081943b7bcd22d7002f5f0fb8098ec1ff21cb6ef/
   black-25.1.0.tar.gz", hash = "sha256:33496
   d5cd1222ad73391352b4ae8da15253c5de89b93a80b3e2c8d9a19ec2666
   ", size = 649449, upload-time = "2025-01-29T04:15:40.373Z"
   }
wheels = [
    { url = "https://files.pythonhosted.org/packages/98/87/0
   edf98916640efa5d0696e1abb0a8357b52e69e82322628f25bf14d263d1
   /black-25.1.0-cp313-cp313-macosx_10_13_x86_64.whl", hash =
   "sha256:8
   f0b18a02996a836cc9c9c78e5babec10930862827b1b724ddfe98ccf2f2fe4f
   ", size = 1650673, upload-time = "2025-01-29T05:37:20.574Z"
    },
    { url = "https://files.pythonhosted.org/packages/52/e5/
   f7bf17207cf87fa6e9b676576749c6b6ed0d70f179a3d812c997870291c3
   /black-25.1.0-cp313-cp313-macosx_11_0_arm64.whl", hash = "
   sha256:
   afebb7098bfbc70037a053b91ae8437c3857482d3a690fefc03e9ff7aa9a5fd3
   ", size = 1453190, upload-time = "2025-01-29T05:37:22.106Z"
    },
    { url = "https://files.pythonhosted.org/packages/e3/ee/
```

```
    adda3d46d4a9120772fae6de454c8495603c37c4c3b9c60f25b1ab6401fe
    /black-25.1.0-cp313-cp313-manylinux_2_17_x86_64.
    manylinux2014_x86_64.manylinux_2_28_x86_64.whl", hash = "
    sha256:030
    b9759066a4ee5e5aca28c3c77f9c64789cdd4de8ac1df642c40b708be6171
    ", size = 1782926, upload-time = "2025-01-29T04:18:58.564Z"
     },
17   { url = "https://files.pythonhosted.org/packages/cc/64/94
    eb5f45dcb997d2082f097a3944cfc7fe87e071907f677e80788a2d7b7a/
    black-25.1.0-cp313-cp313-win_amd64.whl", hash = "sha256:
    a22f402b410566e2d1c950708c77ebf5ebd5d0d88a6a2e87c86d9fb48afa0d18
    ", size = 1442613, upload-time = "2025-01-29T04:19:27.63Z"
    },
18   { url = "https://files.pythonhosted.org/packages/09/71/54
    e999902aed72baf26bca0d50781b01838251a462612966e9fc4891eadd/
    black-25.1.0-py3-none-any.whl", hash = "sha256:95
    e8176dae143ba9097f351d174fdaf0ccd29efb414b362ae3fd72bf0f710717
    ", size = 207646, upload-time = "2025-01-29T04:15:38.082Z"
    },
19 ]
```

Code Snippet 3.13: Muestra de uv.lock

En el caso de Uv, se puede usar alguna característica adicional que tiene para tener los paquetes más ordenados. Uv permite usar categorías para las dependencias. En este caso, esto es útil, ya que hay muchas que son necesarias solo en la fase de desarrollo. Esto tiene efectos a la hora de crear las imágenes de Docker, ya que estas son más grandes de lo que es necesario si estos están segregados.

En los Apéndices se puede encontrar todo el código empleado para el desarrollo del proyecto.

3.11 Presupuesto

En este caso, el proyecto tiene un coste muy bajo. Esto se debe principalmente a que el único equipo necesario ya se tiene. Además, al usar software *Open Source* no es necesario pagar ninguna licencia.

3.12 Viabilidad

El desarrollo del proyecto en sí, la creación del código y la arquitectura, no tienen casi coste alguno. Sin embargo, una gran parte de este proyecto puede venir en el futuro en el caso de usarse a gran escala. Para poder estudiar la viabilidad, en ese caso, es primero necesario poner el proyecto en marcha y ver cuánto éxito tiene. En el caso de que el uso sea limitado, es posible que haya que escalar poco la cantidad de contenedores,y , en ese caso, que el

Tipo de coste	Valor	Comentarios
Horas de trabajo en el proyecto	Número de horas	Solo he trabajado yo en el proyecto.
Ordenador Sobremesa	1200 €	No ha sido necesario adquirirlo.
Software utilizado	0 €	Todo el software es software *Open Source*.
Estudios e informes	0 €	No ha sido necesario ningún informe.
Materiales empleados	0 €	No ha sido necesario ningún material adicional.

Tabla 3.1. Costes del proyecto

coste es muy reducido. También hay que estar vigilando las diversas nubes y sus precios para migrar, siempre que sea más barato, ya que esa es una de las ventajas de esta arquitectura. Cualquier tipo de despliegue en la nube tiene algún coste recurrente y, por lo tanto, hay que plantearse algún tipo de manera de conseguir ingresos; por ejemplo, mediante donaciones o estableciendo *tiers* de uso segregados por número de llamadas a la API.

3.13 Resultados del proyecto

Durante el proyecto se han cumplido con los principales objetivos planteados. Usando Kubernetes y Docker ha sido posible crear una infraestructura escalable para poder soportar el uso de una API para ejecutar métodos numéricos de manera distribuida.

Durante el transcurso del proyecto los objetivos se han mantenido iguales. Sin embargo, sí es necesario en ocasiones refinar o cambiar la tecnología o programa usado para algún menester. Un ejemplo de esto es MiniKube, que en un principio iba a ser usado para replicar un clúster de Kubernetes. Al final, se opta por usar KIND, ya que ofrece la misma funcionalidad pero es más simple. De igual manera, también fue necesario buscar una librería para generar una carga artificial para probar la escalabilidad del clúster de Kubernetes. Al iniciar el proyecto, se desconocen las librerías que existían para generar estas cargas y se opta por Locust porque es muy intuitiva.

Capítulo 4. DISCUSIÓN

En este capítulo se explica cuál es la relación de este proyecto con el mundo del Big Data. Además, se mencionan las diversas materias y temas tratados durante el transcurso de Máster que sirvieron de base de conocimiento para el desarrollo de este trabajo; tanto por las tecnologías que se usan como por los temas teóricos que se abordan de manera práctica.

El mundo del Big Data y la Ciencia de Datos ofrecen herramientas para el desarrollo de servicios distribuidos de gran escala y con respuestas en tiempo real.Tal es el caso del uso de microservicios distribuidos, el uso de Docker para mejorar la interoperabilidad y la portabilidad de la herramienta, el uso de Kubernetes para soportar escalabilidad. En este proyecto Docker se usó tanto para la generación de microservicios como para la generación de prototipos usando Docker Compose. Kubernetes se empleó para facilitar la escalabilidad de los trabajadores.

Otra herramienta sería el *Cloud Computing*, que se mencionó en la sección de Viabilidad, para poner este proyecto en producción. Para poder hacerlo serían necesarios servicios como S3 y EKS de AWS.

Con respecto al conocimiento brindado por el Máster, este proyecto está relacionado con varias materias:

- Bases de Datos de Nueva Generación.
- Arquitecturas Cloud Computing.
- Computación en Sistemas Distribuidos.

Con respecto a la materia de Arquitecturas Cloud Computing se tratan algunos temas cubiertos en las siguientes unidades:

- Unidad 2. Infraestructura para computación en la nube.
- Unidad 5. Desarrollo y despliegue de aplicaciones en la nube.

En la Unidad Dos se trata el uso de contenedores. Durante el proyecto el uso de contenedores es fundamental. Durante el transcurso de este proyecto se ha usado sobre todo Docker y Docker Compose. No solo se han usado Docker con imágenes cogidas del Docker Registry directamente, sino que mediante DockerFiles se crean nuevas imágenes para ser usadas durante el transcurso del proyecto. También en esta unidad, se tratan distintos sistemas de almacenamiento entre los cuales figuran el almacenamiento de objetos. En el caso del proyecto, se emplea también un sistema de almacenamiento de objetos, llamado Minio.

En el caso de la Unidad Cinco, se trata un tema central de este proyecto, que son los microservicios. En este caso, se menciona además de Docker a Kubernetes. También, hay mención de distintos tipos de arquitecturas para los microservicios, entre las cuales figura la de publicador-suscriptor.

En el caso de Computación en Sistemas Distribuidos, se trata el tema de Docker en la Unidad 2. Además, en esta unidad se hace también mención a los sistemas de almacenamiento de objetos.

Por último, en la materia de Bases de Datos de Nueva Generación en la Unidad 7 se explican tecnologías como Apache Kafka en el contexto de arquitecturas de publicador-suscriptor. Además, en la Unidad de Bases de Datos Clave Valor y Columna Ancha se menciona a Redis, que es la base de datos que se ha usado en el proyecto. Durante esta materia, también se usa Docker y también Minikube. Minikube se usa para simular un clúster de Docker. Esto sirve como inspiración para buscar una tecnología similar, que en el caso de este proyecto es KIND.

Finalmente, hay que destacar que durante el transcurso del máster se hace un uso extenso de Python que es el lenguaje de programación principal en este proyecto.

Capítulo 5. CONCLUSIONES

5.1 Conclusiones del trabajo

El trabajo, con relación a los objetivos establecidos al comienzo, ha sido completado de manera exitosa. Ha sido posible cumplir con todos los objetivos principales. Estos se describen a continuación.

Se ha podido crear una API para recibir los *requests*. Esta API se ha creado usando Python con el *framework* de FastAPI. Después, se ha *dockerizado* la API para que encaje en el paradigma de los microservicios. Se han creado *workers* para poder resolver los problemas y estos han sido *dockerizados* para facilitar la escalabilidad. La escalabilidad de estos *workers* se ha habilitado usando Kubernetes. La escalabilidad de los*workers* ha sido posible debido al uso del modelo publicador-subscriptor para desacoplar los servicios. Por ultimo se usó Minio para permitir el almacenaje mediante objetos.

5.2 Conclusiones personales

En este proyecto se han aprendido numerosos aspectos, tanto de desarrollo en Python, como de temas de Kubernetes. En cuanto a Python, se ha aprendido a desplegar en FastApi una API. Esto engloba todos los ámbitos de FastApi, como las validaciones con Pydantic. Hasta ahora, solo se ha tenido experiencia en desarrollo web con Django. Con respecto a Celery, también se ha tenido ocasión de aprender, aunque con ello ya se ha tenido alguna experiencia previa.

Con respecto al tema de Kubernetes, era todo nuevo. En el máster se han tocado algunas tecnologías, pero no han habido muchas ocasiones para desplegar algo desde cero, ni mucho menos configurarlo. Los problemas se reducen significativamente en el ámbito de Docker, ya que de esa tecnología se cuenta con más experiencia y ha sido ampliamente cubierta en varias materias del máster.

El momento más difícil de este proyecto ha sido el principio y la organización paso a paso de los elementos a seguir. En muchas ocasiones, es fácil lanzarse de manera desorganizada y es fácil acabar desmotivado al no ver avances. Sin embargo, con la metodología seguida se iban viendo metas e hitos y era sencillo ver que se cumplían ciertos objetivos.

En este caso, tanto lo que se ha aprendido, como el medio mediante el cual se ha tenido ocasión de aprender, han sido muy interesantes. Por un lado, este proyecto es una manera de aunar temas relacionados con la Ingeniería Aeroespacial y el Big Data. Además, se puede considerar que todo el conocimiento que se ha adquirido puede ser de gran ayuda a la hora de desarrollarse de manera profesional.

Capítulo 6. FUTURAS LÍNEAS DE TRABAJO

Este proyecto tiene mucho potencial para expandirse, tanto en alcance como en profundidad. En cuanto a la profundidad hay muchos métodos numéricos de distintos tipos que aún quedan por implementar. También hay muchos *endpoints* que se pueden añadir. Algunos de ellos son para generar gráficas, una de las más fáciles son, por ejemplo, las gráficas de convergencia.

En cuando al alcance, se puede ampliar para evaluar la viabilidad de este servicio y su coste en varias nubes. En primer lugar, hay que tratar de desplegar el proyecto en distintas nubes y someterlo a cargas iguales para ver cuál es el coste. En el marco teórico se toca el tema de librerías que transforman el código Python en código de C o C++, es relevante poder ver si esto tiene algún efecto sobre el coste que sea importante. Por supuesto, también sería curioso poder observar si el uso de Podman en vez de Docker tiene algún tipo de impacto.

En cuanto al aspecto de la usabilidad desde un Jupyter notebook, es interesante probar la API desde uno y ver cómo se pueden modificar para que sea más fácil el uso desde *notebooks*.

Bibliografía

[1] Josiah L. Carlson. *Redis in Action*. Manning, 2013.

[2] Emiliano Casalicchio y Stefano Iannucci. *The State-of-the-Art in Container Technologies*. Inf. téc. Accessed: 2025-06-23. Mississippi State University CSE, 2020. URL: https://www.cse.msstate.edu/wp-content/uploads/2020/02/j5.pdf.

[3] R. C. Hibbeler. *Structural Analysis*. Pearson, 2017.

[4] Surbhi Kanthed. *F-23-224*. Inf. téc. Accessed: 2025-06-23. All Multidisciplinary Journal, 2025. URL: https://www.allmultidisciplinaryjournal.com/uploads/archives/20250405132101_F-23-224.1.pdf.

[5] Hugo van Kemenade. *ty*. Accessed: 2025-08-1. 2025. URL: https://docs.astral.sh/ty/.

[6] Hugo van Kemenade. *What's new in Python 3.14*. Last updated: 2025-08-1. Accessed: 2025-08-1. 2025. URL: https://docs.python.org/3.14/whatsnew/3.14.html#what-s-new-in-python-3-14.

[7] Angelo Marchese y Orazio Tomarchio. «Enhancing the Kubernetes Platform with a Load-Aware Orchestration Strategy». En: *SN Computer Science* 6.3 (2025). Published 25 February 2025; Accessed: 2025-06-23, pág. 224. DOI: 10.1007/s42979-025-03712-z. URL: https://link.springer.com/article/10.1007/s42979-025-03712-z.

[8] Khairan Marzuki et al. «Analysis and Implementation of Comparison Between Podman and Docker». En: *International Journal of Electronics and Communications Systems* 3.2 (2023). Accessed: 2025-06-23, págs. 57-67. DOI: 10.24042/ijecs.v3i2.19860. URL: https://ejournal.radenintan.ac.id/index.php/IJECS/article/view/19860/pdf.

[9] Andrés Milla y Enzo Rucci. «Performance Comparison of Python Translators for a Multi-threaded CPU-bound Application». En: *arXiv preprint arXiv:2203.08263* (2022). Accessed: 2025-06-23. URL: https://arxiv.org/pdf/2203.08263.

[10] Francesco Pierfederici. *Distributed Computing with Python*. Packt Publishing, 2019.

[11] Nigel Poulton. *Docker Deep Dive*. Nigel Poulton Ltd, 2023.

[12] Nigel Poulton. *The Kubernetes Book*. Nigel Poulton Ltd, 2025.

[13] Celery Project. *Backends and Brokers*. Last updated: 2025-06-01. Accessed: 2025-06-23. 2025. URL: https://docs.celeryq.dev/en/stable/getting-started/backends-and-brokers/index.html.

[14] Sebastián Ramírez. *FastAPI*. Published initially in 2018. Accessed: 2025-06-23. 2025. URL: https://fastapi.tiangolo.com/.

[15] Vincenzo Stoico, Andrei Calin Dragomir y Patricia Lago. «An Empirical Study on the Performance and Energy Usage of Compiled Python Code». En: *arXiv preprint arXiv:2505.02346* (2025). Accessed: 2025-06-23. URL: https://arxiv.org/pdf/2505.02346.

APÉNDICE 1: Código Python

Paquete api

```python
from celery.result import AsyncResult
from fastapi import FastAPI, HTTPException
from fastapi.responses import StreamingResponse
from minio import Minio
from minio.error import S3Error

from api.models import (
    Problem_Bisection_Method,
    Problem_Fixed_Point,
    Problem_Secant_Method,
)
from task_manager.result_int import get_file_in_bucket
from task_manager.task import (
    solve_bisection_method,
    solve_fixed_point,
    solve_secant_method,
)

api = FastAPI()

client = Minio(
    endpoint="minio:9000",
    access_key="minioadmin",
    secret_key="minioadmin",
    secure=False,
)


@api.post("/submit_bisection_method")
async def submit_bisection_method(problem:
   Problem_Bisection_Method):
    solver = solve_bisection_method.delay(problem.to_dict())
    return {"problem_id": solver.id}


@api.post("/submit_fixed_point")
async def submit_fixed_point(problem: Problem_Fixed_Point):
    solver = solve_fixed_point.delay(problem.to_dict())
    return {"problem_id": solver.id}
```

```python


@api.post("/submit_secant_method")
async def submit_secant_method(problem: Problem_Secant_Method):
    solver = solve_secant_method.delay(problem.to_dict())
    return {"problem_id": solver.id}


@api.get("/problem_status/{task_id}")
async def problem_status(task_id):
    task_result = AsyncResult(task_id)
    task_status = {
        "task_id": task_id,
        "task_status": task_result.status,
    }
    return task_status


@api.get("/table_results/{task_id}")
async def table_results(task_id):
    try:
        table_name = "".join(["table_", task_id, ".csv"])
        client.stat_object("results", table_name)
        response = get_file_in_bucket(client, "results", table_name)
        headers = {"Content-Disposition": f'attachment; filename="{table_name}"'}
        return StreamingResponse(response, media_type="text/csv", headers=headers)
    except S3Error:
        raise HTTPException(status_code=404, detail="Table of results not found.")


@api.get("/problem_statement/{task_id}")
async def problem_statement(task_id):
    try:
        statement_name = "".join(["statement_", task_id, ".csv"])
        client.stat_object("results", statement_name)
        response = get_file_in_bucket(client, "results", statement_name)
        headers = {"Content-Disposition": f'attachment;
```

```
    filename="{statement_name}"'}
76          return StreamingResponse(response, media_type="text/
    csv", headers=headers)
77      except S3Error:
78          raise HTTPException(status_code=404, detail="Problem
    statement not found.")
```

```
1  from typing import Self
2
3  import sympy as sy
4  from pydantic import BaseModel, model_validator
5
6
7  class Problem_Bisection_Method(BaseModel):
8      endpoint_a: float
9      endpoint_b: float
10     tolerance: float
11     max_it: int
12     equation: str
13
14     def to_dict(self):
15         return {
16             "endpoint_a": self.endpoint_a,
17             "endpoint_b": self.endpoint_b,
18             "tolerance": self.tolerance,
19             "max_it": self.max_it,
20             "equation": self.equation,
21         }
22
23     @model_validator(mode="after")
24     def int_value_theorem_val(self) -> Self:
25         endpoint_a = self.endpoint_a
26         endpoint_b = self.endpoint_b
27         eq = self.equation
28         x = sy.Symbol("x")
29         func = sy.lambdify(x, eq)
30         eq_at_a = func(endpoint_a)
31         eq_at_b = func(endpoint_b)
32         if eq_at_a * eq_at_b >= 0:
33             raise ValueError(
34                 "Error in Intermediate Value Theorem
    validation of endpoints."
35             )
36         return self
```

```python


class Problem_Fixed_Point(BaseModel):
    point: float
    tolerance: float
    max_it: int
    equation: str

    def to_dict(self):
        return {
            "point": self.point,
            "tolerance": self.tolerance,
            "max_it": self.max_it,
            "equation": self.equation,
        }


class Problem_Secant_Method(BaseModel):
    point_0: float
    point_1: float
    tolerance: float
    max_it: int
    equation: str

    def to_dict(self):
        return {
            "point_0": self.point_0,
            "point_1": self.point_1,
            "tolerance": self.tolerance,
            "max_it": self.max_it,
            "equation": self.equation,
        }

    @model_validator(mode="after")
    def int_value_theorem_val(self) -> Self:
        point_0 = self.point_0
        point_1 = self.point_1
        eq = self.equation
        x = sy.Symbol("x")
        func = sy.lambdify(x, eq)
        eq_at_0 = func(point_0)
        eq_at_1 = func(point_1)
        if eq_at_0 * eq_at_1 >= 0:
            raise ValueError(
```

```
                "Error in Intermediate Value Theorem validation of endpoints."
            )
        return self
```

Paquete load_test

```
from locust import HttpUser, between, task


class MyApiUser(HttpUser):
    wait_time = between(1, 5)

    @task
    def post_endpoint(self):
        self.client.post(
            "http://127.0.0.1:8080/submit_bisection_method",
            json={
                "endpoint_a": 1,
                "endpoint_b": 2,
                "tolerance": 0.001,
                "max_it": 20,
                "equation": "x**3 + 4 * x**2 - 10",
            },
        )
```

Paquete methods

```
from typing import Callable

import numpy as np
import numpy.typing as npt


def bisection_method(
    endpoint_a: float,
    endpoint_b: float,
    tolerance: float,
    max_it: int,
    equation: Callable,
) -> tuple[float, npt.NDArray[np.float64]] | str:
    """Implements the Bisection Method to find roots."""
    table = np.empty([max_it, 5])
    eq_at_endpoint_a = equation(endpoint_a)
```

```
    for iteration in range(1, max_it):
        midpoint = endpoint_a + ((endpoint_b - endpoint_a) / 2)
        eq_at_midpoint = equation(midpoint)
        table[iteration - 1, :5] = [
            iteration,
            endpoint_a,
            endpoint_b,
            midpoint,
            eq_at_midpoint,
        ]
        if eq_at_midpoint == 0 or (endpoint_b - endpoint_a) / 2 < tolerance:
            table = np.delete(table, slice(iteration, max_it), 0)
            return midpoint, table
        if eq_at_endpoint_a * eq_at_midpoint > 0:
            endpoint_a = midpoint
        else:
            endpoint_b = midpoint
    return "Procedure failed."


def fixed_point(
    point: float, tolerance: float, max_it: int, equation: Callable
) -> tuple[float, npt.NDArray[np.float64]] | str:
    """Implements the Fixed-Point Iteration method."""
    table = np.empty([max_it, 2])
    for iteration in range(1, max_it):
        eq_at_point = equation(point)
        table[iteration - 1, :2] = [
            iteration,
            eq_at_point,
        ]
        if abs(eq_at_point - point) < tolerance:
            table = np.delete(table, slice(iteration, max_it), 0)
            return point, table
        point = eq_at_point
    return "Procedure failed."


def secant_method(
```

```
    point_0: float, point_1: float, tolerance: float, max_it: int, equation: Callable
) -> tuple[float, npt.NDArray[np.float64]] | str:
    """Implements the Secant Method."""
    table = np.empty([max_it, 4])
    eq_at_point_0 = equation(point_0)
    eq_at_point_1 = equation(point_1)
    for iteration in range(1, max_it):
        point = point_1 - eq_at_point_1 * (point_1 - point_0) / (
            eq_at_point_1 - eq_at_point_0
        )
        table[iteration - 1, :5] = [
            iteration,
            eq_at_point_0,
            eq_at_point_1,
            point,
        ]
        if abs(point - point_1) < tolerance:
            table = np.delete(table, slice(iteration, max_it), 0)
            return point, table
        point_0 = point_1
        eq_at_point_0 = eq_at_point_1
        point_1 = point
        eq_at_point_1 = equation(point)
    return "Procedure failed."
```

Paquete task_manager

```
import json
import logging
import tempfile
from pathlib import Path
from typing import Union

import numpy as np
from celery import states
from celery.app.task import Task
from celery.exceptions import Ignore
from minio import Minio
from minio.error import S3Error
from urllib3.exceptions import MaxRetryError

```

```
15 logger = logging.getLogger(__name__)
16
17
18 def put_file_in_bucket(
19     client: Minio, source_path: str, bucket_name: str,
   destination_name: str
20 ):
21     """Puts file into minio."""
22     found = client.bucket_exists(bucket_name)
23     if not found:
24         client.make_bucket(bucket_name)
25     client.fput_object(
26         bucket_name,
27         destination_name,
28         source_path,
29     )
30
31
32 def get_file_in_bucket(client: Minio, bucket_name: str,
   source_name: str):
33     """Get file in minio."""
34     try:
35         return client.get_object(
36             bucket_name,
37             source_name,
38         )
39     except MaxRetryError:
40         logger.error("MinIO unreachable. Skipping result
   upload.")
41     except S3Error:
42         logger.error("MinIO internal error.")
43
44
45 def write_results(
46     client: Minio,
47     problem: dict,
48     result: Union[tuple, str],
49     task: Task,
50 ):
51     task_id = task.request.id
52     try:
53         with tempfile.TemporaryDirectory() as temporary_dir:
54             if type(result) is tuple:
55                 table_name = "".join(["table_", task_id, ".csv
```

```
    "])
56                  table_path = Path(temporary_dir) / table_name
57                  np.savetxt(table_path, result[1])
58                  put_file_in_bucket(client, str(table_path), "
    results", table_name)
59                  problem["result"] = result[0]
60              else:
61                  problem["result"] = result
62              statement_name = "".join(["statement_", task_id, "
    .csv"])
63              statement_path = Path(temporary_dir) /
    statement_name
64              with open(statement_path, "w") as statement_file:
65                  json.dump(problem, statement_file)
66              put_file_in_bucket(client, str(statement_path), "
    results", statement_name)
67          except (MaxRetryError, S3Error) as e:
68              if isinstance(e, MaxRetryError):
69                  logger.error("MinIO unreachable. Skipping result
    upload.")
70              else:
71                  logger.error("MinIO internal error.")
72              task.update_state(state=states.FAILURE, meta="Failed
    to save to Minio.")
73              raise Ignore()
```

```
1  import logging
2
3  import sympy as sy
4  from celery import Celery
5  from minio import Minio
6
7  from methods.methods import bisection_method, fixed_point,
       secant_method
8  from task_manager.result_int import write_results
9
10 logger = logging.getLogger(__name__)
11
12 celery = Celery(
13     "task", broker="redis://redis:6379/0", backend="redis://
       redis:6379/0"
14 )
15
16 client = Minio(
```

```
    endpoint="minio:9000",
    access_key="minioadmin",
    secret_key="minioadmin",
    secure=False,
)


@celery.task(name="solve_bisection_method", bind=True)
def solve_bisection_method(self, problem):
    x = sy.Symbol("x")
    eq = problem["equation"]
    func = sy.lambdify(x, eq)
    result = bisection_method(
        problem["endpoint_a"],
        problem["endpoint_b"],
        problem["tolerance"],
        problem["max_it"],
        func,
    )
    write_results(client, problem, result, self)


@celery.task(name="solve_fixed_point", bind=True)
def solve_fixed_point(self, problem):
    x = sy.Symbol("x")
    eq = problem["equation"]
    func = sy.lambdify(x, eq)
    result = fixed_point(
        problem["point"], problem["tolerance"], problem["
   max_it"], func
    )
    write_results(client, problem, result, self)


@celery.task(name="solve_secant_method", bind=True)
def solve_secant_method(self, problem):
    x = sy.Symbol("x")
    eq = problem["equation"]
    func = sy.lambdify(x, eq)
    result = secant_method(
        problem["point_0"],
        problem["point_1"],
        problem["tolerance"],
        problem["max_it"],
```

```
        func,
    )
    write_results(client, problem, result, self)
```

Paquete tests

```
import pytest
from fastapi.testclient import TestClient
from minio import Minio
from minio.datatypes import Object
from minio.error import S3Error

from api.api import api

client = TestClient(api)


def test_submit_bisection_method():
    response = client.post(
        "/submit_bisection_method",
        json={
            "endpoint_a": 1,
            "endpoint_b": 2,
            "tolerance": 0.0001,
            "max_it": 10,
            "equation": "x**3 + 4 * x**2 - 10",
        },
    )
    assert response.status_code == 200
    assert "problem_id" in response.json()


def test_submit_fixed_point():
    response = client.post(
        "/submit_fixed_point",
        json={
            "point": 1.5,
            "tolerance": 0.0001,
            "max_it": 20,
            "equation": "0.5 * (10 - x**3) ** 0.5",
        },
    )
    assert response.status_code == 200
    assert "problem_id" in response.json()
```

```python


def test_submit_secant_method():
    response = client.post(
        "/submit_secant_method",
        json={
            "point_0": 1,
            "point_1": 2,
            "tolerance": 0.0001,
            "max_it": 20,
            "equation": "x**3 + 4 * x**2 - 10",
        },
    )
    assert response.status_code == 200
    assert "problem_id" in response.json()


def test_table_results_success(mocker):
    task_id = "054c9a22-f2ae-4b4b-8e11-1bc312d0e995"
    object_name = "".join(["table_", task_id, ".csv"])
    mock_client = mocker.Mock(spec=Minio)
    mocker.patch("api.api.client", mock_client)
    mock_get_file_in_bucket = mocker.patch("api.api.get_file_in_bucket")
    mock_client.stat_object.return_value = Object("results", object_name)
    response = client.get(f"/table_results/{task_id}")
    assert response.status_code == 200
    mock_get_file_in_bucket.assert_called_once()
    mock_client.stat_object.assert_called_once()


def test_problem_statements_success(mocker):
    task_id = "054c9a22-f2ae-4b4b-8e11-1bc312d0e995"
    object_name = "".join(["statement_", task_id, ".csv"])
    mock_client = mocker.Mock(spec=Minio)
    mocker.patch("api.api.client", mock_client)
    mock_get_file_in_bucket = mocker.patch("api.api.get_file_in_bucket")
    mock_client.stat_object.return_value = Object("results", object_name)
    response = client.get(f"/problem_statement/{task_id}")
    assert response.status_code == 200
    mock_get_file_in_bucket.assert_called_once()
```

```
    mock_client.stat_object.assert_called_once()


class FakeS3Error(S3Error):
    def __init__(self):
        pass


def test_table_results_exception_S3Error(mocker):
    task_id = "054c9a22-f2ae-4b4b-8e11-1bc312d0e995"
    mock_client = mocker.Mock(spec=Minio)
    mocker.patch("api.api.client", mock_client)
    mock_client.stat_object.side_effect = FakeS3Error()
    response = client.get(f"/table_results/{task_id}")
    assert response.status_code == 404
    mock_client.stat_object.assert_called_once()


def test_problem_statements_exception_S3Error(mocker):
    task_id = "054c9a22-f2ae-4b4b-8e11-1bc312d0e995"
    mock_client = mocker.Mock(spec=Minio)
    mocker.patch("api.api.client", mock_client)
    mock_client.stat_object.side_effect = FakeS3Error()
    response = client.get(f"/problem_statement/{task_id}")
    assert response.status_code == 404
    mock_client.stat_object.assert_called_once()
```

```
import pytest
import sympy as sy

from methods.methods import bisection_method, fixed_point,
    secant_method


def test_bisection_method_fail():
    """Test if Bisection Method solves problem fails due to
    lack of iterations."""
    x = sy.Symbol("x")
    eq = x**3 + 4 * x**2 - 10
    func = sy.lambdify(x, eq)
    result = bisection_method(1, 2, 0.0001, 10, func)
    assert result == "Procedure failed."


```

```
16 def test_bisection_method_correct():
17     """Test if Bisection Method solves problem correctly."""
18     x = sy.Symbol("x")
19     eq = x**3 + 4 * x**2 - 10
20     func = sy.lambdify(x, eq)
21     result = bisection_method(1, 2, 0.0001, 20, func)
22     assert result[0] == pytest.approx(1.36, 0.01)
23
24
25 def test_fixed_point_fail():
26     """Test if Fixed Point Method solves problem fails due to
       lack of iterations."""
27     x = sy.Symbol("x")
28     eq = 0.5 * (10 - x**3) ** 0.5
29     func = sy.lambdify(x, eq)
30     result = fixed_point(1.5, 0.0001, 10, func)
31     assert result == "Procedure failed."
32
33
34 def test_fixed_point_correct():
35     """Test if Fixed Point Method solves problem correctly."""
36     x = sy.Symbol("x")
37     eq = 0.5 * (10 - x**3) ** 0.5
38     func = sy.lambdify(x, eq)
39     result = fixed_point(1.5, 0.0001, 20, func)
40     assert result[0] == pytest.approx(1.36, 0.01)
41
42
43 def test_secant_method_fail():
44     """Test if Secant Method solves problem fails due to lack
       of iterations."""
45     x = sy.Symbol("x")
46     eq = x**3 + 4 * x**2 - 10
47     func = sy.lambdify(x, eq)
48     result = secant_method(1, 2, 0.0001, 5, func)
49     assert result == "Procedure failed."
50
51
52 def test_secant_method_correct():
53     """Test if Secant Method solves problem correctly."""
54     x = sy.Symbol("x")
55     eq = x**3 + 4 * x**2 - 10
56     func = sy.lambdify(x, eq)
57     result = secant_method(1, 2, 0.0001, 20, func)
```

```
    assert result[0] == pytest.approx(1.36, 0.01)
```

```
import numpy as np
import pytest
from celery.app.task import Task
from celery.exceptions import Ignore
from minio import Minio
from minio.error import S3Error
from urllib3.exceptions import MaxRetryError

from task_manager.result_int import (
    get_file_in_bucket,
    put_file_in_bucket,
    write_results,
)


def test_put_file_in_bucket(mocker):
    mock_client = mocker.Mock(spec=Minio)
    source_path = "source_path.csv"
    bucket_name = "results"
    destination_name = "destination_name.csv"
    mock_client.bucket_exists.return_value = True
    put_file_in_bucket(mock_client, source_path, bucket_name, destination_name)
    mock_client.make_bucket.assert_not_called()
    mock_client.bucket_exists.assert_called_once_with(bucket_name)
    mock_client.fput_object.assert_called_once_with(
        bucket_name, destination_name, source_path
    )


def test_get_file_in_bucket(mocker):
    mock_client = mocker.Mock(spec=Minio)
    source_name = "source_path.csv"
    bucket_name = "results"
    get_file_in_bucket(mock_client, bucket_name, source_name)
    mock_client.get_object.assert_called_once_with(
        bucket_name, source_name
    )


def test_write_results_failure(mocker):
```

```
41      mock_client = mocker.Mock(spec=Minio)
42      mock_put_file_in_bucket = mocker.patch("task_manager.
    result_int.put_file_in_bucket")
43      mock_file = mocker.Mock()
44      mock_task = mocker.Mock()
45      mock_task.request.id = "7933df7a-7688-4a5c-95ab-94289878
    d81d"
46      mock_open = mocker.patch("task_manager.result_int.open")
47      mock_open.return_value.__enter__.return_value = mock_file
48      mock_json_dump = mocker.patch("task_manager.result_int.
    json.dump")
49      problem = {
50          "endpoint_a": 1,
51          "endpoint_b": 2,
52          "tolerance": 0.0001,
53          "max_it": 5,
54          "equation": " x**3 + 4 * x**2 - 10",
55      }
56      result = "Procedure failed."
57      write_results(mock_client, problem, result, mock_task)
58      mock_put_file_in_bucket.assert_called_once()
59      args, kwargs = mock_json_dump.call_args
60      assert args[0]["result"] == result
61
62
63  class FakeS3Error(S3Error):
64      def __init__(self):
65          pass
66
67
68  def test_write_results_exception_S3Error(mocker):
69      mock_client = mocker.Mock(spec=Minio)
70      mock_put_file_in_bucket = mocker.patch("task_manager.
    result_int.put_file_in_bucket")
71      mock_put_file_in_bucket.side_effect = FakeS3Error()
72      mock_file = mocker.Mock()
73      mock_task = mocker.Mock(spec=Task)
74      mock_task.request.id = "7933df7a-7688-4a5c-95ab-94289878
    d81d"
75      mock_open = mocker.patch("task_manager.result_int.open")
76      mock_open.return_value.__enter__.return_value = mock_file
77      mock_json_dump = mocker.patch("task_manager.result_int.
    json.dump")
78      problem = {
```

```
        "endpoint_a": 1,
        "endpoint_b": 2,
        "tolerance": 0.0001,
        "max_it": 5,
        "equation": " x**3 + 4 * x**2 - 10",
    }
    result = "Procedure failed."
    with pytest.raises(Ignore):
        write_results(mock_client, problem, result, mock_task)
    mock_put_file_in_bucket.assert_called_once()
    mock_task.update_state.assert_called_once()


class FakeMaxRetryError(MaxRetryError):
    def __init__(self):
        pass


def test_write_results_exception_S3Error(mocker):
    mock_client = mocker.Mock(spec=Minio)
    mock_put_file_in_bucket = mocker.patch("task_manager.result_int.put_file_in_bucket")
    mock_put_file_in_bucket.side_effect = FakeMaxRetryError()
    mock_file = mocker.Mock()
    mock_task = mocker.Mock(spec=Task)
    mock_task.request.id = "7933df7a-7688-4a5c-95ab-94289878d81d"
    mock_open = mocker.patch("task_manager.result_int.open")
    mock_open.return_value.__enter__.return_value = mock_file
    mock_json_dump = mocker.patch("task_manager.result_int.json.dump")
    problem = {
        "endpoint_a": 1,
        "endpoint_b": 2,
        "tolerance": 0.0001,
        "max_it": 5,
        "equation": " x**3 + 4 * x**2 - 10",
    }
    result = "Procedure failed."
    with pytest.raises(Ignore):
        write_results(mock_client, problem, result, mock_task)
    mock_put_file_in_bucket.assert_called_once()
    mock_task.update_state.assert_called_once()

```

```
120
121 def test_write_results_success(mocker):
122     mock_client = mocker.Mock(spec=Minio)
123     mock_put_file_in_bucket = mocker.patch("task_manager.
    result_int.put_file_in_bucket")
124     mock_file = mocker.Mock()
125     mock_task = mocker.Mock()
126     mock_task.request.id = "7933df7a-7688-4a5c-95ab-94289878
    d81d"
127     mock_open = mocker.patch("task_manager.result_int.open")
128     mock_open.return_value.__enter__.return_value = mock_file
129     mock_json_dump = mocker.patch("task_manager.result_int.
    json.dump")
130     mock_np_save = mocker.patch("task_manager.result_int.np.
    savetxt")
131     problem = {
132         "endpoint_a": 1,
133         "endpoint_b": 2,
134         "tolerance": 0.0001,
135         "max_it": 5,
136         "equation": " x**3 + 4 * x**2 - 10",
137     }
138     result = (1.36517333984375, np.array([[1, 2], [2, 2]]))
139     write_results(mock_client, problem, result, mock_task)
140     assert mock_put_file_in_bucket.call_count == 2
141     args, kwargs = mock_json_dump.call_args
142     assert args[0]["result"] == result[0]
```

APÉNDICE 2: Kubernetes

Configuración

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
spec:
  replicas: 1
  selector:
    matchLabels:
      app: api
  template:
    metadata:
      labels:
        app: api
    spec:
      containers:
      - name: api
        image: api-micro-calc:0.1
        ports:
        - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: api-service
spec:
  selector:
    app: api
  ports:
  - protocol: TCP
    port: 8080
    targetPort: 8080
  type: ClusterIP
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis
spec:
  replicas: 1
```

```
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis
        image: redis:latest
        ports:
        - containerPort: 6379

---
apiVersion: v1
kind: Service
metadata:
  name: redis
spec:
  selector:
    app: redis
  ports:
    - port: 6379
      targetPort: 6379
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: minio
spec:
  replicas: 1
  selector:
    matchLabels:
      app: minio
  template:
    metadata:
      labels:
        app: minio
    spec:
      containers:
      - name: minio
        image: quay.io/minio/minio:latest
        args: ["server", "/data", "--console-address",
```

```
    ":9001"]
19          ports:
20          - containerPort: 9001
21          env:
22          - name: MINIO_ROOT_USER
23            value: "minioadmin"
24          - name: MINIO_ROOT_PASSWORD
25            value: "minioadmin"
26          volumeMounts:
27          - name: minio-data
28            mountPath: /data
29        volumes:
30        - name: minio-data
31          persistentVolumeClaim:
32            claimName: minio-pvc
33 ---
34 apiVersion: v1
35 kind: PersistentVolumeClaim
36 metadata:
37   name: minio-pvc
38 spec:
39   accessModes:
40     - ReadWriteOnce
41   resources:
42     requests:
43       storage: 1Gi
44 ---
45 apiVersion: v1
46 kind: Service
47 metadata:
48   name: minio
49 spec:
50   selector:
51     app: minio
52   ports:
53   - name: api
54     port: 9000
55     targetPort: 9000
56   - name: console
57     port: 9001
58     targetPort: 9001
```

```
1 apiVersion: apps/v1
2 kind: Deployment
```

```
metadata:
  name: worker
spec:
  replicas: 1
  selector:
    matchLabels:
      app: worker
  template:
    metadata:
      labels:
        app: worker
    spec:
      containers:
      - name: worker
        image: worker-micro-calc:0.1
        resources:
          requests:
            cpu: 100m
            memory: 256Mi
          limits:
            cpu: 500m
            memory: 1Gi
```

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: worker-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: worker
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

APÉNDICE 3: Docker

Configuración

```yaml
services:
  cache:
    image: redis:latest
    container_name: redis-micro-calc
    restart: unless-stopped
  object_store:
    image: quay.io/minio/minio:latest
    container_name: minio-micro-calc
    environment:
      MINIO_ROOT_USER: minioadmin
      MINIO_ROOT_PASSWORD: minioadmin
    volumes:
      - ./data_minio:/data
    command: server /data --console-address ":9001"
    ports:
      - "9001:9001"
    restart: unless-stopped
  api:
    image: api-micro-calc:0.1
    container_name: api-micro-calc
    ports:
      - '8080:8080'
    restart: unless-stopped
  worker:
    image: worker-micro-calc:0.1
    container_name: worker-micro-calc
    restart: unless-stopped
```

```
FROM python:3.13
WORKDIR /code
COPY ./requirements.txt /code/requirements.txt
RUN pip install --no-cache-dir --upgrade -r /code/requirements
   .txt
COPY . /code
CMD ["celery", "-A", "task_manager.task", "worker"]
```

```
FROM python:3.13
WORKDIR /code
COPY ./requirements.txt /code/requirements.txt
```

```
4 RUN pip install --no-cache-dir --upgrade -r /code/requirements
     .txt
5 COPY . /code
6 EXPOSE 8080
7 CMD ["fastapi", "run", "api/api.py", "--port", "8080"]
```