

# Herramienta de monitorización basada en soluciones Open Source para equipos Windows y Linux

Trabajo de Fin de Grado

Grado en Ingeniería Informática



6 de Julio de 2022

**Autor**

Pablo Jesús González Rubio

**Tutores**

Alicia García Holgado

Andrea Vázquez Ingelmo

Francisco José García Peñalvo



Dña. Alicia García Holgado, Dña. Andrea Vázquez Ingelmo y D. Francisco José García Peñalvo, profesorado del Departamento de Informática y Automática de la Universidad de Salamanca.

CERTIFICAN:

Que el trabajo titulado “Herramienta de monitorización basada en soluciones Open Source para equipos Windows y Linux” ha sido realizado por D. Pablo Jesús González Rubio, con DNI 70894492M, para la asignatura “Trabajo de Fin de Grado” de la titulación “Grado en Ingeniería Informática de la Universidad de Salamanca”.

Y para que así conste a todos los efectos oportunos.

En Salamanca, a 6 de Julio de 2022

Dña. Alicia García Holgado

Dpto. Informática y Automática

Universidad de Salamanca

Dña. Andrea Vázquez Ingelmo

Dpto. Informática y Automática

Universidad de Salamanca

D. Francisco José García Peñalvo

Dpto. Informática y Automática

Universidad de Salamanca



# Resumen

En la actualidad existen muchos sistemas informáticos críticos que suponen una parte fundamental de particulares, empresas e incluso el gobierno, los cuales contratan administradores de sistemas para gestionar estos sistemas, servidores, y servicios para proveer un rendimiento adecuado en base a sus necesidades. Para tener un mayor control de los sistemas y su estado se utilizan herramientas de monitorización que permiten detectar patrones y eventos que puedan producir un estado negativo en los equipos. Como sistema monitor, una de las funciones inherentes es generar alertas y enviarlas al administrador.

El origen del proyecto fue la necesidad de crear una herramienta de monitorización para Windows y Linux que permitiera recoger datos de uso para posteriormente crear estadísticas y análisis de esta información con el fin de mejorar la organización y gestión del servicio que proveía la empresa en la que se realizaron las prácticas. Aunque existen muchas herramientas similares, se buscaba una que además permitiera ejecutar comandos de forma sencilla para ofrecer versatilidad al administrador de sistemas.

El principal objetivo del presente trabajo ha sido crear un sistema que pueda supervisar no sólo el estado de los equipos, si no ejecutar comandos y recibir información de los procesos que superen un cierto porcentaje de CPU o RAM. Para ello se ha planteado una arquitectura cliente-servidor por el que el cliente recoge información del equipo y las envía, además supervisa si estas métricas superan cierta límite de CPU o RAM para evitarle al servidor comprobar esto, y se lo supera se manda una alerta al servidor. El servidor por otra parte recoge estas métricas y alertas con una API REST, las serializa y las almacena para luego ser tratadas y presentadas. El servidor dispone de tareas en segundo plano que permiten realizar distintas acciones como borrar métricas del sistema una vez superen 15 días, de tal forma que el servidor no colapse por la cantidad de la información que se recoge, o comprobar las alertas para determinar el estado de un equipo y alertar al administrador por distintos medios como pueda ser correo electrónico, o servicios que acepten integraciones *webhook* como son Discord y Slack.

Como conclusión se podría considerar que se han alcanzado todos los objetivos técnicos y personales, a falta de mejorar tanto los conocimientos adquiridos, como la parte más técnica, extendiéndola con más componentes software.

La aplicación tiene todavía múltiples limitaciones y, por ende, múltiples posibles vías de mejora. El sistema utiliza peticiones HTTP para la comunicación con los equipos, mientras que es versátil para el programador, su rendimiento se podría mejorar con *WebSockets*. Dependiendo del caso específico de uso de esta herramienta, se querrá adaptar su arquitectura para satisfacer las necesidades de una empresa mediana o grande. Se pueden realizar más integraciones como las llamadas o mensajes SMS para casos en los que internet no sea accesible. Para garantizar la máxima precisión de la información recogida de los equipos, se puede utilizar un almacenamiento en tiempo real como *InfluxDB*, y presentar estos datos de manera asíncrona con *AJAX* o componentes recargables de *React* para mejorar la experiencia del usuario. Por último, con la cantidad ingente de datos que los equipos envían a lo largo del tiempo, se pueden realizar análisis y crear modelos de inteligencia artificial en base a estos, que sean capaces de predecir y prevenir eventos, además de tener un mayor control de los equipos y cuáles podrían ser de riesgo.

Palabras Clave: API REST, Python, Django, Sistema Distribuido, Monitorización.

# Abstract

Nowadays many critical computer systems are a fundamental part of individuals, companies, and even the government, which hire system administrators to manage these systems, servers, and services to provide adequate performance based on their needs. To better control the systems and their status, monitoring tools are used to detect patterns and events that can produce a negative state in the equipment. As a monitoring system, one of the inherent functions is to generate alerts and send them to the administrator.

The origin of the project was the need to create a monitoring tool for Windows and Linux that would allow the collection of usage data to later create statistics and analysis of this information to improve the organization and management of the service provided by the company where the internship was conducted. Although there are many similar tools, we were looking for one that would also allow executing commands in an uncomplicated way to offer versatility to the system administrator.

The main objective of this work has been to create a system that can monitor not only the status of the equipment but also execute commands and receive information from the processes that exceed a certain percentage of CPU or RAM. To do this we have proposed a client-server architecture whereby the client collects information from the computer and sends them, also monitor whether these metrics exceed a specific limit of CPU or RAM to prevent the server from checking this and if it exceeds an alert is sent to the server. The server on the other hand collects these metrics and alerts with a REST API, serializes them, and stores them for later processing and presentation. The server has background tasks that allow different actions such as deleting metrics from the system once they exceed 15 days, so that the server does not collapse by the amount of information collected, or checking the alerts to determine the status of a team and alert the administrator by various means such as email, or services that accept webhook integrations such as Discord and Slack.

In conclusion, it could be considered that all the technical and personal objectives have been achieved, with the lack of improving both the acquired knowledge and the more technical part, extending it with more software components.

The application still has many limitations and, therefore, many ways of improvement. The system uses HTTP requests for communication with computers, while it is versatile for the programmer, its performance could be improved with *WebSockets*. Depending on the specific use case of this tool, you will want to adapt its architecture to meet the needs of a medium or large enterprise. Further integrations such as calls or SMS messages can be made for cases where the Internet is not accessible. To ensure maximum accuracy of the information collected from computers, you can use real-time storage such as InfluxDB and present this data asynchronously with AJAX or reloadable React components to enhance the user experience. Finally, with the massive amount of data that computers send over time, you can perform analysis and create artificial intelligence models based on this data, which can predict and prevent events, as well as have better control of the computers and which ones could be at risk.

Keywords: REST API, Python, Django, Distributed System, Monitoring.



# Índice de la Memoria

Resumen .....	5
Abstract.....	7
Índice de figuras de Memoria .....	11
1. Introducción.....	13
1.1. Origen del proyecto.....	14
1.2. El proyecto .....	14
2. Objetivos.....	17
2.1. Objetivos Técnicos.....	17
2.1.1. Gestión de los datos.....	17
2.1.2. Autenticación.....	18
2.1.3. Ejecución de comandos.....	18
2.1.4. Gestión de alertas.....	18
2.1.5. Rendimiento óptimo .....	18
2.2. Objetivos Personales .....	19
2.2.1. Planificación temporal.....	19
2.2.2. Gestión del proyecto con herramientas de control de versiones .....	19
2.2.3. Aprendizaje del <i>framework</i> Django .....	19
2.2.4. Mejora personal en el ámbito del desarrollo <i>full-stack</i> .....	20
2.2.5. Mejora en el desarrollo de código limpio.....	20
3. Descripción de la herramienta.....	21
4. Técnicas y herramientas .....	25
4.1. Gestión del proyecto .....	25
4.1.1. Metodologías de elicitación de requisitos.....	25
4.1.2. Metodologías ágiles .....	26
4.1.3. Github Projects.....	27
4.1.4. Control de versiones .....	28
4.1.5. Google Drive .....	29

4.2.	Entorno de desarrollo.....	29
4.2.1.	Lenguajes de programación .....	29
4.2.2.	Frameworks.....	30
4.2.3.	Librerías y dependencias.....	33
4.2.4.	Herramientas de desarrollo .....	36
4.2.5.	Entorno de pruebas.....	38
4.3.	Entorno de despliegue .....	39
4.3.1.	Despliegue.....	39
5.	Aspectos relevantes.....	43
5.1.	Decisiones sobre la arquitectura .....	43
5.1.1.	Primera propuesta.....	43
5.1.2.	Segunda propuesta.....	45
5.1.3.	Arquitectura final.....	46
5.2.	Gestión del Proyecto .....	48
5.2.1.	Fases del diseño y conceptualización.....	48
5.2.2.	Fases de la implementación .....	51
5.2.3.	Despliegue .....	67
6.	Conclusiones .....	73
7.	Líneas futuras.....	77
7.1.	Integraciones.....	77
7.2.	Análisis de datos.....	77
7.3.	Predicción de fallos .....	77
7.4.	Gestor de colas en el agente.....	77
7.5.	Mejora de la arquitectura.....	78
7.6.	Tiempo Real.....	78
7.7.	Uso de WebSockets .....	78
8.	Referencias.....	79

# Índice de figuras de Memoria

Figura 1. Dashboard .....	21
Figura 2. Github Projects.....	27
Figura 3. Benchmark entre FastAPI y Flask (Imagen tomada a 16 de junio de 2022) .....	31
Figura 4. Benchmark entre FastAPI y Uvicorn.....	32
Figura 5. Primer prototipo de arquitectura.....	43
Figura 6. Segundo prototipo de arquitectura.....	45
Figura 7. Prototipo final de arquitectura.....	47
Figura 8. Formulario para la ejecución de comandos .....	53
Figura 9. Fichero de configuración del agente.....	54
Figura 10. Formulario para configurar el agente .....	55
Figura 11. Arquitectura Modelo-Vista -Plantilla.....	56
Figura 12. Modelo CustomUser .....	58
Figura 13. Modelo AbstractUser .....	58
Figura 14. Creación del token .....	59
Figura 15. Diagrama de secuencia para el envío de métricas.....	60
Figura 16. Workers de Redis .....	61
Figura 17. Planificación de las tareas en segundo plano .....	62
Figura 18. Filtrado y paginación.....	63
Figura 19. HTML con datos anidados.....	64
Figura 20. List Comprehension .....	66
Figura 21. Condición en línea .....	66
Figura 22. Diagrama de despliegue.....	67
Figura 23. Servicio de SystemD.....	68
Figura 24. Procesos de Docker-Compose.....	68
Figura 25. Fichero .env.....	69
Figura 26. Fichero Docker-Compose.yml.....	71
Figura 27. Script para el despliegue de Docker .....	71
Figura 28. Pyproject.toml .....	72



# 1. Introducción

En la actualidad existen muchos sistemas informáticos críticos que suponen una parte fundamental de particulares, empresas e incluso el gobierno, las cuales contratan administradores de sistemas para gestionar estos sistemas, servidores, y servicios para proveer un rendimiento adecuado en base a sus necesidades. Para tener un mayor control de los sistemas y su estado se utilizan herramientas de monitorización que permiten generar alertas dependiendo de ciertos parámetros configurables por el usuario como pueden ser el superar un porcentaje de CPU o de RAM determinada, recibir muchas peticiones e inicios de sesión entre otros. En última instancia, las herramientas de monitorización le permiten liberar a su equipo de las tareas de menor importancia, ahorrar tiempo y dinero en las actividades de las operaciones de servicio, reducir e incluso evitar el tiempo de inactividad del sistema, y ayudar a la empresa con su estrategia, presupuestos y planes de mejora continua.

Actualmente existen herramientas software de monitorización de pago que permiten controlar y observar tanto el estado de los equipos como el de la red.

Algunas herramientas de pago existentes son las siguientes: Dynatrace, AppDynamics by Cisco, New Relic, DataDog, Instana, Elastic: Observability, Splunk, ManageEngine OpManager, PRTG Network Monitor, MicroFocus SiteScope, NetApp OnCommand Insight, Virtana VirtualWisdom, CheckMK, Loggly, Dotcom Monitor, Solarwinds: Server & Application Monitor, Sematext Cloud, Sumo Logic, BMC, Helix Operations Management o Logz.

También existen herramientas Open Source o de código abierto como son: Prometheus, Nagios, Grafana, Zabbix, Icinga, Riemann, Sensu o Heroku Metrics.

Alguna de estas herramientas provee de monitorización para sus propios sistemas como puede ser el caso de Heroku Metrics, que almacena información de las instancias desplegadas en Heroku.

Algunas de las más utilizadas hoy en día son Datadog y Prometheus [40], siendo líderes en monitorización de equipos por su servicio, extensibilidad y opciones.

## 1.1. Origen del proyecto

En la primera empresa en la que se estuvo de prácticas existe un departamento comercial que se dedica a vender productos y servicios a particulares y empresas. El departamento de sistemas quería monitorizar el tiempo que los empleados estaban en llamada a lo largo de la jornada laboral para hacer estadísticas y gestionar mejor su forma de trabajar. Es decir, la necesidad de poder monitorizar si un equipo tenía activo el programa de llamadas 3CX.

En este contexto, surge la necesidad de poder monitorizar equipos Windows con la finalidad de poder controlar el tiempo que un equipo tenía un proceso concreto en ejecución. Así como poder monitorizar el resto de servidores Linux de la empresa que proveían de servicios de almacenamiento, de realización de llamadas y servicios internos de gestión, los cuáles eran usados durante todo el día por empleados y clientes. Por otro lado, también se valoró la necesidad de ejecutar comandos en equipos para realizar actualizaciones del software en los múltiples equipos existentes en la empresa.

## 1.2. El proyecto

En el presente trabajo se propone un sistema de monitorización de equipos informáticos Windows y Linux. Es un software de código abierto basado en Python que pretende solucionar ese problema de monitorización de equipos, de gestión de los procesos que se ejecutan en un equipo, y de ejecución de comandos de manera remota.

El sistema ofrece herramientas como:

- La recogida de métricas con información del estado de los equipos, la lista de procesos con su respectiva información.
- La ejecución de órdenes en equipos multiplataforma.
- Un sistema de notificaciones para que el administrador de los equipos pueda recibir alertas a través de distintos medios como correo electrónico o aplicaciones que integren *WebHooks* como son Discord o Slack, y de esta forma pueda evitar tener que comprobar si un disco duro necesita ser cambiado o si el procesador está dando fallo.

El sistema se compone de dos aplicaciones principales: un agente y una aplicación web, de tal manera que siguen una arquitectura cliente-servidor.

El agente es un pequeño programa en Python que utiliza FastAPI como *framework* web para proveer la arquitectura API REST [46]. Este se encarga de recoger métricas de un equipo de manera continua según unos intervalos que el usuario modifica a través de un fichero de configuración. Para poder mandar peticiones y poder atender a las propias de su servicio web, se ejecuta de manera concurrente como una tarea en segundo plano el mandar métricas y alertas a la aplicación web.

Se trata de un proyecto que a nivel técnico requiere de un gran número de herramientas que permitan, no solo monitorizar, sino ser capaz de gestionar peticiones, datos, y asegurar un sistema robusto. En este sentido, la aplicación web basada en Django utiliza múltiples servicios como:

- Nginx como proxy para gestionar los ficheros estáticos y tener un sistema más robusto en cuanto a la gestión de las peticiones.
- Uvicorn como servidor web ASGI. Permite procesar las peticiones y trasladar esta información a la aplicación web de manera rápida.
- El API REST basada en Django REST Framework para la recogida de la información y su posterior serialización para ser almacenada en la base de datos.
- Redis como gestor de colas para las tareas en segundo plano y PostgreSQL como base de datos.

En el caso del agente, para facilitar la instalación de un software que no es un ejecutable, sino un *script* que se interpreta línea a línea, el agente se permite desplegar con gestores de servicios SystemD en Linux mediante un fichero de servicio, y en Windows con Windows Task Scheduler mediante una tarea programable. También se incluye un fichero con las dependencias a instalar y el fichero del entorno virtual que permite realizar una instalación local de estas, para evitar saturar el equipo con dependencias que no se utilizarán en otros proyectos, o para prevenir conflictos de versiones entre unas dependencias y otras.

En el caso de la aplicación web por tener mucho más desarrollo e integraciones con distintos servicios, para poder aunarlos todos y que el administrador no tenga que configurarlos en sus equipos, se utiliza Docker. También permite realizar una instalación automatizada de todos estos servicios y de sus correspondientes dependencias en un entorno virtual. Esto permite al administrador separar el sistema

de monitorización de sus servicios en producción, y en caso de que necesite más recursos para la aplicación, se pueden replicar los contenedores de Docker, o incluso distribuir entre varios equipos con Docker Swarm.



## 2. Objetivos

Para el proyecto de fin de grado se comprenden dos tipos de objetivos: los técnicos y los personales.

### 2.1. Objetivos Técnicos

Respecto a los objetivos técnicos, se trata de los requisitos que se deben cumplir para que el proyecto cumpla con su propósito. El objetivo principal es monitorizar con efectividad distintos equipos y poder gestionar estos datos con la finalidad de que el administrador de sistemas pueda diagnosticar el rendimiento de sus equipos ante distintas circunstancias. Este objetivo se descompone en una serie de objetivos técnicos.

#### 2.1.1. Gestión de los datos

##### 2.1.1.1. Recogida

Uno de los objetivos más importantes es la recopilación de datos: distintas métricas que indicarán el estado y rendimiento del equipo a monitorizar. Cuanta más información se pueda obtener de los dispositivos, más información se tendrá del estado de este, y se podrá incluso utilizar para prever y detectar con antelación eventos de algo desgaste y consumo en circunstancias similares.

##### 2.1.1.2. Envío

El envío de datos a los servidores principales es otro objetivo técnico que cumplir, pues la información se debe centralizar en una (o varias) bases de datos para su posterior tratamiento y gestión. Para ello se pueden optar por distintas técnicas, desde MQTT, pasando por HTTP hasta *WebSockets*.

##### 2.1.1.3. Tratamiento

Algunos de los datos se envían en un formato incorrecto: al enviar la información, se omiten ciertos caracteres como puedan ser los saltos de línea o los retornos de carro, es por razón que se deben tratar antes de ser enviados.

En otros casos el formato de la información este previene de una mayor utilidad: es el caso de no encontrarse ordenados, lo que luego evita una indexación con un consumo computacional bajo, y aumenta el consumo de rendimiento al buscar palabras clave.

#### **2.1.1.4. Presentación**

El usuario debe poder visualizar la información de una manera sencilla y que le permita diagnosticar rápidamente la razón por la que su equipo esté fallando, tenga un consumo elevado de recursos o tenga otro tipo de incidencia. Debe poderse mostrar gráficamente estos datos, y el estado actual del equipo.

#### **2.1.2. Autenticación**

El sistema tiene que poder gestionar usuarios y filtrar datos en base a estos para que unos usuarios no puedan acceder a los datos del resto [8]. También se utilizará para la seguridad de la arquitectura cliente-servidor, de tal manera que cualquier equipo no pueda mandar datos falseados al servidor, o pueda ejecutar comandos en el agente.

#### **2.1.3. Ejecución de comandos**

Uno de los puntos principales era poder ejecutar comandos de forma remota en los equipos gestionados por la aplicación. Esto permitiría al administrador gestionar de manera flexible y centralizada sus equipos desde la página web, evitando tener que recurrir a otros medios, que quizá no estén disponibles.

#### **2.1.4. Gestión de alertas**

La gestión de las alertas es inherente a la gestión de métricas. Esto permitirá darle tranquilidad al administrador para que no esté pendiente del estado de sus equipos, pues le permitirá gestionar los límites a partir de los cuales se empiecen a generar alertas. Si se dan ciertas condiciones, se notificará al administrador por los medios disponibles.

#### **2.1.5. Rendimiento óptimo**

Un punto clave es que el servidor principal no tenga una carga de trabajo muy elevada en la recogida y tratamiento de los datos. La relación de peticiones que el

servidor puede recibir viene determinada por el número de equipos que monitoriza y la frecuencia del envío de las métricas y las alertas. En un caso con un número alto de equipos, y una frecuencia baja de envío, se podría incluso realizar una denegación de servicios.

Por otro lado, si el tratamiento de datos es costoso computacionalmente por su complejidad o tamaño de la información, el equipo podría tener un consumo alto de CPU y demorarse en realizar otras tareas.

## **2.2. Objetivos Personales**

### **2.2.1. Planificación temporal**

Como objetivo personal se plantea compatibilizar el desarrollo del Trabajo Final de Grado con el curso, así como con un trabajo de jornada completa. Es una labor que requerirá de una planificación y gestión de las tareas muy estricta, así como de una persistencia continua en el desarrollo del proyecto.

### **2.2.2. Gestión del proyecto con herramientas de control de versiones**

Teniendo en cuenta la utilización de sistemas de gestión de versiones como Github en el día a día de muchas empresas y desarrolladores, se quería aprender a manejar de manera adecuada estas herramientas para una segmentación y clasificación de las tareas. Como objetivo se plantea mejorar en el uso de estas herramientas y todas las posibilidades que ofrecen con el fin de mejorar como desarrollador y contribuyente al mundo Open Source. Algunas de las herramientas que provee son la gestión de versiones, *timelines*, *rollbacks* y la gestión de incidencias o *issues* para realizar un seguimiento de los proyectos y aplicaciones.

### **2.2.3. Aprendizaje del *framework* Django**

En un principio el desarrollo del proyecto se planteó con el framework web Flask, por su facilidad de aprendizaje y de desarrollo. Al realizar las prácticas extracurriculares y conocer el framework web Django, se decidió modificar el proyecto en aras de mejorar el diseño, escalabilidad, flexibilidad, modularidad y rendimiento.

## 2.2.4. Mejora personal en el ámbito del desarrollo *full-stack*

Desarrollar una aplicación en sus aspectos visuales y sus aspectos funcionales es una de las habilidades más demandadas hoy día.

El desarrollo *full-stack* implica tareas como la gestión del *frontend* y sus tecnologías: librerías CSS, librerías Javascript, *frameworks* web, gestión asíncrona de las peticiones, interfaces *responsive*, un diseño limpio de la interfaz, útil y usable.

Por otro lado, también implica la gestión del *backend* y sus tecnologías: *frameworks* web, servicios de gestión de datos como gestores de colas, bases de datos, servidores ASGI/WSGI entre otros. Junto con el despliegue de estos mediante herramientas como Docker o Kubernetes en entornos de desarrollo y de producción en equipos de una intranet o en servidores *cloud* como AWS o DigitalOcean.

Por último, en ocasiones la gestión de ciertas tareas como la integración continua que corresponde más a un papel de un ingeniero DevOps, también se le atribuye al desarrollador *full-stack*.

En el presente proyecto se ha intentado reunir cada una de las facetas anteriormente mencionadas a excepción de la integración continua.

## 2.2.5. Mejora en el desarrollo de código limpio

Desarrollar un código fácilmente legible hasta el punto de no hacer falta documentación para entenderlo, modular para poder integrarlo en otras aplicaciones de tal manera que sea un componente, reducir el acoplamiento y aumentar la cohesión refactorizando funciones y clases para facilitar esta modularización, evitar “Code Smells” como funciones forzadas que pueden dar lugar a errores si estos no se tratan, errores de la arquitectura de la información y de su diseño, entre muchas otras características son lo que hacen un código limpio.

Intentar mejorar en este sentido es muy importante para proveer de soluciones óptimas y reutilizables, sobre todo si es de cara a la comunidad Open Source.

### 3. Descripción de la herramienta

La aplicación del presente proyecto es una herramienta de monitorización de equipos Linux y Windows que permite recoger información en forma de métricas, y gestionar límites de CPU y RAM para generar alertas. Esta información se guarda en una base de datos, y sus datos son accesible a través de un API REST y de una aplicación web. La aplicación web permite gestionar múltiples equipos, así como la configuración de estos, y permite visualizar los datos recogidos en forma de gráficas [Figura 1] para tener una visibilidad general del estado del equipo, así como en formato HTML para los datos del equipo, y la información específica de cada métrica. También se permiten configurar dos sistemas de notificación como son correos electrónicos, y *WebHooks*, tantos como desee el usuario, por donde recibirán alertas referentes a sus equipos.

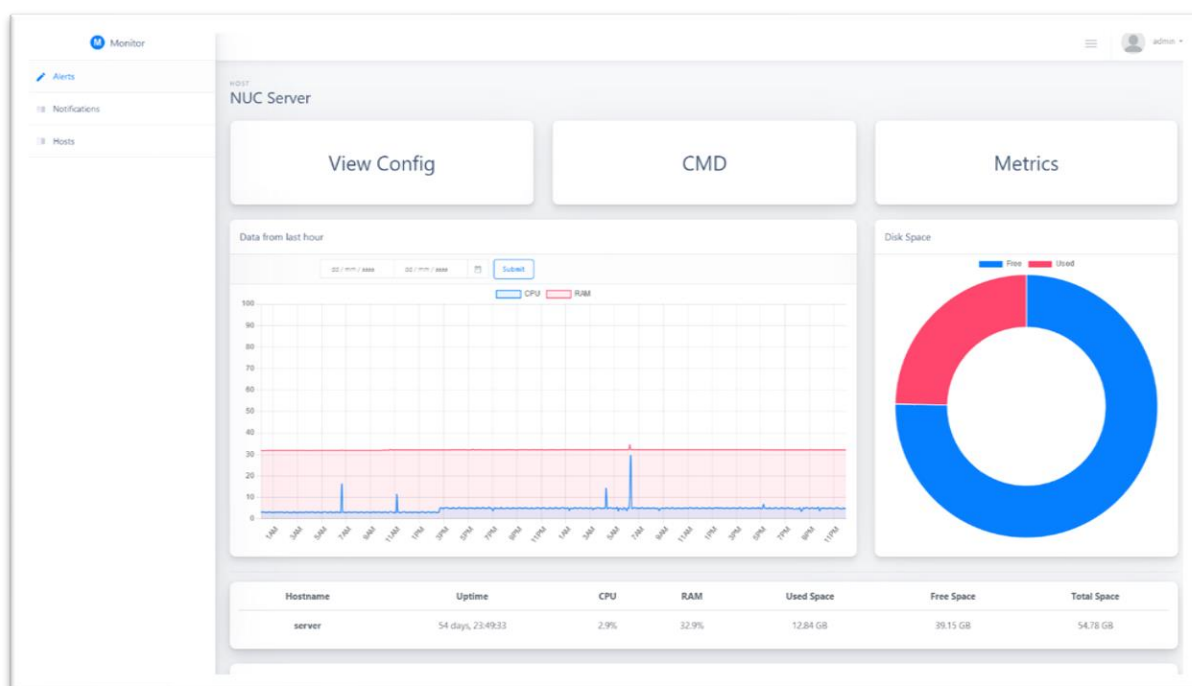


Figura 1. Dashboard

El sistema funciona de la siguiente manera: se despliega en los equipos a monitorizar un pequeño servicio denominado “agente”, el cuál recoge información del software y hardware del equipo y la envía de forma periódica al servidor con una petición HTTP POST. El intervalo de tiempo para el envío de métricas y otras características del agente se pueden configurar en un fichero con formato JSON. Es importante hacer notar que no sólo envía datos, si no que también permite recibir datos mediante el API REST

FastAPI, más concretamente puede recibir peticiones para ejecutar comandos, o para mostrar las métricas del sistema en ese momento concreto.

El servidor por otra parte se compone de dos partes principales, la página web y el API REST. El API REST basado en Django REST Framework se encarga de recoger las peticiones de los equipos y serializarlas para almacenarlas en la base de datos.

La aplicación web basada en Django tiene varios apartados:

- La gestión de usuarios: se permite a usuarios anónimos registrarse, editar sus datos, modificar su contraseña y poder recuperarla en caso de haberla olvidado, iniciar sesión y cerrar sesión.
- La gestión de los equipos: se permite añadir equipos al sistemas configurando su dirección IP, puerto donde escucha el API REST del agente y un nombre para referenciarlo en el sistema, mostrar la lista de los equipos añadidos, así como poder volver a editar los datos del equipo, eliminar este del sistema y mostrar información detallada del equipo y de su evolución temporal.
- La gestión de las métricas: cada agente tiene una serie de métricas que envía el agente. Se puede mostrar una lista paginada de estas métricas ordenadas de tal manera que se vean las últimas recibidas. Esta lista también se puede filtrar por fechas. Cada métrica en particular contiene la información que el agente le ha enviado, la cual se permite mostrar. También se permite eliminar métricas del sistema [Figura 18, Figura 19].
- La gestión de las alertas: el sistema dispone de tareas en segundo plano que se encargan de detectar ciertos patrones en las métricas o la ausencia de ellas, modifica el estado del equipo en el sistema y alerta consecuentemente al administrador de sistemas por varios medios. El administrador puede configurar que se le notifique en las direcciones de correo electrónico que especifique, así como también se le pueden enviar alertas a través de integraciones *WebHook* que tiene aplicaciones como Discord o Slack.
- La gestión de la configuración de los equipos: configurar el agente mediante su fichero JSON es tedioso y muy dado a error, por lo que el agente podría dejar de funcionar adecuadamente. Para ello se ha creado un formulario que permite almacenar la configuración del agente, y presentarla en formato JSON para que pueda copiar-pegarse en el fichero directamente, o exportarse si el usuario así lo desea [Figura 10].

- La ejecución de comandos: en el equipo se permite la ejecución remota de comandos, tanto en los sistemas Windows como en los sistemas Linux. Para ello existe un formulario dentro de la información detallada de cada agente, que permite introducir un comando, un tiempo máximo de espera de respuesta a la petición, y un tiempo máximo de ejecución del comando. De esta manera se evitan posibles errores de red, y la ejecución de comandos interactivos finaliza adecuadamente, evitando así el malgasto de recursos en los equipos [Figura 8].

Es importante destacar que el sistema es seguro a nivel de autenticación y permisos: para toda acción el usuario debe de haber iniciado sesión, tanto en la aplicación web, como en el API REST, como para ejecutar comandos en el equipo remoto. Cada usuario dispone de un *token* único que permite actuar como su usuario y contraseña: los agentes utilizan este *token* para autenticarse contra el sistema, y el servidor lo utiliza para poder ejecutar comandos en los equipos.

La aplicación al no ser ni un instalable ni un ejecutable requiere de un despliegue, es por ello que el agente se ejecuta como un servicio en el caso de Linux mediante SystemD, y como una tarea en el caso de Windows mediante Windows Task Scheduler.

La aplicación web al disponer de múltiples servicios como Redis, Nginx y PostgreSQL, para evitar configurar estos servicios, se despliega en conjunto con Docker.

Para poder probar la aplicación web, se ha habilitado el usuario “tfg” con contraseña “defensaTFGus@l” en la URL: <http://nonuser.onthewifi.com>.





## **4. Técnicas y herramientas**

### **4.1. Gestión del proyecto**

En la gestión del proyecto se han involucrado distintas tecnologías y metodologías para facilitar la organización y manejo de las distintas partes a realizar.

#### **4.1.1. Metodologías de elicitación de requisitos**

Las metodologías de elicitación de requisitos permiten obtener información sobre el negocio del cliente y los requisitos fundamentales que han de cumplirse para llegar al sistema o aplicación objetivo.

Una correcta selección de requisitos facilita en gran medida la fase de diseño de la aplicación y el resto de las fases del ciclo de vida del producto final. Si desde un primer momento se realiza de forma adecuada y se obtienen unos requisitos útiles y bien descritos, puede facilitar la reducción de correcciones y cambios en las siguientes fases e iteraciones del producto.

La metodología de Amador Durán y Beatriz Bernárdez (Departamento de Lenguaje y Sistemas Informáticos de la Universidad de Sevilla) [17] propone las siguientes fases:

1. Obtener información sobre el dominio del problema y el sistema actual.
2. Preparar y realizar las reuniones de obtención/negociación.
3. Identificar/revisar los objetivos del sistema.
4. Identificar/revisar los requisitos de información.
5. Identificar/revisar los requisitos funcionales.
6. Identificar/revisar los requisitos no funcionales.
7. Priorizar objetivos y requisitos.

En el proyecto se ha utilizado esta metodología para poder describir las partes a realizar en la aplicación, desde la información (requisitos de información) que se desea obtener, almacenar y gestionar, hasta la propia funcionalidad explícita (requisitos funcionales) e implícita (requisitos no funcionales) que la aplicación debe tener para garantizar cumplir ciertos objetivos. También ha ayudado al no haber negociación con

distintas partes como pueda ser un cliente, el contrastar las necesidades actuales del mercado y las herramientas existentes, en favor de esta aplicación.

### **4.1.2. Metodologías ágiles**

Las metodologías ágiles son una forma de gestionar un proyecto dividiéndolo en varias fases. Implica la colaboración constante con las partes interesadas y la mejora continua en cada etapa. Una vez que comienza el trabajo, los equipos pasan por un proceso iterativo de planificación, diseño, desarrollo, testeo, ejecución y evaluación.

Para realizar un desarrollo del software organizado y que tuviera una cierta trazabilidad se ha utilizado el marco SCRUM junto con la agrupación de tareas mediante paneles Kanban. La agrupación en tareas mediante los paneles Kanban para evaluar las tareas a realizar en los distintos sprints ha facilitado centrarse en las partes software más importantes a desarrollar, así como tener un seguimiento de las realizadas anteriormente, y las que quedarían por realizar. Tanto el marco SCRUM y el sistema de agrupación Kanban han sido cruciales para una organización y desarrollo eficiente.

SCRUM es un marco para desarrollar, entregar y mantener productos en un entorno complejo, con un énfasis inicial en el desarrollo ágil de software. Está diseñado para equipos de diez miembros o menos, que dividen su trabajo en objetivos que se pueden completar en iteraciones de tiempo limitado, llamadas *sprints*, que no duran más de un mes y, por lo general, dos semanas [3].

El equipo de scrum evalúa el progreso en reuniones diarias con un límite de tiempo de 15 minutos o menos, llamadas scrums diarios (una forma de reunión de pie). Al final del sprint, el equipo realiza dos reuniones más: la revisión del sprint, que demuestra el trabajo realizado a las partes interesadas para obtener retroalimentación, y la retrospectiva del sprint, que permite al equipo reflexionar y mejorar.

En el presente proyecto se han realizado reuniones bisemanales en Slack y Google Meet con las tutoras del proyecto. Explicando los avances significativos de la aplicación y recibiendo una retroalimentación sobre los objetivos actuales y planificar los siguientes objetivos a cumplir con el fin de poder seguir progresando y mejorando el producto final. En los equipos SCRUM existen 3 roles principales: el *product owner*, el *SCRUM master*, y el equipo de desarrollo. Las tutoras han tenido el rol de SCRUM master, supervisando y asegurando que el proyecto llegaba a buen puerto, así como eliminando

los impedimentos existentes para la realización del proyecto mediante la retroalimentación en las continuas iteraciones que han sido las reuniones. Por otra parte, el *product ownery* y el equipo de desarrollo han sido el propio estudiante, pues se ha encargado de desarrollar el producto y maximizar su valor mediante esta memoria.

### 4.1.3. Github Projects

Para la gestión de las tareas del proyecto se ha empleado Github Projects. [Figura 2]. Se trata de una funcionalidad nueva de Github que permite agrupar las incidencias y tareas por bloques siguiendo el método Kanban: a hacer, en desarrollo y hecho, aparte de poder incluir nuevos grupos para una gestión más eficiente [2].

Esto ha permitido centrarse en ciertas tareas en vez de intentar avanzar en múltiples tareas a la vez, lo que se ha visto reflejado en avances significativos. En otras ocasiones que se intentó atender a varios puntos a programar, resultó en una confusión y ralentización importante en el desarrollo.

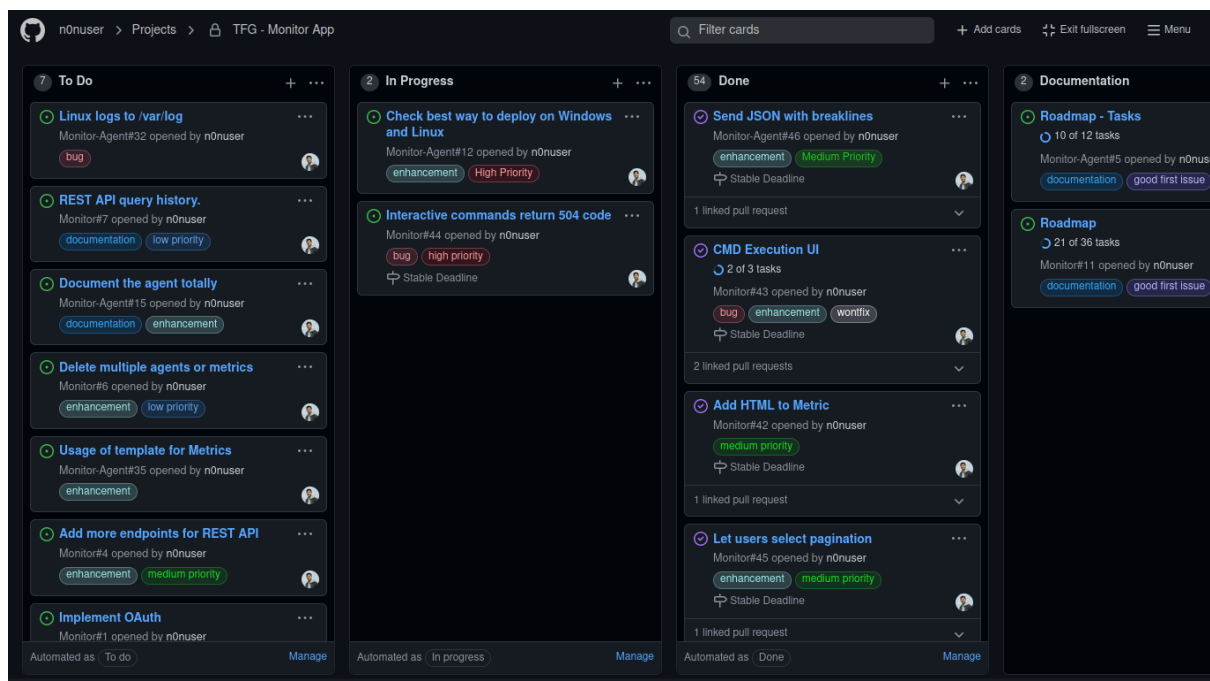


Figura 2. Github Projects

Se ha utilizado esta herramienta para la gestión de tareas, y no otras como puedan ser Trello o Jira, por su integración directa con la gestión de incidencias y partes

a realizar de la aplicación. Permitiendo de una manera óptima integrar el flujo de trabajo con la gestión del proyecto.

#### **4.1.4. Control de versiones**

Para el control de versiones se ha utilizado Git con Github como plataforma. Se trata de un sistema que registra los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo para poder recuperar versiones específicas más adelante [25].

Se ha utilizado para gestionar los cambios del código y registrar los avances en caso de que se necesitara volver a una versión anterior, o en caso de eliminar una parte necesaria, que se implementó anteriormente y se borró, poder volver a utilizarla. También resulta útil pues al segmentar las partes a programar, se puede buscar esa parte en concreto mediante su *commit*, y ver cómo se desarrolló y el código que lo compone para aplicarlo a proyectos y aplicaciones similares.

Se ha utilizado Git por ser la herramienta de control de versiones más conocida y con más documentación (tanto oficial como no oficial) que existe actualmente. Otra de las razones ha sido su uso prolongado a lo largo del Grado y durante Bachillerato.

Un *issue* o incidencia en un repositorio puede ser un error por corregir, una petición para añadir una nueva opción o característica, una pregunta para aclarar algún tema que no está correctamente aclarado, una forma de organizar y gestionar las distintas tareas a realizar por el equipo o cualquier otra cosa que pueda facilitar al desarrollador su labor en la gestión de la aplicación o proyecto. Github permite asignar estas incidencias a los distintos colaboradores del código y clasificarlas por etiquetas [1].

Se ha utilizado para gestionar las tareas a realizar y clasificarlas por su orden de prioridad. Unido a la gestión de proyectos Kanban de Github, ha sido la forma de organizar las diferentes partes a programar y mejorar del código.

Se ha utilizado la gestión de incidencias en Github por encima de otras en Gitlab por la facilidad que Github ofrece en términos de configuración y administración de los repositorios. Gitlab está pensado para ser utilizado en empresas de forma interna, mientras que Github permite un enfoque más orientado al código libre.

### **4.1.5. Google Drive**

Google Drive es un servicio gratuito de almacenamiento en la nube que permite a los usuarios almacenar y acceder a archivos en línea. El servicio sincroniza los documentos almacenados, las fotos y mucho más en todos los dispositivos del usuario, incluidos los dispositivos móviles, las tabletas y los ordenadores.

Se ha utilizado para trabajar de manera colaborativa con las tutoras y gestionar la documentación referente a la memoria, tanto el almacenamiento de los distintos diagramas como de los documentos, y su edición *online*.

Google Drive ha sido la decisión clara por encima de otras herramientas de gestión de documentos por su facilidad de gestión de estos a través de las herramientas de las que dispone y por permitir la edición de documentos a través de un navegador web, en cualquier dispositivo, facilitando así la edición de la memoria desde cualquier lugar.

## **4.2. Entorno de desarrollo**

Para el desarrollo del proyecto se ha utilizado *software* con distintos fines. A continuación se describen clasificados por su utilidad.

### **4.2.1. Lenguajes de programación**

#### **4.2.1.1. Python 3**

Python es un lenguaje de programación interpretado, orientado a objetos y de alto nivel con semántica dinámica. Sus estructuras de datos de alto nivel, combinadas con la tipificación y la vinculación dinámicas, lo hacen muy atractivo para el desarrollo rápido de aplicaciones, así como para su uso como lenguaje de scripting o pegamento para conectar componentes existentes. La sintaxis de Python hace hincapié en la legibilidad y, por tanto, reduce el coste de mantenimiento de los programas. Python admite módulos y paquetes, lo que fomenta la modularidad del programa y la reutilización del código [45].

Python ha sido la base fundamental de todo el proyecto, para el desarrollo de las distintas partes se ha utilizado librerías y *framework* como puedan ser Django o FastAPI.

Se ha utilizado Python por su versatilidad, legibilidad, facilidad de aprendizaje, su fuerte imposición en el mercado actual junto con los lenguajes basados en Javascript, su amplia gama de aplicaciones en el mundo real como puede ser Inteligencia Artificial, IoT, Big Data y demás. La razón principal por la que no se han utilizado otros lenguajes como C o Java es porque por su diseño no están orientados a un desarrollo versátil en multitud de ámbitos, o en este caso más concretamente por la escasa integración directa con *frameworks* Web. En casos como Javascript, no se ha empleado tecnologías basadas en este lenguaje de programación por la dificultad de realizar avances significativos en un periodo breve de tiempo, y la nula integración con el sistema operativo para la ejecución de tareas y comandos en el sistema operativo debido a que los navegadores modernos aíslan el entorno de ejecución para restringir el acceso a los ordenadores de los usuarios.

## **4.2.2. Frameworks**

### **4.2.2.1. FastAPI**

FastAPI es un *framework* web moderno y de alto rendimiento, para la construcción de APIs con Python 3.6+. Sus características clave son la rapidez del servicio, haciéndolo uno de los más potentes para Python al nivel de NodeJS o Go; está basado en el estándar de OpenAPI; y el código es muy similar al de Flask, por lo que portar una aplicación anterior requiere de un esfuerzo mínimo [18].

FastAPI se ha utilizado para la construcción del agente, esto ha sido así por ser un microservicio que no requiere de una base de datos ni una interfaz. Ha facilitado la creación del API REST del agente de forma rápida y su librería FastAPI Utils ha provisto la creación de tareas en segundo plano (hilos concurrentes) sin requerir para ello Celery o Redis [43].

En lugar de Flask se ha preferido emplear FastAPI por tener una sintaxis muy similar y por su mejora en rendimiento de casi un 85% más [Figura 3]. En la página oficial de FastAPI en el apartado “Benchmarks” se explica qué ventajas tiene FastAPI en relación con otros *frameworks* como Django, Flask o Sanic [20].

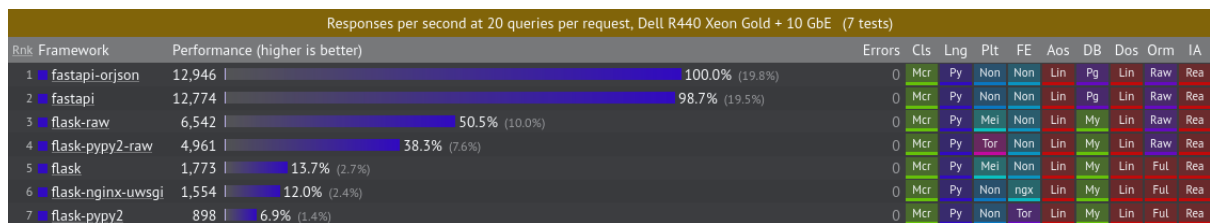


Figura 3. Benchmark entre FastAPI y Flask (Imagen tomada a 16 de junio de 2022)

#### 4.2.2.2. Django

Django es un *framework* web de alto nivel en Python que fomenta un desarrollo rápido y un diseño limpio y pragmático. Sigue una versión similar del Modelo Vista Controlador que facilita la creación y gestión de interfaces junto con una administración incorporada de la base de datos. Uno de los aspectos más potentes es la gestión ORM (Object-Relational Mapper) que permite obtener datos de la base de datos y tratarlos como objetos de Python, pudiendo acceder así a las distintas propiedades de una tabla a través de los atributos del objeto [13, 15].

Django sigue un modelo Model-Template-View (MTV) muy similar al Modelo-Vista-Controlador (MVC) por el que se separa la gestión de los datos (Modelo), de la gestión de estos (Vista) y su presentación (Plantilla). Esto tiene múltiples ventajas como pueden ser un proceso de desarrollo más rápido, la capacidad de proporcionar múltiples vistas e interfaces, el soporte para peticiones asíncrona con herramientas como AJAX y el hecho de que la modificación de una parte no afecte a todo el modelo.

Actualmente Django es utilizado por casi 84,488 páginas web de acuerdo con SimilarTech (). Y es empleado por empresas como Disqus, Youtube, Instagram, Spotify, Bitbucket, Dropbox, Mozilla, Pinterest, The Washington Post y Eventbrite [47, 46].

Una de las mayores ventajas que ofrece, es poder integrar en una interfaz web todas las herramientas de Python existentes y por haber de forma nativa. Desde análisis de datos y estadística, pasando por inteligencia artificial y llegando hasta IoT y más.

Otra de las ventajas grandes que supone como framework, es la gestión de múltiples casos de uso como pueda ser la gestión de usuarios, la gestión de la base de datos y la optimización de ciertos elementos web, porque o viene incluido por defecto, o se puede extender su funcionalidad mediante Middleware, y librerías externas tanto de Python, Javascript, Node y demás. Integrarlo con otros sistemas resulta muy intuitivo.

Django se ha utilizado para el pilar principal del proyecto que ha sido la plataforma web. Su sistema ORM ha facilitado un desarrollo ágil de las consultas a la base de datos y la creación de las tablas, así como los *triggers* al realizar cualquier consulta de tipo CRUD. En lo relativo a la interfaz, el sistema de plantillas de Django ha facilitado la creación de interfaces dinámicas en HTML con la sintaxis de Python.

Pese a no ser tan rápido por defecto como FastAPI, la aplicación web se despliega con Uvicorn, garantizando una velocidad muy similar a la ofrecida por FastAPI (pues este a su vez utiliza Uvicorn) [Figura 4].

Responses per second at 20 queries per request, Dell R440 Xeon Gold + 10 GbE (3 tests)													
Rnk	Framework	Performance (higher is better)	Errors	Cls	Lng	Plt	FE	Aos	DB	Dos	Orm	IA	
1	fastapi-orjson	12,946   100.0% (19.8%)	0	Mcr	Py	Non	Non	Lin	Pg	Lin	Raw	Rea	
2	fastapi	12,774   98.7% (19.5%)	0	Mcr	Py	Non	Non	Lin	Pg	Lin	Raw	Rea	
3	uvicorn	12,663   97.8% (19.4%)	0	Plt	Py	Non	Non	Lin	Pg	Lin	Raw	Rea	

Figura 4. Benchmark entre FastAPI y Uvicorn

La razón más importante por la que se ha utilizado Django en vez de otras plataformas de desarrollo web es su integración directa con múltiples bases de datos, el panel de administración integrado, la modularidad que ofrece su modelo MVT (Model-View-Template) y la facilidad de desarrollo y extensibilidad que esto ofrece. También proporciona distintos Middleware que modifican el comportamiento de la página web y ofrece características completas con añadir una línea, como pueda ser el cacheado en el servidor de las páginas, modificación dinámica de formularios, entre otras. También posee librerías creadas por usuarios que facilitan ciertas interacciones e integraciones con otros sistemas, como pueda ser OAuth.

### 4.2.2.3. Django REST Framework

El *framework* Django REST es un conjunto de herramientas potente y flexible para construir APIs web. Una de las características más importantes es su interfaz web y sistemas de autenticación mediante usuario y contraseña, tokens o incluso OAuth. Permite serializar JSON a la gestión ORM de Django para almacenar los datos en una base de datos a conveniencia: PostgreSQL, Microsoft SQL Server, MongoDB, etc.

DRF se ha utilizado para recibir datos de los agentes y almacenarlos en la base de datos. Pese a que la aplicación no es RESTful por gestionar la información directamente con la base de datos, la API REST dispone de todos los métodos CRUD con el conjunto de modelos con los que el usuario puede interactuar desde la página web. Se ha utilizado



también para garantizar la integridad de la base de datos previniendo el falseado de estos mediante un sistema de autenticación por tokens y la gestión de permisos por usuarios, y grupos de usuarios [9].

En lugar de *frameworks* especializados en APIs REST como Flask o FastAPI se ha utilizado Django REST Framework por su integración directa con Django, su arquitectura MVT y la gestión de los datos con la base de datos mediante serializadores. Los serializadores simplemente convierten los datos en formato JSON a los campos establecidos en la tabla correspondiente [43].

### **4.2.3. Librerías y dependencias**

#### **4.2.3.1. Django Import Export**

Es una librería de Django utilizada para importar y exportar datos de la base de datos en distintos formatos como puedan ser XLS, CSV, TSV, JSON, YAML, Pandas DataFrames, HTML, Jira, ODS y DBF [24]. Se ha utilizado para permitir exportar la información de los agentes y sus métricas a ficheros más convenientes para el usuario. Se realizaría desde la interfaz de administración.

#### **4.2.3.2. Django Crispy Forms**

Es una librería que permite renderizar los formularios de las plantillas con elementos CSS de distintas librerías CSS como puedan ser Bootstrap 3 y 4 y Tailwind entre otras [21]. Se ha utilizado para configurar los elementos visuales de los formularios dinámicos. Estos formularios por defecto tienen el CSS de Django, y para editar el CSS de estos, hay que modificar el formulario y añadirle las propiedades a cada elemento, o en la propia plantilla asociar el CSS (lo cual no siempre es posible por el desconocimiento o variabilidad de los modelos de la base de datos). Crispy Forms soluciona esto asignándole el patrón de Bootstrap a estos elementos dinámicamente.

#### **4.2.3.3. Django Minify HTML**

Minify HTML es una librería que permite minimizar el HTML que se envía al recibir una petición, minimizando caracteres redundantes como puedan ser los espacios, para que la página cargue con mayor rapidez [14].

#### **4.2.3.4. Django Environ**

Django Environ es una librería que permite recoger variables de entorno del sistema y utilizarlas dentro del código como variables. Esto permite aislar datos críticos como *tokens* o contraseñas y evita hardcodear estos datos en el propio código, haciéndolo así más seguro [49].

#### **4.2.3.5. Whitenoise**

Whitenoise es una librería que permite gestionar los ficheros estáticos como puedan ser plantillas de estilos CSS o imágenes, y facilitar el manejo de estos en desarrollo y en producción [50].

#### **4.2.3.6. Requests**

Requests es una librería de Python preparada para hacer peticiones HTTP de manera sencilla. Se ha utilizado para enviar datos a los *webhooks* y los comandos a los agentes.

#### **4.2.3.7. Redis, RQ, Django RQ y RQ Scheduler**

Los cuatro elementos son librerías que se han utilizado en la aplicación web para conectar Django con el bróker de mensajes Redis. Han permitido ejecutar tareas en segundo plano y planificar estas [36].

#### **4.2.3.8. FastAPI Utils**

FastAPI Utils es la librería adicional de FastAPI que incluye una serie de métodos, clases y decoradores que permiten la ampliación del catálogo de opciones de FastAPI [19].

De esta librería se ha utilizado concretamente los decoradores “repeat\_every” y “on\_event” que han permitido lanzar tareas en segundo plano en el agente, como es mandar la información. Utilizar esta librería ha permitido evitar la gestión de hilos y concurrencia con los datos entre el servidor principal de FastAPI y estas tareas en segundo plano.

#### **4.2.3.9. ChartJS**

ChartJS es una librería Javascript que permite el renderizado de gráficas dados unos datos. La manera en la que se ajustan los parámetros de la gráfica y en la que se gestionan los datos a renderizar facilita enormemente su uso [5].

Se ha utilizado para las gráficas de alertas, de rendimiento del equipo y de visualización del espacio ocupado y del espacio libre.

#### **4.2.3.10. JQuery**

jQuery es una biblioteca de JavaScript rápida, pequeña y rica en funciones. Facilita cosas como el recorrido y la manipulación de documentos HTML, el manejo de eventos, la animación y el Ajax mediante una API fácil de usar que funciona en multitud de navegadores. Es versátil y extensible.

Se ha utilizado para algunas funciones de la interfaz web tales como poder seleccionar elementos del navegador y dar apoyo al resto de librerías Javascript.

#### **4.2.3.11. Bootstrap**

Bootstrap es una enorme colección de piezas de código reutilizables y versátiles escritas en CSS, HTML y JavaScript. Dado que también es un framework, todas las bases ya están puestas para el desarrollo web responsive, y lo único que tienen que hacer los desarrolladores es insertar el código [32].

Se ha utilizado para programar la interfaz web y definir sus elementos visuales de una manera rápida y sencilla. La documentación y el framework han evitado tener que lidiar con desajustes visuales que empeorarían la experiencia del usuario.

En lugar de TailwindCSS u otras librerías CSS se ha usado Bootstrap por tener más conocimiento de la sintaxis de clases CSS que utiliza, y por utilizar una plantilla basada en Bootstrap.

#### **4.2.3.12. Moment.js**

Moment.js es un framework JavaScript de código abierto independiente que gestiona objetos de tipo fecha, eliminando así los objetos de tipo fecha nativos de Javascript. Moment.js hace que la fecha y la hora sean fáciles de mostrar, formatear,

analizar, validar y manipular mediante una API limpia y concisa. Moment.js se utiliza para gestionar las fechas en las gráficas de ChartJS.

#### **4.2.3.13. PSUtil**

Es una biblioteca multiplataforma para recuperar información sobre los procesos en ejecución y la utilización del sistema (CPU, memoria, discos, red, sensores) en Python. Es útil principalmente para la monitorización del sistema, la elaboración de perfiles, la limitación de los recursos de los procesos y la gestión de los procesos en ejecución. Implementa muchas funcionalidades ofrecidas por las herramientas de línea de comandos de UNIX como: ps, top, lsof, netstat, ifconfig, who, df, kill, free, nice, ionice, iostat, iotop, uptime, pidof, tty, taskset, pmap [42].

### **4.2.4. Herramientas de desarrollo**

#### **4.2.4.1. Poetry**

Poetry es un gestor de entornos virtuales que permite gestionar las dependencias de una aplicación en un entorno virtual, es decir, sin instalar las dependencias directamente en el sistema. También tiene un sistema de empaquetamiento, de tal forma que se puede subir el proyecto completo como un único paquete a plataformas como Pypi para que cualquier usuario pueda descargarlo [11].

#### **4.2.4.2. Visual Studio Code**

Visual Studio Code es un editor de código redefinido y optimizado para crear y depurar aplicaciones web y en la nube modernas. Se ha utilizado esta herramienta y no otras como Atom o PyCharm por su facilidad de uso, cantidad de extensiones e integración con el sistema operativo y las herramientas del equipo.

#### **4.2.4.3. Github Copilot**

Github Copilot es una herramienta que asiste en la programación mediante la generación de código dado un contexto. Para generar un código acorde utiliza inteligencia artificial entrenada con 2 billones de líneas de código del global de repositorios de Github, junto al código del propio proyecto, brindando una experiencia sin igual [26].

Se ha utilizado para optimizar el tiempo invertido en la programación y dar asistencia en la generación de llamadas a funciones, evitando así tener que revisar de manera continuada el código fuente y la documentación de *frameworks* como Django.

#### **4.2.4.4. Sourcery**

Sourcery es una inteligencia artificial que se ejecuta en segundo plano y sugiere mejoras en términos de refactorización del código en tiempo real: desde la simplificación de condicionales hasta la extracción de métodos [4]. Al igual que Github Copilot, se ha utilizado para optimizar el código, refactorizando y mejorando ciertas funciones y dando indicadores de posibles refactorizaciones.

#### **4.2.4.5. DeepSource**

DeepSource es una plataforma moderna de análisis estático que analiza continuamente el código en busca de vulnerabilidades de seguridad, problemas de rendimiento, riesgos de errores y anti patrones, y ayuda a los desarrolladores a descubrir y corregir automáticamente estos problemas en el código antes de enviarlo a producción [34]. Se ha utilizado para analizar posibles mejoras y optimizaciones del código.

#### **4.2.4.6. Django Debug Toolbar**

Es una librería para Django que permite analizar los tiempos de respuesta y la cantidad de peticiones que se hacen a la base de datos y qué consultas se ejecutan, si estas están duplicadas y da información sobre otros parámetros de ejecución en una aplicación basada en Django [31].

Se ha utilizado primordialmente para verificar el rendimiento de las peticiones y que no se realizaran consultas muy costosas, y en caso de darse verificar cuáles eran para optimizarlas.

#### **4.2.4.7. Black**

Black es una herramienta de linting para Python. Una herramienta de linting formatea el código de tal forma que cumpla unos ciertos estándares, en este caso PEP8 [41].

Black se ha utilizado de manera continua para reformatear el código y así mantener unas ciertas garantías de calidad del código Python según las normas del PEP8. Es importante destacar que se ha alterado la longitud máxima de las líneas de 79 caracteres a 119, según se recomienda en la documentación de Django.

#### **4.2.4.8. Flake8**

Flake8 es otra herramienta de linting, pero esta en vez de formatear el código, da avisos sobre si el código cumple o no PEP8. Es más estricta que Black por defecto [44].

Se ha utilizado Flake8 para revisar el contenido que Black no trataba y poder tener una calidad de código lo más afín a los estándares de la comunidad Python.

#### **4.2.4.9. Figma**

Figma es una herramienta colaborativa para prototipado digital. Permite diseñar de forma rápida la interfaz deseada en el proyecto final [10].

Se ha empleado para el prototipo inicial de la aplicación, lo que ha permitido tener una referencia clara del diseño final de esta.

### **4.2.5. Entorno de pruebas**

Para que el proyecto funcione con unas garantías se ha sometido a una serie de pruebas.

#### **4.2.5.1. Postman**

Postman es una plataforma API para construir y usar APIs. Postman simplifica cada paso del ciclo de vida de la API y agiliza la colaboración crear mejores API, más rápido [37]. Se ha utilizado para realizar baterías de pruebas contra el API REST del agente en cada modificación que se realizaba a los métodos que afectarían a *endpoints*.

En vez de otras herramientas como HTTPie se ha empleado Postman por su facilidad de uso, rapidez a la hora de crear nuevas peticiones, modificar las existentes y lanzar múltiples peticiones a la vez.

#### **4.2.5.2. HTTP Bin**

HTTP Bin es un servicio creado por los desarrolladores de Postman en Flask que permite hacer peticiones y ver la respuesta que devolvería un servicio similar [38]. Se utilizó temporalmente mientras el servicio API REST de la aplicación web no estaba operativo para verificar que las peticiones POST del agente tenían toda la información y que el formato de este era correcto.

### **4.3. Entorno de despliegue**

#### **4.3.1. Despliegue**

La aplicación final requiere de un despliegue para poder ser operativa en un entorno de producción, en la que el uso de la aplicación sea real, con su correspondiente cantidad de peticiones.

##### **4.3.1.1. Docker**

Es un marco de *software* para construir, ejecutar y administrar contenedores en servidores y la nube. En lugar de usar servidores únicos, se basa en matrices de servidores interdependientes y redundantes. Estos servidores basados en software se denominan contenedores y son una combinación híbrida del sistema operativo Linux en el que se ejecutan, más un entorno de tiempo de ejecución hiper localizado (la información almacenada en el contenedor) [16].

Se ha utilizado Docker y Docker Compose para gestionar el despliegue de múltiples servicios en un solo paso. Esto facilita poder realizar el despliegue de la aplicación web en cualquier sistema operativo, pues realmente se está virtualizando un Linux con un espacio en disco mínimo (alrededor de 200 MB).

##### **4.3.1.2. Nginx**

NGINX es un software de código abierto para servicios web, proxy inverso, almacenamiento en caché, equilibrio de carga, transmisión de medios y más. Comenzó como un servidor web diseñado para el máximo rendimiento y estabilidad. Además de sus capacidades de servidor HTTP, NGINX también puede funcionar como un servidor

proxy para correo electrónico (IMAP, POP3 y SMTP) y un proxy inverso y equilibrador de carga para servidores HTTP, TCP y UDP.

Se ha utilizado en la aplicación web para gestionar los ficheros estáticos en una URL dada y hacer de Proxy entre el cliente final y el servidor Uvicorn de Python.

En lugar de Apache se ha preferido Nginx por la cantidad de información existente de cómo vincular Nginx con Gunicorn a través de Docker, y porque la documentación oficial tanto de Gunicorn como de Uvicorn así lo recomiendan.

#### **4.3.1.3. Gunicorn - Uvicorn**

Uvicorn es una implementación de servidor web ASGI para Python. Hasta hace poco, Python carecía de una interfaz mínima de servidor/aplicación de bajo nivel para marcos asíncronos. La especificación ASGI llena este vacío y permite comenzar a crear un conjunto común de herramientas utilizables en todos los framework asíncronos como pueda ser FastAPI [48].

Se ha utilizado tanto para el agente como para la aplicación web y se encarga de proveer un servidor web más óptimo que el propio de Django o de FastAPI. Gunicorn actúa de servidor, y utiliza Uvicorn como worker para las peticiones.

No se plantearon más alternativas pues Uvicorn como servidor es uno de los más rápidos para el entorno de Python, muy por encima de otros servidores ASGI o WSGI.

#### **4.3.1.4. PostgreSQL**

PostgreSQL es uno de los sistemas de bases de datos más potentes y conocidos debido a su confiabilidad, robustez de funciones y rendimiento. Es un sistema de base de datos relacional de objetos de código abierto.

Se ha utilizado PostgreSQL en la aplicación web para almacenar los datos de los usuarios, los agentes, las métricas, las alertas y las configuraciones.

Se valoró utilizar MongoDB para guardar los datos en formato JSON por su variabilidad, pero finalmente se decidió utilizar PostgreSQL porque la mayoría de las tablas son relacionales y en este sentido, PostgreSQL tiene pocos competidores. En relación con los datos en formato JSON, PostgreSQL permite guardarlos gracias a un



campo de tipo JSONField. Otra de las razones para utilizar PostgreSQL es la cantidad de documentación existente para integrar Django con PostgreSQL en Docker.

#### **4.3.1.5. Redis**

Redis es un almacén de datos en memoria RAM de código abierto utilizado como base de datos, caché, motor de streaming y broker de mensajes.

En este proyecto se ha utilizado Redis junto a Redis Queue en la aplicación web para gestionar las tareas de segundo plano ejecutadas en Django. RQ Scheduler ha facilitado además de lanzar las tareas, el poder planificarlas y asignarle a cada tarea una fecha de ejecución, o un intervalo de tiempo tras el cual volver a ejecutarse la tarea.

Se planteó utilizar Celery como alternativa para ejecutar tareas en segundo plano pero por su complejidad y funcionalidad extensa se ha preferido utilizar Redis. Otra de las razones para utilizar Redis es la cantidad de documentación existente para integrar Django con Redis en Docker.

#### **4.3.1.6. SystemD**

Systemd es un paquete de software que proporciona una variedad de componentes del sistema para los sistemas operativos Linux. Su objetivo principal es unificar la configuración y el comportamiento del servicio en todas las distribuciones de Linux. Su componente principal es el "administrador de sistemas y servicios" que permite gestionar los procesos de los usuarios.

Se ha utilizado un servicio de SystemD como daemon para permitir la ejecución del agente en sistemas Linux de manera sencilla, aunque antes ha sido necesario instalar sus dependencias.

#### **4.3.1.7. Windows Task Scheduler**

Windows Task Scheduler o "Programador de Tareas" como aparece en la versión española, es un gestor de tareas que permite planificar y gestionar la ejecución de scripts y programas [29]. Se ha utilizado para desplegar y ejecutar el *script* del agente en el arranque, aunque antes ha sido necesario instalar sus dependencias.



## 5. Aspectos relevantes

### 5.1. Decisiones sobre la arquitectura

#### 5.1.1. Primera propuesta

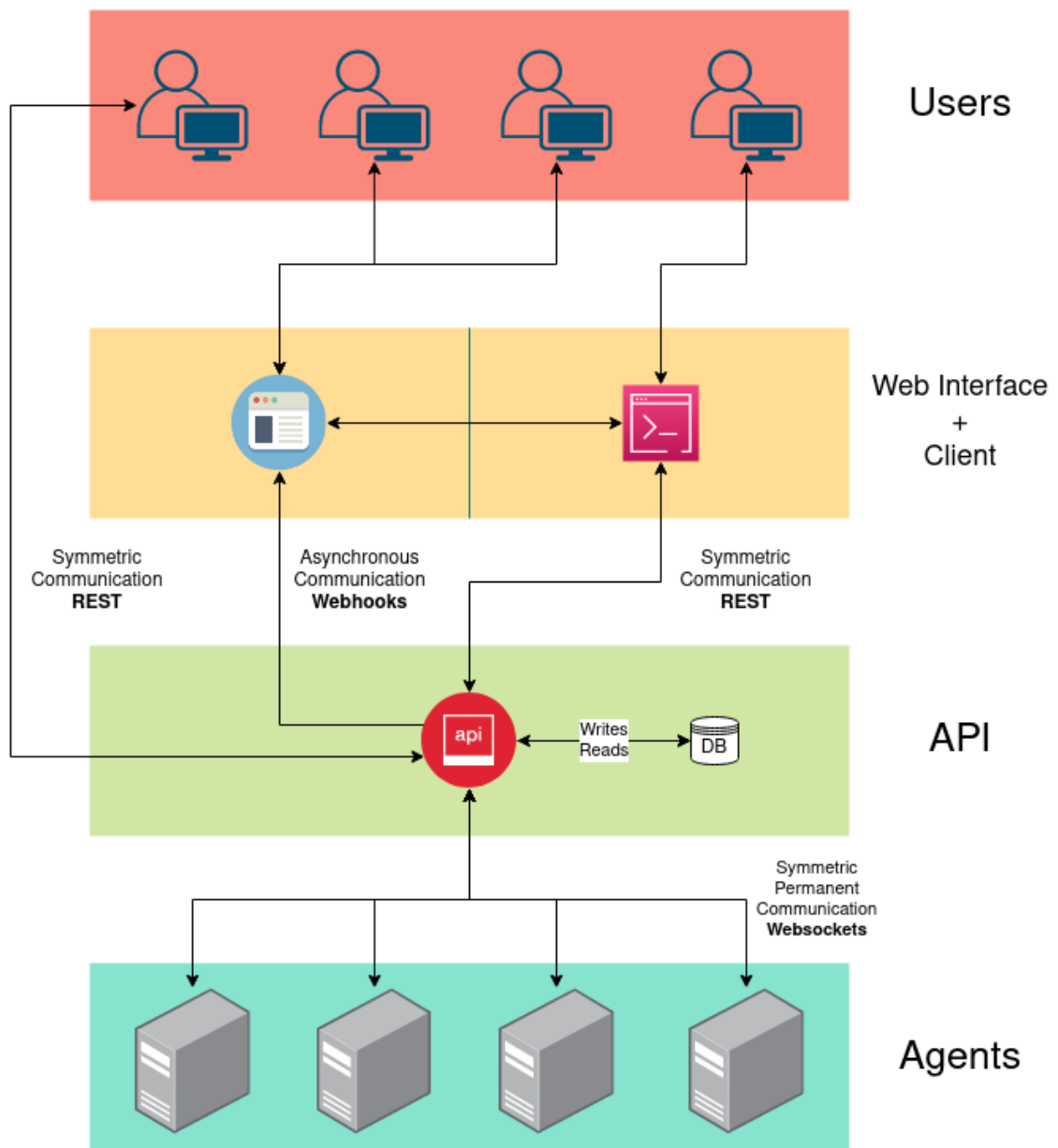


Figura 5. Primer prototipo de arquitectura

Esta es la parte más importante del presente proyecto, pues es la base de la comunicación mediante peticiones y crear un sistema robusto que soporte peticiones continuas de múltiples equipos a un servidor.

En la Figura 5 se muestra un diagrama con la arquitectura propuesta inicialmente y que recoge cómo las peticiones se efectúan finalmente a una API REST y esta serializa los datos para finalmente almacenarlos en la base de datos. De esta manera se consigue una aplicación RESTful que permite realizar peticiones asíncronas desde una página web, desarrollar un cliente Python intermediario para facilitar estas operaciones, crear aplicaciones propias para adoptar una ruta de trabajo más flexible y adaptable al entorno específico de uso particular o de una empresa. De esta manera también se garantiza más flexibilidad para programar los agentes instalados en los equipos.

El agente es una aplicación pequeña que se instala en los equipos y recoge información del equipo continuamente. Este se comunica con el API REST mediante *WebSockets*, el cual produce una mejora en la eficiencia respecto a HTTP debido a que está pensado para comunicaciones y mantiene el socket abierto para facilitar esta comunicación, mientras que HTTP abre y cierra los sockets por cada petición, lo que produce un consumo de rendimiento notable en el servidor por el hecho de monitorizar múltiples equipos.

El API REST está conectado directamente con la base de datos y recibe peticiones HTTP de tipo CRUD de tal manera que la aplicación es RESTful. Permite desligar el API REST de la aplicación principal y aumentar la escalabilidad pues se pueden replicar mediante contenedores y así balancear el tráfico. El gestionar la información a través del API REST permite a los desarrolladores crear aplicaciones externas a través del esquema OpenAPI [28].

El cliente es una aplicación Python que implementa peticiones a los distintos endpoints del API REST mediante su esquema OpenAPI. Permite utilizar y acceder a los elementos de la base de datos en forma de objetos Python, lo que facilita su uso y permite su modularización para ser utilizado en otras aplicaciones Python: desde un bot hasta otra aplicación web o instalable.

La página web hace peticiones al API REST a través del cliente Python para gestionar la información de la base de datos en forma de objetos, y utilizarlos de manera sencilla con los métodos estándar de Python. Los usuarios se conectarán a la interfaz web y podrán visualizar los datos y gráficas de los equipos que deseen monitorizar. Además, para la ejecución de comandos se tendrá un *WebSocket* que permitirá mantener la conexión y ejecutar comandos de manera continuada como si de una terminal se tratase.

El proceso de la comunicación podría verse desde dos puntos: la comunicación de los equipos con el servidor para el envío de métricas, y el uso de la página web y cliente para la comunicación con la base de datos. En el primer punto se tiene que el agente manda a través del canal *WebSocket* que ha establecido métricas como si de un chat se tratase, facilitando el paso a un sistema de tiempo real. En el segundo punto se tiene al usuario que puede utilizar el cliente por consola de comandos para obtener la información de sus equipos de manera programable, o a través de una interfaz web con posibles llamadas asíncronas que facilitarían también la presentación de datos en tiempo real.

### 5.1.2. Segunda propuesta

En la Figura 6, se muestra el segundo prototipo de arquitectura del proyecto.

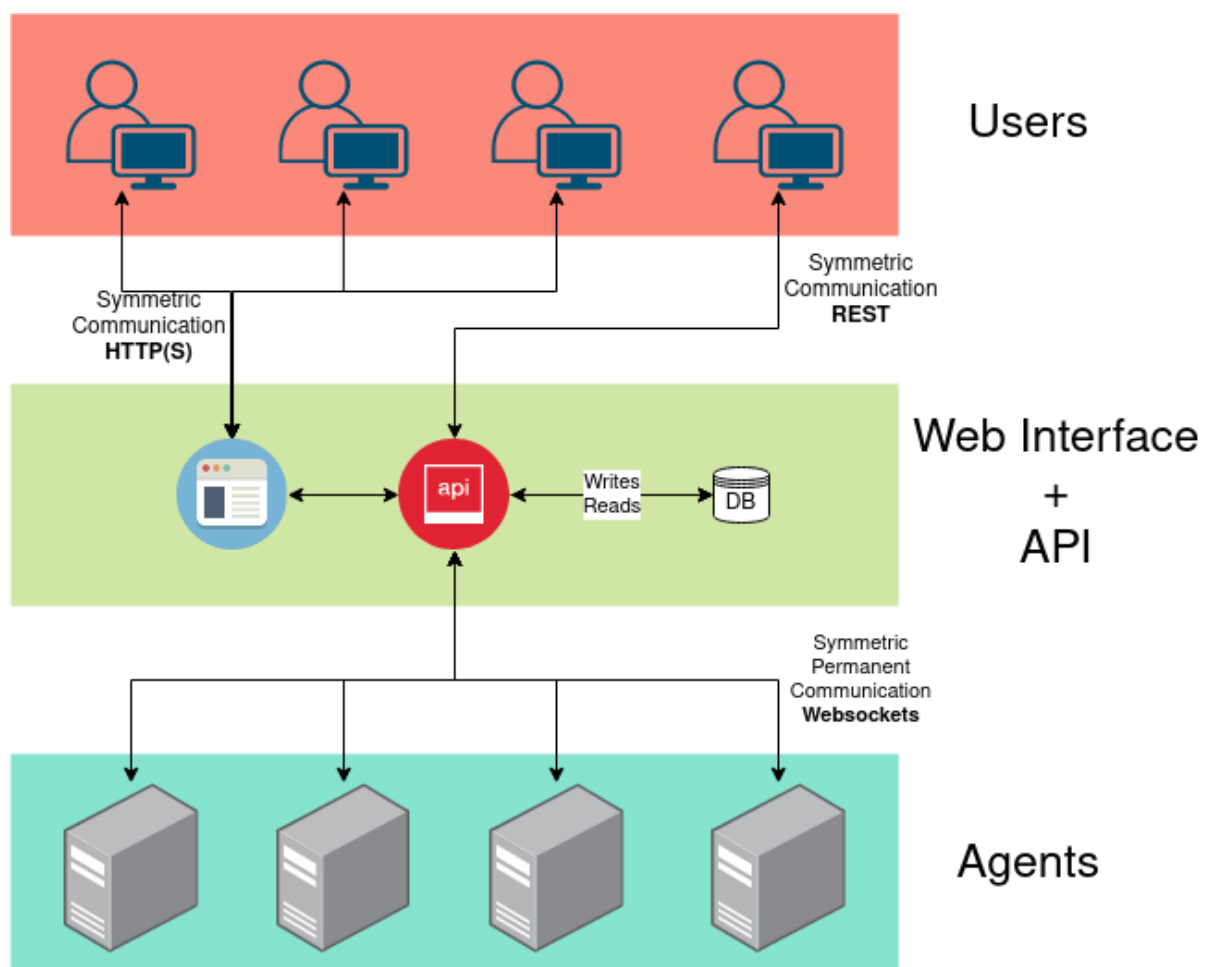


Figura 6. Segundo prototipo de arquitectura

Mientras que el desarrollo del cliente Python ofrece sin lugar a duda un beneficio importante en la gestión de las peticiones, abstrayendo estas y la gestión de las URLs y de las URIs, resultaría difícil de mantener en caso de cambios continuos en el API REST como pueda ser un continuo desarrollo de la aplicación para adaptarla a las necesidades de una empresa, y porque su proyección de uso no tiene mucha viabilidad para este proyecto en concreto.

La aplicación web por otro lado se desplegará unida al API REST en la misma aplicación, facilitando el rendimiento en un entorno pequeño por el acceso directo a las peticiones en el equipo (evitando retardos en la red y las tasas de deriva en la sincronización de paquetes en la red). Como desventaja, en casos en los que la aplicación web no sea necesaria, se incluiría de manera forzada en el despliegue por su uso en la gestión de los modelos de la base de datos.

### **5.1.3. Arquitectura final**

En la Figura 7, se muestra el prototipo de arquitectura del proyecto.

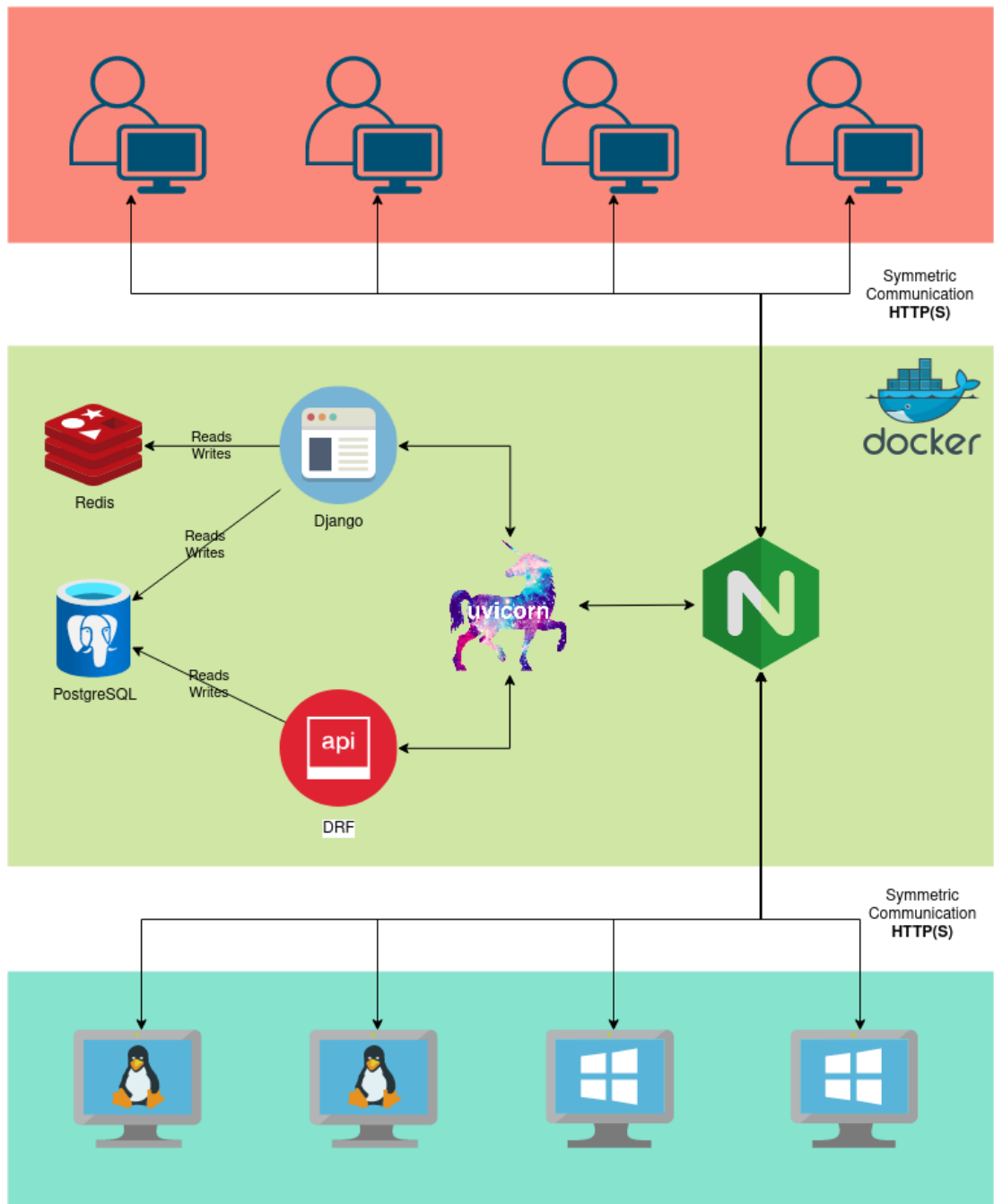


Figura 7. Prototipo final de arquitectura

En la segunda propuesta toda la gestión de datos se realizaba únicamente a través del API REST. En este prototipo se puede acceder también a la información desde la página web a la base de datos y gestionar todos los recursos de los que la aplicación dispone a través de formularios, listados de la información de la tabla y la información en detalle de cada elemento (fila) de la base de datos. Esto da flexibilidad al usuario de poder utilizar la aplicación de la manera que desee.

Aunque exista una interfaz web, el usuario al final es una persona técnica con habilidades para gestionar los recursos de la aplicación a través de la API REST con peticiones, o aplicaciones y clientes personalizados para su uso personal o de empresa.

Una desventaja quizá de este modelo es que utiliza el mismo servidor web para el API REST y la página web, mientras que anteriormente las llamadas a la base de datos se realizaban a través del API REST, forzando así un aplicación web *RESTful*. Esto se contrarresta pues se gestiona a través de contenedores en Docker, lo que permite replicar estos contenedores y gestionarlo por clústeres, ofreciendo replicabilidad y por ende extensibilidad y escalabilidad. En el anterior prototipo también se podía gestionar con Docker y ofrecer unas características similares, pero lo que diferencia este prototipo del anterior es su uniformidad en el despliegue, que para casos de uso particulares o de pequeñas y medianas empresas permite un uso más versátil y dinámico.

## **5.2. Gestión del Proyecto**

### **5.2.1. Fases del diseño y conceptualización**

Para la fase de diseño y conceptualización de la aplicación se han tenido como referencia partes de ciertas aplicaciones existentes de código abierto como pueda ser Prometheus, concretamente la parte de lógica de negocio [39].

#### **5.2.1.1. Escenarios de uso y funcionalidad**

Un aspecto relevante del presente proyecto es la aplicación de metodologías ágiles y la definición de la funcionalidad en cada iteración, de manera que esta incrementa gradualmente en cada sprint.

En el primer sprint se valoraron ciertas funcionalidades base que eran requisitos obligatorios para el funcionamiento básico del sistema monitorización. Entre ellos se encuentran:

- Los datos que el agente envía deben ser recibidos por la página web mediante su API REST, y almacenarlos en una base de datos.
- Una vez almacenados, se puede proceder a su gestión a través del API REST o de la interfaz web, que además presentará y listará estos datos mediante gráficas y listas de elementos paginados, accesibles mediante una barra de búsqueda.



- Dada la complejidad del sistema de configuración del agente, se utiliza un fichero con formato JSON para estandarizar la gestión de la configuración.
- La interfaz web permitirá a su vez ejecutar comandos en los equipos a monitorizar, pero de manera segura autenticándose contra el agente mediante el token de usuario. Este token se almacena en el fichero de configuración del agente.

En el segundo y tercer sprint se refinaron aspectos de la funcionalidad y del diseño de la arquitectura y de la interfaz.

En el cuarto sprint se plantearon nuevos aspectos de la funcionalidad de la aplicación como son:

- Que la aplicación sea configurable por el usuario en aspectos como los intervalos de tiempo de envío de métricas y la configuración de ambos servidores, el del agente y de la página web, tanto para credenciales como para aspectos internos del servicio que provee la gestión de las peticiones HTTP.
- Se debe de poder recoger datos de los equipos y enviarlas a un servidor de manera segura. Esto se consigue gracias al sistema de autenticación por tokens, por el que cada usuario posee un token como alternativa al inicio de sesión por usuario-contraseña, y por el que cada equipo posee un token para identificarse de manera única contra el sistema.
- Se debe de poder generar la configuración del agente a través de un formulario en la página web, haciendo de este proceso algo sencillo. La configuración del equipo se puede ver en la página web y permite copiar al portapapeles y exportar a formato JSON para conveniencia del usuario.
- Un sistema de alertas, por el que, dados unos límites de CPU y RAM configurables por el usuario, el sistema dadas ciertas circunstancias mandará alertas al usuario. El usuario puede configurar el sistema de alertas para que le lleguen mediante correos electrónicos a múltiples correos, o para que le lleguen a distintos servicios como pueda ser Discord, Telegram o Slack mediante *webhooks*.
- Por último, el sistema no deja de funcionar gracias a un servidor. Para evitar usos malintencionados de la herramienta, se hace logging de las peticiones, facilitando así a servicios como Fail2Ban el bloqueo y reporte de usuarios maliciosos.

En el quinto y sexto sprint se revisaron aspectos de la implementación de esta funcionalidad y se mejoraron aspectos de esta para mejorar la experiencia del usuario.

Se encuentra información más detallada de todos los escenarios de uso en el anexo I, junto con las tablas de requisitos obtenidas con la metodología Durán y Bernárdez y los diagramas de clases.

### **5.2.1.2. Diseño de la interfaz**

El diseño de la interfaz se ha realizado en dos fases, el prototipo inicial y el final.

El prototipo digital se realizó en Figma en vez de Adobe XD por probar otra herramienta alternativa a la usada durante el grado, y así disponer de más recursos con los que saber trabajar.

Para ambas interfaces se han empleado métodos y patrones propios del desarrollo UI/UX como puedan ser los principios de diseño CRAP [12], el patrón F [51], los principios de diseño SOLID, la repetición de elementos, la técnica de la proximidad, el espacio negativo, la ley de Fitts [6], la familiaridad de los elementos y los principios de la arquitectura de información de Dan Brown [27]. Estos han sido muy importantes para estructurar y organizar la disposición de los elementos en la interfaz y favorecer una experiencia al usuario óptima para facilitar que este encuentre la información de una manera accesible.

Esto en conjunto con el uso de colores, el diseño responsive, la elección de la tipografía favorece la accesibilidad de la página a cualquier público, y la reducción de la carga cognitiva del usuario tanto por la información que se muestra en pantalla, tanto por cómo se dispone esta información. Estos puntos cada uno ha tenido su correspondiente decisión en la toma de valores: para el conjunto de colores mientras que se ha optado por un diseño minimalista con colores blancos y el azul como realce, se optó en un primer momento por un color naranja o uno verde; el diseño aunque tenía que ser responsive sin optar a dudas, ha sido importante la decisión de la posición de elementos como el menú contextual y la posición de los elementos para poder adaptar la interfaz a cualquier dispositivo; en cuanto a la tipografía se ha tenido claro desde un primer momento que la tipografía debía ser tipo Sans y no Sans Serif pues esta última está pensada y optimizada para facilitar la lectura en textos largos.

Se encuentra información más detallada de las elecciones en los colores, tipografía y estructura de la interfaz en el apartado de “Fases de Diseño” en el anexo I. A su vez, se pueden encontrar capturas de pantalla tanto del prototipo inicial como del prototipo final.

## **5.2.2. Fases de la implementación**

Para la implementación tanto del agente como de la aplicación web se han tenido en cuenta principios de diseño como SOLID [52], divide y vencerás, incrementar la cohesión, la abstracción, la reusabilidad y la flexibilidad, reducir el acoplamiento y planificar la escalabilidad del producto final para facilitar el mantenimiento, la legibilidad, la portabilidad, el testeo y el desarrollo general del mismo. Se complementa la información de este apartado en el anexo II.

### **5.2.2.1. Agente**

La primera aplicación en desarrollarse fue el agente, el cual está desarrollado en Python y basado en FastAPI. Para su implementación se hizo un uso extensivo de la librería PSUtil que permite obtener información del equipo en el que se ejecutan sus funciones.

#### **5.2.2.1.1. Clases para la información**

Una vez recogida la información, se agrupó dentro de clases: una clase estática que mantiene información que no varía en el tiempo, y una clase dinámica que mantiene la información que sí varía en el tiempo. Plantear esta estructura requirió diseñar el sistema de métricas varias veces hasta llegar a la conclusión de que la manera más óptima de recoger las métricas de forma continua sin que existan muchas penalizaciones de rendimiento y temporales, era que las funciones y valores que no iban a variar en el tiempo tales como datos propios de hardware y del sistema operativo, se almacenasen en una clase invariable, y que los datos que podrían modificarse se almacenasen en otra clase distinta. De esta manera se agrupa el contenido de la información y se gestiona más fácilmente desde otros ficheros al importar estas clases desde sus respectivos módulos.

Mediante un método propio de Python, se convierte una clase en un diccionario (lista clave-valor), que permite convertirse a una cadena de texto en formato JSON mediante la librería JSON.

Una vez se tiene la información, esta se manda mediante un evento temporal que se inicia al arrancar el servidor Uvicorn, y que ejecuta un cierto código pasado un intervalo de tiempo determinado por el usuario en el fichero de configuración. Este código hace peticiones al servidor principal con esta información en formato JSON y la autenticación contra el servidor en las cabeceras de la petición.

A su vez, si el estado de CPU y RAM del equipo sobrepasa ciertos parámetros establecidos por el usuario, se manda una alerta al servidor para evitar que este tenga que analizar el contenido enviado por cada métrica recibida de los múltiples equipos que pueda estar monitoreando.

#### **5.2.2.1.2. Ejecución de comandos**

Una de las partes más complejas del agente ha sido desarrollar la ejecución de comandos. Esto es debido a la gestión de subprocesos y de comandos de diversa índole: comandos que devuelven mucha información, comandos interactivos que no retornan información y demás.

El problema más grande encontrado ha sido averiguar que en Windows hay que lanzar el comando (en formato cadena de texto) en una terminal intermedia para poder recoger el resultado de este, mientras que en Linux el mismo comando debía descomponerse en una lista (array) y ejecutarse directamente sin una terminal intermedia. En un primer momento al pensar que en ambos sistemas se hacía falta una terminal intermedia, y mientras que en Windows funcionaba, en Linux se creaba un proceso nuevo en el que se ejecutaba el comando y no devolvía ningún dato por los canales de salida y de error estándar porque la terminal estaba desligada del proceso principal. Eso, o en el caso de no utilizar una terminal intermediaria, en Linux se ejecutaban correctamente pero en Windows no se mostraba nada, precisamente porque sí necesita una terminal intermedia.

Al ejecutar comandos interactivos en el agente a través de una petición a su API REST, la petición se queda bloqueada hasta que recibe una respuesta, pero al ser interactivo y tener una ejecución indefinida la petición sigue esperando una respuesta. La forma de solucionar esto ha sido especificar un tiempo de *timeout* para que, al

ejecutar el comando en otro hilo, si este no ha terminado en el tiempo especificado el hilo se mate con una señal SIGKILL. Si no se llegara a especificar el *timeout* el comando quedaría ejecutándose infinitamente en el equipo remoto ocupando recursos, y el servidor tendría la petición bloqueada hasta que se recibiese una respuesta, consumiendo también recursos.

En la Figura 8 se muestra la interfaz web de ejecución de comandos.

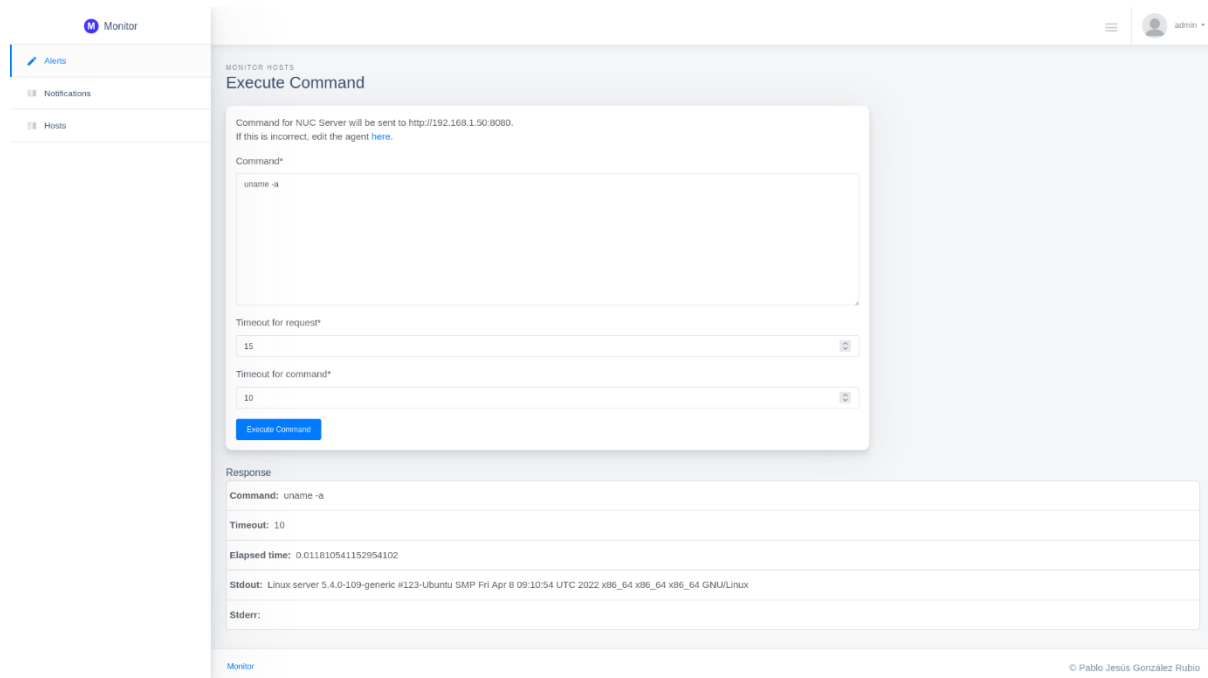
The image shows a web application interface for monitoring hosts. On the left is a sidebar with a 'Monitor' header and three menu items: 'Alerts', 'Notifications', and 'Hosts'. The main area is titled 'MONITOR HOSTS' and 'Execute Command'. It contains a form with a text area for 'Command\*' (containing 'uname -a'), two dropdown menus for 'Timeout for request\*' (set to 15) and 'Timeout for command\*' (set to 10), and a blue 'Execute Command' button. Below the form is a 'Response' section with fields for 'Command: uname -a', 'Timeout: 10', 'Elapsed time: 0.011810541152954102', 'Stdout: Linux server 5.4.0-109-generic #123-Ubuntu SMP Fri Apr 8 09:10:54 UTC 2022 x86\_64 x86\_64 x86\_64 GNU/Linux', and 'Stderr:'. The footer includes the 'Monitor' logo and a copyright notice for Pablo Jesús González Rubio.

Figura 8. Formulario para la ejecución de comandos

### 5.2.2.1.3. Gestión de la configuración

Otro aspecto significativo del desarrollo del agente ha sido basar su funcionamiento en un fichero de configuración JSON. Esto aporta mucha flexibilidad al usuario a la hora de configurar la aplicación y su comportamiento. Este fichero de configuración puede ser rellenado por el usuario a mano teniendo como referencia el fichero de configuración ejemplo, aunque se recomienda su generación automática a través de la aplicación web.

Para la configuración del agente se deben tener en cuenta aspectos como la configuración del servidor Uvicorn, las credenciales de autenticación con la aplicación web y la configuración propia del sistema monitorizador, como puedan ser límites máximos de CPU y RAM.

El agente cada vez que se inicia lee el fichero de configuración, por lo que, si se modifica este, el agente se comporta de forma acorde al cambio en el fichero.

En un primer momento se planteó utilizar un fichero Python en el que especificar las variables, pero lo hacía poco versátil pues en los servicios API REST es frecuente enviar JSON o HTML. Es por ello por lo que se decidió utilizar un fichero JSON del que leer la información [Figura 9].

```
1 {
2   "alerts": {
3     "url": "http://localhost:8000/api/alerts/"
4   },
5   "auth": {
6     "agent_token": "fb79210dd15ea00aeeb4bb21f81b27421a4e11a7",
7     "name": "Laptop",
8     "user_token": "19fc6e57b8fb0e5a1a7b55976952ab02865aa771"
9   },
10  "endpoints": {
11    "agent_endpoint": "http://localhost:8000/api/agents/",
12    "metric_endpoint": "http://localhost:8000/api/metrics/"
13  },
14  "logging": {
15    "filename": "monitor.log",
16    "level": "info"
17  },
18  "metrics": {
19    "enable_logfile": false,
20    "get_endpoint": false,
21    "log_filename": "metrics.json",
22    "post_interval": 60
23  },
24  "thresholds": {
25    "cpu_percent": 50,
26    "ram_percent": 30
27  },
28  "uvicorn": {
29    "backlog": 2048,
30    "debug": false,
31    "host": "0.0.0.0",
32    "log_level": "trace",
33    "port": 8080,
34    "reload": true,
35    "timeout_keep_alive": 5,
36    "workers": 4
37  }
38 }
```

*Figura 9. Fichero de configuración del agente*

Este fichero JSON al no tener especificación de tipos: es decir, si la información es una cadena de texto, un número o un booleano, no se podía serializar de manera sencilla.

La alternativa fue generar de forma automática mediante un formulario en la página web este fichero de configuración para evitar que el usuario pueda cometer errores [Figura 10].

The screenshot shows a web interface for configuring a monitor agent. On the left is a sidebar with a 'Monitor' header and three menu items: 'Alerts' (active), 'Notifications', and 'Hosts'. The main content area is titled 'MONITOR Edit'. It contains a form with the following fields:

- 'Filename of the log file:' with a text input containing 'monitor.log'.
- 'Logging level:' with a dropdown menu set to 'Informative'.
- 'Enable metrics log file:' with an unchecked checkbox.
- 'Enable metrics endpoint:' with an unchecked checkbox.
- 'Filename of the metrics log file:' with a text input containing 'metrics.json'.
- 'Interval in seconds to send metrics:' with a numeric input set to '300'.
- 'Time interval in minutes for Warning status in which alerts are accumulated, or metrics aren't received:' with a numeric input set to '10'.
- 'Time interval in minutes for Bad status in which alerts are accumulated, or metrics aren't received:' with a numeric input set to '60'.
- 'Percentage of CPU usage to trigger alert:' with a numeric input set to '70'.
- 'Percentage of RAM usage to trigger alert:' with a numeric input set to '50'.

*Figura 10. Formulario para configurar el agente*

También se valoró crear un endpoint que recibiera el fichero de configuración mediante una petición POST, y se llegó a desarrollar incluso. El problema que surgió con esta opción es que el servidor se reinicia si detecta un cambio en algún fichero Python, pero no ficheros externos, por lo que el usuario tendría que reiniciar de manera manual el servidor cada vez que mandara la petición desde la página web. Hacer esto, y copiar el fichero de configuración y reiniciar el servidor a mano es prácticamente lo mismo, y el administrador del equipo tendría que hacer el mismo esfuerzo, por lo que no se desarrolló más esta opción.

### 5.2.2.2. Aplicación Web

En la aplicación web aspectos relevantes se podrían considerar los siguientes.

#### 5.2.2.2.1. Arquitectura MVT

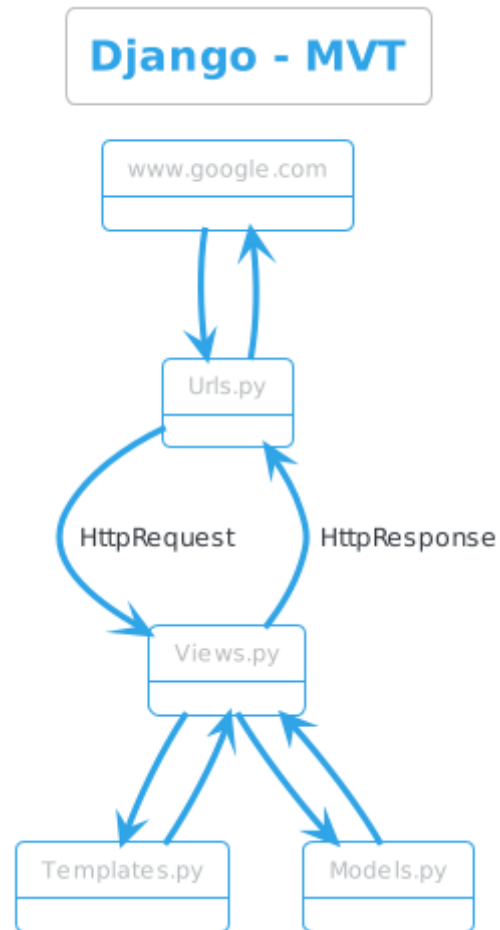


Figura 11. Arquitectura Modelo-Vista -Plantilla



Django sigue el modelo Modelo-Vista-Plantilla [Figura 11], muy similar al Modelo-Vista-Controlador estudiado en el Grado.

El paquete Templates está compuesto por todas las plantillas HTML que Django requiere para poder renderizar la información que la vista le pase a través del contexto web, y una vez renderizada, esta plantilla se muestra al usuario con los datos que le corresponden según el objeto al que ha accedido y los permisos que el usuario tiene sobre el objeto.

El paquete Models dentro del modelo MVT corresponde al paquete Modelo de la arquitectura MVC (Modelo-Vista-Controlador). En este paquete se declaran las clases que más tarde al hacer la migración se convertirán en tablas de la base de datos.

El paquete Views, o Vistas, dentro del modelo MVT de Django sería de forma similar al controlador del modelo MVC. Este permite gestionar los datos mediante el sistema ORM de Django, haciendo consultas a la base de datos y accediendo únicamente a las clases declaradas en el paquete Modelos.

En el caso particular de Django estas vistas pueden ser funciones, o pueden ser clases. En el proyecto se utilizan las vistas basadas en clases por los métodos integrados de los que disponen, ayudando a disminuir la base de código y refactorizar enormemente el código. En caso de haberse necesitado modificar algún comportamiento heredado de estas clases, se sobrescriben los métodos de las clases teniendo como referencia el código fuente de Django.

En el caso del API REST no se tienen Vistas si no Viewsets, que permiten realizar todas las acciones CRUD sobre ese modelo de una forma sencilla. Si no se utilizaran Viewsets, habría que definir manualmente cada método para realizar la creación, obtención, modificación y eliminación de los datos.

El paquete Forms es un paquete único de la página web (Django) que contiene clases que permiten crear formularios dinámicos en base a los datos que se le especifiquen.

El paquete Urls especifica las vistas que se deberán ejecutar al acceder a una determinada URL, y el resultado devuelto es lo que el usuario ve. Es lo que se denomina el enrutador de peticiones. Para verlo desde una perspectiva global, cuando un usuario o un equipo hace una petición al servidor, primero llega a Nginx, Nginx se la transmite a

Uvicorn y Uvicorn actúa y ejecuta las funciones según la URI que ha recibido, y es ahí donde entra el enrutador de peticiones que es el fichero `Urls`.

Cada uno de los elementos del modelo MVT se ha utilizado extensivamente para poder recrear todos los elementos necesarios en una aplicación web, y en cualquier aplicación orientada a objetos.

#### 5.2.2.2. Sistema de autenticación

Una vez se tenían los datos lo siguiente fue implementar el sistema de autenticación para tener un sistema de comunicación más seguro, y para permitir que se mostraran al usuario únicamente los agentes que le corresponden, no también los de los demás usuarios. Para ello se implementó la clase `CustomUser` [Figura 12] que hereda de `AbstractUser` [Figura 13], para permitir tener los mismo métodos y atributos que el usuario estándar de Django. De esta forma se permite extender su funcionalidad.

```
class CustomUser(AbstractUser):
    email = models.EmailField(("email address"), unique=True)

    def __str__(self) -> str:
        return self.username

    class Meta:
        ordering = ["username"]
        verbose_name_plural = "Users"
```

Figura 12. Modelo CustomUser

```
class AbstractUser(AbstractBaseUser, PermissionsMixin):
    username_validator: UnicodeUsernameValidator = ...

    username = models.CharField(max_length=150)
    first_name = models.CharField(max_length=30, blank=True)
    last_name = models.CharField(max_length=150, blank=True)
    email = models.EmailField(blank=True)
    is_staff = models.BooleanField()
    is_active = models.BooleanField()
    date_joined = models.DateTimeField()

    EMAIL_FIELD: str = ...
    USERNAME_FIELD: str = ...
    def get_full_name(self) -> str: ...
    def get_short_name(self) -> str: ...
    def email_user(self, subject: str, message: str, from_email: str = ..., **kwargs: Any) -> None: ...
```

Figura 13. Modelo AbstractUser

Se gestionó también la creación de un *token* [Figura 14] en el momento que un usuario se registra, de forma que los equipos que manden peticiones al servidor utilicen el token del usuario para autenticarse contra el servidor [7].

```
@receiver(post_save, sender=settings.AUTH_USER_MODEL)
def create_auth_token(sender, instance=None, created=False, **kwargs):
    if created:
        Token.objects.create(user=instance)
```

Figura 14. Creación del token

Este *token* se asocia con un usuario y es lo que facilita que aplicaciones programables como es el agente puedan realizar peticiones con el API REST del servidor sin tener que especificar un usuario y una contraseña.

Se planteó utilizar *tokens* JWT pero se seguía teniendo el mismo problema: cómo se identifica de manera única un equipo en el sistema. Se llegó al siguiente diseño: para la gestión de la autenticación, el agente posee un *token* propio que identifica ese equipo en el sistema, y además necesita el *token* del usuario para poder autenticarse. De esta manera el agente manda dos *token* y es lo que permite asociar una métrica de un equipo determinado con el modelo registrado para ese usuario en concreto.

En la Figura 15 se puede apreciar cómo se realiza la comunicación del agente con el servidor a través del API REST: en el caso de que la autenticación sea correcta, el dato se almacena, en caso de no serlo se le devuelve un error al agente.

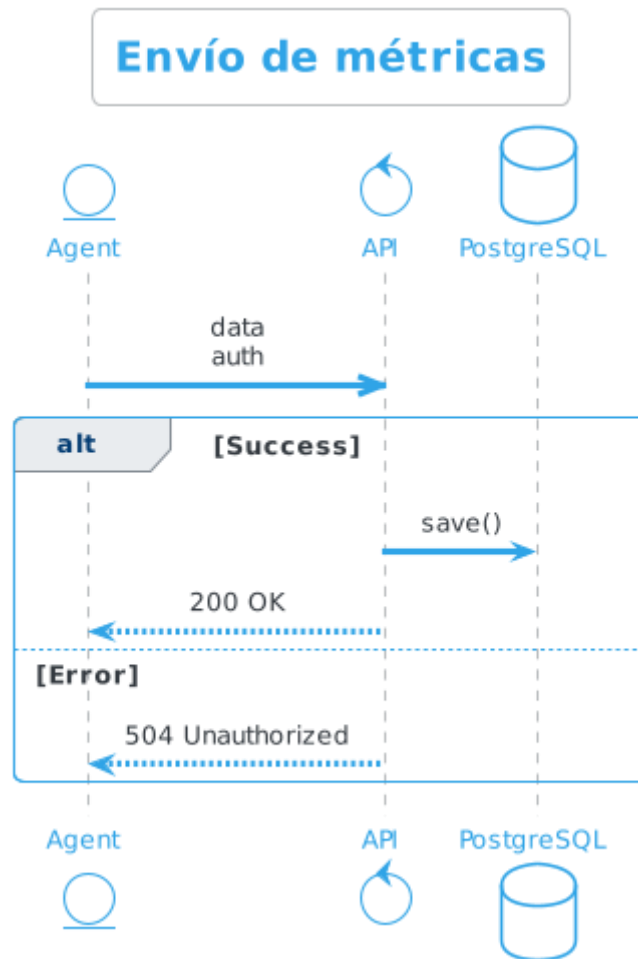


Figura 15. Diagrama de secuencia para el envío de métricas

### 5.2.2.2.3. Ejecución de tareas en segundo plano

La decisión de la elección del sistema de ejecución de tareas en segundo plano fue difícil por la variedad de sistemas que lo hacen de forma nativa, y los que lo podrían hacer mediante extensiones. Algunos de estos han sido el gestor de tareas Celery, el gestor de colas RabbitMQ, el gestor de colas Redis y finalmente Django Channels con los emisores, receptores y consumidores que provee, que no dejan de ser eventos que envían y reciben mensajes. Finalmente se utilizó Redis debido a su integración con Django mediante librerías como Django RQ.

En la figura Figura 16 se puede apreciar el funcionamiento de los workers de Redis, que ejecutan distintas tareas en segundo plano.

```

django-monitor-uvicorn | 16:53:09 Registering birth
django-monitor-uvicorn | Registering birth
django-monitor-uvicorn | default: web.tasks.remove_old() (774836be-655e-41a3-9561-e9cbc69d2cb3)
django-monitor-uvicorn | default: Job OK (774836be-655e-41a3-9561-e9cbc69d2cb3)
django-monitor-uvicorn | Result will never expire, clean up result key manually
django-monitor-uvicorn | default: web.tasks.check_status() (9ef75154-64f1-44e5-9772-4020101b8fb8)
django-monitor-uvicorn | No agents found
django-monitor-uvicorn | default: Job OK (9ef75154-64f1-44e5-9772-4020101b8fb8)
django-monitor-uvicorn | Result will never expire, clean up result key manually
django-monitor-uvicorn | Using selector: EpollSelector
django-monitor-uvicorn | Using selector: EpollSelector
django-monitor-uvicorn | Using selector: EpollSelector
django-monitor-uvicorn | Not Found: /favicon.ico
django-monitor-uvicorn | Using selector: EpollSelector
django-monitor-uvicorn | Not Found: /favicon.ico
django-monitor-uvicorn | Using selector: EpollSelector
django-monitor-uvicorn | Using selector: EpollSelector
django-monitor-uvicorn | Not Found: /favicon.ico
django-monitor-uvicorn | Not Found: /favicon.ico
django-monitor-uvicorn | default: web.tasks.check_status() (7aa8368e-6c75-4b1c-9ed6-5258d45ebd08)
django-monitor-uvicorn | No agents found
django-monitor-uvicorn | default: Job OK (7aa8368e-6c75-4b1c-9ed6-5258d45ebd08)
django-monitor-uvicorn | Result will never expire, clean up result key manually
django-monitor-uvicorn | default: web.tasks.remove_old() (b1dd15a2-a8f9-4b2c-a178-16d0d9a4a543)
django-monitor-uvicorn | default: Job OK (b1dd15a2-a8f9-4b2c-a178-16d0d9a4a543)
django-monitor-uvicorn | Result will never expire, clean up result key manually
django-monitor-uvicorn | Using selector: EpollSelector
django-monitor-uvicorn | Using selector: EpollSelector
django-monitor-uvicorn | Using selector: EpollSelector
django-monitor-uvicorn | Using selector: EpollSelector
django-monitor-uvicorn | Using selector: EpollSelector

```

Figura 16. Workers de Redis

Una vez seleccionado el gestor de tareas en segundo plano era importante que este planificara de forma periódica las tareas una única vez. Esto es así porque de planificarse múltiples veces, las funciones a ejecutar se multiplicarían por cada lectura del método que prepara la planificación de estas tareas. Según la documentación de Django el sitio ideal para que se ejecutara una única vez era en el paquete Apps, pero esto requiere de la inicialización de los modelos de la aplicación antes de poder usarlos, por lo que no se pudo hacer esto. Tras realizar una búsqueda exhaustiva, se dispuso el planificador de tareas en el paquete Urls, pues este sólo se lee una vez, y mientras los modelos de la aplicación ya se han inicializado.

Para las tareas en segundo plano se pueden destacar el borrado de métricas cada 15 días, de tal manera que se filtran todas las métricas que superen esas dos semanas, y se eliminan. Otra de las tareas, y una de las más relevantes, es la de analizar el estado de los equipos en segundo plano en base a las métricas y alertas de estos, lo que permite analizar el estado del equipo: si un equipo ha dejado de enviar métricas en un intervalo de tiempo establecido por el usuario, si el estado del equipo es crítico por su alto consumo de CPU o RAM, o porque el disco duro esté lleno. Si esto ocurre, existe otra tarea en segundo plano que permite mandar un mensaje al administrador a través de los sistemas que haya configurado, como pueden ser el correo electrónico, o peticiones *WebHook* para aplicaciones como Discord o Slack.

```
def startup_scheduling():
    """Function for the RQ scheduler to call at startup. Should be called only once."""
    # Delete any existing jobs in the scheduler when the app starts up
    for each in scheduler.get_jobs():
        each.delete()

    scheduler.schedule(
        scheduled_time=datetime.datetime.now(datetime.timezone.utc), # Time for first execution, in UTC timezone
        func=remove_old, # Function to be queued
        interval=30 * 60, # Time before the function is called again, in seconds
    )

    scheduler.schedule(
        scheduled_time=datetime.datetime.now(datetime.timezone.utc), # Time for first execution, in UTC timezone
        func=check_status, # Function to be queued
        interval=10 * 60, # Time before the function is called again, in seconds
    )
```

*Figura 17. Planificación de las tareas en segundo plano*

#### **5.2.2.2.4. Uso de filtros y paginación**

Uno de los elementos que también se considera relevante es la gestión de las URLs para poder mezclar el sistema de paginación con el sistema de filtrado, pues normalmente al aplicar la paginación, la URL la formaban los métodos que se heredaban de las clases basadas en vista, como pueden ser las ListView. Es por esto que para poder mezclar la paginación con filtrado y viceversa se han tenido que sobrescribir los métodos base de las clases, para lo cuál ha sido necesario ver la implementación de la clase ListView en el código fuente de Django.

En la Figura 18 se puede apreciar el uso combinado del filtrado por fechas, y de la paginación.

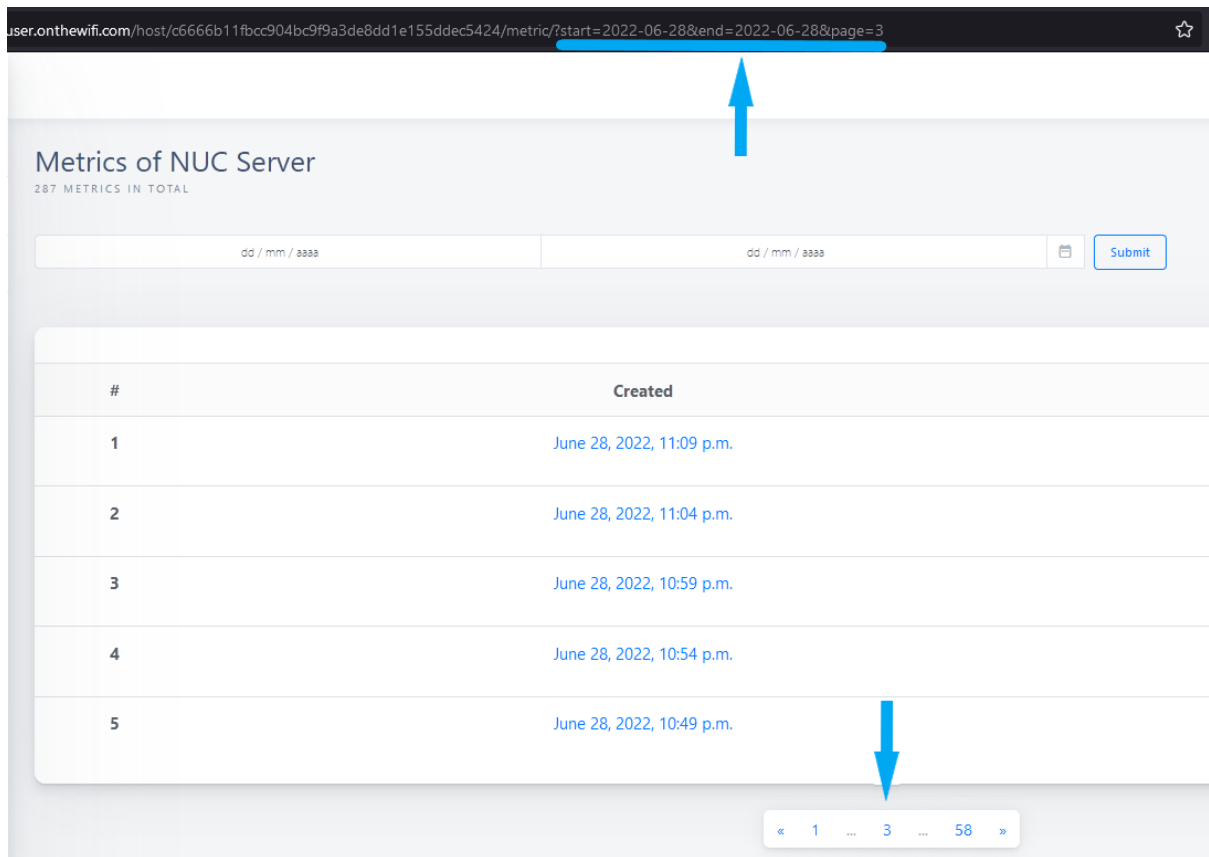


Figura 18. Filtrado y paginación

#### 5.2.2.2.5. Plantilla de datos recursiva

En el agente la información que se quería mostrar eran los datos de las últimas métricas, y mostrar mediante gráficas los datos de la última hora. Esto se consiguió tras tratar los datos, mandarlos al contexto web y que las gráficas fueran capaces de cargarlos y mostrarlos.

Para el detalle del agente y que mostrará la información que este le mandaba, se implementó una plantilla que permitía mostrar la información en formato JSON con coloración de sintaxis e indentación, de la misma forma que se podría visualizar en un editor de código.

Tras una de las reuniones pertenecientes a un sprint, en la parte de las métricas, se modificó la interfaz de tal manera que la información mostrada en formato JSON se mostrará en HTML. Esta fue una de las partes más complejas del sistema de plantillas a desarrollar pues implicaba mostrar de forma recursiva una consulta SQL que contenía los datos en formato JSON y, por ende, contenía pares clave-valor anidados.

Es decir, teniendo una serie de datos en formato JSON, los cuales están anidados uno dentro de otro el objetivo era representar en formato HTML desde el nodo padre hasta el último de los nodos hijos de forma anidada.

En la Figura 19 se puede ver cómo se renderiza en HTML una serie de datos anidados que pertenecían a un formato JSON.

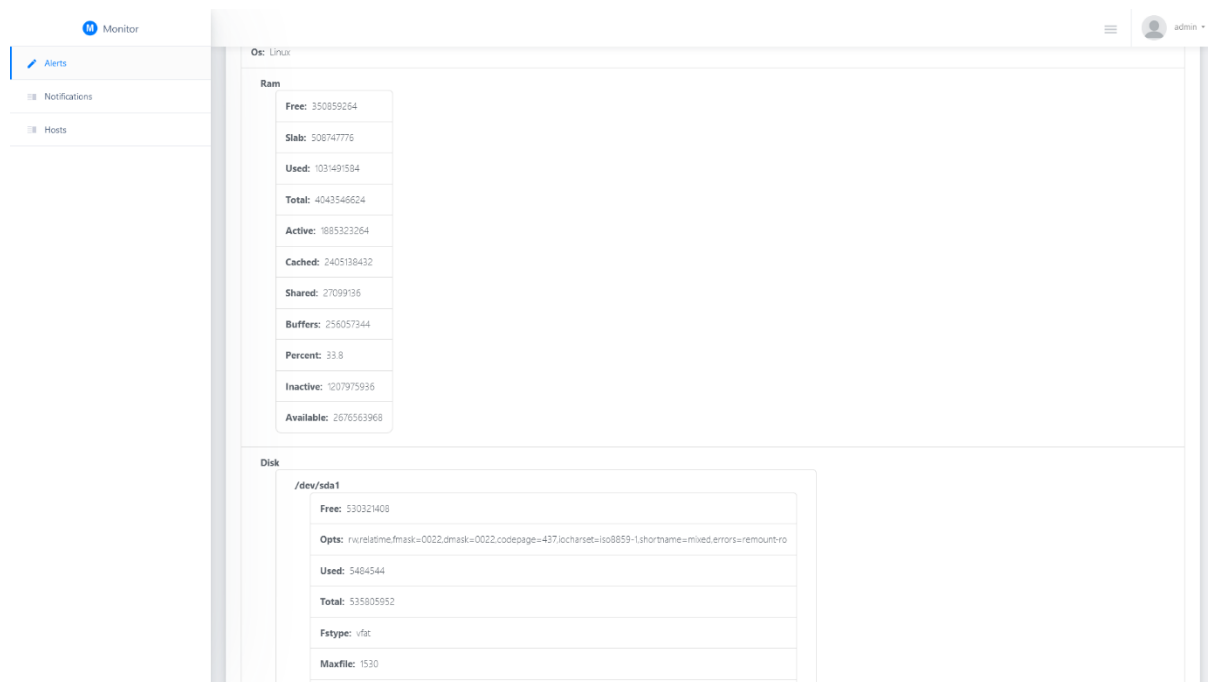


Figura 19. HTML con datos anidados

#### 5.2.2.2.6. Ejecución de comandos

A continuación, se desarrolló el formulario para la ejecución de comandos en los equipos. Para esto se realiza una petición a los equipos con el comando y dos timeout: el timeout de petición y el de comando. El timeout de petición finaliza la espera de la respuesta si no se ha producido esta antes del tiempo establecido por el usuario, y el timeout de comando permite que, si un comando no ha terminado antes del tiempo que ha establecido el usuario, se finalice la ejecución del comando y le devuelva el resultado en la petición. Teniendo en cuenta estos dos valores, es importante que el timeout de petición sea mayor que el del comando; esto es porque hay que tener en cuenta el tiempo que tarda en llegar la respuesta del agente al servidor principal, y si este junto con el del comando es mayor que el del timeout, la respuesta no llegará y se dará el caso de timeout de petición.

Una de las partes más complejas de la ejecución de comandos en la parte de la aplicación web ha sido el tratamiento de la respuesta, ya que hay comandos que utilizan colores ASCII para la terminal, y al enviarlos en la respuesta se muestran en HTML y aparecen como tal sin interpretarse. Para solucionar esto, se ha implementado un filtro mediante una expresión regular.



Tras consultar el estado visual con las tutoras en la finalización de esa iteración, se propuso modificar la interfaz del detalle del agente y de las métricas para ser más accesible y agradable al usuario, a su vez mostrando más información.

Esto se reflejó en mostrar en formato HTML los datos de la última métrica en el agente para que el administrador de sistemas pudiera valorar el estado del agente de una manera rápida y fácil.

### **5.2.2.3. Refactorización**

Refactorizar en términos simples es extraer métodos y funciones que hacen que el código resulte complejo, simplificar expresiones y organizar y agrupar el contenido de la aplicación, tanto los ficheros como las funciones. Refactorizar por lo general hace que el código sea más fácil de modificar, ahorrando así tiempo en el futuro, lo hace más fácil de entender, lo hace más eficiente, mejora el rendimiento general y específico de la aplicación, hace que los errores sean más fáciles de encontrar y mejora el diseño del sistema [22, 23, 33].

Al principio, por el desconocimiento de las vistas en Django se desarrollaron en funciones, por lo que el código para realizar una acción como mostrar datos era extensivo. Una mejora en base a estas líneas fue cambiar las vistas de la aplicación web de funciones a clases, de tal forma que estas heredan de las clases propuestas por el framework Django, facilitando código base que permitía reducir enormemente el código inicial. En casos en los que el código de Django no se adaptaba al caso concreto a desarrollar, se sobrescribieron los métodos para obtener el resultado deseado.

Otros casos son los de funciones que eran utilizados por estas vistas y resultaban complejas de comprender si no se tenía el contexto de la aplicación. En estos casos se han renombrado las variables y funciones, se ha escrito documentación y se han simplificado las expresiones, como puedan ser condiciones anidadas mediante el patrón de guarda, que permite anular múltiples condiciones anidadas utilizando una única condición crítica.

Para concluir con las refactorizaciones utilizadas, en Python existen multitud de recursos que permiten simplificar expresiones como puedan ser bucles o condiciones simples. Para ello existen recursos como las *list comprehensions* [Figura 20] y las condiciones en línea [Figura 21].

```
partitions = {  
    partition: {  
        "Used": format_bytes(values["used"]),  
        "Total": format_bytes(values["total"]),  
        "Mount Point": values["mountpoint"],  
        "File System Type": values["fstype"],  
    }  
    for partition, values in lastMetric["metrics"]["disk"].items()  
}
```

Figura 20. List Comprehension

```
end_status = "Warning" if end_status == "WR" else "Bad"
```

Figura 21. Condición en línea

En la aplicación general ha supuesto un incremento de la eficiencia en la programación por facilitar la reusabilidad de piezas de código que se extrajeron, evitando así duplicar métodos. Ha mejorado la legibilidad general del código al punto que una persona no experta en programación ni en Python pueda entender el código con simplemente leerlo. También ha mejorado el rendimiento de ciertos métodos debido a la simplificación del propio código redundante, pues ejecutar código doblemente únicamente incrementa la complejidad algorítmica  $O$  de las funciones.

### 5.2.3. Despliegue

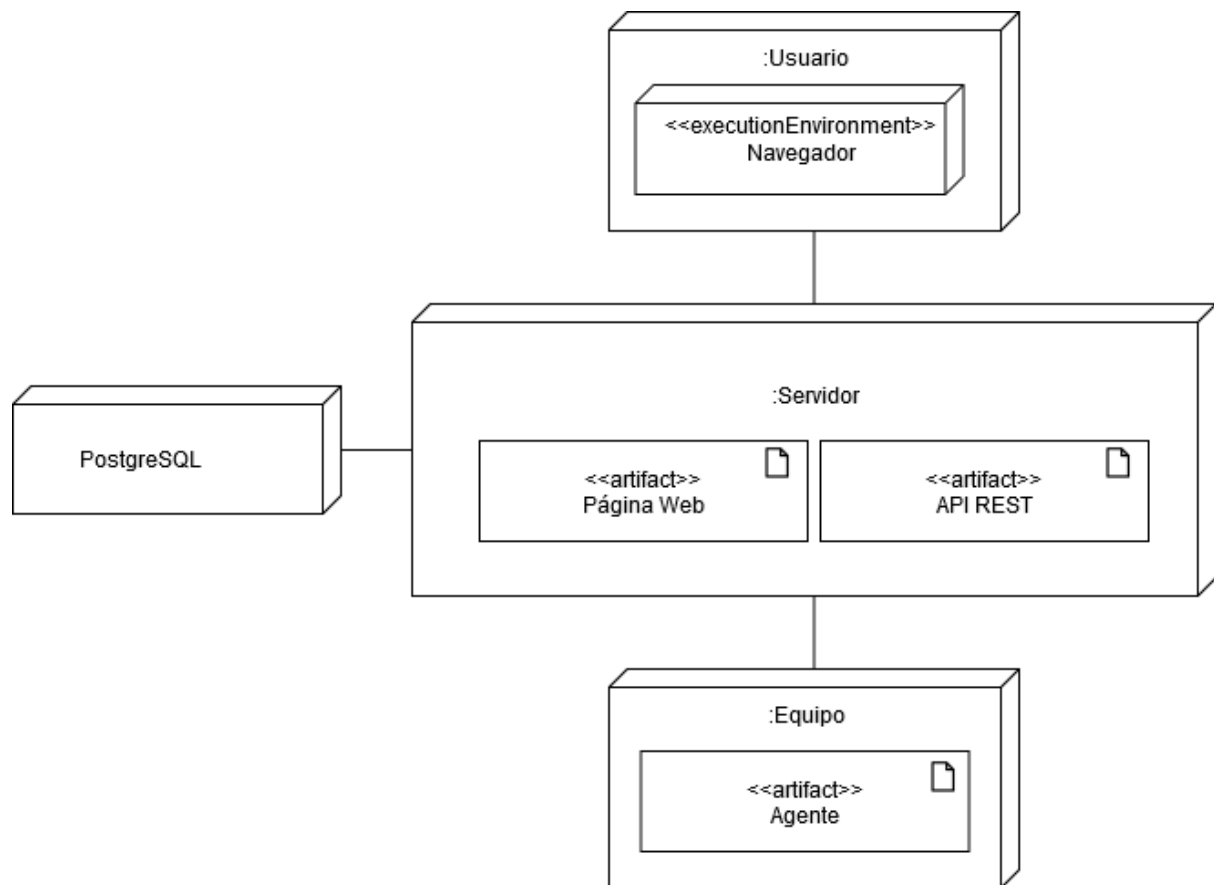


Figura 22. Diagrama de despliegue

En la Figura 22 se muestra un diagrama de despliegue con las distintas partes que componen el presente proyecto.

El despliegue es uno de los aspectos más relevantes, pues es el paso final para finalizar el proyecto, su instalación y ejecución. Al no ser un ejecutable ni un instalable, la forma de interactuar finalmente con la aplicación cambia. En el caso del agente, en Linux se despliega mediante un servicio de SystemD, que ejecuta el script “main.py” [Figura 23], aunque para ello se han de instalar las dependencias del agente. En Windows ocurre de la misma manera, pero en vez de utilizar SystemD que es propio de Linux, se utiliza Windows Task Scheduler.

```
n0nuser@server:~$ cat /etc/systemd/system/agent.service
[Unit]
Description=RESTful web to track agents
After=network.target network-online.target
Requires=network-online.target

[Service]
RemainAfterExit=true
Restart=on-failure
ExecStart=/usr/bin/python3 /home/n0nuser/monitor_agent/monitor_agent/main.py

[Install]
WantedBy=multi-user.target
```

Figura 23. Servicio de SystemD

Para el servidor, al utilizar múltiples servicios, para componer el sistema de manera sencilla sin tener que gestionar la instalación de cada servicio en un equipo mediante scripts, que probablemente fallen debido la variabilidad de los paquetes y características de cada sistema operativo, se utiliza un sistema virtual que permite mantener el entorno invariable. Para construir este entorno en cualquier sistema operativo se utiliza Docker con Docker-Compose. En el despliegue de la aplicación se utiliza Uvicorn y Nginx para el servidor web, Redis para el gestor de colas y la gestión de tareas en segundo plano y PostgreSQL para la base de datos, como se puede apreciar en la Figura 24.

```
n0nuser@server:~/monitor$ docker-compose ps
```

Name	Command	State	Ports
django-monitor-nginx	/docker-entrypoint.sh nginx ...	Up	0.0.0.0:80->80/tcp, :::80->80/tcp
django-monitor-postgres	docker-entrypoint.sh postgres	Up	5432/tcp
django-monitor-redis	docker-entrypoint.sh redis ...	Up	6379/tcp
django-monitor-uvicorn	/bin/sh -c "/entrypoint.sh"	Up	8000/tcp

```
n0nuser@server:~/monitor$
```

Figura 24. Procesos de Docker-Compose

Ha de mencionarse, que para configurar el despliegue de la aplicación web se ha de modificar el fichero “.env” [Figura 25] que mantiene las variables de entorno que se importarán al entorno virtual cuando Docker lo cree. De esta forma, la aplicación puede recoger los valores mediante librerías que obtienen datos del sistema operativo, en este caso las variables de entorno. Esto facilita la flexibilidad al usuario para configurar su aplicación sin tener que mirar el código fuente y modificar variables hardcodeadas, lo cual también es una mala práctica debido a los fallos de seguridad y privacidad a los que conduce.

```
1  ALLOWED_HOSTS=127.0.0.1,localhost,192.168.0.28
2  DEBUG=False
3  DJANGO_SUPERUSER_EMAIL=admin@monitor.tfg
4  DJANGO_SUPERUSER_PASSWORD=admin
5  DJANGO_SUPERUSER_USERNAME=admin
6  EMAIL_HOST=smtp.gmail.com
7  EMAIL_HOST_PASSWORD=password
8  EMAIL_HOST_USER=account@gmail.com
9  EMAIL_PORT=587
10 PORT=8000
11 POSTGRES_NAME=postgres
12 POSTGRES_PASSWORD=postgres
13 POSTGRES_USER=postgres
14 SECRET_KEY=myDummySecretKey
15 SERVER=YOUR FQDN
```

*Figura 25. Fichero .env*

Para realizar el despliegue, aparte del propio fichero Dockerfile y Docker-Compose.yaml [Figura 26] se ha añadido un script en Bash [Figura 27] para facilitar el despliegue de forma sencilla.

```

docker-compose.yml (compose-spec.json)
1  version: '3'
2  services:
3    monitor:
4      build: .
5      container_name: django-monitor-uvicorn
6      volumes:
7        - static_volume:/var/www/app/staticfiles
8      restart: always
9      expose:
10       - 8000
11      env_file:
12       - .env
13      links:
14       - postgres
15       - redis
16      depends_on:
17       - postgres
18       - redis
19
20    postgres:
21      image: postgres
22      container_name: django-monitor-postgres
23      restart: always
24      volumes:
25       - postgres_data:/var/lib/postgresql/data/
26      expose:
27       - 5432
28      environment:
29       - POSTGRES_DB=$POSTGRES_NAME
30       - POSTGRES_USER=$POSTGRES_USER
31       - POSTGRES_PASSWORD=$POSTGRES_PASSWORD
32
33    redis:
34      image: redis
35      restart: always
36      container_name: django-monitor-redis
37
38    nginx:
39      build: ./nginx
40      container_name: django-monitor-nginx
41      restart: always
42      volumes:
43       - static_volume:/var/www/app/staticfiles
44      ports:
45       - $PORT:80
46      links:
47       - monitor
48      depends_on:
49       - monitor
50
51
52    volumes:
53      postgres_data:
54      static_volume:
55

```

Figura 26. Fichero Docker-Compose.yml

```
1 echo "Updating requirements.txt with poetry.lock"
2 poetry export -f requirements.txt --output requirements.txt --without-hashes
3
4 echo "Docker-Compose Build"
5 docker-compose build
6 echo "Docker-Compose Deploy"
7 docker-compose up -d
```

Figura 27. Script para el despliegue de Docker

Al utilizar el entorno virtual con Poetry [Figura 28] tanto el agente como la aplicación web, las dependencias se pueden exportar con sus versiones a un fichero "requirements.txt" que permite instalar las dependencias, y así poder ejecutar la aplicación en cualquier equipo.

```

1  [tool.poetry]
2  name = "monitor"
3  version = "0.1.0"
4  description = "Web app that monitorizes and manages agents."
5  authors = ["Pablo González Rubio <pjgr2000@gmail.com>"]
6  repository = "https://github.com/n0nuser/monitor"
7  license = "GPL-3.0-or-later"
8  packages = [
9      { include = "monitor/" },
10     { include = "monitor/**/*.py" },
11 ]
12
13 [tool.poetry.dependencies]
14 Unipath = "^1.1"
15 asgiref = "^3.5.0"
16 autopep8 = "^1.6.0"
17 dj-database-url = "^0.5.0"
18 django = ">=4.0,<5.0"
19 django-enviro = "^0.9.0"
20 django-import-export = "^2.7.1"
21 djangorestframework = "^3.13.1"
22 gunicorn = "^20.1.0"
23 psycopg2-binary = "^2.9.3"
24 pycodestyle = "^2.8.0"
25 python = ">=3.8,<4.0.0"
26 python-decouple = "^3.6"
27 pytz = "^2021.3"
28 requests = "^2.27.1"
29 sqlparse = "^0.4.2"
30 uvicorn = "^0.17.6"
31 whitenoise = "^6.0.0"
32 redis = "^4.3.3"
33 rq = "^1.10.1"
34 django-rq = "^2.5.1"
35 rq-scheduler = "^0.11.0"
36 django-crispy-forms = "^1.14.0"
37 django-minify-html = "^1.3.0"
38 django-extensions = "^3.1.5"
39
40 [tool.poetry.dev-dependencies]
41 flake8 = "^4.0.1"
42 black = {version = "^22.3.0", allow-prereleases = true}
43 pygraphviz = "^1.9"
44
45 [tool.black]
46 line-length = 119
47
48 [tool.flake8]
49 max-line-length = 119
50
51 [build-system]
52 requires = ["poetry-core>=1.0.0"]
53 build-backend = "poetry.core.masonry.api"
54

```

Figura 28. Pyproject.toml



## 6. Conclusiones

En la presente memoria se ha expuesto en detalle el proceso de construcción de un sistema de monitorización, desde el diseño y conceptualización de la arquitectura, los requisitos funcionales y no funcionales, el sistema de paso de mensajes y la seguridad implicada en este hasta el diseño de la interfaz y el despliegue final.

Como conclusión de este proyecto, obtenemos una herramienta de monitorización de equipos Linux y Windows que permite recoger información en forma de métricas, y gestionar límites de CPU y RAM para generar alertas. Esta información se guarda en una base de datos, y sus datos son accesible a través de un API REST y de una aplicación web. La aplicación web permite gestionar múltiples equipos, así como la configuración de estos, y permite visualizar los datos recogidos en forma de gráficas para tener una visibilidad general del estado del equipo, así como en formato HTML para los datos del equipo, y para el detalle de cada métrica, es decir, su información específica. También se permiten configurar dos sistemas de notificación como son correos electrónicos, y *WebHooks*, tantos como desee el usuario, por donde recibirán alertas referentes a sus equipos.

Del desarrollo de este proyecto se extraen diversas conclusiones, a nivel personal, técnico y profesional. En primer lugar, respecto a los objetivos técnicos alcanzados, a continuación, se analiza la consecución de cada uno de ellos:

- Diseño y refinamiento de la arquitectura. Se ha diseñado una arquitectura que soporta múltiples agentes haciendo peticiones a un servidor central a través de distintas capas que se han ido perfeccionando en base al público objetivo y el uso final de la aplicación.

- Gestión de las métricas. Permite recoger datos del sistema operativo del equipo, tratarlas para enviar en formato JSON y almacenarlas en el sistema para su posterior análisis y presentación.

- Gestión de las alertas. Permite gestionar límites de CPU y RAM y envía una alerta al sistema, que bajo ciertas condiciones cambia el estado del equipo y notifica al usuario.

- Gestión de la ejecución de comandos. Permite ejecutar comandos y procesos en un equipo de forma remota tanto en Linux como en Windows a través de peticiones a un API REST.

- Gestión de la autenticación. Permite gestionar la seguridad de la plataforma mediante permisos y la gestión de sus usuarios, de tal forma que un usuario no pueda acceder al contenido de otro, o que un usuario malicioso se aproveche de la plataforma y pueda ejecutar comandos en equipos de usuarios de la aplicación. También se permite al usuario recuperar su contraseña a través de un email a su dirección de correo electrónico, o cambiar su contraseña desde su perfil de usuario.

- Gestión de las notificaciones. Permite mandar alertas al usuario mediante correos electrónicos a los emails que haya especificado, y mandar mensajes mediante *webhooks* a servicios que haya especificado como puedan ser Discord, Telegram o Slack.

- Gestión de las peticiones. Se ha tenido muy en cuenta la gestión de las peticiones y la secuencia adecuada de estas para el correcto funcionamiento de la arquitectura en la aplicación.

- Gestión de tareas en segundo plano. Permite ejecutar tareas, funciones y métodos de forma concurrente con el programa principal, de tal forma que no produce bloqueos a nivel de sistema operativo y permite al programa principal seguir recibiendo peticiones.

- Gestión de la configuración. Permite al usuario configurar el agente como mejor se adapte a sus necesidades mediante un fichero de configuración en el mismo. Al ser una tarea compleja si no existiera un fichero de referencia, se creó un formulario que permite crear y guardar la configuración del agente, para luego ser mostrada, copiada y exportada por el usuario.

- Gestión del despliegue. Permite ejecutar la aplicación web y el agente en cualquier sistema operativo de forma que esto no suponga una limitación para el usuario ni su entorno de trabajo.

- Gestión del panel de administración. Además de la propia gestión de los datos en la página web, se ha incluido un panel de control que permite gestionar de manera más directa los objetos de la base de datos, así como filtrarlos y poder importar y

exportar estos modelos desde/a una serie de ficheros con formato como pueden ser XLS, CSV, TSV, JSON, YAML, Pandas DataFrames, HTML, Jira, ODS y DBF.

- Optimización y refactorización del código. Haber optimizado y refactorizado el código ha sido una de las labores más satisfactorias del proyecto. Ver cómo el rendimiento, la modularidad y la legibilidad de la aplicación aumenta, a la vez que disminuye la probabilidad de errores futuros, es una experiencia realmente grata.

Por otro lado, gracias a la realización de este proyecto se han adquirido una serie de conocimientos provechosos. Cada una de las partes del proyecto, desde el diseño y conceptualización hasta el despliegue han aportado al producto final.

Algunos de los conocimientos que se han aprendido, desarrollado y afianzado con este trabajo son el desarrollo con Django y distintos *frameworks* para APIs REST como fue al inicio Flask y después FastAPI en el agente, y Django REST Framework para la aplicación web. Así como la gestión de servicios integrados en aplicaciones web, desplegadas en entornos virtuales replicables como Docker.

Todos estos conocimientos adquiridos serán muy importantes en las siguientes etapas como desarrollador, y el haberlos adquirido con este proyecto le otorga más valor si cabe al trabajo.

A nivel personal ha sido una experiencia larga de un año y medio, y costosa por la falta de tiempo debido estudiar y trabajar al mismo tiempo. Pese a estas dificultades, ha sido un proyecto altamente gratificante por mezclar apartados de la informática como la administración de sistemas, la gestión de las redes, los sistemas distribuidos, el desarrollo *backend*, la programación en Python, el despliegue de servicios y las buenas prácticas de programación y el uso de patrones establecidos para la mejora general del proyecto *software*.



## **7. Líneas futuras**

En este apartado se enumeran alternativas y mejoras al sistema actual que por falta de tiempo o de conocimiento no se han podido llevar a cabo.

### **7.1. Integraciones**

La aplicación podría notificar a los usuarios a través de más medios de comunicación como podría ser SMS, para ello se podrían utilizar APIs que ofrecen este servicio, o utilizar servicios en Linux que permitan el manejo de módems GSM.

Esto facilita la recepción de alertas a usuarios en cuya localización no dispongan de acceso a internet o debido a su calidad, el sistema de llamadas o mensajes sea preferible.

### **7.2. Análisis de datos**

Teniendo en cuenta los datos que se almacenan en la base de datos sobre el historial de inicios de sesión, las métricas y las alertas junto las peticiones que se almacenan los logs, se podría hacer un análisis de esta información y presentar al usuario resúmenes relevantes del estado de sus equipos en forma de informes o en la misma aplicación web.

### **7.3. Predicción de fallos**

Con los datos almacenados de las métricas y alertas se puede intentar estimar bajo qué circunstancias se puede volver a dar un cambio significativo de estado en el equipo, y cuándo. Para ello se puede utilizar inteligencia artificial.

### **7.4. Gestor de colas en el agente**

Implementar un gestor de colas en el agente permitiría que en el caso de que no se lleguen a enviar las peticiones por algún error, estas se guarden en la cola hasta que se puedan enviar. Para la cola se puede utilizar Redis.

El agente al no poder desplegarse en Docker (pues monitorizaría la instancia de Docker en vez del equipo que la mantiene), necesitaría que el servicio de colas se instalara también en el equipo.

## **7.5. Mejora de la arquitectura**

La arquitectura se podría mejorar utilizando para hacer las consultas el prototipo 2 de la arquitectura, de esta forma la aplicación sería RESTful al basarse plenamente en consultas al API REST.

Para mejorar la consistencia y la replicabilidad se podrían utilizar varias instancias de API REST y manejarlas a través de un API Gateway de tal forma que sea un proxy y la aplicación que el usuario utilice, sólo vea una API REST.

Para aclaración, un API Gateway es por decirlo de manera sencilla un multiplexor de peticiones a un API REST, muy similar a las tablas NAT de un router.

## **7.6. Tiempo Real**

Para favorecer un uso realista de la aplicación y ver en tiempo real el estado del equipo se podría utilizar una base de datos como InfluxDB [30], y que el agente pudiera optimizar su extracción de datos en menos tiempo. Esto se podría hacer evitando extraer datos cuya variabilidad en tiempo no sea tan frecuente.

Una vez almacenados los datos, se podrían presentar al usuario mediante AJAX y sus peticiones asíncronas.

## **7.7. Uso de WebSockets**

El uso de *WebSockets* sería una mejora sustancial en el rendimiento de la aplicación y la conexión, además de ofrecer funcionalidades como la ejecución de comandos de forma continua como si de una terminal se tratase [35].

A nivel de red, evitaría abrir y cerrar un socket por cada petición que se realiza, al mantener abierta la conexión en todo momento.

## 8. Referencias

1. *About issues*. GitHub Docs. (n.d.). Retrieved June 27, 2022, from <https://docs.github.com/en/issues/tracking-your-work-with-issues/about-issues>
2. *About Project Boards*. GitHub Docs. (n.d.). Retrieved June 27, 2022, from <https://docs.github.com/en/issues/organizing-your-work-with-project-boards/managing-project-boards/about-project-boards>
3. Atlassian. (n.d.). *Scrum: Qué es, cómo funciona y por qué es excelente*. Atlassian. Retrieved June 27, 2022, from <https://www.atlassian.com/es/agile/scrum>
4. *Automatically improve Python Code Quality*. Sourcery. (n.d.). Retrieved June 27, 2022, from <https://sourcery.ai/>
5. *Chart.js*. Chart.js | Chart.js. (n.d.). Retrieved June 27, 2022, from <https://www.chartjs.org/docs/latest/>
6. Chi, C. (2022). *Fitts's law: The UX hack that will strengthen your design*. HubSpot Blog. Retrieved June 27, 2022, from <https://blog.hubspot.com/marketing/fitts-law>
7. Christie, T. (n.d.). *Authentication*. Authentication - Django REST framework. Retrieved June 27, 2022, from <https://www.django-rest-framework.org/api-guide/authentication/>
8. Christie, T. (n.d.). *Filtering*. Filtering - Django REST framework. Retrieved June 27, 2022, from <https://www.django-rest-framework.org/api-guide/filtering/>
9. Christie, T. (n.d.). *Quickstart*. Quickstart - Django REST framework. Retrieved June 27, 2022, from <https://www.django-rest-framework.org/tutorial/quickstart/>
10. *The Collaborative Interface Design Tool*. Figma. (n.d.). Retrieved June 27, 2022, from <https://www.figma.com/>
11. *Commands*. Poetry. (n.d.). Retrieved June 27, 2022, from <https://python-poetry.org/docs/cli/>
12. *CRAP Design Principles*. CRAP Design Principles: Technical Writing, Spring 2021. (n.d.). Retrieved June 27, 2022, from <https://canvas.vt.edu/courses/126055/pages/crap-design-principles>
13. *Django Documentation*. Django. (n.d.). Retrieved June 27, 2022, from <https://docs.djangoproject.com/en/4.0/>





28. Home. OpenAPI Initiative. (2022). Retrieved June 27, 2022, from <https://www.openapis.org/>
29. *How to schedule python script using Windows Scheduler*. Data to Fish. (2020, May 17). Retrieved June 27, 2022, from <https://datatofish.com/python-script-windows-scheduler/>
30. *InfluxDB: Open-source time series database*. InfluxData. (2022, June 17). Retrieved June 27, 2022, from <https://www.influxdata.com/>
31. Jazzband. (n.d.). *Jazzband/django-debug-toolbar: A configurable set of panels that display various debug information about the current request/response*. GitHub. Retrieved June 27, 2022, from <https://github.com/jazzband/django-debug-toolbar>
32. Mark Otto, J. T. (n.d.). *Documentation*. Bootstrap v4.6. Retrieved June 27, 2022, from <https://getbootstrap.com/docs/4.6/getting-started/introduction/>
33. Martin, R. C. (2009). *Clean code. A Handbook of Agile Software Craftsmanship*. Pearson Education.
34. *The modern Static Analysis Platform*. DeepSource. (n.d.). Retrieved June 27, 2022, from <https://deepsource.io/>
35. Pekarsky, M. (2020, January 15). *WebSockets for fun and profit*. Stack Overflow Blog. Retrieved June 27, 2022, from <https://stackoverflow.blog/2019/12/18/WebSockets-for-fun-and-profit/>
36. Phan, D. (2020). *How to run your first task with RQ, Redis, and Python*. Twilio Blog. Retrieved June 27, 2022, from <https://www.twilio.com/blog/first-task-rq-redis-python>
37. Postman. (n.d.). Retrieved June 27, 2022, from <https://www.postman.com/>
38. Postmanlabs. (n.d.). *Postmanlabs/httpbin: HTTP request & response service, written in Python + flask*. GitHub. Retrieved June 27, 2022, from <https://github.com/postmanlabs/httpbin>
39. Prometheus. (n.d.). *Introducing Prometheus agent mode, an efficient and cloud-native way for metric forwarding: Prometheus*. Prometheus Blog. Retrieved June 27, 2022, from <https://prometheus.io/blog/2021/11/16/agent/>
40. Prometheus. (n.d.). *Prometheus - Monitoring System & Time Series Database*. Prometheus Blog. Retrieved June 27, 2022, from <https://prometheus.io/>
41. Psf. (n.d.). *PSF/Black: The uncompromising Python code formatter*. GitHub. Retrieved June 27, 2022, from <https://github.com/psf/black>

42. *PSUTIL documentation*. psutil documentation - psutil 5.9.1 documentation. (n.d.). Retrieved June 27, 2022, from <https://psutil.readthedocs.io/en/latest/>
43. Purkayastha, S. (2022). *Top 15 best python rest api frameworks(2022): Rapidapi*. The Last Call - RapidAPI Blog. Retrieved June 27, 2022, from <https://rapidapi.com/blog/best-python-api-frameworks/>
44. PyCQA. (n.d.). *PyCQA/Flake8: Flake8 is a python tool that glues together pycodestyle, pyflakes, McCabe, and third-party plugins to check the style and quality of some python code*. GitHub. Retrieved June 27, 2022, from <https://github.com/PyCQA/flake8>
45. *Python 3.10.5 documentation*. 3.10.5 Documentation. (n.d.). Retrieved June 27, 2022, from <https://docs.python.org/3/>
46. The Postman Team (2022). *What is a rest api? examples, uses, and challenges*. Postman Blog. Retrieved June 27, 2022, from <https://blog.postman.com/rest-api-examples/>
47. *Top 10 Django apps and why companies are using it?* GeeksforGeeks. (2020, April 18). Retrieved June 27, 2022, from <https://www.geeksforgeeks.org/top-10-django-apps-and-why-companies-are-using-it/>
48. *Uvicorn Deployment*. Uvicorn. (n.d.). Retrieved June 27, 2022, from <https://www.uvicorn.org/deployment/#running-behind-nginx>
49. *Welcome to Django-Environ documentation*. Django. (n.d.). Retrieved June 27, 2022, from <https://django-environ.readthedocs.io/en/latest/>
50. *Whitenoise*. WhiteNoise 6.2.0 documentation. (n.d.). Retrieved June 27, 2022, from <https://whitenoise.readthedocs.io/en/latest/>
51. World Leaders in Research-Based User Experience. (n.d.). *F-shaped pattern of reading on the web: Misunderstood, but still relevant (even on mobile)*. Nielsen Norman Group. Retrieved June 27, 2022, from <https://www.nngroup.com/articles/f-shaped-pattern-reading-web-content/>
52. Yenigün, O. (2022,). *Solid principles explained*. Medium. Retrieved June 27, 2022, from <https://towardsdev.com/solid-principles-explained-635ad3608b20?gi=c7ba151f68cb>