UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

GRADO EN INGENIERÍA INFORMÁTICA

Gemelo digital de la infraestructura de autobuses de Málaga

Digital twin of the bus infrastructure of the city of Malaga

Realizado por

Iván Sánchez Rojas

Tutorizado por

Lola Burgueño Caballero

Julia Robles Medina

Departamento

Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, septiembre de 2025

Abstract

A digital twin is a virtual replica of a physical system that, through the integration of real-time data, allows the simulation, analysis, and optimization of complex infrastructures. In recent years, this technology has gained prominence due to its ability to improve efficiency, sustainability, and decision-making in sectors such as urban mobility.

This work consists of the development of a digital twin of the urban bus infrastructure of Málaga. This model uses data provided by the open data portal of the Málaga City Council, which includes the position of each bus, the lines they travel and the stops, allowing the visualization and analysis of the network's behavior in real time. Thanks to this, it is possible to identify congestion points, calculate the average speed of buses on segments of their route, predict arrival times at different stops, and detect delays.

For the implementation of this work, the OpenTwins platform has been used, establishing a Kubernetes-based cluster as the target environment. Development and testing have been carried out on a virtual machine with Debian 12 operating system, on which the K3s service, a lightweight Kubernetes distribution, has been deployed and configured to manage the orchestration of the different components of the system, among which Eclipse Ditto stands out, responsible for the management of digital twin data; InfluxDB, used for the efficient storage and querying of time series; and Grafana, which facilitates the visualization and monitoring of the data through interactive panels and customizable dashboards.

Keywords: Digital twin, Kubernetes, Eclipse Ditto, Smart cities, Transport infraestructure

Resumen

Un gemelo digital es una réplica virtual de un sistema físico que, mediante la integración de datos en tiempo real, permite simular, analizar y optimizar el funcionamiento de infraestructuras complejas. En los últimos años, esta tecnología ha ganado protagonismo por su capacidad para mejorar la eficiencia, la sostenibilidad y la toma de decisiones en sectores como la movilidad urbana.

El trabajo consiste en el desarrollo de un gemelo digital de la infraestructura de autobuses urbanos de Málaga. Este modelo utiliza datos proporcionados por el portal de datos abiertos del ayuntamiento de Málaga, que incluyen la posición de cada autobús, las líneas que recorren y las paradas, permitiendo visualizar y analizar el comportamiento de la red en tiempo real. Gracias a ello, es posible identificar puntos de congestión, calcular la velocidad media de los autobuses en tramos de su recorrido, predecir tiempos de llegada a las distintas paradas y detectar demoras.

Para la implementación de este trabajo se ha empleado la plataforma *OpenTwins*, estableciendo como entorno objetivo un clúster basado en *Kubernetes*. El desarrollo y las pruebas se han realizado sobre una máquina virtual con sistema operativo *Debian 12*, en la que se ha desplegado y configurado el servicio *K3s*, una distribución ligera de Kubernetes, para gestionar la orquestación de los distintos componentes del sistema, entre los que cabe destacar *Eclipse Ditto*, responsable de la gestión de los datos de los gemelos digitales; *InfluxDB*, utilizado para el almacenamiento y consulta eficiente de series temporales; y *Grafana*, que facilita la visualización y monitorización de los datos a través de paneles interactivos y dashboards personalizables.

Palabras clave: Gemelo digital, Kubernetes, Eclipse Ditto, Ciudades Inteligentes, Infraestructura de transporte

Índice

1 Introducción

1.1. Motivación

Las ciudades actuales, impulsadas por el avance tecnológico, enfrentan desafíos relacionados con la optimización de recursos, la sostenibilidad ambiental y la mejora de la calidad de vida de la ciudadanía. Sin embargo, existe una brecha entre el potencial de las tecnologías emergentes y su efectiva integración en la gestión diaria de la ciudad. El desarrollo de este Trabajo de Fin de Grado surge ante la creciente complejidad y exigencia de gestionar entornos urbanos de manera eficiente e inteligente.

La idea principal de este proyecto reside en aportar una solución concreta a esa brecha, centrándose específicamente en la infraestructura de autobuses urbanos de la ciudad de Málaga, operados por la EMT (Empresa Malagueña de Transportes), aprovechando la disponibilidad de datos abiertos de alta calidad proporcionados por el Ayuntamiento de Málaga. Se busca responder a necesidades detectadas, como la falta de plataformas que permitan una visualización integral y análisis exhaustivo de los datos urbanos, la insuficiencia de herramientas predictivas fácilmente adoptables por los organismos públicos, y la falta de soluciones accesibles y adaptables que fomenten la transparencia y la participación en la gestión urbana. Una herramienta de este tipo puede contribuir a solventar situaciones de congestión en el centro y los principales accesos a la ciudad, lo cual repercute negativamente en la puntualidad de los autobuses operados por la EMT.

1.2. Objetivos

Los objetivos generales de este trabajo se resumen en aportar una solución tecnológica concreta a las necesidades identificadas en el ámbito de la gestión urbana inteligente, específicamente en el contexto del transporte en autobús de Málaga. Para ello, se han establecido los

siguientes objetivos de alto nivel:

- Recoger mediante un proceso automatizado los distintos datos con los que se representarán los gemelos digitales, realizando un procesamiento cuando sea necesario para incluirlos en la plataforma con el formato adecuado.
- Modelar una representación digital de los principales elementos del sistema físico —autobuses, paradas y líneas— mediante gemelos digitales que reflejen con precisión sus características y estado.
- Creación de un panel de control que ofrezca una visualización integral y en tiempo real del estado y la localización de toda la flota de autobuses en el momento de la consulta.
- Creación de un panel de control que permita visualizar la situación concreta de un autobús, mostrando las posiciones por las que ha pasado, el trazado de la ruta que recorre y las distintas paradas del trayecto.
- Desarrollo de un módulo que otorgue a la solución de capacidades predictivas que permitan anticipar una posición futura en un intervalo de tiempo desde la última posición conocida.
- Implementar la solución con una arquitectura modular y abierta, garantizando su accesibilidad y personalización, permitiendo así su integración en distintos entornos y su evolución a futuro.

La necesidad fundamental que se busca suplir es la ausencia de plataformas accesibles, personalizables y orientadas a la monitorización y análisis de datos urbanos, que integren de manera sencilla diversas fuentes de información y presenten los resultados en formatos comprensibles para perfiles técnicos y no técnicos.

1.3. Metodología

Para el desarrollo de este Trabajo de Fin de Grado se ha adoptado una metodología iterativa-incremental, un enfoque ampliamente reconocido en la ingeniería de software por su capacidad de adaptarse a requisitos cambiantes y gestionar la complejidad de los proyectos de manera

progresiva. Este modelo, que forma parte de las llamadas metodologías ágiles, permite entregar incrementos de funcionalidad plenamente operativos en periodos cortos, facilitando la integración continua de mejoras y la adaptación ante nuevas necesidades o descubrimientos surgidos durante el desarrollo [1, págs. 40-66].

Cada iteración consta de una serie de tareas definidas al inicio de cada ciclo, priorizando los objetivos en función del estado del proyecto y de las necesidades detectadas en las revisiones previas. Al término de cada iteración, se realiza una revisión conjunta con las tutoras, quienes aportan feedback y orientaciones. Estas reuniones de seguimiento están programadas cada 2 o 3 semanas, ajustando las siguientes tareas a implementar según los avances conseguidos y los problemas surgidos.

Las principales características del proceso son:

- Planificación incremental: Se establecen fases cortas con objetivos claros y alcanzables, en lugar de intentar definir todos los requisitos al inicio.
- Implementación por tareas: En cada ciclo, se seleccionan y desarrollan funcionalidades concretas, priorizando aquellas de mayor impacto o criticidad para el sistema.
- Revisión y mejora continua: Tras cada entrega parcial, se lleva a cabo una evaluación para identificar posibles mejoras, errores o nuevas necesidades, que serán abordadas en el siguiente ciclo.
- Interacción frecuente con las tutoras: La comunicación periódica permite ajustar el rumbo del proyecto y garantizar que el desarrollo cumple con los requisitos académicos y técnicos esperados.
- Flexibilidad y adaptación: La metodología permite modificar y replanificar tareas en función de las necesidades emergentes durante el desarrollo.

Esta estrategia parte de las premisas propias de las metodologías ágiles, entre las que se encuentra que las fases del desarrollo no siempre son predecibles ni lineales, por lo que se requiere flexibilidad y capacidad de adaptación, o que los requisitos cambian con el tiempo, siendo difícil prever todos desde el inicio. El resultado es un método que favorece la calidad, el control de riesgos y la entrega progresiva de resultados, asegurando que la solución propuesta

evoluciona de manera coherente con los objetivos iniciales y se adapta a los cambios y mejoras sugeridos a lo largo del proceso.

1.4. Estructura del documento

Con el objetivo de documentar el trabajo realizado de manera clara, ordenada y comprensible, la memoria de este Trabajo de Fin de Grado se ha estructurado en los siguientes apartados principales. Cada uno aborda un aspecto fundamental del proyecto, contribuyendo a una comprensión global y detallada de su diseño, desarrollo y resultados.

- *Capítulo 2. Antecedentes*

  Este apartado describe el contexto necesario para comprender y llevar a cabo este proyecto. Por ejemplo, describe la tecnología en la que se basa, como son los gemelos digitales y las ciudades inteligentes.

- *Capítulo 3. Análisis y Diseño*

  Aquí se describen y fundamentan las principales decisiones tomadas durante el análisis y diseño del sistema:

  - Problema / Requisitos:

    Planteamiento del problema a resolver, necesidades detectadas y objetivos funcionales. Incluye la motivación de crear paneles de control inteligentes y capacidades de predicción, describiendo qué se pretende mostrar y medir, y cómo dichas necesidades guiaron el diseño.

  - Diseño conceptual:

    Presentación de los modelos conceptuales que dan forma a la solución propuesta.

  - Tecnologías usadas:

    Revisión de las tecnologías, frameworks, librerías y herramientas seleccionadas, justificando su uso en relación a los requisitos y objetivos del proyecto.

  - Arquitectura:

    Desglose concreto y visual de la arquitectura implementada, apoyado con un esquema que refleje los componentes principales y sus relaciones.

- *Capítulo 4. Implementación*

  En este capítulo se explican en detalle, de forma individual, cada uno de los componentes que constituyen la solución desarrollada. Se abordan sus características técnicas, funcionalidades y el rol que desempeñan dentro del sistema, proporcionando una visión clara y comprensible de la aportación de cada módulo a la arquitectura global

- *Capítulo 5. Pruebas*

  Este bloque especifica el enfoque adoptado para asegurar la calidad y robustez del sistema:

  - Pruebas manuales de funcionalidad y usabilidad:
    Revisiones efectuadas de manera interna, validando que el sistema cumple con las expectativas planteadas.

  - Pruebas de tolerancia a errores:
    Estrategias para garantizar la resistencia y estabilidad del sistema ante fallos.

- *Capítulo 6. Conclusiones y Líneas Futuras*

  Recoge las principales conclusiones derivadas del trabajo, evaluando el grado en que se alcanzaron los objetivos y el impacto de la solución propuesta. Además, se incluyen propuestas de mejora y posibles líneas de desarrollo futuro vinculadas a la evolución de tecnologías y necesidades detectadas en el entorno de las ciudades inteligentes y transporte urbano. En último lugar, se añade también un apartado de valoraciones personales con algunos pensamientos del autor acerca del proyecto.

2

Antecedentes

2.1. Ciudades inteligentes

La transformación digital en el entorno urbano está redefiniendo la manera en que las ciudades abordan sus desafíos más complejos. En este contexto, las **smart cities** representan una evolución natural hacia sistemas urbanos más eficientes, sostenibles y centrados en el ciudadano. Según la Comisión Europea, una smart city es "un lugar donde las redes y servicios tradicionales se vuelven más eficientes mediante el uso de soluciones digitales en beneficio de sus habitantes y empresas" [2]. Esta definición subraya que las ciudades inteligentes trascienden el simple uso de tecnologías digitales para convertirse en ecosistemas integrados que optimizan recursos, reducen emisiones y mejoran la calidad de vida urbana.

El desarrollo de ciudades inteligentes se fundamenta en la integración de múltiples componentes tecnológicos y sociales: infraestructuras inteligentes equipadas con sensores avanzados, dispositivos del Internet de las Cosas (IoT) y redes de comunicación para la recopilación y transmisión de datos urbanos en tiempo real [3]. Esta masiva generación de datos requiere, a su vez, sistemas capaces de procesarlos mediante algoritmos avanzados e inteligencia artificial para generar información procesable sobre los complejos problemas urbanos multifacéticos que enfrentan las ciudades modernas [3].

2.2. Gemelos digitales

En este panorama de transformación urbana, los **gemelos digitales (digital twins)** emergen como una tecnología disruptiva que revoluciona la comprensión y gestión de los sistemas físicos urbanos [4]. Existen numerosas definiciones de gemelo digital. Una de ellas, la cual está bastante extendida, lo define como una representación virtual de un activo físico habilitada mediante datos y simuladores para la predicción, optimización, monitoreo, control y mejora

de la toma de decisiones en tiempo real [5]. En el contexto urbano, estos sistemas combinan datos estáticos, históricos y dinámicos procesados mediante análisis avanzados y/o inteligencia artificial para proporcionar información valiosa sobre el rendimiento de la ciudad.

La aplicación de los gemelos digitales al ámbito urbano permite crear réplicas virtuales de infraestructuras, procesos o servicios que posibilitan la simulación, monitoreo y análisis en tiempo real. Esta capacidad resulta especialmente valiosa para la gestión del transporte urbano, donde la complejidad de las interacciones entre usuarios, vehículos, infraestructura y condiciones ambientales requiere soluciones avanzadas de modelado y predicción. En particular, el sector del transporte público enfrenta desafíos significativos relacionados con la optimización de rutas, la predicción de la demanda, la gestión de flotas y la mejora de la experiencia del usuario [4].

Según investigaciones recientes, los gemelos digitales pueden transformar fundamentalmente la planificación y gestión del transporte urbano. Un estudio de ABI Research estima que las ciudades podrían ahorrar hasta $280 mil millones para 2030 mediante la implementación de gemelos digitales para una planificación urbana más eficiente [6]. En el ámbito específico del transporte público, estos sistemas permiten la simulación de diferentes escenarios operativos, la evaluación del impacto de cambios en las rutas u horarios, y la optimización de la asignación de recursos antes de su implementación en el mundo real.

La evolución hacia sistemas de transporte más inteligentes y eficientes ha impulsado el desarrollo de múltiples iniciativas tecnológicas. Sin embargo, muchas de estas soluciones se basan en plataformas propietarias que limitan la innovación, la colaboración y la reutilización de conocimientos [7]. En respuesta a estas limitaciones, surge la necesidad de plataformas abiertas que democraticen el acceso a las tecnologías de gemelos digitales y fomenten la innovación colaborativa.

2.3. OpenTwins

En este contexto, **OpenTwins** representa una iniciativa pionera desarrollada por el grupo de investigación ERTIS de la Universidad de Málaga [8]. Esta plataforma constituye el primer ecosistema abierto y completo de gemelos digitales, diseñada específicamente para facilitar el desarrollo de gemelos digitales de nueva generación más accesibles y versátiles. La plataforma se caracteriza por su capacidad para combinar las últimas tecnologías de manera integrada:

Internet de las Cosas (IoT), inteligencia artificial (IA), monitorización, modelos de simulación y visualización tridimensional (3D).

El desarrollo de OpenTwins ha sido documentado en la revista científica Computers in Industry con el título "OpenTwins: An open-source framework for the development of next-gen compositional digital twins" [8]. Su código fuente está disponible en GitHub[1], cumpliendo con la filosofía de código abierto que busca democratizar el acceso a la tecnología de gemelos digitales. Como señala Manuel Díaz, catedrático de la Escuela de Ingeniería Informática de la UMA, "hasta ahora la mayoría de las plataformas existentes en este sentido eran de pago y no podían modificarse" [9].

La figura 1 ilustra el diagrama arquitectónico de OpenTwins, detallando la organización y las interrelaciones de sus distintos componentes software. Esta figura ha sido tomada directamente de la documentación oficial de la plataforma[2]

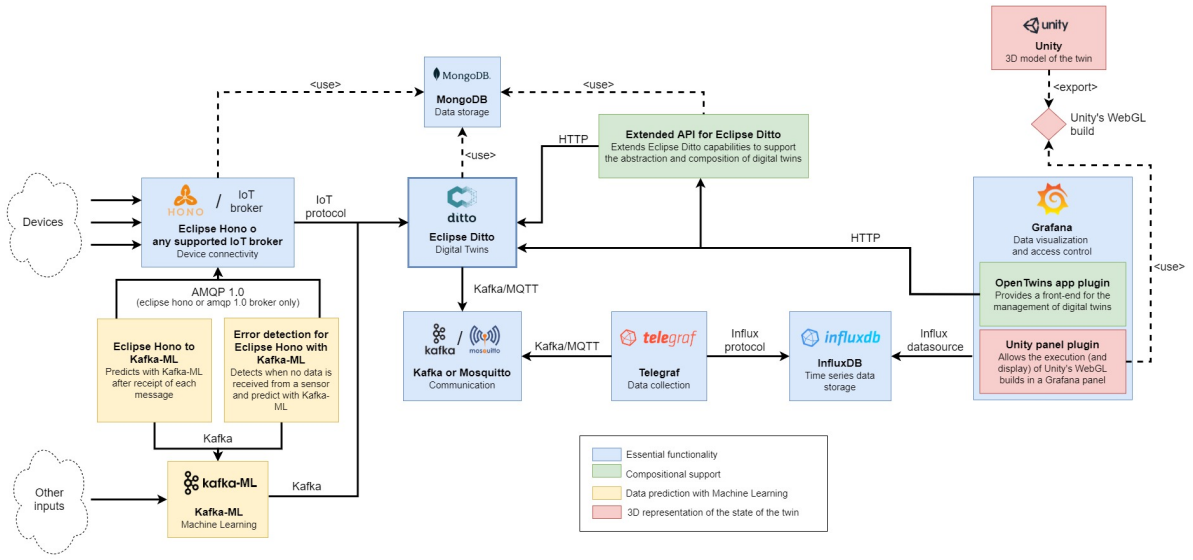


Figura 1: Arquitectura de OpenTwins.

La arquitectura de OpenTwins está estructurada en múltiples componentes interconectados, cada uno encargado de funciones específicas dentro del ecosistema de gemelos digitales. El flujo principal de la arquitectura se desarrolla desde la entrada de datos provenientes de dispositivos físicos o virtuales, pasando por la gestión, procesamiento y almacenamiento de datos, hasta la visualización avanzada y la interacción con modelos tridimensionales:

[1] https://github.com/ertis-research/opentwins
[2] https://ertis-research.github.io/opentwins/docs/overview/architecture

- *Conectividad de dispositivos (Eclipse Hono/IoT Broker):*

  Los dispositivos físicos envían sus datos a través de protocolos IoT utilizando Eclipse Hono, encargado de la conectividad y el manejo de mensajes, o bien directamente mediante cualquier broker IoT compatible.

- *Representación digital (Eclipse Ditto):*

  El componente central es Eclipse Ditto, responsable de la definición y gestión de los gemelos digitales que representan los activos físicos. Eclipse Ditto recibe los datos de los brokers IoT y actúa como punto de enlace entre el mundo físico y el digital.

- *Soporte y extensión composicional:*

  Sobre Eclipse Ditto, una API extendida habilita capacidades adicionales para la abstracción y composición de gemelos digitales, permitiendo modelar sistemas complejos como agregados de entidades más simples.

- *Machine learning:*

  La integración de la plataforma de gemelos digitales con Kafka-ML permite dotar de inteligencia a los gemelos digitales mediante modelos de machine learning y deep learning sobre flujos de datos en tiempo real, facilitando tanto la predicción avanzada de estados como la detección de fallos o interrupciones en sensores a través de la generación automática de datos simulados cuando sea necesario [10].

- *Almacenamiento de datos:*

  Los datos de los gemelos digitales y sus históricos se almacenan en MongoDB (para datos estructurales) e InfluxDB (para datos de series temporales).

- *Recolección e integración de datos:*

  Telegraf y Mosquitto actúan como intermediarios para la recolección, transferencia y distribución eficiente de datos entre los diferentes servicios y bases de datos de la plataforma. Telegraf asegura la ingestión adecuada hacia InfluxDB, mientras que Mosquitto, como broker MQTT, facilita la comunicación desacoplada y la interoperabilidad de los distintos componentes del ecosistema OpenTwins mediante el intercambio de mensajes en tiempo real.

- *Visualización y control:*

  Grafana constituye el principal entorno de visualización y acceso a los datos, integrando plugins específicos para la gestión de gemelos digitales (OpenTwins app plugin) y para la visualización de modelos 3D generados en Unity dentro de paneles gráficos específicos (Unity panel plugin).

- *Modelado tridimensional:*

  Unity se utiliza para la creación y exportación de modelos 3D de los gemelos digitales, integrándose con Grafana a través del plugin correspondiente para ofrecer visualizaciones inmersivas y actualizadas del estado de los activos digitales.

Esta arquitectura modular y extensible proporciona soporte tanto para la interoperabilidad entre componentes como para el escalado y adaptación a distintas necesidades del dominio de gemelos digitales urbanos, facilitando la integración de nuevas fuentes de datos, capacidades analíticas y entornos de visualización avanzados.

La convergencia de las tecnologías de ciudades inteligentes, gemelos digitales y plataformas abiertas como OpenTwins crea un escenario propicio para el desarrollo de soluciones innovadoras en el ámbito del transporte urbano. La infraestructura de autobuses urbanos, como sistema crítico para la movilidad ciudadana, presenta características ideales para la implementación de gemelos digitales: genera grandes volúmenes de datos en tiempo real, que en el caso concreto de Málaga son proporcionados de manera abierta por el ayuntamiento [11], opera en entornos complejos con múltiples variables interconectadas, y requiere optimización continua para mejorar la eficiencia operativa y la experiencia del usuario.

En este contexto, el desarrollo de un gemelo digital de la infraestructura de autobuses de Málaga sobre la plataforma OpenTwins representa no solo una oportunidad tecnológica, sino también una contribución significativa al avance del estado del arte en sistemas de transporte urbano inteligente. Esta aproximación permite aprovechar las ventajas de una plataforma abierta y extensible, facilitando la innovación, la colaboración y la transferencia de conocimientos hacia otros contextos urbanos similares.

3

Análisis y Diseño

3.1. Problema y requisitos

Tal como se expuso en el capítulo anterior, existe una clara carencia de plataformas abiertas capaces de representar, monitorizar y analizar de manera integral la infraestructura del transporte urbano en las ciudades. Este proyecto surge precisamente para abordar esa necesidad: aprovechando la disponibilidad de datos abiertos en tiempo real, se plantea el desarrollo de un gemelo digital orientado a dotar a organismos públicos y ciudadanos de una herramienta versátil. El propósito es que este gemelo digital permita visualizar y analizar de forma eficiente la movilidad urbana en autobús, facilitando así la toma de decisiones fundamentadas y el conocimiento profundo sobre el funcionamiento y dinámica del sistema de transporte público de Málaga.

A continuación se exponen los requisitos funcionales que dan forma a la solución:

- El sistema debe contar con una interfaz principal con la que interactúe el usuario y que proporcione control de acceso, de manera que se requiera autenticación para acceder a las capacidades ofrecidas por la solución.

- La aplicación debe contar con mecanismos automáticos encargados de la recolección de datos de la API de datos abiertos del ayuntamiento de Málaga. La aplicación debe operar sin interacción alguna del usuario, manteniendo actualizada la plataforma con los últimos datos disponibles.

- El sistema debe contar con mecanismos que permitan controlar excepciones fuera del alcance de la propia plataforma, como lo es la falta puntual de disponibilidad de la API del ayuntamiento.

- El sistema debe proporcionar un menú que permita el acceso y la consulta del estado del gemelo digital.
- El sistema debe proporcionar un menú que dé acceso a todos los paneles de control disponibles.
- El sistema debe proporcionar un panel de control que informa de la situación general de los autobuses de la ciudad y debe ser capaz de filtrar los autobuses mostrados por línea.
- El sistema debe permitir que, al seleccionar un autobus, el panel de control que informa de la situación general redirija al usuario al panel de control concreto con la situación del autobús enlazado.
- El sistema debe proporcionar un panel de control que muestre la situación concreta de un autobús, adaptando la información representada según el identificador del autobús seleccionado.
- El sistema debe contar con un módulo que proporcione capacidades predictivas y ciertas métricas que ayuden a entender el comportamiento de la infraestructura de autobuses.
- El sistema debe permitir que el panel de control que muestra la situación concreta de un autobús sea capaz de mostrar la predicción realizada por el módulo de métricas y predicciones.
- El sistema debe almacenar los datos de las posiciones de los autobuses de forma persistente, manteniendo así una base de datos de series temporales que permita consultar datos históricos anteriores.

Además, a continuación se listan los requisitos no funcionales que definen cómo el sistema se debe comportar:

- La interfaz debe responder a las acciones del usuario en menos de 3 segundos bajo condiciones normales de carga.
- La actualización automática de datos desde la API debe realizarse al menos cada 5 minutos, siempre que sea posible.

- En caso de pérdida de conexión con la API externa, el sistema debe seguir intentando re-conectar con esta, sin dejar de funcionar.
- Los componentes deben de contar con un registro (*log*) para poder analizar y auditar el comportamiento del sistema.
- El sistema debe ser modular y admitir la incorporación de nuevos componentes sin modificación profunda de la arquitectura.
- El tiempo de despliegue de actualizaciones no debe exceder los 20 minutos.

3.2. Diseño conceptual

La solución completa desarrollada en este proyecto se concibe como un gemelo digital compuesto. Este abarca la representación digital agregada y organizada de todo el sistema físico de referencia, incorporando la dinámica de interacción entre sus distintos elementos. En este contexto, cada uno de los componentes individuales representados digitalmente con un equivalente en el mundo físico —como un autobús, una parada o cualquier otro subsistema— constituye un gemelo digital atómico [12, pág. 16]. Estos gemelos atómicos son unidades mínimas e independientes, alineadas con la idea de modularidad y descomposición funcional del sistema. Esta distinción metodológica permite reducir la complejidad del diseño, favoreciendo tanto la extensibilidad como el mantenimiento del gemelo digital compuesto, ya que cada entidad puede evolucionar, modificarse o integrarse de forma independiente en la estructura global, de acuerdo con las necesidades del sistema.

A continuación se presenta de manera abstracta y a alto nivel, mediante el diagrama de la figura 2, la arquitectura general sobre la que se fundamenta la solución propuesta. Este diseño guía y estructura el desarrollo del proyecto, estableciendo las bases para la implementación de un sistema modular e integrado.

Este trabajo se construye sobre OpenTwins, que se adapta perfectamente a la arquitectura deseada, proporcionando una base sólida y extensible para el desarrollo del sistema, aprovechando su arquitectura distribuida.

La arquitectura conceptual se distingue por su enfoque modular, en el que cada componente del sistema tiene responsabilidades definidas e interfaces claramente establecidas. Esta

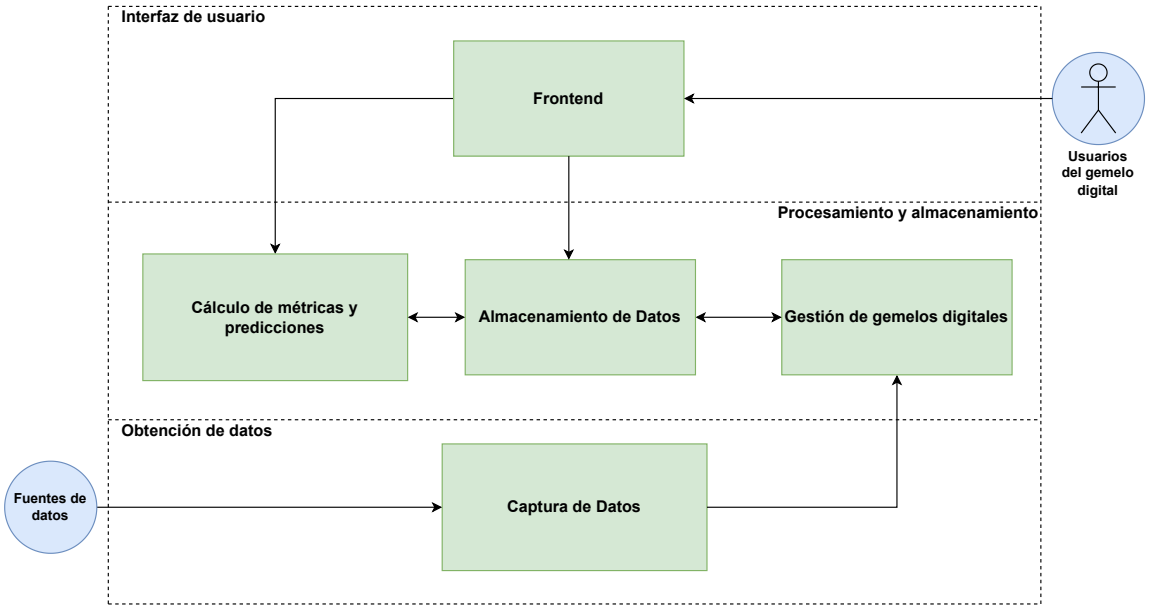


Figura 2: Diagrama conceptual de la solución.

arquitectura se organiza de manera visible en tres capas:

- La primera capa, que está más próxima a los datos en bruto proporcionados por la API del Ayuntamiento de Málaga, se encarga de **capturar y limpiar los datos**, preparándolos para su uso posterior.
- La segunda capa realiza la labor de **gestionar los gemelos digitales** modelados a partir de los datos obtenidos en la primera capa, **almacenando los datos** de estado, que sirven como base para el **cálculo de métricas y predicciones**, también correspondiente a esta capa.
- Por último, la tercera capa, que es la más cercana al usuario que interactuará con la plataforma, se encarga de **presentar una interfaz** con la que un usuario autenticado puede interactuar con el sistema, permitiéndole hacer uso de los paneles de control y demás capacidades de la solución.

Esta modularidad permite que el sistema evolucione de manera incremental, facilitando el mantenimiento, la extensión y la reutilización de componentes en diferentes contextos. El diseño incorpora principios de arquitectura abierta, promoviendo la interoperabilidad y evitando el *vendor lock-in* que frecuentemente limita las soluciones propietarias.

OpenTwins cubre la mayor parte de los componentes y funcionalidades necesarias para cumplir con los requerimientos de este diseño. Sin embargo, para abordar las particularidades y necesidades específicas del caso de uso presentado en este trabajo, es necesario complementar su arquitectura mediante la incorporación de componentes adicionales. Estos nuevos módulos permitirán adaptar y extender la solución base, asegurando que se satisfacen plenamente los objetivos planteados y facilitando una integración eficiente con los elementos ya proporcionados por OpenTwins. Cada uno de los componentes específicos, tanto los de OpenTwins como los desarrollados en el marco de este proyecto, serán tratados en el apartado dedicado a la arquitectura, del mismo capítulo.

La sinergia entre los componentes de OpenTwins y los desarrollos propios del proyecto crea un ecosistema tecnológico coherente que maximiza tanto la eficiencia del desarrollo como la calidad de la solución final. Esta aproximación híbrida permite aprovechar las ventajas de un framework abierto sobre el cual construir gemelos digitales, mientras se incorporan las funcionalidades específicas necesarias para abordar los requisitos particulares del sistema de transporte urbano de Málaga.

3.3. Tecnologías utilizadas

Tras explorar el diseño conceptual del trabajo desarrollado, es preciso abordar las diferentes tecnologías, herramientas, frameworks y librerías que hacen posible la existencia de la solución.

3.3.1. Docker

Docker es una plataforma de código abierto que permite desarrollar, empaquetar y ejecutar aplicaciones dentro de contenedores. Estos contenedores aíslan la aplicación y sus dependencias del sistema anfitrión, garantizando portabilidad, consistencia y eficiencia en el uso de recursos. Docker facilita la integración y entrega continua (CI/CD), permitiendo replicar entornos de desarrollo, pruebas y producción de forma sencilla y rápida [13].

Esta tecnología es esencial para el trabajo desarrollado, ya que permite empaquetar los componentes software implementados y garantizar que puedan ejecutarse posteriormente sin preocuparse por las dependencias. De este modo, se asegura la portabilidad y reproducibilidad

del entorno de ejecución. La elección de Docker responde tanto a la necesidad de desplegar la plataforma sobre Kubernetes, que requiere imágenes de contenedor compatibles, como a su fiabilidad demostrada y su condición de estándar ampliamente adoptado y documentado en la industria. Aunque existen otros motores de contenerización compatibles con Kubernetes, como Podman o containerd, se ha optado por Docker por las ventajas mencionadas.

3.3.2. Debian GNU/Linux

Debian es una distribución del sistema operativo GNU/Linux reconocida por su estabilidad, seguridad y gran repositorio de paquetes. Es ampliamente utilizada como base para servidores y entornos de desarrollo debido a su gestión eficiente de paquetes, soporte a largo plazo y filosofía de software libre. Su arquitectura modular y adaptabilidad la convierten en una opción idónea para infraestructuras críticas y virtualizadas [14] [15].

Debido a las virtudes nombradas, Debian se convierte en un sistema operativo idóneo para el desarrollo y despliegue de este trabajo, aunque cualquier distribución GNU/Linux compatible con K3s, la implementación de kubernetes elegida para este trabajo, podría usarse como sistema operativo host del nodo kubernetes.

3.3.3. Kubernetes

Kubernetes es una plataforma de orquestación de contenedores de código abierto diseñada para automatizar el despliegue, escalado y gestión de aplicaciones contenerizadas. Permite administrar clústeres de servidores, garantizando alta disponibilidad, escalabilidad y autorecuperación de los servicios desplegados. Kubernetes es ampliamente adoptado en entornos de microservicios y DevOps por su robustez y flexibilidad [16] [17].

El uso de kubernetes como plataforma de orquestación de contenedores sobre la que se sostiene este trabajo viene marcado por OpenTwins. Gracias a ello, el proyecto se beneficia de todos los puntos fuertes de kubernetes, y aunque la versión actual expuesta en el presente documento no contemple en el despliegue el uso de varios nodos, es posible adaptar OpenTwins y los componentes del desarrollo realizado para que lo permita.

3.3.3.1. Helm

Helm es el administrador de paquetes para Kubernetes. Permite definir, instalar y actualizar aplicaciones complejas mediante el uso de "charts" que son paquetes de recursos de Kubernetes fácilmente versionables, compartibles y reutilizables. Helm simplifica la gestión del ciclo de vida de las aplicaciones sobre Kubernetes y está respaldado oficialmente por la Cloud Native Computing Foundation (CNCF) [18].

Helm ha sido utilizado a lo largo de este trabajo para desplegar paquetes como Harbor u OpenTwins, en el caso de este último, también se ha llegado a descargar el paquete comprimido para poder modificarlo cuando ha sido necesario.

3.3.4. K3s

K3s es una distribución ligera de Kubernetes, certificada por la CNCF, optimizada para entornos con recursos limitados como edge computing, IoT y laboratorios de desarrollo. Está empaquetada en un único binario de tamaño reducido y simplifica la instalación y gestión de clústeres Kubernetes, manteniendo compatibilidad total con las APIs estándar. Su bajo consumo de recursos y facilidad de despliegue la hacen ideal para pruebas y pequeños entornos productivos [19].

Los populares entornos mínimos de kubernetes pensados para desarrollo como kind o minikube no satisfacían las necesidades de esta propuesta, debido a que prescindían de importantes características que sí que están presentes en una implementación completa de kubernetes. Gracias a su mínima huella de recursos, y su facilidad tanto de instalación como de actualización con el script bash que proporciona, K3s es una excelente elección para usar tanto en el desarrollo y pruebas del trabajo como en su posterior despliegue.

3.3.5. Harbor

Harbor es un registro de contenedores de código abierto diseñado para almacenar, administrar, firmar y escanear imágenes de contenedores y otros artefactos cloud-native como Helm charts. Se utiliza principalmente en entornos de Kubernetes y arquitectura cloud-native para garantizar la seguridad, trazabilidad y gestión centralizada de artefactos utilizados en el ciclo de vida de aplicaciones modernas [20].

En este proyecto, Harbor se utiliza como el registro de contenedores interno del clúster, lo que permite mantener la gestión de imágenes dentro de la propia infraestructura. Esta solución evita las restricciones impuestas por servicios públicos como Docker Hub, que suelen limitar tanto el número de imágenes almacenadas como la cantidad de descargas posibles en función de las cuotas gratuitas o de pago.

3.3.6. VMWare Workstation

VMWare Workstation es un hipervisor de tipo 2 que permite crear y gestionar máquinas virtuales en sistemas operativos Windows y Linux. Facilita la ejecución simultánea de múltiples sistemas operativos invitados, proporcionando entornos aislados para pruebas, desarrollo y simulación de redes. Incluye funcionalidades avanzadas como snapshots, clonación, configuración de redes virtuales y soporte para una amplia variedad de sistemas operativos invitados [21].

Para desarrollar este proyecto de manera cómoda y eficiente en un entorno de orquestación de contenedores como Kubernetes, ha resultado fundamental contar con un motor de virtualización como VMware Workstation. Esta herramienta permite una creación rápida de máquinas virtuales configuradas con los recursos necesarios, posibilitando la instalación del sistema operativo de preferencia de forma independiente al sistema operativo del host. La capacidad de disponer de múltiples clústeres de manera simultánea, tanto para el desarrollo como para la realización de pruebas, así como la funcionalidad de crear snapshots del estado de las máquinas virtuales para guardar y restaurar puntos específicos del proceso, han sido factores clave que han contribuido a un desarrollo fluido y controlado del proyecto.

3.3.7. Proxmox

Proxmox Virtual Environment (VE) es una plataforma de virtualización de código abierto que integra la virtualización basada en KVM (máquinas virtuales completas) y LXC (contenedores Linux) en una única interfaz de gestión web. Permite administrar clústeres, almacenamiento definido por software, redes y copias de seguridad de manera centralizada, siendo una solución robusta y flexible para la gestión de infraestructuras virtualizadas tanto a pequeña como a gran escala [22] [23].

De igual forma que contar con una buena herramienta de virtualización es algo muy de-

seable durante el desarrollo, también es muy positivo contar con estas virtudes durante un despliegue en producción, en este caso se ha utilizado un ordenador de bajo consumo ejecutándose de forma continuada, en el cual también se ha desplegado la máquina virtual *Debian* con *K3s* con el proyecto en sus distintas etapas del desarrollo, pero en este caso con el objetivo de probar su comportamiento en un despliegue real, en el cual nunca deja de ejecutarse. Proxmox VE ha sido el sistema operativo *host* instalado en esta máquina y de nuevo, la capacidad de realizar *backups* y *snapshots* es una contribución enorme de esta herramienta a la hora de proporcionar fiabilidad y agilidad. La interfaz web de Proxmox es una manera excelente de poder monitorizar el uso de recursos del trabajo desarrollado a lo largo del tiempo, y al ser un hipervisor, Proxmox ha permitido la ejecución paralela de otras cargas de trabajo virtualizadas de forma aislada, como por ejemplo un contenedor LXC con *Wireguard*.

3.3.8. WireGuard

WireGuard es un protocolo y software libre de VPN (Virtual Private Network) diseñado para establecer conexiones seguras y rápidas mediante criptografía de última generación. Su principal objetivo es proporcionar un sistema más sencillo, eficiente y seguro en comparación con protocolos tradicionales como IPsec y OpenVPN [24].

WireGuard se ha empleado para proporcionar un acceso seguro a las máquinas que ejecutan la solución desarrollada. La herramienta se despliega como un contenedor LXC en el servidor Proxmox, lo que permite establecer una VPN que actúa como punto de acceso protegido a la red privada en la que residen tanto la máquina de producción como la de desarrollo. De este modo, es posible conectar a estos entornos desde cualquier lugar sin necesidad de exponer sus servicios o recursos directamente a Internet, preservando la seguridad y privacidad del sistema.

3.3.9. Git

Git es un sistema de control de versiones distribuido y de código abierto, diseñado para gestionar y registrar los cambios realizados en el código fuente de un proyecto de software. Desarrollado por Linus Torvalds en 2005, Git permite que cada desarrollador disponga de una copia completa del repositorio, lo que facilita el trabajo independiente, la colaboración y la integración eficiente de cambios [25] [26].

Entre sus principales características destacan la gestión precisa del historial de cambios, la facilidad para crear y fusionar ramas, y la capacidad de restaurar versiones anteriores del código.

Git ha sido el sistema de control de versiones utilizado en los repositorios del proyecto debido a su predominancia indiscutible en el ámbito del desarrollo de software, siendo la herramienta más adoptada y reconocida a nivel mundial para la gestión eficiente de cambios en el código fuente.

3.3.10. Bash

Bash (Bourne Again Shell) es un intérprete de comandos y lenguaje de scripting ampliamente utilizado en sistemas Unix y GNU/Linux. Permite automatizar tareas administrativas, manipular archivos y ejecutar secuencias de comandos complejas, siendo fundamental para la gestión eficiente de sistemas y la integración de herramientas en entornos DevOps [27].

El lenguaje de scripting por excelencia en distribuciones Linux modernas, *Bash*, ha sido el utilizado en la mayoría de scripts del repositorio *emtrack-k8s*, que recopila todos aquellos scripts usados para desplegar el proyecto, haciendo uso de herramientas de línea de comandos como *kubectl* y *helm*. Su simplicidad, integración con GNU/Linux y el alto grado de familiaridad que el autor tiene con la herramienta han sido los motivos que hacen de *bash* el candidato ideal para esta función.

3.3.11. Python

Python es un lenguaje de programación interpretado, de alto nivel y propósito general, conocido por su sintaxis clara y legibilidad. Soporta múltiples paradigmas de programación y dispone de una extensa biblioteca estándar, lo que facilita el desarrollo rápido de aplicaciones, automatización, análisis de datos y scripting. Es uno de los lenguajes más populares en la actualidad por su versatilidad y comunidad activa [28].

Python se ha empleado para la realización de los distintos componentes software desarrollados para este proyecto debido a su efectividad en el prototipado rápido y el desarrollo eficiente de soluciones robustas. Su sintaxis sencilla, junto con una amplia variedad de bibliotecas y herramientas, facilita la implementación de funcionalidades complejas en menos tiempo. Además, su versatilidad y portabilidad permiten adaptar el código a diferentes entornos, y el

respaldo de una comunidad activa asegura acceso constante a recursos y soporte técnico.

Pese a que se han utilizado una variedad de herramientas y librerías en aquellos componentes software desarrollados en Python, mencionaremos a continuación los más destacables.

3.3.11.1. Poetry

Poetry es una herramienta moderna para la gestión de dependencias y el empaquetado en proyectos de Python. Su objetivo principal es simplificar y unificar el flujo de trabajo de desarrollo, permitiendo declarar, instalar y actualizar las dependencias de manera sencilla y reproducible, así como construir y publicar paquetes de Python desde una única interfaz.

A diferencia de herramientas tradicionales como pip, Poetry centraliza la gestión del proyecto en el archivo pyproject.toml, donde se definen tanto las dependencias como la configuración del paquete, siguiendo los estándares modernos de la comunidad Python. Además, genera un archivo poetry.lock que bloquea las versiones exactas de las dependencias, garantizando entornos consistentes en cualquier máquina donde se despliegue el proyecto. Poetry también automatiza la creación y gestión de entornos virtuales, evitando conflictos y facilitando la portabilidad del código. [29]

En este proyecto, Poetry ha sido fundamental para gestionar de manera eficiente las dependencias y asegurar la reproducibilidad del entorno de trabajo. Su enfoque declarativo facilita la instalación y actualización de paquetes, evitando conflictos y garantizando la compatibilidad entre versiones. Además, permite empaquetar y distribuir el proyecto de forma sencilla, agilizando el flujo de trabajo y mejorando la organización del código.

3.3.11.2. FastAPI

FastAPI es un framework web moderno para Python, especialmente enfocado en el desarrollo rápido y eficiente de APIs REST. Utiliza las anotaciones de tipo del propio lenguaje para validar peticiones y generar documentación interactiva automáticamente, lo que simplifica tanto la programación como el mantenimiento y la escalabilidad de los servicios web. Al estar basado en ASGI, permite el desarrollo de aplicaciones asíncronas y de alto rendimiento, resultando ideal para entornos donde se requieren servicios robustos, concurrentes y fácilmente integrables con herramientas y estándares actuales [30].

FastAPI es la base sobre la que se desarrolla EMTMetrics, un componente con funcionalidades predictivas que ofrece una API HTTP. Esta interfaz es consumida por Grafana mediante el plugin Infinity, permitiendo la consulta y visualización de los resultados generados por EMTMetrics de forma eficiente.

3.3.12. Postman

Postman es una plataforma ampliamente utilizada para el desarrollo, prueba y documentación de APIs. Permite a los desarrolladores crear y enviar solicitudes HTTP de manera sencilla, organizar colecciones de peticiones, automatizar pruebas y generar documentación dinámica y actualizada para APIs. Entre sus funcionalidades destacan la creación de entornos de prueba, la simulación de servidores (mock servers), la monitorización de APIs y la integración con flujos de CI/CD. Postman es especialmente valorado por su interfaz intuitiva y su capacidad para facilitar la colaboración entre equipos de desarrollo y calidad [31].

En este trabajo Postman se ha utilizado para poder realizar peticiones y depurar *endpoints*, en un entorno cómodo y eficiente, en el que poder guardar aquellas peticiones y configuraciones (variables, credenciales, etc.) que se vayan a usar con frecuencia.

3.3.13. Eclipse Ditto

Eclipse Ditto es un framework de código abierto orientado al desarrollo de gemelos digitales fácilmente integrable con aplicaciones IoT. Permite crear representaciones digitales de dispositivos físicos, facilitando la interacción, monitorización y gestión de dichos dispositivos a través de APIs estandarizadas (HTTP, WebSocket, MQTT, Kafka, entre otros). Eclipse Ditto proporciona control de acceso, almacenamiento de estado, gestión del ciclo de vida de los dispositivos y procesamiento de mensajes [32].

Eclipse Ditto es el elemento central de la arquitectura de OpenTwins, siendo el framework que permite la gestión de los gemelos digitales, y permitiendo una conexión con otras herramientas gracias a su amplio soporte de protocolos de comunicación.

3.3.14. MongoDB

MongoDB es un sistema de gestión de bases de datos NoSQL, de código abierto, orientado a documentos y diseñado para almacenar, consultar y procesar grandes volúmenes de datos de

manera flexible y escalable. A diferencia de las bases de datos relacionales tradicionales, donde los datos se organizan en tablas y filas, MongoDB utiliza documentos en formato BSON (una representación binaria de JSON), agrupados en colecciones, lo que permite almacenar datos estructurados, semiestructurados y no estructurados en una misma base de datos [33].

Eclipse Ditto utiliza MongoDB como su base de datos principal para almacenar el estado y los datos asociados a los gemelos digitales. Esta elección permite a Eclipse Ditto gestionar información compleja y estructurada de manera flexible y eficiente, aprovechando las capacidades de escalabilidad de MongoDB y su modelo orientado a documentos, lo que facilita la representación de los gemelos digitales y sus atributos.

3.3.15. Mosquitto/MQTT

Mosquitto es un broker de mensajería open source que implementa el protocolo MQTT (Message Queuing Telemetry Transport), ampliamente utilizado en aplicaciones de IoT por su eficiencia y bajo consumo de recursos. MQTT emplea un modelo de comunicación publish/-subscribe, ideal para la transmisión de mensajes entre dispositivos y servidores en entornos con ancho de banda limitado o alta latencia. Mosquitto destaca por su ligereza, facilidad de despliegue y soporte para múltiples versiones del protocolo, así como por sus utilidades de línea de comandos para pruebas y automatización de flujos de mensajes [34].

El servidor Mosquitto actúa como el bróker de mensajería principal para comunicarse con Eclipse Ditto, permitiendo por una parte alimentarlo con los datos publicados en distintos topics MQTT por los componentes software desarrollados en Python, que se han bautizado como “EMTScraper”, y, por otro lado, la comunicación con InfluxDB a través de Telegraf.

3.3.16. Telegraf

Telegraf es un agente open source para la recopilación de métricas, logs y datos de series temporales. Es altamente extensible gracias a su arquitectura basada en plugins, permitiendo la integración con más de 300 fuentes de datos, incluidos sistemas, aplicaciones, sensores IoT y plataformas cloud. Telegraf puede transformar y enviar los datos recopilados a sistemas de almacenamiento como InfluxDB, soportando diversos formatos y protocolos. Su diseño ligero y sin dependencias externas facilita su despliegue en entornos heterogéneos y su uso en monitorización, observabilidad y análisis de infraestructuras [35].

En el presente proyecto, se ha utilizado Telegraf para capturar los eventos generados por Eclipse Ditto que son publicados en Mosquitto, aprovechando su plugin de entrada específico. Posteriormente, estos datos son transmitidos a InfluxDB mediante el plugin de salida de Telegraf, estableciendo así una integración eficiente entre Eclipse Ditto e InfluxDB que posibilita la persistencia y gestión continua de los datos.

3.3.17. InfluxDB

InfluxDB es una base de datos orientada a series temporales, diseñada específicamente para almacenar, consultar y analizar grandes volúmenes de datos indexados por tiempo, como métricas, logs y eventos de sensores. Su modelo de datos y su lenguaje de consulta (Flux) están optimizados para operaciones de agregación, filtrado y análisis en tiempo real. InfluxDB se utiliza ampliamente en monitorización de infraestructuras, IoT y análisis de datos científicos por su alto rendimiento y escalabilidad [36] [37].

Gracias a InfluxDB se supera la limitación de Eclipse Ditto respecto al almacenamiento histórico del estado de los gemelos digitales, ya que posibilita la persistencia de series temporales de estos estados. De este modo, es posible consultar y analizar la evolución de los estados a lo largo del tiempo de manera precisa, utilizando consultas en el lenguaje Flux según los requerimientos específicos del proyecto.

3.3.18. MySQL

MySQL es un sistema de gestión de bases de datos relacional (RDBMS) de código abierto ampliamente utilizado para almacenar, organizar y gestionar datos estructurados en tablas. Utiliza el lenguaje SQL para realizar consultas y manipulación de datos, y destaca por su alto rendimiento, escalabilidad, facilidad de uso y compatibilidad con múltiples plataformas y lenguajes de programación. Gracias a su robustez, soporte comunitario y flexibilidad, MySQL es una opción preferente tanto para aplicaciones web como para sistemas empresariales de gran escala [38].

Inicialmente, el uso de MySQL no estaba previsto en este proyecto. Sin embargo, al incrementarse considerablemente la cantidad de datos almacenados en InfluxDB, surgieron problemas de escalabilidad [39] debido a un aumento significativo de la cardinalidad, lo que afectó negativamente al rendimiento del sistema. Ante esta situación, se evidenció la necesidad de

incorporar un sistema gestor de bases de datos eficiente y escalable, capaz de manejar grandes volúmenes de datos estáticos. Por ello, se optó por MySQL para satisfacer estos requerimientos.

3.3.19. Grafana

Grafana es una plataforma open source para la visualización y análisis de datos, especialmente diseñada para trabajar con bases de datos de series temporales como InfluxDB. Permite crear paneles de control interactivos y personalizables que facilitan la interpretación y el monitoreo de métricas en tiempo real. Grafana soporta alertas, exploración dinámica de datos, integración con múltiples fuentes y una amplia variedad de paneles gráficos, siendo una herramienta clave para la observabilidad y la toma de decisiones basada en datos en entornos tecnológicos e IoT [40].

En este trabajo, Grafana se emplea como frontend para la visualización de información relevante que facilita la toma de decisiones informadas. Mediante paneles de control organizados en paneles, se presentan diversas visualizaciones adaptadas al tipo de datos a representar, obtenidos a través de consultas a las fuentes de datos InfluxDB y MySQL.

3.3.20. Visual Studio Code

Visual Studio Code (VS Code) es un editor de código fuente multiplataforma desarrollado por Microsoft, disponible para Windows, Linux, macOS y navegadores web. Ofrece soporte integrado para depuración, control de versiones con Git, resaltado de sintaxis, autocompletado inteligente, fragmentos de código y refactorización. VS Code es altamente personalizable mediante extensiones, permitiendo añadir soporte para nuevos lenguajes, temas, herramientas de depuración y funcionalidades adicionales. Su arquitectura ligera y su integración con herramientas modernas de desarrollo lo han convertido en uno de los editores más populares entre la comunidad de desarrolladores [41].

Visual Studio Code ha sido el editor de texto utilizado para el desarrollo del código fuente para el despliegue del proyecto, incluido en el repositorio “emtrack-k8”.

3.3.21. PyCharm

PyCharm es un entorno de desarrollo integrado (IDE) desarrollado por JetBrains, especializado en el desarrollo de aplicaciones Python. Soporta tanto Python 2.7 como las versiones más recientes de Python 3, y ofrece herramientas avanzadas como autocompletado inteligente, inspección de código, refactorización, depuración, testing y soporte para entornos virtuales y remotos. PyCharm incorpora funcionalidades específicas para ciencia de datos, como soporte para Jupyter Notebooks, así como integración con sistemas de control de versiones. La documentación oficial detalla la configuración de intérpretes, la creación de proyectos y el uso de sus principales herramientas para el desarrollo eficiente en Python [42].

Para el desarrollo de los componentes software escritos en Python de este proyecto se ha utilizado PyCharm en lugar de Visual Studio Code, debido a las ventajas que ofrece este IDE para proyectos Python, como sus funcionalidades avanzadas integradas, herramientas de depuración, refactorización y soporte especializado para frameworks y bibliotecas del ecosistema Python.

3.3.22. DBeaver

DBeaver es una herramienta cliente SQL y de administración universal de bases de datos, multiplataforma y de código abierto, que permite conectarse y gestionar una amplia variedad de sistemas de gestión de bases de datos tanto relacionales como NoSQL. Ofrece funciones avanzadas como ejecución de consultas SQL, edición de datos y metadatos, generación de diagramas entidad-relación (ERD), autocompletado de código y soporte para plugins [43].

DBeaver ha sido usado para explorar y manejar los datos de la base de datos MySQL añadida en el propio proyecto mediante una interfaz gráfica potente e intuitiva.

3.4. Arquitectura

Este apartado profundiza en la arquitectura de la solución propuesta. El gemelo digital se apoya en la plataforma OpenTwins, desplegada sobre un clúster de Kubernetes y cimentada en componentes de software libre que se comunican mediante diversos protocolos estándares. Sobre esta base, el Trabajo de Fin de Grado incorpora módulos adicionales para satisfacer los requisitos específicos del caso de uso planteado. A continuación, se presenta un diagrama

detallado que ilustra la arquitectura resultante, distinguiendo visualmente aquellos elementos nativos de OpenTwins de los desarrollados en esta propuesta.

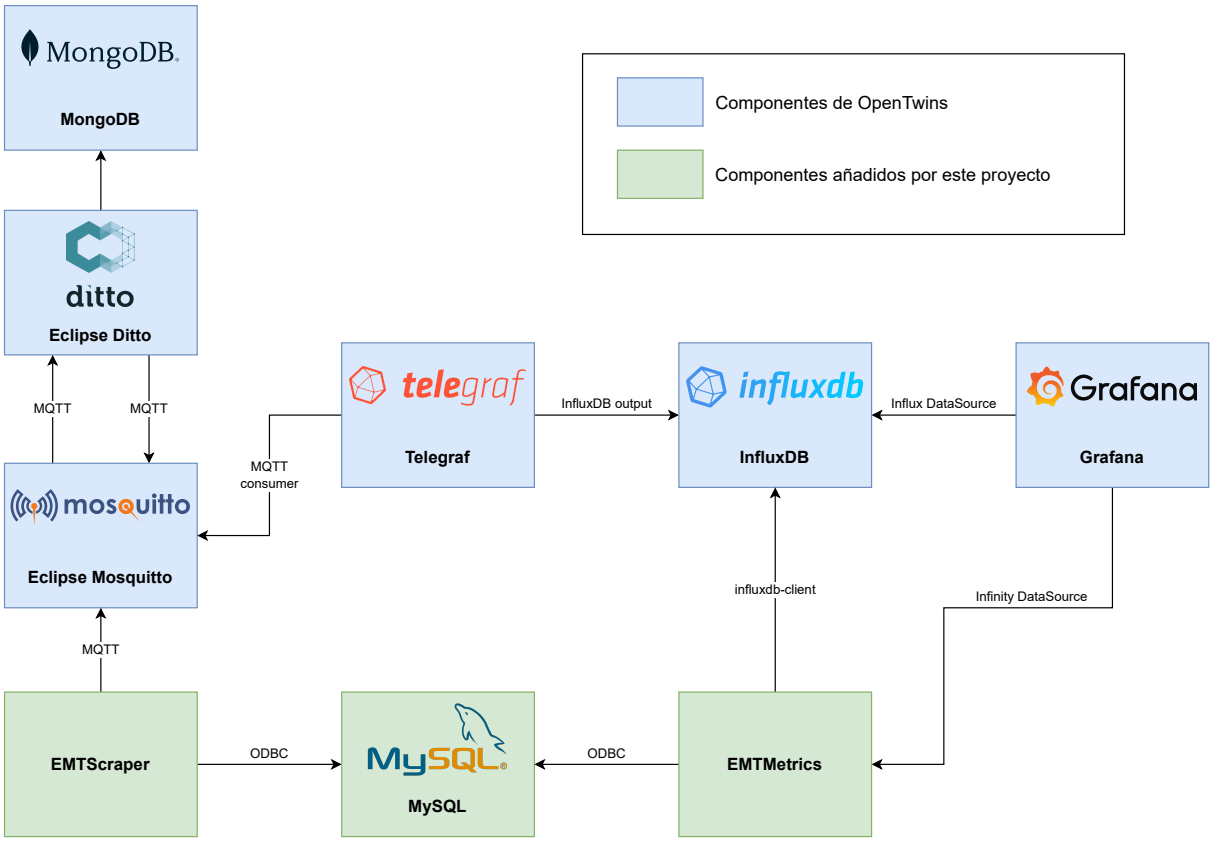


Figura 3: Arquitectura concreta de la solución.

Si hacemos una comparación entre las figuras 2 y 3, podremos identificar qué componentes forman parte de cada una de las categorías o elementos de alto nivel:

- Captura de datos:

  La solución incorpora un conjunto de componentes denominados **EMTScraper**, cuya función principal es extraer información de la API del Ayuntamiento de Málaga. Cada componente está especializado en la recolección y procesamiento de diferentes tipos de datos relevantes para el gemelo digital. Gracias a EMTScraper, la plataforma recibe de manera continua y automatizada los datos necesarios para alimentar y actualizar en tiempo real la representación digital de la infraestructura de transporte urbano.

- Gestión de gemelos digitales:

  Su componente principal es **Eclipse Ditto**, que permite gestionar los gemelos digitales

atómicos, permitiendo múltiples posibilidades de importación y exportación de datos gracias a su amplio soporte de protocolos de comunicación.

- Almacenamiento de datos:

  Aquí entran componentes como **MongoDB**, usado por Eclipse Ditto para la persistencia de los datos de los gemelos digitales que maneja, **InfluxDB**, que suple la carencia de Eclipse Ditto de no almacenar un histórico de los datos de estado de los gemelos digitales, y por último **MySQL**, software añadido en este proyecto debido a que hay datos estáticos que no forman parte de los gemelos digitales y que almacenarlos en InfluxDB como series temporales no resulta adecuado (ver capítulo 5 para más información), por lo que se requería una base de datos tradicional dónde almacenarlos de manera eficiente.

- Cálculo de métricas y predicciones:

  El componente encargado de proporcionar esta funcionalidad es **EMTMetrics**. Este módulo calcula métricas que facilitan la predicción de posiciones futuras y tiempos estimados de llegada, y expone diversos endpoints a través de su API REST para que puedan ser consumidos directamente por Grafana.

- Frontend:

  Grafana constituye la interfaz principal a través de la cual los usuarios interactúan con el gemelo digital. Proporciona control de acceso y paneles de control interactivos que presentan la información de una manera cómoda e intuitiva. Grafana facilita la presentación de datos relevantes de manera clara y visual, permitiendo al usuario explorar métricas clave, analizar tendencias en tiempo real y adaptar las vistas según sus necesidades. Todo esto posiciona a Grafana como una herramienta central para el monitoreo y análisis de la infraestructura urbana modelada.

- Conectores de datos:

  Aunque estos componentes no tienen un propósito funcional específico por sí mismos, resultan fundamentales para la solución global, ya que aportan cohesión a la plataforma y permiten el intercambio eficiente de información entre sus distintas partes. Gracias a su integración, los diferentes módulos pueden mantenerse desacoplados y modulares,

funcionando como una unidad coordinada. En este grupo se encuentra **Eclipse Mosquitto**, un bróker de mensajería que implementa el protocolo MQTT y facilita la transferencia de mensajes desde EMTScraper hacia Eclipse Ditto. Asimismo, se incluye **Telegraf**, el cual consume los eventos generados por Eclipse Ditto y los traslada a InfluxDB, perpetuando los datos de estado de los gemelos digitales en series temporales de modo que puedan ser consultados a posteriori.

4

Implementación

El presente capítulo profundiza en los detalles correspondientes a la implementación y los aspectos técnicos del sistema desarrollado. A continuación se describen con detalle los componentes implementados en este trabajo.

4.1. EMTScraper

Se trata del conjunto de componentes software que capturan los datos de la API abierta del ayuntamiento de Málaga, haciendo peticiones a esta de forma periódica para introducirlos en el sistema.

El sistema está conformado por dos programas desarrollados en Python y un script en Bash. Los componentes Python, cuyo código fuente se encuentra disponible en un repositorio de GitHub[3], están organizados bajo la estructura de un pequeño *monorepo*, es decir, un repositorio que contiene más de un proyecto lógico [44]. Dicho repositorio alberga dos proyectos principales en el directorio `projects/`, correspondiendo a los componentes previamente mencionados, denominados *BusLocation* y *Routes*. La decisión de implementar una estructura de monorepo, en lugar de utilizar repositorios individuales para cada componente, obedece a la necesidad de compartir librerías comunes (almacenadas en el directorio `libs/`), lo que facilita la reutilización y una mejor organización del código. Asimismo, se mantiene la gestión de dependencias mediante el uso de *Poetry*, como se expuso en el apartado relativo a las tecnologías empleadas.

4.1.1. EMTScraper BusLocation

EMTScraper BusLocation es un software escrito en Python cuya función es capturar e integrar en el sistema los datos expuestos por el endpoint relativos a las lineas y ubicaciones de

[3] https://github.com/isrojas1/EMTScraper

los autobuses[4], que devuelve un GeoJSON con las ubicaciones en tiempo real de cada uno de los autobuses físicos que recorren la ciudad en el momento de la consulta. Este componente opera en un *while-loop*, que se ejecuta de forma continua, tal y como se muestra en el listado 1. En cada iteración, ejecutadas con una diferencia de un minuto entre sí (la página de información de estos datos menciona que la frecuencia de actualización es de un minuto, aunque en la práctica es de cinco), se realiza una petición al endpoint; si se detecta que la respuesta es la primera recibida o difiere de la última versión cacheada en memoria, se procede a realizar la actualización de los datos. Para ello, primero se comprueba para cada autobús si este ya existe en el sistema, mediante una petición GET a la API HTTP de Eclipse Ditto en el endpoint `/api/2/things/{thing_id}`. En caso de no existir, que sería informado por el endpoint al devolver el código 404, se hace una petición de nuevo a la API HTTP de Eclipse Ditto, esta vez un PUT en `/api/2/things/{thing_id}`. Para más información acerca de la API HTTP de Eclipse Ditto el lector puede referirse a su documentación oficial[5]. Tras haber asegurado la existencia del gemelo digital del autobús, se publica un mensaje MQTT en el servidor Eclipse Mosquitto, empleando el topic `malaga/bus/{id_bus}`, este sigue la especificación del Protocolo Ditto[6], utilizando el comando *merge* y un JSON Merge Patch[7] para modificar el gemelo digital deseado. Un ejemplo de este mensaje, puede consultarse en el listado 2. Eclipse Ditto se suscribe a estos topics para actualizar los correspondientes gemelos digitales de los autobuses. Para posibilitar esta suscripción y la recepción de la información enviada por BusLocation, ha sido necesario añadir una nueva conexión con dirección `malaga/#`, lo cual puede realizarse tanto mediante la API de Ditto como a través del menú de OpenTwins en Grafana. Este proyecto persigue que todas estas configuraciones se realicen mediante API, permitiendo así la reproducción automatizada de entornos con la mínima interacción del usuario que sea posible. Por último, tras integrar satisfactoriamente los datos actualizados en el gemelo digital correspondiente, Eclipse Ditto publica un mensaje de evento confirmando la operación. Se puede ver un ejemplo en el listado 3. Gracias a EMTScraper BusLocation, el sistema cuenta con la información de los autobuses actualizada en tiempo real, según se va actualizando la API que

[4] https://datosabiertos.malaga.eu/recursos/transporte/EMT/EMTlineasUbicaciones/lineasyubicaciones.geojson

[5] https://eclipse.dev/ditto/http-api-doc.html

[6] https://eclipse.dev/ditto/3.3/protocol-specification.html

[7] https://datatracker.ietf.org/doc/html/rfc7396

consulta. La figura 4 muestra un ejemplo de un bus creado y con datos actualizados por este componente siendo mostrado en la interfaz de OpenTwins.

En relación con el despliegue, se construye una imagen Docker a partir de un Dockerfile, en la que se establece como punto de entrada (*entrypoint*) el código del programa, interpretado por Python. Dicha imagen se emplea como base para el contenedor que forma parte del pod especificado en el manifiesto del *deployment* de Kubernetes. Este manifiesto se encuentra disponible en el repositorio dedicado al despliegue.

```python
cached_buses_data = None
skip_publish_counter = 0
bus_twin_manager = BusTwinManager(ditto_base_url, (ditto_username, ditto_password))
mqtt_client = MQTTClient(mosquitto_host, mosquitto_port)

while True:
    buses_data = fetch_buses_data()

    if cached_buses_data is None or cached_buses_data != buses_data:
        skip_publish_counter = 0
        logger.info("Publishing bus data.")
        opentwins_publish_buses_data(
            buses_data,
            bus_twin_manager,
            mqtt_client,
        )
        cached_buses_data = buses_data
    else:
        skip_publish_counter += 1
        logger.info(
            "Fetched data is the same to cached one. Skipping bus data publish."
        )
        if skip_publish_counter > 4:
            logger.warning(
                f"Bus data publish was skipped more than {skip_publish_counter} times."
            )
    time.sleep(60)  # Wait for 1 minute before the next request
```

Listing 1: Código principal ejecutado por EMTScraper BusLocation

```
{
  "topic": "buses/648/things/twin/commands/merge",
  "headers": {
```

```
    "content-type": "application/merge-patch+json"
  },
  "path": "/features",
  "value": {
    "gps": {
      "properties": {
        "latitude": "36.729504",
        "longitude": "-4.4147377",
        "time": "2025-07-22 17:46:24"
      }
    },
    "line": {
      "properties": {
        "code": "1.0",
        "direction": "1"
      }
    }
  }
}
```


Listing 2: Mensaje MQTT en el topic malaga/bus/648 enviado por EMTScraper BusLocation


```
{
  "topic": "buses/648/things/twin/events/merged",
  "headers": {
    "ditto-originator": "nginx:ditto",
    "response-required": false,
    "version": 2,
    "requested-acks": [],
    "content-type": "application/merge-patch+json"
  },
  "path": "/features",
  "value": {
    "gps": {
      "properties": {
        "latitude": "36.729504",
        "longitude": "-4.4147377",
        "time": "2025-07-22 17:46:24"
      }
    },
    "line": {
      "properties": {
        "code": "1.0",
        "direction": "1"
```

```
      }
    }
  },
  "extra": {
    "thingId": "buses:648"
  },
  "revision": 28679,
  "timestamp": "2025-07-22T15:46:06.088966967Z"
}
```


Listing 3: Mensaje MQTT en el topic opentwins/twin/events/buses/648 generado por Eclipse Ditto

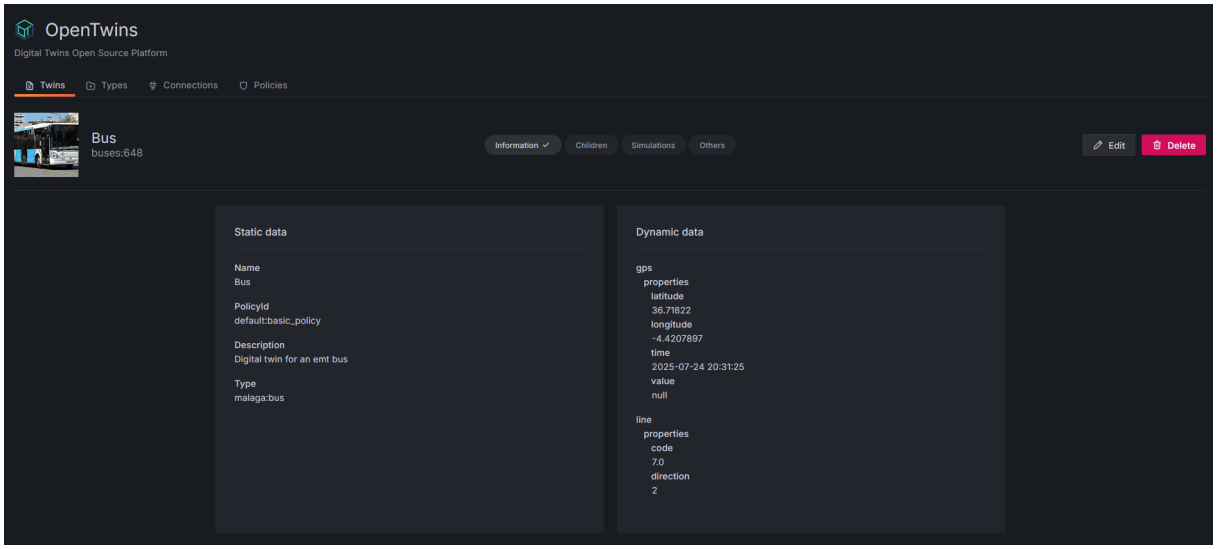


Figura 4: Bus 648 en la interfaz de OpenTwins en Grafana.

4.1.2. EMTScraper Routes

Respecto al otro componente Python de este grupo, su función es capturar e integrar en el sistema los datos que devuelve el endpoint relativo a lineas y paradas[8], el cual entrega cada una de las líneas de autobús de la EMT, junto con una lista de las paradas pertenecientes a esa línea concreta, indicando a qué sentido de la línea pertenecen, así como su orden en el recorrido. El software se encarga de la creación de los gemelos digitales de las líneas y las paradas, así como de la actualización de sus datos.

[8] https://datosabiertos.malaga.eu/recursos/transporte/EMT/EMTLineasYParadas/lineasyparadas.geojson

La frecuencia de actualización de estos datos es diaria, por lo que no requiere una ejecución constante como es el caso del componente mencionado en el apartado anterior. Por ello, en lugar de operar en un *while-loop*, este programa está pensado para realizar la actualización de los datos y finalizar su ejecución. En cuanto a su operación, este componente funciona de una manera muy similar a EMTScraper BusLocation, usa el GET `/api/2/things/{thing_id}` de la API de Eclipse Ditto para comprobar la existencia de los gemelos, creándolos mediante el método PUT, en la misma ruta, en caso de que no existiesen previamente. El listado 4 muestra el código para crear una línea. Para la actualización de los datos, de nuevo se emplea el uso del Protocolo Ditto a través de mensajes MQTT enviados al servidor Mosquitto, los topics en los que se envían los mensajes son `malaga/line/{id_linea}` y `malaga/stop/{id_parada}`, para las líneas y las paradas respectivamente.

Como se ha mencionado, la ejecución del programa no es indefinida, si no que arranca, actualiza una vez la información y finaliza su ejecución. En este caso la manera de controlar la ejecución periódica del refresco de la información de estos gemelos digitales es mediante el tipo de despliegue elegido en Kubernetes, el cual es un CronJob, en lugar de un Deployment. Un CronJob está pensado para ejecutar una tarea de forma regular. Ejecuta un Job periódicamente según un horario establecido, que se define en formato Cron. El manifiesto yaml de este CronJob se muestra en el listado 5, como se puede observar tanto en este como en todos los despliegues se hace uso de variables de entorno para garantizar un despliegue desacoplado del código del propio programa y una protección de los valores sensibles como pueden ser credenciales.

```
def create_line(
    self,
    line_id: str,
) -> Dict[str, Any]:
    """
    Create a line twin's in Ditto.

    :param line_id: The ID of the line twin
    :return: JSON response of the creation result
    """
    default_payload = {
        "thingId": f"lines:{line_id}",
        "policyId": "default:basic_policy",
```

```
        "attributes": {
            "name": "Line",
            "description": "Digital twin for an emt line",
            "image":
                ↪  "https://estaticos-cdn.prensaiberica.es/clip/a27f9338-233f-4bad-97b2-"
            "bf3808eda65c_16-9-aspect-ratio_default_0.jpg",
            "type": "malaga:line",
        },
        "features": {},
    }
    self.update_line(line_id, default_payload)

def update_line(self, line_id: str, update_payload: dict):
    """
    Update a line twin's data in Ditto.

    :param line_id: The ID of the line twin
    :param update_payload: Dictionary containing updated line data
    :return: JSON response of the update result
    """
    url = f"{self.ditto_url}/api/2/things/lines:{line_id}"
    response = requests.put(url, json=update_payload, auth=self.auth)
    response.raise_for_status()
```

Listing 4: Métodos de la clase LineTwinManager para crear y actualizar el gemelo digital de una línea de bus vía API

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: emtscraper-routes
  namespace: scraping
spec:
  schedule: "0 3 * * *"
  successfulJobsHistoryLimit: 3
  failedJobsHistoryLimit: 1
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: emtscraper-routes
            image: harbor.local/library/routes:0.2.3
            env:
```

```
            - name: MOSQUITTO_HOST
              value: "$MOSQUITTO_HOST"
            - name: MOSQUITTO_PORT
              value: "$MOSQUITTO_PORT"
            - name: DITTO_BASE_URL
              value: "$DITTO_BASE_URL"
            - name: DITTO_USERNAME
              value: "$DITTO_USERNAME"
            - name: DITTO_PASSWORD
              value: "$DITTO_PASSWORD"
            imagePullPolicy: Always
          restartPolicy: OnFailure
```

Listing 5: Manifiesto YAML del CronJob de EMTScraper Routes

4.1.3. MySQL CSV Loader

Último componente de EMTScraper, esta vez un CronJob de ejecución diaria, encargado de cargar datos estáticos en MySQL. Inicialmente, se pensó en desarrollar este componente en Python, al igual que sus homólogos BusLocation y Routes, sin embargo, la facilidad de desarrollo y versatilidad que aporta el realizar un script en bash, sumado al rendimiento aportado por la opción `LOAD DATA LOCAL INFILE` del cliente MySQL de línea de comandos, hicieron de esta la opción ideal para este componente. Los datos cargados por este software son los correspondientes a los trayectos recorridos por los autobuses, conteniendo los puntos que trazan el recorrido de cada una de las rutas. Estas siluetas contienen muchas curvas, por lo que el número de puntos necesario para definirlas es elevado. Aquí entran en juego los problemas de escalabilidad tanto de Eclipse Ditto como de InfluxDB, los cuales no son adecuados para albergar este tipo de datos, ya que el hacerlo supondría crear una serie temporal por cada uno de los puntos existentes, lo cual elevaría de forma significativa la cardinalidad de InfluxDB, haciendo que consultas simples que antes tomaban pocos milisegundos eleven su orden de magnitud hasta más de 30 segundos [45], incumpliendo el requisito no funcional del rendimiento del sistema y haciéndolo prácticamente inutilizable para su uso fluido en una interfaz de usuario.

Los datos importados son proporcionados por el endpoint relativo a lineas y horarios[9], ac-

[9] https://datosabiertos.malaga.eu/recursos/transporte/EMT/lineasYHorarios/google_transit.

tualizado cada día y que proporciona un archivo comprimido *zip* el cual contiene múltiples ficheros en formato *csv* con diferente información relativa a rutas y horarios. Los datos de interés que importa este script son los contenidos en los ficheros `trips.csv` y `shapes.csv`. El primero contiene todas las posibles rutas que puede tomar un autobús, organizadas por línea y sentido, así como el id de su silueta (shape) correspondiente. El segundo contiene las siluetas propiamente, cada fila representa un punto de esta y contiene los siguientes campos:

- **shape_id:** Id de la silueta
- **shape_pt_lat:** Latitud del punto
- **shape_pt_lon:** Longitud del punto
- **shape_pt_sequence:** Orden del punto en la ruta
- **shape_dist_traveled:** Distancia recorrida desde el inicio de la ruta hasta el punto

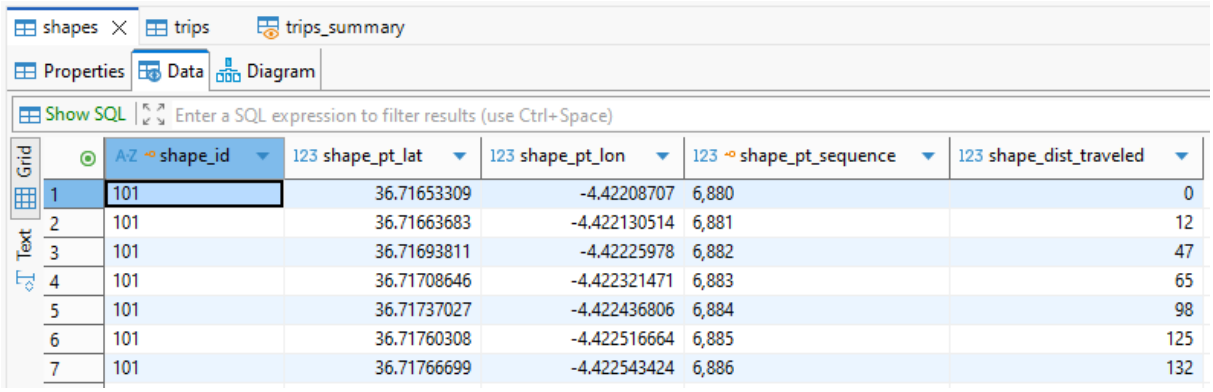
Filtrando los datos de la tabla trips, es posible obtener de manera unívoca para cada línea y sentido el id de su silueta. Para facilitar la obtención de este id, se crea la vista *trips_summary*, el cual solo tiene dichos tres campos.
El código del script que ejecuta este componente se muestra en el listado 6
Las figuras 5, 6 y 7 muestran datos de las tablas *shapes* y *trips*, y de la vista *trips_summary* respectivamente.

```
# Descargar y descomprimir archivos en /tmp
wget -O /tmp/google_transit.zip https://datosabiertos.malaga.eu/recursos/transporte/EMT/
        lineasYHorarios/google_transit.zip
unzip -o /tmp/google_transit.zip -d /tmp

# Crear base de datos emtdata
mysql --host=$(MYSQL_HOST) --user=$(DB_USER) --password=$(DB_PASSWORD) --skip-ssl \
-e "CREATE DATABASE IF NOT EXISTS emtdata;"

# Resetear tablas y cargar datos en una transacción
mysql --host=$(MYSQL_HOST) --user=$(DB_USER) --password=$(DB_PASSWORD)
    ↪  --database=$(DB_NAME) --local-infile=1 --skip-ssl \
-e "START TRANSACTION;
    DROP TABLE IF EXISTS trips, shapes;
```

zip

```
    CREATE TABLE trips (
      route_id INT,
      service_id VARCHAR(255),
      trip_id VARCHAR(255) NOT NULL,
      trip_headsign VARCHAR(255),
      direction_id INT,
      block_id VARCHAR(255),
      shape_id INT,
      PRIMARY KEY (trip_id)
    );
    CREATE TABLE shapes (
      shape_id VARCHAR(5) NOT NULL,
      shape_pt_lat DECIMAL(10,8),
      shape_pt_lon DECIMAL(11,9),
      shape_pt_sequence INT NOT NULL,
      shape_dist_traveled INT,
      PRIMARY KEY (shape_id, shape_pt_sequence)
    );
    LOAD DATA LOCAL INFILE '/tmp/trips.csv' INTO TABLE trips
      FIELDS TERMINATED BY ','
      ENCLOSED BY '\"'
      LINES TERMINATED BY '\n'
      IGNORE 1 LINES
      (route_id, service_id, trip_id, trip_headsign, direction_id, block_id, shape_id);
    LOAD DATA LOCAL INFILE '/tmp/shapes.csv' INTO TABLE shapes
      FIELDS TERMINATED BY ','
      ENCLOSED BY '\"'
      LINES TERMINATED BY '\n'
      IGNORE 1 LINES
      (shape_id, shape_pt_lat, shape_pt_lon, shape_pt_sequence, shape_dist_traveled);
    COMMIT;"

# Refrescar la vista (fuera de la transacción)
mysql --host=$(MYSQL_HOST) --user=$(DB_USER) --password=$(DB_PASSWORD)
    ↪  --database=$(DB_NAME) --skip-ssl \
-e "CREATE OR REPLACE VIEW trips_summary AS
      SELECT DISTINCT
        route_id,
        direction_id + 1 AS direction_id,
        shape_id
      FROM trips
      ORDER BY route_id ASC;"
```

Listing 6: Script ejecutado por MySQL CSV Loader

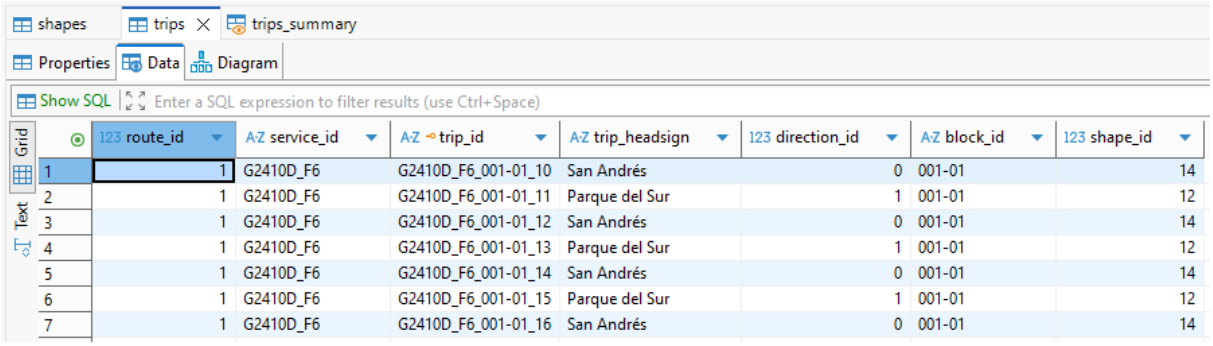

shapes | trips | trips_summary

Properties | Data | Diagram

Show SQL | Enter a SQL expression to filter results (use Ctrl+Space)

	shape_id	shape_pt_lat	shape_pt_lon	shape_pt_sequence	shape_dist_traveled
1	101	36.71653309	-4.42208707	6,880	0
2	101	36.71663683	-4.422130514	6,881	12
3	101	36.71693811	-4.42225978	6,882	47
4	101	36.71708646	-4.422321471	6,883	65
5	101	36.71737027	-4.422436806	6,884	98
6	101	36.71760308	-4.422516664	6,885	125
7	101	36.71766699	-4.422543424	6,886	132

Figura 5: Tabla *shapes* en MySQL.


shapes | trips | trips_summary

Properties | Data | Diagram

Show SQL | Enter a SQL expression to filter results (use Ctrl+Space)

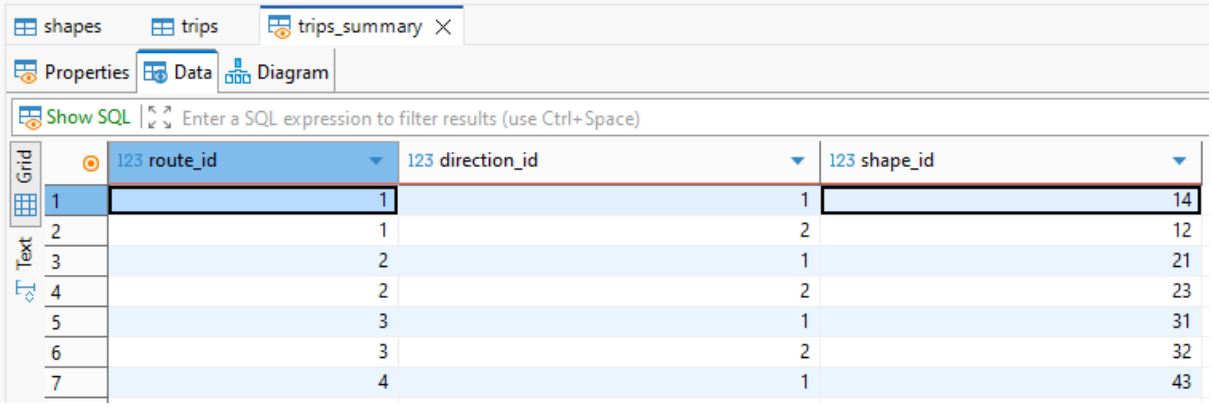
	route_id	service_id	trip_id	trip_headsign	direction_id	block_id	shape_id
1	1	G2410D_F6	G2410D_F6_001-01_10	San Andrés	0	001-01	14
2	1	G2410D_F6	G2410D_F6_001-01_11	Parque del Sur	1	001-01	12
3	1	G2410D_F6	G2410D_F6_001-01_12	San Andrés	0	001-01	14
4	1	G2410D_F6	G2410D_F6_001-01_13	Parque del Sur	1	001-01	12
5	1	G2410D_F6	G2410D_F6_001-01_14	San Andrés	0	001-01	14
6	1	G2410D_F6	G2410D_F6_001-01_15	Parque del Sur	1	001-01	12
7	1	G2410D_F6	G2410D_F6_001-01_16	San Andrés	0	001-01	14

Figura 6: Tabla *trips* en MySQL.

4.2. MySQL

Como se mencionó al describir MySQL CSV Loader, resultaba imprescindible contar con una base de datos relacional que ofreciera altos niveles de escalabilidad para el almacenamiento de datos estáticos del sistema. En este contexto, se optó por MySQL debido a su amplia adopción en la industria, además de su facilidad de integración y despliegue en un entorno Kubernetes, permitiendo la provisión automatizada y escalable de instancias de base de datos mediante recursos nativos como *StatefulSets* y *PersistentVolumes*.

La elección de despliegue consiste en un *StatefulSet* de una réplica, siendo en este caso más adecuado que un Deployment debido a que MySQL es un servicio con estado, que necesita contar con almacenamiento persistente para salvaguardar los datos. El manifiesto YAML de este *StatefulSet* puede ser consultado en el listado 7. El *StatefulSet* de MySQL definido en el manifiesto utiliza almacenamiento persistente mediante la inclusión de la sección volumeClaimTemplates. Esta sección permite que Kubernetes cree automáticamente un *PersistentVo-*



	123 route_id	123 direction_id	123 shape_id
1	1	1	14
2	1	2	12
3	2	1	21
4	2	2	23
5	3	1	31
6	3	2	32
7	4	1	43

Figura 7: Vista *trips_summary* en MySQL.

lumeClaim (PVC) individual para cada réplica del *StatefulSet*. El volumen obtenido a partir de este PVC se monta en la ruta `/var/lib/mysql` dentro del contenedor, que es el directorio donde MySQL almacena sus datos. De esta forma, los datos de la base de datos se conservan incluso si el pod se reinicia o se reemplaza, garantizando la persistencia y la integridad de la información almacenada. lo cual es esencial para un sistema gestor de base de datos. Es digno de mención también el uso del argumento `--local-infile=1` con el que se lanza MySQL, que permite el uso de la opción `LOAD DATA LOCAL INFILE` del cliente de línea de comandos de MySQL, mencionado en el apartado de MySQL CSV Loader.

Además del *StatefulSet*, se define también un *Service*, que permite exponer el MySQL como un solo servicio independientemente del número de pods con los que está siendo ejecutado. El *Service* actúa como un punto de entrada estable dentro del clúster de Kubernetes, asignando un nombre DNS fijo al despliegue de MySQL. De este modo, tanto si existe una sola réplica como si se escala a varias, las aplicaciones clientes pueden conectarse siempre a la misma dirección, sin preocuparse por los cambios internos en los pods. Este mecanismo simplifica la integración con otras aplicaciones y garantiza la disponibilidad del servicio, gestionando automáticamente el enrutamiento del tráfico hacia el pod (o los pods) correspondientes. Así, el Service abstrae la dinámica de los pods subyacentes y asegura una comunicación robusta y transparente dentro del entorno de Kubernetes. Cabe destacar que el *Service* no permite acceso externo al clúster, por lo que solo es accesible por aplicaciones internas desplegadas dentro del propio clúster.

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

```yaml
metadata:
  name: mysql
  namespace: mysql
spec:
  serviceName: mysql
  replicas: 1
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: mysql
        image: mysql:8.0
        args:
        - "--local-infile=1"
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-secret
              key: password
        ports:
        - containerPort: 3306
        volumeMounts:
        - name: mysql-data
          mountPath: /var/lib/mysql
  volumeClaimTemplates:
  - metadata:
      name: mysql-data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

Listing 7: Manifiesto YAML del *StatefulSet* de MySQL

4.3. EMTMetrics

Servicio encargado del cálculo de las predicciones y métricas, escrito en Python y usando el framework FastAPI, constituye una API REST que expone diferentes endpoints a través de los cuales se puede solicitar que haga distintos tipos de predicciones. A continuación se presentan los endpoints disponibles:

- `GET /api/details/bus`

  Dado un bus_id mediante un *query param*, devuelve detalles acerca del autobús con ese id:

  - La línea actual que se encuentra recorriendo
  - El sentido en el cual recorre esta línea
  - Última posición conocida (latitud y longitud)
  - Última distancia recorrida desde el inicio de la ruta (en metros)
  - Distancia total de la ruta que recorre
  - Paradas de la ruta que recorre

  La figura 8 muestra un ejemplo de petición realizada en Postman a este endpoint, junto con su respectiva respuesta.

- `GET /api/details/shape`

  Dado un bus_id mediante un *query param*, devuelve todos los puntos que constituyen la silueta de la ruta que recorre el autobús con ese id.

- `POST /api/predictions/position`

  Recibe un objeto JSON que contiene un bus_id y un valor prediction_time_seconds, el cual indica el número de segundos a futuro para realizar la predicción. A partir de estos datos, estima la posición que ocupará el autobús identificado por el bus_id transcurrido el intervalo de tiempo especificado desde su última ubicación conocida. La respuesta incluye:

  - El id del bus consultado

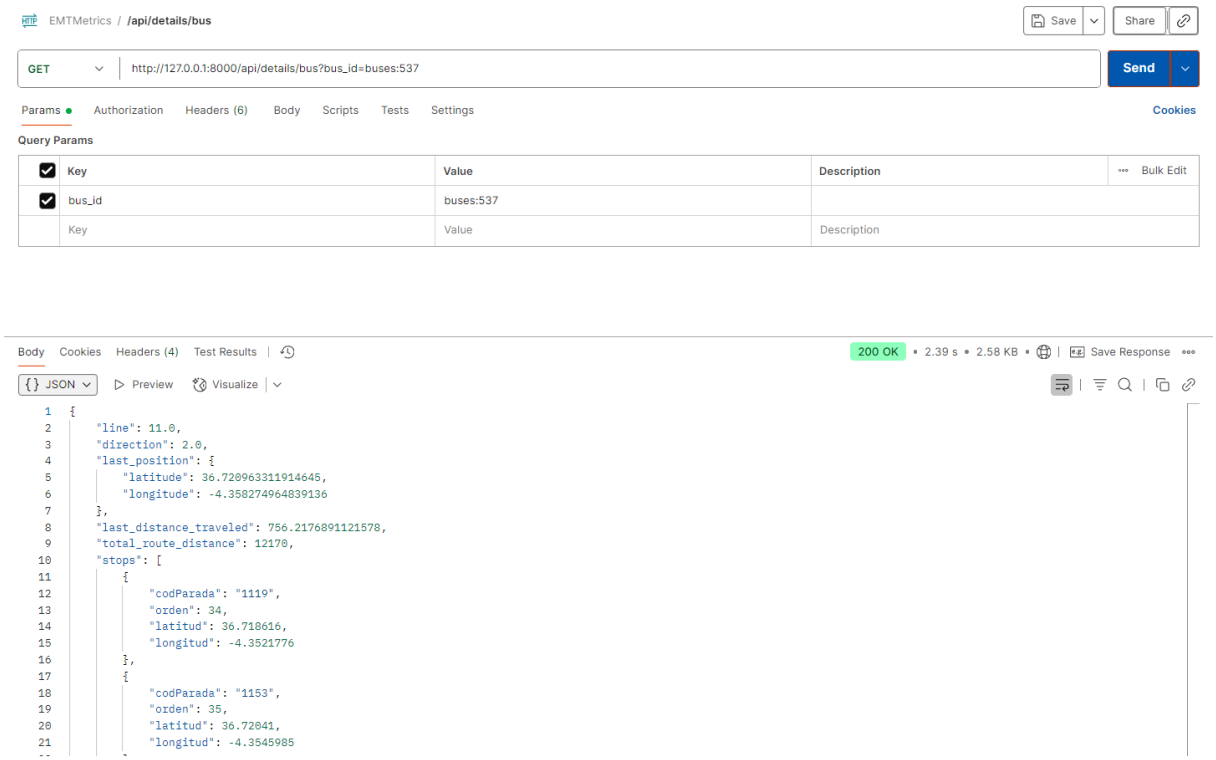


Figura 8: Ejemplo de petición y respuesta de la acción GET sobre el endpoint `/api/details/bus`.

- Última distancia recorrida desde el inicio de la ruta (en metros)
- Posición predicha (latitud y longitud)
- Distancia recorrida predicha desde el inicio de la ruta (en metros)
- Marca de tiempo en la que el autobús estará en la posición predicha
- Segundos a futuro para realizar la predicción
- Velocidad media del bus en lo que lleva de recorrido (metros por segundo)
- Mensaje adicional de la API

La figura 9 muestra un ejemplo de petición realizada en Postman a este endpoint, junto con su respectiva respuesta.

- `POST /api/predictions/time`

  Recibe un objeto JSON que contiene un bus_id y un target_location, el cual contiene los campos de latitud y longitud. Se estima entonces el momento en qué el bus alcanzará esta posición. La respuesta contiene:

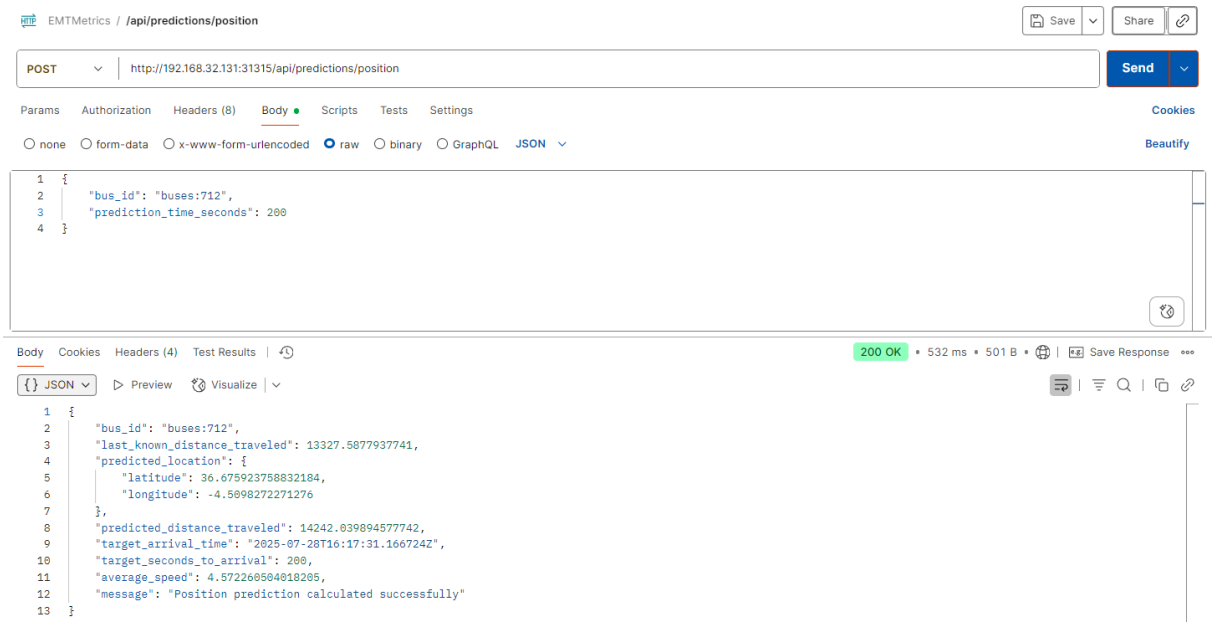


Figura 9: Ejemplo de petición y respuesta de la acción POST sobre el endpoint `/api/predictions/position`.

- El id del bus consultado
- Última distancia recorrida desde el inicio de la ruta (en metros)
- Posición cuyo tiempo de llegada se quiere predecir (latitud y longitud)
- Distancia recorrida desde el inicio de la ruta correspondiente a la posición consultada (en metros)
- Marca de tiempo en la que el autobús estará en la posición consultada
- Tiempo que tardará el autobús en alcanzar la posición consultada (segundos)
- Velocidad media del bus en lo que lleva de recorrido (metros por segundo)
- Mensaje adicional de la API

Podría decirse que este endpoint es el inverso de `POST /api/predictions/position`, aunque es ligeramente más complejo de implementar y cuenta con algunas comprobaciones adicionales como que la posición solicitada sea un punto que se encuentre en un margen de bastante cercanía a la ruta, o que este sea un punto posterior al recorrido que ya ha realizado. La figura 10 muestra un ejemplo de petición realizada en Postman a este endpoint, junto con su respectiva respuesta.

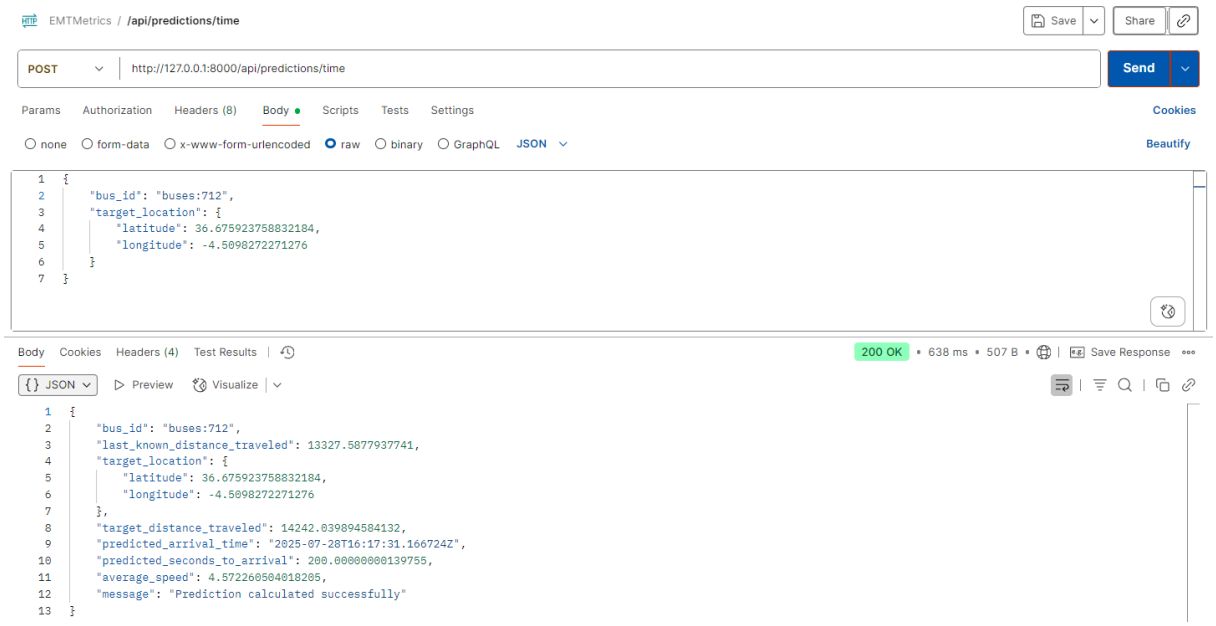


Figura 10: Ejemplo de petición y respuesta de la acción POST sobre el endpoint `/api/predictions/position`.

- `POST /api/predictions/time/bydistance`

  Realiza la misma predicción y devuelve el mismo objeto que `POST /api/predictions/time`, con la diferencia de que ahora recibe la distancia recorrida desde el inicio de la ruta como parámetro para especificar la posición cuyo tiempo se quiere predecir.

- `POST /api/predictions/time/bystop`

  Realiza la misma predicción y devuelve el mismo objeto que `POST /api/predictions/time`, esta vez recibe el número de parada de la cual se quiere saber cuanto tiempo se tardará en alcanzarse.

4.3.1. Cálculo de las predicciones

Una vez descritos los distintos endpoints que ofrece EMTMetrics a través de su API, es conveniente describir el proceso mediante el cual este realiza sus predicciones.

El código del programa está organizado en distintos paquetes y estructurado según el patrón de arquitectura por capas, separando claramente las responsabilidades en modelo, servicio y controlador. El código encargado de realizar los cálculos de las predicciones, se encuentra en

la clase con nombre *PredictionService*, de la capa de servicio.
A continuación se describen, a alto nivel, las métricas calculadas y el proceso general que sigue el sistema para realizar una predicción de la posición futura de un autobús. Nótese que estos pasos se describen con más detalle más adelante.

1. **Corrección de posiciones:** Las posiciones reportadas pueden no coincidir exactamente con un punto de la ruta. Por eso, se calcula cuál es el punto de la ruta más cercano a las posiciones inicial y final reportadas. Esto reduce los errores debidos al ruido en los datos.

2. **Cálculo de distancias sobre la ruta:** Se calcula la distancia acumulada recorrida entre los puntos corregidos, haciendo uso de los vértices de la ruta y la información almacenada en la base de datos.

3. **Cálculo de la velocidad media:** Se obtiene la velocidad media recorrida como el cociente entre la distancia recorrida y el tiempo transcurrido.

4. **Estimación de la posición futura:** Se estima la distancia total que alcanzará el autobús tras un intervalo de tiempo dado, suponiendo velocidad constante.

5. **Obtención de coordenadas:** Se interpola sobre la ruta para convertir la distancia en ruta predicha en las coordenadas geográficas correspondientes.

4.3.1.1. Corrección de posiciones

Antes de calcular la velocidad media que ha llevado el bus en el tramo recorrido, se deben corregir las posiciones inicial y final. Esto es debido a que las posiciones entregadas por el endpoint de líneas y ubicaciones no son del todo precisas, sino que existe algo de ruido. Para ello se ha implementado un algoritmo que corrige las posiciones al punto de la ruta más cercano, aprovechando que contamos con los datos de las siluetas de cada ruta. Este algoritmo, llamado `correct_position()`, se encuentra en el fichero `calculations.py` del paquete *utils*.
El funcionamiento del algoritmo de corrección de posición es el siguiente:

- Primero se asegura de que los datos de entrada sean del tipo correcto, convirtiendo los datos de posición a *float* en caso necesario.

- Posteriormente, utiliza la estructura de datos *cKDTree*, de la librería *SciPy*, para obtener de forma eficiente los dos puntos de la ruta más cercanos al punto a corregir.

- A continuación, se forman los segmentos de la ruta a analizar: para cada uno de los puntos cercanos, se construyen los segmentos que los unen con el punto siguiente y/o el anterior en la ruta (si existen); en caso de no disponer de alguno de estos puntos, ese segmento simplemente se descarta.

- Para cada uno de estos segmentos, calcula la proyección ortogonal de la posición del autobús sobre el segmento (lo que devuelve el punto más cercano—que puede estar entre los vértices—y no solo en los nodos de la ruta). De entre todos los puntos proyectados sobre los segmentos, se queda con aquel más cercano al punto original. La figura 11 presenta de forma visual como funciona esta corrección de posición, mostrando los distintos elementos involucrados.

- Si el punto más cercano encontrado está a más distancia del máximo permitido por el parámetro max_distance, lanza una excepción para indicar que la posición es demasiado lejana y no debe corregirse.

- Finalmente, devuelve el punto corregido en la ruta (coordenadas más cercanas), la distancia entre la posición original y la corregida, el segmento de la ruta donde se proyectó el punto corregido.

La figura 12 presenta un mapa interactivo basado en Leaflet[10] y generado mediante la librería Folium[11]. En el mapa se distinguen dos posiciones del autobús: la posición original, indicada en rojo, corresponde a la coordenada proporcionada directamente por la API de ubicación; la posición corregida, representada en verde, se sitúa sobre el segmento de la ruta (línea morada) sobre el que se proyecta la posición original. La silueta azul muestra el recorrido completo que realiza el autobús.

4.3.1.2. Cálculo de distancias sobre la ruta

Una vez se han corregido las posiciones inicial y final, se calcula la distancia entre ambas. Esto se realiza haciendo uso de nuevo de la tabla de *shapes*, de MySQL. Calcularemos para cada uno de los vértices de los segmentos donde se encuentran proyectados las posiciones inicial y final

[10] https://leafletjs.com/
[11] https://python-visualization.github.io/folium/latest/

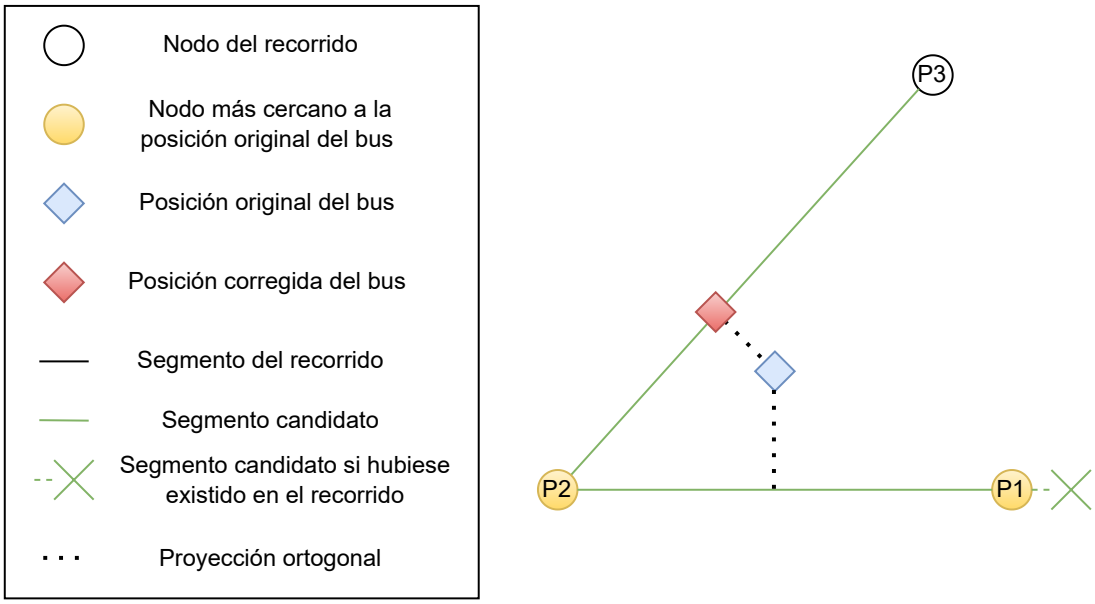


Figura 11: Diagrama que ilustra la corrección de posición de un bus.

corregidas, que distancia recorrida (dist_traveled) en la ruta le corresponde. Como los vértices de los segmentos son puntos que definen la ruta, es tan sencillo como consultar en la tabla de la base de datos.

Estas distancias nos serán de ayuda en el siguiente paso, el cual es calcular las distancias recorridas en la ruta de los puntos corregidos, usando la función `calculate_distance_along_route()`, de nuevo del fichero `calculations.py`. Esta función sirve para estimar con precisión la distancia recorrida desde a hasta el punto proyectado p sobre el segmento (a, b), conociendo la distancia entre a y b:

Primero, se calcula la distancia en línea recta desde a hasta p (d_{ap}) y la distancia en línea recta total del segmento (d_{ab}^{straight}), ambas usando la fórmula de Haversine [46]:

$$d_{ap} = \text{Haversine}(a, p)$$
$$d_{ab}^{\text{straight}} = \text{Haversine}(a, b)$$

A continuación, se calcula el cociente (ratio), que indica la proporción recorrida sobre el segmento:

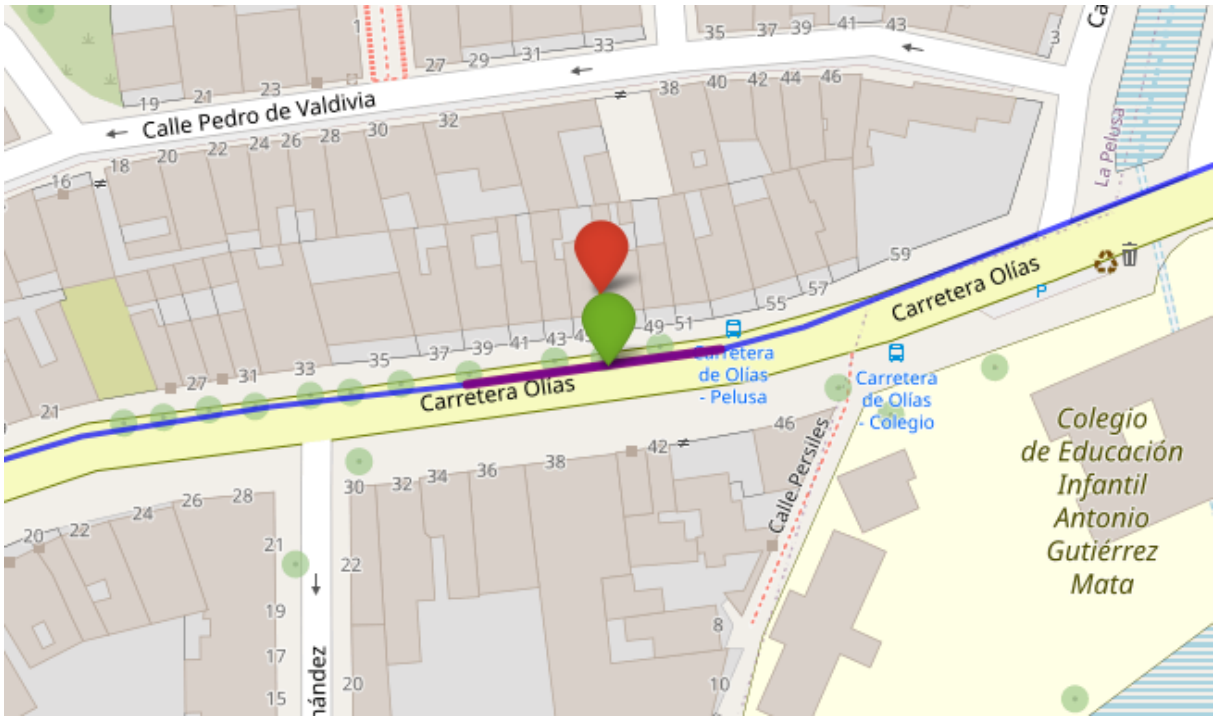


Figura 12: Posición corregida frente a la original del bus 646 al recorrer la línea 3.

$$\text{ratio} = \frac{d_{ap}}{d_{ab}^{\text{straight}}}$$

Después, este ratio se multiplica por la distancia real del segmento a lo largo de la ruta, d_{ab}, que corresponde a la diferencia de las distancias acumuladas de los dos vértices ($d_b - d_a$). Aunque pueda parecer igual que d_{ab}^{straight}, no lo es necesariamente: d_{ab}^{straight} representa la distancia en línea recta sobre la superficie esférica (es decir, la distancia del arco de círculo máximo entre los puntos, según la fórmula de Haversine), mientras que d_{ab} refleja la distancia real recorrida sobre la ruta, que puede incluir pequeñas curvas o desviaciones respecto a esa recta ideal:
Por tanto, la distancia real sobre la *shape* desde a hasta p queda:

$$\text{Distancia\_ruta}(a, p) = \text{ratio} \times d_{ab}$$

Finalmente, para los puntos inicial y final, se suma la distancia acumulada hasta el inicio del segmento y la adicional recorrida hasta la posición proyectada, calculada con la función anterior. Así, se obtiene la distancia recorrida total de cada punto corregido sobre la ruta:

- Para el punto inicial:

$$D_{\text{ini}} = d_a + \text{calculate\_distance\_along\_route}(a, b, p_{\text{ini}}, d_{ab})$$

- Para el punto final:

$$D_{\text{fin}} = d_{a'} + \text{calculate\_distance\_along\_route}(a', b', p_{\text{fin}}, d_{a'b'})$$

donde a, b son los vértices del segmento donde cae la proyección de la posición inicial, y a', b' los correspondientes a la posición final.
La distancia total recorrida entre ambos puntos será:

$$|D_{\text{fin}} - D_{\text{ini}}|$$

4.3.1.3. Cálculo de la velocidad media

Una vez obtenidas las distancias recorridas sobre la ruta corregida en los puntos inicial y final, el siguiente paso es calcular la velocidad media, y a partir de ella, estimar la distancia recorrida en la ruta para cualquier incremento de tiempo dado.
La velocidad media se calcula simplemente como el cociente entre la diferencia de distancias y la diferencia de tiempos:

$$v_{\text{media}} = \frac{D_{\text{fin}} - D_{\text{ini}}}{t_{\text{fin}} - t_{\text{ini}}}$$

donde:

- D_{ini} y D_{fin} son las distancias acumuladas sobre la ruta en los puntos inicial y final (tal y como se han obtenido en los pasos previos),
- t_{ini} y t_{fin} son los instantes temporales (timestamps) asociados a estos puntos.

4.3.1.4. Estimación de la posición futura

Para predecir la distancia recorrida sobre la ruta para un punto futuro a un incremento de tiempo conocido Δt respecto al instante inicial, basta con suponer velocidad media constante y aplicar:

$$D_{\text{pred}} = D_{\text{ini}} + v_{\text{media}} \cdot \Delta t$$

donde:

- D_{pred} es la distancia acumulada sobre la ruta que se espera recorrer tras un intervalo de tiempo $\Delta t = t_{\text{pred}} - t_{\text{ini}}$,
- v_{media} es la velocidad media calculada arriba, en metros por segundo,
- Δt está expresado en segundos.

4.3.1.5. Obtención de coordenadas

Una vez calculada la distancia acumulada sobre la ruta correspondiente al instante a predecir (D_{pred}), el objetivo es determinar la latitud y longitud que corresponden exactamente a esa posición.
Para ello:

- Se localiza en la *shape* el tramo $[A,\ B]$ donde D_{pred} está comprendido entre las distancias acumuladas de los vértices ($d_a \leq D_{\text{pred}} \leq d_b$).
- Se interpola linealmente la posición dentro del segmento usando la función `interpolate_point` de `calculations.py`, siguiendo:

$$f = \frac{D_{\text{pred}} - d_a}{d_b - d_a}$$

$$\text{lat}_p = \text{lat}_a + f \cdot (\text{lat}_b - \text{lat}_a)$$

$$\text{lon}_p = \text{lon}_a + f \cdot (\text{lon}_b - \text{lon}_a)$$

- El resultado, $(\text{lat}_p,\ \text{lon}_p)$, se toma como la posición estimada del vehículo sobre la ruta, exactamente a la distancia D_{pred} desde el origen.

De esta forma, es posible traducir cualquier distancia acumulada predicha sobre la ruta a unas coordenadas geográficas precisas, mejorando la precisión y coherencia espacial del sistema.

> **Apunte:** El procedimiento inverso, que también implementa este módulo, simplemente se trata de estimar cuánto tiempo se tardará en alcanzar una posición concreta sobre la ruta, partiendo de la información ya calculada de la distancia recorrida acumulada y la velocidad media. Aunque no se detalla en esta memoria, el proceso es similar al implementado para estimar posiciones a partir del tiempo,

pero en sentido contrario: una vez que se conoce la distancia deseada sobre la ruta, se calcula el tiempo necesario para llegar hasta ese punto a partir del momento inicial.

4.4. Interfaz de Usuario

La interfaz de usuario de este trabajo está implementada en Grafana, dado que es la solución adoptada por la plataforma OpenTwins para la visualización y monitorización de datos. Grafana ofrece una integración nativa con múltiples fuentes de datos y destaca por su capacidad para construir paneles interactivos y personalizables, lo que facilita la presentación clara y en tiempo real de datos relevantes para la plataforma.

Este trabajo implementa dos paneles de control (dashboards) de Grafana con finalidades diferentes. Ambos paneles de control se encuentran definidos en formato *json* en el repositorio al que se dedica la sección 4.5.

4.4.1. Panel de control de la situación general de la infraestructura

Se encarga de dar una perspectiva global de la red de autobuses urbanos de la ciudad en el momento de consulta del mismo. Mostrando la última posición conocida de cada autobús en circulación en la última hora, además del número de autobuses cuya posición esta siendo representada.

Presenta una variable de tipo *query*, *Linea*, utilizada para filtrar las posiciones mostradas a aquellas correspondientes únicamente a los autobuses que recorren la línea seleccionada, además de mostrar la silueta del recorrido de esta. La variable usa la fuente de datos de InfluxDB para consultar los posibles valores que esta puede tomar, que son todas aquellas líneas de las que se dispongan datos en la última hora. El listado 8 muestra la consulta realizada en lenguaje Flux, para obtener los posibles valores de esta variable.

```
import "strings"

from(bucket: "default")
  |> range(start: -1h)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
  |> filter(fn: (r) => r["_field"] == "value_line_properties_code")
  |> keep(columns: ["_value"])
```

```
  |> distinct(column: "_value")
  |> map(fn: (r) => ({ r with code_num: float(v: r._value) }))
  |> sort(columns: ["code_num"])
  // Convertir a string y eliminar el ".0"
  |> map(fn: (r) => ({
      _value: strings.split(v: r._value, t: ".")[0]
  }))
  |> keep(columns: ["_value"])  // Solo dejas la columna string original
```

Listing 8: Consulta de las líneas a InfluxDB para la variable del panel de control de la situación general de la infraestructura en Grafana

La información se muestra a través de dos paneles:

- Un panel de tipo *Stat*, denominado **Numero de buses circulando**, se encarga de mostrar el número de autobuses en circulación. Esto lo realiza a través de una sencilla consulta a la fuente de datos InfluxDB.
- Un panel de tipo *GeoMap*, con nombre **Mapa actual**, muestra la posición de cada uno de los autobuses que han estado circulando en la última hora. Realiza dos consultas, la primera a la fuente de datos InfluxDB, se encarga de obtener las posiciones de los autobuses para representarlo en una capa de tipo *markers*, el código de la consulta se encuentra recogido en el listado 9. La segunda consulta, a la base de datos MySQL, obtiene los datos de la ruta de la línea seleccionada por la variable *Linea*, en caso de que esta sea distinta del valor "All", estos datos se usan para representar la silueta de la ruta que recorre cada línea. Al hacer clic sobre una de las posiciones en el mapa, se despliega un recuadro informativo con datos relativos al autobús correspondiente. En este se incluye el enlace denominado «Ubicación Bus»; al seleccionarlo, el sistema redirige al panel de control de la situación de un autobús concreto, donde es posible consultar de manera detallada la situación específica del vehículo.

```
from(bucket: "default")
  |> range(start: -1h)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
  |> filter(fn: (r) =>
    r["_field"] == "value_gps_properties_longitude" or
```

```
    r["_field"] == "value_gps_properties_latitude" or
    r["_field"] == "value_line_properties_direction" or
    r["_field"] == "value_line_properties_code"
  )
  |> map(fn: (r) => ({ r with _value: float(v: r._value) }))
  |> pivot(rowKey: ["_time"], columnKey: ["_field"], valueColumn: "_value")
  // Filtros opcionales usando variables de Grafana
  |> filter(fn: (r) =>
    ${Linea} == 0.0 or
    (exists r.value_line_properties_code and r.value_line_properties_code == ${Linea})
  )
  |> group(columns: ["thingId"])
  |> sort(columns: ["_time"], desc: true)
  |> group()
  |> unique(column: "thingId")
```

Listing 9: Consulta de las posiciones de los autobuses a InfluxDB para mostrarlas en el panel de control de la situación general de la infraestructura en Grafana

Ambos paneles variarán la información mostrada en función del valor de la variable *Linea*, pudiendo mostrar la información correspondiente a cada una de las líneas de forma sencilla, o para todas las líneas si así se desea.

La figura 13 muestra el panel de control con la información de todas las líneas, sin aplicar ningún filtro. Por el contrario, en la figura 14 se presenta la información filtrada para la línea 3.

En caso de no filtrar por ninguna línea (valor por defecto de la variable *Linea*, "*All*"), se mostrarán las últimas posiciones conocidas de todos los autobuses.

4.4.2. Panel de control de la situación de un autobús concreto

Su función es mostrar el último recorrido que lleva realizado un bus concreto. Para ello, muestra todas las posiciones conocidas, que tengan el mismo valor de código de línea y de sentido (lo que garantiza que son posiciones del mismo recorrido). Además, este panel de control es capaz de hacer uso de las capacidades predictivas del módulo *EMTMetrics*, de forma que puede representar, junto con las posiciones conocidas, la próxima posición que tomará el bus en un tiempo determinado.

El panel de control presenta dos variables:

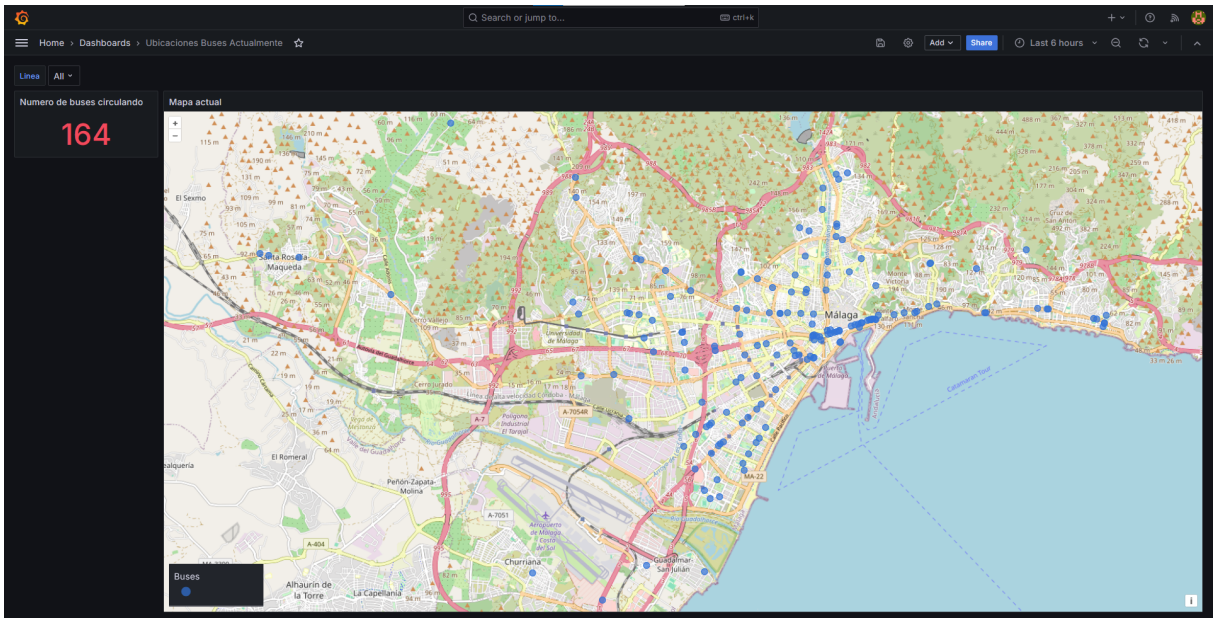


Figura 13: Panel de control de la situación general de la infraestructura mostrando datos de todas las líneas.

- *Identificador del bus*, el cual no es más que el id del bus cuyo recorrido queremos visualizar. Los posibles valores que puede tomar esta variable vienen dados por una consulta a *InfluxDB*.

- *Segundos predicción*, que representa el tiempo en segundos usado para la predicción de la próxima posición. Los valores predefinidos en el desplegable que puede tomar esta variable son los múltiplos de 60, hasta 500, aunque el usuario puede escribir el valor que desee escribiéndolo manualmente.

Los paneles presentes en el panel de control para representar la información son:

- *Detalles del bus*, de tipo *Table*, muestra información relevante al recorrido que realiza el bus en cuestión, como lo son: la línea, el sentido, distancia en ruta de la última posición conocida y la distancia total de la ruta que recorre. Esta información la obtiene de *EMTMetrics* mediante el plugin *Infinity*, el cual es extremadamente versátil, pues proporciona la capacidad de consultar a una API REST, y obtener sus datos para usarlos en el panel que se desee. Como *EMTMetrics* devuelve los datos en formato JSON, se utiliza el parser *JSONata* para extraer los datos. El endpoint de *EMTMetrics* consultado es el `GET /api/details/bus`.

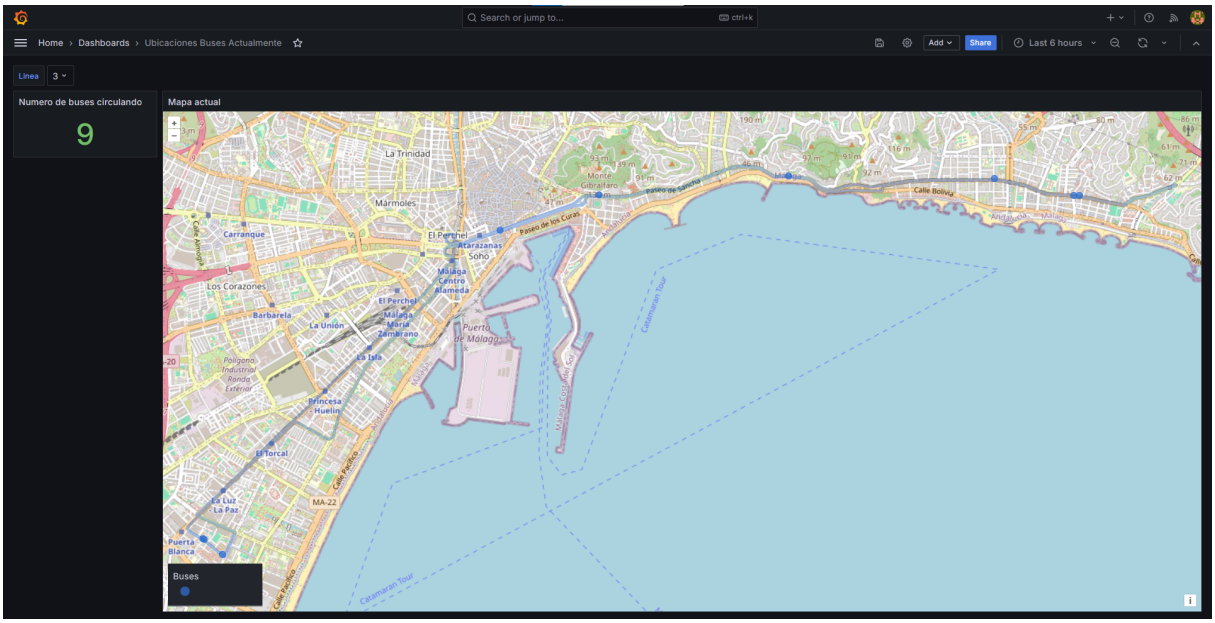


Figura 14: Panel de control de la situación general de la infraestructura mostrando datos de la línea 3.

- *Mapa*, de tipo *Geomap*, es el que muestra propiamente las posiciones del recorrido del bus. Para lograrlo, emplea cuatro consultas diferentes, dos a *InfluxDB* y dos a *EMT-Metrics*, estas últimas mediante *Infinity*. Se describen a continuación cada una de estas consultas, relacionándolas con su utilidad para el panel.

  - Consulta A:

    Se encarga de obtener las posiciones de cada una de las paradas correspondientes a la línea que recorre el bus consultado, esta información la obtiene de *InfluxDB*. La capa o *layer* que hace uso de esta consulta, es de tipo *markers*, por lo que utiliza la latitud y la longitud para representar las posiciones de cada una de estas paradas en el mapa, en color rojo y con forma de cruz.

  - Consulta B:

    Obtiene las posiciones del bus en su recorrido actual, al igual que la consulta anterior, realiza la consulta a *InfluxDB* y las representa en una capa *markers*, mostrando cada posición del bus como un círculo de color azul.

  - Consulta C:

    Obtiene los puntos que describen la silueta del recorrido actual del bus, realizando

una peticion al endpoint `GET /api/details/shape` de *EMTMetrics*, a través de *Infinity*. Posteriormente, estos puntos son usados en una capa de tipo ruta, la cual muestra el trazado que une estos puntos de forma suave.

- Consulta D:

  Se encarga de obtener la predicción de posición que proporciona *EMTMetrics* en su endpoint `POST /api/predictions/position`. De nuevo hace uso del plugin *Infinity*, usando la variable *Segundos predicción* para introducirla en el JSON de la petición de forma que el usuario pueda controlar el tiempo deseado para la predicción. La figura 15 muestra el uso de *Infinity* para realizar la consulta D, en el menú de configuración del panel.

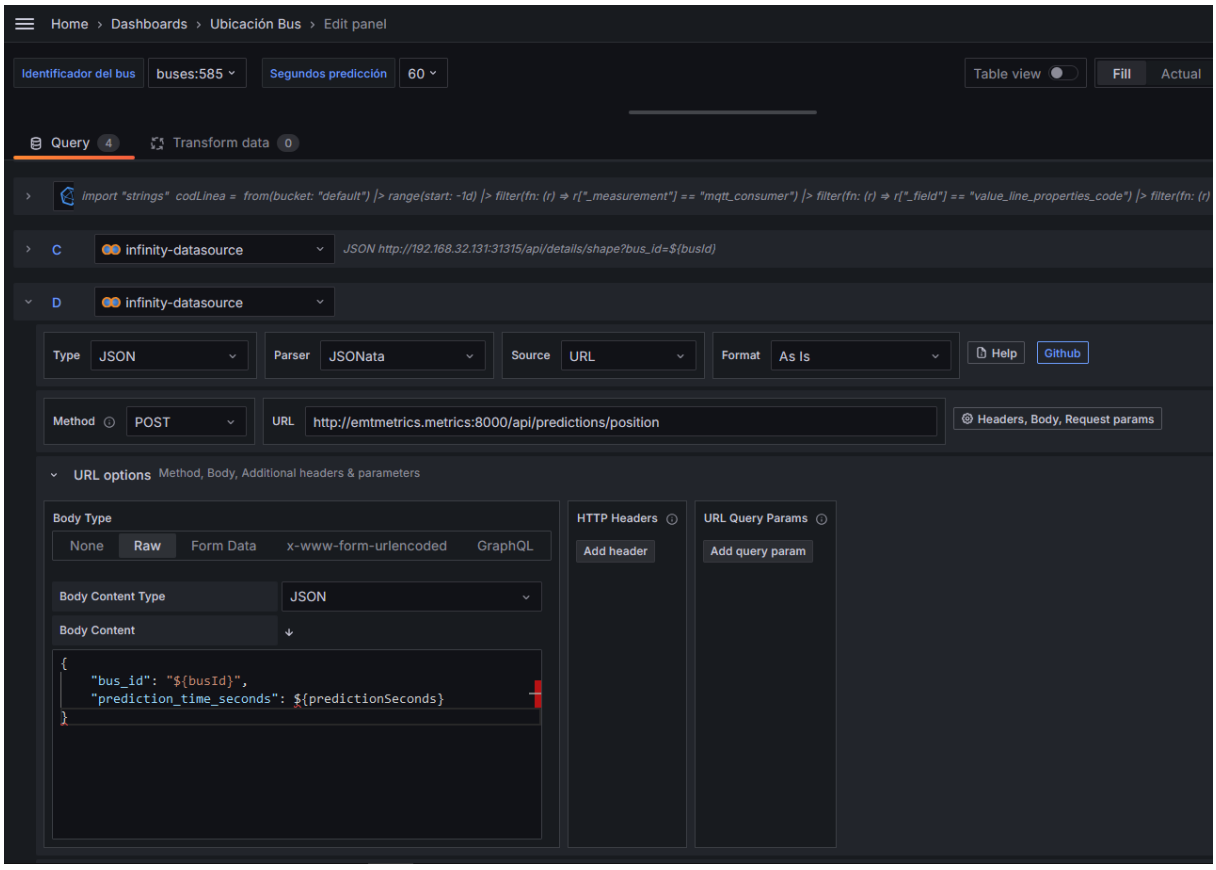


Figura 15: Configuración de la consulta D del panel *Mapa*.

La figura 16 muestra una captura de la interfaz del panel de control de la situación de un autobús concreto.

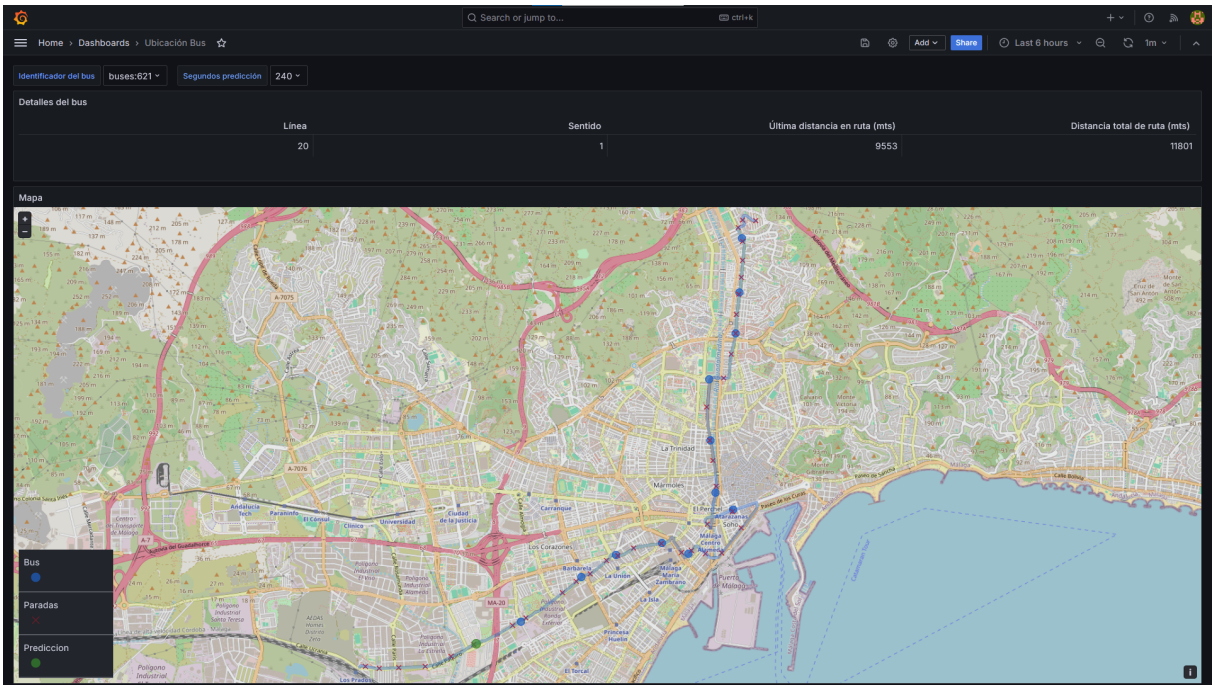


Figura 16: Panel de control de la situación de un autobús concreto mostrando la situación del bus 621 recorriendo la línea 20

4.5. EMTrack-k8s

Aunque no constituye un componente software propiamente dicho, este repositorio desempeña un papel central en el proyecto, ya que es responsable de la orquestación del despliegue de la solución. En él se reúnen todos los archivos necesarios para poner en marcha el sistema en un clúster de Kubernetes desde cero. Entre estos archivos se encuentran los manifiestos YAML que describen los componentes a desplegar en Kubernetes, archivos JSON que configuran conexiones de Eclipse Ditto o paneles de control (dashboards) de Grafana, así como scripts en Bash que contienen las instrucciones para la puesta en funcionamiento del proyecto.

Cuenta con un *README*, que describe los requisitos previos al despliegue, y los procedimientos de despliegue y actualización. Consultando el README, el lector podrá notar que el script central encargado de desplegar el proyecto es `tfg-deploy.sh`, el cual para ser ejecutado requiere recibir como primer y único argumento un fichero `.env`, que variará dependiendo del entorno en el que se despliegue, ya que contiene valores como la IP del clúster, tokens o contraseñas. Pese a que no se proporciona ningún archivo .env concreto ya que este debe ser secreto y custodiado por la persona encargada de su despliegue, sí que se distribuye un *template* para

que el usuario sepa que valores debe establecer.

El script `tfg-deploy.sh` realiza los siguientes pasos:

- Crea los espacios de nombre (*namespaces*) de kubernetes en los que se estructurarán los distintos componentes, los cuales se definen en el fichero `./deployments/namespaces.yaml`.

- Despliega el *helm chart* de OpenTwins, se debe utilizar un *fork*, distribuido en un repositorio GitHub debido a que ha sido necesario realizar un ligero cambio al *upstream* de OpenTwins: la versión de Grafana a desplegar debe ser perteneciente a la rama v10.x, en lugar de la v11.x que despliega OpenTwins, esto es debido a un *bug* observado durante el desarrollo del proyecto, en el que la capa de tipo "Ruta" del panel "Geomap" en los *dashboards* de Grafana se mostraba con opacidad mínima, haciendo que fuese prácticamente inapreciable la silueta de la ruta. La rama v10.x de Grafana no presenta este error, por lo que se ha modificado OpenTwins para que instale la versión 10.4.19. Aunque este trabajo ha intentado contribuir a OpenTwins cuando se ha detectado algún error[12], este cambio como tal no es un fallo de OpenTwins, si no que es un problema de Grafana que afecta concretamente al caso de uso de este proyecto, de ahí la necesidad del *fork*[13].

- Sube al registro de artefactos, por defecto el proyecto usa Harbor, todas las imágenes Docker que el clúster requiera utilizar.

- Despliega todos los componentes adicionales a OpenTwins añadidos por este trabajo, los cuales son:

    - MySQL
    - EMTScraper BusLocation
    - EMTScraper Routes
    - EMTScraper MySQL CSV Loader
    - EMTMetrics

[12] https://github.com/ertis-research/Helm-charts/pull/1
[13] https://github.com/isrojas1/OpenTwins-Chart

- Fuerza una primera ejecución de los CronJobs, los cuales son EMTScraper Routes y MySQL CSV Loader. De este modo, no se tiene que esperar a que llegue la hora establecida en el horario del cron, dejando al sistema sin los datos aportados por estos componentes hasta que se ejecuten, si no que una primera ejecución tras el despliegue o actualización hace que se cuenten con datos desde el primer momento.

Hay otro script que se encarga de importar en Eclipse Ditto la conexión con el servidor Mosquitto en las direcciones donde envían mensajes los componentes de EMTScraper, además de importar en Grafana tanto la fuente de datos de MySQL como los *dashboards* con los que interacciona el usuario que accede al sistema. Este script tiene el nombre de `import-resources.sh`, de nuevo requiere un archivo .env como parámetro. El script define funciones para realizar peticiones PUT y POST, las cuales utiliza posteriormente para importar los datos mencionados, en formato JSON, a través de las APIs HTTP de Eclipse Ditto y Grafana.
Cabe por último hacer una mención al script `harbor-deploy.sh`, el cual puede ejecutarse para desplegar Harbor en el clúster, así como configurar este para que lo utilice como registro de imágenes de contenedores. El script asume que se usa k3s como implementación de kubernetes, ya que modifica el archivo `/etc/rancher/k3s/registries.yaml`, específico de esta distribución de Kubernetes.

5 Pruebas

5.1. Pruebas manuales de funcionalidad y usabilidad

En esta sección se exponen las pruebas manuales realizadas para probar que la usabilidad del TFG sea correcta y que las funcionalidades principales descritas a lo largo de la memoria actúen de forma adecuada.

5.1.1. Prueba de despliegue

Con el objetivo de validar la robustez y utilidad de los scripts de despliegue proporcionados en el repositorio emtrack-k8s[14], esta prueba consiste en verificar tanto el correcto funcionamiento de dichos scripts como la claridad y precisión del manual de despliegue descrito en el anexo A. Para ello, se realiza el despliegue completo siguiendo las instrucciones suministradas.

El resultado obtenido de esta prueba ha sido favorable, ya que permitió no solo verificar que los scripts de despliegue funcionaban correctamente, sino también identificar y corregir aspectos susceptibles de mejora. Durante el proceso se detectaron intervenciones manuales que, aunque resultaban sencillas para el autor por su conocimiento previo del sistema, podrían suponer un inconveniente para usuarios ajenos. Por este motivo, se introdujeron ajustes en los scripts orientados a automatizar tareas repetitivas y resolver problemas comunes de forma transparente, facilitando así la experiencia de despliegue.

En la versión final, se constató que el gemelo digital opera de manera adecuada cuando se siguen los procedimientos indicados; todas sus funcionalidades pasan a estar disponibles a medida que el sistema comienza a recolectar datos suficientes desde las APIs abiertas relativas a las posiciones de los autobuses, logrando así una solución funcional y accesible para cualquier usuario que desee implantarla. Esta evolución no solo mejora la usabilidad, sino que refuerza

[14] https://github.com/isrojas1/emtrack-k8s

la robustez del despliegue y la fiabilidad operativa del gemelo digital desde las primeras fases de recogida de datos.

5.1.2. Prueba de ejecución a largo plazo

El sistema se ha ejecutado en una máquina que operaba de forma ininterrumpida, permitiendo analizar su estabilidad y comportamiento bajo condiciones de uso prolongado. Durante este periodo, se monitorizó el funcionamiento del gemelo digital y de todos sus servicios asociados, observando la capacidad del sistema para recopilar, procesar y visualizar información en tiempo real a medida que los datos eran recibidos desde las APIs abiertas de los autobuses. Asimismo, se verificó la ausencia de errores críticos o degradaciones significativas en el rendimiento, lo que demuestra la robustez de la infraestructura desplegada y su idoneidad para uso continuado en entornos reales. Esta prueba confirma que el sistema mantiene la sincronización constante entre los datos recibidos y su representación digital, garantizando la disponibilidad y fiabilidad de las funcionalidades proporcionadas por el gemelo digital a lo largo del tiempo. Como prueba de la realización de esta prueba, se provee como parte de los ficheros adicionales entregados una copia de los datos recolectados por esta máquina durante el tiempo que estuvo operativo, con datos que datan desde el 11 de diciembre de 2024, por lo que hay datos recolectados por distintas versiones de la solución, siendo algunas muy tempranas. Estos datos pueden ser restaurados y usados para consultas históricas, aunque el gemelo digital no permite aún una visualización de estos datos desde su interfaz gráfica, pueden ser analizados mediante consultas directas a InfluxDB.

Esta prueba ayudó a identificar algunos errores, lo que permitió fortalecer el sistema en fases tempranas del desarrollo. Un ejemplo destacado fue la resolución de un fallo en OpenTwins relacionado con la caída del servicio Eclipse Mosquitto cuando su pod se reiniciaba[15]. Adicionalmente, se detectaron varios episodios de indisponibilidad en la API de posiciones de los autobuses, lo cual llevó a implementar mecanismos de tolerancia a errores en el componente EMTScraper BusLocation. Esto previene que el sistema colapse por completo ante fallos temporales en la obtención de datos, dotando al sistema de mayor robustez y capacidad de recuperación ante eventos externos inesperados.

[15] https://github.com/ertis-research/Helm-charts/pull/1

5.1.3. Prueba de usabilidad de la interfaz

La interfaz principal del sistema se ha evaluado de forma exhaustiva durante el desarrollo del proyecto, verificando en cada iteración que su comportamiento en condiciones normales es el esperado. En general, la aplicación responde de manera correcta: los datos se recuperan en un tiempo adecuado y se muestran de forma coherente en los paneles. Asimismo, las variables desplegables actualizan su contenido dinámicamente en función de los valores devueltos por InfluxDB, y los paneles reflejan de forma inmediata los cambios al modificar dichas variables.

La información presentada resulta consistente en prácticamente todos los casos, salvo en una situación puntual detectada cuando las paradas y el trazado de la ruta no coinciden, lo que provoca que algunas paradas aparezcan fuera del recorrido previsto. Estas incidencias son poco frecuentes y se deben a inconsistencias entre los distintos endpoints de la API abierta, no a un mal funcionamiento del sistema desarrollado.

En lo que respecta a la interfaz de la lista de gemelos digitales ofrecida por el plugin de Grafana de OpenTwins, se ha observado que su rendimiento es limitado, llegando a tardar hasta medio minuto en cargar la lista completa. No obstante, esta ralentización está asociada a problemas de escalabilidad en Eclipse Ditto al manejar un gran número de gemelos, y queda fuera del alcance del presente proyecto. Conviene destacar que, una vez cargada la lista, el acceso a cada gemelo individual se realiza de manera prácticamente instantánea.

5.2. Pruebas de tolerancia a errores

Esta sección presenta estrategias para garantizar la resistencia y estabilidad del sistema ante fallos. Este enfoque incluye también la realización de pruebas específicas para simular escenarios de error, verificando la capacidad del sistema para mantener la operatividad y evitar la pérdida de datos ante condiciones adversas.

5.2.1. EMTScraper: Caída de API de datos abiertos

Esta prueba trata de comprobar la resiliencia del sistema ante la falta de disponibilidad de la API esencial para su funcionamiento, la del portal de datos abiertos del ayuntamiento de Málaga. Para simular esta caída, se ha aplicado una política de red (*network policy*) de kubernetes que bloquea el acceso a la IP donde se encuentra alojada la API desde la cual se obtienen

los datos. El código de esta política de red se encuentra en el listado 10.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: block-emt-api
  namespace: scraping
spec:
  podSelector: {}
  policyTypes:
  - Egress
  egress:
  - to:
    - ipBlock:
        cidr: 95.129.152.231/32
    ports:
    - protocol: TCP
      port: 443
```

Listing 10: Política de red Kubernetes usada para bloquear el tráfico de salida a la API de datos abietos del ayuntamiento de Málaga

De forma prácticamente inmediata se puede apreciar que los datos mostrados en la interfaz de usuario, en cualquiera de los dos paneles de control de Grafana, se encuentran congelados, teniendo los mismos datos por más que pase el tiempo. Al acceder a los *logs* de EMTScraper BusLocation, se puede comprobar como la API es innaccesible, estos *logs* se pueden consultar en la figura 17. Pese a que la API no es accesible, el componente de recogida de datos que se encuentra en continua ejecución, EMTScraper BusLocation, no deja de funcionar en ningún momento, sino que sigue ejecutándose, capturando la excepción y volviendo a intentar conectar con la API cada minuto, hasta que consiga volver a contactarla. Ninguno de los demás componentes en ejecución deja de funcionar, por lo que el sistema permanece en funcionamiento (sin nuevos datos, como es evidente) aun cuando no puede obtener nueva información. Los paneles de control muestran "No data" cuando han pasado más de 15 minutos sin recibir datos nuevos.

Tras varios minutos, se elimina la política de red aplicada previamente, de forma que la API vuelve a estar disponible para los componentes de EMTScraper. El comportamiento es el deseado, automáticamente se obtienen los datos nuevos y los paneles de control vuelven a mos-

Figura 17: Logs de EMTScraper BusLocation al perder conectividad con la API abierta.

trar la información consultada, sin ningún tipo de interacción por parte del usuario más que refrescar el navegador web.

5.2.2. EMTMetrics: Tratar de predecir el tiempo en llegar a una posición que no se encuentra en la ruta

Esta prueba consiste en probar a hacer una consulta al endpoint `POST /api/predictions/time` de la API de EMTMetrics, usando de forma deliberada una posición de destino que no se encuentre dentro de la ruta que recorre el autobús especificado. Para ello se puede utilizar cualquier posición retornada por el endpoint `POST /api/predictions/position`, y modificarla ligeramente para ver si tolera cambios ligeros de posición.

El resultado es que la API es algo permisiva con posiciones que no se encuentren exactamente en la ruta, puesto que modificaciones de ordenes de magnitud muy pequeñas seguirán devolviendo predicciones válidas. Esto es un comportamiento deseado por la propia naturaleza del proyecto, ya que las posiciones devueltas por la API de datos abiertos, como ya se ha comentado, a menudo vienen acompañadas de algo de ruido que hace que las posiciones no sean exactas, siendo corregidas a los puntos más cercanos de la ruta. Sin embargo, cuando se desvía

el punto consultado de forma más drástica, sí que se puede ver como la API se niega a procesar la petición, devolviendo el código 500 y un mensaje explicativo, que se puede observar en la figura 18.

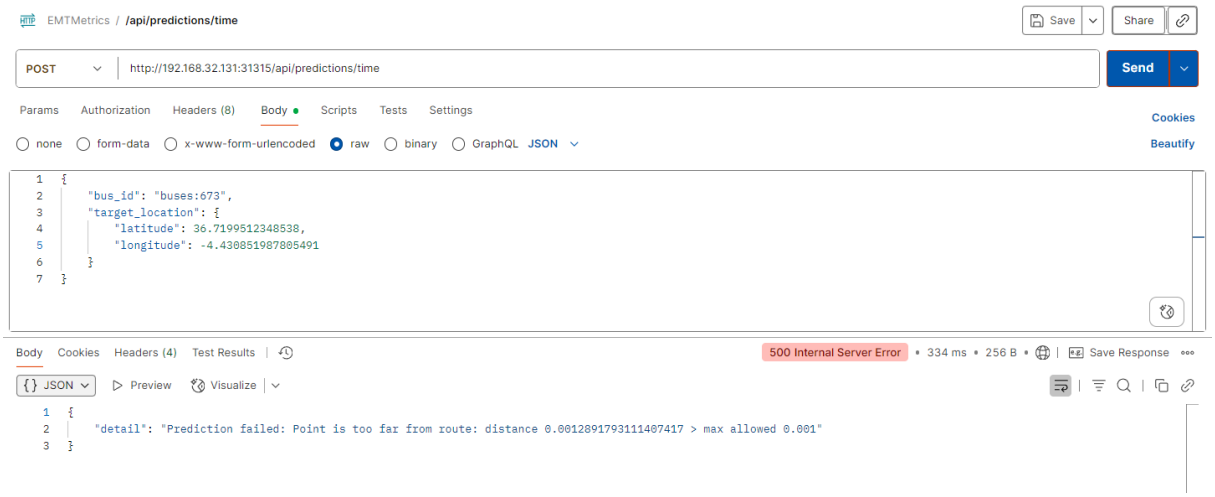


Figura 18: Petición y respuesta de EMTMetrics al consultar una posición fuera de la ruta.

5.2.3. EMTMetrics: Predecir una posición cuando no hay datos suficientes para obtener la velocidad media

Como se explicó en la sección 4.3.1, las predicciones se basan en el uso de la velocidad media del autobús en el trayecto actual que recorre. No obstante, hay un caso particular en el que es imposible obtener esta velocidad media, se trata de cuando solo se conoce una posición del trayecto actual recorrido. En este caso, en el que no se tiene una velocidad media estimada del trayecto, el sistema no calcula ninguna predicción. El objetivo es que no se muestre ninguna predicción errónea, ni haya un fallo no controlado que inutilice la plataforma.

Para realizar la prueba, se accede al panel de control “Ubicación Bus” de Grafana y se usa el desplegable para encontrar un autobús que se encuentre en la situación deseada, esto es, un autobús que justo haya comenzado a recorrer un trayecto nuevo, en el que solo haya reportado una situación por el momento. La figura 19 muestra una captura de pantalla de la situación específica. Se puede observar como no existe un punto verde en el mapa (que es el que representa la posición predicha). Además, en la esquina superior izquierda del panel se muestra un símbolo de advertencia en un recuadro de color rojo, el cual al colocar el ratón sobre él indica que la API de EMTMetrics ha devuelto un error 500.

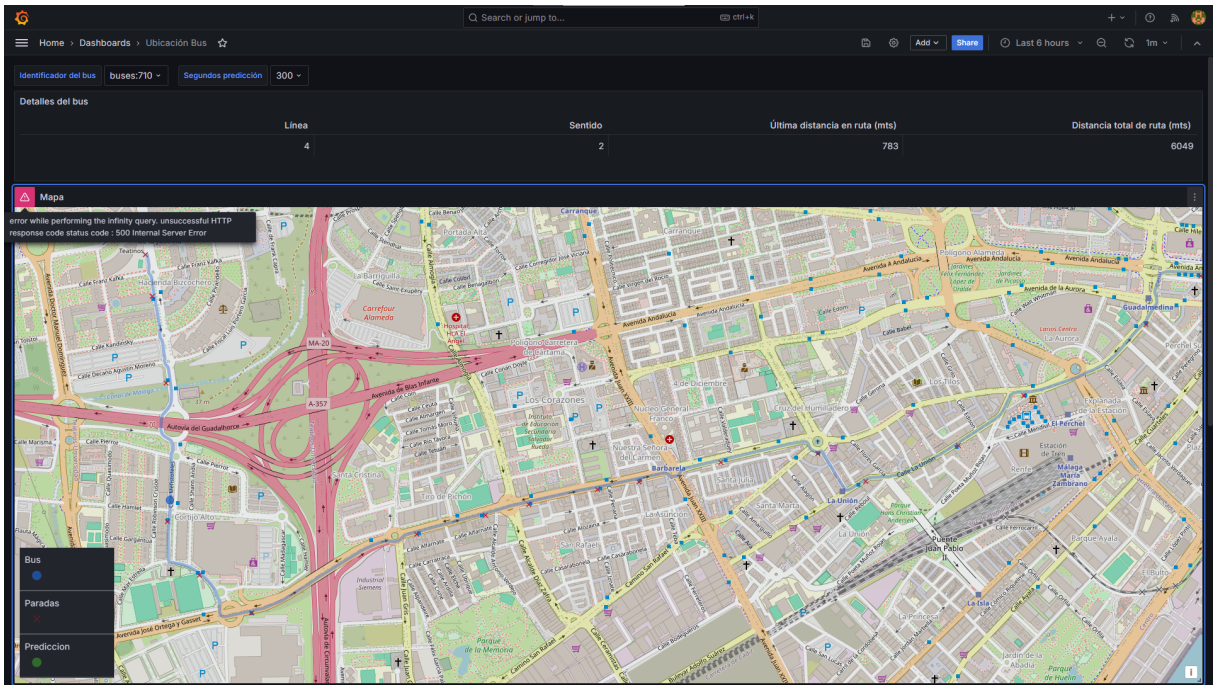


Figura 19: Panel de control “Ubicación Bus”, mostrando la situación en la que no hay datos suficientes para obtener la velocidad media.

5.2.4. EMTMetrics: La posición predicha queda más allá del límite de la ruta

En el proceso de estimación de posiciones, el sistema proyecta el avance del autobús a partir de su velocidad media y del trayecto recorrido hasta el momento. No obstante, puede suceder que el cálculo de la proyección sitúe la posición predicha fuera del recorrido planificado, concretamente más allá del punto final de la ruta establecida.

En estas circunstancias, el sistema no genera ninguna predicción, ya que mostrar un punto fuera del trazado supondría introducir un error en la visualización y dar información engañosa. En su lugar, la plataforma descarta esa estimación, evitando que aparezcan predicciones erróneas.

Para comprobar esta situación, se accede al panel de control “Ubicación Bus” en Grafana y se selecciona un autobús cuya predicción calculada sobrepase la longitud total de la ruta. La figura 20 muestra una captura de pantalla con este escenario. Puede observarse que, al igual que en el caso detallado en la sección anterior, no aparece ningún punto verde representando la posición estimada. Asimismo, en la esquina superior izquierda del panel se muestra un símbolo de advertencia en un recuadro rojo, que al situar el ratón sobre él indica que la API de EMTMetrics

ha devuelto un error 500, consecuencia de haber superado el final del recorrido definido.

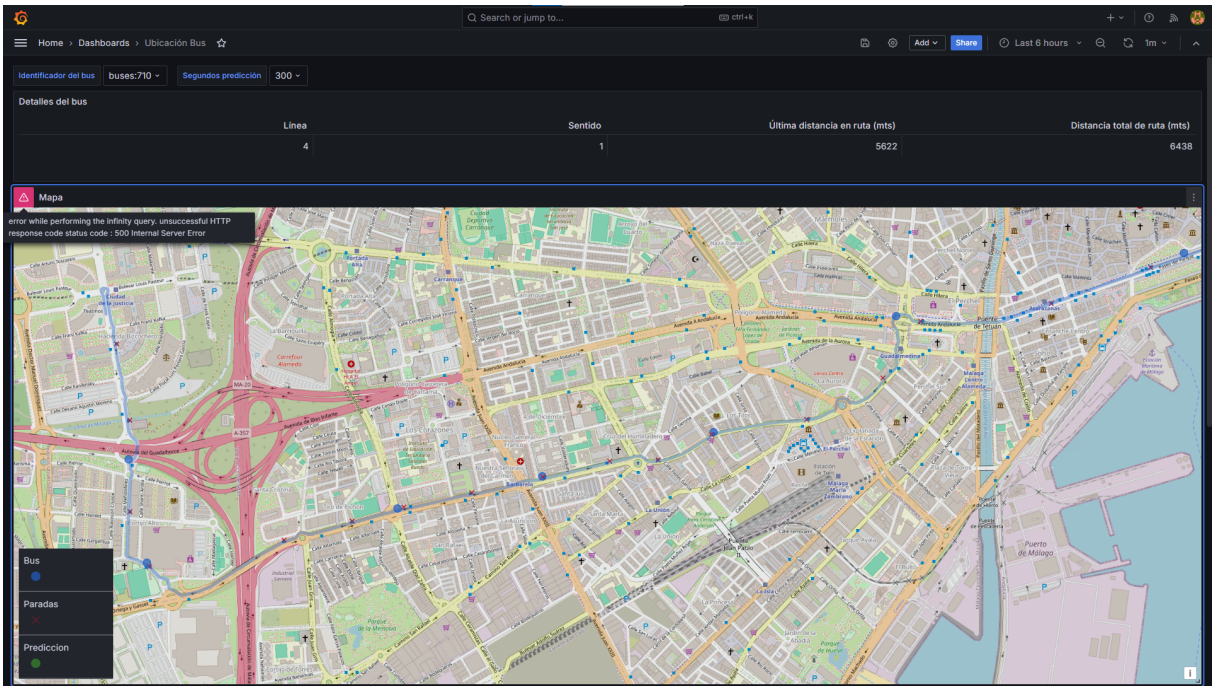


Figura 20: Panel de control “Ubicación Bus”, mostrando la situación en la que la posición predicha queda más allá del límite de la ruta.

6

Conclusiones y Líneas Futuras

6.1. Conclusiones

En este TFG se ha desarrollado un gemelo digital compuesto que modela y representa la infraestructura de autobuses urbanos de la ciudad de Málaga. Para ello, se ha usado la plataforma OpenTwins, desarrollada por el grupo de investigación ERTIS y concebida para impulsar el desarrollo de gemelos digitales de nueva generación. Gracias a la arquitectura modular y abierta con la que cuenta OpenTwins, este trabajo implementa componentes adicionales que complementan la plataforma de gemelos digitales, haciendo posible la creación de un gemelo digital urbano. El gemelo digital es un sistema completo que funciona de forma automatizada.

Una de las principales características del gemelo digital es la interoperatividad entre distintos componentes tecnológicos implementados, pues no solo se añaden funcionalidades aisladas, sino que se logra que los componentes trabajen de forma conjunta como una unidad, de forma que la solución desarrollada es un sistema completo y coherente. En este proceso de integración, se han incorporado desarrollos clave como EMTScraper, software escrito en Python que se divide en dos microservicios; Bus Location, que se ejecuta de manera continua como un *Deployment* de Kubernetes y que obtiene las posiciones de la flota autobuses que recorre la ciudad; y Routes, que se ejecuta diariamente como un *CronJob* encargado de obtener las líneas disponibles y las paradas que las conforman. Asimismo, se ha desplegado un servicio MySQL responsable de almacenar y suministrar datos estáticos esenciales para el funcionamiento del sistema, que no son apropiados ni en el MongoDB que usa Eclipse Ditto para almacenar sus datos, ni en InfluxDB, que almacena las series temporales. También se ha desarrollado *EMT-Metrics* un componente que es capaz de predecir la posición en la que estará un autobús en un

momento dado, o de sacar el tiempo que tardará en llegar a una posición concreta. El sistema se completa con unos paneles de control de Grafana que permiten consultar la información que maneja el gemelo digital de forma interactiva, visual y cómoda, permitiendo filtrar usando variables, navegar de un panel de control a otro y usar las capacidades predictivas de forma gráfica. De este modo, el sistema está conformado por elementos con responsabilidades claramente diferenciadas, los cuales se comunican entre sí mediante brokers de mensajería, APIs REST y conexiones a bases de datos, facilitando así la interacción y el flujo de información entre los distintos componentes del entorno.

La modelización de los principales elementos del sistema (autobuses, paradas y líneas) ha demostrado la viabilidad de emplear esta aproximación para el estudio y optimización de infraestructuras de movilidad. La solución presenta capacidades de visualización gráfica y monitorización en tiempo real, así como características predictivas básicas que permiten anticipar el comportamiento de los autobuses, demostrando que es posible desplegar un gemelo digital funcional aplicable a escenarios urbanos reales.

El desarrollo del proyecto ha seguido un proceso fundamentado en la aplicación de buenas prácticas de ingeniería de software, priorizando la calidad y la mantenibilidad en cada etapa. Ha resultado esencial la existencia de un entorno de pruebas específico para desarrollo, que ha permitido desplegar y verificar iterativamente cada nueva versión del sistema. Este enfoque ha facilitado no solo la validación funcional individual de los distintos componentes y de la solución en su conjunto, sino también la comprobación exhaustiva de los *scripts* de despliegue. Como resultado, se ha podido entregar un repositorio documentado con los pasos necesarios para que cualquier persona pueda poner en funcionamiento la solución en cuestión de minutos en cualquier equipo moderno.

En definitiva, este trabajo demuestra el potencial de las arquitecturas abiertas y composicionales para el desarrollo de gemelos digitales urbanos, consolidando una base sólida y ampliable para futuras aplicaciones en el ámbito de la movilidad inteligente. El enfoque adoptado no solo valida la viabilidad técnica de la solución, sino que sienta un precedente para la aplicación de estas tecnologías en entornos reales, abriendo la puerta a nuevas líneas de investigación y mejoras continuas en la gestión y optimización de infraestructuras urbanas.

6.2. Líneas Futuras

Este trabajo asienta los cimientos de una plataforma capaz de aunar datos de diversas fuentes, para representar la información de la infraestructura de autobuses de Málaga, con algunas capacidades predictivas básicas. Es un primer acercamiento que se ha centrado en la integración de las distintas tecnologías y componentes, para otorgarle al trabajo desarrollado una cohesión como plataforma completa y funcional. Por ello, existen diversas áreas de ampliación mediante las cuales este proyecto puede ser complementado, ya que su arquitectura modular abre las puertas a contribuciones que mejoren y completen la solución. Algunas de estas posibles aportaciones se exponen a continuación:

- **Generalización:** Aunque el sistema se ha desarrollado específicamente para los autobuses urbanos de Málaga, resultaría interesante poder abrir el sistema de forma que se acepten datos de distintas ciudades, incluso en formatos diversos. Hasta ahora, no se ha evaluado la viabilidad de esta mejora, que probablemente supondría un importante esfuerzo de abstracción y adaptación de los componentes, en especial de *EMTScraper*, responsable de obtener y procesar los datos de fuentes abiertas para integrarlos en el sistema. De este modo, el proyecto podría evolucionar hacia una plataforma abierta de gemelos digitales de transporte, capaz de modelar cualquier sistema de transporte, siempre que se disponga de los datos necesarios para ello.

- **Integración de datos complementarios:** Una forma de enriquecer la información disponible es incorporar datos provenientes de fuentes adicionales, lo que permite obtener métricas más explicativas y detalladas a partir de los datos recopilados. Entre las posibles fuentes de datos a integrar se incluyen, por ejemplo, información sobre el estado del tráfico en cada uno de los segmentos de las rutas recorridas por los autobuses, aunque es importante considerar que muchas de las APIs que ofrecen este tipo de datos suelen ser de pago y no siempre abiertas al público. Otra opción es utilizar imágenes captadas por cámaras públicas distribuidas en la ciudad y ubicadas en puntos estratégicos para el monitoreo del tráfico; este enfoque requeriría un procesamiento adicional mediante algoritmos de *machine learning* para inferir la situación y el estado del tráfico a partir de las imágenes obtenidas. También se pueden incorporar datos meteorológicos, que permitirían analizar cómo las condiciones climáticas afectan el tráfico y la operación

de los autobuses, facilitando la generación de predicciones más precisas y una mejor planificación del servicio.

- **Métricas explicativas e históricas:** Hasta el momento, las métricas que utiliza el sistema son internas, destinadas principalmente a alimentar las predicciones, y no se presentan de manera visualmente atractiva en la interfaz de usuario, ya que se tratan de métricas básicas como la velocidad media o la distancia recorrida. Una futura ampliación interesante sería integrar nuevas métricas que reflejen el comportamiento histórico —o de rangos temporales específicos— de cada línea, facilitando la obtención de conclusiones útiles para su optimización. Estas métricas podrían abarcar el tiempo medio de finalización de ruta, la fiabilidad del servicio a lo largo del tiempo, la precisión de las predicciones y la detección de retrasos en la llegada de los autobuses a las paradas.

- **Mejora de las capacidades predictivas:** Actualmente, las predicciones del sistema se basan en el cálculo de la velocidad media y la distancia recorrida, aplicando la relación básica $v = \frac{x}{t}$. Esto implica que el modelo asume comportamientos relativamente simples para estimar los tiempos de llegada y otros parámetros operativos. Una posible ampliación del trabajo consistiría en incorporar métodos de predicción más avanzados y complejos, como algoritmos de aprendizaje automático o modelos estadísticos sofisticados, que permitirían captar mejor las variaciones y los factores externos que afectan al servicio. Cabe señalar que la baja granularidad de los datos disponibles —ya que sólo se dispone de una actualización de posición cada 5 minutos— supondría una dificultad adicional a la hora de aplicar métodos de predicción más avanzados. No obstante, sigue siendo recomendable implementar técnicas más efectivas, ya que incluso con estas limitaciones podrían lograrse mejoras significativas en la precisión y utilidad de las predicciones.

- **Mejorar la integración entre la interfaz gráfica y el módulo de predicciones:** La integración de la interfaz gráfica, concretamente del panel de control de *Grafana* destinado a la visualización de la situación de un autobús específico, junto con *EMTMetrics*, el módulo encargado de las predicciones, resulta adecuada para el caso de la predicción de la posición futura de un vehículo. No obstante, en la actualidad solo se emplean dos de los *endpoints* disponibles en *EMTMetrics*. Por tanto, sería recomendable mejorar la in-

terfaz mediante la incorporación de nuevos paneles que permitan aprovechar de forma más cómoda y visual el resto de los *endpoints*, facilitando así cálculos adicionales, como la estimación del tiempo de llegada a ubicaciones concretas, o a las paradas del recorrido. Incluso, podría valorarse la posibilidad de desarrollar una solución propia de visualización específica para este sistema, abandonando el uso de *Grafana*, con el objetivo de adaptar completamente la interfaz a las necesidades y particularidades del proyecto.

6.3. Valoraciones personales

El proceso ha implicado un elevado grado de complejidad técnica, partiendo desde la plataforma *OpenTwins* y requiriendo una intensa labor de investigación y aprendizaje en áreas como la orquestación de contenedores mediante *Kubernetes*, la gestión e integración de gemelos digitales, la utilización y configuración de brókeres de mensajería, así como el despliegue y la coordinación de múltiples componentes software.

Este trabajo ha supuesto un esfuerzo multidisciplinar que ha permitido superar numerosos retos tecnológicos, logrando el desarrollo de un sistema estable y escalable que sienta las bases para futuras ampliaciones y aplicaciones en el ámbito de la movilidad urbana y el análisis de datos. Durante el desarrollo de este se han adquirido diferentes competencias técnicas, así como se ha mostrado una considerable capacidad y resolución de problemas. Un ejemplo especialmente relevante ha sido la necesidad de adaptarse a una frecuencia de actualización de datos de posiciones de autobuses inferior a la inicialmente prevista, lo que ha obligado a buscar soluciones creativas para realizar predicciones fiables y representar la información de manera útil, incluso con la limitación de la baja granularidad de datos.

Por último, destacar que la apuesta por el software abierto ha sido un pilar esencial para la viabilidad de la solución desarrollada. Gracias al propio *OpenTwins*, los estándares abiertos, la colaboración activa de la comunidad y el acceso a proyectos de software de gran calidad bajo licencias libres, todo este desarrollo ha sido posible, promoviendo la reutilización, la transparencia y el crecimiento colaborativo.

Referencias

[1] Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education. 692 págs. ISBN: 978-1-260-42329-7.

[2] *Smart Cities - European Commission*. URL: https://commission.europa.eu/eu-regional-and-urban-development/topics/cities-and-urban-development/city-initiatives/smart-cities_en (visitado 16-07-2025).

[3] *Smart City: Definition, Components, and Translation into Real-World, Studio Projects*. Monash University, Indonesia. 7 de nov. de 2024. URL: https://www.monash.edu/indonesia/news/what-is-smart-city-and-implementation-in-real-world (visitado 16-07-2025).

[4] Dechen Peldon et al. «Navigating Urban Complexity: The Transformative Role of Digital Twins in Smart City Development». En: *Sustainable Cities and Society* 111 (15 de sep. de 2024), pág. 105583. ISSN: 2210-6707. DOI: 10.1016/j.scs.2024.105583. URL: https://www.sciencedirect.com/science/article/pii/S2210670724004086 (visitado 16-07-2025).

[5] Adil Rasheed, Omer San y Trond Kvamsdal. «Digital Twin: Values, Challenges and Enablers From a Modeling Perspective». En: *IEEE Access* 8 (2020), págs. 21980-22012. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2970143. URL: https://ieeexplore.ieee.org/document/8972429 (visitado 23-07-2025).

[6] *The Use of Digital Twins for Urban Planning to Yield US$280 Billion in Cost Savings By 2030*. URL: https://www.abiresearch.com/press/use-digital-twins-urban-planning-yield-us280-billion-cost-savings-2030 (visitado 16-07-2025).

[7] Roberto Minerva, Gyu Myoung Lee y Noël Crespi. «Digital Twin in the IoT Context: A Survey on Technical Features, Scenarios, and Architectural Models». En: *Proceedings of the IEEE* 108.10 (oct. de 2020), págs. 1785-1824. ISSN: 1558-2256. DOI: 10.1109/JPROC.2020.2998530. URL: https://ieeexplore.ieee.org/document/9120192 (visitado 16-07-2025).

[8] Julia Robles, Cristian Martín y Manuel Díaz. «OpenTwins: An Open-Source Framework for the Development of next-Gen Compositional Digital Twins». En: *Computers in Industry* 152 (1 de nov. de 2023), pág. 104007. ISSN: 0166-3615. DOI: 10.1016/j.compind.2023.104007. URL: https://www.sciencedirect.com/science/article/pii/S0166361523001574 (visitado 01-02-2025).

[9] *Conjuntos de Datos - Datos Abiertos Ayto. Málaga.* URL: https://datosabiertos.malaga.eu/dataset (visitado 16-07-2025).

[10] Cristian Martín et al. «Kafka-ML: Connecting the Data Stream with ML/AI Frameworks». En: *Future Generation Computer Systems* 126 (2022), págs. 15-33.

[11] *Ingenieros Desarrollan 'Gemelos Digitales' de Última Generación, Más Accesibles y Versátiles - Universidad de Málaga.* URL: https://www.uma.es/sala-de-prensa/noticias/ingenieros-de-la-uma-desarrollan-gemelos-digitales-de-ultima-generacion-mas-accesibles-y-versatiles/ (visitado 16-07-2025).

[12] Roberto Minerva, Gyu Myoung Lee y Noël Crespi. «Digital Twin in the IoT Context: A Survey on Technical Features, Scenarios, and Architectural Models». En: *Proceedings of the IEEE* 108.10 (oct. de 2020), págs. 1785-1824. ISSN: 1558-2256. DOI: 10.1109/JPROC.2020.2998530. URL: https://ieeexplore.ieee.org/document/9120192 (visitado 24-08-2025).

[13] *What Is Docker?* Docker Documentation. URL: https://docs.docker.com/get-started/docker-overview/ (visitado 29-05-2025).

[14] *Debian – Reasons to Use Debian.* URL: https://www.debian.org/intro/why_debian (visitado 29-05-2025).

[15] *Introduction to Debian Linux.* GeeksforGeeks. URL: https://www.geeksforgeeks.org/introduction-to-debian-linux/ (visitado 29-05-2025).

[16] *Overview.* Kubernetes. URL: https://kubernetes.io/docs/concepts/overview/ (visitado 29-05-2025).

[17] *What Is Container Orchestration?* Red Hat. URL: https://www.redhat.com/en/topics/containers/what-is-container-orchestration (visitado 29-05-2025).

[18] *Helm.* URL: https://helm.sh/es/ (visitado 24-08-2025).

[19] Hussein Galal. *Introduction to K3s*. URL: https://www.suse.com/c/rancher_blog/introduction-to-k3s/ (visitado 29-05-2025).

[20] *Harbor*. URL: https://goharbor.io/ (visitado 18-07-2025).

[21] *VMware Workstation*. En: *Wikipedia*. 26 de mayo de 2025. URL: https://en.wikipedia.org/w/index.php?title=VMware_Workstation&oldid=1292382332 (visitado 29-05-2025).

[22] *Proxmox Virtual Environment*. Proxmox. URL: https://www.proxmox.com/en/products/proxmox-virtual-environment/overview (visitado 29-05-2025).

[23] *Features*. Proxmox. URL: https://www.proxmox.com/en/products/proxmox-virtual-environment/features (visitado 29-05-2025).

[24] Jason A. Donenfeld. *WireGuard: Fast, Modern, Secure VPN Tunnel*. URL: https://www.wireguard.com/ (visitado 18-07-2025).

[25] mijacobs. *¿Qué es Git? - Azure DevOps*. URL: https://learn.microsoft.com/es-es/devops/develop/git/what-is-git (visitado 15-06-2025).

[26] *Git*. En: *Wikipedia, la enciclopedia libre*. 5 de jun. de 2025. URL: https://es.wikipedia.org/w/index.php?title=Git&oldid=167794642 (visitado 15-06-2025).

[27] *Bash (Unix Shell)*. En: *Wikipedia*. 27 de mayo de 2025. URL: https://en.wikipedia.org/w/index.php?title=Bash_(Unix_shell)&oldid=1292506574 (visitado 29-05-2025).

[28] *What Is Python? Executive Summary*. Python.org. URL: https://www.python.org/doc/essays/blurb/ (visitado 29-05-2025).

[29] *Getting Started with Poetry | Better Stack Community*. URL: https://betterstack.com/community/guides/scaling-python/poetry-explained/ (visitado 15-06-2025).

[30] *FastAPI*. URL: https://fastapi.tiangolo.com/ (visitado 18-07-2025).

[31] *Postman API Platform: From Siloed to Synced | Postman*. URL: https://www.postman.com/api-platform/ (visitado 05-06-2025).

[32] Anonymous. *Eclipse Ditto | Projects.Eclipse.Org*. 6 de abr. de 2017. URL: https://projects.eclipse.org/projects/iot.ditto (visitado 05-06-2025).

[33] *MongoDB: Flexible Database for the Agile Age*. URL: https://www.oracle.com/es/database/mongodb/ (visitado 18-07-2025).

[34] *Eclipse Mosquitto*. Eclipse Mosquitto. 8 de ene. de 2018. URL: https://mosquitto.org/ (visitado 08-06-2025).

[35] *Telegraf/README.Md at Master · Influxdata/Telegraf · GitHub*. URL: https://github.com/influxdata/telegraf/blob/master/README.md (visitado 23-07-2025).

[36] Community / Product, Use Cases Aug 17 y 2022. *InfluxDB's Strengths and Use Cases Applied in Data Science*. InfluxData. Wed, 17 Aug 2022 07:00:00 +0000. URL: https://www.influxdata.com/blog/influxdb-strengths-use-cases-data-science/ (visitado 08-06-2025).

[37] *Time Series Database (TSDB) Guide | InfluxDB*. InfluxData. Tue, 29 Nov 2022 02:56:06 +0000. URL: https://www.influxdata.com/time-series-database/ (visitado 08-06-2025).

[38] *MySQL en detalle: guía para dominar esta potente base de datos*. URL: https://www.oracle.com/es/mysql/what-is-mysql/ (visitado 15-06-2025).

[39] J. Robles et al. «Scalability of Composite Digital Twins on OpenTwins Platform». En: HDL: 11705/JISBD/2024/70. URL: https://hdl.handle.net/11705/JISBD/2024/70.

[40] Shivang. *What Is Grafana? Why Use It? Everything You Should Know About It*. Scaleyourapp. 25 de ene. de 2019. URL: https://scaleyourapp.com/what-is-grafana-why-use-it-everything-you-should-know-about-it/ (visitado 08-06-2025).

[41] *Visual Studio Code*. En: *Wikipedia*. 22 de mayo de 2025. URL: https://en.wikipedia.org/w/index.php?title=Visual_Studio_Code&oldid=1291632062 (visitado 08-06-2025).

[42] *PyCharm: The Only Python IDE You Need*. JetBrains. URL: https://www.jetbrains.com/pycharm/ (visitado 08-06-2025).

[43] *DBeaver*. En: *Wikipedia, la enciclopedia libre*. 2 de abr. de 2025. URL: https://es.wikipedia.org/w/index.php?title=DBeaver&oldid=166579987 (visitado 24-08-2025).

[44] Atlassian. *Monorepos in Git | Atlassian Git Tutorial*. Atlassian. URL: https://www.atlassian.com/git/tutorials/monorepos (visitado 23-07-2025).

[45] *Designing Your Schema*. Time to Awesome. URL: https://awesome.influxdata.com/docs/part-2/designing-your-schema/ (visitado 29-07-2025).

[46] *Fórmula del semiverseno*. En: *Wikipedia, la enciclopedia libre*. 13 de jun. de 2025. URL: https://es.wikipedia.org/w/index.php?title=F%C3%B3rmula_del_semiverseno&oldid=167948933 (visitado 30-07-2025).

Apéndice A
Manual de Despliegue

A.1. Requisitos

Para realizar el despliegue del proyecto se recomienda separar el entorno de administración de comandos del clúster de Kubernetes donde se alojará la aplicación. Esta separación puede lograrse de las siguientes formas:

- Dos máquinas virtuales independientes.
- Un equipo físico como máquina principal y una máquina virtual para el clúster.
- Dos equipos físicos separados.

El objetivo es mantener independencia operativa entre la administración y el clúster, mejorando la seguridad y la gestión del sistema.

Para el clúster se utilizará un nodo único con k3s como implementación de Kubernetes. No se detalla la instalación completa de k3s[16], aunque en la mayoría de los casos bastará con ejecutar en una distribución GNU/Linux soportada:

```
curl -sfL https://get.k3s.io | sh -
```

En la máquina de administración deben estar instalados:

- Docker Engine
- kubectl
- helm

Además, deben clonarse los siguientes repositorios:

[16] https://docs.k3s.io/installation/configuration

- Archivos de despliegue Kubernetes
- EMTScraper
- EMTMetrics
- OpenTwins

A.2. Preparación

1. Configurar acceso por `ssh` desde la máquina de administración al nodo del clúster. Se recomienda autenticación con clave para reducir el uso de contraseñas.

2. Copiar el archivo `kubeconfig` desde el nodo del clúster a la máquina de administración:

```
mkdir ~/.kube
ssh USUARIO@IP_NODO sudo cat /etc/rancher/k3s/k3s.yaml >
~/.kube/config
```

sed

3. Modificar la IP del archivo copiado, reemplazando `127.0.0.1` por la IP real del nodo, por ejemplo con `sed`:

```
sed -i 's/127.0.0.1/IP_NODO/g' ~/.kube/config
```

4. (Opcional) Si se usará Harbor dentro del clúster como registro de contenedores, permitir su uso desde el cliente Docker de administración editando el archivo `/etc/docker/daemon.json`:

```
{
   "insecure-registries" : [ "IP_NODO:30002" ]
}
```

Listing 11: Configuración de Docker con registro inseguro

Posteriormente, se debe reiniciar el servicio de docker:

```
sudo systemctl daemon-reload
sudo systemctl restart docker
```

A.3. Despliegue

1. Construir las imágenes de **EMTScraper** y **EMTMetrics**. Para ello, acceder a la raíz de cada repositorio y ejecutar los comandos de construcción especificados en su respectivo README. Es fundamental construir las imágenes en la versión correspondiente a los archivos de despliegue utilizados, lo cual también está documentado en el README de `emtrack-k8s`. Por ejemplo, la versión `0.6.3` de `emtrack-k8s` requiere:

- EMTScraper Buslocation `v0.2.2`
- EMTScraper Routes `v0.2.3`
- EMTMetrics `v0.2.0`

2. Acceder a la raíz del repositorio `emtrack-k8s`. Aunque el README ya contiene los pasos de despliegue, a continuación se incluyen de forma explícita para mayor claridad:

1. Copiar el archivo `.env.template` y editarlo con los valores deseados.
2. (Opcional) Desplegar Harbor dentro del clúster:

   ```
   bash scripts/harbor-deploy.sh <ENV_FILE>
   ```

   También es posible usar un registro local o en la nube como alternativa. Una vez desplegado, autenticar usando las credenciales por defecto: usuario `admin`, contraseña `Harbor12345`.

   ```
   docker login IP_NODO:30002
   ```

3. Ejecutar:

   ```
   bash scripts/tfg-deploy.sh <ENV_FILE>
   ```

para desplegar todos los componentes.

4. Acceder a Grafana mediante navegador en la IP del clúster y puerto `30718`. Credenciales iniciales: usuario `admin`, contraseña `admin`. Una vez dentro, cambiar la contraseña al valor definido en `GRAFANA_API_PASSWORD`.

5. En el panel de Grafana, acceder a *Administration*, *Plugins*, buscar OpenTwins y pulsar `Enable`. Configurar:

   - **Eclipse Ditto URL:** `http://IP_NODO:30525`
   - **Eclipse Ditto Extended API URL:** `http://IP_NODO:30526`
   - **Ditto username:** `ditto`
   - **Ditto password:** `ditto`

6. Importar recursos de Ditto y Grafana, y ejecutar por primera vez los *cronjobs*:

```
bash scripts/import-resources.sh <ENV_FILE>
```

7. El sistema estará listo para usarse, los componentes de EMTScraper están trabajando en obtener datos de las APIs abiertas, por lo que en cuestión de minutos podrán usarse los paneles de control de Grafana con los datos obtenidos. Es recomendable refrescar Grafana en el navegador antes de su uso, pulsando las teclas CTRL + F5.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA