UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Grado en Ingeniería del Software

Herramienta de Análisis de los Accidentes de Tráfico en el Ámbito Urbano

Analysis Tool for Road Accidents at Urban Enviroments

Realizado por
Miguel Galdeano Rodríguez

Tutorizado por
Eduardo Guzmán De Los Riscos

Departamento
Lenguajes Y Ciencias De La Computación
UNIVERSIDAD DE MÁLAGA

MÁLAGA, junio de 2025

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA DEL SOFTWARE

HERRAMIENTA DE ANÁLISIS DE LOS ACCIDENTES DE TRÁFICO EN EL ÁMBITO URBANO

ANALYSIS TOOL FOR ROAD ACCIDENTS AT URBAN ENVIROMENTS

Realizado por
Miguel Galdeano Rodríguez

Tutorizado por
Eduardo Guzmán de los Riscos

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO DE 2025

Fecha defensa: Julio de 2025

Resumen

La siniestralidad vial en entornos urbanos sigue siendo un factor de riesgo elevado a día de hoy, especialmente en ciudades con alta densidad de tráfico y población como Madrid. Comprender las causas que provocan los accidentes de tráfico es fundamental para diseñar medidas de prevención eficaces y reducir su impacto. En este contexto, el análisis de grandes volúmenes de datos se presenta como una herramienta clave para identificar patrones, tendencias y relaciones entre los accidentes y variables externas que puedan influir en su aparición.

Este Trabajo Fin de Grado tiene como objetivo desarrollar una plataforma interactiva que permita explorar, filtrar y visualizar distintas variables que puedan afectar a los accidentes de tráfico. El propósito es facilitar a los organismos responsables una mejor comprensión de los factores asociados a la siniestralidad urbana, permitiéndoles tomar decisiones fundamentadas. Para ello, el sistema se alimenta de conjuntos de datos de acceso público (open data) proporcionados por el Ayuntamiento de Madrid.

El proyecto incluye un proceso completo de ingeniería de datos, que abarca desde la extracción hasta la ingesta de datos en un sistema accesible a través de un servidor intermedio. Este servidor se encarga de facilitar los datos al panel de control, el cual los presenta mediante diferentes visualizaciones interactivas tales como mapas y gráficos estadísticos. Además de los datos de accidentes, se han integrado fuentes de información complementarias, como condiciones meteorológicas, ubicación de centros educativos o días festivos, con el fin de enriquecer el contexto del análisis.

Palabras clave: Accidentes urbanos, ingeniería de datos, open data, dashboard, FastAPI, Dash.

Abstract

Road traffic accidents in urban environments remain a significant risk factor, especially in cities with high traffic density and population such as Madrid. Understanding the causes behind traffic accidents is essential for designing effective prevention strategies and reducing their impact. In this context, the analysis of large volumes of data emerges as a key tool for identifying patterns, trends, and relationships between accidents and external variables that may influence their occurrence.

This Final Degree Project aims to develop an interactive platform that allows the exploration, filtering, and visualization of various variables that may affect traffic accidents. The goal is to provide responsible authorities with a better understanding of the factors associated with urban road accidents, enabling more informed actions. The system relies on public open-data datasets published by the Madrid City Council.

The project includes a complete data engineering workflow, covering everything from data extraction to ingestion into a system accessible via an intermediate server. This server is responsible for delivering the data to a dashboard, which presents it through a variety of interactive visualizations such as maps and statistical charts. In addition to accident data, complementary information sources have been integrated — such as weather conditions, schools or official holidays — in order to enrich the analysis.

Keywords: Urban road accidents, data engineering, open data, dashboard, FastAPI, Dash

Índice

1. Introducción **9**

1.1. Motivación . . . 9

1.2. Objetivos . . . 10

1.3. Metodología . . . 11

1.4. Estructura de la memoria . . . 12

2. Tecnologías y herramientas utilizadas **15**

2.1. Base de datos . . . 15

2.1.1. PostgreSQL . . . 15

2.2. Tecnologías de Ingeniería de Datos . . . 15

2.2.1. Python . . . 15

2.2.2. Pandas . . . 16

2.2.3. Psycopg2 . . . 16

2.2.4. Astral . . . 16

2.3. Tecnologías back-end . . . 16

2.3.1. FastAPI . . . 16

2.3.2. SQLAlchemy . . . 16

2.4. Tecnologías front-end . . . 17

2.4.1. Dash (Plotly) . . . 17

2.5. Herramientas y software adicional . . . 17

2.5.1. Visual Studio Code . . . 17

2.5.2. Docker . . . 17

2.5.3. Visual Paradigm . . . 17

2.5.4. Draw.io . . . 18

2.5.5. Balsamiq . . . 18

2.5.6. pgAdmin4 . . . 18

2.5.7. Github . . . 18

3. Especificación y análisis **19**

3.1. Requisitos funcionales . . . 19

3.2. Requisitos no funcionales . . . 20

3.3. Casos de uso . . . 20

3.4. Perfiles de usuario potenciales . . . 28

3.5. Diagramas de secuencia . . . 28

3.6. Diagramas de flujo . . . 31

4. Diseño del sistema **35**

4.1. Diseño de la base de datos . . . 35

4.2. Diseño del backend . . . 38

4.3. Diseño del *frontend* . . . 40

5. Implementación y pruebas **43**

5.1. Ingeniería de datos . . . 43

5.1.1. Descarga de datos . . . 43

5.1.2. Transformación y limpieza . . . 45

5.1.3. Ingesta de datos . . . 50

5.2. Implementación del backend . . . 51

5.2.1. Módulo `models/` . . . 51

5.2.2. Módulo `services/` . . . 52

5.2.3. Módulo `routes/` . . . 54

5.3. Implementación del frontend . . . 56

5.3.1. Visualizador de mapas y estadísticas . . . 56

5.3.2. Detalles del accidente . . . 63

5.3.3. Aplicar filtros . . . 64

6. Despliegue y pruebas **67**

6.1. Despliegue . . . 67

6.2. Docker . . . 68

6.3. Pruebas . . . 70

7. Conclusiones y trabajos futuros **77**

7.1. Objetivos cumplidos 77

7.2. Dificultades encontradas 78

7.3. Posibles ampliaciones 79

Apéndice A. Manual de Instalación **85**

A.1. Requisitos previos 85

A.2. Instalación con Docker 85

A.3. Instalación manual 86

A.3.1. Crear entorno virtual e instalar dependencias 86

A.3.2. Configurar la base de datos PostgreSQL 86

A.3.3. Ejecutar el backend 86

A.3.4. Ejecutar el frontend 86

Apéndice B. Manual de Usuario **87**

B.1. Visualización del mapa y estadísticas 87

B.2. Aplicación de filtros 91

B.3. Detalles de accidente 92

1 Introducción

Este primer capítulo analiza la motivación detrás del proyecto, los objetivos que buscamos alcanzar, la metodología seguida para el desarrollo del proyecto y la estructura de la presente memoria.

1.1. Motivación

Los accidentes de tráfico en entornos urbanos representan un desafío persistente, no solo por sus consecuencias humanas, sino también por el impacto que generan en la dinámica de la ciudad. Cada siniestro puede desencadenar una serie de complicaciones: interrupción del tráfico, dificultades para que los equipos de emergencia accedan al lugar, e incluso nuevos accidentes derivados de frenazos bruscos o distracciones, como ocurre con el llamado efecto mirón [25] .

Uno de los tipos de accidentes más preocupantes en el entorno urbano son los atropellos. Este tipo de siniestro afecta principalmente a peatones, considerados usuarios vulnerables de la vía, y suponen una proporción significativa de la mortalidad en ciudad.
"Concretamente, un 37 % de las personas que pierden la vida por algún siniestro vial en las ciudades son peatones en el conjunto de la UE, proporción que asciende a un 44 % en el caso de España."[24]

Los datos abiertos (open-data) son una herramienta con un enorme potencial para el análisis y la toma de decisiones fundamentadas, ya que permiten el acceso libre a grandes volúmenes de información generada por organismos públicos. Aunque a simple vista sea algo sencillo, ya que estos organismos nos ofrecen los datos de manera pública, estos datos no siempre se encuentran estructurados o normalizados, lo que dificulta cualquier acción de análisis. Para

que estos datos sean realmente útiles, es necesario aplicar procesos previos de limpieza, transformación, integración y normalización, tareas que forman parte de la ingeniería de datos. Solo mediante estas etapas es posible convertir un conjunto de datos en bruto en información clara, coherente y explotable, capaz de generar valor real para usuarios técnicos, instituciones o ciudadanos.

Trabajar con datos reales, obtenidos a través de plataformas de open data, añade un valor significativo al análisis, ya que permite abordar problemas concretos desde una perspectiva cercana a la realidad de la ciudad. A diferencia de los datos sintéticos o simulados, los datos reales reflejan con precisión la complejidad, las inconsistencias y las particularidades propias del entorno urbano.

Aunque el Ayuntamiento de Madrid dispone de una herramienta oficial que permite visualizar de forma básica los datos abiertos sobre siniestralidad vial [12], esta solución presenta limitaciones en cuanto a personalización, profundidad analítica e integración de fuentes complementarias. Mediante el desarrollo de una plataforma propia, construida completamente desde cero se busca una solución más flexible y orientada a un análisis técnico, que pueda ser integradas por otros servicios externos de manera sencilla y eficiente. Además, se ha enriquecido el análisis mediante la incorporación de fuentes externas como datos meteorológicos, días festivos o ubicación de centros educativos, que permiten contextualizar los accidentes y explorar correlaciones que no pueden abordarse con la herramienta oficial. Esta visión integral enriquece el análisis, favorece la detección de patrones complejos y mejora la calidad de las conclusiones extraídas, lo cual resulta esencial para diseñar políticas públicas más eficaces en materia de seguridad vial.

1.2. Objetivos

El objetivo principal de este Trabajo Fin de Grado (TFG) es desarrollar una herramienta que permita visualizar y filtrar información obtenida del portal de datos abiertos del Ayuntamiento de Madrid [17]. La información principal serán los accidentes de tráfico. Además, esta información será complementada con otros conjuntos de datos disponibles en el mismo portal,

de manera que se pueda aumentar el contexto en el que se producen dichos accidentes.

Para ello se desarrollarán una serie de mapas interactivos y gráficos en los que se muestren distintos conjuntos de datos relacionados que pueden afectar a los accidentes de tráfico en entornos urbanos. Para ello será necesario realizar las siguientes fases:

- **Análisis y obtención de datos:** Realizar una búsqueda en profundidad de los distintos conjuntos de datos disponibles y proceder a la descarga de los más significativos.
- **Transformación y normalización de datos:** Transformar y normalizar los datos para poder desarrollar un sistema escalable y eficaz.
- **Inserción de datos:** Modelar y desarrollar una base de datos en la que poder cargar los datos para que puedan ser utilizados por el resto de capas del sistema.
- **API REST:** Construir un sistema servidor o *backend* que se encargue de facilitar los datos al panel interactivo.
- **Centro de control:** Desarrollar un panel de control con elementos interactivos tales como filtros, mapas y gráficos para visualizar e interpretar los datos.

1.3. Metodología

El desarrollo del proyecto se ha llevado a cabo siguiendo una metodología incremental iterativa, lo que ha permitido construir el sistema de forma progresiva, partiendo de una base funcional mínima e incorporando nuevas funcionalidades y mejoras a lo largo del tiempo. Este enfoque ha resultado especialmente útil en un contexto donde ha sido necesario integrar múltiples fuentes de datos, diseñar una estructura de base de datos coherente y desarrollar tanto el backend como el frontend de manera coordinada. A diferencia de una planificación rígida, esta metodología permite una mayor flexibilidad para adaptar el rumbo del desarrollo en función de los resultados obtenidos en cada fase. A medida que se avanzaba en la implementación, se podían detectar y resolver errores, replantear decisiones técnicas o mejorar ciertos aspectos de diseño sin comprometer el trabajo ya realizado.

En la primera iteración se analizaron los distintos conjuntos de datos disponibles, se descargaron y procesaron. El siguiente paso fue el diseño de la base de datos conforme a los conjuntos de datos obtenidos. Con el diseño de la base de datos se procedió a su montaje y carga de datos. Después de esto se desarrolló el servidor y por último el panel de visualización de los datos. El desarrollo del panel de visualización se realizó en varias iteraciones. En cada iteración se desarrollaba una funcionalidad y se probaba, a continuación pasábamos a la siguiente. Gracias a esto hemos podido realizar mejoras continuas que nos han permitido refinar el diseño e implementación favoreciendo la calidad del producto final.

1.4. Estructura de la memoria

A continuación vamos a describir la organización que seguiremos a lo largo del documento. Nombraremos los capítulos y una breve descripción sobre su contenido:

- **Capítulo 2. Tecnologías y herramientas utilizadas:** Se explican y detallan las tecnologías utilizadas para la creación y gestión de la base de datos, el desarrollo del servidor y el desarrollo de la interfaz gráfica. Además se comentan las herramientas utilizadas para los procesos adyacentes como creación del modelo relacional, bocetos, diagramas, casos de uso ...

- **Capítulo 3. Especificación y análisis:** En este capítulo se recoge una descripción detallada de los requisitos del sistema, tanto funcionales como no funcionales, así como los distintos modelos empleados para representar el comportamiento y la estructura del sistema.

- **Capítulo 4. Diseño del sistema:** Se desarrolla la arquitectura de la solución propuesta. Se detallan las decisiones de diseño tomadas, incluyendo el modelo relacional (E/R) de la base de datos, la arquitectura del sistema y el diseño de la interfaz interactiva. El objetivo es garantizar la coherencia entre los requisitos y el sistema implementado.

- **Capítulo 5. Implementación :** Detalla el proceso de desarrollo del sistema, abarcando desde la creación de la base de datos, el volcado de datos a la misma mediante scripts, el desarrollo del backend y frontend.

- **Capítulo 6. Despliegue y pruebas:** Se detallan los pasos seguidos para realizar el despliegue de la aplicación y los mecanismos de control de calidad realizados.
- **Capítulo 7. Conclusiones y trabajos futuros:** En este capítulo, se reflexiona acerca del proyecto realizado, las dificultades encontradas a lo largo del desarrollo y los objetivos alcanzados. Además, se comentan posibles ampliaciones al resultado actual.

2

Tecnologías y herramientas utilizadas

En este capítulo se centra en describir las tecnologías y herramientas utilizadas para el modelado, diseño y desarrollo del sistema.

2.1. Base de datos

2.1.1. PostgreSQL

Para el almacenamiento y gestión de los datos del proyecto se ha elegido PostgreSQL, un sistema de gestión de bases de datos relacional de código abierto ampliamente utilizado en entornos tanto académicos como profesionales. Esta elección se fundamenta en su fiabilidad, escalabilidad y compatibilidad con el estándar SQL, lo que lo convierte en una solución robusta y estable para trabajar con grandes volúmenes de datos.

2.2. Tecnologías de Ingeniería de Datos

2.2.1. Python

Python es un lenguaje de programación de alto nivel, interpretado y de propósito general, ampliamente utilizado en el desarrollo de aplicaciones web, análisis de datos, inteligencia artificial, automatización y scripting. Su integración con PostgreSQL y Dash ha permitido construir una solución eficiente y fácilmente mantenible.

2.2.2. Pandas

Pandas es una librería de Python diseñada para la manipulación y análisis de datos estructurados. Ha sido utilizada en este proyecto para limpiar, transformar y explorar los datasets de accidentes, gracias a su facilidad para manejar grandes volúmenes de datos en estructuras como DataFrame.

2.2.3. Psycopg2

Psycopg2 es un driver de bajo nivel que permite la conexión directa entre Python y bases de datos PostgreSQL. Su utilización en este proyecto ha sido clave para la ejecución de operaciones SQL específicas y tareas de carga masiva de datos, especialmente durante las etapas de ingesta y limpieza de los datasets históricos.

2.2.4. Astral

Astral es una librería de Python diseñada para calcular información astronómica, como la salida y puesta del sol, fases lunares y horas crepusculares en una ubicación y fecha determinadas. En el contexto de este proyecto, se ha utilizado para enriquecer los datos de accidentes con información temporal adicional, como la determinación del periodo del día en el que ocurrió el accidente

2.3. Tecnologías back-end

2.3.1. FastAPI

FastAPI es un microframework ligero de desarrollo web en Python, utilizado para construir el backend de la aplicación. Ha permitido crear una API REST modular y flexible, que actúa como intermediario entre la base de datos y el sistema de visualización interactiva.

2.3.2. SQLAlchemy

SQLAlchemy es una librería de Python que actúa como un ORM (Object-Relational Mapping), permitiendo interactuar con bases de datos relacionales mediante objetos Python. Su uso

ha facilitado la conexión con PostgreSQL y la gestión de consultas complejas sin necesidad de escribir directamente sentencias SQL.

2.4. Tecnologías front-end

2.4.1. Dash (Plotly)

Dash es un framework de Python desarrollado por Plotly que permite construir aplicaciones web interactivas orientadas a la visualización de datos. En este proyecto se ha utilizado para desarrollar el dashboard, integrando gráficos, mapas y filtros de forma intuitiva y dinámica, sin necesidad de escribir código HTML o JavaScript.

2.5. Herramientas y software adicional

2.5.1. Visual Studio Code

Visual Studio Code ha sido el editor de código principal utilizado durante el desarrollo del proyecto. Su ligereza, amplia disponibilidad de extensiones y soporte para múltiples lenguajes lo convierten en una herramienta versátil para escribir, depurar y organizar el código fuente de la aplicación.

2.5.2. Docker

Docker es una plataforma que permite crear, empaquetar y ejecutar aplicaciones en contenedores, los cuales incluyen todo lo necesario (código, librerías, dependencias) para funcionar de forma consistente en cualquier entorno. Gracias a esto, facilita el despliegue, la portabilidad y la escalabilidad de aplicaciones, eliminando problemas de compatibilidad entre entornos de desarrollo y producción.

2.5.3. Visual Paradigm

Visual Paradigm se ha utilizado como herramienta de modelado para diseñar diagramas UML, como casos de uso y modelos de base de datos. Su interfaz intuitiva ha permitido representar visualmente la estructura lógica y funcional del sistema antes de su implementación.

2.5.4. Draw.io

Draw.io es una herramienta de diseño que permite crear esquemas, diagramas de flujo y arquitecturas de sistema de manera visual e intuitiva. Se ha utilizado para crear diagramas de flujo de manera sencilla y directa.

2.5.5. Balsamiq

Balsamiq es una herramienta de prototipado rápido que permite crear wireframes de interfaces de usuario de forma sencilla y visual. En el presente proyecto, se ha utilizado durante las fases iniciales de diseño para crear bocetos del dashboard interactivo.

2.5.6. pgAdmin4

Se ha utilizado pgAdmin4 como herramienta administradora de PostgreSQL mediante la cual gestionaba la base de datos con una interfaz amigable lo que facilitaba la creación/modificación de las tablas, ejecución de consultas SQL y supervisión del funcionamiento de la base de datos.

2.5.7. Github

GitHub ha sido la plataforma de control de versiones empleada para alojar el repositorio del proyecto. Su integración con Git ha permitido gestionar el historial de cambios, facilitar el trabajo modular y mantener un flujo de desarrollo ordenado y trazable.

3

Especificación y análisis

3.1. Requisitos funcionales

Los requisitos funcionales del proyecto se enumeran a continuación:

- **RF-1. Carga de datos:** el sistema debe permitir cargar conjuntos de datos de accidentes en formato CSV.
  - **RF-1.1. Obtención de datos:** los datos deben poder obtenerse desde fuentes abiertas externas.
  - **RF-1.2. Transformación de datos:** los datos deben ser limpiados y transformados al formato estructurado de la base de datos.
  - **RF-1.3. Envío a base de datos:** los datos transformados deben insertarse correctamente en la base de datos PostgreSQL.
- **RF-2. Visualización de datos y estadísticas:** la aplicación debe permitir visualizar estadísticas globales mediante gráficos y paneles.
- **RF-3. Visualización de detalles de accidentes:** el usuario debe poder consultar información detallada de accidentes individuales seleccionados desde el mapa.
- **RF-4. Filtrado combinado de datos:** el sistema debe permitir aplicar múltiples filtros combinados (distrito, tipo de accidente, rango de edad, etc.).
- **RF-5. Interacción con el mapa:** los usuarios deben poder interactuar con el mapa para seleccionar zonas y visualizar los accidentes asociados.

- **RF-6. Actualización incremental de datos:** el sistema debe permitir añadir nuevos registros sin duplicar ni sobrescribir datos previos.

3.2. Requisitos no funcionales

- **RNF-1. Compatibilidad:** la aplicación debe funcionar correctamente en los principales navegadores modernos.
- **RNF-2. Escalabilidad:** el sistema debe estar preparado para incorporar nuevos años de datos o nuevas fuentes sin rediseñar su arquitectura.
- **RNF-3. Tolerancia a errores:** el sistema debe ser capaz de gestionar entradas incompletas o inconsistentes sin interrumpir su funcionamiento.
- **RNF-4. Eficiencia:** las consultas deben resolverse con un tiempo de respuesta razonable, incluso con grandes volúmenes de datos.
- **RNF-5. Usabilidad:** la interfaz debe ser clara, accesible y comprensible para usuarios no técnicos.
- **RNF-6. Seguridad:** el sistema no debe exponer información personal ni comprometer la privacidad de los datos.

3.3. Casos de uso

Los casos de uso permiten describir de forma estructurada las interacciones entre los usuarios del sistema y las funcionalidades que ofrece la aplicación. Se trata de una herramienta fundamental dentro del análisis funcional, ya que ayuda a delimitar el comportamiento esperado del sistema desde la perspectiva del usuario final.

En este proyecto, se han identificado y documentado varios casos de uso que cubren tanto las acciones realizadas por el usuario general (consultas, visualización y filtrado de datos) como las funciones propias del administrador (proceso de ETL, extracción, transformación, carga, y actualización del conjunto de datos). Cada caso de uso se ha descrito siguiendo una

estructura formal que incluye: nombre, actor, descripción, precondiciones, flujo principal, posibles escenarios alternativos y poscondiciones.
A través de esta representación se puede obtener una visión clara del funcionamiento funcional de la aplicación, sirviendo además como base para el diseño de la interfaz y la implementación posterior. La definición precisa de estos casos también permite validar funcionalidades, y planificar pruebas.

Debemos puntualizar que, aunque la transformación y limpieza de datos, así como la ingesta de datos parezcan una copia de la actualización de datos, realmente son procesos completamente diferentes. Los dos primeros se refieren al proceso inicial de estructuración del proyecto en el que se obtenían y creaban las estructuras por primera vez. En ese caso se debían realizar procesos manuales para transformar y normalizar de manera correcta los datos. El proceso de actualización, aparte de ser automático (se centraliza todo desde un único *script*), no requiere de la intervención del administrador siempre y cuando el conjunto de datos mantenga la estructura actual.

Nombre	Transformar y limpiar datos
Actores	Administrador
Descripción	Permite procesar un archivo de datos brutos para limpiarlo, normalizarlo y prepararlo para su inserción en la base de datos.
Precondiciones	El archivo CSV con los datos debe estar disponible en el directorio correspondiente.
Escenario Principal	1. El administrador ejecuta el *script* de transformación. 2. El sistema elimina registros vacíos o corruptos. 3. Se indexan los valores categóricos. 4. Se genera una salida estructurada lista para insertar.
Escenario Alternativo	Si el archivo contiene errores de estructura o no se encuentra, el sistema muestra un mensaje de error y detiene el proceso.
Postcondiciones	El archivo de datos queda limpio y estructurado, listo para ser insertado en la base de datos.

Cuadro 1: Caso de uso: Proceso de transformación de datos

Nombre	Ingestar datos
Actores	Administrador
Descripción	Permite cargar los datos previamente transformados en las tablas del modelo relacional definido en PostgreSQL.
Precondiciones	La estructura de la base de datos debe haber sido creada.
Escenario Principal	1. El administrador lanza el *script* de carga. 2. El sistema abre conexión con PostgreSQL. 3. Se insertan los registros en sus tablas correspondientes. 4. Se cierra la conexión y se confirma la operación.
Escenario Alternativo	Si la conexión falla o los datos no cumplen las restricciones del modelo, se genera un error y no se insertan registros.
Postcondiciones	La base de datos queda actualizada con los nuevos datos y lista para ser consultada.

Cuadro 2: Caso de uso: Proceso de ingesta de datos

Nombre	Actualizar datos
Actores	Administrador
Descripción	Permite cargar los nuevos datos disponibles con solo ejecutar un *script*.
Precondiciones	Tenemos la URL de dónde se encuentra el archivo CSV.
Escenario Principal	1. El administrador ejecuta el *script* . 2. El sistema comprueba cuales son los nuevos accidentes. 3. El sistema filtra, normaliza y transforma los datos. 4. El sistema inserta los accidentes 5. El sistema inserta los heridos de cada accidente
Escenario Alternativo	Si no hay nuevos accidentes disponibles o el archivo no se encuentra.
Postcondiciones	La base de datos queda actualizada con los últimos accidentes disponibles

Cuadro 3: Caso de uso: Proceso de actualización de datos

Nombre	Visualizar mapa de accidentes
Actores	Usuario
Descripción	Permite al usuario consultar la distribución espacial de los accidentes urbanos a través de un mapa interactivo
Precondiciones	El sistema debe tener acceso a la base de datos con los accidentes cargados y procesados.
Escenario Principal	1. El usuario accede al panel de visualización. 2. El sistema muestra el mapa con los accidentes disponibles.
Escenario Alternativo	No hay escenario alternativo.
Postcondiciones	El usuario puede navegar por el sistema viendo la información del panel.

Cuadro 4: Caso de uso: Visualizar mapa de accidentes

Nombre	Aplicar filtros
Actores	Usuario
Descripción	Permite al usuario utilizar varios filtros de manera combinada sobre distintos aspectos de los accidentes.
Precondiciones	El sistema debe haber cargado los datos exitosamente.
Escenario Principal	1. El usuario accede al dashboard. 2. El usuario despliega el panel de filtros. 3. El sistema muestra los diversos filtros disponibles. 4. El usuario selecciona distintos filtros. 5. El sistema aplica los filtros. 6. El sistema muestra los datos filtrados
Escenario Alternativo	No hay escenario alternativo.
Postcondiciones	Los mapas y estadísticas muestran los datos coincidentes con los filtros aplicados.

Cuadro 5: Caso de uso: Aplicar filtros

Nombre	Consultar detalles de un accidente
Actores	Usuario
Descripción	Permite al usuario acceder a información detallada de un accidente concreto seleccionado en el mapa.
Precondiciones	El mapa muestra al menos un accidente.
Escenario Principal	1. El usuario selecciona un accidente en el mapa. 2. El sistema consulta los detalles del accidente. 3. El sistema abre un panel con información acerca del accidente y de los heridos implicados en el mismo.
Escenario Alternativo	No hay escenario alternativo.
Postcondiciones	El usuario puede visualizar la información detallada y cerrar el panel si lo considera oportuno.

Cuadro 6: Caso de uso: Consultar detalles de un accidente

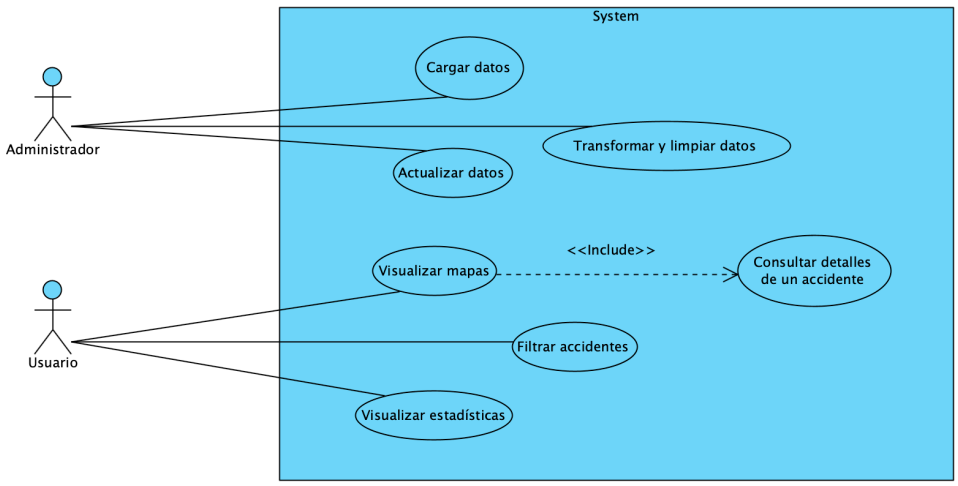

Figura 1: Diagrama de casos de uso

3.4. Perfiles de usuario potenciales

Aunque el sistema no contempla diferentes roles de usuario a nivel funcional, puede ser de interés para diversos perfiles profesionales. Entre los cuales podemos destacar:

- **Dirección General de Gestión y Vigilancia de la Circulación:** Es el organismo encargado de controlar el tráfico en Madrid. Dicho organismo podrá utilizar la herramienta para implantar límites de velocidad variables en las carreteras, lo que adaptaría el límite de velocidad en función del día de la semana, hora del día o meteorología. Cataluña ha comenzado ya las pruebas de este sistema en su red de carreteras, ver en [21].

- **Autoescuelas:** Complementar la formación que se ofrece en las autoescuelas a los jóvenes con esta herramienta puede ser un punto de inflexión en la concienciación sobre la siniestralidad vial y sus consecuencias. Además, también puede ser implantada en los cursos de recuperación de puntos o de recuperación de carnet.

- **Responsables de la infraestructura vial:** El uso de esta herramienta por parte de los responsables de las infraestructuras públicas los dotará de un mayor conocimiento y contexto a la hora de realizar estudios para ubicar nuevos centros educativos y hospitales.

- **Investigadores:** Tanto investigadores como estudiantes podrán utilizar la herramienta para realizar estudios y análisis sobre la siniestralidad vial en entornos urbanos.

3.5. Diagramas de secuencia

Los diagramas de secuencia se han utilizado para representar el comportamiento dinámico del sistema ante acciones concretas del usuario. A diferencia de los diagramas de casos de uso, que muestran qué funcionalidades ofrece el sistema, los diagramas de secuencia describen cómo se comunican internamente los distintos módulos para ejecutar esas funcionalidades, indicando el orden de las interacciones y los mensajes intercambiados.

Aunque el sistema contempla diversas funcionalidades, en este trabajo se han incluido los diagramas de secuencia que representan aquellas interacciones relacionadas con el usuario: la

visualización de los mapas y estadísticas, la aplicación de filtros y la consulta de los detalles de un accidente.

La razón de esta selección radica en que el núcleo funcional y técnico de la herramienta se encuentra en el proceso de ingeniería de datos, que comprende la obtención, transformación y carga de grandes volúmenes de información desde fuentes abiertas. Este proceso constituye la base sobre la que se apoya el resto del sistema, y su correcta definición, automatización y validación es lo que permite que las visualizaciones y consultas posteriores sean posibles y eficaces.
Por ello, se ha optado por representar ese flujo técnico mediante diagramas de flujo específicos del proceso de tratamiento de datos, reservando los diagramas de secuencia solo para aquellos casos de uso que implican una interacción significativa entre el usuario y los distintos componentes del sistema.

- **Visualizar mapas y estadísticas:** El diagrama de secuencia desarrollado en la figura 2 define la interacción entre el usuario y los distintos elementos del sistema para visualizar los distintos mapas y estadísticas sobre los accidentes.

- **Detalle de accidente:** En la figura 3 se describen las interacciones entre el usuario y los distintos elementos del sistema para visualizar los detalles de un accidente.

- **Aplicar filtros:** La figura 4 especifica las interacciones necesarias para filtrar los datos.

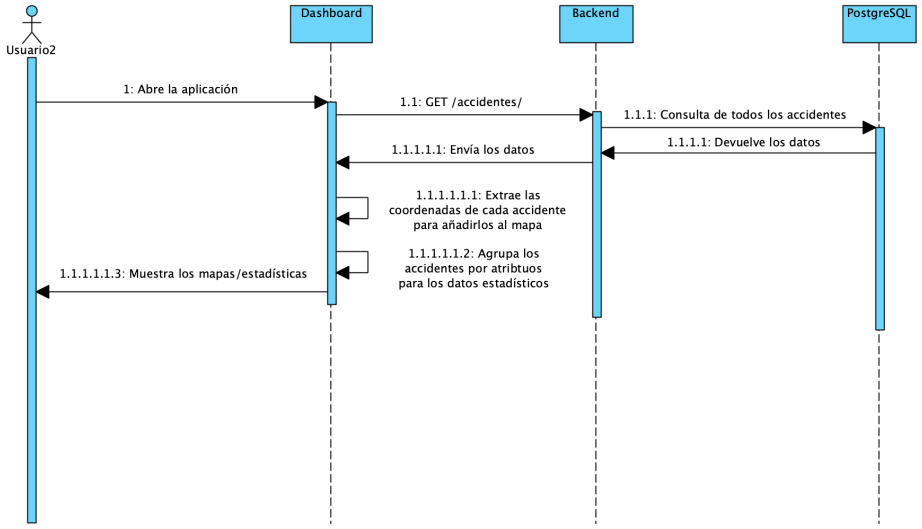

Figura 2: Diagrama de secuencia: Visualizar mapas/estadísticas

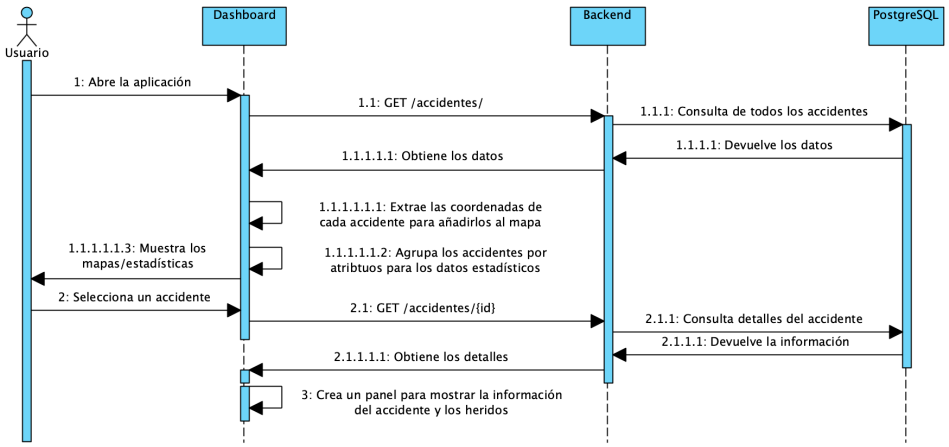

Figura 3: Diagrama de secuencia: Detalle de accidente

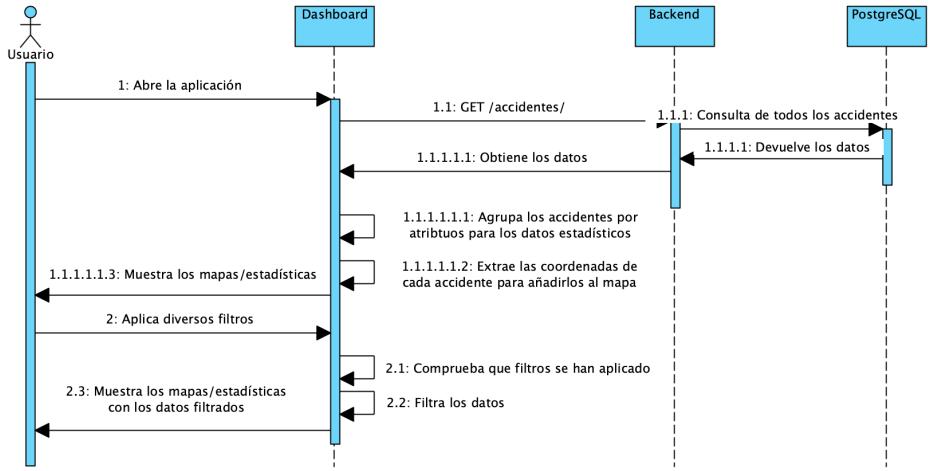

Figura 4: Diagrama de secuencia: Aplicar filtros

3.6. Diagramas de flujo

Los diagramas de flujo han sido utilizados para representar gráficamente los distintos procesos implicados en el sistema, especialmente los procedimientos ETL aplicados a los datos de accidentes de tráfico. Este tipo de representación visual ha resultado fundamental para comprender y comunicar de manera clara la lógica de procesamiento seguida en cada etapa del proyecto.

Transformación y carga de dimensiones: En este diagrama de flujo 5 vemos los pasos que siguen los procesos encargados de leer los valores dimensionales, extraerlos y cargarlos en la base de datos. Las tablas dimensionales son:

- Tipo de vehículo
- Tipo de accidente
- Género
- Rango de edad
- Lesividad
- Tipo de persona (Pasajero, conductor, peatón)

- Momento del día

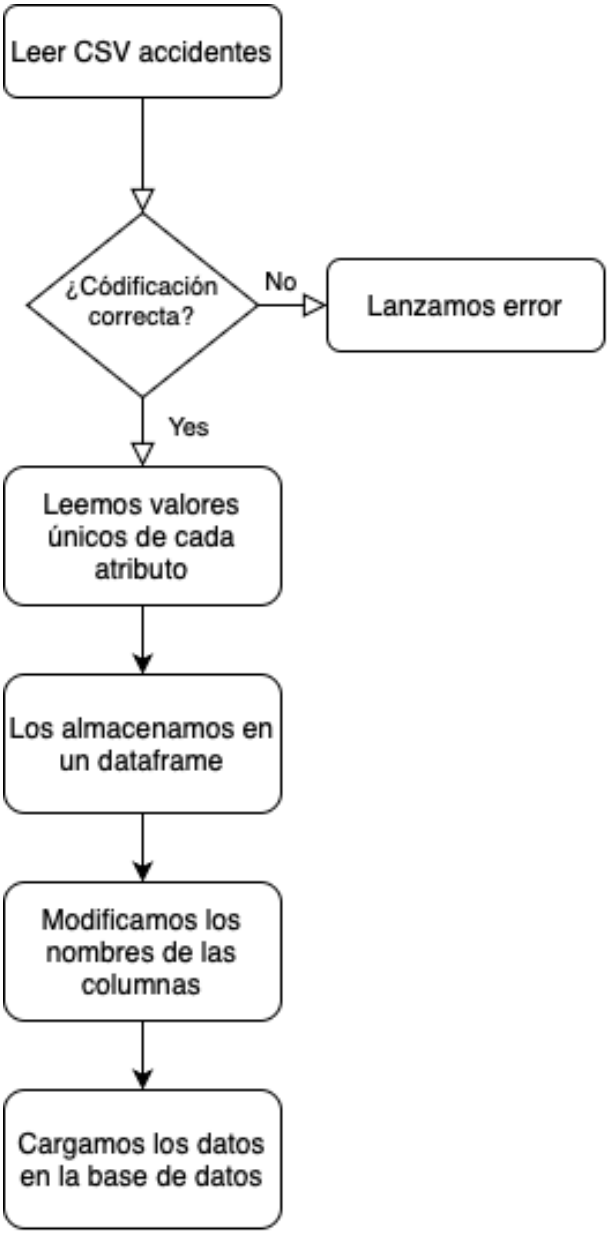

Figura 5: Diagrama de flujo: Transformación y carga de dimensiones

Transformación y carga de accidentes: En la figura 6 describimos mediante un diagrama de flujo el proceso que sigue el script encargado de leer el CSV con los datos de los accidentes y las transformaciones que realiza para posteriormente cargar el dataset en la base de datos.

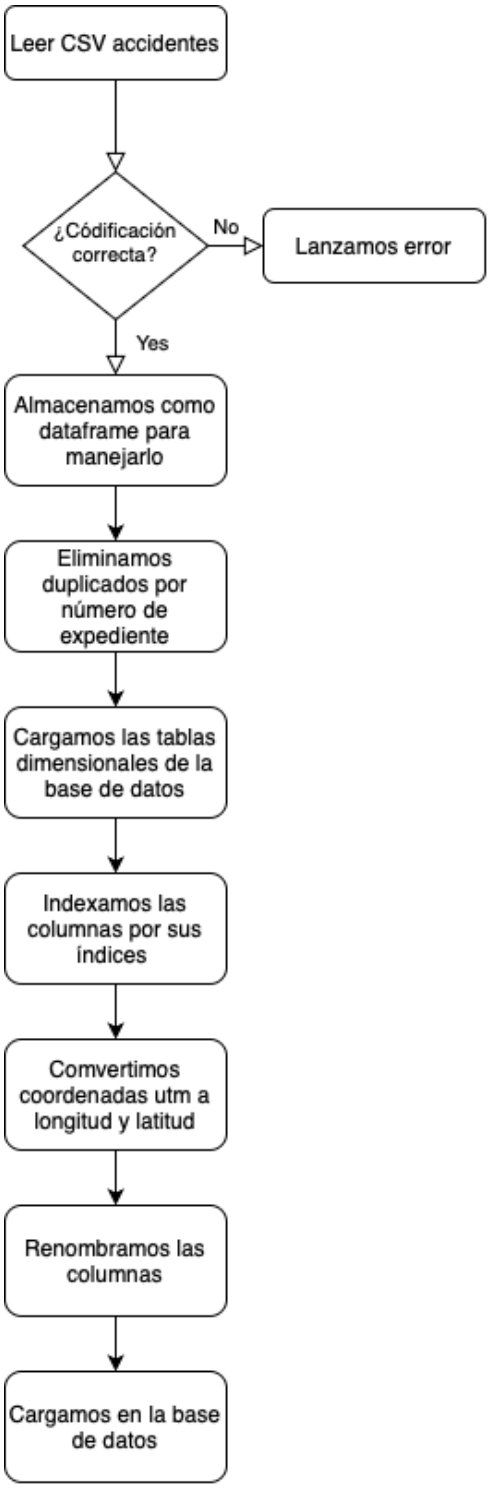

Figura 6: Diagrama de flujo: Transformación y carga de accidentes

Actualización de datos En la figura 7 describimos mediante un diagrama de flujo el proceso que sigue el script encargado de actualizar la base de datos con los nuevos accidentes. Realmente es un proceso similar al de la transformación y carga de accidentes pero en este

caso están todos los procesos unificados en un solo script, de forma que se puede realizar la actualización automática de los datos con solo ejecutarlo, sin necesidad de realizar operaciones adicionales.

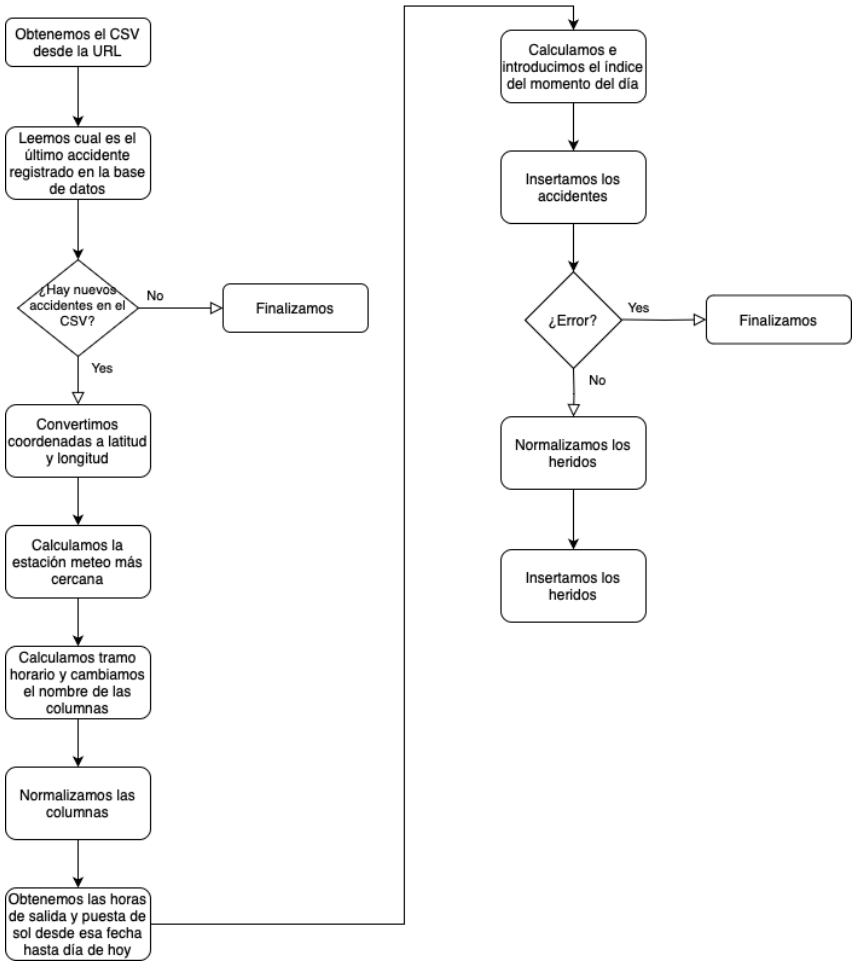

Figura 7: Diagrama de flujo: Actualización de datos

4

Diseño del sistema

El sistema desarrollado sigue una arquitectura cliente-servidor, siguiendo un modelo fullstack, tal y como podemos ver en la Figura 8, siendo esta una de las más extendidas en el diseño de aplicaciones web modernas. Los componentes del sistema se dividen en dos partes principales:

Cliente (frontend): Encargado de la interacción con el usuario y la presentación de los datos.

Servidor (backend): Responsable de procesar las solicitudes, gestionar la lógica de negocio y acceder a la base de datos.

Gracias a esta separación, el sistema puede funcionar de manera independiente. La comunicación se realiza mediante solicitudes HTTP, el cliente envía peticiones al servidor para solicitar la información procesada. El servidor se encarga de realizar la solicitud a la base de datos, realizar la lógica necesaria y devolver al cliente los datos solicitados.

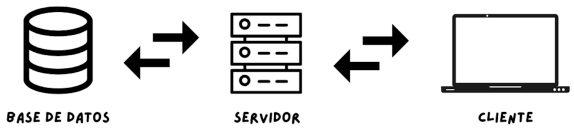

Figura 8: Arquitectura cliente-servidor. Fuente: Elaboración propia a partir de [4]

4.1. Diseño de la base de datos

La estructura de la base de datos es un aspecto realmente importante para poder realizar las consultas de los datos a gran escala de manera rápida y eficiente. Además, la actualización de datos de accidentes es constante, por lo que diseñar una base de datos escalable es crucial para seguir añadiendo información sin comprometer el rendimiento.

Si queremos mantener una gran cantidad de datos y gestionarlos de manera eficiente es necesario utilizar un enfoque relacional, estructurado entorno a distintas entidades de manera que podamos relacionar los datos sin tener que duplicarlos.

Con este objetivo, se han creado tablas dimensionales (para el género del accidentado, su rango de edad, el tipo de vehículo, gravedad del accidente y distrito entre otros) que permiten trabajar con índices en lugar de cadenas de texto, facilitando la eficiencia de las consultas y la consistencia en la representación de los datos.
Este modelo nos permite obtener todos los valores que puede tomar cada atributo desde una sola consulta sin tener que recorrer el conjunto principal y buscar los valores únicos, útil para crear distintos filtros en el dashboard. Este proceso se conoce como normalización de la base de datos.

Tras analizar el conjunto de datos a utilizar [7], se observa que cada fila detalla la información de una persona implicada. Como consecuencia, la información relativa al accidente se muestra duplicada tantas veces como implicados hay en el mismo, generando una redundancia innecesaria.
Para evitar esta duplicidad, seguimos normalizando el modelo mediante la creación de dos tablas diferenciadas: una que almacena la información general del accidente, denominada `accidente`, y otra que recoge los atributos de cada persona implicada, denominada `perfil_persona`.

La relación entre accidentes y personas se modela como una relación de tipo muchos a muchos (M:N), ya que un accidente puede involucrar a múltiples personas, y, conceptualmente, una misma persona podría estar implicada en más de un accidente. Para representar esta relación, hemos introducido una tabla intermedia denominada `heridoaccidente`, que enlaza ambas entidades y contiene información específica sobre la implicación de cada perfil en un accidente concreto (como el tipo de herido, la lesividad o el resultado de alcohol y drogas).

Esta estructura nos permite reducir la redundancia, mejorar la integridad de los datos y reflejar de forma precisa la naturaleza relacional del conjunto. Véase la Figura 9.

Además, por privacidad de los heridos, amparados por el Reglamento General de Protección de Datos (RGPD)[23] de la Unión Europea y a la Ley Orgánica 3/2018 (LOPDGDD) [22], no se muestra información personal en el conjunto de datos. Esto hace que los heridos sean clasificados de forma genérica, de forma que un herido no es una persona física sino un perfil de persona. De igual manera, la información personal no es relevante para el análisis de los accidentes.

Para poder llegar a hacer un análisis más exhaustivo de los datos, se han incluido los siguientes datos de fuentes externas :

- **Horas de salida y puesta de sol:** Esto nos permite clasificar los accidentes según hayan ocurrido de día o de noche. Es útil ya que un accidente ocurrido a las 19:00 no puede clasificarse como día o noche, dependerá de la fecha en la que ocurrió. En un principio se barajó almacenar en una tabla las fechas desde el 01-01-2019 (fecha en la que comienza el conjunto de datos de accidentes) y para cada día almacenar la hora de salida y puesta de sol. Tras analizarlo se optó por almacenar directamente en la tabla de accidentes si éste había ocurrido de día, de noche, al amanecer o al anochecer. En el siguiente capítulo se detalla el proceso.

- **Día de la semana y festivos:** Esta información es bastante útil para ver la incidencia del fin de semana o días festivos y analizar cómo varían las horas de siniestralidad. Es información que el dataset original no ofrece y se ha obtenido desde [3].

- **Climatología:** Información de gran utilidad para aumentar el contexto del análisis de accidentes. El dataset original ofrece algo de información meteorológica pero se ha complementado con información más precisa obtenida del mismo portal de datos abiertos [8].

- **Ubicación de elementos de seguridad vial:** En este apartado se recogen los elementos disuasorios e informativos, incluidos radares, paneles de información variable, cámaras de vigilancia y cruces con cámara en los semáforos que multan si se cruza en rojo. Se han obtenido del portal de datos abiertos [17]

- **Ubicación de infraestructuras de asistencia sanitaria:** Los centros de asistencia

sanitaria son un factor clave a la hora de asistir a los heridos. Si se sitúan muy lejos del lugar del accidente, el tiempo de respuesta elevado supone una mayor probabilidad de complicaciones. Se han obtenido en [5]

- **Ubicación de centros educativos:** Las horas punta de entradas y salidas de los centros educativos pueden afectar directamente a una mayor tasa de accidentes en la zona debido a las retenciones ocasionadas. Se han obtenido en [6]

Las tablas auxiliares como `radar`, `panel_info_variable`, `centro_educativo` y `hospital` no están relacionadas directamente con la tabla principal `accidente`, ya que afirmar la implicación de un centro educativo o un hospital en un accidente debe realizarse con un mayor contexto. Esta información almacenada será de utilidad para realizar un análisis visual. También se ha prestado especial atención a la inclusión de coordenadas geográficas normalizadas (latitud y longitud), con el objetivo de habilitar su uso en mapas interactivos. El modelo ha sido diseñado para integrarse de forma directa con la lógica de backend y la capa de visualización, respetando los principios de modularidad y escalabilidad del sistema. Todos estos detalles, tienen como resultado el modelo E/R de la Figura 9.

4.2. Diseño del backend

El backend de la aplicación se ha implementado en Python siguiendo una arquitectura modular por capas. Esta estructura ha sido elegida para separar claramente las responsabilidades de cada componente del sistema, facilitar su mantenimiento y permitir una evolución ordenada del proyecto a medida que crecen los requisitos funcionales.

Cada módulo del sistema cumple una función específica dentro de la lógica de negocio:

- `utils/` aquí se definen todos los scripts necesarios para el proceso de Ingeniería de Datos.

- `models/` contiene las definiciones de los modelos que representan las tablas de la base de datos PostgreSQL. Estos modelos son utilizados por el ORM, lo que permite trabajar con los datos a través de objetos Python, mejorando la legibilidad del código y evitando errores típicos de las consultas SQL manuales.

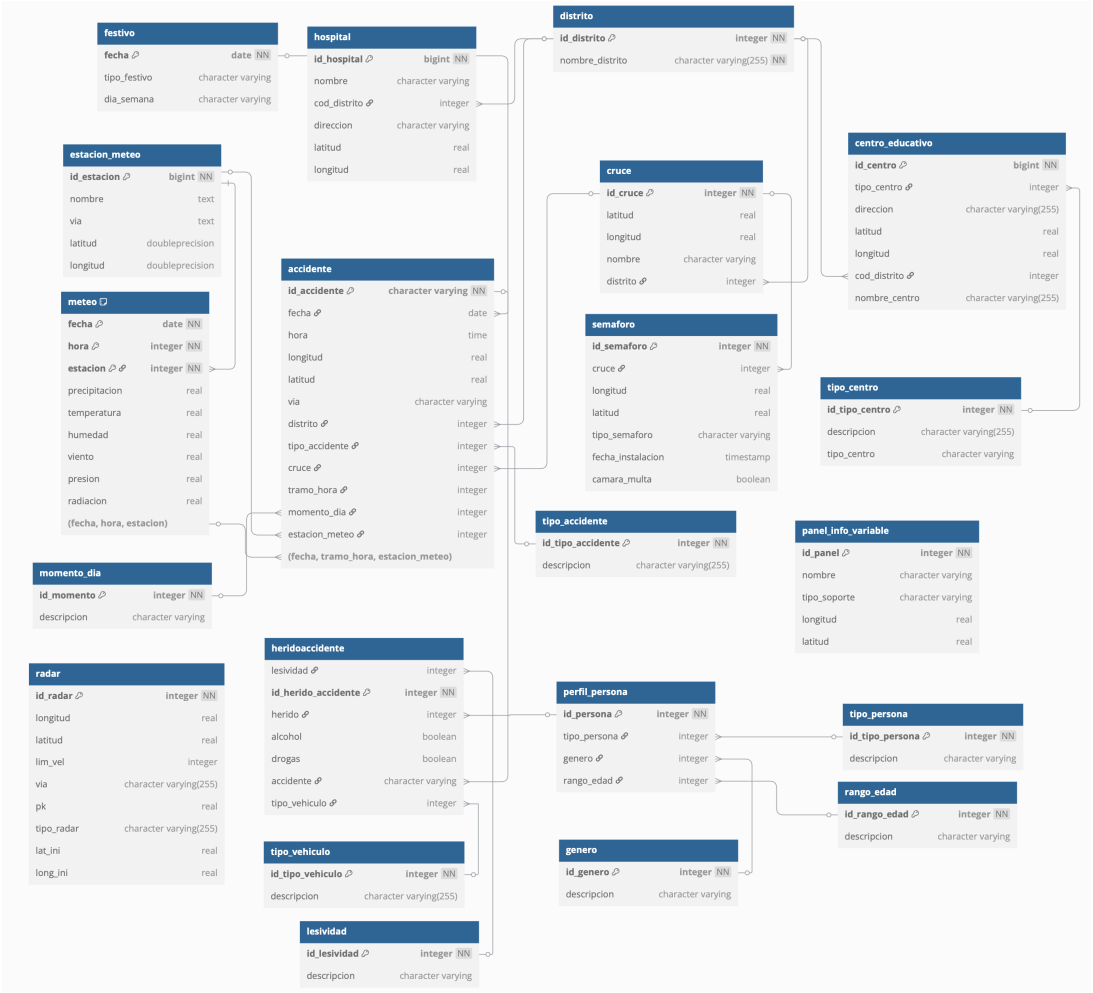

Figura 9: Diagrama E/R

- `routes/` implementa los distintos endpoints de la API, organizados por áreas funcionales (accidentes, heridos, datos complementarios, etc.). Cada archivo en este módulo expone rutas que responden a peticiones HTTP, delegando la lógica de procesamiento a los servicios correspondientes.
- `services/` constituye la capa intermedia encargada de realizar las consultas a la base de datos y de formatear los resultados para su envío al cliente. Esta capa actúa como puente entre los endpoints y la base de datos, encapsulando la lógica necesaria para transformar los datos en estructuras JSON preparadas para ser consumidas por el frontend. Esta separación permite mantener el código de las rutas limpio y enfocado únicamente en la gestión de peticiones y respuestas.
- `config.py` y `database.py` gestionan la configuración global de la aplicación y estable-

cen la conexión con la base de datos, proporcionando un entorno flexible y desacoplado del resto del código.

- `main.py` es el archivo encargado de orquestar el resto de módulos y desplegar la API Rest.

Esta organización modular ha sido seleccionada por su capacidad para escalar de manera ordenada, facilitar la trazabilidad del código y adaptarse a buenas prácticas de desarrollo backend. Cada capa está diseñada para cumplir un propósito concreto, lo que mejora tanto la claridad como la mantenibilidad del sistema. Asimismo, esta estructura prepara el proyecto para posibles extensiones futuras, como la incorporación de nuevas rutas, nuevas fuentes de datos o mecanismos avanzados de autenticación, sin que ello implique una reestructuración completa del backend.

4.3. Diseño del *frontend*

El diseño del frontend se ha estructurado en una única vista principal organizada de forma vertical y desplazable, con el objetivo de mantener una interfaz coherente, accesible y fluida para el usuario. Esta decisión permite integrar todos los elementos clave (filtros, visualizaciones y detalles) en un único espacio, evitando saltos entre pantallas y facilitando una experiencia de navegación más natural.

En la parte superior de la pantalla se sitúa el conjunto de filtros dinámicos, que permiten acotar la visualización por variables como el distrito, la franja horaria, el tipo de persona, la lesividad o el día de la semana, entre otros. Estos filtros interactúan directamente con el resto de componentes de la interfaz. Para optimizar el espacio visual disponible, se ha diseñado un botón que permita ocultar y mostrar los filtros.
Justo debajo, se encuentra el mapa interactivo, que muestra geográficamente los accidentes registrados. El usuario puede aplicar filtros y visualizar cómo se actualiza la distribución espacial en tiempo real. Además, al seleccionar un accidente del mapa, se abre un panel adicional que muestra los detalles del accidente seleccionado: ubicación, hora, fecha, día de la semana y características de las personas implicadas, entre otros.
Finalmente, en la parte inferior, se presentan diversas visualizaciones estadísticas agregadas

(gráficas de barras, circulares, indicadores numéricos), que ofrecen una visión global del conjunto de datos filtrados.

Este diseño vertical y continuo responde a una filosofía de interfaz centrada en el usuario, donde la información se despliega de forma jerárquica y lógica conforme se avanza en la exploración. Los bocetos presentados a continuación son la base conceptual para definir dicha estructura:

- Figura 10: Boceto con filtros y mapa interactivo

- Figura 11: Boceto con panel de detalles de accidente

- Figura 12: Boceto con estadísticas generales

Figura 10: Boceto filtros y mapa

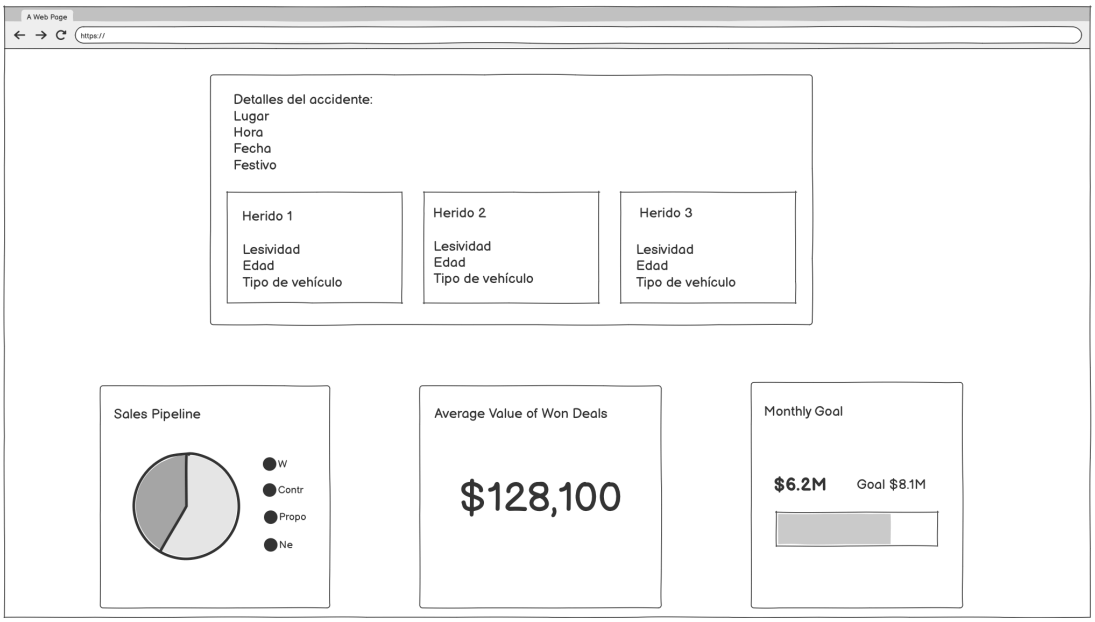

Figura 11: Boceto detalles del accidente

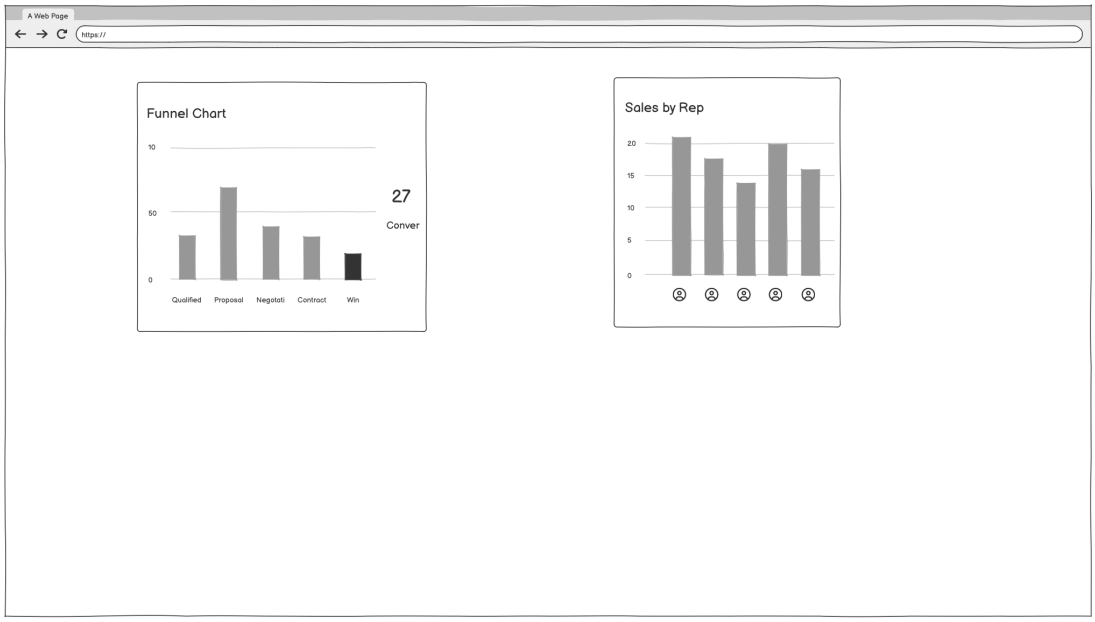

Figura 12: Boceto estadísticas

5

Implementación y pruebas

La implementación de la solución propuesta se ha estructurado en tres bloques funcionales principales: el proceso ETL, el backend y el dashboard interactivo. Cada uno de estos componentes ha sido desarrollado en Python utilizando librerías especializadas para la gestión de datos, la conexión con bases de datos y la construcción de interfaces web. En esta sección se detallan los pasos seguidos en la implementación de cada uno de ellos, así como las decisiones técnicas adoptadas para garantizar el correcto funcionamiento del sistema.

5.1. Ingeniería de datos

Ya hemos mencionado los pasos y procedimientos que se han seguido a la hora de diseñar la base de datos. Una vez definida la estructura es hora de obtener, transformar y limpiar los datos para que se puedan introducir en la base de datos.

5.1.1. Descarga de datos

El primer paso para dotar de contenido a la base de datos es descargar los archivos que contienen los datos en bruto. Debemos acceder al portal de datos abiertos del Ayuntamiento de Madrid y buscar el subdirectorio que contiene los archivos de accidentes de tráfico. Una vez aquí vemos que tenemos disponible para descargar los archivos desde 2010 hasta 2025. Para los datos de 2010 a 2018 se utilizaba una estructura distinta a los de 2019 en adelante. Además, para los más antiguos solo hay registros si el accidente incluye heridos o daños al patrimonio público. En nuestro caso hemos elegido los archivos que siguen la estructura actual. De esta forma podemos mantener la actualización de datos recurrente sin necesidad de realizar grandes reformas estructurales. Estos archivos se descargan como CSV y se almacenan en el

subdirectorio correspondiente, desde donde serán accedidos para su transformación y normalización.
Los conjuntos de datos de hospitales [5] y centros educativos [6] han sido accedidos desde sus respectivas direcciones dentro del portal y almacenados de forma local. Este proceso se ha repetido también con los conjuntos de datos de localización de radares, semáforos,cruces y paneles de información variable.

La información de meteorología se ha tratado de obtener de la API de AEMET [1]. Primero se obtiene la *API-KEY*, una clave que debe ser incluida en el encabezado de la solicitud HTTP. Esta clave permite identificar cada una de las solicitudes, de esta forma se pueden evitar abusos del servicio y se asegura la autenticación previa. Tras realizar las consultas, estas han fallado de forma reiterada. Entendemos que el problema es el volumen de datos solicitados (histórico de 2019 hasta la actualidad).
Tras barajar varias alternativas, hemos optado por utilizar los registros históricos meteorológicos que ofrece el portal de datos abiertos del Ayuntamiento de Madrid. En este caso se ha utilizado la técnica de *web-scraping* para descargar simultáneamente todos los archivos, ya que los ficheros se encontraban por meses. Para aplicar este método debemos seguir una serie de pasos:

- Identificar la URL donde se encuentra la información.
- Concretar la información que necesitamos (archivos CSV en este caso).
- Lanzar el programa para que localice y descargue los archivos.

Esta técnica ha sido utilizada únicamente con fines de eficiencia, gracias a este método se ha conseguido descargar 72 ficheros de forma instantánea. Se han unificado antes de ser almacenados, para tener la información centralizada en un solo archivo y facilitar la posterior lectura para su transformación. La librería encargada de realizar esta técnica es Beautiful Soup, podemos encontrar su documentación en [10].

Para poder calcular el momento del día en el que ha ocurrido el accidente hemos decidido obtener las horas de salida y puesta de sol y en función de esto determinar si el accidente ocurrió de día, de noche, amaneciendo o anocheciendo. Tras realizar una investigación decidimos

utilizar una API que ofrece esta información. Se le pasa por parámetros la ubicación del lugar donde se desea consultar las horas de salida y puesta de sol y la fecha. Por algún motivo que desconocemos, la API dejaba de devolver las solicitudes y no podíamos obtener toda la información. Se ha vuelto a investigar y hemos encontrado una librería en Python llamada Astral cuya documentación podemos encontrar en [2]. Gracias a esta librería obtenemos las horas de salida y puesta de sol para cada día desde el 01-01-2019 (fecha del primer registro del conjunto de datos) hasta la actualidad. Esta información se almacena en un CSV que será utilizado para determinar el momento del día de cada accidente.

5.1.2. Transformación y limpieza

Este paso es importante para dar el formato correspondiente a los datos y tratar todos sus errores e inconsistencias. Cada conjunto de datos seguía una estructura distinta, es por ello que cada script realiza operaciones diversas, pero a nivel general el procedimiento es:

- Leer del fichero CSV
- Convertir los datos leidos en un Dataframe de Pandas
- Realizar las transformaciones necesarios
- Cargar los datos a la base de datos

Para la lectura de los ficheros CSV se ha utilizado una función que detecta la codificación del archivo y procede a su lectura con la función *read_csv* de pandas. Veáse el fragmento de código 1

```
def cargar_csv_a_df():
    try:
        with open(ARCHIVO_ENTRADA, 'rb') as file:
            result = chardet.detect(file.read(10000))
            encoding = result['encoding']
            print(f"Codificación detectada: {encoding}")
            datos = pd.read_csv(ARCHIVO_ENTRADA, encoding=encoding,
   delimiter=";", on_bad_lines="skip")
            return datos
```

```
    except Exception as e:
        print(f"Error al detectar la codificación: {e}")
        return None
```

Listing 1: Función de lectura de archivo CSV

Debemos seguir los principios de normalización ya definidos: primero debemos tratar e introducir las tablas dimensionales, de forma que a la hora de añadir la tabla principal de accidentes se puedan indexar las columnas de manera correcta. Para ello, se lee el dataset principal de accidentes y se extraen los valores únicos de cada columna.

```
        datos = datos[columnas_validas].rename(columns=
    column_map_rename)
        datos_unicos = datos["descripcion"].dropna().unique()
```

Listing 2: Función de extracción de valores unicos

Esta función se debe ejecutar con cada columna de la que se necesite extraer datos. El atributo `column_map_rename` es un diccionario donde la clave corresponde a la columna del CSV y el valor es el nuevo nombre de dicha columna. Primero se filtra por las claves, para dejar en el dataframe solo las columnas que vamos a utilizar y después se renombran. Con el método `dropna().unique()` estamos eliminando las filas con valores nulos, y después dejando solo los valores únicos. Por último se inserta en la base de datos el dataframe resultante.

El siguiente paso es crear los perfiles de accidentados, es decir, todas las combinaciones de genero, rango de edad y tipo de persona (conductor, peatón o pasajero). Generamos todas las combinaciones posibles y las guardamos en el dataframe para posteriormente subirlas a la base de datos.

Una vez normalizada la información que incluye el propio archivo de accidentes, procedemos a transformar la información sobre la meteorología. Este ha sido un proceso más complejo de lo que parecía.
Una vez tenemos el archivo descargado con todos los históricos, la información se presenta con una estructura peculiar. Por cada fila se muestra la fecha, estación y magnitud que se está midiendo, y se muestra el valor que tomó por cada hora dicha magnitud, en dicha fecha y ob-

tenido desde esa estación. Para poder interpretar correctamente la información se necesitaba obtener las estaciones meteorológicas. Nuevamente, se accedió al portal de datos abiertos del Ayuntamiento de Madrid y se obtuvo las localizaciones de las estaciones de meteorología de la provincia de Madrid. Esta información se cargó en la base de datos para poder ser normalizada, posteriormente se relacionará cada accidente con una de estas estaciones.
Una vez obtenidas las estaciones, necesitamos saber a qué magnitud corresponde cada código. Para ello accedimos al documento que encontramos en [13] y encontramos la tabla presente en la Figura 13:

ANEXO II.PARÁMETROS, UNIDADES Y TÉCNICAS DE MEDIDA

CÓDIGO	PARÁMETRO	UNIDAD DE MEDIDA	TÉCNICA DE MEDIDA
80	RADIACIÓN ULTRAVIOLETA	Mw/m2	98
81	VELOCIDAD VIENTO	m/s	98
82	DIR. DE VIENTO	-	98
83	TEMPERATURA	ºC	98
86	HUMEDAD RELATIVA	%	98
87	PRESION BARIOMETRICA	mb	98
88	RADIACION SOLAR	W/m2	98
89	PRECIPITACIÓN	l/m2	98

Figura 13: Asociación entre los códigos y la descripción de las magnitudes meteorológicas.

Tras analizar todo esto, ya podemos proceder a transformar y limpiar los datos meteorológicos, dejando un registro por cada día y hora de cada estación y los valores de las distintas magnitudes. Como cada registro presenta la fecha, hora y estación, entre otros, podemos relacionarlo fácilmente con cada accidente.

La información de los distritos también fue normalizada, en la tabla `distrito` se almacena el código del distrito y su nombre.

Una vez que tenemos toda la información normalizada y almacenada en la base de datos, procedemos a leer los accidentes. Debemos recordar, que los accidentes se encuentran duplicados tantas veces como implicados hay. Ya que cada registro corresponde con un implicado. Para ello debemos extraer los valores únicos y quedarnos solo con aquellas columnas propias del accidente: fecha, hora, número de expediente, coordenadas, vía, distrito y tipo de accidente.

Las coordenadas se muestran en el formato UTM, para garantizar la consistencia con el resto de datos debemos convertirlas a longitud y latitud. Para ello utilizamos la librería `Pyproj`, que proporciona herramientas para trabajar con sistemas de referencia espacial. Concretamente se ha utilizado la clase `Transformer` cuya documentación encontramos en [20].

Para terminar el proceso de normalización debemos obtener de la base de datos las distintas tablas dimensionales para proceder a sustituir los valores por sus índices correspondientes, de manera que no generen conflicto con las claves foráneas. Utilizamos la función de pandas para leer de la base de datos y cuando tenemos las tablas en un dataframe procedemos a pasarlo a un diccionario para poder realizar fácilmente la traducción. Para convertir cada dataframe en un diccionario utilizamos la función del fragmento de código 3.

```
map_accidente = dict(zip(df_tipo_accidente["descripcion"],
df_tipo_accidente["id_tipo_accidente"]))
```

Listing 3: Función para convertir un DF a diccionario

Después sustituimos, como podemos ver en el fragmento de código 4.

```
1 df_accidentes["tipo_accidente"] = df_accidentes["tipo_accidente"
].map(map_accidente)
```

Listing 4: Función para sustituir valores por clave desde un diccionario

Repetimos este proceso con todas las tablas dimensionales correspondientes.

Para establecer correctamente la relación entre el accidente y la información meteorológica necesitamos calcular cuál es la estación meteorológica más cercana a cada accidente.
Leemos de la base de datos todas las estaciones que tenemos almacenadas, y para cada uno de los accidentes iteramos buscando la más cercana. Para ello utilizamos la función *Distance* de la librería GeoPy [11], mediante esta librería podemos calcular la distancia entre dos puntos. Una vez más, estos datos también se añaden de forma normalizada. En la columna `estacion_meteo` encontramos el índice de la estación, relacionada mediante una clave foránea a la tabla `estacion_meteo`.

Debemos calcular y añadir el momento del día en el que el accidente ha ocurrido. Para ello leemos el CSV que hemos generado con las horas de salida y puesta del sol en la zona de Madrid.
El criterio es simple: para facilitar su comprensión vamos a recrear el algoritmo en un diagrama de flujo, como se puede ver en la Figura 14.

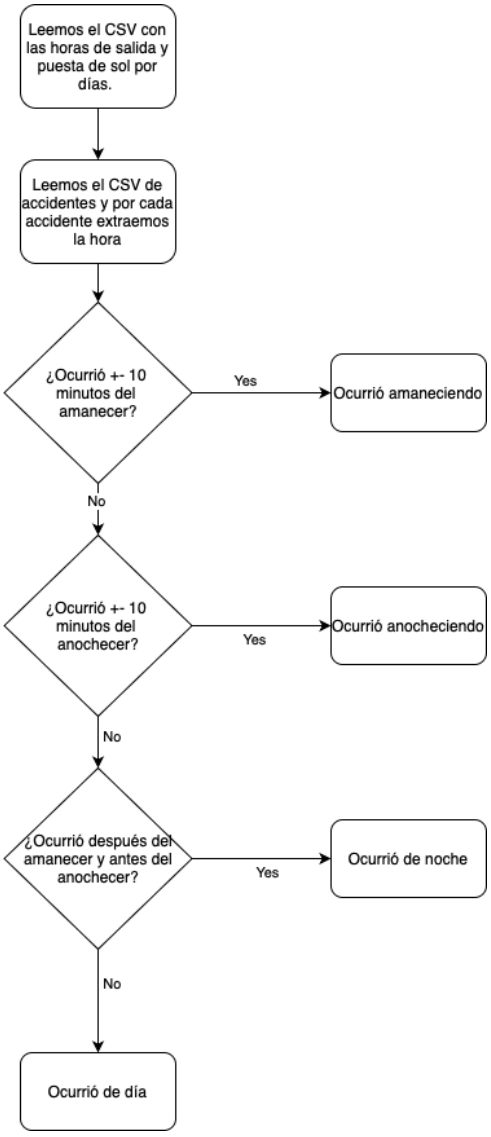

Figura 14: Diagrama de flujo: Cálculo del momento del día

Para poder añadir los datos de los heridos es crucial haber cargado previamente los accidentes y la información sobre los distintos perfiles de personas, ya que al tratarse de una tabla intermedia, si estos no se han cargado se producirán conflictos con las claves foráneas.
Debemos normalizar las columnas relativas al perfil del herido, su lesividad y el tipo de vehícu-

lo. Para ello, al igual que hemos mencionado en los accidentes, debemos consultar las tablas correspondientes a la base de datos y convertir en un diccionario los valores para que puedan ser sustituidos. Podemos eliminar también todas las columnas que no nos interesan en esta tabla (fecha, hora, nombre de la vía, coordenadas ..) información que ya se encuentra almacenada en los accidentes. Tras realizar estas acciones tendremos un registro por cada herido, asignado a un accidente y con la información del herido regida por índices. Una vez realizado esto, ya podríamos insertar los heridos en la base de datos.

En último lugar, procedemos a volcar a la base de datos la información que tenemos almacenada en los ficheros CSV acerca de centros educativos, hospitales, radares y paneles de información variable. En el caso de los centros educativos, el tipo de centro educativo se encuentra normalizado, para ello procedemos a extraer los valores únicos y almacenarlos en la tabla `tipo_centro`. Posteriormente se leen los valores, se pasan a un diccionario y se sustituyen por los índices correspondientes. Es irrelevante en que orden añadamos el resto de tablas ya que no dependen de otras. Se realizan algunas transformaciones como eliminar columnas irrelevantes, centralizar el tipo de vía, nombre y número en un solo atributo. Nos aseguramos que las coordenadas se encuentran en el formato correcto y podemos proceder a la inserción en la base de datos.

Para situar los accidentes ocurridos en cruces, se ha procedido a redondear los decimales de las coordenadas a 5 dígitos. De esta forma mantenemos la precisión y podemos filtrar por aquellos donde las coordenadas coincidan. Aunque este método puede arrojar falsos negativos, ya que puede que el agente que registrase el accidente no introdujese las coordenadas exactas o el sistema encargado de determinar las coordenadas no sea preciso al 100 %.

5.1.3. Ingesta de datos

Gracias a la función 5 de la librería SQLAlchemy podemos crear el motor de conexión a la base de datos, introduciendo por parámetros la URI de la misma.

```
1    engine = create_engine(DATABASE)
```

Listing 5: Motor de conexión a BD

Con este motor podemos hacer las consultas directamente desde Pandas utilizando la función que se muestra en el fragmento de código 6.

```
pd.read_sql("SELECT * FROM tipo_vehiculo", engine)
```

Listing 6: Lectura de la tabla tipo_vehiculo

Además, podemos introducir los datos directamente desde Pandas también como podemos ver en la función que se muestra en el fragmento de código 7.

```
df_merged.to_sql("accidente", engine, if_exists="append", index=
False)
```

Listing 7: Función para insertar un dataframe directamente a la BD

Indicamos en primer lugar en qué tabla queremos que añada los datos, después indicamos el motor de conexión con la base de datos, le decimos que si ya hay registros en la tabla que los añada y que no añada una columna adicional con los índices.

5.2. Implementación del backend

En el capítulo anterior mencionamos la estructura del backend, en este vamos a detallar cómo se ha implementado.

5.2.1. Módulo `models/`

Comenzaremos por el módulo `models/` ya que es el que contiene la definición de las entidades del sistema, tanto para su representación en la base de datos como para su uso en los endpoints de la API. Estos modelos permiten estructurar los datos que se reciben o envían en formato JSON, así como establecer validaciones, relaciones entre tablas y conversiones automáticas. El concepto de *Object-Relational Mapping* (ORM) hace referencia a una técnica de programación que permite interactuar con bases de datos relacionales a través de objetos y clases en un lenguaje de programación orientado a objetos. En lugar de utilizar sentencias SQL directamente para consultar o modificar los datos, un ORM permite acceder a las tablas de la base de datos como si fueran estructuras nativas del lenguaje (por ejemplo, clases en Python). Esto nos permitirá en el resto de módulos trabajar con las tablas SQL como si fuesen objetos, por lo que minimizamos cualquier error posible a la hora de realizar las consultas y

obtener los datos.

Creamos una clase por cada tabla de la base de datos y para cada una de las clases debemos definir el nombre de la tabla y sus columnas. Debemos añadir las claves primarias y foráneas, para que cada vez que se trate un objeto se compruebe, antes de ser insertado, que cumple con las restricciones necesarias. De esta manera definimos los objetos con las mismas propiedades de las tablas en la base de datos.

5.2.2. Módulo `services/`

El módulo services es el encargado de realizar las peticiones a la base de datos. Utiliza los modelos anteriormente mencionados para validar la estructura de los datos que recibe y poder devolverlos en formato JSON para que la API los publique en el formato adecuado.

Gracias a tener bien definidos los modelos y utilizar un ORM como SQLAlchemy, podemos realizar consultas bastante sencillas utilizando objetos y atributos. Como vemos en el fragmento de código 8, esta sería la estructura de la consulta. Gracias al uso del ORM indicamos por parámetro el objeto, que está referenciando una tabla, al que queremos hacerle la consulta. Al no indicar ninguno de sus atributos, en `respuesta` tendremos tantas tuplas como registros haya en la base de datos y cada elemento de la tupla será un atributo de la tabla `accidente`. Si queremos obtener más tablas en la misma consulta solo debemos concatenar los objetos dentro de `query()` que referencien a cada una de las tablas que queremos obtener. Si de alguna de las tablas queremos obtener algún atributo concreto simplemente, podemos indicarlo, como vemos en el fragmento de código 9. En este caso nos devolverá una tupla compuesta por otra tupla y un atributo.

El método `filter()` filtra los resultados según el criterio que le indiquemos. Con el método `join()` indicamos a que tablas queremos hacer join, y mediante que atributo realizarlo, en caso de necesitarlo. Por último con `all()` le decimos que queremos todos los registros encontrados.

```
1    respuesta = db.query(Accidente).filter().join().all()
```

Listing 8: Ejemplo 1: Función para realizar la solicitud a la BD

```
1    respuesta = db.query(Accidente, Heridoaccidente.lesividad).filter
   ().join().all()
```

Listing 9: Ejemplo 2: Función para realizar la solicitud a la BD

Gracias al método `limit(100)` se limita a 100 el número de registros devueltos. Esto ha sido muy útil para realizar las pruebas ya que reducíamos la carga de la API y del frontend haciendo la aplicación mucho más ligera y rápida.

Al combinar entidades con atributos en el `SELECT`, debemos ser cuidadosos a la hora de devolver los resultados. La salida del método `db.query()` devuelve objetos ORM como instancias de las clases del modelo, para una correcta serialización debemos formatear la salida para que se adapte al texto plano que devolverá la API.

Para poder habilitar uno de los filtros, en el que se filtran los accidentes por números de heridos hemos realizado una consulta apoyándonos en el método `func` de SQLAlchemy. Como vemos en el fragmento de código 10, podemos agrupar por accidente y contar cuántos heridos hay en cada grupo, lo que nos dará el número de heridos por accidente.

```
1    num_heridos = (
         db.query(
3            Heridoaccidente.accidente,
             func.count(Heridoaccidente.id_herido_accidente).label("
   num_heridos")
5        )
         .group_by(Heridoaccidente.accidente)
```

Listing 10: Heridos por accidente

Dentro del módulo hemos diferenciado 3 clases: una para leer los accidentes de la base de datos, otra para leer las tablas dimensionales y una tercera para leer la tabla intermedia `herido_accidente`. De esta manera tenemos los métodos bien organizados y facilitará su reutilización e implementación.

5.2.3. Módulo `routes/`

El módulo routes es la capa más cercana al frontend, dentro del backend. Es la encargada de definir las rutas que consumirá el dashboard para obtener los datos.

En este caso, hemos creado 3 rutas principales, una para cada clase existente en el módulo `services/`. Es decir:

- `accidentes/`
- `heridos/`
- `complementarios/`

Dentro de la primera ruta, la que devuelve los accidentes, hemos creado subrutas específicas:

- **Accidentes completos.** Se devuelve la información de los accidentes procesados, con los respectivos `join` realizados. Podríamos clasificarlo como información legible por cualquier usuario.
- **Accidentes sin tratar.** En esta otra ruta se devuelven los accidentes donde la mayoría de sus columnas contienen índices, que referencian a los valores almacenados en las tablas dimensionales. Con estar ruta podemos obtener la información básica de los accidentes de manera más eficiente que procesando todos los campos mediante los `join`. La información acerca del día festivo y de la meteorología ha sido de obligado requisito devolverla ya procesada realizando el `join`, ya que realizar esta operación en la capa frontend requería demasiado coste computacional para la interfaz lo que hacía que los gráficos tardasen demasiado en ser renderizados.
- **Información completa de un solo accidente.** Esta ruta es útil para cuando se hace clic en un accidente, podemos cargar toda la información ya procesada de dicho accidente. De esta manera nos ahorramos tener que cargar toda la información de todos los accidentes desde el principio.
- **Actualización de datos.** Se ha decidido añadir esta ruta que facilita la ejecución del archivo encargado de actualizar automáticamente los datos. Además, también actualiza el conjunto de datos meteorológicos, de forma que los datos quedarían completamente integrados y listos para analizar.

Los pequeños detalles son realmente importantes ya que trabajamos con un volumen de datos bastante elevado. Por ello debemos tratar de reducir al mínimo la carga del dashboard para que funcione adecuadamente y sin demoras demasiado altas. Para ello se utilizan los distintos endpoints en función de los requerimientos de la aplicación, si necesitamos una simple representación en el mapa basta con obtener los datos indexados. En cambio, si necesitamos un alto nivel de detalle podemos obtener los datos con sus descripciones para comprenderlos correctamente.

En el fragmento de código11 podemos ver cómo definimos la ruta en la que vamos a devolver la información.

```
    @router.get("/{accidente_id}/completo")
```

Listing 11: Definición de la ruta

La segunda de las rutas principales la tenemos disponible para obtener información acerca de los heridos. La ruta raíz sería `/heridos` y a partir de ahí encontramos:

- `/` Se devuelve la información de `herido_accidente`, tal y como se encuentra en la base de datos.
- `/completos` Devolvemos todos los registros de la tabla `herido_accidente` con la información legible, es decir, con la información cruzada.
- `/{accidente_id}` En esta ruta devolvemos toda la información, legible, acerca de los heridos del accidente pasado por parámetros.
- `/num_heridos/all` Devolvemos una tupla con el identificador del accidente y el número de heridos.

Por último, para la ruta de las entidades dimensionales, hemos creado una subruta por cada entidad. De esta manera tenemos fácil acceso a todos los datos dimensionales para poder crear los filtros y que el usuario pueda filtrar viendo una cadena de texto en lugar de un número. Aunque a nivel interno el filtro utiliza el índice en lugar del valor para filtrar. Ya que, como hemos mencionado antes, el conjunto de datos que se carga desde el principio no contiene los

valores de las tablas dimensionales, sino sus índices.

Todas las funciones del módulo routes siguen el mismo procedimiento, definen la ruta de la API con el `@router.get` y después hacen una llamada a la función correspondiente del módulo `services/` para obtener el JSON con los datos que debe devolver para ser consumidos por el frontend.

5.3. Implementación del frontend

En el desarrollo del dashboard podríamos establecer una analogía con el pico de un iceberg. Realmente es la única parte que el usuario va a percibir, pero para su correcto funcionamiento se requiere un gran trabajo de ingeniería de datos y un servidor bien estructurado que devuelva los datos de manera rápida y eficiente.

Como ya hemos mencionado, el panel interactivo ha sido desarrollado utilizando el framework Dash. Esto nos ha permitido implementar la interfaz de manera eficaz y aprovechar la integración con Plotly y Plotly Express. El objetivo principal a la hora del diseño era poder crear una herramienta visual, atractiva y sencilla de utilizar. Todo el código ha sido desarrollado en un archivo Python divido en distintas funciones.

5.3.1. Visualizador de mapas y estadísticas

Nada más cargar la página, el dashboard solicita al backend los datos necesarios para cargar los mapas, estadísticas y filtros. Se ha tratado en todo momento reducir al mínimo la cantidad de operaciones lógicas que debe gestionar el frontend. En la siguiente función 12 vemos como se realiza con requests las solicitudes a la API. Se realiza una llamada por cada conjunto de datos necesitado para los filtros. Es decir, se solicitan todos los tipos de vehículos, todos los días de la semana, todos los momentos del día, todos los géneros, etc. De esta manera podemos crear los desplegables de los filtros de manera sencilla.

```
1 def request(url):
      r = requests.get(LOCALHOST+url)
3     if r.status_code == 200:
          data = r.json()
```

```
        df = pd.DataFrame(data)
        return df
    else:
        print(f"Error {r.status_code} al acceder a {url}")
        return pd.DataFrame()
```

Listing 12: Función de consulta al backend REST

Por otro lado, también se solicitan a la API todos los accidentes tal y como se encuentran en la base de datos, con los atributos en bruto, definidos por su índice. De esta manera reducimos la carga en memoria de la interfaz.

La primera vista al abrir el panel sería la mostrada en la Figura 15, podemos ver como se muestra el mapa con distintos marcadores. Por un lado, los puntos rojos corresponden a los accidentes. Los azules corresponden con centros educativos, los verdes son los centros de asistencia médica y los naranjas serían los radares. No se han incluido el resto de elementos de seguridad vial (paneles de información variable o cruces semaforizados) ya que sobrecargarían demasiado el mapa.
Gracias a esto, podemos ver las posibles relaciones entre los accidentes y la ubicación de los colegios, por ejemplo.

También podemos utilizarlo para tratar de explicar por qué algunos accidentes presentan una mayor tasa de heridos graves. Quizás si se encuentran muy alejados de un centro médico, la asistencia se demora y por tanto las consecuencias son mayores.

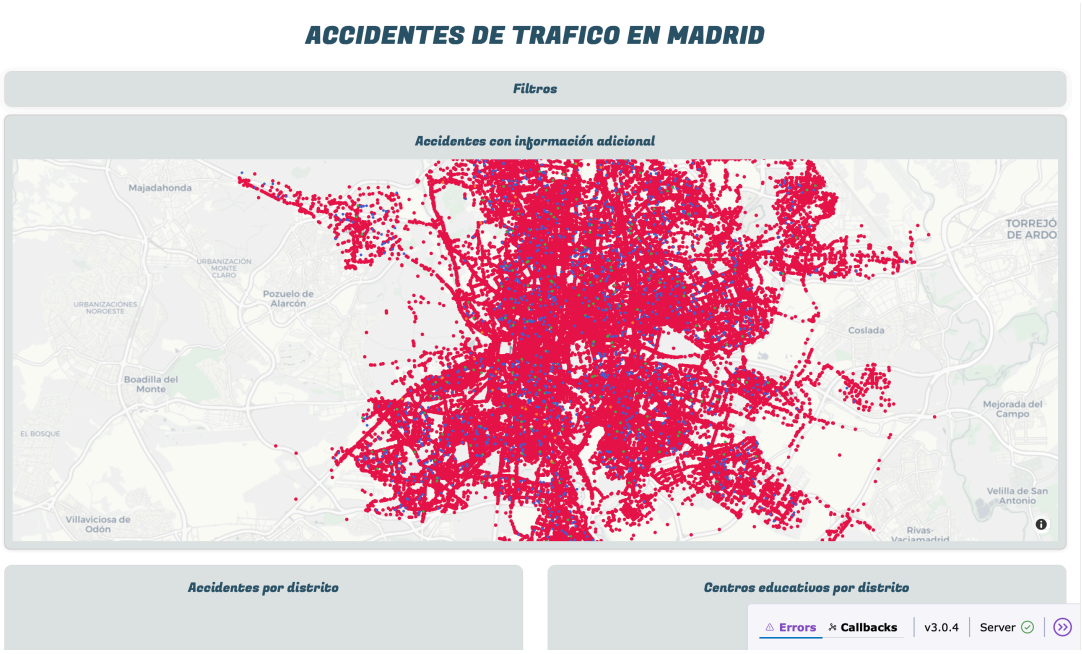

Figura 15: Vista al abrir la herramienta

Un pequeño desafío enfrentado fue crear el mapa coroplético, ya que para ser creado necesitamos un archivo GEOJSON donde se delimita el área de cada distrito. En el portal de datos abiertos del Ayuntamiento de Madrid se ofrece el archivo en formato SHP. Para poder ser utilizado por Plotly Express necesitamos convertirlo a GEOJSON. Tras un breve análisis encontramos la herramienta MyGeodata [15] gracias a la cual pudimos convertir el fichero al formato soportado por Plotly.

Este mapa nos ofrece una visión cromática de los distritos de mayor incidencia. Hemos creado dos mapas de este tipo, uno para mostrar la densidad de accidentes por distrito y otro para mostrar la densidad de centros educativos por distrito. Podemos verlos en la Figura 16. En dicha figura podemos observar también información estadística sobre el número de accidentes en el que se ha visto implicado determinado género o cada tipo de persona. Evidentemente sin al menos un conductor no se puede dar un accidente, pero es útil para ver la proporción de accidentes ocurridos por personas que viajan solas o acompañadas o si suceden muchos atropellos.

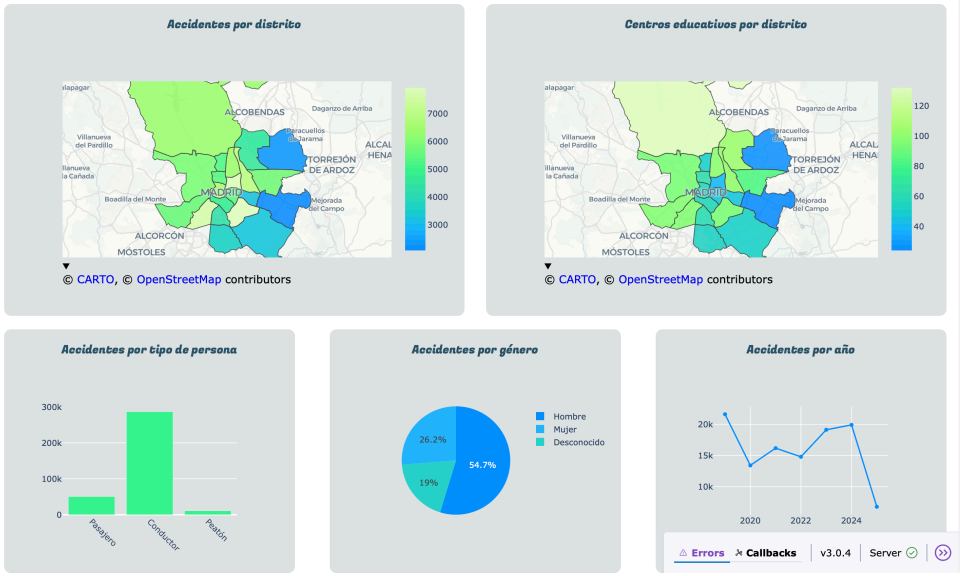

Figura 16: Vista I de estadísticos

En la Figura 17 podemos ver otros estadísticos variados que muestran desde información relacionada con el día y hora del accidente hasta información asociada a la presencia de lluvia o alcohol.

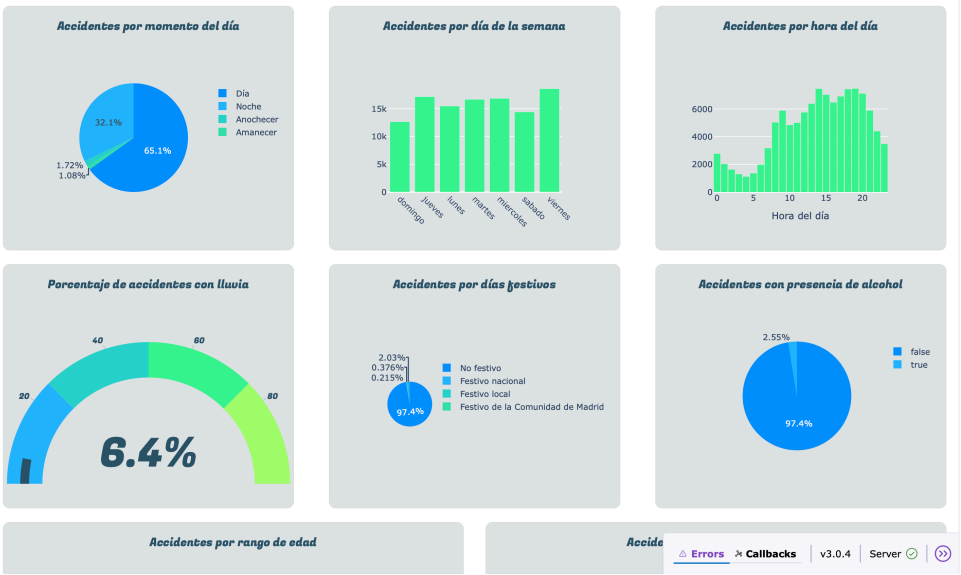

Figura 17: Vista II de estadísticos

Entre los últimos estadísticos encontramos una serie de diagramas de barras que ofrecen de un vistazo información acerca del número de accidentes en los que se ven implicados los diversos rangos de edad o cada tipo de vehículo. También podemos ver cuántos accidentes

ocurren con los distintos tipos de gravedad de heridos o los distintos tipos de accidentes.

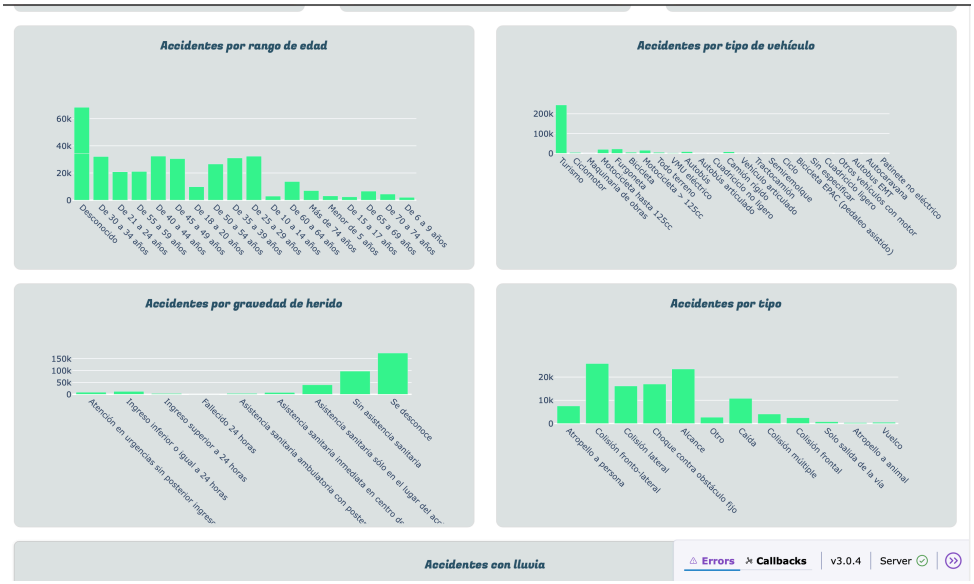

Figura 18: Vista III de estadísticos

Las figuras estadísticas han sido creadas con `plotly.express`, una API de alto nivel de Plotly que permite crear gráficos comunes con pocas líneas de código. Su integración con Dash la convierte en una herramienta ideal para construir dashboards dinámicos y profesionales. [16] Las figuras las podemos crear como vemos en el fragmento de código 13.

```
    fig = px.pie(
        df,
        names=names,
        values=values,
        labels=labels
    )
```

Listing 13: Creación de la figura

y de esta manera (fragmento de código 14), le podemos eliminar el fondo:

```
    fig.update_layout(
        paper_bgcolor='rgba(0,0,0,0)',
        plot_bgcolor='rgba(0,0,0,0)'
    )
```

Listing 14: Eliminado de fondo de la figura

El proceso es similar para los diagramas de barras, histogramas y diagramas de líneas. Los mapas coropléticos aunque son algo más complejos y requieren de otros datos, como la delimitación de los distritos que ya hemos comentado, también han sido creados con Plotly Express.

Los mapas normales, en cambio, han sido creados de una forma distinta. Se ha utilizado un diccionario de Plotly en el que se han definido las capas. Como podemos ver en 15, cada capa indica el tipo de mapa, en este caso scattermapbox, que permite representar marcadores georeferenciados. Indicamos dónde encontrar la información de la posición y el dato que queremos mostrar al pasar el ratón por encima. El campo `cusstomdata` nos permite poder referenciar en qué tipo de dato se ha hecho clic para luego recuperar su información, importante si queremos diferenciar si se ha seleccionado un accidente o un centro educativo.

```
{   "type": "scattermapbox",
    "lat": df_filtrado["latitud"],
    "lon": df_filtrado["longitud"],
    "mode": "markers",
    "marker": {"size": 5, "color": COLORES_LLAMATIVOS[0]},
    "text": df_filtrado["id_accidente"],
    "hoverinfo": "text",
    "customdata": ["accidente"] * len(df_filtrado)}
```

Listing 15: Configuración del mapa

También podemos indicar el tipo de mapa que queremos que se utilice así como la configuración inicial del mapa 16. Mapbox es una plataforma de mapeo y geolocalización que proporciona mapas personalizados e interactivos [14]. Mapbox incluye distintos tipos de mapas, algunos gratuitos, otros de pago y otros gratuitos pero que requieren un registro previo. Por simplicidad hemos escogido el mapa gratuito *carto-positron* ya que es suficiente para nuestra herramienta.

```
"mapbox": {
    "style": "carto-positron",
    "center": {"lat": 40.4168, "lon": -3.7038},
    "zoom": 11
}
```

Listing 16: Configuración del mapa

Una vez tenemos todas las figuras y mapas creadas, las devolvemos mediante `@app.callback()`. Esta función es un decorador que permite definir funciones dinámicas que se ejecutan automáticamente cuando el usuario interactúa con la interfaz. Por ejemplo, cada vez que el usuario filtra, todas las demás estadísticas se actualizan de forma dinámica y automática, además del propio mapa.

Todos estos objetos son renderizados para que el usuario final pueda visualizarlos. Para ello utilizamos la función que se muestra en el fragmento de código17, a la cual indicamos el id de la figura que deseamos renderizar. Esta función se incluye en el `app.layout` que es el elemento principal de una aplicación Dash. En él se definen la estructura y contenido de la interfaz.

```
1     dcc.Graph(id="mapa-accidentes")
```

Listing 17: Función de renderizado

En la Figura 19 podemos ver los accidentes en los que ha habido lluvia. Estos accidentes varían en función de los filtros aplicados, es decir este mapa, a igual que los anteriores es dinámico y su información se actualiza cada vez que un filtro es aplicado.

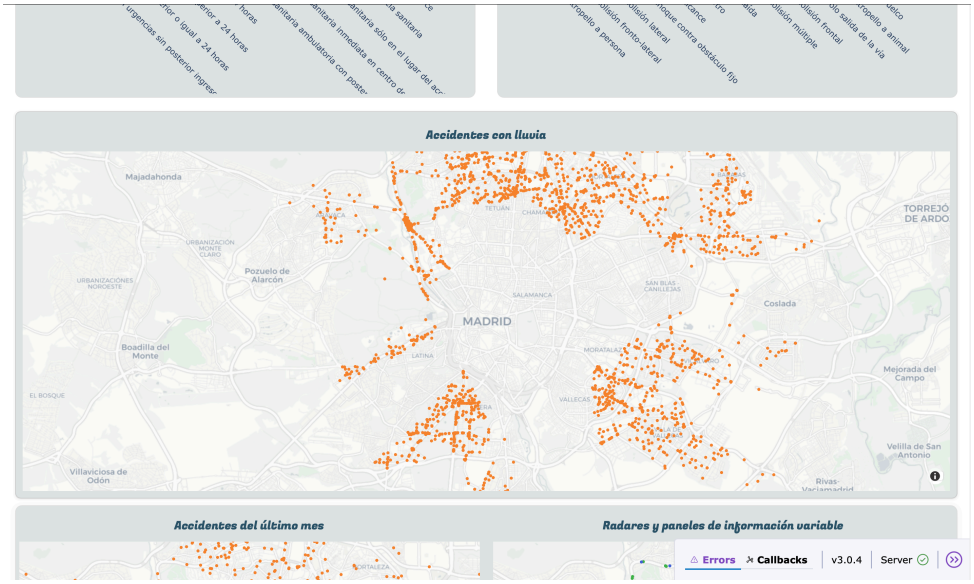

Figura 19: Vista de accidentes con lluvia

Los dos últimos mapas que podemos ver en la Figura 20 corresponden a mapas estáticos, cuya información es la misma sea cual sea el filtro aplicado. En el primer mapa podemos visualizar los accidentes ocurridos en el último mes, muy útil para poder ver nuevas tendencias o patrones en el corto plazo. El mapa de la derecha hemos incluido información variada sobre elementos de seguridad vial como radares y paneles de información variable.

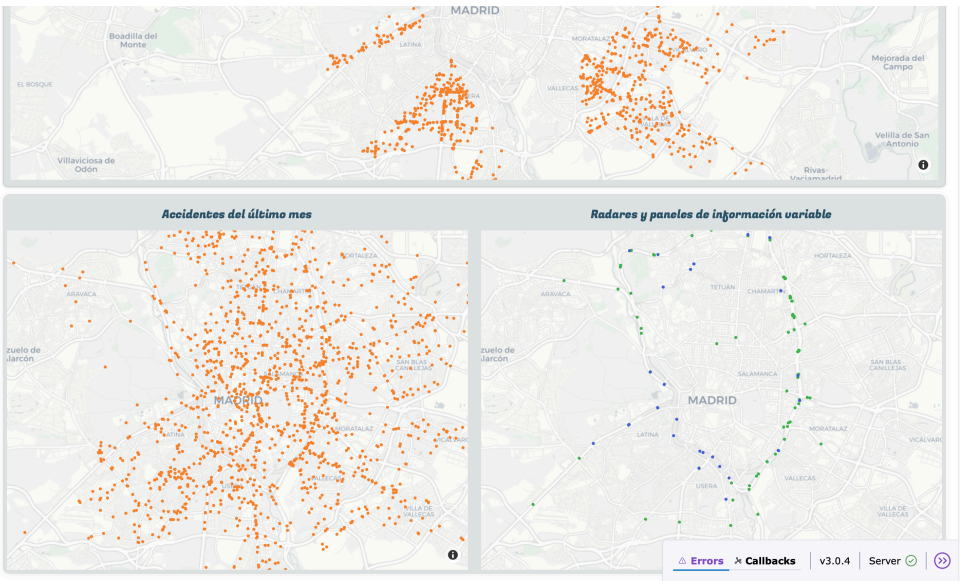

Figura 20: Vista de mapas estáticos

La finalidad de estos marcadores es mejorar la accesibilidad y eficiencia en la interpretación de los datos, facilitando una lectura sencilla sin necesidad de analizar gráficos complejos o explorar en profundidad. De esta forma, el usuario puede detectar patrones o anomalías desde el primer vistazo.
Además, su implementación resulta algorítmicamente rápida y altamente eficiente, ya que se trata de agregaciones básicas calculadas dinámicamente sobre los datos filtrados. Esto contribuye a enriquecer la experiencia de usuario sin afectar al rendimiento de la aplicación.

5.3.2. Detalles del accidente

Si hacemos clic en un accidente, podemos ver cómo se abre un panel en el que se muestra información relativa al accidente y de cada uno de los heridos. Podemos verlo en la Figura21.

Figura 21: Panel detalles de accidente

Gracias al elemento que hemos mencionado en la sección anterior, podemos identificar que tipo de dato se ha seleccionado. Si es un accidente debemos recuperar qué accidente y obtener toda su información completa mediante una llamada a la API. La llamada a la API consiste en una solicitud GET a la siguiente dirección `{URL_BASE}/heridos/{id_accidente}` pasamos por parámetros el identificador del accidente para que el backend pueda solicitar toda la información asociada a dicho accidente. Esta información es mostrada tal y como vemos en la Figura 21.

Una vez hayamos analizado la información mostrada podemos cerrar el panel haciendo clic en el botón de la parte superior derecha del panel. Esta acción es recuperada por el decorador que maneja el clic y devuelve un panel vacío, entonces el cuadro que había mostrado la información desaparece.

5.3.3. Aplicar filtros

El espacio de la pantalla es limitado, y en un dashboard lo primordial son los elementos informativos. Es por ello que el panel de filtros aparece plegado cuando se accede a la interfaz. Debemos pulsar en el botón `Filtros`, lo que desplegará un panel con todos los filtros disponibles, tal y como se observa en la Figura 22.

Los filtros son independientes, es decir podemos aplicar todos los que queramos de manera combinada.
Si se está buscando encontrar un patrón de siniestralidad concreto, esta acción facilita la labor. Además, los filtros se actualizan en tiempo real nada más se seleccionan.

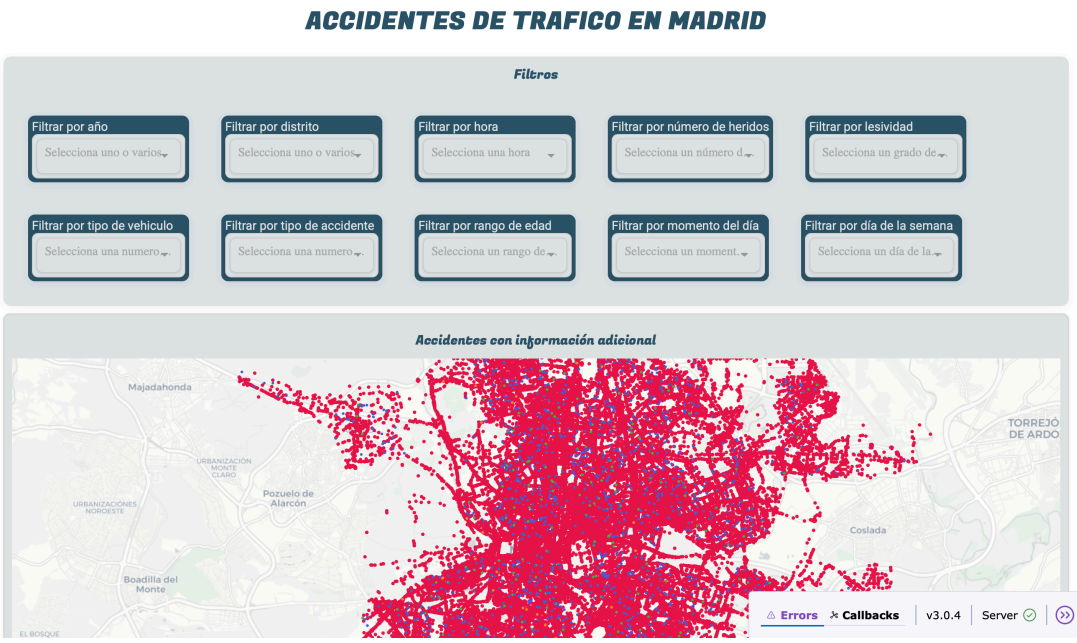

Figura 22: Panel de filtros

También tenemos la posibilidad de buscar escribiendo en lugar de recorrer todo el desplegable. Gracias a obtener los valores desde la API los filtros siempre se encuentran actualizados y controlamos de manera sólida los errores, si añadimos un aparece un nuevo tipo de vehículo, bastará con actualizar la tabla normalizada y ya los filtros contemplan este nuevo tipo de vehículo, sin necesidad de actualizar la interfaz. Esto es realmente importante para mantener la escalabilidad y la eficiencia en nuestra herramienta. Además, como los desplegables muestran las opciones que existen no hay lugar a errores por escribir mal un tipo de vehículo o un rango de edad, ni se contempla la opción de elegir un criterio inexistente.

En la figura 23 podemos ver como se pueden seleccionar varios filtros simultáneamente y además como el mapa ya se encuentra actualizado.

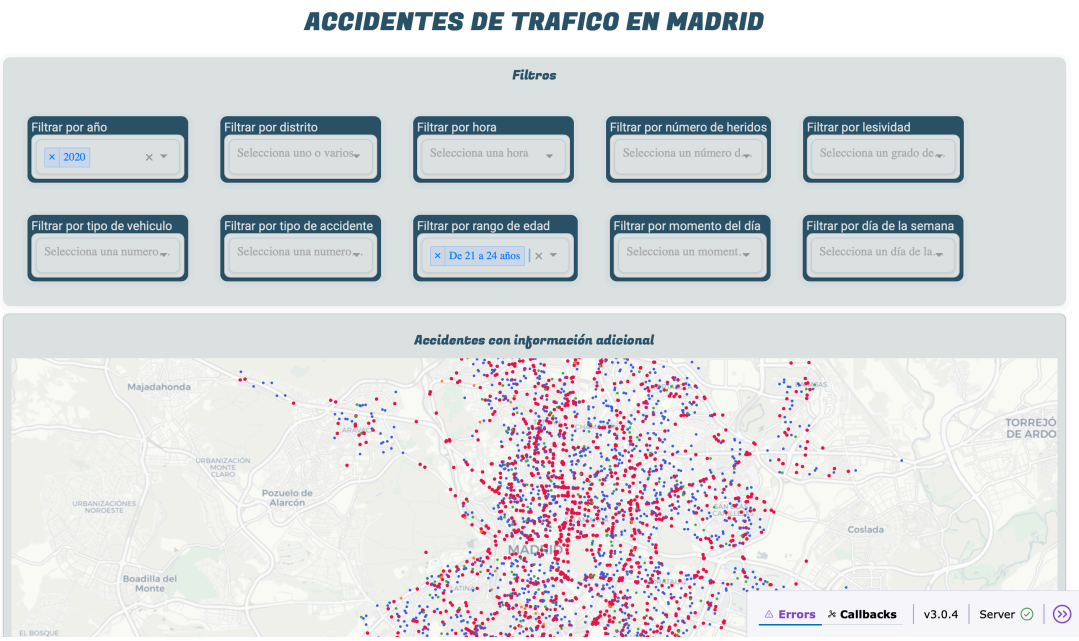

Figura 23: Selección de filtros

Debemos puntualizar que, los filtros relacionados con los heridos como `Filtro por rango de edad` filtrará los accidentes en los que haya al menos un accidentado que cumpla dicha condición. Si filtramos por rango de edad de 21 a 24 años, por ejemplo, y visualizamos el estadístico, el resto de rangos de edad no tienen por qué estar a 0. Ya que los estadísticos analizan el número de accidentes en el que cada rango de edad se ha visto implicado, entonces pueden existir accidentes con múltiples heridos de diversos rangos de edad. Si uno de los heridos está en el rango de edad del filtro, y además hay un herido del rango de edad de 30 a 34 años, el mismo accidente computará para el cálculo de ambos rangos de edad.

6

Despliegue y pruebas

Este capítulo describe los pasos seguidos para el despliegue, los problemas encontrados y la solución adoptada. El objetivo de desplegar la aplicación es facilitar la interoperabilidad y permitir que usuarios menos técnicos puedan utilizar la herramienta sin necesidad de ejecutar comandos. Además, gracias al despliegue se consigue automatizar aún más el proceso de actualización de datos.

6.1. Despliegue

El despliegue se ha intentado sin éxito debido a que las plataformas que ofrecen planes de prueba gratuitos limitan la cantidad de memoria del servidor y nuestra herramienta supera esos límites dada la gran cantidad de datos que maneja. Vamos a detallar los pasos y los servicios utilizados.

Tras realizar un estudio de servicios gratuitos que permitan desplegar una base de datos, backend y frontend hemos encontrado dos grandes candidatos tanto por sus buenas valoraciones como por sus planes de prueba gratuitos.

La primera herramienta es Railway [18], tras autenticarnos vemos que tenemos agotado el crédito gratuito por un proyecto anterior. Railway implementa una política muy estricta para evitar el abuso de su servicio gratuito y como la cuenta GitHub del anterior proyecto y del actual son la misma no podemos crear una cuenta nueva para aprovechar ese crédito gratuito. Aún así, nos permite crear una instancia de la base de datos para poder ser accedida desde cualquier lugar con acceso a internet, por lo que procedemos a crearla en Railway.

Comenzamos creando el fichero SQL que incluye la estructura y los datos. Este fichero

se puede crear fácilmente con `pg_dump`, una herramienta CLI incluida en la instalación de PostgreSQL. Mediante el comando `pg_dump -U USUARIO -d DATABASE -F p -f init.sql` indicamos el usuario y el nombre de la base de datos y nos genera en el fichero `init.sql` que contiene tanto la estructura de la base de datos como los datos.
Procedemos a subir el archivo a Railway y nos genera la base de datos, una vez generada podemos acceder a ella desde la URI que nos da Railway.

Una vez desplegada la base de datos, necesitamos desplegar el backend.
La segunda herramienta analizada es Render [19]. Esta herramienta posee características muy similares a Railway por lo que decidimos crearnos una cuenta, sincronizar con GitHub y subir el proyecto.
Tras subir el proyecto mediante GitHub debemos indicar el comando de construcción `pip install -r requirements.txt` y el comando para lanzar la aplicación `uvicorn main:app --host=0.0.0.0 --port=10000`
Tras configurar correctamente las variables, creamos la variable de entorno con la URI de la base de datos, obtenida desde los ajustes de Railway.
Railway procede a compilar y ejecutar la aplicación para desplegarla. Una vez desplegada, cogemos la URL que nos ha dado Railway y la modificamos por la que teníamos de localhost en el frontend.
Cuando tratamos de realizar peticiones a la API desde el frontend nos salta mensaje de error. Se muestra un *log* en el que indica que la cantidad de memoria de nuestra aplicación supera los límites de su prueba gratuita, por lo que debemos adquirir un plan de pago. Debido a la finalidad y el contexto de este proyecto esa opción no es viable.

6.2. Docker

Para solucionar los problemas del despliegue, se ha decidido contenerizar la aplicación mediante Docker [9]. Esto permitirá facilitar la instalación en prácticamente cualquier dispositivo, aunque no sea un despliegue accesible desde internet.
Docker ofrece numerosas ventajas:

- **Portabilidad entre entornos:** Docker empaqueta la aplicación con todas sus depen-

dencias, lo que garantiza que se ejecutará de la misma forma en cualquier entorno y dispositivo.

- **Despliegue rápido y automatizado:** Basta con ejecutar un comando para lanzar toda la infraestructura desde base de datos hasta la interfaz.
- **Entorno aislado:** Docker actúa como un entorno virtual ligero, lo que evita la necesidad de instalar manualmente Python, PostgreSQL o las distintas bibliotecas necesarias para el despliegue de la aplicación.

El primer paso es crear el archivo `docker-compose.yml`. Este archivo permite definir y gestionar múltiples servicios Docker (como la base de datos, el backend y el frontend) de forma conjunta y estructurada. A través de este archivo se especifican las imágenes, los puertos, las variables de entorno, los volúmenes y las dependencias entre contenedores. Al utilizar este archivo automatizamos y simplificamos el despliegue de entornos complejos mediante un solo comando `docker-compose up`, asegurando que todos los servicios se inicien correctamente y en el orden adecuado.
A continuación, debemos añadir y configurar los ficheros `dockerfile`. Estos ficheros se encargan de construir la imagen en la que se encapsula el sistema para poder ser montado mediante contenedores. En este fichero se define la tecnología necesaria y su versión, el comando para instalar las dependencias, así como el comando para lanzar el proceso. En este caso serían los mismos que hemos mencionado en la sección anterior.
Para la base de datos, utilizaremos el fichero `init.sql` que ya hemos generado para el despliegue. La base de datos se encapsulará en un volumen y solo será generada la primera vez, gracias a la persistencia que ofrece Docker. Siempre que ejecutemos el comando `docker-compose down`, los contenedores se eliminarán pero los datos de los volúmenes no se borraran. Si añadimos el *flag* `-v` indicamos que que también borre esos volúmenes. Si necesitamos reconstruir la base de datos este flag será útil.

Una vez todos estos ficheros han sido definidos podemos proceder a realizar la construcción de las imágenes y la orquestación. Realizamos `docker-compose up --build` y comienza la orquestación. Primero intenta buscar el volumen de datos; si no lo encuentra ejecuta el comando para crear la instancia desde el archivo SQL. Después lanza el backend y por último

el frontend. El orden es importante ya que todos tienen dependencias, si no se ha montado correctamente el volumen de datos el backend dará error. Y si la API no está disponible cuando se lance el frontend también saltará un error. Aunque en el archivo `docker-compose.yml` se han definido las dependencias (fragmento de código18), docker-compose solo asegura que para iniciar un servicio, los que dependen de él, deben estar iniciados pero quizás todavía no aceptan conexiones.

```
    depends_on:
      - backend
      - db
```

Listing 18: Descripción de dependencias

Para ello se ha añadido una solución (el fragmento de código 19) que reintenta conectarse varias veces antes de lanzar error. Sin este sencillo comando, las conexiones eran rechazadas y la aplicación se cerraba.

```
@retry(wait=wait_fixed(2), stop=stop_after_attempt(10))
```

Listing 19: Herramienta para reintentar la ejecución de una función.

6.3. Pruebas

La realización de pruebas es una fase fundamental en el desarrollo de cualquier sistema software. Permite garantizar la calidad, fiabilidad y robustez de la aplicación, asegurando que el comportamiento del sistema sea el esperado en diferentes situaciones. Las pruebas no solo permiten detectar errores antes de que lleguen al usuario final, sino que también facilitan el mantenimiento del software y la incorporación de nuevas funcionalidades de forma controlada.

En proyectos donde intervienen múltiples componentes como bases de datos, APIs y entornos gráficos, las pruebas se vuelven aún más necesarias para comprobar que la integración entre los distintos módulos se realiza correctamente. En este contexto, resulta especialmente útil el uso de casos de prueba documentados y estructurados.

ID	CP-1
Nombre	Proceso de ETL
Descripción	Verificar que los datos se transforman y se insertan correctamente en PostgreSQL.
Condiciones previas	Archivos CSV descargados en el directorio correspondiente.
Datos de prueba	Archivo 'accidentes_mad.csv', 'centros_educativos_mad.csv', 'centros_medicos_mad.csv','datos_meteorologicos_mad.csv' y 'radadres.csv'
Pasos a ejecutar	Ejecutar el script de carga de datos
Resultados esperados	Los registros aparecen en sus tablas correspondientes sin errores
Resultados reales	Caso de prueba superado con éxito.
Caso de uso relacionado	Proceso de ingeniería de datos 1 y 2

Cuadro 7: Caso de prueba CP-1: Proceso de ETL

ID	CP-2
Nombre	Consulta del backend
Descripción	Verificar que una ruta del backend devuelve correctamente los datos solicitados en formato JSON.
Condiciones previas	La base de datos debe contener datos y el backend debe estar desplegado.
Datos de prueba	Petición GET a `/accidentes`
Pasos a ejecutar	Lanzar la petición desde Postman o navegador. Comprobar el contenido de la respuesta.
Resultados esperados	El servidor responde con un JSON válido con los datos esperados.
Resultados reales	Caso de prueba superado con éxito.
Caso de uso relacionado	Sin caso de uso asociado directo

Cuadro 8: Caso de prueba CP-2: Consulta del backend

ID	CP-3
Nombre	Manejo de errores en el endpoint
Descripción	Verificar que el backend responde adecuadamente ante una petición incorrecta o malformada.
Condiciones previas	Backend desplegado y con acceso a la base de datos.
Datos de prueba	Petición GET a `/accidentes/001` (Accidente inexistente)
Pasos a ejecutar	Realizar la petición. Observar el código de estado y mensaje devuelto.
Resultados esperados	El servidor responde con código 404 o 422 y un mensaje de error claro.
Resultados reales	El servidor ha respondido como se esperaba
Caso de uso relacionado	Sin caso de uso asociado directo

Cuadro 9: Caso de prueba CP-3: Manejo de errores en el endpoint

ID	CP-4
Nombre	Visualización de detalles de un accidente
Descripción	Comprobar que el mapa carga correctamente los accidentes y permite la interacción.
Condiciones previas	Frontend desplegado y backend aceptando peticiones.
Datos de prueba	Accidente con datos
Pasos a ejecutar	Acceder al dashboard. Hacer clic en un punto de accidente.
Resultados esperados	Se abre el panel con los detalles del accidente
Resultados reales	Caso de prueba superado con éxito.
Caso de uso relacionado	Caso de Uso: Visualización de detalles de accidentes 6

Cuadro 10: Caso de prueba CP-4: Visualización de detalles de un accidente

ID	CP-5
Nombre	Aplicación de filtros (frontend)
Descripción	Verificar que los filtros aplicados afectan correctamente al contenido del mapa y las estadísticas.
Condiciones previas	La aplicación debe estar ejecutándose con datos cargados.
Datos de prueba	Filtros: Distrito = Centro, Día = Lunes.
Pasos a ejecutar	Aplicar filtros desde el panel. Observar el cambio en el mapa y gráficos.
Resultados esperados	La interfaz se actualiza mostrando solo los datos filtrados.
Resultados reales	Caso de prueba superado con éxito.
Caso de uso relacionado	Caso de uso: Aplicar Filtros 5

Cuadro 11: Caso de prueba CP-5: Aplicación de filtros (frontend)

ID	CP-6
Nombre	Orquestación completa con Docker Compose
Descripción	Verificar que al ejecutar `docker-compose up --build` todos los servicios se levantan correctamente y en orden.
Condiciones previas	Archivos Dockerfile y docker-compose.yml configurados.
Datos de prueba	Contenedores de base de datos, backend y frontend.
Pasos a ejecutar	Ejecutar `docker-compose up --build`. Observar el orden de arranque y logs.
Resultados esperados	La base de datos, el backend y el frontend se levantan sin errores.
Resultados reales	Despliegue con éxito.
Caso de uso relacionado	Sin caso de uso asociado

Cuadro 12: Caso de prueba CP-6: Orquestación completa con Docker Compose

7

Conclusiones y trabajos futuros

Para finalizar la memoria, haremos un breve repaso sobre los objetivos cumplidos, las dificultades encontradas y comentaremos posibles ampliaciones que podrían complementar la herramienta desarrollada.

7.1. Objetivos cumplidos

El desarrollo del proyecto ha supuesto un reto personal y académico. El objetivo de realizar una herramienta que pudiese ayudar a minimizar uno de los mayores problemas sociales en materia de salud pública tenía un gran incentivo por el valor que puede aportar a la sociedad. Además, no ha sido tarea fácil gestionar y filtrar una cantidad de datos tan elevada.
Se ha logrado desarrollar una herramienta capaz de facilitar el análisis de los accidentes de tráfico en entornos urbanos, integrando y visualizando de forma efectiva una gran cantidad de datos. Para ello, se han utilizado datos abiertos reales proporcionados por organismos públicos, lo que aporta un importante valor añadido tanto en términos de realismo como de utilidad práctica.

Se han recopilado datos de distinta naturaleza, incluyendo no solo los accidentes, sino también variables externas como condiciones meteorológicas, festividades, ubicación de hospitales, centros escolares o radares. Esta diversidad ha permitido contextualizar cada accidente y enriquecer su análisis, aumentando el potencial de la herramienta para extraer conclusiones relevantes.

Los datos han sido transformados, normalizados e integrados en una base de datos relacional estructurada, lo que ha permitido mantener un buen rendimiento del sistema incluso

ante volúmenes elevados de información.

En cuanto al backend, se ha implementado una API REST funcional que permite acceder a la información estructurada y enriquecida, con endpoints bien definidos. Además consideramos haber mejorado la calidad de algunos requisitos iniciales como la actualización de datos, ya que hemos logrado que con una simple llamada al endpoint se obtengan, transformen, normalicen y almacenen todos los nuevos datos de manera automática.

El panel interactivo desarrollado permite una exploración visual clara y dinámica de los datos, facilitando la identificación de patrones y relaciones. Gracias a sus filtros y elementos gráficos, el usuario puede centrarse en variables específicas y obtener información detallada de cada accidente en tiempo real. Esta interfaz mejora notablemente la accesibilidad a los datos, incluso para usuarios no técnicos.

Además, se ha añadido valor al proyecto mediante la contenerización completa de la solución con Docker, facilitando su despliegue, portabilidad y mantenimiento, incluso en entornos con recursos limitados.

7.2. Dificultades encontradas

A lo largo del desarrollo del proyecto han surgido diversas dificultades que han hecho cambiar ligeramente el enfoque de algunas soluciones. Pero gracias a la metodología incremental iterativa se han podido adoptar soluciones rápidas, sin comprometer la estructura del proyecto.

Uno de los problemas encontrados es la ausencia de detalles en la información utilizada como entrada. Cada registro ofrece información muy generalista y poco detallada lo que hace que conocer el contexto del accidente sea una tarea complicada.
Encontrar información completa sobre el estado meteorológico en el momento del accidente ha sido una tarea difícil que ha supuesto un reto añadido al proyecto. Entre las soluciones barajadas, consideramos haber elegido la más apropiada aunque no nos ofrezca unos datos 100 % exactos que en muchos casos ofrecen valores nulos.

Otro desafío enfrentado ha sido utilizar correctamente los elementos externos a los accidentes: radares, cámaras de tráfico, paneles de información variable ... Estos elementos fueron incluidos en el conjunto de datos para ser relacionados directamente con cada uno de los accidentes, pero relacionarlos de manera directa suponía establecer un criterio que quizás no sea cierto. Es por ello que se ha preferido mostrar la información de manera paralela, para que sean los expertos quienes tomen las decisiones de si cierto elemento (bien sea un radar, o su ausencia, un centro educativo o un panel de información variable) ha sido una posible causa del accidente. Utilizando los filtros y poniendo el foco de atención en el mapa que muestra tanto accidentes como otros elementos, se pueden llegar a sacar conclusiones realistas.

7.3. Posibles ampliaciones

El desarrollo de este proyecto es solo una pequeña muestra del potencial de la herramienta. Gracias al diseño modular y la base de datos normalizada, la herramienta es altamente escalable de forma que se pueden seguir añadiendo funcionalidades sin alterar la estructura actual. Como posibles mejoras y aspectos a desarrollar en el futuro se plantea desarrollar:

- **IA predictiva:** El desarrollo de un modelo de IA predictiva, o la integración con cualquiera de los modelos actuales ya existentes, aprovecharía todo el proceso de ingeniería de datos realizado. Ya que un modelo así puede analizar miles de registros en plazos de tiempo muy reducidos y podría sacar conclusiones, predecir accidentes o generar informes de manera automática.

- **Mapa con accidentes por cercanía:** Para poder utilizar toda la información existente en la base de datos de una manera más dinámica, queda pendiente crear una serie de mapas interactivos en los que el usuario pueda seleccionar un radio de distancia y se muestren todos los accidentes que estén a dicho radio de distancia de cada centro educativo, radar, cruce, o cualquier otro elemento ya almacenado en la base de datos. Con esto se podría aumentar aún más el contexto y mejorar el análisis.

Referencias

[1] Aemet opendata. Disponible en: https://opendata.aemet.es/centrodedescargas/inicio.

[2] Astral v3.0. Documentación disponible en: https://astral.readthedocs.io/en/latest/.

[3] Calendario laboral en el ámbito de la ciudad de madrid. Disponible en: https://datos.madrid.es/sites/v/index.jsp?vgnextoid=9f710c96da3f9510VgnVCM2000001f4a900aRCRD&vgnextchannel=374512b9ace9f310VgnVCM100000171f5a0aRCRD.

[4] Canva. Disponible en: url{https://www.canva.com.

[5] Centros de asistencia médica de madrid. Disponible en: https://datos.madrid.es/sites/v/index.jsp?vgnextoid=da7437ac37efb410VgnVCM2000000c205a0aRCRD&vgnextchannel=374512b9ace9f310VgnVCM100000171f5a0aRCRD.

[6] Centros educativos de madrid. Disponible en: https://datos.madrid.es/sites/v/index.jsp?vgnextoid=f14878a6d4556810VgnVCM1000001d4a900aRCRD&vgnextchannel=374512b9ace9f310VgnVCM100000171f5a0aRCRD.

[7] Dataset de accidentes de tráfico del ayuntamiento de madrid. Disponible en: https://datos.madrid.es/sites/v/index.jsp?vgnextoid=7c2843010d9c3610VgnVCM2000001f4a900aRCRD&vgnextchannel=374512b9ace9f310VgnVCM100000171f5a0aRCRD.

[8] Datos meteorológicos históricos - ayto. de madrid. Disponible en: https://datos.madrid.es/sites/v/index.jsp?vgnextoid=fa8357cec5efa610VgnVCM1000001d4a900aRCRD&vgnextchannel=374512b9ace9f310VgnVCM100000171f5a0aRCRD.

[9] Docker - accelerated container application development. Disponible en: https://www.docker.com.

[10] Documentación beautiful soup. Disponible en: https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

[11] Geopy. Documentación disponible en: https://geopy.readthedocs.io/en/stable/#.

[12] Herramienta de visualización del conjunto de datos abiertos de accidentes. Disponible en: https://datos.madrid.es/portal/site/egob/menuitem.400a817358ce98c34e937436a8a409a0/?vgnextoid=507d8de9675b3910VgnVCM2000001f4a900aRCRD&vgnextchannel=507d8de9675b3910VgnVCM2000001f4a900aRCRD&vgnextfmt=default.

[13] Intérprete de ficheros de datos meteorológicos. Disponible en: https://datos.madrid.es/FWProjects/egob/Catalogo/MedioAmbiente/DatosMeteorologicos/Ficheros/Interpretación_datos_meteorologicos.pdf.

[14] Mapbox - mapas, navegación, búsqueda y datos. Disponible en: https://www.mapbox.com.

[15] Mygeodata - shapefile to geojson converter. Disponible en: url{https://mygeodata.cloud/converter/shp-to-geojson.

[16] Plotly express (plotly). Disponible en: https://plotly.com/python/plotly-express/.

[17] Portal de datos abiertos del ayuntamiento de madrid. Disponible en: https://datos.madrid.es/portal/site/egob/menuitem.9e1e2f6404558187cf35cf3584f1a5a0/?vgnextoid=374512b9ace9f310VgnVCM100000171f5a0aRCRD&vgnextchannel=374512b9ace9f310VgnVCM100000171f5a0aRCRD&vgnextfmt=default.

[18] Railway - infrastructure for developers. Disponible en: https://railway.com.

[19] Render - cloud hosting for developers. Disponible en: https://render.com/.

[20] Transformer (pyproj). Documentación disponible en: https://pyproj4.github.io/pyproj/stable/api/transformer.html.

[21] Generalitat de Cataluña. Ia predictiva para luchar contra los accidentes de tráfico, 2025. Disponible en: https://web.gencat.cat/es/actualitat/detall/IA-predictiva-per-lluitar-con-els-accidents-de-transit.

[22] Gobierno de España. Ley orgánica 3/2018 de protección de datos personales y garantía de los derechos digitales, 2018. Disponible en: https://www.boe.es/buscar/act.php?id=BOE-A-2018-16673.

[23] Unión Europea. Reglamento (ue) 2016/679 del parlamento europeo y del consejo, 2016. Disponible en: https://eur-lex.europa.eu/legal-content/ES/TXT/?uri=CELEX%3A32016R0679.

[24] El Mundo. España: cara y cruz de los accidentes de tráfico urbanos, Domingo, 5 febrero 2023. URL: https://www.elmundo.es/motor/2023/02/05/63dff64be4d4d81d338b457a.html.

[25] RACE. El efecto mirón, un riesgo que como conductor debes evitar. URL: https://www.race.es/efecto-miron-coche.

Apéndice A
Manual de Instalación

Este apartado describe el proceso necesario para instalar y ejecutar la herramienta localmente o en un entorno contenerizado, permitiendo su uso por parte de usuarios con conocimientos técnicos básicos. Se ofrecen dos opciones de instalación: una mediante `Docker`, recomendada por su simplicidad y portabilidad, y otra manual en caso de no disponer de esta tecnología.

A.1. Requisitos previos

Antes de proceder a la instalación, es necesario contar con:

- `Python` versión 3.13.2
- `PostgreSQL` versión 15 o superior
- `Docker` y `Docker Compose` (opcional pero recomendado)

A.2. Instalación con Docker

La forma más sencilla y rápida de ejecutar el proyecto es mediante `Docker Compose`, que lanza la base de datos, el backend y el frontend de forma orquestada. Nos situamos en el directorio raíz del proyecto y ejecutamos `docker-compose up --build` Docker se encargará de construir el volumen de datos, si es la primera vez que se ejecuta, o si se ha realizado `docker-compose down -v` anteriormente. Si ya existe el volumen entonces lo asigna y despliega el backend y el frontend.

Una vez desplegado:

- El panel interactivo estará disponible en: `http://127.0.0.1:8050`
- La API estará accesible en: `http://127.0.0.1:8000/`

A.3. Instalación manual

En caso de no disponer de Docker, se puede instalar manualmente cada componente.

A.3.1. Crear entorno virtual e instalar dependencias

```
python -m venv venv
source venv/bin/activate  # o .\venv\Scripts\activate en Windows
pip install -r app/requirements.txt
pip install -r dashboard/requirements.txt
```

A.3.2. Configurar la base de datos PostgreSQL

1. Crear una base de datos llamada `road_accidents`.
2. Ejecutar el script `init.sql` para crear las tablas:

   ```
   psql -U usuario -d road_accidents -f db/init.sql
   ```

3. Modificar el archivo `.env` con la URI de conexión a la base de datos.

A.3.3. Ejecutar el backend

```
cd app
uvicorn main:app --reload
```

A.3.4. Ejecutar el frontend

Primero debemos modificar la dirección a la que se harán las peticiones. Para ello abrimos el archivo `app_dash.py` y cambiamos `URL_BASE=DOCKER` por `URL_BASE=LOCALHOST`

```
cd ../dashboard
python app_dash.py
```

Apéndice B
Manual de Usuario

Este manual proporciona las instrucciones necesarias para que un usuario final pueda interactuar con la herramienta desarrollada. Se describen las funcionalidades principales del sistema, así como el uso del panel web para visualizar, filtrar y analizar los accidentes de tráfico urbanos.

B.1. Visualización del mapa y estadísticas

Una vez desplegada la aplicación correctamente (véase el manual de instalación), el usuario podrá acceder a través de un navegador web introduciendo la siguiente URL:

```
http://127.0.0.1:8050
```

Esto abrirá el panel de la aplicación. Nada más abrirlo los filtros se encuentran ocultos y podemos visualizar el mapa principal. En este mapa podemos diferenciar los accidentes, de color rojo, los centros educativos, de color azul, los centros médicos, de color verde y los radares, de color rojo (Figura 24).

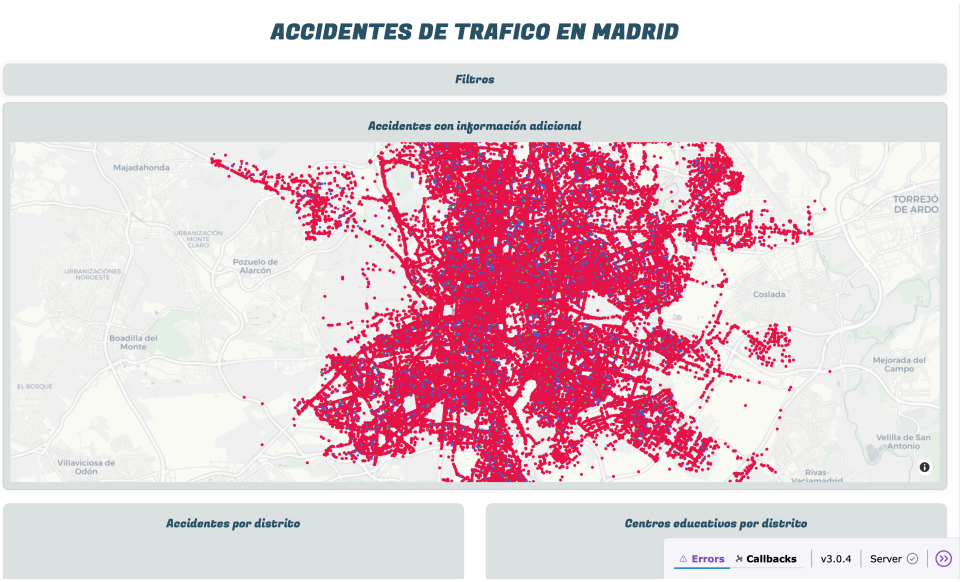

Figura 24: Vista inicial

Si continuamos deslizando hacia abajo encontramos la sección de estadísticas. Podemos ver la información que se muestra en las Figuras 25, 26 y 27.

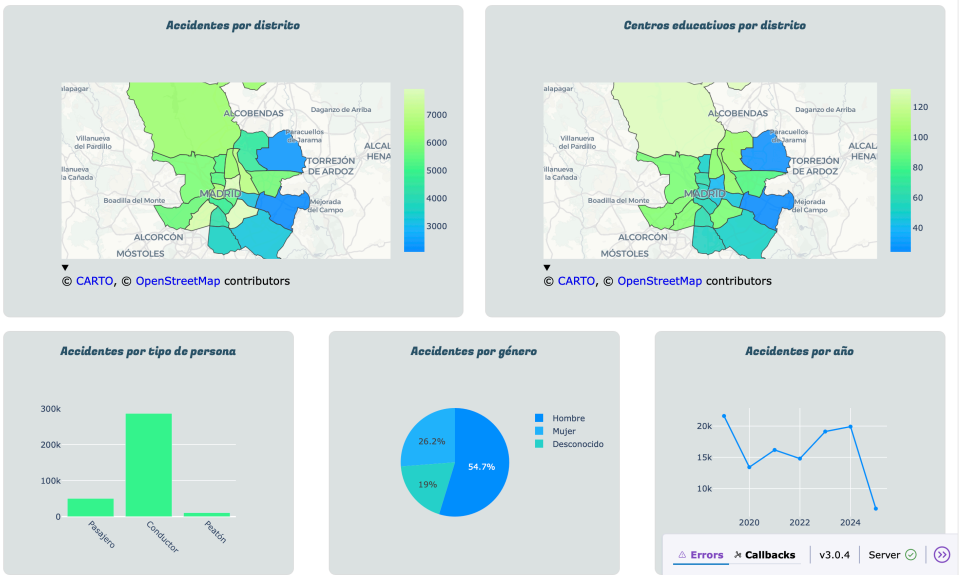

Figura 25: Vista de estadísticas I

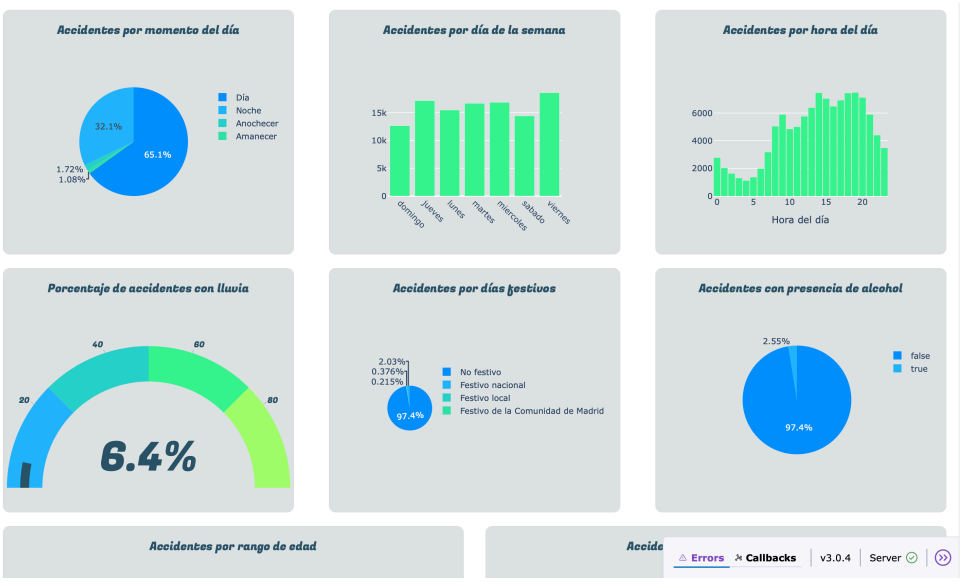

Figura 26: Vista de estadísticas II

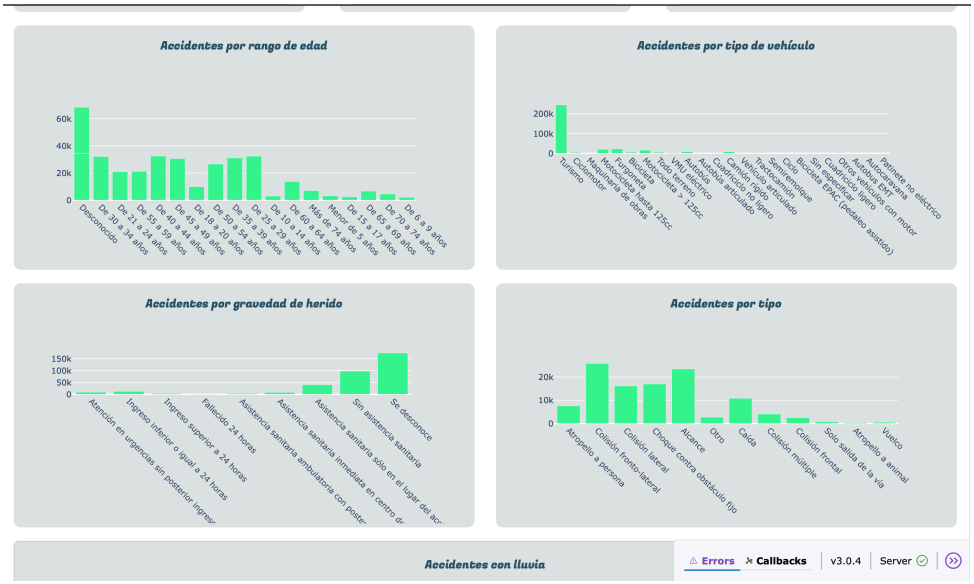

Figura 27: Vista de estadísticas III

A continuación encontramos mapas con diversa información. El primer mapa, también se actualiza según los filtros aplicados. Este mapa nos muestra los accidentes con presencia de lluvia (Figura 28).

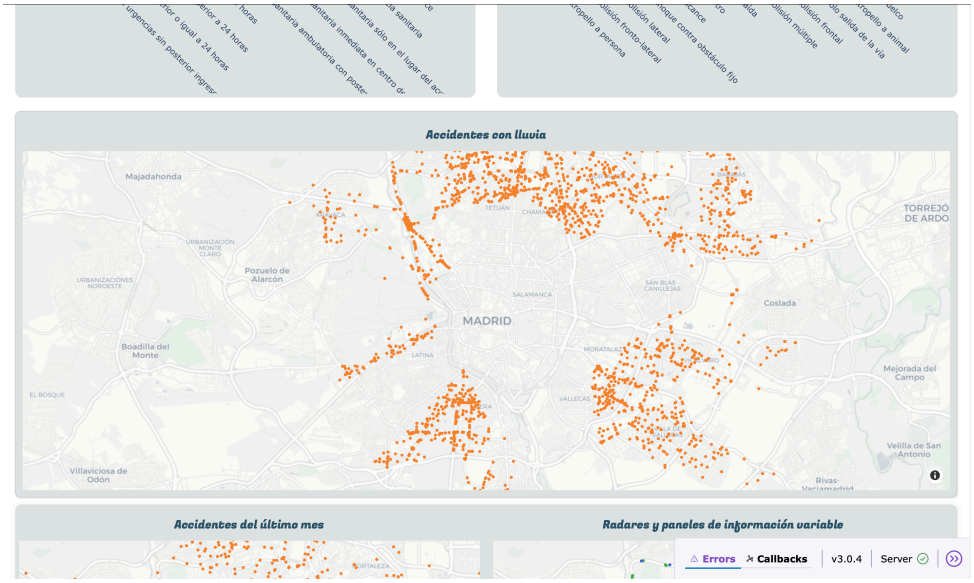

Figura 28: Vista de mapa de lluvia

Los dos mapas siguientes muestran información estática, es decir no cambia aunque apliquemos diversos filtros. En el primer mapa encontramos los accidentes del último mes y en el segundo mapa vemos la ubicación de distintos elementos de tráfico como radares y paneles de información variable (Figura 29).

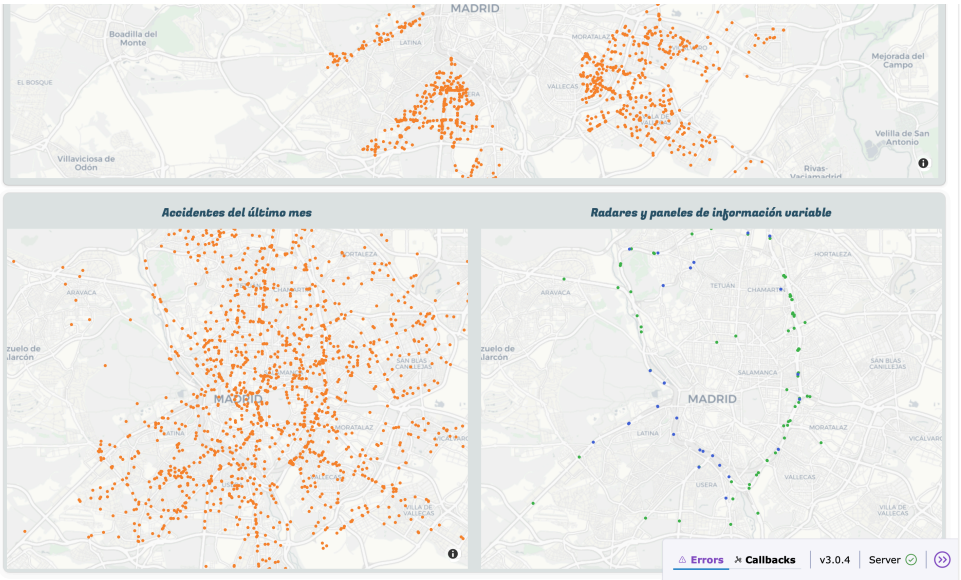

Figura 29: Vista de mapas varios

B.2. Aplicación de filtros

Los filtros se encuentran ocultos. Para poder desplegarlos debemos pulsar en `Filtros` y se abrirá un panel como el de la Figura 30.

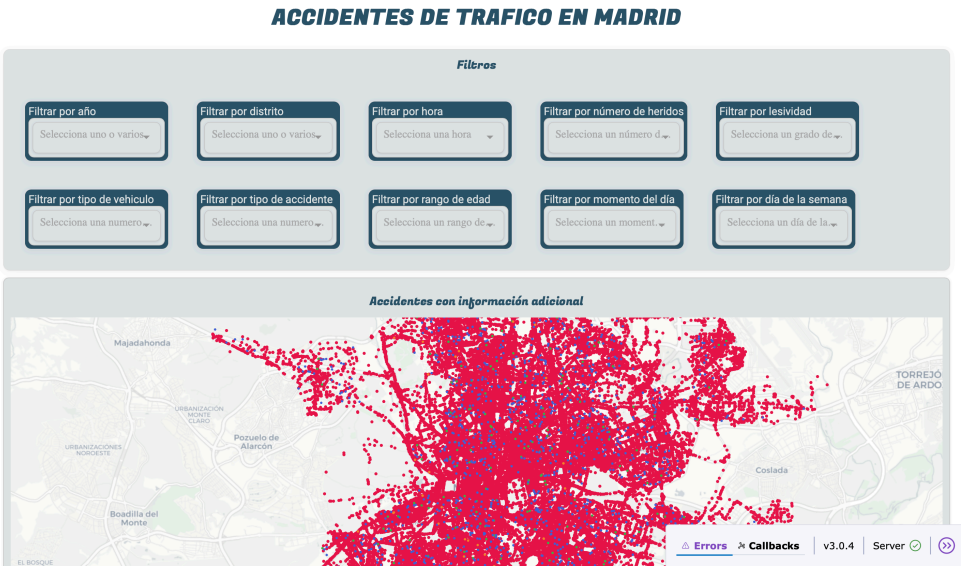

Figura 30: Vista de filtros I

Figura 31: Vista de filtros II

Podemos interactuar con todos los filtros que consideremos oportuno. Nada más seleccionar un elemento del desplegable el filtro se aplicará. Lo podemos comprobar en el mapa, ya

que con cada filtro aplicado el número de accidentes se verá reducido, tal y como se muestra en la Figura 31. Cada vez que se selecciona un filtro se actualiza automáticamente toda la información del dashboard, desde el mapa principal, estadísticas hasta el mapa de accidentes con lluvia.

B.3. Detalles de accidente

Por último, debemos mencionar la posibilidad de ver los detalles de un accidente. Podemos hacer clic en cualquier accidente del primer mapa para ver sus detalles.
Se nos abrirá un panel como el que vemos en la Figura 32
Si seleccionamos otro accidente distinto en el mapa la información se actualizará al instante. Una vez finalizado el análisis de los detalles del accidente, podemos cerrarlo haciendo clic en el botón superior derecho del panel.

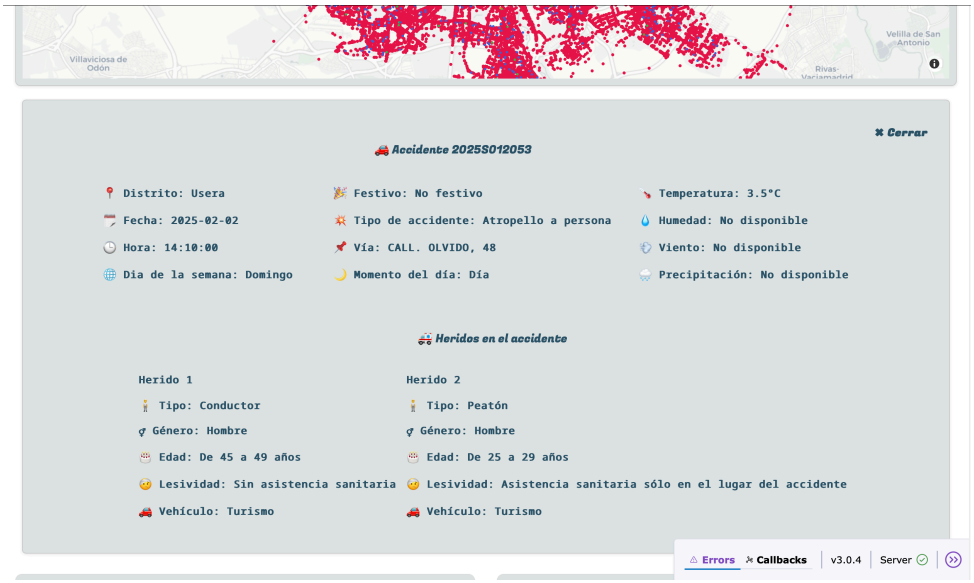

Figura 32: Vista de detalles del accidente

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