Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

# Plataforma Serverless de Procesado de Datos Abiertos

Trabajo Fin de Máster

**Máster Universitario en Gestión de la Información**

**Autor**: Jesús Ortiz Amaya

**Tutor**: Germán Moltó Martínez

2019/2020

# Resumen

Este Trabajo de Fin de Master desarrolla un proyecto serverless para procesar datos abiertos. Se basa en la creación de una aplicación compuesta por dos elementos, un script serverless desarrollado en Python y una página web estática desarrollada utilizando el framework Nuxt.js. Juntos, procesan y representan visualmente las incidencias de la vía pública de la Ciudad de Madrid, a partir de datos ofrecidos de forma abierta en el portal de transparencia del Ayuntamiento de Madrid. Como resultado, se pueden obtener en la página web estática gráficos que aportan información sobre los incidentes, así como un mapa que ubica y describe cada incidente en la ciudad. La página web es actualizada diariamente de forma automática y se puede desplegar sin necesidad de crear una infraestructura previa. Para ello se hace uso de la plataforma Cloud de Amazon Web Services.

**Palabras clave:** serverless, datos abiertos, Python, Nuxt, AWS Lambda.

# Abstract

This Master's Thesis develops a serverless project to process open data. It explains the creation of an application that is composed of two elements, a serverless script developed in Python and a static web page developed using the Nuxt.js framework. With these two elements, the application can process and represent the incidents on the public roads of the city of Madrid from, data offered openly on the transparency portal of the Madrid City Council. As a result, it can be obtained a variety of graphics on the static website that provide information about the incidents, as well as a map that locates and describes each incident in the city. The website is updated daily automatically and can be deployed without create a previous infrastructure. To accomplish this, the Amazon Web Services Cloud platform is used.

**Keywords :** serverless, open data, Python, AWS Lambda.



# Tabla de contenidos

# Tabla de ilustraciones

# 1. Introducción

La tecnología Cloud fue creada para proporcionar servicios como gran capacidad de almacenamiento o cómputo, que hasta entonces solo se podían obtener si se disponían de equipos con un alto rendimiento. Esta tecnología cambia por completo el ámbito informático, ya que permite acceder a sus servicios a cualquier cliente, independientemente de si es un particular o una empresa, lo único que se necesita es disponer de acceso a Internet.

Cloud Computing supuso una alternativa a las soluciones hasta entonces clásicas, denominadas *on-premises*, que consistían en la creación de una infraestructura informática, donde se incluyen tanto materiales físicos como equipos, discos duros o procesadores, como material software, como sistemas operativos o sistemas de gestión de bases de datos. A estas soluciones, había que añadir el personal necesario para mantener y proporcionar los servicios de estas infraestructuras, además de tener en cuenta aspectos como la disponibilidad del servicio o la seguridad del sistema.

La tecnología Cloud supone un cambio en este paradigma, pues ya no es necesario crear toda la infraestructura. Con la tecnología Cloud, se puede comprar y gestionar toda o parte de la infraestructura previa, evitando de esta manera la necesidad de equipos físicos o de personal especializado en estos. Con la tecnología Cloud, se pueden comprar equipos virtuales de cómputo que sustituyan a los equipos físicos o se pueden contratar sistemas de almacenamiento que reemplacen los discos duros. Se crea por lo tanto el pago por despliegue, lo que se traduce, en pagar únicamente por los recursos que se necesitan en la plataforma Cloud. Una de estas plataformas, es Amazon Web Services (AWS) [2] la plataforma Cloud más utilizada en el momento en el que se desarrolla el trabajo.

AWS ofrece cada año nuevos servicios que incorpora a su plataforma. Uno de estos servicios, es AWS Lambda [3]. Este servicio supuso una segunda revolución dentro del ámbito informático, ya que evolucionaba del pago por despliegue al pago por uso. Sigue un modelo serverless, es decir, un modelo de ejecución en el que el proveedor Cloud ejecuta y administra dinámicamente la asignación de recursos. Con la tecnología Cloud de hasta el momento, se debía comprar y gestionar la infraestructura virtual. Por el contrario, con AWS Lambda, tan solo se debe de proporcionar el código ejecutable del servicio, siendo AWS quien crea la infraestructura necesaria de forma automática, además de encargarse de aspectos como el escalado o la replicación. Con AWS Lambda, solo se paga el tiempo que dura el código ejecutándose, por lo que cuando la función no se está ejecutando lleva asociado un coste cero.

AWS Lambda ofrece la oportunidad de crear un servicio sin necesidad de tener conocimientos sobre creación y gestión de infraestructuras informáticas, ya que lo único que se requiere es el código del programa. Por ello, se ha propuesto la creación de este trabajo, el cual pretende crear una aplicación que contenga una página web estática que muestre información relativa a las incidencias en la vía pública de la ciudad de Madrid, aportando gráficos con información relacionada y un mapa con la ubicación de las incidencias, siendo capaz de actualizarse de forma automática y sin la necesidad

de aprovisionar una infraestructura previa. La aplicación se desarrolla utilizando AWS Lambda, así como otros servicios ofrecidos por AWS. La página web propuesta seguirá un modelo serverless para que, de esta manera, se pueda evitar la gestión de servidores y, además, aprovechar las ventajas de este modelo de ejecución, siendo la forma más barata, escalable y segura para exponer una página web estática.

Los datos utilizados para este proyecto son ofrecidos públicamente por el Ayuntamiento de Madrid [12], siguiendo de esta forma la política de datos abiertos. Esta política defiende que los datos deben estar disponibles para todo el mundo, sin restricciones de uso o mecanismos de control. Por parte de los gobiernos, se sigue esta política con el fin de promocionar e incentivar la innovación tecnológica, la creación de nuevas aplicaciones y de nuevos servicios digitales.

## 1.1 Motivación

Este proyecto se crea con la idea de ofrecer un servicio a partir de la obtención y el procesamiento de datos abiertos. Dentro del Máster Universitario de Gestión de la Información de la Universidad Politécnica de Valencia, se expone la importancia de los datos abiertos, así como las posibilidades que ofrecen. La mayoría de las veces, estos datos quedan destinados al olvido, sin que se acaben usando ni creando servicios relacionados con ellos. Por esta razón, se crea este proyecto, para defender e impulsar el uso de datos abiertos como una forma de crear nuevas aplicaciones y servicios que ofrezcan estos datos al público general con un formato útil y comprensible.

Por otra parte, se decidió utilizar un servicio que en el momento en el que se desarrolla el trabajo ha supuesto una revolución en las plataformas Cloud. Este servicio es AWS Lambda, el cual permite la creación de aplicaciones con un bajo coste asociado y sin necesidad de tener infraestructura pre-aprovisionada. Por ello, se decide utilizar AWS Lambda, como forma de crear un nuevo servicio a partir de datos abiertos, utilizando una tecnología innovadora, tal y como propone uno de los objetivos de la política de datos abiertos.

## 1.2 Objetivos

El proyecto presentado en esta memoria se puede dividir en los siguientes objetivos primarios:

- Identificar un dataset público que cumpla con los siguientes requisitos:
    - Sea actualizado diariamente.
    - Se mantenga actualizado durante las fechas en las que se desarrolla el trabajo y previsiblemente tras la finalización de este.

- Programar un script que sea capaz de cumplir con las funciones que se exponen a continuación:
    - Descargar los datos del dataset público.
    - Realizar un proceso de transformación y limpieza sobre los datos para que puedan ser representados de una forma visual en una página web estática.

- Desarrollar una página web estática que pueda representar visualmente los datos previamente mencionados. La representación debe ser interactiva con el usuario y la página web debe tener la capacidad de actualizar los datos de forma automática y sin intervención del usuario ni el desarrollador.

Como se puede observar, los objetivos primarios están divididos según los componentes del proyecto, siendo el dataset público, el script y la página web estática. A continuación, se exponen los objetivos secundarios del proyecto, siguiendo la misma división:

- El dataset elegido debe cumplir el requisito de ofrecer los datos con un formato legible por máquinas, preferiblemente optando por archivos CSV, XML o JSON.

- El script debe consumir los mínimos recursos posibles ofrecidos por AWS Lambda, para que de esta manera la aplicación sea lo más barata posible. Por lo tanto, la duración debe estar contenida entre 5 y 15 segundos y su utilización de memoria debe ser inferior a 128 MB, las cantidades más pequeñas ofrecidas por AWS Lambda.

- La página web debe tener una alta disponibilidad, preferiblemente mayor al 99%.

## 1.3 Estructura

Esta memoria se estructura en seis secciones, siendo la primera la introducción previamente expuesta. La segunda sección expone las tecnologías empleadas a lo largo del proyecto. La tercera sección muestra el análisis realizado sobre diferentes portales de datos abiertos, para elegir un dataset público que cumpla con los requisitos definidos. En la cuarta, se detallan los pasos que se han seguido para desarrollar el script y la página web, los dos elementos fundamentales de este proyecto. La sección quinta trata sobre el despliegue de los elementos desarrollados en la sección cuarta en la plataforma Cloud de Amazon Web Services. Por último, se exponen las conclusiones del proyecto, así como los trabajos futuros previstos.

# 2. Tecnologías empleadas

En esta sección se exponen las tecnologías, plataformas y librerías que se han utilizado para desarrollar la aplicación presentada en este trabajo. Está dividida en cuatro puntos: el primer punto explica el lenguaje de programación Python, utilizado para desarrollar el código que permite descargar el dataset público, procesar los datos que contiene y almacenarlos de tal forma que la página web estática pueda acceder a ellos. El segundo presenta el framework Vue que, junto al tercero, que expone el framework Nuxt.js, se han utilizado para desarrollar la página web estática y, por último, se expone la plataforma Amazon Web Services, la plataforma Cloud que se ha utilizado para desplegar la aplicación.

## 2.1 Python

Python es un lenguaje de programación interpretado cuya filosofía tiene como objetivo la legibilidad del código. Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y en menor medida, programación funcional. Es un lenguaje interpretado, dinámico y multiplataforma. Es administrado por Python Software Foundation [31]. Posee una licencia de código abierto, denominada Python Software Foundation License [30].

Python incluye un modo interactivo por el cual se escriben las instrucciones en un intérprete de comandos, es decir, las expresiones pueden ser introducidas una a una, pudiendo verse el resultado de su evaluación inmediatamente.

Como se ha mencionado anteriormente, Python fue diseñado para ser leído con facilidad. Una de sus características es el uso de palabras, por ejemplo, el uso de las palabras reservadas *not*, *or* y *and*, mientras que otros lenguajes utilizan símbolos como *¡*, || y *&&,* para representar las anteriores palabras.

Los archivos que contienen código Python son exportados en archivos con extensiones *.py*. Cada archivo *.py* puede estar compuesto por una o varias funciones Python. El contenido de las funciones de Python se divide en bloques de código, delimitados mediante espacios o tabuladores antes de cada línea de órdenes perteneciente a un bloque.

Un ejemplo de cómo se distribuyen las funciones y los bloques de código es el que se muestra en la ilustración 1:

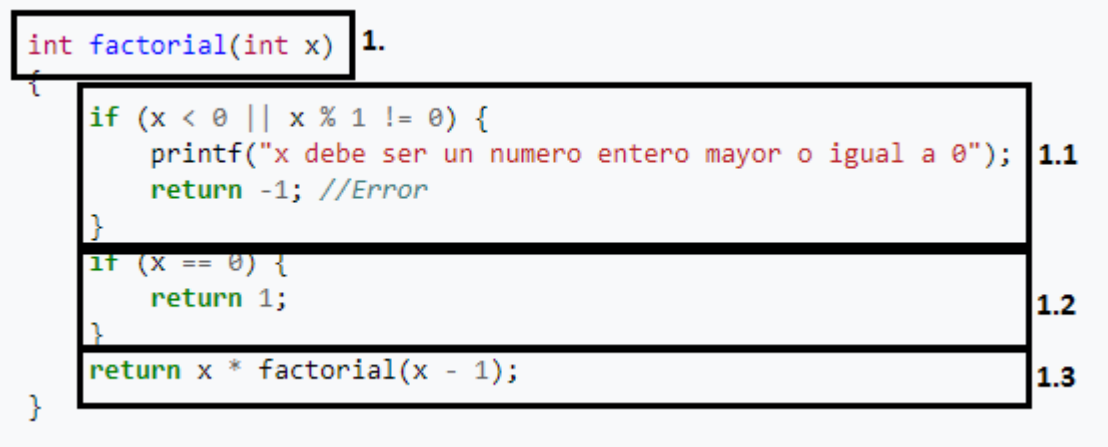


*Ilustración 1. Ejemplo de una función Python. Fuente: Elaboración propia.*

En la ilustración 1, se puede observar el ejemplo de una función llamada *factorial* (1.), y los tres bloques de código que contiene, el bloque 1.1, 1.2 y 1.3. Cada bloque se puede identificar debido a la delimitación de espacio que existe entre órdenes de distintos bloques. Por ejemplo, el primer bloque empieza con el primer *if* (1.1), y sus siguientes órdenes mantienen una tabulación respecto al inicio del bloque. Como el segundo *if* (1.2) es un nuevo bloque de código, la tabulación deja de mantenerse, para que, de esta forma, se pueda identificar el final de un bloque de código.

Para este proyecto, Python se ha utilizado como lenguaje de programación para crear un script que sea capaz de descargar los archivos almacenados en el dataset público, para posteriormente procesar los datos y extraer aquellos que se quieren representar en la página web estática. Para ello una vez se ha transformado y limpiado los datos, la función los exporta en archivos con extensión JavaScript (*.js*), para que la página web estática sea capaz de leerlos.

### 2.1.1 Anaconda

Anaconda [9] es una plataforma libre y de código abierto que permite programar funciones en los lenguajes de programación Python y R, utilizados para la ciencia de datos y para el aprendizaje automático. Está disponible para Windows, Linux y macOS.

Para este proyecto, Anaconda se ha utilizado como plataforma para programar el script Python. Se ha decidido utilizar esta plataforma ya que permite la utilización del lenguaje de programación Python en el sistema operativo Windows.

## 2.2 Vue.js

Vue.js [36] es un framework JavaScript de código abierto para construir interfaces de usuario y aplicaciones de una sola página (SPA). Vue.js presenta una arquitectura adaptable que se centra en la representación declarativa y en la composición de una aplicación a través de componentes.

Los componentes Vue extienden los elementos HTML básicos para encapsularlos en código reutilizable. Los componentes son elementos personalizados a los que el compilador de Vue asocia un comportamiento.

Vue.js presenta un sistema de reactividad que utiliza objetos JavaScript. Cada componente realiza un seguimiento de sus dependencias reactivas durante su renderizado, por lo que el sistema sabe cuándo volver a renderizar y qué componentes volver a renderizar.

Vue utiliza una sintaxis de plantilla basada en HTML, que permite vincular el Document Object Model (DOM [1]) representado a los datos de la instancia Vue subyacente. Todas las plantillas de Vue son HTML válidos que pueden analizarse mediante navegadores y analizadores HTML. Vue compila las plantillas en funciones virtuales DOM. Un DOM virtual permite a Vue.js representar componentes en la memoria antes de actualizar el navegador. Combinado con el sistema de reactividad de Vue.js, se puede calcular la cantidad mínima de componentes para volver a renderizar y aplicar la cantidad mínima de manipulaciones DOM cuando cambia el estado de la aplicación.

La librería principal de Vue.js se centra solo en la capa de vista. Las características avanzadas requeridas para aplicaciones complejas como enrutamiento, administración de estado y herramientas de compilación se ofrecen a través de librerías y paquetes de soporte mantenidos oficialmente, con el framework Nuxt.js como una de las soluciones más populares.

## 2.3 Framework Nuxt.js

Nuxt.js [17] es un framework web gratuito y de código abierto basado en Vue.js, Node.js, Webpack y Babel.js. El objetivo de Nuxt.js es permitir a los usuarios crear páginas web en JavaScript utilizando el sistema de componentes de Vue.js y que puedan funcionar como una aplicación de una sola página en el navegador (SPA), así como vistas web renderizadas por el servidor. Además, el framework permite a los usuarios mantener el contenido o partes de él, totalmente prerrenderizados en el servidor y que puedan ser ofrecidos en forma de sitios web estáticos.

Los beneficios de este enfoque son: tiempo reducido para la interactividad y un Search Engine Optimization (SEO) mejorado en comparación con los SPA, debido al

[1] El Document Object Model (DOM) es una interfaz multiplataforma e independiente del lenguaje que trata un documento XML o HTML como una estructura de árbol en la que cada nodo es un objeto que representa una parte del documento. El DOM representa un árbol lógico. Cada rama del árbol termina en un nodo, y cada nodo contiene objetos.

hecho de que el servidor web sirve el contenido completo de cada página antes de que se ejecute el código JavaScript en el lado del cliente. Es decir, se pueden mantener los beneficios de las páginas HTML tradicionales del lado del servidor y la interactividad mejorada y la interfaz del usuario avanzada de los SPA. El beneficio principal del framework Nuxt.js es que el desarrollador puede crear componentes de la aplicación como si fueran archivos Vue.js.

Para desarrollar este proyecto, se ha utilizado el framework Nuxt.js, para crear la página web estática que representa los datos extraídos del dataset público, y poder visualizarlos. Para ello, se han utilizado los plugins que se detallan en las siguientes subsecciones.

### 2.3.1 Análisis de librerías gráficas

En esta subsección se presenta el análisis realizado sobre distintas librerías gráficas que se utilizan para representar datos en forma de gráficos. En este análisis, se presentan diversas propuestas, así como la librería que finalmente se ha utilizado para representar los datos extraídos del dataset público:

- **D3.js [16]:** Es una librería de JavaScript para producir, a partir de datos, infogramas dinámicos e interactivos en navegadores web. Hace uso de tecnologías como SVG, HTML5, y CSS. Esta librería es la sucesora de la librería Protovis [25]. En contraste con muchas otras librerías, D3.js permite tener el control completo sobre el resultado visual final.

- **Matplotlib [32]:** Es una librería para la generación de gráficos a partir de datos contenidos en listas o arrays en el lenguaje de programación Python y su extensión matemática NumPy. Proporciona una API, denominada *pylab*. Es útil para obtener una idea gráfica de los datos.

- **Seaborn [35]:** Es una librería de visualización construida sobre Matplotlib. Puede crear gráficos más complejos y detallados que Mathplotlib siendo estos visualmente más atractivos, personalizables y con la posibilidad de añadir color.

- **Bokeh [15]:** Es una librería de visualización para navegadores web. Puede generar visualizaciones complejas, dando la opción de configurar distintos atributos dentro de cada visualización para personalizarlas y se utiliza para crear presentaciones interactivas basadas en web.

- **Pygal [26]:** Es una librería de visualización disponible en el lenguaje de programación Python para generar gráficos vectoriales. Se utiliza para generar gráficos basados en vectores y para crear archivos interactivos.

- **Plotly [29]:** Proporciona herramientas para la generación de gráficos, análisis y estadísticas. Contiene librerías de gráficos para Python, R, MATLAB, Perl, Julia, Arduino y REST. Es una opción para crear visualizaciones altamente interactivas basadas en web.

- **Vue-charts.js [23]:** Es una librería para crear gráficos JavaScript en Vue.js como componentes. La ventaja de esta librería es que cada componente que se crea se puede reutilizar posteriormente. Es útil para crear gráficos sencillos, que se renderizan con rapidez en las páginas webs.

- **Apexcharts.js [10]:** Es una librería de gráficos que ayuda a los desarrolladores a crear visualizaciones interactivas para páginas web. Es un proyecto de código abierto con licencia del MIT y es de uso gratuito en aplicaciones comerciales.

Las anteriores librerías se analizaron para escoger cuál de ellas se ajustaba a los objetivos de la página web estática, los cuales eran: la creación de gráficos interactivos y que los gráficos fueran reactivos, es decir, que se actualizarán cuando los datos que representan cambien de valor.

Para lograr satisfacer los anteriores requisitos, se plantearon dos opciones: la primera, crear los gráficos a través del lenguaje de programación Python o la segunda opción, generar los gráficos usando una librería JavaScript, la cual estaría instalada como plugin en los archivos de la página web estática.

La primera opción, requiere la instalación de la librería en el script Python y posteriormente, programar dentro del script la creación de los gráficos deseados. Para que estos gráficos puedan mostrarse en la página web, se deben exportan en formato de archivos de imágenes. Sin embargo, está aproximación tiene un inconveniente, y es que los gráficos no serían interactivos, ya que la página web los mostraría como imágenes estáticas. Otro inconveniente descubierto en esta opción es que cada vez que los datos cambian, se debe crear de nuevo cada gráfico y posteriormente exportarlos, afectando al rendimiento de la función Python, lo que provoca la necesidad de un mayor tiempo de ejecución y de utilización de memoria, afectando al costo asociado a la función.

La segunda opción, requiere la instalación de la librería como un plugin dentro de los archivos de la página web estática. Esta aproximación necesita leer los datos que se quieren graficar a partir de archivos. Con esta opción, se pueden crear gráficos interactivos ya que las representaciones se renderizan junto a la página web y no se muestran como imágenes estáticas. También son reactivos, ya que cada vez que se modifica un dato dentro de los archivos leídos, lo que cambia es el valor representado, evitando de esta manera sobrecargar la función Python.

Por tanto, observando el análisis expuesto, se descartó el uso de librerías gráficas en la función Python, dejando tres posibles opciones: D3.js, Vue-charts.js y Apexcharts.js.

Para realizar la elección final, se descartó la librería D3.js debido a que daba problemas de incompatibilidad a la hora de usarla junto al framework Nuxt.js. La librería Vue-charts.js se descartó debido a las pocas opciones de personalización que ofrecía frente a las que se podían escoger en la librería Apexcharts.js. Por tanto, para desarrollar la aplicación se escogió finalmente Apexcharts.js, una librería capaz de generar una gran variedad de gráficos (se puede observar la lista completa en [11]), y que ofrece flexibilidad a la hora de crearlos, pudiendo configurarlos para generar distintos tipos de gráficos 2D o 3D, los colores utilizados, el tamaño, los títulos y la

interacción con ellos. Además, es compatible con el framework Nuxt.js, pudiendo de esta manera, crear los gráficos como componentes de la página web estática y reutilizarlos.

### 2.3.2 Vue2-google-maps

En esta subsección se presenta el plugin vue2-google-maps [28], que se ha utilizado para ubicar los datos del dataset público.

Este plugin utiliza la longitud y latitud para ubicar cada dato en un mapa en forma de marcador. Para definir qué mapa se quiere representar, este también se debe configurar declarando su latitud, longitud y un atributo llamado *zoom*, el cual permite personalizar la cercanía o la lejanía con la que se va a mostrar el mapa.

Este plugin es compatible con el framework Nuxt.js y se instala dentro de los archivos de la página web, al igual que pasa con la librería gráfica Apexcharts.js. Este plugin necesita una *API key* de Google Cloud Platform [21], la plataforma Cloud de Google, con permiso para utilizar el servicio de Google Maps [20].

## 2.4 Amazon Web Services

Amazon Web Services es una plataforma Cloud que ofrece más de 175 servicios web a nivel global. Es utilizada por más de un millón de clientes, entre los que se encuentran tanto empresas emergentes, como grandes empresas y organismos gubernamentales. Es la plataforma líder en servicios en la nube. Las características de esta plataforma son:

- Funcionalidad: AWS ofrece múltiples servicios y características entre los que se encuentran: tecnologías de infraestructura como cómputo, almacenamiento y bases de datos, tecnologías emergentes como aprendizaje automático, inteligencia artificial o Internet de las Cosas. AWS también ofrece una amplia variedad dentro de sus servicios, como distintos tipos de bases de datos o de tecnologías de almacenamiento con el fin de ajustar el costo asociado y el rendimiento.

- Seguridad: AWS está diseñado para ser un entorno flexible y seguro. Cuenta con un amplio conjunto de herramientas de seguridad en la nube, con 230 servicios y características de seguridad, conformidad y gobernanza. Además, AWS es compatible con 90 estándares de seguridad y certificaciones de conformidad, y los 117 servicios de AWS que almacenan datos de los clientes ofrecen la función de cifrar los datos.

- Innovación: AWS ofrece la posibilidad de usar tecnologías emergentes como AWS Lambda, que permite que los desarrolladores ejecuten su código sin aprovisionar ni administrar servidores o Amazon SageMaker, un servicio de aprendizaje automático completamente administrado que permite a los desarrolladores y científicos utilizar el aprendizaje automático, sin necesidad de tener experiencia previa.

- Red global de regiones: AWS tiene una infraestructura subyacente para ofrecer regiones con múltiples zonas de disponibilidad conectadas por redes de baja latencia, alto rendimiento y redundantes. Incluye 76 zonas de disponibilidad en 24 regiones geográficas de todo el mundo.

Entre los servicios que ofrece AWS, para desarrollar la aplicación que se presenta en esta memoria se han hecho uso tres de ellos: Amazon CloudWatch, AWS Lambda y Amazon S3. Estos servicios son explicados de forma más detallada en las siguientes subsecciones.

### 2.4.1 Amazon CloudWatch

Amazon CloudWatch [1] es un servicio de monitorización de recursos y aplicaciones de AWS que se ejecuta en tiempo real. CloudWatch se puede utilizar para recopilar y hacer un seguimiento de métricas, que son las variables que pueden medir los recursos y las aplicaciones del usuario.

La página de CloudWatch muestra automáticamente métricas sobre todos los servicios de AWS que el usuario utilice. También se pueden crear paneles para mostrar métricas sobre las aplicaciones personalizadas, y mostrar colecciones personalizadas de métricas.

Se pueden crear alarmas que observen métricas y que envíen notificaciones o que realicen cambios automáticamente en los recursos que se está monitorizando cuando se infringe un umbral. Por ejemplo, se puede monitorizar el uso de la CPU y las lecturas y escrituras de disco de una instancia de cómputo, y a continuación, utilizar estos datos para determinar si se deben lanzar instancias adicionales para gestionar el aumento de la carga de trabajo. También se pueden utilizar estos datos para parar las instancias infrautilizadas para ahorrar en costes.

Con CloudWatch, se puede obtener información sobre la utilización de recursos, el desempeño de aplicaciones y el estado operativo de todo el sistema.

CloudWatch también tiene una funcionalidad llamada CloudWatch Events [8], la cual permite la generación de eventos. Para generar un evento, se deben definir reglas. Cuando se cumplen las condiciones de una regla, es entonces cuando se genera un evento. Existen dos tipos de reglas: el primer tipo se genera cuando se detecta una operación sobre un recurso o servicio de AWS, el segundo tipo se genera de forma programada, pudiendo crear reglas tipo *Cron*, que se ejecutan en un horario establecido de forma periódica. Estos eventos se pueden utilizar para activar servicios de AWS, como puede ser la ejecución de una función de AWS Lambda o el envío de un mensaje de Amazon SNS [4].

Para el desarrollo de este trabajo, CloudWatch se utiliza para lanzar un evento periódicamente para que se active una función en AWS Lambda, que está dedicada a obtener y procesar los datos del dataset público.

### 2.4.2 AWS Lambda

AWS Lambda es un servicio de informática sin servidor (serverless) que ejecuta código en respuesta a eventos y administra automáticamente los recursos informáticos subyacentes.

Lambda ejecuta el código en una infraestructura de alta disponibilidad y se encarga de la administración integral de los recursos informáticos, incluido el mantenimiento del servidor y del sistema operativo, el aprovisionamiento de capacidad y el escalado automático, la implementación de parches de seguridad y código, así como la monitorización de código y los registros. Lo único que se tiene que proporcionar a Lambda es el código de la función.

El código que se ejecuta en AWS Lambda se le denomina función Lambda. Después de crear una función Lambda, está siempre estará lista para ejecutarse cuando se active. Para crear una función Lambda, se debe indicar el código a ejecutar, así como otros parámetros, como la definición de variables de entorno, el rol que va a utilizar, el lenguaje usado por la función, la cantidad de memoria utilizada y el tiempo máximo que puede durar la función en ejecución.

Las funciones Lambda no tienen estado y no tienen ninguna afinidad con la infraestructura subyacente, por lo que AWS Lambda puede lanzar copias de la función si resultan necesarias para ajustar la escala de eventos entrantes.

Lambda tiene integrada la funcionalidad de tolerancia a errores. AWS Lambda mantiene la capacidad de cómputo en varias zonas de disponibilidad en cada región para ayudar a proteger el código frente a fallos en equipos individuales o fallos en las instalaciones del centro de datos.

AWS Lambda invoca el código sólo cuando resulta necesario y se escala automáticamente para atender el porcentaje de solicitudes entrantes sin que sea necesario que el usuario realice ninguna configuración adicional. No hay ningún límite en cuanto al número de solicitudes que el código puede gestionar.

Con AWS Lambda, se sigue la estrategia de pagar por la duración de la ejecución en lugar de hacerlo por la unidad de servidor. Cuando se utilizan las funciones Lambda, solo se paga por las solicitudes que se atienden y por el tiempo de cómputo que se necesita para ejecutar el código.

Para este trabajo, AWS Lambda se ha utilizado para ejecutar el código que permite descargar los datos del dataset público, procesarlos y escribirlos en archivos almacenados en un bucket de S3 para que, de esta manera, la página web estática pueda representarlos visualmente.

### 2.4.3 Amazon S3

Amazon Simple Storage Service (Amazon S3) es un servicio de almacenamiento de objetos. Los clientes pueden utilizarlo para almacenar y recuperar cualquier cantidad de datos. Amazon S3 proporciona características de administración que permite a los

clientes organizar los datos y configurar los controles de acceso a los objetos. Amazon S3 está diseñado para ofrecer una durabilidad del 99,99999999%.

Amazon S3 almacena los datos como objetos dentro de buckets. Un objeto consiste en un archivo y opcionalmente cualquier metadato que describa ese archivo. Para almacenar un objeto en Amazon S3, se debe cargar el archivo que se desea almacenar dentro de un bucket. Cuando el archivo está cargado, se pueden establecer los permisos en el objeto.

Los buckets son los contenedores de los objetos. Se pueden tener uno o más buckets. Por cada bucket, se puede controlar el acceso al mismo, quién puede crear, eliminar o enumerar los objetos dentro del bucket, ver los registros de acceso al bucket o elegir la región geográfica donde se almacenará el bucket y su contenido.

Para desarrollar este proyecto, se ha utilizado Amazon S3 para almacenar la página web estática, así como los datos extraídos del dataset público para que puedan ser representados gráficamente por la página web.

# 3. Análisis del problema

Para realizar el proyecto que se ha propuesto, el primer paso es localizar una base de datos que contenga datos abiertos, los cuales se extraerán y posteriormente, se realizarán operaciones sobre ellos para visualizarlos en una página web estática.

En esta sección, se presenta el análisis realizado sobre tres páginas que ofrecen bases de datos abiertas. Para localizar una base de datos que se pudiera utilizar para este proyecto, se analizaron los siguientes portales de datos abiertos: el portal de transparencia y datos abiertos del Ayuntamiento de Valencia, el portal de datos abiertos del Gobierno de España y, por último, el portal de datos abiertos de la Unión Europea.

Tras exponer el análisis sobre los portales de datos, se detalla la base de datos elegida, así como los datos que ofrece, para que, de esta manera, se puedan saber cuáles de ellos se van a utilizar y a representar en la página web estática.

## 3.1 Elección de la base de datos abierta

### 3.1.1 Portal de transparencia y datos abiertos del Ayuntamiento de Valencia

El primer portal analizado es el portal de transparencia y datos abiertos del Ayuntamiento de Valencia [13].

En este portal se pueden localizar múltiples conjuntos de datos abiertos, clasificados en distintas temáticas. Las temáticas que contienen una mayor cantidad de datos abiertos son: medio ambiente, sociedad y bienestar, transporte, urbanismo e infraestructuras, salud y, por último, turismo.

Entre estos conjuntos de datos, se busca una base de datos que cumpla con los requisitos establecidos en la sección Objetivos (1.2). Estos requisitos son: el periodo de actualización de la base de datos debe ser diario, se debe mantener actualizado durante las fechas en las que se desarrolla el trabajo y que se prevea su continuación tras la finalización de este y, por último, que ofrezca los datos en un formato legible por máquinas. Si el dataset cumple con estos requisitos, la aplicación propuesta puede ofrecer información actualizada y mantenida sobre el tiempo.

A continuación, se muestran los resultados obtenidos tras realizar un análisis sobre los conjuntos de datos ofrecidos por este portal, así como los posibles conjuntos que se pueden utilizar para el proyecto:

- De todos los datasets ofrecidos por el portal de transparencia y datos abiertos del Ayuntamiento de Valencia, en el momento en el que se realizó el análisis, solo uno cumplía parcialmente los requisitos exigidos. Los demás conjuntos no habían sido actualizados desde un rango de fechas que cubre el año 2015 como año más antiguo y el año 2019 como el año más contemporáneo.

- El dataset que cumplía parcialmente con las exigencias es el que se muestra a continuación:

- **Recursos sociales para personas sin techo.** Su última actualización fue el día en el que se analizó este dataset, por lo que el dataset se actualizaba hasta la fecha en la que se procedió a realizar el análisis. Su frecuencia de actualización es semanal. Se creó el 20/11/2014.

### 3.1.2 Portal de datos abiertos del Gobierno de España

El segundo portal analizado es el portal de datos abiertos del Gobierno de España [19].

En este portal se publican conjuntos de datos de todos los gobiernos autonómicos, ayuntamientos y datos recogidos por el propio gobierno de España. En este portal de datos, al igual que pasaba con el portal del Ayuntamiento de Valencia, se pueden encontrar los conjuntos clasificados en distintas temáticas. Los temas que contienen más datos son: sector público, medio ambiente y, por último, sociedad y bienestar.

A continuación, se muestran los resultados obtenidos tras realizar un análisis sobre los conjuntos de datos ofrecidos por este portal, así como los posibles conjuntos que se pueden utilizar para el proyecto:

- La mayoría de datasets que se mantienen actualizados en la fecha en la que se realiza el análisis son pertenecientes a tres ciudades: Madrid, Barcelona y San Sebastián.

- Estos datasets son actualizados con una periodicidad que va desde diariamente hasta mensualmente.

- Estos datasets tienen como propósito exponer eventos sociales o deportivos, incidencias urbanísticas, datos sobre el tráfico, que incluyen automóviles, autobuses, bicicletas o taxis y datos sobre la contaminación o el ruido de sus respectivas ciudades.

- Tras el análisis, 52 datasets fueron elegidos como posibles candidatos para ser seleccionados. Los 52 datasets eran actualizados hasta la fecha en la que se realiza el análisis. Son los que se exponen a continuación:

    - **Agenda cultural de la ciudad de Barcelona:** Fecha de creación 19/11/2015. Frecuencia de actualización: Diario.

    - **Sistema de información al usuario (SIU) de predicción del estacionamiento en el área azul de la ciudad de Barcelona:** Fecha de creación 29/11/2018. Frecuencia de actualización: Diario.

    - **Perfil del contratante del Ayuntamiento de Barcelona:** Fecha de creación 04/10/2013. Frecuencia de actualización: Diario.

- **Agenda de actividades deportivas de la ciudad de Barcelona:** Fecha de creación 01/10/2014. Frecuencia de actualización: Diario.

- **Obras en el espacio público de la ciudad de Barcelona:** Fecha de creación 30/09/2014. Frecuencia de actualización: Diario.

- **Oferta pública del Ayuntamiento de Barcelona:** Fecha de creación 08/06/2016. Frecuencia de actualización: Diario.

- **Agenda de actos de Anella Olímpica de la ciudad de Barcelona:** Fecha de creación 26/07/2019. Frecuencia de actualización: Diario.

- **Provisión y promoción de personal del Ayuntamiento de Barcelona:** Fecha de creación 08/06/2016. Frecuencia de actualización: Diario.

- **Ocupación de Parkings subterráneos de San Sebastián:** Fecha de creación 11/06/2019. Frecuencia de actualización: Diario.

- **Oferta de empleo de San Sebastián:** Fecha de creación 04/01/2018. Frecuencia de actualización: Diario.

- **Agenda turística de la ciudad de Madrid:** Fecha de creación 11/04/2016. Frecuencia de actualización: Diario.

- **Calidad del aire. Episodios de alta contaminación atmosférica por dióxido de nitrógeno de la ciudad de Madrid:** Fecha de creación 22/11/2019. Frecuencia de actualización: Diario.

- **Taxi. Modelos de vehículos autorizados de la ciudad de Madrid:** Fecha de creación 25/06/2014. Frecuencia de actualización: Diario.

- **Tráfico. Incidencias en vía pública de la ciudad de Madrid:** Fecha de creación 12/03/2014. Frecuencia de actualización: Diario.

- **Tráfico. Información en paneles informativos de información variable en superficie de la ciudad de Madrid:** Fecha de creación 12/03/2014. Frecuencia de actualización: Diario.

- **Oposiciones y empleo de la ciudad de Madrid:** Fecha de creación 12/03/2014. Frecuencia de actualización: Diario.

- **EMT. Incidencias del servicio de la ciudad de Madrid:** Fecha de creación 12/03/2014. Frecuencia de actualización: Diario.

- **Eventos municipales para el fomento de la igualdad de oportunidades los próximos 100 días de la ciudad de Madrid:** Fecha de creación 15/12/2014. Frecuencia de actualización: Diario.

- **Tráfico. Mapa de tramas de intensidad del tráfico de la ciudad de Madrid:** Fecha de creación 08/08/2014. Frecuencia de actualización: Diario.

- **Callejero Oficial del Ayuntamiento de Madrid (Servicio Web):** Fecha de creación 15/10/2018. Frecuencia de actualización: Diario.

- **Tráfico Calle 30. Datos de tráfico y estado del tráfico tiempo real (M-30):** Fecha de creación 09/04/2015. Frecuencia de actualización: Diario.

- **Agenda de actividades y eventos de la ciudad de Madrid:** Fecha de creación 29/03/2017. Frecuencia de actualización: Diario.

- **Tráfico Calle 30 (M-30). Trabajos planificados en tiempo real:** Fecha de creación 09/04/2015. Frecuencia de actualización: Diario.

- **Escuelas de español para extranjeros de la ciudad de Madrid:** Fecha de creación 11/04/2016. Frecuencia de actualización: Diario.

- **Agenda de actividades deportivas de la ciudad de Madrid:** Fecha de creación 09/04/2015. Frecuencia de actualización: Diario.

- **Tráfico Calle 30. Histórico de datos de usuarios que han circulado desde 2013:** Fecha de creación 16/04/2015. Frecuencia de actualización: Diario.

- **Locales de diversión y entretenimiento con perfil turístico de la ciudad de Madrid:** Fecha de creación 11/04/2016. Frecuencia de actualización: Diario.

- **Contaminación acústica. Datos diarios de la ciudad de Madrid:** Fecha de creación 02/07/2015. Frecuencia de actualización: Diario.

- **Participación ciudadana. Debates y propuestas de la ciudad de Madrid:** Fecha de creación 04/05/2017. Frecuencia de actualización: Diario.

- **Boletín Oficial del Ayuntamiento de Madrid:** Fecha de creación 12/03/2014. Frecuencia de actualización: Diario.

- **Puntos de Interés turístico de la ciudad de Madrid:** Fecha de creación 11/04/2016. Frecuencia de actualización: Diario.

- **EMT. Calendario, grupos, líneas y paradas de la ciudad de Madrid:** Fecha de creación 23/09/2014. Frecuencia de actualización: Diario.

- **Taxi. Datos diarios de flota de vehículos de la ciudad de Madrid:** Fecha de creación 29/05/2018. Frecuencia de actualización: Diario.

- **Datos meteorológicos de la ciudad de Madrid:** Fecha de creación 15/10/2019. Frecuencia de actualización: Diario.

- **Callejero Oficial del Ayuntamiento de Madrid:** Fecha de creación 30/03/2015. Frecuencia de actualización: Diario.

- **Actividades gratuitas en Bibliotecas Municipales en los próximos 60 días de la ciudad de Madrid:** Fecha de creación 27/05/2014. Frecuencia de actualización: Diario.

- **Contenedores o sacos de residuos de construcción y demolición (RCD) llenos más de 24 horas de la ciudad de Madrid:** Fecha de creación 14/03/2017. Frecuencia de actualización: Diario.

- **Estaciones regeneradoras de aguas residuales de la ciudad de Madrid:** Fecha de creación 31/05/2018. Frecuencia de actualización: Diario.

- **Próximas carreras urbanas de la ciudad de Madrid:** Fecha de creación 27/09/2018. Frecuencia de actualización: Diario.
- **Medidas de las estaciones meteorológicas de la ciudad de Barcelona:** Fecha de creación 06/11/2019. Frecuencia de actualización: Mensual.

- **Listado de equipamientos de asociaciones de la ciudad de Barcelona:** Fecha de creación 25/03/2011. Frecuencia de actualización: Mensual.

- **Catálogo de fotografías de planes y proyectos urbanísticos en el Repositorio Abierto de Conocimiento del Ayuntamiento de Barcelona (BCNROC):** Fecha de creación 22/01/2018. Frecuencia de actualización: Mensual.

- **Catálogo de bases de ayudas y subvenciones en el Repositorio Abierto de Conocimiento del Ayuntamiento de Barcelona (BCNROC):** Fecha de creación 23/01/2018. Frecuencia de actualización: Mensual.

- **Datos demográficos de las secciones censales de la ciudad de Barcelona:** Fecha de creación 02/03/2011. Frecuencia de actualización: Mensual.

- **Catálogo de memorias en el Repositorio Abierto de Conocimiento del Ayuntamiento de Barcelona (BCNROC):** Fecha de creación 22/01/2018. Frecuencia de actualización: Mensual.

- **Catálogo de dosieres y notas de prensa en el Repositorio Abierto de Conocimiento del Ayuntamiento de Barcelona (BCNROC):** Fecha de creación 23/01/2018. Frecuencia de actualización: Mensual.

- **Paro registrado de la ciudad de Barcelona:** Fecha de creación 01/10/2014. Frecuencia de actualización: Mensual.

- **Información de los colores de los aparcamientos en superficie de la ciudad de Barcelona:** Fecha de creación 07/02/2018. Frecuencia de actualización: Mensual.

- **Catálogo de medidas de gobierno en el Repositorio Abierto de Conocimiento del Ayuntamiento de Barcelona (BCNROC):** Fecha de creación 22/01/2018. Frecuencia de actualización: Mensual.

### 3.1.3 Portal de datos abiertos de la Unión Europea

El último portal analizado es el portal de datos abiertos de la Unión Europea [18].

Es el portal de datos abiertos donde la Unión Europea pone a disposición los datos abiertos de los países miembros. Al igual que en los anteriores portales, se pueden encontrar una gran variedad de conjuntos de datos clasificados por temáticas. De los tres portales, es el que mayor cantidad de bases de datos abiertas almacena.

A continuación, se muestran los resultados obtenidos tras realizar un análisis sobre los conjuntos de datos ofrecidos por este portal, así como los posibles conjuntos que se pueden utilizar para este proyecto:

- El principal problema encontrado en este portal es que los datasets están ofrecidos en sus respectivos lenguajes, lo cual puede dificultar la extracción de información de ellos.

- La mayoría de datasets eran actualizados hasta la fecha en la que se realiza el análisis.

- Al igual que pasaba con el portal de datos abiertos del Gobierno de España, la periodicidad abarca desde diariamente hasta mensualmente.

- Tras el análisis, se escogieron cinco datasets como posibles candidatos. Estos datasets fueron escogidos ya que se ofrecían en español o en inglés, estaban

siendo actualizados hasta el momento en el que se realizó el análisis y su actualización era diaria. Estos datasets se exponen a continuación:

- **Precio de la electricidad por tipo de usuario**. Frecuencia de actualización: Diaria.
- **Emisiones contaminantes del transporte.** Frecuencia de actualización: Diaria.
- **AirBase - La base de datos europea de calidad del aire.** Frecuencia de actualización: Diaria.
- **Productividad energética.** Frecuencia de actualización: Diaria.
- **Productividad del agua.** Frecuencia de actualización: Diaria.

### 3.1.4 Elección de base de datos abierta

Tras analizar los tres portales, se obtienen 58 posibles datasets que cumplen con los requisitos necesarios. Entre todos los datasets, se cubre una gran variedad de temáticas que abarcan eventos sociales, deportivos, culturales, información sobre el medio ambiente, tráfico, ruido, incidentes de la vía pública, contaminación, recursos públicos, etcétera. Por lo tanto, tras observar las distintas temáticas disponibles, se opta por elegir datasets que traten sobre el medio ambiente o incidencias.

Siguiendo el anterior razonamiento, dos datasets son escogidos como posibles candidatos. A continuación, se presentan estos dos datasets, así como información destacable sobre ellos:

- **Calidad del aire. Episodios de alta contaminación atmosférica por dióxido de nitrógeno de la ciudad de Madrid:**
  - Fecha de creación: 22/11/2019.
  - Frecuencia de actualización: Diario.
  - Descripción: El conjunto de datos incluye una línea por cada una de las medidas adoptadas en aplicación del protocolo vigente en el momento en el que se desarrolla el trabajo para episodios de alta contaminación del aire por dióxido de nitrógeno. Cada línea detalla el período de aplicación, el escenario del protocolo que se aplica, la medida adoptada, así como su alcance (geográfico y, en caso de restricciones a la circulación y estacionamiento, los vehículos a los que afecta).
  - Formato del fichero: CSV separados por punto y coma.
  - API REST: Disponible.

- **Tráfico. Incidencias en vía pública de la ciudad de Madrid:**
  - Fecha de creación: 12/03/2014.
  - Frecuencia de actualización: Diario.

- Descripción: Este conjunto de datos, presenta las incidencias en vía pública en formato XML. Las incidencias en vía pública pueden ser del siguiente tipo: obras, accidentes, alertas, previsiones y eventos. Esta información se actualiza casi en tiempo real, con una periodicidad de unos 5 minutos. Las incidencias asociadas a eventos se graban con 24 horas de antelación aproximadamente.
- Formato del fichero: XML.
- API REST: Disponible.

Debido a que el dataset de incidentes en la vía pública ofrecía una mayor cantidad de datos y, por lo tanto, se podía representar más información en una página web estática a través de gráficos y mapas, se decide escoger este. En el siguiente punto, se explican los datos por los que está compuesto el dataset.

## 3.2 Análisis de la base de datos abierta elegida

Tal y como se concluye en el anterior punto, se decide escoger el dataset de incidencias en la vía pública de la ciudad de Madrid. El Ayuntamiento de Madrid ofrece este dataset en formato XML. A continuación, se explican los datos por lo que está compuesto el dataset:

- **Id_incidencia:** Número de incidencia dentro de GATAM[2].

- **Codigo:** Año y número de incidencia dentro del año. Formato aaaa/nnnnn.

- **Cod_tipo_incidencia:** Código de la incidencia según la normativa DATEX 2[3].

- **Nom_tipo_incidencia**: Nombre de la incidencia DATEX 2.

- **Fh_inicio:** Fecha de inicio de la incidencia. Formato yyyy-mm-ddTHH:mm: ss+dd:00.

- **Fh_final:** Fecha de finalización de la incidencia prevista o programada. Formato yyyy-mmddTHH:mm: ss+dd:00.

- **Incid_prevista:** Indica si la incidencia estaba prevista. Valores: 'S' para incidencia prevista, 'N' para no prevista.

- **Incid_planificada:** Indica si la incidencia ha sido planificada. Valores: 'S' para incidencia planificada, 'N' para no planificada.

[2] GATAM es el sistema de Gestión Automática de Tecnología Abierta de Movilidad usado por el Ayuntamiento de Madrid, por lo tanto, el número ofrecido por este sistema se utiliza como identificador único para cada incidencia almacenada.

[3] DATEX 2 es un estándar de intercambio de datos para transmitir información de tráfico entre centros de gestión y operadores de tráfico. Contiene, por ejemplo, incidentes, obras y otro tipo de eventos especiales relacionados con el tráfico. En este caso, se utiliza para codificar el tipo de incidencia según la normativa.

- **Incid_estado:** Estado actual de la incidencia. Valores: 1 activa, 4 en espera.

- **Descripcion:** Texto descriptivo de la incidencia.

- **Utm_x:** Coordenada UTM X de la posición de la incidencia. Sistema de referencia EPSG:23030, ED50 / UTM Zone 30N. Si la incidencia es de contaminación este campo contendrá el valor NaN.

- **Utm_y:** Coordenada UTM Y de la posición de la incidencia. Sistema de referencia EPSG:23030, ED50 / UTM Zone 30N. Si la incidencia es de contaminación este campo contendrá el valor NaN.

- **Longitud:** Posición de la incidencia en coordenadas geográficas angulares: longitud. Si la incidencia es de contaminación este campo contendrá el valor NaN.

- **Latitud:** Posición de la incidencia en coordenadas geográficas angulares: latitud. Si la incidencia es de contaminación este campo contendrá el valor NaN.

- **Tipoincid:** Tipo de Incidencia a efectos de icono a mostrar: 1 obras 2 accidente 3 alerta 4 evento 5 previsión 10 contaminación

- **Es_obras:** Indicación de si la incidencia corresponde a una obra: Sí = 'S' o No = 'N'.

- **Es_accidente:** Indicación de si la incidencia corresponde a un accidente: Sí = 'S' o No = 'N'.

- **Es_contaminacion:** Indicación de si la incidencia corresponde con un escenario de contaminación: Sí = 'S' o No = 'N'.

- **Escenario_contaminacion:** Identificación del escenario activo en un episodio de contaminación. Si no hay escenario activo el campo estará en blanco.

- **Fecha_escenario_contaminacion:** Fecha de activación del escenario de contaminación.

- **Descripcion_escenario:** Texto abreviado de los motivos de activación del escenario. Si no hay escenario activo el campo estará en blanco.

- **Medidas_escenario:** Texto con las medidas a tomar en el escenario activo. Si no hay escenario activo el campo estará en blanco.

- **Excepciones_escenario:** Texto con las excepciones a las restricciones del escenario activo. Si no hay escenario activo el campo estará en blanco.

Analizando todos los datos que ofrece el dataset, se decide realizar cuatro gráficos y un mapa. Los gráficos representan: la cantidad de incidencias previstas, la cantidad de incidencias planificadas, la cantidad de cada tipo de incidencias y una evolución diaria del total de incidencias. En el mapa se ubica cada incidencia de la ciudad de Madrid, y se incluye una descripción a cada ubicación que explica la incidencia.

# 4. Desarrollo de la solución

En esta sección se explica paso a paso como se ha desarrollado la aplicación propuesta. El primer punto, expone la arquitectura utilizada para desarrollar la aplicación, la cual está dividida por dos elementos principales: el script Python y la página web estática. Siguiendo esta división, los dos siguientes puntos tratan cada uno de estos elementos de forma individual, un punto dedicado a explicar el script desarrollado en Python y el último punto dedicado a explicar el desarrollo de la página web estática, utilizando el framework Nuxt.js.

## 4.1 Arquitectura de la solución propuesta

En este punto se explica la arquitectura de la aplicación propuesta para representar las incidencias en la vía pública de la ciudad de Madrid, así como el funcionamiento de los elementos que componen la aplicación. Asimismo, se expone la interacción que existe entre cada uno de estos componentes.

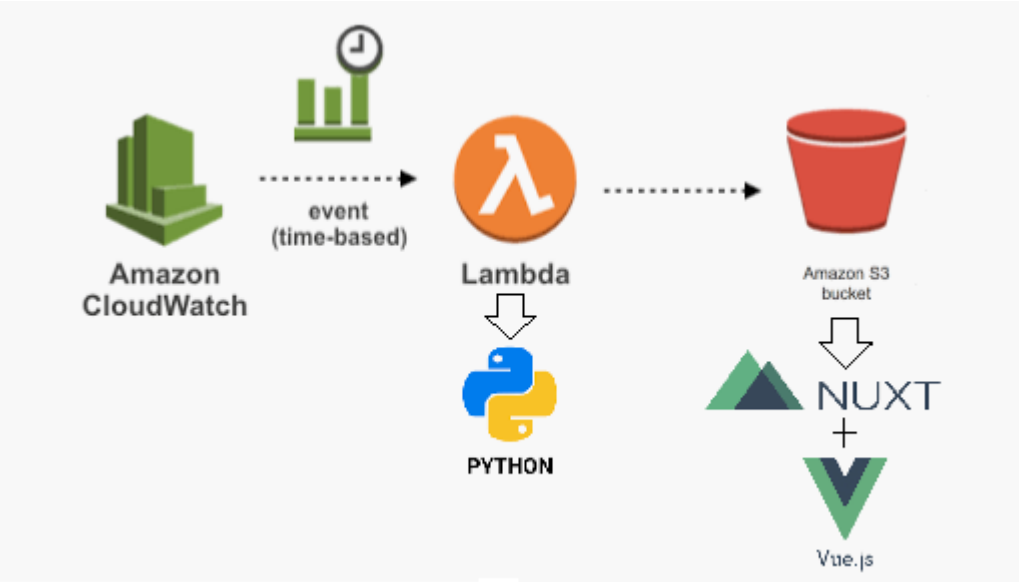


*Ilustración 2. Arquitectura de la solución propuesta. Fuente: Elaboración propia.*

En la ilustración 2 se muestra la arquitectura creada para que la aplicación propuesta complete los objetivos previstos. Como se puede observar en la ilustración, el primer componente de la arquitectura es el servicio Amazon CloudWatch, en el cual se crea una regla para que se genere un evento periódicamente cada día. Este evento es enviado al servicio AWS Lambda, donde se almacena el script Python como una función Lambda, desarrollado para descargar y procesar el dataset público. Cada vez que AWS Lambda recibe el evento, se ejecuta la función Lambda de forma automática y sin necesidad de configurar una infraestructura subyacente. Por último, la función Lambda genera dos archivos con extensión JavaScript que se almacenan en un bucket de Amazon S3. Estos dos archivos son los que contienen los datos a representar visualmente por la página web, la cual también está almacenada en un bucket de Amazon S3. La página web ha sido desarrollada utilizando el framework Nuxt.js, el cual

hace uso a su vez del framework Vue.js. Al estar la página web almacenada en Amazon S3, se logra ofrecer la página web de forma pública y con una alta disponibilidad.

Como se puede observar en la ilustración 2, se puede dividir la aplicación en dos elementos principales, como son el script Python y la página web estática. A continuación, se expone brevemente el funcionamiento de cada uno de estos elementos, así como la interacción existente entre ellos.

El script Python como se ha mencionado anteriormente, tiene el objetivo de descargar los datos del dataset público, el cual se ofrece a través de un archivo XML. Tras descargar el archivo y realizar un proceso de transformación y limpieza, se obtienen los datos a representar, siendo almacenados en archivos *.js*. Estos archivos *.js* son guardados en el bucket *tfm-pspda* de Amazon S3. El script Python es ejecutado utilizando el servicio de AWS Lambda, el cual provee la infraestructura necesaria de forma automática. Tan solo se debe de configurar el tiempo máximo que el script puede durar y la memoria que necesita.

Los atributos que se van a representar del dataset público y, por lo tanto, los que van a ser extraídos con el script Python, son los siguientes: *nom_tipo_incidencia, incid_planificada, incid_prevista, descripción, longitud y,* por último*, latitud*.

La página web estática se desarrolla utilizando el framework Nuxt.js. En ella se representan cuatro gráficos utilizando la librería gráfica Apexcharts.js: dos gráficos de barras que indican el número de incidencias previstas y planificadas, un gráfico de tarta que indica la cantidad de tipos de incidencia y por último un gráfico de línea que abarca la evolución de las incidencias diariamente. También se va a crear un mapa, que se genera utilizando el plugin vue2-google-maps, donde se van a ubicar las incidencias junto con una descripción. Los datos que se van a representar son leídos a partir de los archivos *.js* que genera el script Python. En el *index.html* de la página web, el archivo donde se encuentra el código que genera la página, se indica el enlace de Amazon S3 donde se almacenan los archivos *.js*. Concretamente, existen dos archivos *.js*, uno que almacena los datos que se representan en los gráficos y otro que representa la ubicación y las descripciones de las incidencias para visualizarlas en el mapa de la ciudad de Madrid. El *index.html* y los demás archivos necesarios para que la página web sea generada son almacenados en el bucket *tfm-pspda*, donde se configura a Amazon S3 para que almacene una página web estática.

Con estos dos componentes se genera una aplicación sin estado, que se ejecuta automáticamente en la nube cada día sin necesidad de aprovisionar infraestructura. La actualización de los datos también se hace de forma automática, es decir, sin necesidad de intervención humana.

## 4.2 Desarrollo del script Python

El script Python ha sido desarrollada para cumplir tres objetivos:

- Descargar los datos en formato XML de la base de datos pública sobre las incidencias en la vía pública de la ciudad de Madrid.

- Procesar y transformar los datos para que puedan ser leídos por la página web estática, adaptándolos al formato de la página web.

- Almacenar los datos en archivos con extensión *.js*, dentro de un bucket de Amazon S3, para que la página web estática pueda leer los datos accediendo tan solo a un enlace, proporcionado por Amazon S3, consiguiendo de esta manera, transparencia a la hora de actualizar los archivos *.js*. Así se consigue que la página web estática represente los datos, evitando la necesidad de regenerar la página web cada vez que los datos sean actualizados.

Teniendo en cuenta estos tres objetivos, se propone la creación del script Python a través de la plataforma de código abierto Anaconda, la cual permite programar funciones en Python en el sistema operativo Windows. Tras seleccionar la plataforma de desarrollo, el siguiente paso es la creación de un script que sea capaz de cumplir con los tres objetivos previamente mencionados.

A continuación, se exponen las distintas partes de código por la cual se compone el script:

```
1  # encoding: utf-8
2
3  import json
4  import boto3
5  import os
6  import sys
7  import urllib2
8  import xml.etree.ElementTree as ET
9  import re
10 from datetime import datetime
11 import unicodedata
12 import uuid
13
14 s3_client = boto3.client('s3')
15
```

*Ilustración 3. Importación de librerías. Fuente: Elaboración propia.*

La primera parte del script incluye la importación de las librerías que se observa en la ilustración 3. Entre el conjunto de librerías, se van a mencionar las más importantes:

- **Boto3:** Esta librería se utiliza para interactuar con los servicios de Amazon. En este caso, se va a utilizar para interactuar con el servicio Amazon S3. De esta forma, se pueden almacenar y recuperar objetos de buckets de S3, así como modificar las características de los objetos almacenados.

- **Urllib2:** Esta librería se utiliza para acceder a archivos a través de URLs. Para este proyecto, se utiliza para acceder al archivo XML que contiene los datos de incidencias en la vía pública de la ciudad de Madrid.

- **Xml.etree.ElementTree:** Esta librería se utiliza para acceder a datos que están estructurados en formato XML. En este caso se utiliza para obtener un listado de incidencias, así como para extraer los datos que se quieren representar en la página web.

- **Datetime:** Esta librería se utiliza para obtener la fecha en la que la función se ejecuta, para que, de esta manera, se pueda crear un gráfico que represente la evolución histórica del número de incidencias de cada día.

- **Unicodedata:** Esta librería se utiliza para transformar el formato de una cadena de caracteres a Unicode. Se utiliza debido a que las descripciones de las incidencias están almacenadas en un formato que no puede ser leído por la página web, por lo que se tienen que transformar a un formato Unicode.

La línea 14 de la ilustración 3, muestra el comando para configurar el script para poder acceder al servicio Amazon S3.

```
def strip_accents(s):
   return ''.join(c for c in unicodedata.normalize('NFD', s)
                  if unicodedata.category(c) != 'Mn')
```

*Ilustración 4. Función para transformación a Unicode. Fuente: Elaboración propia.*

La ilustración 4 muestra una función que se va a utilizar en el script para transformar las descripciones de las incidencias a un formato que sea legible por la página web estática. Este formato es Unicode, por lo que la función transforma los caracteres especiales que utiliza la codificación del dataset público del Ayuntamiento de Madrid a caracteres Unicode.

```
21  def lambda_handler(event, context):
22
23      response = urllib2.urlopen('http://informo.munimadrid.es/informo/tmadrid/incid_aytomadrid.xml')
24      html = response.read()
25
26  #Procesado del archivo XML
27
28      tree = ET.ElementTree(ET.fromstring(html))
29      root = tree.getroot()
30      incidencias = root.findall('Incidencia')
```

*Ilustración 5. Acceso al dataset público. Fuente: Elaboración propia.*

La ilustración 5 muestra el inicio de la función encargada de lograr los objetivos expuestos al principio de esta sección, llamada *lambda_handler*. En la ilustración, en la línea 23 se muestra el acceso al dataset público. En las líneas 28, 29 y 30, se configura el script para que pueda acceder al archivo XML obtenido y se pueda extraer el listado de incidencias almacenadas en el archivo, así como los atributos. De esta forma, en la línea 30, se crea la variable *incidencias*, la cual tiene almacenada todas las incidencias del archivo XML, permitiendo acceder a cada incidencia de forma individual y al atributo que se requiera.

```
#Obtencion de nombre de tipos de incidencia y sus cantidades

    nom_incidencia = []
    num_incidencia = []

    for incidencia in incidencias:
        if incidencia.find('nom_tipo_incidencia').text in nom_incidencia:
            pos = nom_incidencia.index(incidencia.find('nom_tipo_incidencia').text)
            num_incidencia[pos] = num_incidencia[pos] + 1
        else:
            nom_incidencia.append(incidencia.find('nom_tipo_incidencia').text)
            num_incidencia.append(1)
```

*Ilustración 6. Obtención de tipo de incidencia y cantidades. Fuente: Elaboración propia.*

En la ilustración 6 se pueden observar las líneas que se utilizan para acceder a un atributo en concreto de las incidencias, en este caso, es al tipo al que pertenece la incidencia, denominado *nom_tipo_incidencia* en el archivo XML. Estas líneas se utilizan para crear dos variables, una que contenga los tipos de incidencias, y otra que contenga la cantidad de incidencias de cada tipo. De esta manera, se puede crear un gráfico que represente estos dos valores.

```
#Obtencion de incidencias planificadas y sus cantidades

    incidencia_pla = ['S','N']
    incidencia_pla_num = [0,0]

    for incidencia in incidencias:
        pos = incidencia_pla.index(incidencia.find('incid_planificada').text)
        incidencia_pla_num[pos] = incidencia_pla_num[pos] + 1

#Obtencion de incidencias previstas y sus cantidades

    incidencia_pre = ['S','N']
    incidencia_pre_num = [0,0]

    for incidencia in incidencias:
        pos = incidencia_pre.index(incidencia.find('incid_prevista').text)
        incidencia_pre_num[pos] = incidencia_pre_num[pos] + 1

```

*Ilustración 7. Obtención de incidencias planificadas, previstas y cantidades. Fuente: Elaboración propia.*

En la ilustración 7, se muestran las líneas de código encargadas de acceder al número de incidencias planificadas y de incidencias previstas. Se crean cuatro variables, *incidencia_pla* e *incidencia_pre* almacenan los dos valores que pueden tomar los atributos *incid_planificada* e *incid_prevista*, *'S'* significando que sí han sido planificadas o previstas y *'N'* significando lo contrario. *Incidencia_pla_num* almacena la cantidad de incidencias que han sido planificadas y cuantas no, y lo mismo ocurre con *incidencia_pre_num*, pero con las incidencias que han sido previstas y las que no. En este caso, las líneas de código son utilizadas para cuantificar el número de incidencias que han sido planificadas y cuantas no y el número de incidencias que han sido previstas y cuántas no han sido. Estas cantidades se representan en dos gráfico independientes disponibles en la página web estática.

```
73  #Obtencion de la descripción de la incidencia
74
75      incidencia_des = []
76
77      for incidencia in incidencias:
78          incidencia_des.append(incidencia.find("descripcion").text)
79
80  #Formateo de la descripcion de la incidencia. De unicode a string
81
82      incidencia_des_uni = []
83
84      for i in incidencia_des:
85          if(isinstance(i, unicode)):
86              incidencia_des_uni.append(re.sub('\W+',' ', strip_accents(i).replace("\n",' ')))
87          else:
88              incidencia_des_uni.append(str(i.decode("utf-8").replace("\n"," ")))
89
```

*Ilustración 8. Obtención de descripción de incidencias y transformación a Unicode. Fuente: Elaboración propia.*

La ilustración 8 expone las líneas de código que se utilizan para acceder a las descripciones de las incidencias. Estas descripciones son almacenadas en la variable *incidencia_des*. Posteriormente, a partir de la línea 84 se realiza un proceso para transformar el formato de las descripciones a Unicode, para que de esta forma puedan ser leídas por la página web estática y no se generen inconsistencias. Las descripciones en formato Unicode se almacenan en la variable *incidencia_des_uni*. Las descripciones se utilizan para añadirlas como información complementaria a las ubicaciones de las incidencias dentro del mapa que se quiere crear.

```
90  #Obtener longitud y latitud de la incidencia
91
92      incidencia_lon=[]
93      incidencia_lat=[]
94
95      for incidencia in incidencias:
96          incidencia_lon.append(incidencia.find("longitud").text)
97          incidencia_lat.append(incidencia.find("latitud").text)
98
```

*Ilustración 9. Obtención de latitud y longitud. Fuente: Elaboración propia.*

A los últimos atributos que se van a acceder son a la *longitud* y a la *latitud*. Estos son necesarios para representar la incidencia en el mapa que se quiere crear. Para ello, en la ilustración 9 se muestran las líneas de código necesarias para acceder a estos atributos y almacenarlos en dos variables, *incidencia_lon* para las longitudes e *incidencia_lat* para las latitudes.

```
 99  #Recuperar las fechas almacenadas y añadir la actual
100
101     download_path = '/tmp/{}{}'.format(uuid.uuid4(), "fechas.js")
102
103     s3_client.download_file("tfm-pspda", "fechas.js", download_path)
104
105     f = open(download_path, "r")
106     contents = f.readlines()
107     f.close()
108
109     contents.append(",'"+ str(datetime.today().strftime('%Y-%m-%d')) +"'")
110
111     f = open(download_path, "w")
112     contents = "".join(contents)
113     f.write(contents)
114     f.close()
115
116     s3_client.upload_file(download_path, "tfm-pspda", "fechas.js")
```

*Ilustración 10. Acceso y actualización del histórico de fechas. Fuente: Elaboración propia.*

En la ilustración 10, se observan las líneas de código encargadas de acceder a un archivo llamado *fechas.js* almacenado en el bucket *tfm-pspda* del servicio Amazon S3. Para ello se utiliza la librería boto3, previamente explicada, para descargar este archivo *.js* y poder manipularlo. En este archivo se almacenan las fechas en la que la función se ha ejecutado. De esta manera, se crea una evolución de cómo han aumentado o disminuido las incidencias a lo largo del tiempo. Las líneas representadas en la ilustración 9, leen el archivo descargado *fechas.js*, para posteriormente añadirle la fecha en la que la función está siendo ejecutada, y, por último, almacenan el archivo en la ubicación donde se encontraba al principio, creando de esta manera, un histórico de fechas. Las fechas almacenadas junto a la nueva fecha, son traspasadas a la variable *contents* para que posteriormente sean transmitidas para poder ser representadas en la página web estática.

```
118 #Recuperar las incidencias almacenadas y añadir la actual
119
120     download_path = '/tmp/{}{}'.format(uuid.uuid4(), "acumulado.js")
121
122     s3_client.download_file("tfm-pspda", "acumulado.js", download_path)
123
124     f = open(download_path, "r")
125     contents2 = f.readlines()
126     f.close()
127
128     total_inc = incidencia_pla_num[0] + incidencia_pla_num[1]
129
130     contents2.append(","+ str(total_inc))
131
132     f = open(download_path, "w")
133     contents2 = "".join(contents2)
134     f.write(contents2)
135     f.close()
136
137     s3_client.upload_file(download_path, "tfm-pspda", "acumulado.js")
```

*Ilustración 11. Acceso y actualización del histórico de incidencias totales. Fuente: Elaboración propia.*

En la ilustración 11, se pueden encontrar líneas de código parecidas a las encontradas en la ilustración 10, ya que se utilizan para almacenar el número de incidencias totales del día en el que se ejecuta la función. Con los datos obtenidos tanto en la ilustración 10 como en la 11, se puede crear una evolución histórica de incidencias totales de la vía pública de la ciudad de Madrid a lo largo de los días. Como en la ilustración 10, se recupera un archivo *.js* llamado *acumulado.js* donde se almacena el histórico de incidencias totales por día, que está almacenado en el bucket *tfm-pspda*. Tras añadirle la nueva incidencia total del día en el que se ejecuta la función, se vuelve a almacenar en la misma ubicación. Por último, se traspasan los datos que contenía *acumulado.js* y el nuevo valor de total de incidencias a la variable *contents2* para que puedan ser transferidos a la página web estática.

```
    download_path = '/tmp/{}{}'.format(uuid.uuid4(), "data.js")

    s3_client.download_file("tfm-pspda", "data.js", download_path)

    with open(download_path, 'r') as file :
        filedata = file.read()

    # Remplazar valores dentro del fichero
    filedata = filedata.replace('var r=[]', 'var r=' + str(incidencia_pla_num))
    filedata = filedata.replace('c=[]', 'c=' + str(incidencia_pre_num))
    filedata = filedata.replace('d=[]', 'd=' + str(nom_incidencia_uni))
    filedata = filedata.replace('h=[]', 'h=' + str(num_incidencia))
    filedata = filedata.replace('f=[]', 'f=[' + str(contents) + ']')
    filedata = filedata.replace('x=[]', 'x=[' + str(contents2) + ']')
# Write the file out again
with open(download_path, 'w') as file:
    file.write(filedata)

s3_key = "alucloud-lambda/114"
foldername = os.path.dirname(s3_key)
filename = os.path.basename(s3_key)
print "alucloud114"
s3_client.upload_file(download_path, "tfm-pspda", '114/{}'.format("4d8bc723606233a9aad1.js"), ExtraArgs
```

*Ilustración 12. Traspaso de datos a la página web estática y guardado de valores. Fuente: Elaboración propia.*

En la ilustración 12, se observan las líneas de código encargadas de transmitir todos los datos obtenidos del dataset a la página web estática, para que esta sea capaz de leerlos y representarlos visualmente. Para ello, se ha dividido en dos partes la transmisión de los datos. En la ilustración 12 se puede observar la primera parte, encargada de transmitir los datos de: tipos de incidencias, cantidad de tipos de incidencias, cantidad de incidencias planificadas, cantidad de incidencias previstas, fechas almacenadas incluyendo la fecha en la que la función está siendo ejecutada y, por último, la cantidad total de incidencias almacenadas por día, incluyendo la incidencia total de la fecha en la que la función está siendo ejecutada. Para ello, se recupera un archivo llamado *data.js*, almacenado en el bucket *tfm-pspda*.

Como en las dos ilustraciones anteriores, se hace uso de la librería boto3 para interactuar con el servicio de Amazon S3. Una vez recuperado el archivo *data.js*, se sustituyen las variables *r, c, d, h, f* y *x* del archivo por las variables con los datos obtenidos a lo largo del script, *incidencia_pla_num, incidencia_pre_num, nom_incidencia_uni, num_incidencia, contents y contents2*, respectivamente. El archivo *data.js* contiene la estructura que utiliza la página web para leer datos externos. Sin embargo, este archivo contiene las variables de los datos a representar vacías para que, de esta forma, se puedan sustituir con los datos que el script obtiene cada vez que se ejecuta. Se utiliza para no provocar inconsistencias a la hora de actualizar los datos, evitando de esta manera, que algún dato antiguo siga siendo representado cuando no debe.

En el archivo *data.js* la variable *r* representa el número de incidencias planificadas, la variable *c* el número de incidencias previstas, la variable *d* el nombre de cada tipo de incidencia, la variable *h* la cantidad de cada tipo de incidencia, la variable *f* las fechas almacenadas y la variable *x* el total de incidencias almacenadas. Estas variables las crea automáticamente el framework Nuxt.js cuando se genera la página web estática, por lo que las denominaciones de estas no han podido ser escogidas.

Por último, la línea 165 almacena el nuevo archivo que contiene todas las variables con contenido al bucket *tfm-pspda* dentro de la carpeta *114*, con visibilidad pública para que pueda ser accedido por la página web. El archivo que contiene los datos extraídos, se le denomina *4d8bc723606233a9aad1.js* ya que es el archivo que el framework Nuxt.js genera automáticamente a la hora de crear la página web estática, y lo configura como el archivo de donde tiene que obtener los datos a representar gráficamente. Como pasa con las variables, el nombre se genera automáticamente por lo que no se ha podido escoger.

```
177     download_path = '/tmp/{}{}'.format(uuid.uuid4(), "incidencia_lat_lon.js")
178
179     s3_client.download_file("tfm-pspda", "incidencia_lat_lon.js", download_path)
180
181     # Remplazar valores dentro del fichero
182
183     with open(download_path,"w") as file:
184         file.write("[{ position: { lat: "+str(incidencia_lat[0])+", lng: "+str(incidencia_lon[0])+"}}")
185         for index, i in enumerate(incidencia_lat[1:]):
186             file.write(",{ position: { lat: "+str(i)+", lng: "+str(incidencia_lon[index+1])+"}}")
187         file.write("]")
188
189     s3_client.upload_file(download_path, "tfm-pspda", "incidencia_lat_lon.js")
```

*Ilustración 13. Fusionar latitud con longitud. Fuente: Elaboración propia.*

La ilustración 13, muestra el código necesario para fusionar los datos de latitud y longitud. En el archivo obtenido del dataset público del Ayuntamiento de Madrid, estos dos valores se ofrecen por separado. Sin embargo, el plugin que ubica las incidencias en el mapa de la ciudad de Madrid, vue2-google-maps, necesita los valores de latitud y longitud como una cadena de caracteres juntos, separados por una coma, por lo que es necesario realizar un procesado sobre estos datos. Por ello, como se observa en la ilustración 12, se utiliza un archivo auxiliar llamado *incidencia_lat_lon.js*, el cual es recuperado del bucket *tfm-pspda*, donde se almacenan las latitudes y las longitudes de cada incidencia en el formato requerido por el plugin. Finalmente, este archivo es almacenado de nuevo en el bucket *tfm-pspda*.

```
183     with open(download_path,"w") as file:
184         file.write("[{ position: { lat: "+str(incidencia_lat[0])+", lng: "+str(incidencia_lon[0])+"}}")
185         for index, i in enumerate(incidencia_lat[1:]):
186             file.write(",{ position: { lat: "+str(i)+", lng: "+str(incidencia_lon[index+1])+"}}")
187         file.write("]")
188
189     s3_client.upload_file(download_path, "tfm-pspda", "incidencia_lat_lon.js")
190
191     download_path = '/tmp/{}{}'.format(uuid.uuid4(), "data_map.js")
192
193     s3_client.download_file("tfm-pspda", "data_map.js", download_path)
194
195     download_path2 = '/tmp/{}{}'.format(uuid.uuid4(), "incidencia_lat_lon.js")
196
197     s3_client.download_file("tfm-pspda", "incidencia_lat_lon.js", download_path2)
198
199     with open(download_path, 'r') as file :
200         filedata = file.read()
201
202     with open(download_path2, 'r') as file2 :
203         filedata2 = file2.read()
208     # Write the file out again
209     with open(download_path, 'w') as file:
210         file.write(filedata)
211
212     s3_key = "alucloud-lambda/114"
213     foldername = os.path.dirname(s3_key)
214     filename = os.path.basename(s3_key)
215     s3_client.upload_file(download_path, "tfm-pspda", '114/{}'.format("87eea0093ec967d98493.js"), ExtraArgs
216
217     return "fin"
```

*Ilustración 14. Traspaso de datos al mapa de la página web estática y guardado de valores. Fuente: Elaboración propia.*

La última parte del script Python se puede analizar en la ilustración 14. En esta ilustración se pueden observar las líneas de código encargadas de traspasar los últimos datos, completando de esta manera la segunda parte de proceso. En la ilustración 14, se traspasan los datos relativos a las ubicaciones de las incidencias y las descripciones de estas, para que puedan ser visualizadas en el mapa contenido en la página web estática.

Para ello, se utiliza como en el caso de la ilustración 12, un archivo que contiene la estructura para que la página web estática lea datos desde el exterior, *data_map.js*, pero con las variables vacías. Esta vez, las variables utilizadas son *l* y *t*. La variable *l* representa la longitud y la latitud de la incidencia, la variable *t* la descripción de cada una de estas. La longitud y la latitud son obtenidas mediante la recuperación del archivo que previamente se había rellenado, en la ilustración 13, llamado *incidencia_lat_lon.js*.

Tanto *data_map.js* como *incidencia_lat_lon.js* están almacenados en el bucket *tfm-pspda*. Una vez leído ambos archivos, se traspasan los datos contenidos a sus respectivas variables, *l* y *t*, y son almacenados en el archivo *87eea0093ec967d98493.js*. Como en el caso de los anteriores datos traspasados, este archivo es el que genera el framework Nuxt.js cuando crea la página web. Este archivo es almacenado también en el bucket *tfm-pspda* dentro de la carpeta *114* con acceso público para que pueda ser accedido por la página web estática.

Con esto, finaliza el script Python, concluyendo con el almacenamiento de dos archivos *.js* para que la página web estática pueda leer los datos del dataset y representarlos, pudiendo leer los datos en el formato requerido por los plugins para crear gráficos y para crear mapas. En el siguiente punto, se expone la creación la página web estática, así como su funcionamiento.

## 4.3 Desarrollo de la página web estática.

En este punto, se explica el desarrollo que se ha seguido para crear la página web estática que compone la parte visual del proyecto. Esta página web, representa los datos que se extraen del dataset público de incidencias de la vía pública de la ciudad de Madrid. Para desarrollar la página web estática, se hace uso del framework Nuxt.js. Además de este framework, como se ha explicado en la sección 2, se utilizan dos librerías para representar gráficamente los datos y para ubicar las incidencias, Apexcharts.js y vue2-google-maps, respectivamente.

Para utilizar el framework Nuxt.js, se ha hecho uso de la plataforma de Microsoft Visual Studio Code, donde se ha instalado el framework. Visual Studio Code, permite programar aplicaciones, pudiendo instalar plugins para utilizar un lenguaje de programación en concreto. En este caso, aparte de instalar el framework Nuxt.js, es necesario instalar el framework Vue.js a través del plugin Vetur, disponible en el catálogo de plugins de Visual Studio Code.

```
$ npx create-nuxt-app <project-name>
```

*Ilustración 15. Creación de un Proyecto Nuxt. Fuente: Elaboración propia.*

En la ilustración 15, se muestra el comando encargado de crear un proyecto Nuxt. Con este comando se instalan las dependencias necesarias para utilizar el framework Nuxt.js. Donde lee <*project-name*>, se debe sustituir con el nombre del proyecto que se quiere crear.

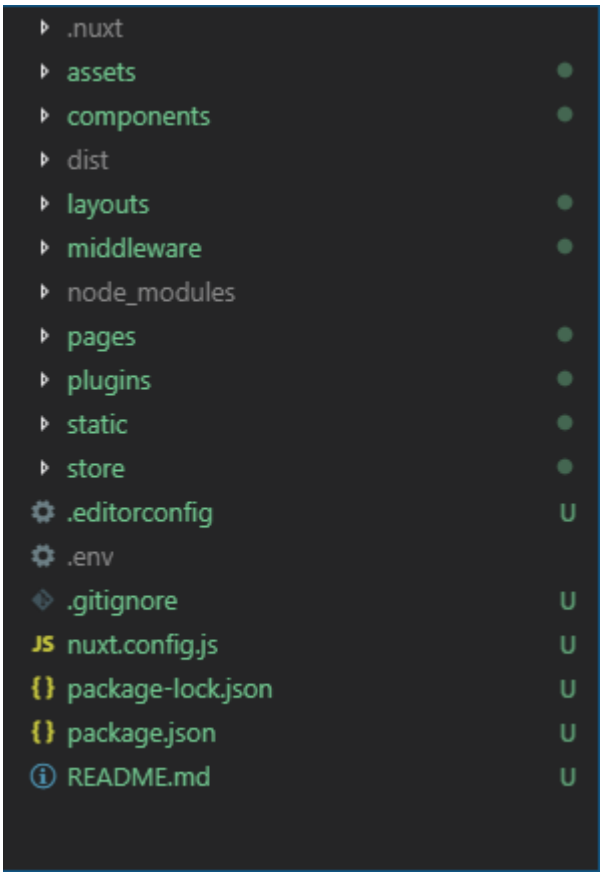


*Ilustración 16. Estructura de un Proyecto Nuxt.*
*Fuente: Elaboración propia.*

Tras crear un proyecto Nuxt, se genera la estructura que se puede observar en la ilustración 16. En ella, se observan diversas carpetas y archivos. Para explicar este proyecto, se van a detallar los elementos más importantes:

- **Pages:** Esta carpeta es donde se ubican y se crean las páginas que contendrá el proyecto que se desea crear. Para este proyecto, se van a crear dos páginas, *index.vue* donde se localizan los gráficos con los distintos datos y *mapa.vue* donde se localiza el mapa con las incidencias.

- **Components:** En esta carpeta se ubican los componentes Vue que forman una página. Se ubican en esta carpeta para que puedan ser reutilizados por todo el proyecto. En este proyecto, es donde se ubica el logo de la página web, los gráficos y el mapa.

- **Plugins:** En esta carpeta se ubican los plugins que se van a utilizar en el proyecto. En este caso, es donde se ubican los plugins de Apexcharts.js y vue2-google-maps.

- **Static:** En esta carpeta se ubican los elementos estáticos del proyecto. Normalmente, esta carpeta se utiliza para ubicar archivos que contienen datos para que sean leídos por los componentes del proyecto. En este caso, es donde se ubican los archivos con los datos extraídos del dataset de incidencias de la vía pública, indicando de esta manera, que los datos son externos al proyecto.

- **Nuxt.config.js:** Es el archivo donde se localizan las distintas opciones de configuración del proyecto. En este archivo se deben declarar los plugins que se utilizan, los módulos, si va a ser renderizado por parte del servidor o por parte del cliente u opciones especiales a la hora de construir el proyecto. Para este proyecto, en este archivo es donde se declaran los dos plugins que se utilizan, dejando las demás opciones por defecto.

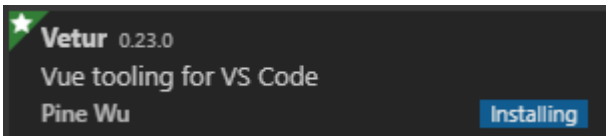


*Ilustración 17. Plugin Vetur. Fuente: Elaboración propia.*

Una vez se tiene creada ya la estructura principal del proyecto Nuxt, se procede a instalar el plugin de Visual Studio Code para utilizar el framework Vue.js. Este plugin es recomendado por Visual Studio Code cuando reconoce un proyecto Nuxt. En la ilustración 17 se puede observar la pantalla de instalación del plugin.

El siguiente paso es instalar los dos plugins previamente mencionados, Apexcharts.js y vue2-google-maps. Para ello, es necesario insertar los comandos que se muestran a continuación:

```
npm install --save apexcharts
npm install --save vue-apexcharts
```

*Ilustración 18. Instalación del plugin apexcharts.js. Fuente: Elaboración propia.*

```
npm i -D @nuxtjs/dotenv
npm i vue2-google-maps
```

*Ilustración 19. Instalación del plugin vue2-google-maps. Fuente: Elaboración propia.*

En la ilustración 18 y 19 se muestran los comandos necesarios para instalar las dependencias para utilizar los dos plugins. Una vez instaladas las dependencias, se deben importar los dos plugins al proyecto dentro de la carpeta *plugins* y posteriormente, declararlos en el archivo *nuxt.config.js*.

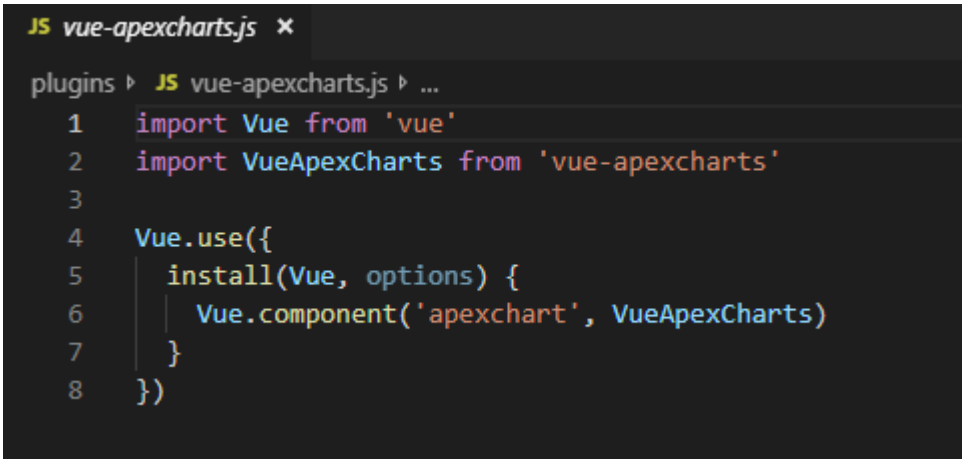


*Ilustración 20. Importación del plugin apexcharts.js. Fuente: Elaboración propia.*

En la ilustración 20 se observa la forma en la que se importa el plugin Apexcharts.js dentro del proyecto Nuxt. Para ello se crea un archivo *.js* llamado *vue-apexcharts.js* dentro de la carpeta *plugins*, y se configura para que pueda ser utilizado por cualquier elemento del proyecto.

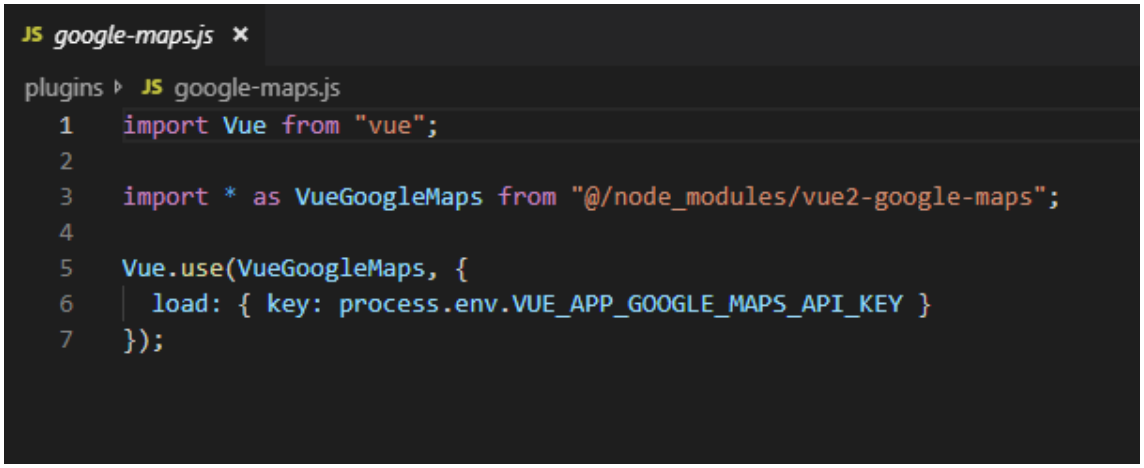


*Ilustración 21. Importación del plugin vue2-google-maps. Fuente: Elaboración propia.*

Parecida a la ilustración anterior, la número 21 muestra la importación del plugin vue2-google-maps dentro del proyecto Nuxt. En este caso, aparte de configurar el plugin para que pueda ser utilizado por cualquier elemento del proyecto, se debe declarar una *API key* de Google Cloud Platform, con los permisos para utilizar el servicio Google Maps. Para ello, se crea un archivo *.env*, donde se declara la *API key*, que se almacena en la variable *key*, mostrada en la línea 6 de la ilustración 21. Esto se hace por motivos de seguridad, para no exponer la *API key* al público. Al igual que en el caso de la ilustración 20, todo el código está contenido en un archivo *.js* llamado *google-maps.js* ubicado dentro de la carpeta *plugins*.

```
plugins: ["@/plugins/google-maps",
{
  src: '~/plugins/vue-apexcharts',
  ssr: false
}
],
```

*Ilustración 22. Configuración de nuxt.config.js. Fuente: Elaboración propia.*

El último paso, para acabar de configurar los plugins dentro del proyecto, es declararlos dentro del archivo *config.nuxt.js*, dentro de la sección plugins, señalando la ubicación de los archivos *.js* previamente creados, tal como muestra la ilustración 22.

Tras realizar todos estos pasos, ya se tiene preparado al proyecto Nuxt para poder utilizar los plugins y generar páginas web con el framework Nuxt. Por lo tanto, lo siguiente que se va a realizar es crear las páginas que van a componer el proyecto: *index.vue* y *mapa.vue*. En la primera se van a ubicar los gráficos y en la segunda el mapa. La forma en la que se componen estas páginas es explicada a continuación.

Las páginas web construidas sobre el framework Nuxt, están divididas en tres secciones: *template*, *script* y *style*. En *template*, se declaran todos los componentes que va a tener la página web. En este caso, es donde se declaran los componentes HTML y JavaScript. En *script*, se declara el funcionamiento y los datos que utilizan los componentes JavaScript. Por último, en *style*, se configuran las opciones de personalización y la ubicación de los componentes declarados en *template*. Es en esta sección donde se pueden configurar los colores, tipos de letras o el color de fondo, entre otras opciones.

```
<template>
<div id="page">
  <div class='button' id="mapa">
      <a href='./mapa'>
        <button>Generar mapa</button>
      </a>
  </div>
    <div id="first">
        <apexchart type="bar" height="350" width="500" :options="chartOptionsBar" :series="series1"></a
    </div>
    <div id="second">
        <apexchart type="bar" height="350" width="500" :options="chartOptionsLine" :series="series2"></
    </div>
    <div id="third">
        <apexchart type="donut" height="400" width="700" :options="chartOptionsDonut" :series="series3"
    </div>
    <div id="forth">
        <apexchart type="area" height="385" width="500" :options="chartOptionsBarEv" :series="series4">
    </div>
</div>
</template>
```

*Ilustración 23. Template de index.vue. Fuente: Elaboración propia.*

En la ilustración 23, se puede observar cómo se han declarado los cuatro gráficos que se quieren crear para representar los datos relativos a las incidencias. Estos gráficos están contenidos dentro de la página *index.vue*, la cual se crea dentro de la carpeta *pages* del proyecto Nuxt. En la ilustración, se pueden observar tanto elementos HTML como JavaScript. Cada elemento es englobado dentro de la etiqueta <*div*> para poder ser personalizado en la sección *style* de la página. Los gráficos se declaran como elementos dentro de la etiqueta <*apexchart*>. Dentro de esta etiqueta, se configuran distintos atributos como son: *type* para el tipo de gráfico, *height* para la altura, *width* para la anchura, *options* para personalizar las características del gráfico y por último *series*, para indicar los datos que se quieren representar. Siguiendo esta explicación, como se puede observar en la ilustración 23, se ha declarado:

- Un primer gráfico de tipo barras, con altura de 350 píxeles y anchura de 500 píxeles. Este gráfico representa la cantidad de incidencias planificadas. En *options*, se configura para que tenga de título "Cantidad de incidencias planificadas" y para que salga el número total de cada categoría en la parte superior de cada barra. En *series*, se inserta la variable *series1* que contiene los datos relativos a este gráfico.

- Un segundo gráfico de tipo barras, con altura de 350 píxeles y anchura de 500 píxeles. Este gráfico representa la cantidad de incidencias previstas. En *options*, se configura para que tenga el título de "Cantidad de incidencias previstas" y para que salga el número total de cada categoría en la parte superior de cada barra. En *series*, se inserta la variable *series2* que contiene los datos relativos a este gráfico.

- Un tercer gráfico de tipo donut, con altura de 400 píxeles y anchura de 700 píxeles. Este gráfico representa la cantidad de tipos de incidencias. En *options*, se configura para que tenga el título de "Tipos de incidencias y cantidad", así como que cada tipo de incidencia esté representado con un color y que se muestre en porcentaje la cantidad de cada tipo. En *series*, se inserta la variable *series3* que contiene los datos relativos a este gráfico.

- Por último, un cuarto gráfico de tipo línea, con una altura de 385 píxeles y una anchura de 500 píxeles. Este gráfico representa la evolución histórica de incidencias totales por día. En *options*, se configura para que tenga el título de "Evolución de incidencias respecto al tiempo", así como que cada cantidad se muestre encima de cada punto de la línea del gráfico, y que pueda expandirse o contraerse para analizar un periodo de tiempo en concreto. En *series*, se inserta la variable *series4* que contiene los datos relativos a este gráfico.

El resultado de la declaración de cada uno de estos gráficos se puede observar a continuación:

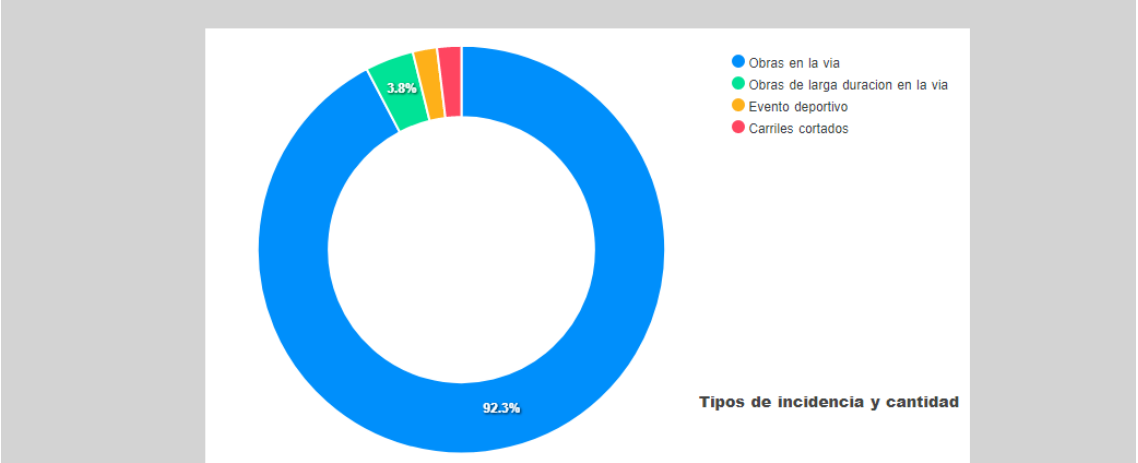


*Ilustración 24. Gráfico tipos de incidencias y cantidades. Fuente: Elaboración propia.*

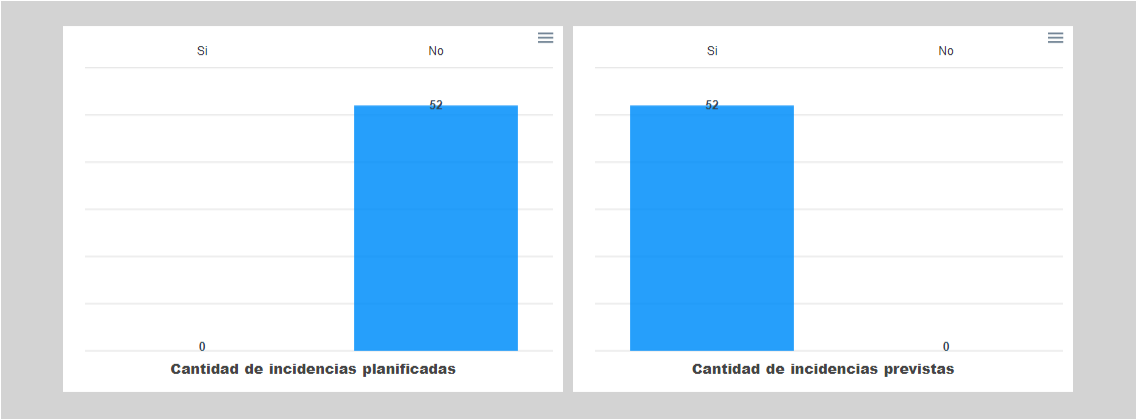


*Ilustración 25. Gráficos de incidencias planificadas y previstas. Fuente: Elaboración propia.*


*Ilustración 26. Gráfico de evolución de incidencias respecto al tiempo. Fuente: Elaboración propia.*

En la ilustración 24, 25 y 26 se muestran los tres gráficos creados con la librería Apexcharts.js tras la configuración realizada en la sección *template* en la página *index.vue*. Estos gráficos, son interactivos, se actualizan automáticamente y extraen los datos de archivos externos al proyecto. Estas características, son configuradas en la sección *script* de la página.

```
<script>
import inc_pla_num from "~/static/data_inc_pla_num.js"
import inc_pla from "~/static/data_inc_pla.js"
import inc_pre_num from "~/static/data_inc_pre_num.js"
import inc_pre from "~/static/data_inc_pre.js"
import inc_nom from "~/static/data_nom_inc.js"
import inc_num from "~/static/data_num_inc.js"
import fechas from "~/static/data.js"
import inc_ac from "~/static/data_inc.js"
```

*Ilustración 27. Importación de datos externos. Fuente: Elaboración propia.*

La ilustración 27 contiene la primera parte de la sección *script* de la página *index.vue*. En esta primera parte, se declara que se van a importar datos desde archivos externos al proyecto ubicados en la carpeta *static*. Estos archivos contienen los datos extraídos del dataset público de incidencias de la ciudad de Madrid.

```
export default {
  name: 'pageWithChart',
  data: function() {
        return{
        series1: [{
          name: 'Incidencias planificadas',
          data: inc_pla_num
        }],
        series2: [{
          name: 'Incidencias previstas',
          data: inc_pre_num
        }],
        series3: inc_num,
        series4: [{
          name: 'Evolución de incidencias',
          data: inc_ac
        }],
```

*Ilustración 28. Traspaso de datos extraídos. Fuente: Elaboración propia.*

En la ilustración 28, se encuentran ubicadas las líneas de código utilizadas para traspasar los datos extraídos de los archivos externos a variables internas del proyecto. De esta manera, se declaran las variables *series1*, *series2*, *series3* y *series4*, que como se ha mostrado anteriormente, son utilizadas para indicar los datos que van a representar los gráficos. Cada variable *series* contiene un atributo *name* para identificarla y un atributo *data* que almacena los datos. En este caso, en *inc_pla_num* se almacena la cantidad de incidencias planificadas, en *inc_pre_num* se almacena la cantidad de incidencias previstas, en *inc_num* se almacena la cantidad de cada tipo diferente de incidencia y en *inc_ac* se almacena la cantidad total de incidencias a lo largo del tiempo.

```
chartOptionsBar: {
  chart: {
    height: 350,
    type: 'bar',
  },
  plotOptions: {
    bar: {
      dataLabels: {
        position: 'top', // top, center, bottom
      },
    }
  },
```

*Ilustración 29. Ejemplo de options de un gráfico. Fuente: Elaboración propia.*

Tras almacenar los datos externos en variables internas del proyecto, se procede a mostrar un ejemplo de configuración del atributo *options* de uno de los cuatro gráficos, en este caso del primer gráfico. Este ejemplo se puede encontrar en la ilustración 29. En esta ilustración, se ubican las líneas de código encargadas de configurar el tipo de gráfico dentro del tipo *bar*, ya que existen distintas variaciones para cada tipo de gráfico, pudiéndose escoger entre opciones 3D o 2D. Además, se configura la posición del título, optándose por la opción *top*, es decir, que el título se muestre arriba del gráfico.

Tras configurar el atributo *options* de cada gráfico, finaliza la sección *script* de la página *index.vue*. La última sección que queda por completar es *style*.

```
<style scoped>

body{
  background-color: lightgray;
}


#first {
float: left;
left: 170px;
top: 50px;
background-color: white;
}

#second {
float: left;
left: 180px;
top: 50px;
background-color: white;
}

#third{
  float: left;
  left: 30px;
  top: 100px;
  background-color: white;
}
```

*Ilustración 30. Sección style de index.vue.*
*Fuente: Elaboración propia.*

En la ilustración 30, se provee un breve resumen de lo que se puede realizar en esta sección, que está enfocada a ofrecer detalles estéticos. En la ilustración, se pueden ver las características definidas para varios componentes como son: *body*, *first*, *second* y *third*. En ellos se define la ubicación del componente en la página, la distancia entre cada componentes y el color de fondo de cada uno.

Con esto último, la página *index.vue* queda completada y lista para poder ser generada como página web estática. Sin embargo, para este proyecto, se desea ofrecer otra página donde se localice un mapa con las ubicaciones de las incidencias, así como sus descripciones. Para ello, se crea una segunda página llamada *mapa.vue* dentro de la carpeta *pages* del proyecto Nuxt. A continuación, se detallan los pasos a seguir para la creación y configuración del mapa.

```
<template>
  <div>
    <h1>Mapa de incidencias de Madrid</h1>

    <gmap-map :center="center" :map-type-id="mapTypeId" :zoom="12">
      <gmap-marker
        v-for="(item, index) in markers"
        :key="index"
        :position="item.position"
        :clickable="true"
        :label="item.label"
        @click="openWindow(item,index)"
      />
      <gmap-info-window
            @closeclick="window_open=false"
            :opened="window_open"
            :position="infowindow"
            :content="infoContent"
            :options="infoOptions"
      >
      </gmap-info-window>

    </gmap-map>

  </div>
</template>
```

*Ilustración 31. Template de mapa.vue. Fuente: Elaboración propia.*

Como en el caso de la página que contiene los gráficos, en *mapa.vue* lo primero que se programa es la sección *template,* mostrado en la ilustración 31. En este caso, también se pueden ver elementos HTML y JavaScript. El componente del mapa se declara dentro de la etiqueta <*gmap-map*>. Dentro de esta etiqueta, se crea un bucle *v-for* que selecciona cada elemento de la variable *markers*. Esta variable contiene la ubicación de cada incidencia. Por ello, de cada elemento que compone *markers,* llamado *item,* se extrae su atributo *position*. En *position* se almacena la latitud y longitud de la incidencia. Por último, se configura el atributo *@click* para que cada vez que se dé un click sobre una ubicación de una incidencia, se inicie a la función *openWindow*, para que aparezca la descripción de la incidencia. Esta función es programada en la sección *script*.

```
<script>

import posiciones from "~/static/data_incidencia_lat_lon.js"
import descripciones from "~/static/data_inc_des.js"
```

*Ilustración 32. Importación de datos externos en mapa.vue. Fuente: Elaboración propia.*

Como también fue en el caso de la página *index.vue*, en *mapa.vue* los datos que se utilizan son externos al proyecto. Por eso lo primero que se programa en la sección *script* de *mapa.vue*, como muestra la ilustración 32, es que los datos utilizados son importados desde archivos externos al proyecto, ubicados como los anteriores en la carpeta *static*.

```
export default {
  data() {
    return {
      center: { lat: 40.4228295867338, lng: -3.69103959214929},
      mapTypeId: "terrain",
      markers: posiciones,

      info_marker: null,
      infowindow:null,
      window_open: false,
      infoContent: null,
      infoOptions:{
        pixelOffset: {
          width: 0,
          height: -35
        },
        content: null
      }

    };
  },
  methods: {
        openWindow: function (marker,idx) {
          this.infowindow = marker.position
          this.infoOptions.content = descripciones[idx]
          this.window_open = true
          }

  }
};
</script>
```

*Ilustración 33. Sección script de mapa.vue. Fuente: Elaboración propia.*

Tras programar la importación de los datos externos, se procede a configurar el componente del mapa, como se puede observar en la ilustración 33. Lo primero que se configura es el mapa que se va a mostrar, indicando la latitud y longitud de la ciudad de Madrid. Después, se configura que las ubicaciones de las incidencias se van a almacenar en la variable interna *markers*, previamente expuesta. La última parte, llamada *methods*, es donde se configura el comportamiento de la función *openWindow*. En esta función se pasan como parámetros la incidencia y su posición dentro de la variable *markers*. Cuando esta función se activa, se muestra la descripción de la incidencia que se ha pasado como parámetro. Esta función solo se activa si se pulsa sobre la ubicación de una incidencia en el mapa.

Tras finalizar la sección *script* de la página *mapa.vue*, ya se completa el proyecto Nuxt que se desea crear. Por lo tanto, como último paso tras configurar y crear todo lo necesario para que el proyecto cumpla con los objetivos propuestos, se introduce el comando *nuxt generate* en la consola de la plataforma de Visual Studio Code. Este comando genera automáticamente la página web estática. Como resultado, genera una carpeta llamada *dist*, la cual contiene un archivo *index.html*, donde se ubica la página web estática con cada uno de los elementos que se han creado, como son los gráficos, incluyendo enrutamientos a las otras páginas que puedan haber sido creadas, como en este caso, la página que contiene el mapa. Como en este caso se ha indicado dentro del proyecto, que los datos que utiliza son externos, Nuxt prepara al proyecto para esta situación, generando dos archivos *.js* independientes del *index.html*, que contienen la estructura necesaria para que la página web estática pueda acceder a datos externos. Estos archivos son los que se han mencionado en la sección 4.2, *4d8bc723606233a9aad1.js* y *87eea0093ec967d98493.js*. Genera dos archivos, debido a que existen dos páginas dentro del proyecto, la que contiene los gráficos y la que contiene el mapa, por lo que separa los datos en dos archivos, para que cada página pueda leer independientemente sus datos.

Como en este caso, la página web estática se va a ubicar en un bucket de Amazon S3, es decir, que no va a ser renderizada en local, se debe especificar dentro del archivo *index.html* donde se van a poder encontrar los dos archivos *.js*. Para ello, se suben los dos archivos *.js* a un bucket de Amazon S3 y se obtiene la dirección de enlace que ofrece el servicio S3 para cada objeto.

```
    <div data-server-rendered="true" id="__nuxt"><!----><div
6b74a4e0><button data-v-6b74a4e0>Generar mapa</button></a></
Object]" data-v-6b74a4e0></apexchart></div> <div id="second"
6b74a4e0></apexchart></div> <div id="third" data-v-6b74a4e0>
</div> <div id="forth" data-v-6b74a4e0><apexchart type="area"
</div></div><script>window.__NUXT__={layout:"default",data:[
<script src="/_nuxt/4d8bc723606233a9aad1.js" defer></script>
<script src="/_nuxt/596af08d5e5bd3e0d6a3.js" defer></script>
  </body>
```

*Ilustración 34. Configuración del archivo index.html. Fuente: Elaboración propia.*

En la ilustración 34, se puede observar marcado en rojo, la parte que se tiene que configurar. Por defecto, el framework Nuxt.js configura que los archivos *.js* van a ser

leídos en local, por ello, se debe de sustituir la dirección por defecto por la dirección de enlace obtenida de Amazon S3. De esta forma se logran dos objetivos, que la página web estática pueda ser alojada en Amazon S3 y que los datos puedan ser independientes de la página web, por lo que pueden ser modificados dentro de los archivos *.js* de forma trasparente a la página, es decir, que la página web se actualiza sin necesidad de generarse desde cero y sin necesitar intervención humana.

# 5. Despliegue de la solución

Una vez finalizadas las secciones 3 y 4, la última parte correspondiente con el proyecto que se presenta en esta memoria es desplegar la solución de un entorno Cloud, para que, de esta manera, la página web estática pueda ser visible públicamente y para que el script Python se ejecute automáticamente de forma periódica cada día, ofreciendo de esta manera datos actualizados de las incidencias de la vía pública de la ciudad de Madrid. Para ello, como se ha mencionado en los apartados anteriores, se va a hacer uso de la plataforma Cloud de Amazon Web Services. En concreto se van a utilizar tres servicios de Amazon Web Services: Amazon S3 para almacenar y publicar la página web estática, AWS Lambda para ejecutar el script Python y, por último, Amazon CloudWatch para generar un evento diariamente, con el objetivo de actuar como disparador para que el script Python se ejecute. Este despliegue se expone paso a paso en esta sección, dividida en tres partes, cada una dedicada a cada uno de los servicios que se van a utilizar.

El último punto de esta sección muestra el resultado final de la aplicación, exponiendo el dashboard compuesto por los cuatro gráficos y el mapa representando las incidencias.

## 5.1 Configuración de Amazon CloudWatch

Para que el script Python se ejecute, debe existir un evento que se produzca diariamente a una determinada hora y que actúe como disparador. Por lo tanto, para lograr este objetivo se va a utilizar el servicio de Amazon CloudWatch. Este servicio permite crear reglas para generar eventos de dos formas: la primera forma es generar un evento cuando se detecta un cambio en un determinado servicio de Amazon y la segunda forma es generar un evento en un horario específico. Como en este proyecto, el dataset de donde se obtienen los datos es externo a Amazon Web Service, se debe utilizar la segunda forma. Para ello, se decide establecer una regla para generar un evento a una hora concreta de forma periódica.

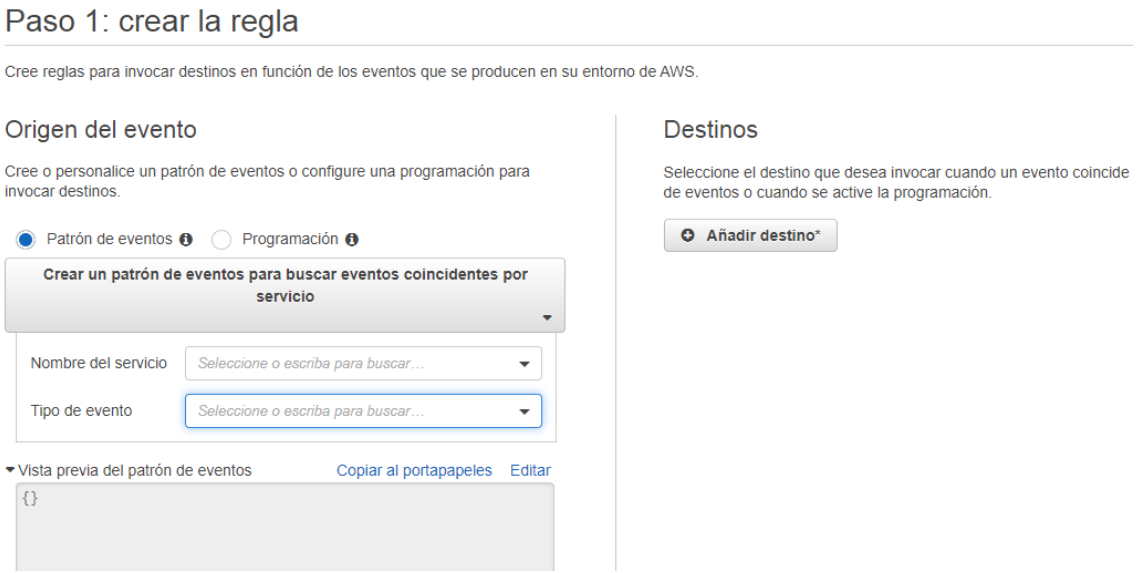


*Ilustración 35. Configuración de Amazon CloudWatch (I). Fuente: Elaboración propia.*

En la ilustración 35, se puede observar la pantalla para configurar reglas para generar eventos del servicio Amazon CloudWatch. Como se ha mencionado antes, se ofrece la opción de generar eventos de dos formas, la primera llamada patrón de eventos y la segunda llamada programación. En esta caso se va a utilizar programación.

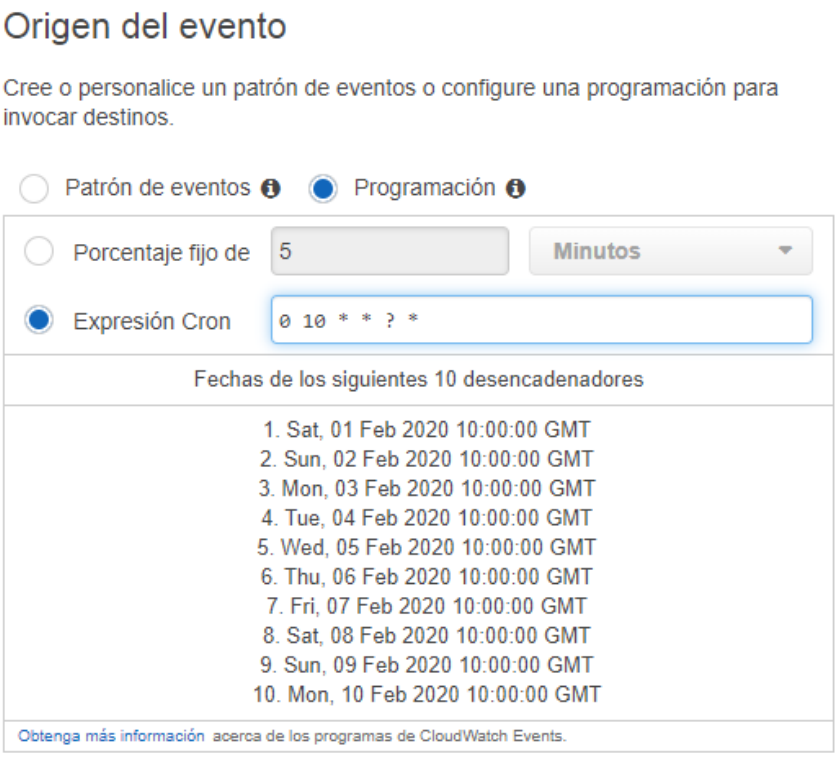


*Ilustración 36. Configuración de Amazon CloudWatch (II). Fuente: Elaboración propia.*

La ilustración 36 expone la configuración que se ha utilizado para generar un evento periódico. Para ello, en la opción programación se utiliza una expresión *Cron* para crear un horario. En este caso, se ha programado una expresión para generar un evento cada día a las 10 a.m. Una vez creada la regla, el siguiente paso es configurar el servicio de AWS Lambda.

## 5.2 Configuración de AWS Lambda

Para mantener en la nube el código Python, lo primero que se ha de realizar es la creación de una función Lambda donde quede almacenado el script. Para ello, a la hora de crear una función Lambda, es necesario establecer cuatro parámetros fundamentales: el lenguaje de programación que se va a usar, la memoria que se prevé utilizar, el tiempo de ejecución máximo que se va a permitir a la función y, por último, el rol que va a estar disponible para la función.


*Ilustración 37. Configuración de AWS Lambda. Fuente: Elaboración propia.*

La ilustración 37 muestra la configuración que se ha seguido para utilizar el servicio AWS Lambda en este proyecto. Se configura declarando que se va a utilizar el lenguaje de programación Python versión 2.7, un total de memoria de 128 MB, la más pequeña entre las opciones, un tiempo de espera máximo de 15 segundos y, por último, se escoge el rol de *s3-execution-role*.

Para escoger la memoria y el tiempo de espera máximo, se realiza un análisis para encontrar el equilibrio entre estos dos parámetros, ya que son los que se asocian al coste de la ejecución de la función. Cuanta más memoria se utilice y cuanto mayor tiempo dure, mayor serán los costes, ya que AWS Lambda ofrece su servicio siguiendo la estrategia de pago por uso, es decir, el cliente paga el tiempo en el que la función está en ejecución. Por ello, tras analizar el rendimiento del script Python creado en la sección 4.2, se observa que necesita una media de 50 MB de memoria y tarda un tiempo medio de ejecución de entre 5 y 7 segundos.

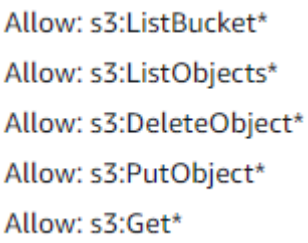


*Ilustración 38. Permisos s3-execution-role.*
*Fuente: Elaboración propia.*

En lo que respecta a rol escogido, se utiliza el rol denominado *s3-execution-role*. Este rol se ajusta a las necesidades de este proyecto, ya que la función Lambda va a necesitar los permisos para interactuar con el servicio Amazon S3, recuperando, subiendo objetos y modificándolos para que sean públicos. Para ello, se necesitan los permisos de *PutObject** y *Get** para S3, tal y como se pueden observar en la ilustración 38.

Una vez se ha configurado la función Lambda, lo único que se tiene que proveer a este servicio es el código ejecutable, ya que el aprovisionamiento de la infraestructura lo realiza el propio servicio. Por lo tanto, se sube el script Python creado en la sección 4.2 a la función que se acaba de configurar. El último paso, es configurar la función Lambda para que se ejecute cuando reciba el evento que genera la regla de la sección 5.1. Para ello, se declara como desencadenante el nombre de la regla creada, comunicando de esta manera, la regla de Amazon CloudWatch con la función de AWS Lambda.

## 5.3 Configuración de Amazon S3

La última parte del despliegue del proyecto es almacenar los archivos de la página web estática en el servicio de Amazon S3. Para ello, se crea el bucket *tfm-pspda*. En *tfm-pspda* se almacena el archivo *index.html*, que es el que genera la página y los archivos auxiliares explicados en la sección 4.2, como son, *data.js*, *data_map.js*, *fechas.js, acumulado.js* e *incidencias_lat_lon.js*. También se almacena un archivo llamado *favicon.ico*, que se trata del logo de la página web. Todos estos archivos, una vez subidos se deben de configurar como acceso público, para que puedan ser accedidos tanto por otros servicios de Amazon Web Service, como es el caso de la función Lambda creada en la sección 5.2, como para que la página web pueda ser visible por cualquier cliente. Esta configuración se puede observar en la ilustración 39. Esto se debe realizar para todos los archivos almacenados en el bucket *tfm-pspda*.


*Ilustración 39. Configuración de Amazon S3. Fuente: Elaboración propia.*


*Ilustración 40. Alojamiento de sitio web estático. Fuente: Elaboración propia.*

Una vez subidos todos los archivos necesarios, se configura el bucket *tfm-pspda* para que almacene una página web estática, y se selecciona el archivo *index.html* como archivo que contiene la página web, lo cual da como resultado un enlace público para acceder a la página, mostrado en la ilustración 40.

Tras esta última configuración, la aplicación ya está desplegada completamente en la nube, estando disponible públicamente, ofreciendo datos actualizados diariamente y con una alta disponibilidad, sin haber necesitado aprovisionar ninguna infraestructura para ofrecer este servicio.

## 5.4 Resultado de la aplicación propuesta

A continuación, se muestra el resultado que se obtiene después de desarrollar y desplegar los elementos que componen la aplicación. La aplicación, como se comenta en la sección 4.3, está compuesta por dos páginas web, la primera ofrece un dashboard con cuatro gráficos y la segunda página contiene el mapa de la ciudad de Madrid con las incidencias ubicadas, además de ofrecer información adicional sobre cada incidencia si se decide hacer click sobre una de ellas.

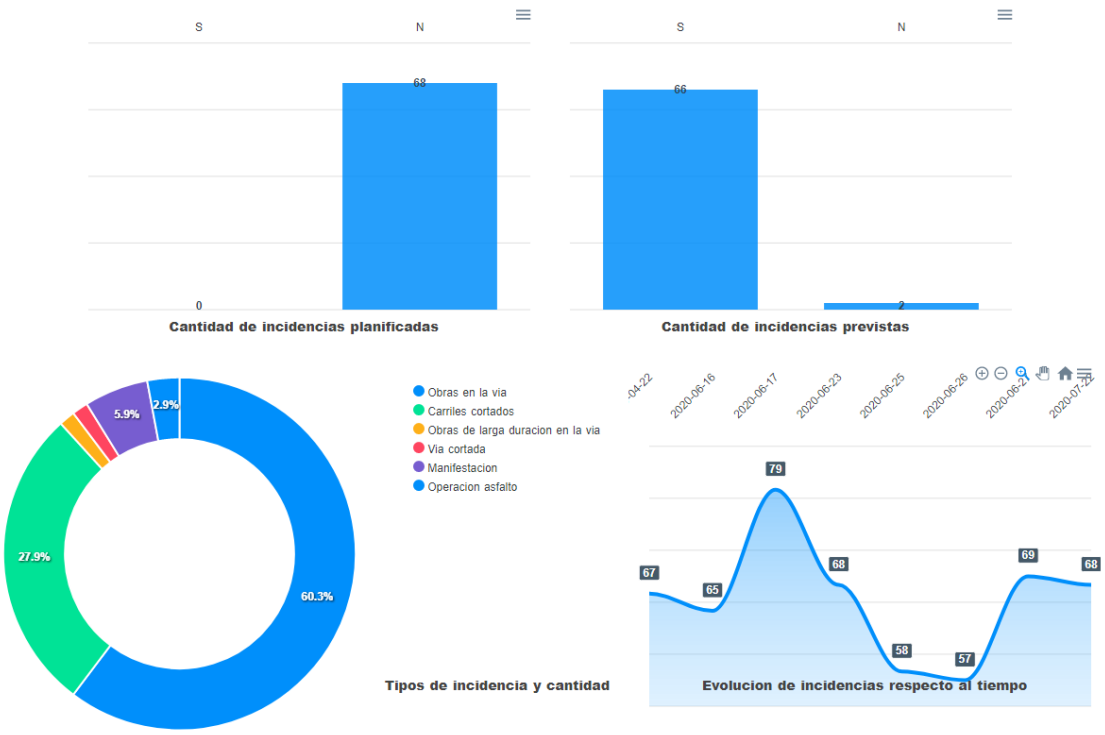


*Ilustración 41. Dashboard compuesto por cuatro gráficos. Fuente: Elaboración propia.*

En la ilustración 41, se puede observar el dashboard ofrecido por la página web estática. Como se muestra en la ilustración, los dos gráficos de la parte superior exponen la cantidad de incidencias que han sido planificadas y previstas, mientras que los dos gráficos de la parte inferior se centran en la cantidad de tipos de incidencias y en la evolución histórica de la cantidad de incidencias a lo largo de los días. Los gráficos ofrecen información actualizada de forma diaria, por lo que los datos disponibles en la página web pertenecen al mismo día en la que se visita, pudiendo de esta manera, entender de una forma rápida y sencilla la situación en cuanto a incidentes en la vía pública de la ciudad de Madrid. Por otra parte, cada gráfico es interactivo pudiendo, por ejemplo, centrarse en un solo tipo de incidencia en el gráfico tipo donut o buscando un periodo en concreto en el gráfico de línea.


*Ilustración 42. Mapa de incidencias. Fuente: Elaboración propia.*

Como se ha mencionado anteriormente, la aplicación esta compuesta por otra página donde se ubica el mapa de la Ciudad de Madrid junto con las incidencias. En la ilustración 42, se puede observar un ejemplo del mapa en un día concreto. Como se puede observar en la ilustración, todas las incidencias ofrecidas por el Ayuntamiento de Madrid son ubicadas. Además, a cada incidencia, se le añade información adicional si se decide hacer click sobre ella, pudiendo de esta manera, entender el motivo de la incidencia. La información que se muestra es la información que ofrece el Ayuntamiento de Madrid. El mapa es interactivo, por lo que a parte de interactuar con las incidencias, también se puede acercar o alejar la camara para ubicar de forma más precisa la dirección de la incidencia. Esto se expone en la ilustración 43. La aplicación se encuentra disponible y funcional en el siguiente enlace: < http://tfm-pspda.s3-website-us-east-1.amazonaws.com >.

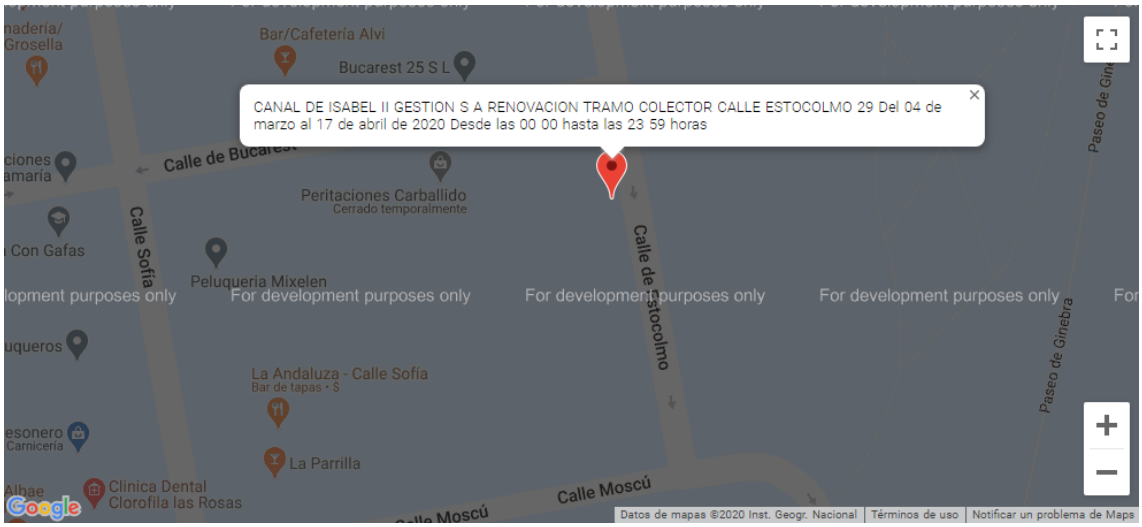


*Ilustración 43. Mapa de incidencias a nivel de calle. Fuente: Elaboración propia.*

# 6. Conclusiones

Tras la finalización del proyecto, habiendo escogido el dataset público de incidencias en la vía pública de la ciudad de Madrid, y tras haber desarrollado y desplegado con éxito cada elemento del proyecto, se obtienen las siguientes conclusiones:

- El dataset escogido cumple con los requisitos esperados:
    - Se actualiza cada 5 minutos, por lo que su periodicidad de actualización es mayor de la necesaria para este proyecto, el cual se actualiza una vez al día.
    - El dataset se ha mantenido actualizado durante el tiempo en el que ha durado el desarrollo del proyecto y se prevé que continúe actualizándose después de la finalización de este.
    - El dataset ofrece sus datos en un formato legible por máquinas como es XML. Los datos son estructurados en nodos, facilitando de esta manera su localización y extracción.
- El script encargado de descargar, transformar y limpiar los datos para que puedan ser representados visualmente, se ha desarrollado en el lenguaje de programación Python, teniendo que añadir diversas librerías para conseguir estas funciones. Se ha desarrollado haciendo uso de la plataforma Anaconda, que permite la programación en Python en el sistema operativo Windows. El script es capaz de descargar los datos del dataset público de incidencias de la vía pública a través de un archivo XML, procesar el archivo para extraer los datos que se quieren representar y, por último, realizar una transformación y limpieza sobre ellos para que puedan ser leídos por la página web estática. Para que sean accedidos por la página web, los datos son almacenados en dos archivos independientes con extensión JavaScript, en el servicio Amazon S3.
- El script creado se ha analizado para observar si cumple con los objetivos secundarios establecidos para este. El resultado del análisis ofrece resultados positivos en este aspecto, debido a que el script tarda de media en ejecutarse entre 5 y 7 segundos, dependiendo de la carga de datos del dataset y utiliza una memoria de media de 50 MB.

- La página web estática se ha desarrollado utilizando el framework Nuxt.js junto a dos plugins, Apexcharts.js y vue2-google-maps. Con ello se ha logrado representar visualmente los datos, utilizando cuatro gráficos y un mapa. Todos estos elementos son interactivos. Como los datos que se representan se leen de dos archivos independientes a la página web, se consigue actualizar los elementos de la página de una forma automática y transparente.

- Como se utiliza el servicio Amazon S3, que garantiza una disponibilidad del 99,99999999%, se supera el mínimo establecido en los objetivos secundarios para la disponibilidad de la página web estática.

- Todo el proyecto se ha logrado implementar en la plataforma Amazon Web Service, haciendo uso de tres de sus servicios, como son Amazon CloudWatch, AWS Lambda y Amazon S3. Por lo tanto, se ha logrado crear una aplicación de bajo coste, sin infraestructura pre-aprovisionada, con una alta disponibilidad y que se actualiza de forma automática y transparente, sin necesidad de involucrar a los clientes o a los desarrolladores de la aplicación.

## 6.1 Relación del trabajo desarrollado con los estudios cursados

Para este proyecto se han utilizado tecnologías vistas durante el máster Universitario en Gestión de la Información de la Universidad Politécnica de Valencia. La plataforma Cloud de Amazon Web Services es impartida en la asignatura Servicios en la Nube (SEN) del máster Universitario en Gestión de la Información de la Universidad Politécnica de Valencia. En esta asignatura, servicios utilizados para este proyecto como Amazon S3, Amazon CloudWatch o AWS Lambda, son introducidos. También se incluyen en esta asignatura otros servicios de Amazon Web Services, como Amazon EC2 o SimpleDB, servicios que, por las características del proyecto, no han sido necesarios utilizar.

La metodología para encontrar y analizar bases de datos públicas, en este caso, bases de datos publicadas por gobiernos, se imparte durante varias asignaturas dentro del máster, debido a que se les da gran importancia a los datos abiertos. Los datos abiertos resultan potencialmente útiles para crear nuevos servicios o aplicaciones que pueden ayudar a informar a la población general. Sin embargo, estos datos en su mayoría quedan sin utilizar, lo que va provocando que cada vez más, las bases de datos se actualicen menos, reduciendo el número de bases de datos útiles. Entre las asignaturas que tratan y exponen la importancia de los datos abiertos se incluyen Sociedad de la Información, Datos en la Web y Servicios de Datos y Contenido.

Por último, en la asignatura Explotación de Datos Masivos, se expone el lenguaje de programación Python y su utilidad para procesos de carga, extracción, transformación y limpieza de datos, así como para realizar análisis estadísticos o predictivos sobre ellos. En este proyecto, se utiliza la plataforma vista en la asignatura, Anaconda, para

programar el script en Python, así como la versión vista de Python, la 2.7, ya que ofrece las funciones necesarias para conseguir los objetivos de este proyecto.

Sin embargo, para desarrollar la página web se ha utilizado una tecnología de la que no se tenía conocimiento previo, como es el framework Nuxt.js y Vue.js. Por ello, se ha seguido una metodología autodidacta, siguiendo cursos online de aprendizaje y creando proyectos de pruebas, hasta que se ha logrado conseguir la creación de la página web estática, y que esta fuese capaz de interactuar con el script Python.

## 6.2 Trabajos futuros

Tras completar todos los objetivos propuestos de una forma satisfactoria, como trabajos futuros se planea extender el proyecto para abarcar incidencias de la vía pública de ciudades que ofrecen estos datos como abiertos, como puede ser la ciudad de Barcelona, Santander o Málaga. Se pretende mostrar información agregada de estas ciudades, dependiendo de los datos que estén disponibles, ya que no son los mismos en todas las ciudades, así como ofrecer un mapa de cada ciudad ubicando las incidencias y mostrando una descripción sobre cada una de ellas.

Por otra parte, se pretende extender la información que ya está siendo ofrecida por la página web, haciendo comparativas entre ciudades u ofreciendo nueva información si alguna ciudad decide ampliar los datos que ofrece.

# 7. Referencias

[1] Amazon Web Services, Inc. (2020b). *Amazon CloudWatch: Monitoreo de infraestructuras y aplicaciones*. Recuperado 23 de julio de 2020, de https://aws.amazon.com/es/cloudwatch/

[2] Amazon Web Services, Inc. (2020c). *AWS | Cloud Computing - Servicios de informática en la nube*. Recuperado 23 de julio de 2020, de https://aws.amazon.com/es/

[3] Amazon Web Services, Inc. (2020d). *AWS | Lambda - Gestión de recursos informáticos*. Recuperado 23 de julio de 2020, de https://aws.amazon.com/es/lambda/

[4] Amazon Web Services, Inc. (2020e). *AWS | Servicio de notificaciones Push (SNS)*. Recuperado 23 de julio de 2020, de https://aws.amazon.com/es/sns/

[5] Amazon Web Services, Inc. (2020f). *Introducción a AWS Lambda. En AWS Lambda: Guía para desarrolladores* (pp. 3-32). Recuperado 23 de julio de 2020, de https://docs.aws.amazon.com/es_es/lambda/latest/dg/getting-started.html

[6] Amazon Web Services, Inc. (2020g). *Programación de expresiones con rate o cron. En AWS Lambda: Guía para desarrolladores* (pp. 196-197). Recuperado 23 de julio de 2020, de https://docs.aws.amazon.com/es_es/lambda/latest/dg/tutorial-scheduled-events-schedule-expressions.html

[7] Amazon Web Services, Inc. (2020h). *Tutorial: Uso de AWS Lambda con eventos programados. En AWS Lambda: Guía para desarrolladores* (pp. 192-195). Recuperado 23 de julio de 2020, de https://docs.aws.amazon.com/es_es/lambda/latest/dg/with-scheduledevents-example.html

[8] Amazon Web Services, Inc. (2020i). *What Is Amazon CloudWatch Events?. En Amazon CloudWatch Events: User Guide* (pp. 1-2). Recuperado 23 de julio de 2020, de https://docs.aws.amazon.com/AmazonCloudWatch/latest/events/WhatIsCloudWatchEvents.html

[9] Anaconda Inc. (2020). *Anaconda | The World's Most Popular Data Science Platform*. Recuperado 15 de julio de 2020, de https://www.anaconda.com/

[10] ApexCharts. (2020a). *ApexCharts.js - Open Source JavaScript Charts for your website*. Recuperado 3 de junio de 2020, de https://apexcharts.com/

[11] ApexCharts. (2020b). *Vue Chart Examples & Samples Demo - ApexCharts.js*. Recuperado 3 de junio de 2020, de https://apexcharts.com/vue-chart-demos/

[12] Ayuntamiento de Madrid. (2016). *En portada - Portal de datos abiertos del Ayuntamiento de Madrid*. Recuperado 23 de julio de 2020, de https://datos.madrid.es/

[13] Ayuntamiento de Valencia (2016). *Adjuntament de València - Portal de Transparencia y Datos Abiertos Valencia*. Recuperado 7 de febrero de 2020, de http://gobiernoabierto.valencia.es/es/

[14] Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., ... & Suter, P. (2017). Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing* (pp. 1-20). Springer, Singapore.

[15] Bokeh contributors. (2020). *Bokeh*. Recuperado 23 de julio de 2020, de https://bokeh.org/

[16] Bostock, M. (2019). *D3.js - Data-Driven Documents*. Recuperado 23 de julio de 2020, de https://d3js.org/

[17] Chopin, A. (s. f.). *Nuxt.js - The Intuitive Vue Framework*. Recuperado 1 de julio de 2020, de https://nuxtjs.org/

[18] European Data Portal (2014). *Portal Europeo de Datos*. Recuperado 7 de febrero de 2020, de https://www.europeandataportal.eu/es

[19] Gobierno de España (2016). *Iniciativa de datos abiertos del Gobierno de España*. Recuperado 7 de febrero de 2020, de https://datos.gob.es/

[20] Google LLC. (2020). *Google Maps*. Recuperado 23 de julio de 2020, de https://www.google.es/maps/?hl=es

[21] Google LLC. (2020). *Servicios de cloud computing | Google Cloud*. Recuperado 23 de julio de 2020, de https://cloud.google.com/

[22] Halliday, P. (2018). *Vue. js 2 Design Patterns and Best Practices: Build enterprise-ready, modular Vue. js applications with Vuex and Nuxt. Packt* Publishing Ltd.

[23] Juszczak, J. (2018-2020). *vue-chartjs*. Recuperado 23 de julio de 2020, de https://vue-chartjs.org/

[24] McGrath, G., & Brenner, P. R. (2017, June). Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)* (pp. 405-410). IEEE.

[25] Mike Bostock (2014-2020). *A graphical approach to visualization Protovis*. Recuperado 23 de julio de 2020, de https://mbostock.github.io/protovis/

[26] Mounier, F. (2015). *Pygal — pygal 2.0.0 documentation*. Recuperado 23 de julio de 2020, de http://www.pygal.org/en/2.0.11/

[27] Munns, C. (2018). *NYC-Parks-Events-Crawler [Aplicación]. Amazon.com, Inc.* Recuperado 23 de julio de 2020, de

https://serverlessrepo.aws.amazon.com/applications/arn:aws:serverlessrepo:us-east-1:242860417674:applications~NYC-Parks-Events-Crawler

[28] npm, Inc. (s. f.). *vue2-google-maps – npm*. Recuperado 23 de julio de 2020, de https://www.npmjs.com/package/vue2-google-maps

[29] Plotly. (2020*). Plotly: The front-end for ML and data science models*. Recuperado 23 de julio de 2020, de https://plotly.com/

[30] Python Software Foundation. (2001-2020a). *Python 2.7 license | Python.org*. Recuperado 16 de julio de 2020, de https://www.python.org/download/releases/2.7/license/

[31] Python Software Foundation. (2001-2020b). *Python Software Foundation | Python Software Foundation*. Recuperado 16 de julio de 2020, de https://www.python.org/psf/

[32] The Matplotlib Development Team. (2012-2020). *Matplotlib: Python plotting — Matplotlib 3.3.0 documentation*. Recuperado 23 de julio de 2020, de https://matplotlib.org/

[33] Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., ... & Lang, M. (2017). Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. *Service Oriented Computing and Applications, 11*(2), 233-247.

[34] Villamizar, M., Garces, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., ... & Lang, M. (2016, May). Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)* (pp. 179-182). IEEE.

[35] Waskom, M. (2012-2020). *seaborn: statistical data visualization — seaborn 0.10.1 documentation*. Recuperado 23 de julio de 2020, de https://seaborn.pydata.org/

[36] You, E. (2014-2020). *Vue.js*. Recuperado 23 de julio de 2020, de https://vuejs.org/

# 8. Glosario

**API:** La interfaz de programación de aplicaciones o API, por sus siglas en inglés, es un conjunto de subrutinas, funciones y procedimientos que ofrece una aplicación para ser utilizada por otro software.

**API key:** Es un identificador único utilizado para autentificar a un usuario, desarrollador o programa para llamar a una API.

**Código:** Es un conjunto de líneas de texto que indica los pasos que debe de seguir un programa informático.

**Cloud:** Se refiere a la computación en la nube. Es un paradigma que permite ofrecer servicios a través de una red, normalmente Internet.

**Cron:** Es el nombre de un programa que permite ejecutar automáticamente comandos o scripts a una hora o fecha específica.

**Dashboard:** Es una interfaz gráfica que ofrece vistas o gráficos sobre indicadores claves de alguna información u objetivo en particular o proceso de negocio.

**Dataset:** Es un conjunto de datos ofrecidos por un proveedor de datos. Su definición más común es la que se refiere a una tabla de una base de datos.

**Dataset público:** Es un conjunto de datos ofrecidos por una proveedor de forma pública, por lo que puede ser obtenido por cualquier usuario.

**Framework:** Es una estructura conceptual y tecnología, normalmente incluye módulos concretos, que sirve como base para la organización y el desarrollo software, facilitando operaciones de desarrollo y programación de software.

**Función:** Es una parte de un programa que realiza una tarea en particular, ya sea independiente al programa principal o bien dentro del contexto del programa principal, pero siempre devolviendo un resultado.

**Librería:** Es un archivo o conjunto de archivos, codificados en un lenguaje de programación, que ofrecen una interfaz para utilizar las funcionalidades que ofrecen.

**Metadato:** Son datos que describen el contenido informativo de un recurso.

**On-premises:** También denominada como “en local”, es un tipo de instalación de una solución software. Esta instalación se lleva a cabo dentro de los servidores y la infraestructura que pertenecen a la propia empresa. Es el modelo tradicional de aplicaciones empresariales.

**Script:** Es una secuencia de ordenes o comandos. Es un término que se utiliza para referirse a un programa informático.

**Serverless:** Es un modelo de ejecución de computación en la nube en el que el proveedor Cloud ejecuta el servidor y administra dinámicamente la asignación de recursos.