Universidad de la República
Facultad de Ingeniería

Parkwise: Sistema Inálambrico para el Mapeo de Vehículos Estacionados utilizando TinyML y Computer Vision en el Borde

Memoria de proyecto presentada a la Facultad de Ingeniería de la Universidad de la República por

Michael Dickinson, Andrés Guido, Santiago Pérez

en cumplimiento parcial de los requerimientos para la obtención del título de Ingeniero Electricista.

Tutores

Álvaro Gómez	Universidad de la República
Leonardo Steinfeld	Universidad de la República

Tribunal

Guillermo Carbajal	Universidad de la República
Mariana Del Castillo	Universidad de la República
Federico La Rocca	Universidad de la República

Montevideo
martes 2 septiembre, 2025

Parkwise: Sistema Inálambrico para el Mapeo de Vehículos Estacionados utilizando TinyML y Computer Vision en el Borde, Michael Dickinson, Andrés Guido, Santiago Pérez.

Esta tesis fue preparada en LaTeX usando la clase iietesis (v1.1).
Contiene un total de 163 páginas.
Compilada el martes 2 septiembre, 2025.
http://iie.fing.edu.uy/

"Character is the real foundation of all worthwhile success."

— John Hays Hammond

"El futuro empieza hoy, no mañana."

— S. Juan Pablo II

Esta página ha sido intencionalmente dejada en blanco.

Agradecimientos

En primer lugar, queremos expresar nuestro más profundo agradecimiento a aquellos amigos, familiares, docentes y compañeros de trabajo que estuvieron presentes para apoyar en todos y cada uno de los momentos difíciles de la tesis. Ya sea desde el apoyo ético-moral y emocional, hasta las recomendaciones técnicas y avanzadas.

—Sin su acompañamiento, nada de esto hubiese sido posible.

Extendemos también un especial reconocimiento a la Facultad de Ingeniería de la Universidad de la República, por brindar tantas veces sus instalaciones como espacio de trabajo y encuentro, y por facilitar financiando el acceso a plataformas, herramientas y recursos tecnológicos que resultaron esenciales para el desarrollo de este proyecto.

Finalmente, para quienes cierran una etapa de carrera de grado, queremos agradecer profundamente a todos los involucrados en semejante trayectoria como lo es cursar y recibirse en la formación en Ingeniería Eléctrica.

Esta página ha sido intencionalmente dejada en blanco.

A nuestros valores.

Esta página ha sido intencionalmente dejada en blanco.

Resumen

Este trabajo presenta el diseño, desarrollo e implementación de Parkwise, un sistema que permite la detección y mapeo de vehículos en entornos urbanos empleando Inteligencia Artificial y Comunicación Inalámbrica Distribuida.

El objetivo del proyecto es construir una plataforma para el monitoreo de la ocupación vehicular en la vía pública. Para ello, se adopta un enfoque descentralizado, donde la detección de objetos se realiza directamente al borde del sistema, utilizando microcontroladores de bajo consumo que ejecutan modelos de detección de objetos. Estas técnicas reciben el nombre de TinyML.

Cada Dispositivo Parkwise está conformado por un microcontrolador Seeed XIAO ESP32-S3, que emplea inferencias locales sobre imágenes capturadas periódicamente con una cámara OV2640 acoplada en el módulo sense, utilizando modelos optimizados con TensorFlow Lite para microcontroldores. Las detecciones se transmiten de forma inalámbrica a un Gateway. El mismo, reenvía los datos vía NB-IoT al Servidor Central, donde se almacenan, procesan y visualizan a través de una interfaz web interactiva desarrollada con Flask, HTML y JavaScript. La configuración del sistema se lleva a cabo mediante un procedimiento que permite calibrar visualmente cada nodo a su entorno físico. A través de una interfaz gráfica, el operario selecciona puntos de referencia sobre una imagen tomada por la cámara y los asocia con coordenadas geográficas en el mapa. Esta vinculación permite calcular una matriz de *homografía* para la cámara de cada dispositivo, que servirá para proyectar las detecciones sobre el plano georreferenciado.

Entre los principales desafíos abordados se encuentran: El procesamiento de imágenes en microcontroladores con recursos computacionales limitados, la comunicación inalámbrica confiable y la calibración geométrica precisa para la proyección de datos espaciales.

El resultado es un sistema adaptable y de bajo costo, que puede escalarse fácilmente a un gran número de nodos. Este enfoque descentralizado ofrece aplicaciones potenciales en el monitoreo de estacionamientos, análisis de movilidad urbana en tiempo real y otras soluciones de gestión urbana inteligente.

Este tipo de soluciones pueden contribuir significativamente al desarrollo de ciudades inteligentes y sostenibles, donde la gestión eficiente del espacio urbano resulta crucial.

El proyecto representa una convergencia exitosa entre ingeniería embebida, inteligencia artificial ligera y diseño de sistemas distribuidos, demostrando su viabilidad técnica y su potencial de aplicación en el mundo real.

Palabras Clave: Edge AI, TinyML, TensorFlow Lite Micro, Microcontrolador, ESP32-S3, BLE, NB-IoT, Homografía.

Esta página ha sido intencionalmente dejada en blanco.

Prefacio

El prefacio comparte el lado más personal del proyecto: cómo surgió la idea, cómo fue el trabajo en equipo y qué experiencias atravesamos en el camino. Porque, más allá de los resultados técnicos, hubo un recorrido humano que también forma parte de esta historia.

Este proyecto nació en una charla entre integrantes del equipo, como suelen empezar muchas ideas: a partir de un problema cotidiano —la dificultad de estacionar en el centro de la ciudad— y la intuición de que podíamos llegar a una solución con herramientas de la carrera. La propuesta inicial buscaba aplicar técnicas de inteligencia artificial, pero desde un enfoque diferente al que domina la conversación global. En lugar de grandes modelos fundacionales, pensamos en soluciones pequeñas, específicas y útiles: inteligencia artificial en el borde, en dispositivos embebidos, soluciones a medida que se pudieran implementar en Uruguay.

La idea nos permitió integrar distintos intereses y perfiles: desde la visión por computadora hasta las telecomunicaciones, desde el diseño de hardware hasta la transmisión de datos en tiempo real. A lo largo del proceso, fuimos descubriendo que no basta con tener una buena idea o una buena pregunta. Hay que encontrar la profundidad en esa idea, cuánta tela hay para cortar, y eso requiere tiempo, paciencia y muchas pruebas. El firmware fue implacable. La integración entre módulos, dolorosa. El tiempo, escaso. Uno de nosotros incluso se casó en medio del proyecto. Pero el trabajo siguió. Y entre pausas, dudas, pruebas y errores, fueron apareciendo los momentos que nos recordaban por qué estábamos haciendo esto: cada pequeña victoria técnica, cada modelo que funcionaba en un dispositivo ridículamente chico, cada vez que el sistema mostraba que, tal vez, podía funcionar.

Este trabajo no pretende ser una solución cerrada. Es una prueba de concepto, una evidencia de que es posible construir tecnología útil, segura y moderna desde Uruguay, sin necesidad de replicar modelos masivos de otros contextos. También es una muestra del lugar único desde el que la Ingeniería Eléctrica puede aproximarse a la Inteligencia Artificial, permitiendo trabajar con los sensores que adquieren los datos, así como con su transmisión y procesamiento.

Cuando imaginábamos este sistema funcionando, lo veíamos como algo más que un detector de autos: un módulo al que podrían sumarse más capacidades, como detectar contenedores llenos, alertar accidentes o robos, o simplemente mejorar la forma en que usamos el espacio urbano. Aún no llegamos a todo eso. Pero lo que hicimos, lo hicimos en serio. Lo que está acá se probó, se rompió, se discutió y se volvió a hacer.

Esta tesis también representa, en cierto sentido, el final de una etapa. Como la carrera misma, fue un camino largo, con curvas inesperadas y decisiones difíciles. No elegimos una carrera fácil, ni un proyecto fácil. Pero elegimos bien. Y si este trabajo algún día ve la luz, cada vez que logremos estacionar más rápido gracias a él, sabremos que valió la pena.

Quizás no sea el invento que soñábamos de niños, pero es, de algún modo, una

versión adulta de ese mismo sueño. Y tal vez, con suerte, alguien que lea estas páginas se convenza de que todavía se puede inventar lo que uno imagina.

Tabla de contenidos

Prefacio ix

1. Diseño General **1**
1.1. Motivación . . . 1
1.2. Diseño de la Solución . . . 2
1.2.1. Dispositivo Parkwise . . . 4
1.2.2. Servidor Central . . . 5
1.2.3. Gateway . . . 5
1.3. Diagrama del Sistema Completo . . . 6
1.3.1. Funcionamiento del Sistema . . . 6
1.4. Conclusiones del Diseño General . . . 7

2. Fundamentación del Hardware y Framework Utilizados **9**
2.1. Selección de Hardware para el Dispositivo Parkwise . . . 9
2.1.1. Evaluación de dispositivos . . . 10
2.2. Evaluación de Frameworks . . . 12
2.2.1. Frameworks para ESP32S3 . . . 13
2.3. Diseño, Justificación y Pruebas con Gateway en el Sistema Completo . . . 14
2.3.1. Motivación de la introducción del Gateway . . . 15
2.3.2. Diseño del Gateway en hardware . . . 15
2.3.3. Desarrollo y pruebas con el Gateway físico . . . 15
2.3.4. Conclusiones del Gateway y posibles mejoras . . . 16
2.4. Conclusiones en Hardware . . . 17

3. Desarrollo e Implementación de Algoritmos de Aprendizaje Automático **19**
3.1. Enfoques de Detección de Ocupación Vehicular Basados en Visión Artificial . . . 20
3.2. Selección y Preparación de Datasets . . . 21
3.2.1. Edge Impulse Toy Dataset . . . 21
3.2.2. Car Parking Lot Dataset (CARPK) . . . 23
3.2.3. Proceso de etiquetado . . . 24
3.2.4. Tomas Diago Dataset . . . 25
3.2.5. Drone Dataset . . . 26
3.3. Implementación de Modelos de Aprendizaje Automático en Microcontroladores . . . 28
3.3.1. Diferencias entre TensorFlow Lite y TensorFlow Lite For Microcontrollers . . . 28
3.3.2. Relación del Converter e Interpreter en TensorFlow Lite y TensorFlow Lite for Microcontrollers . . . 29

3.4. Integración de Edge Impulse 30
3.4.1. Flujo de trabajo con Edge Impulse para dispositivos embebidos 30
3.4.2. Arquitectura FOMO: Faster Objects, More Objects 31
3.4.3. Detalles Técnicos del Modelo FOMO 32
3.5. Resultados 35

4. Desarrollo e Implementación del Firmware 37
4.1. Arquitectura General y Flujo de Ejecución 37
4.2. Diseño de Tareas 39
4.3. Ejecución del Modelo de Inferencia 41
4.4. Diseño e Integración del Modelo de Inferencia 42
4.4.1. Fundamentos Aplicados de TinyML en el Firmware 43
4.4.2. Desarrollo de Modelos Propios y Limitaciones 44
4.5. Diseño del Pipeline de Inferencia Autónomo 45
4.6. Retorno a Edge Impulse: Optimización y Resultados 46
4.7. Integración de la Cámara 47
4.7.1. Desafíos Técnicos y Soluciones Implementadas 48
4.7.2. Importancia de Librerías Preexistentes 48
4.7.3. Conversión RGB888 y Configuración Final 48

5. Comunicación inalámbrica 49
5.1. Introducción General 49
5.2. Introducción al Stack NimBLE 51
5.2.1. GAP y GATT: Arquitectura Funcional y Modelo de Datos en BLE 51
5.2.2. Diseño e Implementación del Servidor GATT con NimBLE . . 51
5.2.3. Proceso de Conexión y Suscripción 53
5.3. Comunicación BLE entre Dispositivo Parkwise y Gateway 54
5.3.1. Actores en BLE 54
5.3.2. Serialización a Formato JSON 55
5.3.3. Actualización del Búfer de Detecciones 56
5.3.4. Recepción desde el Gateway 56
5.4. Comunicación BLE entre Parkwise (ESP32-S3) y Operario 58
5.4.1. Servicio de envío de imágenes 58
5.4.2. Reconstrucción de imágenes 60
5.4.3. Posibles mejoras a futuro 61
5.5. Comunicación BLE entre el Dispositivo Parkwise y Servidor Central . 61
5.6. Introducción al Broker Mosquito y NB-IoT 62
5.6.1. MQTT: un protocolo liviano para IoT 63
5.6.2. Mosquitto como broker MQTT 63
5.6.3. NB-IoT: comunicación celular de bajo consumo 63
5.6.4. Resumen de integración 63
5.7. Comunicación IoT entre Gateway y Servidor Central a través del Broker Mosquito 64
5.7.1. Estructura general de la comunicación 64
5.7.2. Plataforma de desarrollo y bibliotecas utilizadas 64
5.7.3. Gestión de publicaciones y calidad de servicio 65
5.7.4. Gestión de reconexión y latencia 65
5.7.5. Recepción de detecciones en el Servidor Central 65
5.7.6. Reenvío de comandos desde el servidor 66
5.8. Hosteo del Servidor Local y Acceso desde la Red 66
5.9. Integración y Flujo Completo 66

5.10. Conclusión del capítulo 67

6. Desarrollo e Implementación del Software de Alto Nivel 69
6.1. Introducción 69
6.1.1. Propósito del Software desarrollado 69
6.1.2. Modos de funcionamiento y Guía de Uso de la Interfaz 70
6.1.3. Metodología de desarrollo 73
6.1.4. Tecnologías utilizadas 74
6.2. Arquitectura General del Sistema 75
6.2.1. Diagrama de bloques funcionales 75
6.2.2. Módulos principales: Frontend, Backend, Comunicaciones, Base de Datos 76
6.2.3. Explicación del Flujo General de datos y responsabilidades . . 77
6.3. Base de Datos Local (parkwise.db) 78
6.3.1. Estructura de la base `parkwise.db` (tablas, claves foráneas) . . 78
6.3.2. Esquema Entidad-Relación 78
6.3.3. Módulo de acceso: consultas, inserciones y actualizaciones . . . 79
6.3.4. Consideraciones de Acceso Remoto, Persistencia, Sincronización y posibles mejoras 80
6.4. Backend: Lógica del Servidor Central (app.py) 81
6.4.1. Descripción de módulos clave: Flask app, manejo de rutas, persistencia 81
6.4.2. Diagrama de Secuencia de Sistema (DSS) y casos de uso principales 82
6.4.3. Diagrama de secuencia y casos de uso principales 82
6.5. Frontend Web (index.html) 84
6.5.1. Estructura del `index.html` y componentes 84
6.5.2. Diagrama de dependencias de JavaScript 84
6.5.3. Interacción dinámica con el usuario y el servidor 84
6.6. Módulo de Comunicación Inalámbrica (`wireless_comm.py`) 85
6.6.1. Diagrama de clases 86
6.6.2. Diagrama de comunicación 87
6.6.3. Principios aplicados: SOLID, DRY, encapsulamiento, etc 88
6.7. Comunicación entre Módulos 89
6.7.1. Diagrama de Módulos 89
6.7.2. Sincronización y Coordinación entre Subsistemas 89
6.8. Conclusiones del capítulo 92
6.8.1. Resumen y fortalezas del diseño 92
6.8.2. Lecciones aprendidas y desafíos enfrentados 92
6.8.3. Consideraciones para seguridad y escalabilidad futura 93
6.8.4. Proyecciones de mejora y líneas de trabajo futuro 93

7. Pruebas y Validación 95
7.1. Pruebas del Modelo de Inferencia FOMO 95
7.2. Pruebas del Dispositivo Parkwise 98
7.2.1. Verificación de la captura periódica de imágenes mediante la cámara integrada. 98
7.2.2. Validación de la ejecución local del modelo FOMO para detección de vehículos. 98
7.2.3. Medición de latencia entre adquisición e inferencia. 99
7.2.4. Evaluación del uso eficiente de recursos embebidos (CPU, memoria). 100

7.2.5. Verificación del estado de estacionamiento: Detección de ocupación instantánea . . . 100
7.3. Pruebas de Comunicación BLE (Parkwise → Gateway) . . . 101
7.3.1. Estado inicial sin suscripciones . . . 101
7.3.2. Conexión y suscripción desde el Gateway . . . 101
7.3.3. Recepción de detecciones por parte del Gateway . . . 102
7.3.4. Confirmación de transmisión en el Dispositivo Parkwise . . . 102
7.3.5. Desconexión y retorno al estado inactivo . . . 102
7.3.6. Medición del alcance efectivo de la comunicación BLE. . . . 103
7.3.7. Evaluación del resultado . . . 103
7.4. Pruebas de Comunicación NB-IoT (Gateway → Servidor Central) . . . 103
7.4.1. Publicación y suscripción a tópicos MQTT sobre la red NB-IoT 103
7.4.2. Medición de latencia y confiabilidad en la transmisión de detecciones . . . 104
7.5. Pruebas del Servidor Central . . . 105
7.5.1. Verificación del almacenamiento de detecciones en la base de datos 105
7.5.2. Aplicación correcta de la homografía para georreferenciación . 106
7.6. Pruebas del Proceso de Instalación . . . 107
7.6.1. Funcionamiento del script `operario.py` para captura y envío de imagen . . . 107
7.6.2. Configuración de homografía y veredas desde la interfaz web . 108
7.7. Pruebas de Integración Punta a Punta . . . 109
7.7.1. Flujo completo . . . 109
7.8. Pruebas de Tolerancia y Escalabilidad del Sistema . . . 109
7.8.1. Escenario con múltiples dispositivos y Gateways . . . 110
7.8.2. Tolerancia ante fallos parciales . . . 111

8. Conclusiones **113**

A. Curso de Proyecto **115**
A.1. Objetivo del proyecto . . . 115
A.2. Alcance . . . 115
A.3. Criterios de Éxito . . . 115
A.3.1. Software . . . 116
A.3.2. Comunicación . . . 116
A.3.3. Hardware . . . 116
A.4. Supuestos y Restricciones . . . 116

B. Material de Referencia **119**
B.1. Person Detection . . . 119

C. Avance en la Implementación del Gateway **121**

D. Exploración Inicial con TinyML **123**
D.0.1. Introducción: Predecir Valores de la Función Seno . . . 123
D.0.2. Funcionamiento de TensorFlow Lite Converter e Interpreter . . 125
D.0.3. Conversión Final para uso en Microcontroladores . . . 125

Referencias **127**

Glosario **138**

Índice de tablas **138**

Índice de figuras **140**

Esta página ha sido intencionalmente dejada en blanco.

Capítulo 1

Diseño General

1.1. Motivación

En numerosos centros urbanos del mundo, encontrar un lugar disponible para estacionar en la vía pública representa un desafío recurrente, especialmente en determinadas franjas horarias del día. La escasez de espacios suele concentrarse en zonas de alta demanda como áreas comerciales, educativas, deportivas o turísticas. Generando no solo pérdidas de tiempo para los conductores, sino también dificultades logísticas para los comercios locales y un impacto negativo sobre la fluidez del tránsito urbano.

La ciudad de Montevideo no es ajena a esta problemática, que responde a una tendencia clara: cada vez hay más vehículos en circulación. Como se observa en la figura 1.1, en Uruguay la cantidad de vehículos por habitante no ha dejado de crecer en los últimos años, sin señales de reversión a corto plazo.

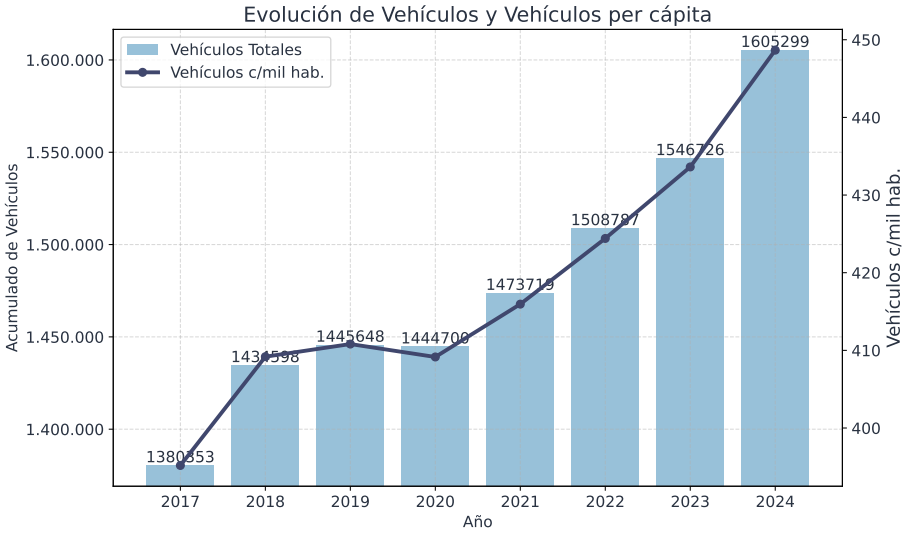


Figura 1.1: Acumulado de vehículos en Uruguay y la proporción de vehículos cada mil habitantes.Fuente [27].

Este crecimiento sostenido del parque vehicular, combinado con los resultados del

Censo 2023 —el cual indica que el tamaño medio de los hogares uruguayos es de tan solo 2,5 personas— [21], permite inferir una alta tasa de motorización a nivel doméstico, cercana a un vehículo por hogar.

Esta creciente densidad vehicular sobre la misma infraestructura urbana exige repensar la movilidad para hacerla más fluida y eficiente, reduciendo tanto la saturación de las calles como el estrés cotidiano de quienes las transitan. En este contexto, el fenómeno del *cruising* —circulación lenta de vehículos en busca de un lugar para estacionar— representa un problema doble: incrementa los tiempos de traslado, y al mismo tiempo contribuye directamente a la congestión general del tránsito, afectando a todos los usuarios de la vía pública.

Las soluciones tradicionales aplicadas por las autoridades locales son variadas. Algunas optan por ampliar los espacios de estacionamiento mediante obras estratégicas; sin embargo, estas medidas suelen ser costosas y, durante su ejecución, pueden agravar temporalmente la congestión. Mas aún, la capacidad de expansión física es limitada, lo que las vuelve insostenibles a largo plazo.

Otras municipalidades han implementado esquemas de circulación vehicular basados en el número de matrícula, restringiendo el tránsito de ciertos vehículos según el día. No obstante, esta estrategia no ha tenido los resultados esperados, ya que ha incentivado la adquisición de vehículos adicionales por parte de los hogares, lo cual incrementa la motorización [16].

Por otro lado, proyectos como *SFpark* en San Francisco (Estados Unidos) [34] han adoptado parquímetros con tarifado dinámico basado en la demanda. Esta estrategia logró una reducción del tiempo promedio de búsqueda de estacionamiento de hasta un 43 %.

En función de estos antecedentes, este trabajo permite dos soluciones relacionadas al estacionamiento en la vía pública, orientadas a disminuir el tiempo y la distancia recorrida para buscar estacionamiento.

Por un lado, plantea una solución de bajo costo que permitiría llevar adelante un tarifado dinámico con precios ajustados en tiempo real según la demanda. Por otro, brindar a los conductores la disponibilidad de estacionamientos en su lugar de destino en tiempo real; si se conecta con una aplicación que indique el camino a destino, puede tener en cuenta el punto de estacionamiento más cercano. Esto les facilita decidir si utilizar el vehículo particular o recurrir al transporte público, o incluso redirigir su trayecto hacia zonas menos saturadas.

La ocupación de los espacios se infiere mediante una pequeña computadora, conocida como sistema embebido, equipada con una cámara capaz de reconocer en tiempo real los lugares libres. Al procesar la información localmente, la cantidad de datos generados es mucho menor que en la transmisión de video en directo, lo que reduce los costos de comunicación y operación, y facilita su implementación a gran escala.

La información procesada se envía a un servidor central, que la refleja sobre un mapa digital en tiempo real, de modo que los usuarios puedan visualizar con facilidad las plazas disponibles o integrarla a otros servicios de movilidad.

El desarrollo de un sistema de estas características no solo valida la viabilidad técnica del enfoque, sino que también contribuye a una gestión más inteligente del tránsito y genera una fuente de datos valiosa para analizar el comportamiento vehicular y planificar políticas de movilidad.

1.2. Diseño de la Solución

La solución propuesta se estructura en tres componentes principales interconectados: el Dispositivo Parkwise, el Servidor Central y el Gateway.

El dispositivo Parkwise identifica los vehículos estacionados en la vía pública mediante la captura de imágenes y su procesamiento con algoritmos de Inteligencia Artificial, lo que permite detectar la ocupación de los espacios y enviar la información resultante a un servidor remoto.

El Servidor Central actúa como núcleo de integración al recopilar las detecciones enviadas por los dispositivos, procesa la información y determina en tiempo real los espacios libres de estacionamiento. Esta información se refleja en un mapa interactivo que permite a los usuarios identificar con facilidad los espacios disponibles y orientarse hacia el más cercano a su destino.

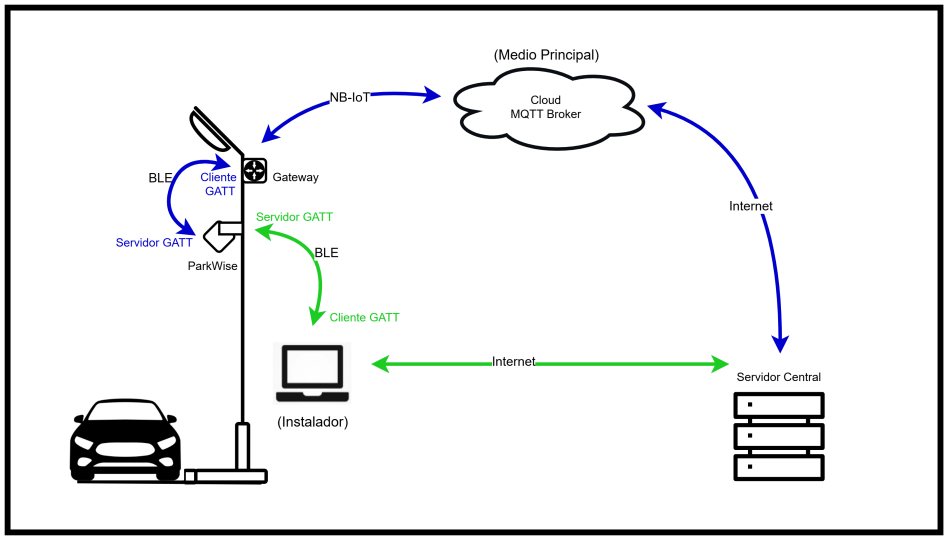


Figura 1.2: Diagrama de la arquitectura física. Se muestran los vínculos inalámbricos entre los componentes. El instalador, con conexiones en verde, estará únicamente al inicio.

Para permitir que varios Dispositivos Parkwise cercanos se conecten a Internet, el sistema incorpora un *Gateway* que funciona como punto de acceso centralizado. Este dispositivo recibe las detecciones de los Parkwise, traduce los datos entre los protocolos locales y los envía al Servidor Central, y a su vez permite que los dispositivos reciban comandos desde el servidor en tiempo real. De esta manera, se garantiza la interoperabilidad entre todos los dispositivos cercanos y el servidor, sin necesidad de dar acceso a Internet individual a cada Parkwise. El Gateway permite además que existan otra serie de dispositivos que aprovechen esta conexión.

Actualmente, algunas luminarias en las calles de Montevideo cuentan con un módulo controlador ubicado en su parte superior. Este módulo puede ser utilizado como Gateway dentro del enfoque propuesto en esta tesis. La idea de su uso es aprovechar un recurso ya existente y disponible, optimizando así la infraestructura urbana.

Durante la instalación, un operario próximo al Dispositivo Parkwise se conecta de manera inalámbrica y le solicita una fotografía. Esta imagen la utiliza el instalador para asociar los puntos visibles por la cámara con coordenadas en el mapa, a través de una función de transformación (homografía) que se calcula y almacena en el Servidor Central.

1.2.1. Dispositivo Parkwise

La implementación de una solución capaz de detectar vehículos estacionados en un área determinada requiere la integración de múltiples tecnologías. El proceso se inicia en el Dispositivo Parkwise, sistema embebido que integra un microprocesador junto con memoria y periféricos específicos para ejecutar tareas dedicadas. Está diseñado para operar en tiempo real y en entornos con recursos limitados, gestionando la interacción con sensores y otros dispositivos mediante protocolos de comunicación definidos. En este caso, se incorpora una cámara que proporciona capacidad visual al sistema, permitiendo la adquisición y procesamiento de imágenes directamente en el dispositivo. Además, cuenta con un módulo de comunicación inalámbrica BLE y su correspondiente antena, que facilitan la transmisión de datos hacia un Gateway.

El Dispositivo Parkwise periódicamente adquiere una imagen y luego realiza una inferencia haciendo uso de un modelo de inteligencia artificial basado en redes neuronales para la detección de vehículos. Este modelo recibe la imagen como entrada y produce, como salida, un conjunto de cajas delimitadoras (*bounding boxes*) que indican la ubicación en la imagen de los vehículos presentes. A partir de estas coordenadas se genera un paquete de datos que se transmite mediante BLE al Gateway correspondiente.

El dispositivo seleccionado para llevar a cabo estas tareas es el Seeed XIAO ESP32S3 Sense, un módulo basado en el SoC ESP32 y el módulo de expansión Sense, que integra los componentes necesarios para ejecutar el modelo neuronal, capturar imágenes y transmitir las *bounding boxes*. La figura 1.3 muestra el dispositivo.

Las características del módulo ESP32S3 que resultan especialmente relevantes para este proyecto son: un procesador Xtensa LX7 de 32 bits con doble núcleo, 512 KB de memoria SRAM, 8 MB de PSRAM, conectividad Bluetooth Low Energy (BLE) en el módulo de expansión Sense se encuentra una cámara OV2640 de 2 megapíxeles y una velocidad de captura de hasta 15 cuadros por segundo (fps).

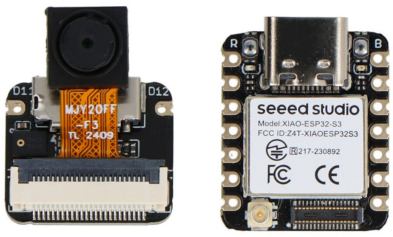


Figura 1.3: Seeed XIAO ESP32S3 (Derecha) y el módulo Sense (Izquierda)

La red neuronal utilizada para la detección de vehículos en este proyecto es la red FOMO (Faster Objects, More Objects), tiene una arquitectura de detección de objetos optimizada para dispositivos con recursos limitados, como microcontroladores. Esta arquitectura permite realizar detección en tiempo real con un requerimiento de memoria bajo (inferior a 200KB), lo que la hace ideal para aplicaciones embebidas.

En FOMO, la imagen de entrada la interpreta como una cuadrícula, donde cada celda corresponde a una región específica de la imagen. A cada celda se le asigna un valor que indica la probabilidad de que un objeto de interés se encuentre dentro de esa región. De este modo, el modelo produce un mapa de calor (heatmap) en el que las celdas con valores más altos representan una mayor probabilidad de contener

un vehículo. A partir de estas activaciones, es posible reconstruir la ubicación de los objetos mediante técnicas de posprocesamiento que derivan las cajas delimitadoras.

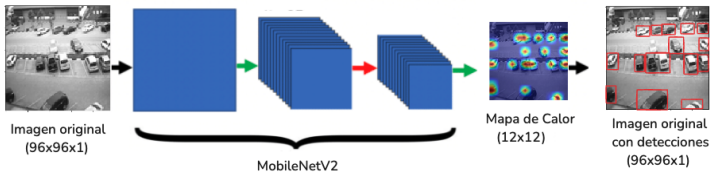


Figura 1.4: Representación entrada-salida de la red.

Mediante la aplicación de una homografía en el Servidor Central, se transforman automáticamente las coordenadas de las detecciones desde la perspectiva de la cámara hacia su representación precisa sobre un mapa para su visualización por el usuario.

1.2.2. Servidor Central

El Servidor Central ejecuta el software encargado de procesar la información proveniente de los Dispositivos Parkwise. Para comunicarse con los dispositivos, utiliza MQTT, un protocolo ligero de mensajería diseñado para redes IoT. En MQTT, los datos se publican en tópicos dentro de un servidor llamado broker, y cualquier suscriptor de ese tópico puede recibirlos. En nuestro sistema, el Gateway publica las detecciones de los dispositivos en un tópico, y el Servidor Central se suscribe a él para recibir en tiempo real las *bounding boxes* detectadas, y en ciertos casos, imágenes capturadas durante la instalación. Además, permite enviar comandos a los dispositivos, como reinicios remotos, garantizando la operación coordinada de toda la red.

El sistema ofrece una interfaz de usuario accesible tanto para el Usuario Final como para el Operario encargado de la instalación. Esta interfaz permite al Operario configurar la homografía a partir de la fotografía enviada y de la posición de la cámara, estableciendo la transformación necesaria entre la imagen capturada y el entorno georreferenciado.

Una vez recibidas las detecciones, el servidor aplica la homografía previamente definida para proyectar las coordenadas desde la perspectiva de la cámara hacia un sistema de referencia geográfico. Esta información es visualizada en un mapa dentro de la interfaz web del sistema.

Finalmente, el Servidor Central gestiona una base de datos de dos tablas: una destinada al almacenamiento de las detecciones, y otra para registrar el estado operativo y la configuración de cada Dispositivo Parkwise (ESP32-S3) desplegado. Este diseño modular permite escalar la solución fácilmente, ya que un único servidor puede gestionar múltiples Gateways en simultáneo ($1 : N$).

1.2.3. Gateway

El Gateway es un dispositivo intermediario que actúa como puente entre los Dispositivos Parkwise y el Servidor Central, asegurando la interoperabilidad entre distintos protocolos de comunicación, que permiten establecer la manera en que interactúan los actores. Con los dispositivos locales se comunica mediante Bluetooth Low Energy (BLE) usando el protocolo GATT (Generic Attribute Profile), que organiza la información en atributos que pueden leerse, escribirse o suscribirse para recibir actualizaciones automáticas. Para conectarse al servidor, el Gateway utiliza la red celular NB-IoT, que

permite transmitir datos de forma bidireccional, eficiente en consumo de energía y adecuada para comunicaciones de bajo ancho de banda o poca frecuencia.

De forma análoga, el Gateway se suscribe a las características definidas por Parkwise para recibir las detecciones de vehículos, las cuales posteriormente transmite al servidor utilizando MQTT, un protocolo de mensajería ligero basado en el modelo publicador/suscriptor, que permite enviar datos a un tópico en un servidor intermediario conocido como broker.

Asimismo, el Gateway está suscrito a un tópico específico desde el cual el Servidor Central puede publicar instrucciones, como una orden de reinicio, que luego es enviada por BLE al Dispositivo Parkwise.

1.3. Diagrama del Sistema Completo

En la figura 1.5 encontramos el flujo de información, los bloques y sus interacciones.

El Dispositivo Parkwise captura imágenes de los estacionamientos mediante su cámara y ejecuta inferencias para detectar vehículos, generando una lista de bounding boxes. Estas son enviadas al Gateway a través de Bluetooth Low Energy (BLE), protocolo en el que cada dispositivo posee un identificador único (dirección MAC). El Gateway añade esta dirección como identificador de origen y reenvía las bounding boxes al Servidor Central.

La transmisión se realiza mediante el protocolo MQTT, que publica la información en un topic dentro de un servidor MQTT (denominado broker). El Servidor Central está suscrito a dicho topic, por lo que recibe automáticamente las detecciones asociadas a cada dirección MAC.

El Servidor Central integra tanto el *backend* como el *frontend* del sistema: el primero recibe, procesa y almacena las *bounding boxes*, mientras que el segundo despliega la información al usuario. Además, el dispositivo envía al *backend* una imagen de referencia junto con su posición y puntos de calibración. Con estos datos, el sistema calcula una transformación que proyecta las *bounding boxes* detectadas sobre un mapa digital, ubicando los vehículos en los cordones de las veredas. Esta transformación, que permite llevar los puntos del plano de la imagen al plano del mapa, se denomina homografía. El frontend toma forma sobre una interfaz web que permite ver sobre un mapa digital la posición de las cámaras, los cordones de las veredas que cambian de color de verde a rojo para marcar que son espacios ocupados y las etiquetas de los vehículos estacionados detectados en tiempo real.

Finalmente, el sistema también admite solicitudes de reinicio del dispositivo, que se transmiten empleando los mismos protocolos y medios de comunicación.

1.3.1. Funcionamiento del Sistema

El instalador u operario ejecuta el algoritmo encargado de conectarse al Dispositivo Parkwise, solicitarle una fotografía y enviarla por internet al Servidor Central.

En una segunda etapa, el operario accede a la interfaz web del servidor, donde puede seleccionar el Dispositivo Parkwise para configurar su homografía y ubicarlo en el mapa. También permite dimensionar y delimitar las veredas, que mostrarán dentro del rango que permite ver la cámara, una proyección de las detecciones, para marcar los lugares ocupados. La figura 1.8 muestra estos ocupados en rojo y los libres en verde.

A partir de dos imágenes se presenta la pantalla del modo instalador en la interfaz web del servidor: la imagen capturada por el operario (fig. 1.6) y el mapa (fig. 1.7). Una vez definidos los puntos de correspondencia, el dispositivo queda configurado para la correcta recepción de sus detecciones.

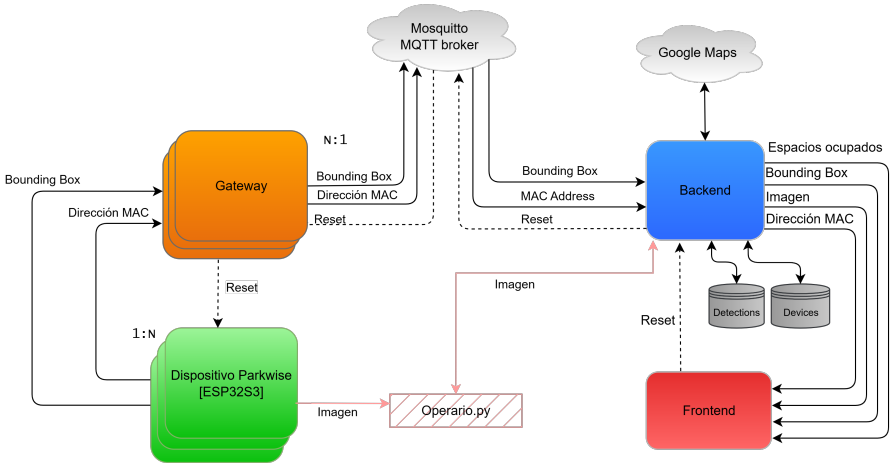


Figura 1.5: Diagrama de flujo de información del Sistema Parkwise.

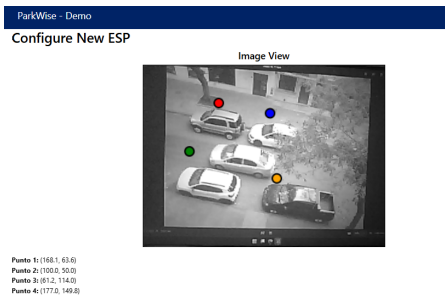


Figura 1.6: Image View en modo instalador: despliega la imagen recibida del Operario, con los puntos de referencia en la imagen, que se corresponderán en el mapa.

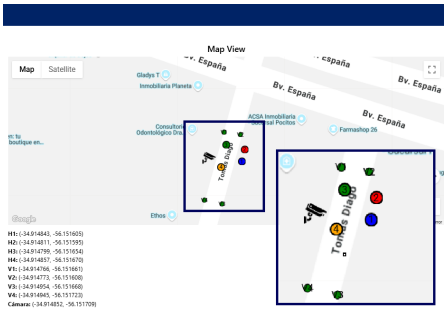


Figura 1.7: Vista del mapa en modo Instalador, con la correspondencia entre puntos de la imagen y del entorno. En el recuadro azul (abajo a la derecha) se amplía la zona de interés.

Una vez configurado el Dispositivo Parkwise, se pueden apreciar sus detecciones en la interfaz web de usuario, donde se muestran además los espacios ocupados, la cámara y los puntos que definen la homografía.

1.4. Conclusiones del Diseño General

El diseño general del sistema Parkwise permitió integrar de forma coherente y funcional múltiples tecnologías, desde hardware embebido y redes neuronales ligeras hasta comunicaciones inalámbricas y visualización web. Cada componente fue desarrollado con un enfoque modular y orientado a la eficiencia operativa.

En este sentido, el módulo con el SoC ESP32S3 fue seleccionado por sus características técnicas, la disponibilidad de firmware compatible, y su eficiencia en la comunicación con los Gateways. Estos Gateways pueden implementarse como algoritmos escritos en Python, ejecutados en dispositivos capaces de comunicarse mediante BLE y NB-IoT, o alternativamente mediante plataformas como Arduino, siempre que el

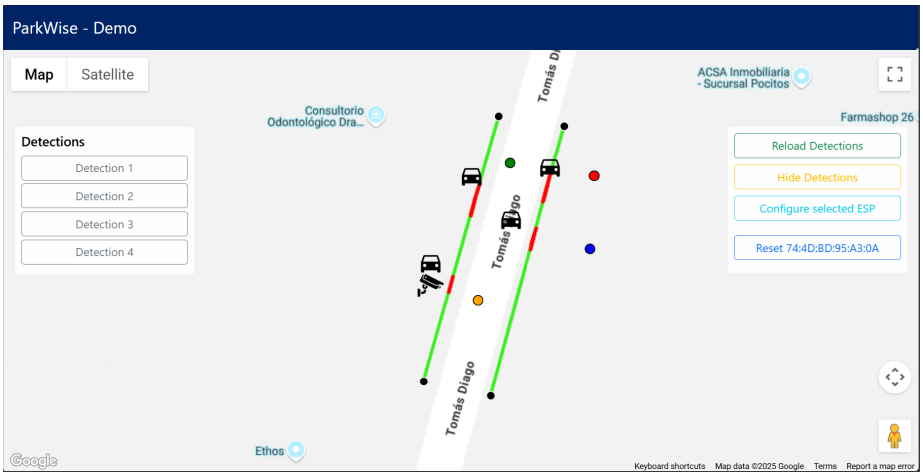


Figura 1.8: Interfaz de usuario: Mapa que ven los usuarios.

hardware permita esta doble conectividad.

El Servidor Central, por su parte, se comunica con los Dispositivos Parkwise a través de los Gateways. Entre los datos que recibe se encuentran las inferencias sobre ocupación de los espacios de estacionamiento, el estado de conexión de los dispositivos y, durante la instalación, imágenes capturadas por las cámaras. Estas imágenes permiten establecer puntos de referencia sobre el mapa para calcular la homografía.

Toda la información se almacena en una base de datos en el *backend*. Estos datos son consumidos por el *frontend* del sistema, para desplegar las predicciones y el estado del sistema en una interfaz web.

A través de este diseño, se logró implementar una solución completa que detecta vehículos en tiempo real, transforma sus coordenadas a un sistema georreferenciado y las representa sobre un mapa digital interactivo.

Capítulo 2

Fundamentación del Hardware y Framework Utilizados

En el presente capítulo se describen los elementos considerados para la elección del hardware y firmware para llevar adelante este proyecto.

Para la implementación del Dispositivo Parkwise se seleccionó el dispositivo Seeed XIAO ESP32S3 y su módulo Sense, una solución compacta que integra capacidades de procesamiento, visión embebida y comunicación inalámbrica. La elección de este dispositivo respondió principalmente al objetivo de minimizar los costos de hardware, seleccionando la opción más económica que permitiera ejecutar las tareas requeridas por el sistema. Entre estas tareas se destaca la ejecución local de modelos de aprendizaje automático para detección de vehículos en imágenes, utilizando técnicas de detección de objetos, así como la comunicación eficiente con otros dispositivos a través de Bluetooth Low Energy (BLE).

El desarrollo del firmware se llevó a cabo utilizando el entorno oficial provisto por Espressif Systems, el Espressif IoT Development Framework (ESP-IDF) [12]. Una ventaja relevante de este ecosistema es que el núcleo de Arduino para ESP32 está construido sobre ESP-IDF [11], lo que asegura compatibilidad con la amplia base de recursos y ejemplos disponibles en la plataforma Arduino. Esta característica resultó especialmente valiosa en el contexto de TinyML (Inteligencia Artificial en Sistemas Embebidos de Recursos Limitados), donde se tomaron como referencia diversos proyectos de la compañía Edge Impulse [9], especializada en la investigación y el desarrollo soluciones de TinyML, los cuales sirvieron como punto de partida para el desarrollo del firmware del sistema embebido.

2.1. Selección de Hardware para el Dispositivo Parkwise

El dispositivo debía satisfacer varios requisitos clave: capturar imágenes, ejecutar modelos de detección de objetos localmente y comunicarse de forma inalámbrica con un Gateway. La conectividad debía ser compatible con BLE, dado que el Gateway disponible se encontraba ya instalado en luminarias del espacio público, y estaba compuesto por un Arduino Nano 33 BLE conectado a un módulo Mikroe NB-IoT 2 Click. Esta infraestructura motivó la decisión de instalar los Dispositivos Parkwise en luminarias, permitiendo una red distribuida sin requerir conectividad celular en cada nodo.

La elección del hardware constituye uno de los aspectos más determinantes en

un proyecto de *TinyML*, especialmente al tratarse de un sistema embebido donde la arquitectura del dispositivo impone restricciones directas sobre el software. Un error en esta etapa puede derivar en la necesidad de rediseñar completamente el firmware, e incluso sustituir periféricos y sensores por incompatibilidad, generando importantes pérdidas de tiempo y recursos.

Desde el punto de vista económico, el diseño del sistema priorizó reducir el costo total de propiedad. Por ello, se optó por un esquema de arquitectura con un único Gateway por grupo de dispositivos, lo que permite compartir el plan de datos y simplifica el mantenimiento operativo. No obstante, esto introdujo desafíos de software importantes: dado que la comunicación BLE es de tipo 1:N, el Gateway debe gestionar la conexión secuencial con cada dispositivo para recopilar sus datos, incluyendo las *bounding boxes* generadas a partir de los modelos de detección.

Al iniciar el diseño del sistema, se consideró la alternativa de procesar las imágenes en la nube, pero se descartó al ser incompatible con el criterio de descentralización del sistema. Este enfoque no solo centraliza la carga computacional, sino que también introduce costos operativos y variables adicionales que se deberían prever. En contraste, la arquitectura de Edge AI adoptada permite que cada dispositivo funcione de manera autónoma, aumentando la robustez del conjunto. Además, este modelo ofrece una ventaja fundamental en términos de privacidad: al procesar la información localmente, no se transfieren imágenes de la vía pública, sino únicamente los metadatos anonimizados resultantes, garantizando la confidencialidad y minimizando el uso de la red.

Los modelos empleados de inteligencia artificial utilizan TensorFlow Lite Micro, diseñados para ejecutarse en entornos con recursos limitados. Entre las tareas más exigentes se encuentran la carga de la red neuronal en memoria, la captura y preprocesamiento de imágenes, y la generación de bounding boxes a partir de mapas de calor que identifican vehículos en la escena. En un principio, el sistema debía construirse desde cero, sin contar con una versión previa del software, lo que implicaba cierto nivel de incertidumbre respecto al uso real de memoria y procesamiento. Por ello, se priorizó la selección de dispositivos que ofrecieran compatibilidad entre frameworks y la posibilidad de reutilizar código entre alternativas. Como mecanismo de contingencia, también se consideraron placas con mayor nivel de abstracción —como aquellas que permiten programar en Python— para facilitar una adaptación rápida en caso de ser necesario.

La búsqueda se centró en dispositivos capaces de ejecutar modelos de visión por computadora con bajo consumo y buen soporte comunitario. Se relevaron foros, repositorios y plataformas como Edge Impulse para identificar casos de uso similares. Los criterios de selección incluyeron: bajo costo, capacidad de captura de imagen, conectividad inalámbrica BLE, compatibilidad con modelos de detección en imágenes, y existencia de proyectos documentados en plataformas activas.

Finalmente, el dispositivo Seeed XIAO ESP32S3 Sense fue seleccionado por cumplir con todos los requisitos funcionales, ser el más accesible en términos de costo entre las opciones viables, y contar con una comunidad activa y ejemplos funcionales similares al caso de uso propuesto. Su arquitectura basada en el framework ESP-IDF, con compatibilidad con Arduino, ofreció la flexibilidad necesaria para integrar componentes, acelerar el desarrollo y garantizar la escalabilidad futura del sistema.

2.1.1. Evaluación de dispositivos

Se evaluaron distintas plataformas ampliamente utilizadas en el desarrollo de sistemas embebidos y aplicaciones de inteligencia artificial. En la tabla 2.1 se presenta un resumen comparativo de las especificaciones técnicas relevantes, costos estimados y observaciones específicas de cada dispositivo.

Dispositivo	**Precio (US$)**	**CPU**	**RAM**	**Comentarios**
Raspberry Pi 5	93	Quad-core ARM Cortex-A76	4GB	Potente, pero costo elevado. Se le debe sumar el costo de la cámara.
Raspberry Pi 4 model B. 4GB RAM	60	Quad-core ARM Cortex-A72	4GB	Potente, pero costo elevado. Se le debe sumar el costo de la cámara.
Google Coral	80	Quad-core ARM Cortex-A53 + Edge TPU		Muy potente, pero costo elevado. Se le debe sumar el costo de la cámara.
Seeed Xiao ESP32S3 Sense	13,99	Xtensa LX7 dual-core, 32-bit	512 KB SRAM + 8MB PS-RAM	Muy accesible, incluye la cámara y se encuentran proyectos para detección de personas disponibles públicamente. Compatible con múltiples plataformas. También cuenta con BLE.

Tabla 2.1: Comparativa de dispositivos

A partir de los dispositivos evaluados y las dificultades que podían presentar si se encuentra que no es posible implementar esta aplicación en ellos, se optó por un plan de acción que fuera incrementando las prestaciones, manteniéndose en el precio por unidad más bajo, e intentando mantener el software tan compatible como sea posible en la medida que cambiamos de dispositivo. Se creó asi una lista de alternativas a evaluar, XIAO ESP32S3 Sense, XIAO ESP32S3 + Grove Vision AI V2 kit, Raspberry Pi 4 4GB. En estos dispositivos hay similitudes, por un lado la compatibilidad de XIAO ESP32S3 con el Grove Vision AI V2, que permite llevar la comunicación, y por último un Raspberry Pi 4, que permitiría incluso implementaciones de machine learning en python y permitiendo además instalar TensorFlow.

Luego del análisis comparativo, se seleccionó el módulo XIAO ESP32S3 Sense como plataforma principal del sistema. Esta decisión respondió a una combinación de factores clave para el proyecto: capacidad de procesamiento, soporte de frameworks, compatibilidad con modelos tinyML, tamaño reducido y costo competitivo.

El ESP32S3 incorpora un procesador dual-core Xtensa LX7 a 240 MHz, aceleradores para operaciones vectoriales SIMD, y soporte nativo para inferencia en redes neuronales con instrucciones especializadas. Además, dispone de 512KB de SRAM interna y 8MB de PSRAM externa, lo que permite cargar modelos ligeros de visión por computadora y procesar imágenes en tiempo real. La cámara en el módulo de extensión es de 1600×1200 píxeles, combinada con su conectividad BLE/WiFi, facilita la captura, inferencia y transmisión de datos sin necesidad de módulos externos.

Durante el desarrollo, resultó especialmente útil la posibilidad de alternar entre el entorno de desarrollo Arduino y el framework nativo de Espressif (ESP-IDF), lo que permitió realizar pruebas rápidas y, al mismo tiempo, optimizar el firmware para producción.

Desde el punto de vista físico, su factor de forma compacto ($21 \times 17{,}5mm$) permite una integración sencilla en luminarias u otros dispositivos embebidos. A esto se suma un precio significativamente menor respecto a otros módulos con capacidades similares, lo que lo convierte en una solución eficiente y escalable. En la figura 2.2 se ilustran sus dimensiones físicas, y en la figura 2.1, sus principales componentes.

2.2. Evaluación de Frameworks

Dado el alcance y los plazos establecidos para este proyecto, se descartó tempranamente la opción de desarrollar controladores de bajo nivel o módulos personalizados para el hardware. Si bien dicha estrategia podría haber generado un conocimiento más profundo del sistema, representaba un riesgo considerable en términos de tiempo y desviación del objetivo principal. Por este motivo, la selección del framework se convirtió en un eje estratégico, buscando maximizar la reutilización de componentes existentes y reducir la carga de desarrollo.

El framework elegido debía ofrecer soporte completo para el uso de TensorFlow Lite Micro, gestión eficiente de PSRAM, integración con la cámara y conectividad BLE, así como capacidades de concurrencia tipo RTOS (Real Time Operating System). También se priorizó la existencia de ejemplos funcionales en línea para acelerar la curva de adopción.

Durante la fase de exploración, se probaron diversos entornos: Arduino, Zephyr y PlatformIO (combinando tanto ESP-IDF como Arduino). La opción que mejor equilibró flexibilidad, profundidad de control y ecosistema fue **ESP-IDF**, gracias a su robusto soporte para periféricos, compatibilidad con bibliotecas Arduino mediante el *Arduino Core* y documentación clara y extensa.

El enfoque inicial se apoyó en proyectos abiertos como *Car Parking Occupancy Detection (FOMO)* y *Fruits vs Bugs* de Edge Impulse, y un ejemplo básico de predicción de una función seno. La adaptación de estos modelos requirió comprender a fondo el funcionamiento de TensorFlow Lite Micro, el preprocesamiento de imágenes, la gestión de la memoria y la interpretación de salidas del modelo (bounding boxes). También se integró el stack de comunicación BLE utilizando el ejemplo de GATT Server provisto por Espressif, extendido para transmitir imágenes y resultados de inferencia.

En términos generales, el proyecto se posicionó entre dos extremos del espectro TinyML: los entornos cerrados de tipo "entry-level" —que facilitan resultados rápidos pero sin capacidad de adaptación— y los entornos de desarrollo avanzados —que

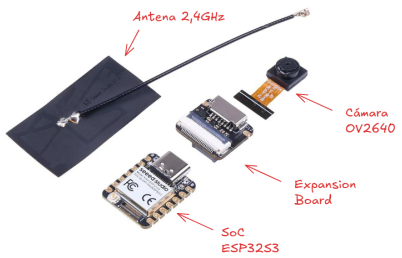


Figura 2.1: Seeed XIAO ESP32S3 desarmado, vista de sus componentes

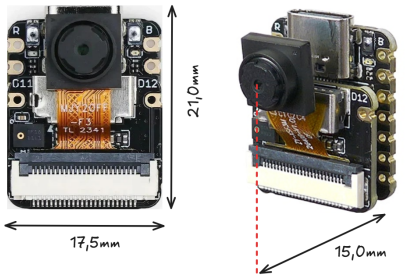


Figura 2.2: Dimensiones de Seeed XIAO ESP32S3

requieren modificar firmware o SDKs. Elegir ESP-IDF permitió encontrar un punto de equilibrio: alta personalización sin comprometer tiempos ni complejidad técnica excesiva.

Este proceso de selección se detalla con mayor profundidad en la siguiente sección, donde se comparan formalmente los frameworks evaluados.

2.2.1. Frameworks para ESP32S3

Arduino

El *Arduino Build System* transforma los *sketches* (archivos `.ino`) en código `C++` válido, generando automáticamente los prototipos de funciones necesarios y agregando la inclusión de la librería `Arduino.h`, que proporciona acceso a las funciones básicas y a los periféricos del dispositivo. Además, gestiona la inclusión y validación de las librerías utilizadas en el proyecto durante la compilación, incluyendo aquellas que permiten ejecutar modelos de aprendizaje automático, como TensorFlow Lite Micro (TensorFlow Lite for Microcontrollers), la cual fue utilizada en este proyecto.

Este sistema se utiliza comúnmente junto con el Arduino IDE, donde se crean y editan los *sketches*. El IDE cuenta con un *Board Manager*, encargado de dar soporte a los distintos dispositivos. También puede emplearse junto con PlatformIO.

Zephyr

Zephyr es un sistema operativo en tiempo real (RTOS). Tiene una arquitectura modular que hace a su código adaptable a diferentes arquitecturas de hardware y le permite funcionar únicamente con los componentes necesarios, optimizando así el uso de recursos. Además, ofrece un manejo de memoria flexible, que puede configurarse según las necesidades de cada aplicación. Utiliza Device Tree, una estructura jerárquica que permite al sistema operativo entender la estructura del hardware, y también KConfig que se encarga de definir el hardware que se utilizará.

Este sistema, en la familia ESP32 no tiene un IDE, por lo que se compila y desde la consola con West[1]. En el momento que escribimos este documento, Zephyr tiene un soporte muy limitado para TensorFlow Lite Micro[2].

ESP-IDF

Espressif – IoT Development Framework, abreviado como ESP-IDF, es el framework oficial proporcionado por Espressif para el desarrollo de aplicaciones sobre sus microcontroladores, pertenecientes a las diferentes series ESP. Según su documentación, se recomienda utilizarlo junto con alguno de los siguientes entornos de desarrollo integrados (IDE):[3] el complemento para Eclipse o la extensión para Visual Studio Code. No obstante, también es posible utilizarlo desde la terminal o mediante PlatformIO.

Si bien el framework se instala normalmente junto con el IDE, cada instalación puede configurarse para utilizar una versión específica de ESP-IDF, e incluso cambiarla modificando la variable de entorno que define su ubicación (`IDF_PATH`).

[1] https://docs.zephyrproject.org/latest/develop/west/index.html

[2] TensorFlow Lite for Microcontrollers en Zephyr Project https://docs.zephyrproject.org/latest/samples/modules/tflite-micro/tflite-micro.html Acccedido el 19-04-2025

[3] ESP-IDF for ESP32S3, Getting Started: `https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/get-started/index.html`

Una de las características más destacadas de ESP-IDF es la posibilidad de configurar prácticamente todos los parámetros del dispositivo a través del archivo sdkconfig, lo que otorga una gran flexibilidad en comparación con otros frameworks y permite explotar al máximo el potencial del microcontrolador. Asimismo, ESP-IDF permite integrar bibliotecas como TensorFlow Lite for Microcontrollers (TFLM) para ejecutar modelos de aprendizaje automático directamente en el dispositivo.

En el caso del ESP32, Edge Impulse genera una librería compatible con el entorno Arduino, basada en el Arduino Core para ESP32, el cual se construye sobre el propio ESP-IDF. De este modo, es posible escribir el código del proyecto en $C++$, utilizando las funciones típicas del entorno Arduino, como `setup()` y `loop()`.[4][5]

Según las diferentes pruebas realizadas, encontramos que el Framework ESP-IDF fue el mas conveniente para nosotros. Por un lado, cuenta con las librerías que necesitamos para poder trabajar con la Xiao ESP32S3 Sense, también cuenta con muchos repositorios, mientras que los que están en Arduino fueron adaptables en muchos casos. Por otra parte, con Zephyr encontramos que no existía firmware que soporte la cámara OV2640 en el ESP32S3. Por lo que se decidió descartar este camino. Por el lado de platformIO, si bien permitía tomar proyectos tanto de Arduino como de Espressif, no tenía todas las librerías que soportaba ESP-IDF, además no permite usar ambas en el mismo proyecto (Se debe elegir sobre qué framework estamos trabajando). Y su uso tenía complicaciones específicas de la plataforma. Dado que no se podía concebir un código que fuera independiente del dispositivo y que estamos sobre hardware de ESP, se decidió utilizar ESP-IDF.

2.3. Diseño, Justificación y Pruebas con Gateway en el Sistema Completo

El Gateway fue concebido originalmente como parte de un proyecto para instalar un sistema embebido con conexión a internet sobre luminarias en la vía publica. Esta solución permitiría utilizar la infraestructura de comunicación al reducir la cantidad de puntos de acceso a internet y aprovechar de manera más eficiente los datos generados por las luminarias.

Aunque el desarrollo completo en el sistema embebido no pudo concretarse debido a contratiempos con el hardware, en esta sección se presenta el enfoque planificado para su finalización, junto con los fundamentos que respaldan la viabilidad técnica de la propuesta. Se ofrece además una visión general del rol del Gateway en la arquitectura del sistema: su incorporación resulta clave para validar la comunicación entre el Dispositivo Parkwise y el Servidor Central.

Para su evaluación, se implementó una maqueta funcional que contemplaba la conectividad mediante NB-IoT; aunque esta tecnología no llegó a utilizarse efectivamente en el prototipo final, se documenta su integración por motivos de completitud.

Finalmente, se describe la versión emulada del Gateway, que reproduce su funcionalidad desde una PC o laptop mediante scripts en Python. Esta versión establece comunicación con los Dispositivos Parkwise a través de BLE, y se conecta directamente a Internet mediante Wi-Fi, sin requerir NB-IoT.

[4] Documentación de Edge Impulse para Arduino: `https://docs.edgeimpulse.com/docs/run-inference/arduino-library`

[5] Arduino Core para ESP32: `https://github.com/espressif/arduino-esp32`

2.3.1. Motivación de la introducción del Gateway

Dado que uno de los objetivos del proyecto es minimizar los costos de implementación y operación, la posibilidad de enviar nuestros datos a través de un dispositivo que permita el acceso a internet a múltiples dispositivos representa una reducción significativa en los costos de implementación. Si bien es una opción técnicamente viable, no es necesario que cada Dispositivo Parkwise cuente con acceso propio a internet, lo que también disminuye los costos fijos. Como el procesamiento se realiza localmente en el Dispositivo Parkwise, la cantidad de datos a transmitir es mínima. Esta arquitectura permite además desacoplar la conectividad a internet del procesamiento: en lugares donde hay poca señal, se puede ubicar el Dispositivo Parkwise en la posición más conveniente para su tarea y colocar el Gateway en un punto con mejor acceso a internet.

2.3.2. Diseño del Gateway en hardware

El diseño del Gateway tuvo como objetivo construir un dispositivo capaz de actuar como puente entre los Dispositivos Parkwise y el servidor central, gestionando la comunicación entre tecnologías distintas: BLE para la conexión local y NB-IoT para la conexión a internet. Para lograrlo, se recurrió a una arquitectura modular compuesta por tres bloques de hardware, cada uno con un rol bien definido dentro del sistema.

Se contaba con un prototipo de controlador de luminaria con conectividad BLE . El controlador tiene conexión a la luminaria vía un conector NEMA-7. Está compuesto por dos placas dedicadas y está basada en *Arduino Nano 33 BLE* y una placa que provee conectividad a Internet vía NB-IoT o LoRaWAN. Sin embargo, el controlador fue desarrollado usando Zephyr, siendo complejo introducir modificiaciones en tiempos acotados. Es por eso que se decidió optar por armar una maqueta con los mismos módulos o kits utilizados.

La arquitectura del Gateway se compone de tres bloques. El primero es la unidad de procesamiento central, materializada en la placa Arduino Nano 33 BLE [3]. Se eligió este dispositivo por su microcontrolador nRF52840, que ofrece conectividad Bluetooth Low Energy 5.0 nativa, ideal para actuar como punto de enlace con los Dispositivos Parkwise. El segundo bloque es el módulo de comunicación celular MIKROE LTE IoT 2 Click [26], basado en el chipset Quectel BG96. Este se encarga de la conectividad a la red de área amplia (WAN) a través de NB-IoT, gestionada mediante comandos AT. Su función es publicar los datos recibidos en un broker MQTT, un servidor intermediario que gestiona la distribución de mensajes bajo un modelo de publicación-suscripción, desacoplando los clientes que producen información de los que la consumen [30]. Finalmente, una placa controladora de luminaria integra ambos componentes, conectando el puerto serial del Arduino con el MikroBUS del módulo LTE y proporcionando alimentación al conjunto.

La figura 2.3 muestra el diagrama general del sistema, mientras que en la figura 2.4 se puede observar el montaje físico del hardware y sus conexiones de alimentación, tanto a 5V como a 3.3V.

2.3.3. Desarrollo y pruebas con el Gateway físico

El desarrollo del Gateway sobre el controlador de la luminaria implicó la implementación de un algoritmo que permitiera traducir entre dos tecnologías y protocolos distintos: BLE, utilizado por los Dispositivos Parkwise, y NB-IoT, utilizado para la comunicación con el servidor. Mientras que el Arduino Nano 33 BLE intercambia datos mediante el protocolo GATT usando NimBLE, el módulo LTE IoT 2 Click se comunica con el servidor a través del protocolo MQTT sobre NB-IoT.

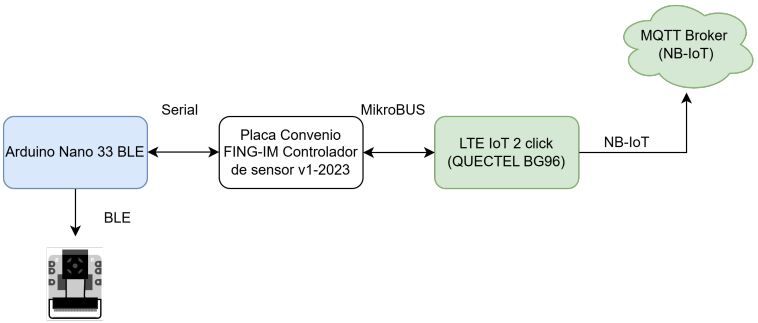


Figura 2.3: Diagrama del Gateway y sus conexiones

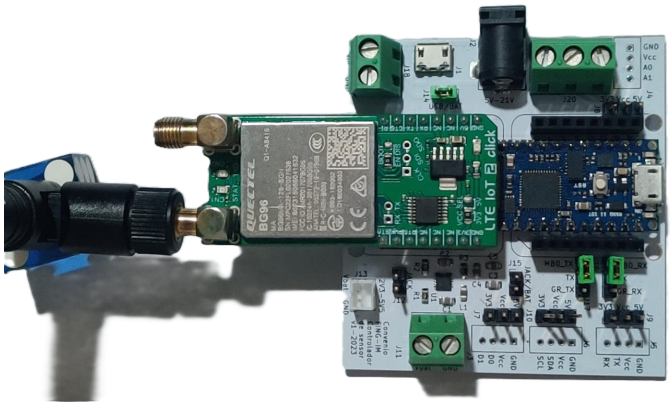


Figura 2.4: Hardware del Gateway

Para lograr esta interoperabilidad, se definieron los pasos funcionales, que se resumen en la Tabla 2.2.

Dado el tiempo disponible para la implementación, se optó por desarrollar estas funcionalidades en un script de Python corriendo sobre una laptop como alternativa al Gateway físico mencionado anteriormente. Los avances realizados para poner en marcha este último se pueden apreciar en el apéndice C.

2.3.4. Conclusiones del Gateway y posibles mejoras

La implementación de un Gateway con conectividad $1 : N$ a través de BLE presenta ciertos desafíos operativos, ya que, si bien el protocolo permite conexiones simultáneas con múltiples periféricos, estas están limitadas por los recursos del sistema, como la memoria disponible y el *stack* utilizado. En dispositivos con recursos restringidos, como el ESP32, esto puede implicar que el Gateway deba gestionar las conexiones de forma secuencial, rotando entre los distintos Dispositivos Parkwise conectados a la red. En este contexto, resulta fundamental incorporar mecanismos de control, como comandos que instruyan a los dispositivos a mantenerse inactivos o en modo espera tras la conexión inicial, permitiendo al Gateway reconectarse de manera ordenada según el ciclo de atención definido.

Desde el punto de vista del flujo de datos, se propone unificar en un único paquete

Tabla 2.2: Funciones requeridas para la operación del Gateway

Función	**Descripción**
GATT Write	Suscripción a un servicio BLE y escritura en una característica GATT desde el Arduino. Incluye la inicialización correspondiente.
GATT Read	Suscripción a un servicio BLE y lectura desde una característica GATT, con su respectiva inicialización.
AT MQTT Write	Configuración de comandos AT en el LTE IoT 2 Click para conectarse al broker MQTT y publicar datos recibidos por serial desde el Arduino.
AT MQTT Read	Configuración de comandos AT para leer datos del broker MQTT y enviarlos al Arduino por serial.
Loop principal	Ciclo principal que integra las funciones anteriores y las gestiona como tareas dentro de un sistema basado en RTOS.

las *bounding boxes* generadas por todos los dispositivos en una cuadra, enviándolas como una única publicación MQTT. Esta estrategia permitiría reducir significativamente el tráfico de datos, y facilitaría la aplicación de técnicas de compresión y cifrado, optimizando así el uso del canal de comunicación.

Aunque estas mejoras agregan cierta complejidad al Gateway, no comprometen su rol de componente intermedio del sistema. Por el contrario, habilitan su escalabilidad, permitiendo que una única conexión a internet gestione múltiples dispositivos de forma eficiente. Este enfoque contribuye a una arquitectura distribuida con mayor inteligencia en el borde, lo que se traduce en una reducción significativa de los costos operativos, especialmente considerando el bajo costo relativo del nodo Seeed XIAO ESP32S3 Sense frente al costo del acceso a internet.

2.4. Conclusiones en Hardware

El diseño del hardware se abordó considerando desde el inicio la interacción entre sus componentes principales: el dispositivo embebido, el *framework*, el Gateway y los protocolos de comunicación. Esto permitió garantizar que todos los elementos fueran compatibles y funcionaran de manera integrada, evitando problemas en etapas posteriores.

Se seleccionó el Seeed XIAO ESP32S3 Sense principalmente por su capacidad de procesamiento local, que facilita realizar inferencias directamente en el dispositivo, reduciendo la necesidad de enviar grandes volúmenes de datos al servidor.

El Gateway fue diseñado como un puente entre los Dispositivos Parkwise y el servidor, convirtiendo la comunicación BLE (GATT) de los dispositivos en mensajes MQTT que se transmiten a través de NB-IoT. Esta solución permite que múltiples dispositivos compartan una única conexión a internet, lo que reduce costos y simplifica la infraestructura del sistema.

Esta arquitectura modular favorece la escalabilidad y flexibilidad del sistema, al facilitar la conexión de más dispositivos sin incrementar la complejidad ni los costos asociados a la conectividad.

Esta página ha sido intencionalmente dejada en blanco.

Capítulo 3

Desarrollo e Implementación de Algoritmos de Aprendizaje Automático

El desarrollo e implementación de algoritmos de Inteligencia Artificial (IA) para dispositivos de bajo consumo, implica una serie de etapas que van desde la selección y preparación de los datos hasta la optimización e integración del modelo en hardware. Este capítulo presenta una visión general de este proceso, así como de las herramientas y recursos disponibles.

Uno de los primeros desafíos en este camino es la disponibilidad de conjuntos de datos adecuados. Existen numerosos *datasets* públicos que han sido utilizados para tareas similares, como el Edge Impulse Toy Dataset, proporcionado por Edge Impulse, que ofrece ejemplos simples para la tarea de detección de vehículos en entornos controlados, y el Car Parking Lot Dataset (CARPK), utilizado para la detección y conteo de vehículos en estacionamientos desde vistas aéreas. Sin embargo, las características específicas del entorno —como las marcas de automóviles, las señales de tránsito y el tamaño de las calles—, hacen necesaria la generación de *datasets* propios. En este trabajo, se desarrollaron dos conjuntos de datos personalizados: Tomas Diago Dataset y Drone Dataset, con el objetivo de tener un modelo capaz de adaptarse a condiciones similares a las de Montevideo. El proceso de etiquetado y organización de los datos se desarrolla en la sección 3.2.3.

Al diseñar modelos de IA destinados a dispositivos embebidos, es necesario enfrentar restricciones significativas en términos de memoria, capacidad de cómputo y consumo energético. Este tipo de hardware impone límites estrictos que hacen inviable el uso directo de modelos convencionales de Aprendizaje Profundo. Por lo tanto, es indispensable emplear técnicas de compresión y optimización que reduzcan el tamaño y la complejidad del modelo de IA sin comprometer su precisión en exceso. Estas limitaciones se abordan específicamente en la sección del apéndice D, donde se introduce la exploración inicial de modelos simples utilizando Aprendizaje Automático en Sistemas Embebidos con Recursos Limitados (*TinyML*), y se detalla el Funcionamiento de TensorFlow Lite Converter e Interpreter, así como la Conversión Final para uso en Microcontroladores.

Para la realización de este proyecto se utilizó TensorFlow Lite Micro (TFLM), esta herramienta permite ejecutar modelos de aprendizaje automático en microcontroladores con menos de 256 KB de RAM [41]. No obstante, su uso implica una serie de

desafíos adicionales: requiere una conversión previa del modelo entrenado, ajustes de cuantización y una adaptación de las operaciones a aquellas compatibles con la versión de TFLM. Estos aspectos técnicos se desarrollan en la sección 3.3, donde se explican las diferencias entre TensorFlow Lite y TensorFlow Lite for Microcontrollers, la relación entre la herramienta *Converter* (encargada de preparar y optimizar el modelo) y el componente *Interpreter* (encargado de reservar memoria, ejecutar el modelo y manejar entradas y salidas). Finalmente se describe el flujo completo, desde el entrenamiento del modelo utilizando la API Keras hasta el despliegue en un microcontrolador.

El campo de TinyML ha ganado notable relevancia en los últimos años, dando lugar a un ecosistema en expansión de herramientas que buscan simplificar este pipeline (desde el entrenamiento hasta el despliegue). Estas plataformas ofrecen entornos integrados donde es posible recolectar datos, etiquetarlos, entrenar modelos, optimizarlos para diversos tipos de hardware y desplegarlos en dispositivos reales. Algunas de las más destacadas incluyen Edge Impulse [9], *Neuton* [29], *SensiML* [35] y los flujos de trabajo combinados de *Google Colab* con *TFLite*. En particular, *Edge Impulse* se ha consolidado como una plataforma clave en el ecosistema *TinyML*, ya que, además de tener una gran comunidad, permite una integración fluida entre sensores, flujos de datos, herramientas de etiquetado, entrenamiento y exportación de modelos optimizados para una variedad de microcontroladores, todo desde un entorno web accesible. Esta plataforma se explora en detalle en la sección 3.4, donde se describe su flujo de trabajo, el uso de la arquitectura FOMO, y los detalles técnicos del modelo entrenado.

3.1. Enfoques de Detección de Ocupación Vehicular Basados en Visión Artificial

Esta sección presenta un relevamiento del estado del arte en relación con sistemas de detección de ocupación vehicular basados en visión artificial y aprendizaje automático para dispositivos con recursos limitados (TinyML). Se exploran diversas aproximaciones tecnológicas desarrolladas en los últimos años, incluyendo enfoques descentralizados, métodos tradicionales de visión por computadora, arquitecturas de deep learning y casos de implementación en plataformas embebidas.

En el trabajo de Cai et al. [7] se distinguen dos enfoques principales para la detección de ocupación vehicular: los métodos basados en el espacio, que analizan zonas específicas para determinar si están ocupadas o libres, pero requieren anotaciones manuales intensivas y presentan baja generalización; y los métodos basados en vehículos, que utilizan técnicas de detección de objetos para identificar directamente automóviles en la escena, siendo más adecuados para entornos dinámicos. En este contexto, los autores implementan Mask R-CNN, entrenado con el conjunto de datos COCO, como modelo principal de segmentación de instancias, permitiendo obtener tanto las cajas delimitadoras como las máscaras de los vehículos. Para mejorar la robustez ante oclusiones, condiciones climáticas adversas o errores aleatorios, se complementa con un filtro de seguimiento diseñado para reforzar la continuidad y fiabilidad de las detecciones.

La detección de ocupación vehicular puede abordarse de forma descentralizada mediante cámaras inteligentes que procesan la información localmente, evitando así la dependencia de servidores centrales. En el trabajo de Amato et al. [2], se propone una solución basada en redes neuronales convolucionales profundas integradas en cámaras inteligentes, lo que permite reducir significativamente el tráfico de datos y los cuellos de botella computacionales.

Complementariamente, Kumar y Singh [23] presentan una alternativa basada en visión por computadora clásica, utilizando el algoritmo ORB (Oriented FAST and

Rotated BRIEF) para la detección de puntos clave. Este enfoque fue implementado sobre una plataforma NVIDIA Jetson Nano y demostró su capacidad para clasificar en tiempo real el estado de los espacios de estacionamiento. A diferencia de los métodos basados en aprendizaje profundo, esta solución impone una menor carga computacional y resulta especialmente adecuada para entornos donde el bajo consumo energético y la velocidad de respuesta son prioritarios.

El potencial del aprendizaje profundo para optimizar la gestión urbana ha sido ampliamente explorado en trabajos recientes. En particular, Zhou y Wang [43] analizan diversas aplicaciones del mismo en ciudades inteligentes, destacando su uso en la gestión del tráfico, la vigilancia en tiempo real, la predicción del consumo energético y la planificación urbana orientada a datos. Estos sistemas permiten una toma de decisiones más informada y adaptativa, gracias a su capacidad para procesar grandes volúmenes de información en entornos complejos.

Un caso específico de aplicación en el ámbito del estacionamiento urbano es presentado por Matsuda y Matsui [24], quienes desarrollan un sistema de detección y visualización en tiempo real utilizando cámaras móviles instaladas en vehículos patrulleros. Esta solución, implementada sobre una plataforma Jetson TX2, permite monitorear la disposición de los vehículos en la vía pública, facilitando tareas de fiscalización y control dinámico en escenarios urbanos. A diferencia de los enfoques estacionarios, la movilidad del sistema amplía su cobertura y eficacia operativa.

La revisión de las investigaciones y desarrollos actuales en el ámbito de los sistemas de estacionamiento inteligente revela un panorama en constante evolución, impulsado por avances en aprendizaje profundo, visión artificial y TinyML. La combinación de estas tecnologías permite la creación de soluciones más eficientes, escalables y adaptativas para la gestión del estacionamiento en entornos urbanos. Sin embargo, persisten desafíos relacionados con la generalización de los modelos, la variabilidad de las condiciones ambientales y la integración con infraestructuras existentes. La continua investigación y desarrollo en estas áreas será crucial para superar estas barreras y avanzar hacia ciudades más inteligentes y sostenibles.

3.2. Selección y Preparación de Datasets

En esta sección se introducen cuatro conjuntos de datos utilizados durante el desarrollo del sistema: dos de ellos son de acceso público —Edge Impulse Toy Dataset y Car Parking Lot Dataset— y los otros dos fueron recolectados y anotados por el equipo — Tomas Diago Dataset y Drone Dataset— con el objetivo de representar condiciones reales similares a las del entorno urbano de Montevideo.

Para los conjuntos desarrollados por el equipo, se detallan las herramientas empleadas para el proceso de etiquetado, incluyendo funcionalidades de anotación asistida por inteligencia artificial, y se establecen criterios específicos para garantizar la coherencia y calidad del dataset.

Estos conjuntos de datos cumplen un rol fundamental en la validación y entrenamiento de los modelos, así como en la evaluación integral del sistema, permitiendo abordar una variedad de escenarios de aplicación. Cada uno de ellos será descrito en mayor detalle a continuación.

3.2.1. Edge Impulse Toy Dataset

Descripción

Edge Impulse es una plataforma orientada al desarrollo y despliegue de soluciones de *machine learning on the edge*, es decir, modelos que se ejecutan directamente sobre

dispositivos de cómputo de baja potencia como microcontroladores, sensores o sistemas embebidos. Con el objetivo de facilitar la experimentación inicial, la plataforma pone a disposición diversos proyectos públicos con datasets, modelos entrenados y scripts de inferencia listos para ser utilizados.

Uno de estos proyectos es el experimento Car Parking Occupancy Detection, cuyo objetivo es detectar la ocupación de espacios de estacionamiento a partir de imágenes capturadas desde una vista cenital. Este proyecto incluye un conjunto de datos etiquetado, dividido en particiones de entrenamiento (train) y prueba (test), junto con un modelo preentrenado que puede descargarse en distintos formatos compatibles con dispositivos embebidos.

Características

El conjunto de entrenamiento se encuentra en formato NumPy e incluye un total de 101 imágenes de entrenamiento y 26 imágenes de prueba. Las imágenes presentan una resolución de 96x96 píxeles y están representadas en escala de grises, con valores normalizados entre 0 y 1. Se pueden apreciar algunos ejemplos en la figura 3.1.

Cada imagen puede contener una o más instancias del objeto de interés (vehículos), y sus correspondientes anotaciones están disponibles en formato JSON. Las etiquetas están definidas mediante *bounding boxes* con codificación en formato `[x, y, width, height]`, expresadas en coordenadas relativas respecto al tamaño de la imagen. Esto facilita su utilización en tareas de detección de objetos con arquitecturas livianas.

Figura 3.1: Ejemplo de imágenes de entrenamiento junto con sus anotaciones en formato `xywh` (Edge Impulse Toy Dataset)

Relevancia para este trabajo

Este dataset fue particularmente útil durante las etapas iniciales de validación del modelo FOMO, del que se hablará mas adelante, permitiendo probar la configuración de entrenamiento, la estructura de salida del modelo y los primeros ensayos de cuantización e inferencia en el MCU (Microcontroller Unit). Su simplicidad y compatibilidad con la plataforma Edge Impulse lo convierten en un recurso ideal para familiarizarse con el *pipeline* completo de entrenamiento, evaluación y despliegue de modelos TinyML.

3.2.2. Car Parking Lot Dataset (CARPK)

Descripción

Es un conjunto de datos diseñado para tareas de detección y conteo de vehículos en imágenes aéreas. Fue introducido por Hsieh et al. [18] en su trabajo sobre conteo de objetos basado en drones mediante redes convolucionales. Este conjunto contiene aproximadamente 90.000 automóviles capturados en 4 estacionamientos diferentes mediante un dron DJI Phantom 3 Professional, volando a una altura aproximada de 40 metros. Las imágenes ofrecen una vista cenital de alta resolución, lo que facilita la identificación y anotación precisa de los vehículos.

Características

El conjunto de datos CARPK presenta la siguiente estructura:

- **Imágenes**: Capturadas desde drones, en formato `*.png`, con nombres que indican el estacionamiento correspondiente.
- **Anotaciones**: Archivos de texto `*.txt` que contienen una línea por cada vehículo anotado, especificando las coordenadas de la caja delimitadora (*bounding box*) en formato `[x, y, width, height]` donde el punto referido es el centro de la imagen. Las anotaciones están alineadas con los nombres de las imágenes correspondientes.

Además, sus autores completaron las anotaciones de todos los automóviles en un subconjunto del conjunto de datos PUCPR, denominado PUCPR+, que contiene aproximadamente 16.456 automóviles. Las imágenes de PUCPR+ fueron tomadas desde el décimo piso de un edificio en la Pontificia Universidad Católica de Paraná, ofreciendo una perspectiva diferente pero complementaria a las imágenes de CARPK, ver ejemplos del dataset en la Figura 3.2.

Herramientas adicionales

El conjunto de datos incluye herramientas auxiliares para facilitar su uso:

- **Herramienta de visualización de cajas delimitadoras**: Permite superponer las anotaciones sobre las imágenes para verificar su precisión.
- **Herramienta de evaluación de conteo**: Scripts para calcular métricas como el *Mean Absolute Error (MAE)* y el *Root Mean Squared Error (RMSE)* en tareas de conteo de vehículos.

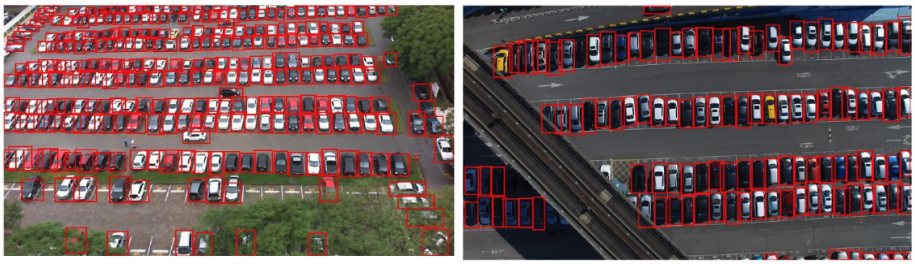


Figura 3.2: Ejemplos de imágenes del conjunto de datos PUCPR+ (izquierda) y CARPK (derecha), con sus respectivas anotaciones de cajas delimitadoras.

Relevancia para este trabajo

El conjunto de datos CARPK fue considerado en las etapas iniciales de este trabajo debido a su entorno realista y desafiante para la detección y el conteo de vehículos en imágenes aéreas. Su diversidad en escenarios y condiciones de iluminación, así como sus anotaciones detalladas, lo convierten en un recurso útil para realizar exploraciones preliminares. Sin embargo, no fue empleado directamente en ninguna aplicación ni en el desarrollo final del modelo.

3.2.3. Proceso de etiquetado

En las secciones anteriores se presentaron dos conjuntos de datos públicos disponibles en línea. Sin embargo, como se adelantó en el resumen de este capítulo, para lograr un modelo verdaderamente útil en contextos reales como el de Montevideo, fue necesario construir y etiquetar conjuntos de datos propios. En esta sección se describen las herramientas empleadas —principalmente Edge Impulse Studio— y las funcionalidades utilizadas para facilitar el proceso de anotación, incluyendo el etiquetado asistido por IA. Además, se detallan los criterios establecidos para garantizar la calidad, coherencia y relevancia de las anotaciones. Este proceso de etiquetado permitió la creación de los conjuntos de datos Tomas Diago Dataset y Drone Dataset, los cuales serán presentados en las próximas secciones.

Herramientas y estrategias de etiquetado

Para el proceso de etiquetado de los conjuntos de datos recolectados, se utilizó la plataforma Edge Impulse Studio, la cual ofrece una interfaz web intuitiva y herramientas avanzadas para la anotación de datos en proyectos de detección de objetos. Una de las funcionalidades destacadas es la *Labeling Queue*, que permite gestionar eficientemente las imágenes pendientes de etiquetado, mostrando únicamente aquellas que aún no han sido anotadas, lo que facilita el seguimiento del progreso y evita duplicaciones [20].

Para agilizar el proceso de etiquetado, se empleó la funcionalidad de AI-Assisted Labeling de Edge Impulse, específicamente utilizando el modelo YOLOv5 [22] preentrenado con el conjunto de datos COCO. Esta herramienta proporciona sugerencias automáticas de cajas delimitadoras para objetos comunes en las imágenes. Posteriormente, un operario revisa y corrige estas sugerencias, en un proceso de etiquetado supervisado, asegurando la precisión y relevancia de las anotaciones [19].

Criterios de etiquetado y control de calidad

Durante el proceso de anotación, se establecieron las siguientes directrices para garantizar la calidad y consistencia del conjunto de datos:

- **Exclusión de vehículos incompletos**: Si un automóvil no es completamente visible en la imagen (por ejemplo, parcialmente fuera del encuadre), se omite su etiquetado.
- **Filtrado de clases irrelevantes**: Si YOLOv5 sugiere una clase diferente a "car" (por ejemplo, "potted_plant"), se descarta dicha anotación (se desactiva fácilmente desmarcando la opción correspondiente en la interfaz).
- **Oclusiones parciales**: En casos donde un vehículo está parcialmente ocluido (por ejemplo, por un árbol), se etiqueta únicamente la parte visible.

- **Corrección de sugerencias incorrectas**: Si las sugerencias automáticas no están correctamente superpuestas sobre los objetos o presentan errores evidentes, se eliminan y se crean nuevas anotaciones precisas.
- **Ajuste de cajas delimitadoras**: Se ajustan manualmente las cajas sugeridas que no se alinean adecuadamente con los vehículos.
- **Omisión de objetos altamente ocluidos**: Si un vehículo está significativamente ocluido y no fue detectado por YOLOv5, se omite su etiquetado, considerando que el modelo final, al ser más ligero, probablemente tampoco lo detectaría.

Aunque existen métodos más avanzados de etiquetado asistido utilizando Modelos de Lenguaje Visual (Visual Large Models), como OWL-ViT [1] o GPT-4o [31], estos no fueron empleados en este proyecto debido a restricciones de recursos y al enfoque en herramientas integradas en Edge Impulse. Sin embargo, se reconoce el potencial de estas tecnologías para futuras mejoras en la eficiencia y precisión del etiquetado [19].

3.2.4. Tomas Diago Dataset

Descripción

Este conjunto de datos fue recolectado de forma manual por el equipo de trabajo como parte del desarrollo experimental de este proyecto. Las imágenes fueron capturadas desde el balcón de una vivienda particular ubicada en la calle Tomas Diago, en el barrio Pocitos de la ciudad de Montevideo, Uruguay. Se puede acceder al conjunto de datos haciendo click sobre este link e ingresando a la pestaña "Data acquisition".

El proceso de recolección se llevó a cabo durante un día, utilizando un sistema automático que capturaba una imagen por segundo de forma continua. De este modo, se logró obtener una base de datos variada con escenas reales del tránsito cotidiano sobre una calle urbana típica de Montevideo. El conjunto final está compuesto por un total de 312 imágenes. Se pueden ver algunos ejemplos en la figura 3.3.

Características

Las imágenes presentan condiciones visuales diversas, tanto en lo que respecta a la iluminación natural como a la variedad de vehículos presentes, incluyendo autos estacionados y en movimiento. Las imágenes fueron capturadas con una resolución de 2 mega píxeles y posteriormente redimensionadas y normalizadas para adaptarse a los requisitos de los modelos utilizados.

La anotación de los datos fue realizada manualmente mediante herramientas específicas de etiquetado, siguiendo el formato `xywh`, respetando los estándares planteados anteriormente.

Importancia del Dataset local

Uno de los principales beneficios de contar con un conjunto de datos capturado en el entorno específico de despliegue es la mejora en la capacidad de generalización del modelo. A diferencia de otros datasets públicos, cuyos escenarios pueden estar dominados por marcas, modelos de vehículos, y disposiciones urbanas propias de otras regiones del mundo; el dataset Tomas Diago refleja con precisión las condiciones reales del entorno en el que el sistema será implementado.

Entre los beneficios específicos de utilizar un dataset local se destacan:

Figura 3.3: Ejemplo de imágenes de entrenamiento junto con sus anotaciones en formato `xywh` (Tomas Diago Dataset)

- **Contexto urbano realista**: Las características físicas de la calle (ancho, orientación, bordes de acera, tipos de árboles y señalización) coinciden con las del entorno objetivo.
- **Relevancia del parque automotor**: Los modelos, marcas y estilos de vehículos presentes en Montevideo pueden diferir significativamente de los disponibles en datasets internacionales. Contar con ejemplos reales mejora la capacidad del modelo para detectar correctamente vehículos que efectivamente circulan y estacionan en Uruguay.
- **Condiciones lumínicas representativas**: Las variaciones de luz durante el día, típicas del clima local, están directamente representadas en las imágenes, permitiendo entrenar modelos más robustos ante las condiciones reales de operación.

3.2.5. Drone Dataset

Descripción

El **Drone Dataset** fue generado por el equipo de trabajo como parte de los ensayos exploratorios destinados a validar el funcionamiento del sistema en condiciones similares a las de producción. Las imágenes fueron capturadas utilizando un dron comercial en las inmediaciones del *Shopping Punta Carretas*, una zona céntrica y de alta circulación vehicular en la ciudad de Montevideo, Uruguay.

El dispositivo utilizado fue un CHUBORY A77 WiFi FPV Quadcopter, un dron para principiantes equipado con una cámara HD 2K. Esta cámara cuenta con un lente tipo ojo de pez, lo cual introduce una distorsión radial sobre las imágenes, se pueden apreciar ejemplos en la figura 3.4. Aunque esta distorsión no impide el reconocimiento general de los vehículos, podría afectar sutilmente el proceso de entrenamiento del modelo, especialmente en tareas donde la precisión es crítica.

Para la recolección de este conjunto de datos, se optó por capturar una serie de videos en las inmediaciones del Shopping Punta Carretas, específicamente en las calles Ariosto, Parva Domus y José Luis Zorrilla de San Martín. Cada vuelo del dron tuvo una duración aproximada de 15 segundos por toma. Posteriormente, los videos fueron procesados para extraer las capturas claras y representativas, haciendo énfasis en que se vieran ambas veredas, lo que resultó en un conjunto final compuesto por un total de 85 imágenes.

Características

Las capturas se realizaron en diferentes calles para representar variaciones lumínicas y contextuales. Las imágenes presentan una perspectiva cenital que facilita la visualización de los techos de los vehículos, su disposición respecto al cordón de la vereda, y la densidad del tráfico en un entorno urbano.

Las imágenes fueron posteriormente procesadas y anotadas de forma manual, utilizando herramientas de etiquetado respetando los criterios antes mencionados.

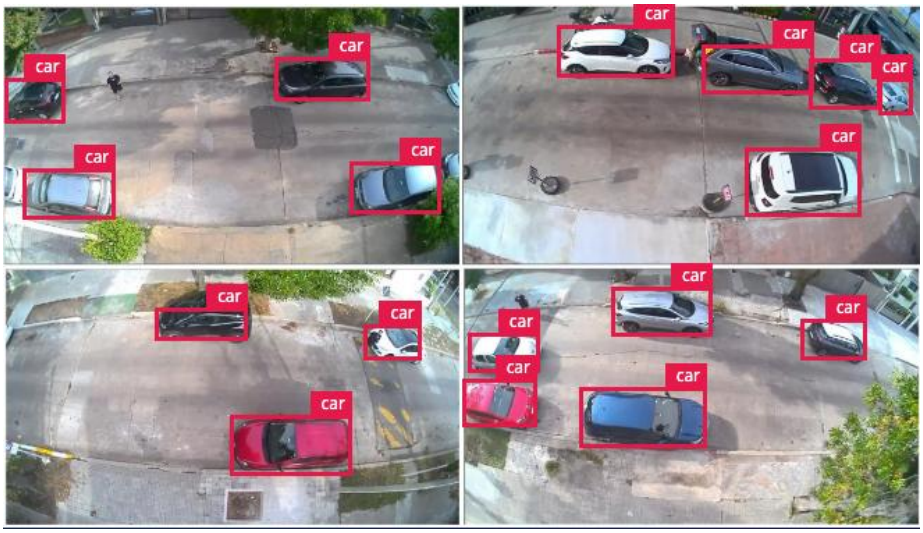


Figura 3.4: Ejemplo de imágenes de entrenamiento junto con sus anotaciones en formato `xywh` (Dron Dataset)

Importancia del Dataset Local

Al igual que *Tomas Diago Dataset*, *Drone Dataset* representa una valiosa fuente de datos adaptada a las condiciones reales del entorno de despliegue. En este caso, además, se aporta una perspectiva cenital clave para modelos de detección que operan desde puntos elevados, como luminarias, postes o infraestructura pública.

Entre sus principales beneficios se destacan:

- **Correspondencia espacial con la vista del sistema embebido**: Al capturar desde un punto elevado, las imágenes reflejan el ángulo de visión que tendrá el modelo en condiciones reales, optimizando su capacidad de detección.
- **Entorno urbano representativo**: La zona de Punta Carretas incluye veredas angostas, zonas de alto tránsito y variedad de situaciones comunes de estacionamiento, lo que enriquece el entrenamiento del modelo con casos reales.
- **Presencia de vehículos del parque automotor local**: Tal como se señaló en la sección anterior, el reconocimiento de vehículos típicos de Uruguay mejora el desempeño del modelo al reducir la brecha entre entrenamiento y producción.
- **Evaluación ante distorsiones ópticas reales**: El uso de un lente ojo de pez permite observar cómo se comporta el modelo ante distorsiones ópticas moderadas, un aspecto relevante al considerar la diversidad de cámaras posibles en el entorno de despliegue.

3.3. Implementación de Modelos de Aprendizaje Automático en Microcontroladores

En esta sección se explora el proceso completo que permite entrenar un modelo utilizando herramientas modernas como la API Keras, convertirlo a un formato optimizado para dispositivos embebidos mediante TensorFlow Lite, y finalmente integrarlo y ejecutarlo en un microcontrolador utilizando TensorFlow Lite for Microcontrollers (TFLM).

Para ello, se analiza en primer lugar la diferencia entre TensorFlow Lite (TFLite) y TensorFlow Lite for Microcontrollers, destacando las particularidades del entorno embebido. Luego, se profundiza en el rol que cumplen el *Converter* y el *Interpreter* durante las etapas de conversión y ejecución del modelo.

Finalmente, se presenta una subsección dedicada al flujo completo de implementación, detallando paso a paso cómo se pasa de un modelo entrenado en Python a un firmware compilado en C++ que puede ser ejecutado de forma autónoma en un microcontrolador. Esta etapa integra todos los componentes discutidos anteriormente y sienta las bases para el despliegue en el sistema final.

3.3.1. Diferencias entre TensorFlow Lite y TensorFlow Lite For Microcontrollers

TensorFlow Lite (TFLite) y TensorFlow Lite for Microcontrollers (TFLM) son dos variantes de la plataforma TensorFlow diseñadas para ejecutar modelos de aprendizaje automático en dispositivos con recursos limitados. Si bien ambos comparten el objetivo de facilitar la inferencia en sistemas embebidos, presentan diferencias significativas en cuanto a capacidades, requisitos de hardware y casos de uso. En la tabla 3.1 se resumen las características de ambas variantes.

TensorFlow Lite

TensorFlow Lite es una versión optimizada de TensorFlow que permite ejecutar modelos en dispositivos móviles, sistemas embebidos de gama media y entornos de *edge computing*. Está diseñado para funcionar en plataformas como Android, iOS, Raspberry Pi, y otros dispositivos con sistemas operativos ligeros. Algunas de sus características clave incluyen:

- **Compatibilidad con sistemas de archivos:** TFLite asume la existencia de un sistema de archivos, desde el cual se pueden cargar los modelos en tiempo de ejecución.
- **Soporte para aceleración por hardware:** Puede aprovechar unidades de procesamiento especializadas como GPU y TPU, lo que permite una ejecución más eficiente de modelos complejos.

TensorFlow Lite For Microcontrollers

TensorFlow Lite for Microcontrollers está específicamente diseñado para funcionar en microcontroladores de bajo consumo, que típicamente cuentan con muy poca memoria (en el orden de kilobytes) y sin sistemas operativos ni sistemas de archivos. Por esta razón, presenta un conjunto de características adaptadas a estas restricciones:

- **Ejecución sin sistema operativo:** TFLM no depende de un sistema de archivos ni de un sistema operativo. Los modelos son incluidos directamente en el binario de la aplicación, como arreglos de bytes definidos en código `C`.

- **Uso eficiente de memoria:** Todo el motor de inferencia está diseñado para operar en entornos con recursos muy limitados, reduciendo al mínimo el uso de RAM y ROM.
- **Conjunto limitado de operaciones soportadas:** Debido a las restricciones de hardware, TFLM solo admite un subconjunto de operaciones de TensorFlow, por lo que los modelos deben ser previamente adaptados o entrenados con esta compatibilidad en mente.
- Diseñado para prescindir de dependencias externas, eliminar asignaciones dinámicas y operar con una huella de memoria inferior a 100 KB.

Característica	**TensorFlow Lite**	**TensorFlow Lite for Microcontrollers**
Requiere sistema operativo	Sí (Android, Linux, etc.)	No
Requiere sistema de archivos	Sí	No
Tamaño de memoria requerido	Moderado (MB)	Muy bajo (KB)
Soporte para aceleradores de hardware	Sí (GPU, DSP, TPU)	No
Formato del modelo	FlatBuffer (.tflite)	Arreglo de bytes en código C
Casos de uso típicos	Móviles, Raspberry Pi, edge devices	Sensores, dispositivos IoT, sistemas embebidos de bajo consumo

Tabla 3.1: Comparación entre TensorFlow Lite y TensorFlow Lite for Microcontrollers

3.3.2. Relación del Converter e Interpreter en TensorFlow Lite y TensorFlow Lite for Microcontrollers

Tanto TensorFlow Lite como TensorFlow Lite for Microcontrollers utilizan el **TensorFlow Lite Converter** para transformar modelos entrenados en TensorFlow a un formato optimizado. No obstante, el uso que se hace de los modelos convertidos y la forma en que se interpretan durante la inferencia varía de forma considerable entre ambos entornos.

Uso del Converter

El TensorFlow Lite Converter cumple la misma función en ambos entornos: convierte un modelo de TensorFlow o Keras al formato `.tflite`, utilizando una representación binaria eficiente denominada *FlatBuffer*. En esta etapa también se pueden aplicar optimizaciones como la cuantización. Este paso es común tanto para TFLite como para TFLM, aunque en el caso de microcontroladores, es especialmente importante aplicar técnicas de compresión para reducir el tamaño del modelo.

Uso del Interpreter

La forma en que se utiliza el TensorFlow Lite Interpreter difiere marcadamente entre TFLite y TFLM, debido a las diferencias en capacidades del hardware de destino.

En entornos como Android o Raspberry Pi, el modelo de en TensorFlow Lite `.tflite` se puede cargar directamente desde el sistema de archivos utilizando la clase `Interpreter` dede esta librería en Python.

En TensorFlow Lite for Microcontrollers (TFLM) el modelo no se carga desde un archivo, sino que se incluye directamente como una variable en el código fuente, gracias a herramientas como `xxd` que convierten el archivo `.tflite` a un arreglo de bytes en `C`. El uso del intérprete requiere inicializar manualmente el entorno de ejecución y asignar la memoria necesaria.

A pesar de estas diferencias, se mantiene un paralelismo en la nomenclatura de los métodos utilizados, lo cual facilita la comprensión conceptual del flujo general. En el presente proyecto, se ha optado por utilizar TFLM, dado que su arquitectura está especialmente diseñada para entornos embebidos como el que se propone desarrollar.

3.4. Integración de Edge Impulse

En este proyecto se optó por utilizar la plataforma Edge Impulse, que abstrae y automatiza gran parte de los procedimientos involucrados en la conversión y ejecución de modelos mediante TFLite y TFLM. Edge Impulse proporciona un entorno web integrado que permite cargar y etiquetar datos, entrenar modelos optimizados para dispositivos embebidos, y exportar directamente una librería en `C++` lista para ser compilada en microcontroladores. De esta manera, tareas complejas como la conversión del modelo, la generación del arreglo binario para su inclusión en firmware, y la selección de operaciones compatibles con TFLM son gestionadas de forma transparente para el desarrollador, lo que permite enfocarse en el diseño del sistema y en la calidad de los datos de entrenamiento. En este trabajo, se empleó la arquitectura FOMO (Fast Object Detection for Mobile Devices), especialmente diseñada para detección eficiente en dispositivos con recursos muy limitados. La figura 3.5 ilustra el flujo completo del sistema, desde la adquisición de imágenes, el paso por la red base MobileNet (parte central de la arquitectura FOMO), el proceso de detección de objetos y el postprocesamiento necesario para interpretar las predicciones. Además, se muestran las cajas delimitadoras (bounding boxes) generadas como salida del modelo.

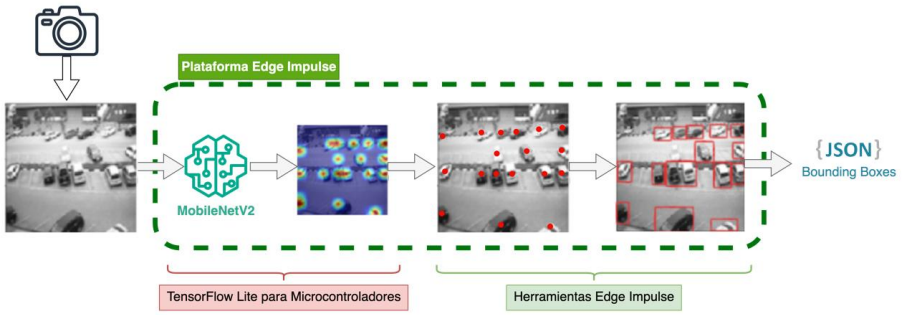


Figura 3.5: Plataforma Edge impulse

3.4.1. Flujo de trabajo con Edge Impulse para dispositivos embebidos

El siguiente paso-a-paso permite desarrollar e implementar modelos de aprendizaje automático en dispositivos embebidos aprovechando las herramientas y recursos que ofrece Edge Impulse.

1. **Registro y creación de proyecto:** Ingresar, registrarse en la plataforma Edge Impulse (`https://studio.edgeimpulse.com/`) y crear un nuevo proyecto, asignándole un nombre representativo, como *Conteo de autos Montevideo*.
2. **Adquisición o importación de datos:** Capturar datos directamente desde sensores conectados o importar conjuntos de datos existentes en formatos compatibles (imágenes JPG/PNG, videos MP4/AVI, audio WAV, archivos CSV). Edge Impulse proporciona herramientas para facilitar esta tarea, incluyendo aplicaciones móviles y soporte para diversas placas de desarrollo.
3. **Etiquetado de datos:** Organizar y etiquetar, es decir, marcar con *bounding boxes* los objetos de interés (en nuestro caso: *auto*). Un etiquetado preciso mejora el entrenamiento efectivo del modelo y permite evaluar sus predicciones con precisión.
4. **Diseño del "impulse":** En la sección *Create impulse*, definir el flujo de procesamiento de datos: agregar bloques de preprocesamiento (por ejemplo, *Image* para imágenes) y seleccionar un bloque de aprendizaje adecuado, como *FOMO* para detección de objetos en dispositivos con recursos limitados.
5. **Entrenamiento del modelo:** Confirgurar los parámetros de entrenamiento (número de épocas, tasa de aprendizaje, etc.) y entrenar el modelo utilizando los datos etiquetados. Edge Impulse ofrece visualizaciones para monitorear el rendimiento durante el entrenamiento.
6. **Evaluación del modelo:** Utilizar la sección *Model testing* para evaluar el rendimiento del modelo con datos de prueba, analizando métricas como precision, recall y matriz de confusión.
7. **Implementación del modelo:** Ir a la sección *Deployment* y seleccionar *C++ library* para generar una biblioteca que incluirá el modelo entrenado y el código necesario para su ejecución en dispositivos embebidos. Se podrá descargar el archivo ZIP generado que contendrá todos los archivos para poder implementar el modelo.
8. **Integración en el dispositivo:** Descomprimir la biblioteca descargada e intégrala en su entorno de desarrollo. Edge Impulse proporciona ejemplos y documentación para facilitar esta integración en diversas plataformas, como Arduino, STM32, Raspberry Pi, y particularmente para nuestro dispositivo ESP32S3.

3.4.2. Arquitectura FOMO: Faster Objects, More Objects

En el proceso de selección de modelos para detección de objetos en dispositivos con recursos limitados, se optó por FOMO (Faster Objects, More Objects), una arquitectura desarrollada por Edge Impulse. FOMO está diseñada específicamente para realizar detección de objetos en tiempo real en microcontroladores, superando las limitaciones de modelos tradicionales como MobileNet-SSD y YOLOv5, que requieren mayor capacidad de procesamiento y memoria.

Características destacadas de FOMO

Eficiencia y velocidad: FOMO es hasta 30 veces más rápida que MobileNet-SSD y puede ejecutarse con menos de 200 KB de RAM. Por ejemplo, en un Arduino Nicla Vision (Cortex-M7 MCU), se han logrado tasas de hasta 30 fps utilizando 245 KB de RAM. [10]

Salida basada en centroides: A diferencia de otras redes neuronales que generan *bounding boxes*, FOMO corta las últimas capas de un modelo de ckasificación

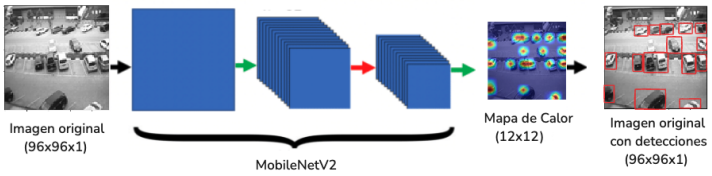


Figura 3.6: Representación entrada-salida de la red.

de imágenes las remplaca con una probabilidad por region de la clase a clasificar, es decir, divide la imagen en regiones o celdas a las que le asigna una probabilidad, produciendo asi un mapa de calor (heatmap) indicando la presencia de esta clase. Aunque no proporciona información sobre el tamaño de los objetos, este enfoque es adecuado para aplicaciones donde la ubicación es más relevante que las dimensiones exactas.

Compatibilidad y despliegue: FOMO es compatible con una amplia gama de placas de desarrollo que incluyen cámaras, y puede desplegarse fácilmente utilizando Edge Impulse Studio. Esto simplifica el flujo de trabajo desde la recopilación de datos hasta el despliegue del modelo en el dispositivo final.

Limitaciones de FOMO

Objetos de tamaño similar: FOMO funciona mejor cuando los objetos a detectar tienen tamaños y formas similares. La detección de objetos de tamaños muy variados puede afectar la precisión del modelo.

Proximidad entre objetos: Si los objetos están muy próximos entre sí, especialmente si sus centroides caen en la misma celda del mapa de salida, FOMO puede fusionarlos en una única detección. Aumentar la resolución de entrada puede mitigar este comportamiento.

Ausencia de información sobre el tamaño del objeto: Al centrarse en la detección de centroides, FOMO no proporciona directamente las dimensiones de los objetos detectados. Sin embargo, es posible estimar *bounding boxes* a partir del mapa de calor generado.

Justificación de la elección

La elección de FOMO se basó en su equilibrio entre eficiencia y precisión para dispositivos con recursos limitados. Su capacidad para realizar detección de objetos en tiempo real con un uso mínimo de memoria y procesamiento lo convierte en una solución ideal para aplicaciones embebidas, como la detección de ocupación de espacios de estacionamiento. Además, la integración con la plataforma Edge Impulse Studio facilita el proceso de entrenamiento y despliegue del modelo, reduciendo significativamente el tiempo de desarrollo. A su vez la plataforma incorpora la etapa que estima las *bounding boxes* a partir del mapa de calor generado por FOMO, lo que permite predecir múltiples objetos en una sola pasada.

3.4.3. Detalles Técnicos del Modelo FOMO

Función de pérdida: Cross-Entropy y Focal Loss

Una de las funciones más utilizadas para entrenar modelos de clasificación y segmentación es la entropía cruzada (*cross-entropy loss*). Esta mide la diferencia entre

la distribución de probabilidad predicha por el modelo y la distribución real. Para un solo ejemplo, se define como:

$$\text{Loss} = -\sum_{i=1}^{C} y_i \cdot \log(\hat{y}_i)$$

donde C es el número de clases, y_i es la etiqueta verdadera (en codificación one-hot) y $\hat{y}_i$ es la probabilidad predicha para la clase i. Esta función penaliza fuertemente las predicciones incorrectas cuando el modelo asigna alta probabilidad a una clase errónea.

Sin embargo, en tareas con alto desbalance de clases —por ejemplo, objetos pequeños en escenas dominadas por fondo— la entropía cruzada puede sesgar el entrenamiento hacia las clases mayoritarias. Para abordar este problema, FAIR (Facebook AI Research) desarrolló la *focal loss*, una extensión ponderada de la entropía cruzada:

$$\text{Focal Loss} = -\alpha_t (1 - \hat{y}_t)^{\gamma} \log(\hat{y}_t)$$

donde:

- α_t es un factor de ponderación para balancear las clases.
- γ es un parámetro que ajusta el enfoque en los ejemplos difíciles.
- $\hat{y}_t$ es la probabilidad predicha para la clase verdadera.

La *focal loss* es especialmente útil en tareas de detección de objetos con muchos negativos y pocos positivos. En este proyecto se implementa una versión simplificada mediante una pérdida de entropía cruzada ponderada según el peso relativo entre las clases “objeto” y “fondo”.

Centroid Scoring

En lugar de evaluar píxel a píxel como en segmentación semántica tradicional, Edge Impulse propone una métrica denominada **CentroidScoring**, diseñada para modelos de detección entrenados mediante segmentación.

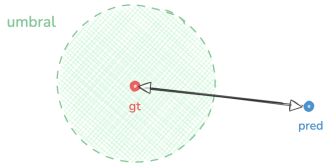


Figura 3.7: Centroides 'ground-truth' y 'pred' junto con el umbral

Para obtener un score a partir de dos centroides se compara la distancia entre el centroide predicho por el modelo y el centroide real (ground truth) de un objeto en una imagen. Esta comparación se realiza utilizando la distancia euclidiana, y se considera una predicción correcta si esta distancia está dentro de un umbral predefinido.

$$d(C_{\text{gt}}, C_{\text{pred}}) = \sqrt{(x_{\text{gt}} - x_{\text{pred}})^2 + (y_{\text{gt}} - y_{\text{pred}})^2}$$

Una predicción se considera correcta si la distancia calculada entre los centroides está por debajo de un umbral establecido. Este umbral puede definirse en función de la resolución de la imagen o del tamaño promedio de los objetos detectados (en nuestro caso, el tamaño promedio de los autos en montevideo).

A partir de la comparación de los centroides, se pueden calcular métricas como *Precisión* (Precision), la proporción de predicciones correctas sobre el total de predicciones realizadas, *Exhaustividad* (Recall), proporción de predicciones correctas sobre el total de objetos reales. *F1-Score*, media armónica entre la precisión y la exhaustividad, proporcionando un balance entre ambas métricas.

La metodología se basa en los siguientes pasos:

1. El modelo produce un mapa de segmentación reducido (por ejemplo, de 12×12 celdas) con probabilidades para cada clase.
2. Se aplica argmax para obtener la clase más probable por celda.
3. Se agrupan las activaciones por clase y se identifican las regiones conectadas en el mapa de calor, conocidas como *blobs*. Un *blob* corresponde a un conjunto de celdas adyacentes que comparten la misma clase y forman una región compacta en el mapa de segmentación.
4. Para cada *blob*, se calcula su centro geométrico (centroide).
5. Se comparan los centroides predichos con los reales (centros de las cajas de anotación), usando un umbral de distancia.
6. A partir de estas coincidencias se derivan métricas como *Precision*, *Recall* y *F1 Score*.

Este enfoque es adecuado para dispositivos embebidos, ya que evita la regresión directa de bounding boxes y permite una detección eficiente a partir de mapas de segmentación.

Cálculo de métricas: Precisión, Recall, F1 Score y Matriz de Confusión

La calidad de las predicciones se evalúa mediante métricas estándar, definidas a partir de las detecciones verdaderas y predichas:

- **True Positive (TP)**: el centroide predicho coincide con un objeto real.
- **False Positive (FP)**: el modelo predice un objeto que no existe.
- **False Negative (FN)**: el modelo no detecta un objeto real.

Con estas cantidades se definen:

$$\text{Precision} = \frac{TP}{TP + FP} \qquad \text{Recall} = \frac{TP}{TP + FN} \qquad \text{F1 Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

3.5. Resultados

El modelo FOMO, basado en MobileNet, fue entrenado con un total de 557 imágenes que contienen 2981 vehículos etiquetados manualmente. Se realizó una partición estratificada del 80 % del conjunto de imágenes para entrenamiento y 20 % para pruebas. En su versión original con precisión `FLOAT32`, el modelo alcanzó una exactitud del $97,12\,\%$ y un F1 Score de $0,96$, demostrando una alta capacidad de detección de vehículos.

Posteriormente, se aplicó una cuantización post-entrenamiento a precisión `INT8` con el fin de facilitar su implementación en hardware embebido. A pesar de la reducción en precisión numérica, el modelo mantuvo un rendimiento competitivo, alcanzando una exactitud del $95,19\,\%$ y un F1 Score de $0,95$.

Tabla 3.2: Comparación de métricas entre modelo FLOAT32 e INT8

Métrica	**Modelo FLOAT32**	**Modelo INT8**
Accuracy	97.12 %	95.19 %
Precision (non-background)	0.98	0.98
Recall (non-background)	0.94	0.93
F1 Score (non-background)	0.96	0.95

Un análisis más detallado del rendimiento del modelo se presenta el capitulo de pruebas y validación 7.

Esta página ha sido intencionalmente dejada en blanco.

Capítulo 4

Desarrollo e Implementación del Firmware

El firmware desarrollado para este proyecto constituye el componente central que permite la interacción entre los diferentes módulos del sistema embebido. Su diseño modular permitió abordar de forma escalonada y progresiva desafíos complejos como la captura de imágenes, la ejecución de modelos de inteligencia artificial, la comunicación inalámbrica mediante BLE y la gestión eficiente de recursos de hardware en un entorno con severas restricciones como se mencionó a lo largo del capitulo anterior.

Este capítulo presenta una visión integral de los aspectos técnicos del firmware, comenzando por su arquitectura general y el flujo de ejecución, seguido por la integración de componentes escenciales como la cámara, el uso de frameworks específicos como TensorFlow Lite Micro, y el diseño del pipeline de inferencia en tiempo real.

4.1. Arquitectura General y Flujo de Ejecución

El firmware fue desarrollado sobre el sistema operativo en tiempo real *FreeRTOS*, ampliamente adoptado en entornos embebidos por su eficiencia y bajo consumo de recursos, además de contar con un sólido soporte comunitario. A diferencia de los sistemas operativos de propósito general, *FreeRTOS* no dispone de un sistema de archivos nativo, por lo que resulta necesario emplear *TensorFlow Lite for Microcontrollers (TFLM)* para la implementación de modelos de IA en el dispositivo embebido.

La integración con *ESP-IDF* permite estructurar el proyecto de forma modular, siguiendo un patrón similar al de los proyectos basados en Arduino, con funciones `setup()` y `loop()`, aunque en este caso la función `loop()` permanece vacía. La concurrencia se gestiona mediante tareas e interrupciones (*ISR*), lo que permite una operación no bloqueante y una respuesta eficiente frente a eventos del sistema.

La arquitectura se ejecuta sobre el microcontrolador ESP32-S3, cuyas capacidades fueron analizadas en el capítulo 2. Si bien el desarrollo del Gateway también puede considerarse parte del firmware, su implementación se detalla por separado en el capítulo 5, con el objetivo de facilitar su comprensión y mantener una separación lógica entre ambos componentes.

Las Figura 4.1 y 4.2 ilustran de manera esquemática la arquitectura general del firmware desarrollado, resaltando las etapas clave de su ejecución. Esta se organiza en dos fases fundamentales: la fase de **inicialización** (*setup*) y la fase de **ejecución continua** mediante “tareas virtuales”.

Las tareas virtuales pueden entenderse, en términos simples, como "hilos ligeros" que se ejecutan de forma concurrente dentro del microcontrolador. Se las denomina virtuales porque no están directamente asociadas a núcleos físicos (como sucede en sistemas multinúcleo), sino que su ejecución es gestionada y alternada por el kernel del sistema operativo en tiempo real (RTOS). En la práctica, el RTOS simula paralelismo mediante planificación por interrupciones y conmutación de contexto, incluso en dispositivos con un único núcleo. En caso de que el hardware disponga de múltiples núcleos, el RTOS puede distribuir ciertas tareas entre ellos, pero este comportamiento depende de la configuración específica del sistema y del soporte del framework utilizado. Gracias a este enfoque, es posible distribuir las diferentes tareas de módulos como el manejo de sensores, la comunicación inalámbrica y el procesamiento de inferencias.

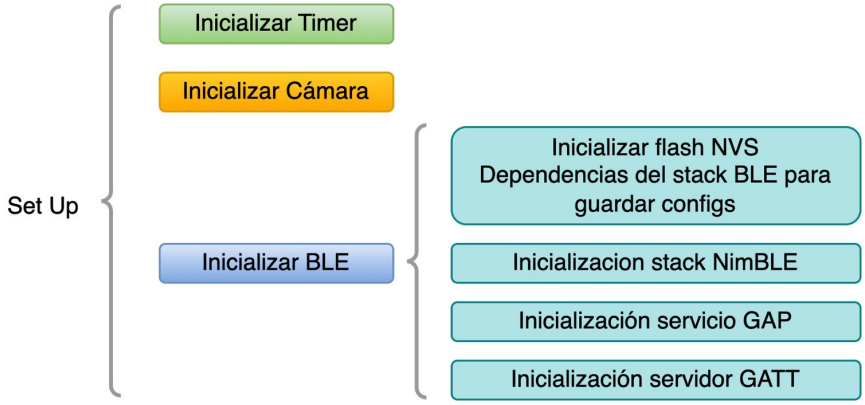


Figura 4.1: Diagrama de bloques del pipeline del firmware: etapa set-up.

Durante la fase de *setup*, el firmware se encarga de la configuración inicial de los recursos a utilizar dentro del sistema. En particular, se realiza la inicialización del temporizador principal, la cámara (modelo OV2640) y el subsistema de comunicaciones BLE. Esta última implica, además, la configuración de la memoria flash NVS (Non-Volatile Storage) para almacenar parámetros, la carga del stack NimBLE (una implementación ligera de BLE) y la definición de los servicios llamados GAP (acceso general) y GATT (transferencia de atributos), fundamentales para el intercambio de datos inalámbricos.

Completada esta etapa, el sistema procede a la creación de tareas virtuales que operan en paralelo dentro de un ciclo infinito. Este diseño permite atender distintos procesos simultáneamente, garantizando una respuesta eficiente ante eventos del entorno y un procesamiento continuo de imágenes.

Se aprecia en la figura 4.2 la creación de tres tareas virtuales: Vision Task, NimBLE Host Task, y Restart System Timer Task. Donde se destaca el flujo de ejecución de la tarea Vision Task, que ejecuta de forma iterativa el pipeline de procesamiento visual.

Este comienza con la captura de una imagen mediante la cámara OV2640 integrada, seguida por el encendido de un LED indicador que señala la operación activa. A continuación, el sistema verifica si existen *suscriptores activos*, es decir, dispositivos que hayan solicitado recibir datos visuales en tiempo real.

Si hay al menos un suscriptor conectado, la imagen se convierte al formato RGB888 (una codificación de 24 bits compatible con el modelo de inferencia), se procesa utilizando el clasificador entrenado en Edge Impulse, y el resultado se transforma en un mensaje estructurado en formato JSON. Dicho mensaje organiza las predicciones

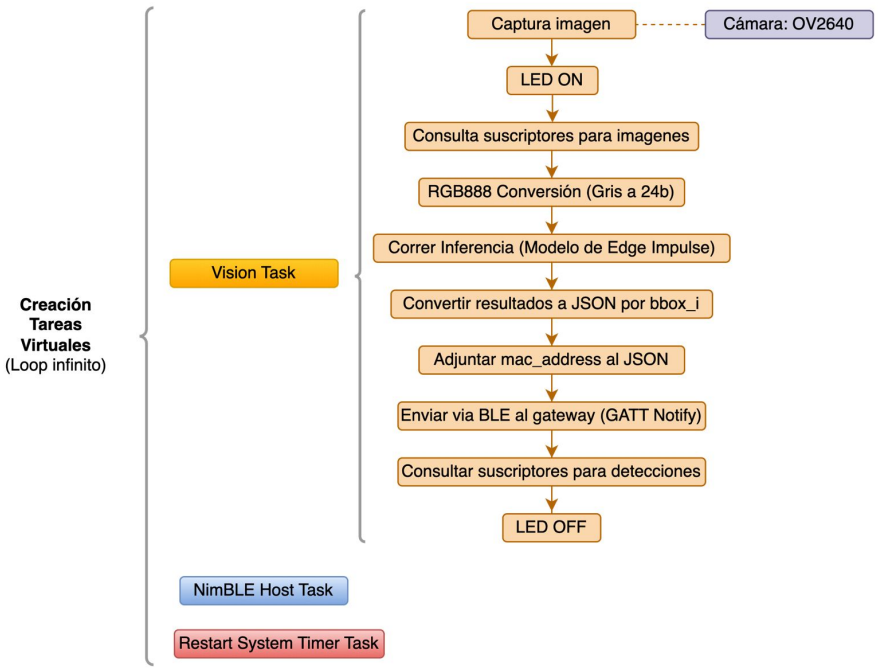


Figura 4.2: Diagrama de bloques del pipeline del firmware: etapa creación de tareas virtuales.

en una lista donde cada elemento representa una caja delimitadora (bounding box) asociada a una detección específica. Finalmente, se transmite el resultado vía BLE mediante notificaciones GATT.

Como se detalla en el capítulo 2, el sistema Parkwise opera en dos modos principales: *setup* y *live inferencing*. Esta distinción tiene implicancias directas en el comportamiento del firmware: durante el modo *setup*, el dispositivo responde a solicitudes de imágenes completas que serán utilizadas para el proceso de calibración visual; en cambio, en el modo *live inferencing*, las solicitudes están orientadas a la transmisión periódica de bounding boxes detectadas localmente. El presente capítulo se enfoca en describir el funcionamiento interno del firmware bajo ambas modalidades, con especial énfasis en el pipeline de inferencia y la gestión de tareas concurrentes mediante FreeRTOS.

4.2. Diseño de Tareas

Se entiende por tarea virtual a una unidad lógica de ejecución dentro del RTOS, que representa un flujo independiente de operaciones asignadas a un subsistema específico. Aunque no necesariamente corresponde a un hilo físico, cada tarea virtual opera de forma concurrente con las demás, facilitando la organización modular del *firmware* y permitiendo que las tareas operen sin bloqueos.

Tareas virtuales en el diseño del firmware

Las tareas descritas en la figura 4.2, fueron diseñadas con responsabilidades específicas, priorizando la separación de funcionalidades críticas y la estabilidad en la

ejecución del sistema embebido.

- **Vision Task:** Esta es la tarea principal del sistema y opera en bucle infinito. Su función central es capturar imágenes desde la cámara OV2640, realizar la conversión a formato RGB888, ejecutar la inferencia sobre el modelo de Edge Impulse, y gestionar la serialización de resultados en formato JSON. También se encarga de consultar los suscriptores activos y transmitir los datos relevantes vía BLE. Esta tarea fue diseñada para ser autosuficiente, ejecutándose de manera continua independientemente del estado de conexión externa.
- **NimBLE Host Task:** Esta tarea gestiona el stack de comunicación Bluetooth de bajo consumo (BLE), utilizando la pila NimBLE integrada en el framework ESP-IDF. Es responsable del mantenimiento del enlace con los dispositivos suscriptores, la gestión de eventos GAP (Generic Access Profile) y GATT (Generic Attribute Profile), y el procesamiento de interrupciones vinculadas al protocolo inalámbrico. Opera de forma paralela a la Vision Task, asegurando una respuesta inmediata ante nuevas conexiones o eventos de suscripción.
- **Reset Timer Task:** Esta tarea implementa un sistema de reinicio periódico supervisado, actuando como una suerte de watchdog personalizado. Su propósito es garantizar la recuperación automática del sistema ante condiciones de fallo silencioso o bloqueo parcial. Reinicia el ciclo de inferencia tras un tiempo predefinido de inactividad o tras finalizar un proceso completo, manteniendo así el flujo del sistema operativo en tiempo real.

La definición explícita de estas tareas permitió aprovechar al máximo la arquitectura del sistema operativo de tiempo real, favoreciendo la concurrencia real entre captura de datos, procesamiento y comunicación inalámbrica. Además, esta organización modular facilita el mantenimiento y escalabilidad del firmware, permitiendo futuras ampliaciones como el manejo de nuevos periféricos, mayor variedad de modelos o la introducción de algoritmos de control en paralelo.

4.3. Ejecución del Modelo de Inferencia

Uno de los componentes centrales del sistema embebido desarrollado es el modelo de inferencia basado en aprendizaje automático. Esta sección describe cómo dicho modelo es implementado, ejecutado y monitoreado en tiempo real dentro del microcontrolador ESP32-S3, integrando el firmware, el pipeline de procesamiento y los módulos de comunicación inalámbrica.

El flujo de inferencia comienza con la captura de una imagen, su preprocesamiento (recorte, reducción de resolución, conversión a escala de grises y normalización), y continúa con la ejecución del modelo utilizando una implementación optimizada de TensorFlow Lite Micro (TFLM). Posteriormente, las salidas del modelo —un mapa de activaciones o *heatmap*— son convertidas en *bounding boxes* con información sobre clase, coordenadas y tiempo de inferencia. Estas detecciones son luego empaquetadas en estructuras JSON para su transmisión.

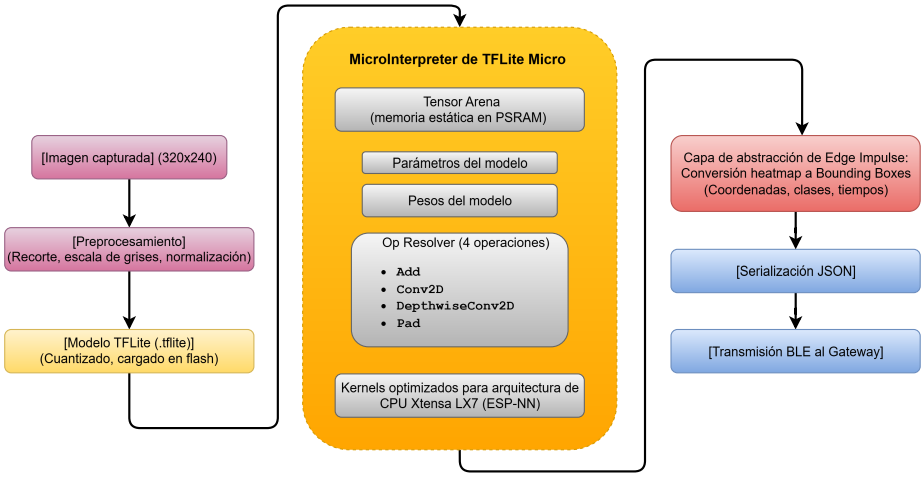


Figura 4.3: Pipeline de inferencia embebido con TFLite Micro: captura, ejecución y transmisión de resultados optimizados.

Cuando no hay suscriptores conectados, el *firmware* continúa ejecutando el pipeline de inferencia localmente, procesando imágenes de forma continua y desechando los resultados.

En cambio, cuando se detecta al menos un suscriptor activo, se habilita la transmisión de resultados a través de BLE. En el caso de imágenes, esta transmisión incluye un mecanismo de fragmentación para dividir la imagen en paquetes adecuados al protocolo. Los detalles técnicos de este mecanismo se desarrollarán en capítulos posteriores.

El ciclo completo —desde la captura hasta la transmisión— está ilustrado en la Figura 4.3. Este diagrama destaca los componentes clave del pipeline embebido y permite visualizar el flujo lógico de datos en el sistema.

Este flujo comienza con la captura de imágenes por parte del periférico de cámara, seguido de una serie de transformaciones que permiten adaptar los datos de entrada al formato requerido por el modelo de detección. A diferencia del procesamiento convencional en plataformas de alto rendimiento, todo el pipeline mostrado ha sido adaptado para operar en un entorno donde el sistema operativo no tiene un sistema de archivos, con recursos de memoria limitados y capacidad de cómputo restringida.

El bloque izquierdo representa las etapas iniciales de captura y preprocesamiento de la imagen. El formato nativo (320×240 píxeles en color) se reduce y transforma

mediante operaciones de recorte, conversión a escala de grises y normalización. Esta transformación busca adaptar la imagen para enviársela al modelo.

El bloque central muestra el núcleo de ejecución del modelo: el TFLite MicroInterpreter. El interpreter es el responsable de orquestar la inferencia sobre el microcontrolador reclutando todos los recursos que hagan falta. Este intérprete se configura durante el *setup* y mantiene en memoria los siguientes elementos que va a orquestar (hacer uso):

- Una región preasignada de memoria estática denominada Tensor Arena, alojada en PSRAM para maximizar el espacio disponible sin interferir con la RAM interna.
- Los pesos del modelo y sus parámetros en formato FlatBuffer, cargados desde la Flash al iniciar el sistema.
- Una estructura denominada Op Resolver (resolvedor de operaciones), que se encarga de mapear únicamente las operaciones necesarias durante la inferencia. Esta limitación tiene motivaciones de eficiencia: incluir más operaciones aumentaría el binario (código compilado en flash) y el tiempo de carga.

Desde el punto de vista del firmware, es clave resaltar que cada una de estas operaciones se ejecuta mediante funciones llamadas "kernel functions". En su forma más simple, un kernel (núcleo por su denominación en inglés) es la parte fundamental de un sistema operativo que gestiona los recursos del hardware. Por lo que los roles de estas funciones incluyen operaciones matemáticas concretas que manipulan tensores (por ejemplo, aplicar una convolución o sumar matrices). Sin embargo, para lograr tiempos de inferencia aceptables, estas *kernel functions* deben estar optimizados para la arquitectura del microcontrolador.

En nuestro caso, al contar con la ventaja de compilar el proyecto con soporte para la biblioteca ESP-NN , los kernels estándar fueron reemplazados automáticamente por versiones optimizadas para la arquitectura Xtensa LX7 del ESP32-S3. Estos kernels escritos en bajo nivel permiten aprovechar instrucciones específicas de multiplicación y acumulación, pudiendo lograr una reducción de tiempo de inferencia hasta 10 veces respecto a la implementación genérica (ver pruebas realizadas en [38]). Esta sustitución es transparente para el desarrollador, pero debe estar habilitada explícitamente en el entorno de compilación.

Una vez ejecutado el modelo, su salida consiste en un mapa de activación (heatmap) que representa las regiones donde es probable que haya un objeto detectado. En esta instancia interviene una capa de abstracción propia del modelo generado por Edge Impulse, que traduce este heatmap en una serie de *bounding boxes* etiquetadas con clase, coordenadas y tiempo de inferencia. Este paso también se ejecuta dentro del firmware y se realiza en RAM, antes de la serialización de los resultados.

Finalmente, los objetos detectados se empaquetan en estructuras JSON, y se elevan al servidor GATT copiándolos por parámetro, de forma que puedan ser transmitidos al Gateway, completando así el ciclo de inferencia y transmisión de datos del sistema.

Este diagrama no solo resume el flujo lógico de la inferencia en el dispositivo, sino que también evidencia el grado de adaptación necesario a nivel de firmware para que todo el proceso funcione en tiempo real sobre una plataforma con recursos limitados. La elección cuidadosa de operaciones, la gestión estática de memoria y el uso de kernels optimizados fueron condiciones necesarias para alcanzar el desempeño observado.

4.4. Diseño e Integración del Modelo de Inferencia

Esta sección introduce el proceso integral que permitió incorporar capacidades de visión por computadora al sistema embebido, centrándose en el diseño, adaptación y

despliegue de un modelo de inferencia basado en redes neuronales. Lejos de tratarse de una tarea aislada, la integración del modelo de detección supuso abordar múltiples dimensiones técnicas, desde la elección de frameworks adecuados hasta la adaptación a las severas restricciones de memoria y procesamiento del microcontrolador utilizado.

El enfoque adoptado fue progresivo y exploratorio. Se partió de una base teórica y experimental provista por el ecosistema TinyML y TensorFlow Lite Micro, que se estudiaron a partir de ejemplos y casos de uso reales. Posteriormente, se exploraron diferentes rutas de implementación, incluyendo el uso de modelos de referencia, el desarrollo de arquitecturas propias y el diseño de un pipeline autónomo embebido capaz de ejecutar inferencias en tiempo real.

Cada una de estas etapas será abordada en profundidad a lo largo de las siguientes tres subsecciones, que recorren el trayecto desde la formación conceptual inicial hasta la integración final del modelo en el firmware. Este recorrido no sólo describe las decisiones técnicas adoptadas, sino también las dificultades encontradas, los aprendizajes extraídos y las soluciones desarrolladas, sentando las bases para una arquitectura de inferencia robusta y reutilizable.

4.4.1. Fundamentos Aplicados de TinyML en el Firmware

Nuestro acercamiento inicial al desarrollo del firmware estuvo fuertemente influenciado por el libro *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*, recomendado por los tutores desde el comienzo del proyecto. Esta obra fue clave para introducirnos al mundo del Edge AI, ayudándonos a entender las limitaciones de los sistemas embebidos y brindándonos una guía práctica para diseñar modelos funcionales en dispositivos de bajos recursos.

A partir de esta lectura se trazó un recorrido progresivo que constituyó la base conceptual y técnica de nuestro trabajo. Comenzamos con ejemplos introductorios, como la predicción de funciones seno, que nos permitieron familiarizarnos con la arquitectura de TensorFlow Lite Micro y verificar el funcionamiento completo de un modelo cuantizado corriendo en un microcontrolador. Ese caso simple encapsulaba todos los elementos clave: entrenamiento en Python, conversión del modelo a formato `.tflite`, y ejecución embebida mediante C++.

Este punto de partida nos llevó a explorar el SDK de TensorFlow Lite Micro, familiarizándonos con su arquitectura modular. En particular, destacamos:

- La separación entre entrenamiento (Python) e inferencia (C++), que nos permitió trabajar de forma paralela sobre el modelo y el firmware.
- La conversión de modelos al formato FlatBuffer (`.tflite`), optimizado para bajo consumo de memoria y fácil parsing embebido.
- La implementación de inferencia en C++ usando el *microinterpreter* de TFLite Micro, capaz de operar sin sistema operativo.
- La asignación de memoria estática a través de la *Tensor Arena*, una técnica crítica para evitar fallos por heap fragmentation.
- La ejecución eficiente de modelos cuantizados (8/16-bit), indispensable para plataformas sin FPU como el ESP32-S3.

Este conocimiento fue validado en hardware real usando ejemplos incluidos en el repositorio oficial de TensorFlow Lite. A partir de ahí, dimos el siguiente paso: migrar el entorno de pruebas de Arduino al framework ESP-IDF, mucho más cercano al entorno productivo del proyecto. Esta transición nos permitió ganar control total sobre periféricos y memoria, sentando así las bases para una integración más profunda.

Finalmente, llegamos al repositorio `esp-tflite-micro` de Espressif, donde encontramos modelos como *Person Detection* ya adaptados a la arquitectura del ESP32-S3.

Este ejemplo no solo funcionaba correctamente con nuestra cámara, sino que además incluía optimizaciones importantes, como kernels acelerados (ESP-NN) y soporte para PSRAM.

En resumen, la exploración de TinyML no fue solo una etapa formativa, sino también un proceso de construcción progresiva del firmware. Lo aprendido se refleja en decisiones clave que guiaron el diseño del sistema tal como se describe en las siguientes secciones.

Así, si bien el libro fue una referencia inicial, el enfoque adoptado consistió en transformar esas bases en un sistema robusto y funcional adaptado a nuestro caso de uso.

4.4.2. Desarrollo de Modelos Propios y Limitaciones

Desde las etapas iniciales de esta investigación se consideró a Edge Impulse como una plataforma potencial para el desarrollo de modelos de Aprendizaje Automático en Sistemas Embebidos. Sin embargo, inicialmente se realizó un estudio exploratorio de sus capacidades sin profundizar en su implementación, priorizando primero comprender los fundamentos del flujo de trabajo para estos modelos.

A partir de los ejemplos analizados en la plataforma, se comprendió el flujo de trabajo con el que se implementan estos modelos. No obstante, subestimando inicialmente el valor agregado de Edge Impulse en cuanto a optimizaciones preimplementadas, se decidió emprender el desarrollo de modelos independientes sin utilizar sus dependencias.

Se realizaron pruebas con dos arquitecturas principales:

- **MobileNetV2 modificada**: En primer lugar, se implementó una versión modificada de MobileNetV2, utilizando su estructura base preentrenada (comúnmente denominada "backbone" en la terminología de redes neuronales, que se refiere a la parte fundamental de la red responsable de la extracción de características). Esta arquitectura fue adaptada para generar regiones de interés específicas para la detección vehicular. Sin embargo, este enfoque presentó limitaciones significativas, incluyendo requerimientos de memoria del orden de 3MB que excedían la capacidad disponible, baja precisión en los resultados y *bounding boxes* con niveles inaceptables de inconsistencia visual.

- **YOLO Nano**: Como alternativa, se exploró el uso de YOLO Nano, encontrando obstáculos técnicos sustanciales durante el proceso de conversión al ecosistema TensorFlow Lite. El flujo de conversión requerido (desde PyTorch al formato ONNX, posteriormente a TFLite y finalmente a TFLite Micro) demostró ser particularmente complejo, principalmente debido a la presencia de operaciones no soportadas en TFLite Micro, como la operación de concatenación, y a la falta de documentación disponible para resolver estas incompatibilidades específicas.

Estos experimentos revelaron desafíos técnicos no anticipados en tres dimensiones críticas: la necesidad de optimización extrema para cumplir con los requisitos de memoria de los sistemas embebidos (inferiores a 1MB), las limitaciones inherentes de TFLite Micro respecto al conjunto de operaciones soportadas, y la dificultad de alcanzar precisiones aceptables con arquitecturas estándar sin las optimizaciones específicas que proveen plataformas como Edge Impulse.

Como conclusión, esta fase del proyecto demostró empíricamente el valor diferencial de soluciones integradas, particularmente en lo que respecta a: (1) optimizaciones específicas para microcontroladores, ejemplificadas en implementaciones como FOMO que logran versiones altamente eficientes de arquitecturas convencionales; (2) manejo

transparente de las limitaciones de TFLite Micro; y (3) automatización completa del flujo desde el entrenamiento hasta el despliegue final.

Aunque los resultados no fueron los esperados, este proceso resultó inevitable para comprender en profundidad los desafíos inherentes al edge AI, proporcionando las bases que posteriormente guiaron la selección de herramientas para la implementación final del sistema. Subrayando la importancia de evaluar cuidadosamente el balance entre desarrollo independiente y el uso de plataformas especializadas.

4.5. Diseño del Pipeline de Inferencia Autónomo

En paralelo a las pruebas de arquitecturas descritas en la sección anterior, se desarrolló un pipeline de implementación en firmware que combinaba los aprendizajes obtenidos del modelo *Person Detection* con los conceptos fundamentales extraídos del análisis de Edge Impulse. Aunque no se logró integrar completamente un modelo funcional, se establecieron las bases necesarias para que el sistema estuviera preparado para ejecutar inferencias una vez se dispusiera del binario de la red neuronal adecuado.

El proceso de implementación que se desarrolló siguió una secuencia lógica de las siguientes etapas críticas:

1. **Carga de pesos del modelo**: Se implementó el mecanismo para mapear los pesos del modelo preentrenado en una estructura de datos utilizable. Este proceso, que opera a nivel de memoria, no involucra operaciones costosas de copiado o parsing, manteniendo una huella computacional mínima. Se incluyeron verificaciones de compatibilidad de versiones del esquema TFLite para garantizar la interoperabilidad.
2. **Asignación de memoria para tensores**: Se diseñó un sistema dinámico de gestión de memoria para los *buffers* de entrada, salida y los tensores intermedios (estructuras que generalizan el concepto de vector o matriz a múltiples dimensiones, utilizadas en las distintas etapas del modelo). Este sistema aprovechaba la PSRAM del ESP32-S3 cuando se encontraba disponible e incluía verificaciones exhaustivas de disponibilidad de memoria, así como mecanismos de fallo controlado.
3. **Configuración del resolver de operaciones**: En lugar de utilizar el `AllOpsResolver` (que incluye todas las operaciones posibles con el consiguiente costo en espacio de código), se implementó un resolver específico que solo incorporaba las operaciones estrictamente necesarias (Quantize, Conv2D, DepthwiseConv2D, Pad, Dequantize y Add), optimizando así el uso de recursos.
4. **Configuración del intérprete**: Se configuró el microinterprete de TFLite como orquestador principal, encargado de gestionar el flujo de ejecución del modelo. Este componente se encarga de asignar los tensores y preparar la estructura de ejecución para las inferencias.
5. **Integración con periféricos**: Se implementó la inicialización controlada de la cámara OV2640, incluyendo manejo de errores y verificación de estados. Garantiza que las imágenes de entrada cumplan con los requisitos de formato y resolución del modelo.
6. **Configuración de la comunicación BLE**: Se establecieron las bases del subsistema de comunicación inalámbrica mediante el protocolo BLE, incluyendo la inicialización del stack NimBLE, los servicios GAP y GATT, y el manejo de la memoria NVS para almacenamiento de configuraciones. Esta fue una instancia de primer contacto con las comunicaciones inalámbricas, se ahondara en su desarrollo a lo largo de las próximas secciones.

7. **Tareas Virtuales**: Se sentaron las bases para el diseño y coordinación de tareas virtuales (también conocidas como *FreeRTOS tasks*). Estas tareas permiten dividir responsabilidades lógicas —como la captura de imágenes, la inferencia del modelo, la transmisión de resultados o la actualización de temporizadores— en unidades independientes de ejecución concurrente. En esta primera etapa, se diseñó una arquitectura basada en la creación explícita de tareas tras la fase de `setup`, asegurando que cada hilo de ejecución contara con su propio ciclo de vida, prioridad y stack de memoria.

Cabe destacar que este desarrollo representó un desafío técnico significativo, ya que requirió implementar y comprender en profundidad el funcionamiento interno de TFLite Micro para sistemas embebidos, mientras posteriormente y simultáneamente al diseño de arquitectura (MobileNetV2 y YOLO nano) se avanzaba en la integración de los demás componentes del sistema.

La implementación en firmware resultante demostró ser robusta y preparada para la incorporación del modelo final, validando el enfoque de desarrollo modular adoptado.

4.6. Retorno a Edge Impulse: Optimización y Resultados

Tras la fase de exploración e integración manual del modelo de inferencia detallada en las secciones anteriores, se reconsideró el uso de la plataforma Edge Impulse. Si bien esta decisión implicaba retroceder parcialmente en el enfoque autónomo que se venía desarrollando, permitió capitalizar el conocimiento adquirido sobre la estructura interna de TensorFlow Lite Micro y, a su vez, explotar las ventajas de una herramienta específicamente diseñada para el despliegue de modelos en sistemas embebidos.

La reintroducción de Edge Impulse no fue un retorno ingenuo, sino un movimiento técnicamente fundamentado. La experiencia previa permitió comprender en profundidad cada una de las abstracciones ofrecidas por la plataforma, dotando al equipo de un marco conceptual claro sobre su funcionamiento interno. Esta nueva etapa comenzó con la identificación de un modelo preentrenado de detección vehicular (*Car Detection*), que funcionaba de forma similar al conocido *Person Detection*, pero con una salida ajustada a los objetivos específicos del sistema: generar coordenadas de *bounding boxes* a partir de imágenes cargadas de forma estática.

La validación inicial de este modelo se realizó con imágenes manuales, obteniendo resultados prometedores. A partir de allí, se inició una nueva fase de integración progresiva, recuperando y adaptando funcionalidades previamente desarrolladas en el firmware:

- Se implementó el proveedor de la cámara, habilitando a la captura y almacenamiento de imágenes. Pudiendo así conectarlo con el modelo de inferencia, sustituyendo las features estáticas que tomaba como entrada.
- Se reimplementó el servidor BLE utilizando la pila NimBLE, configurando desde cero los servicios GATT y sus características asociadas para la transmisión de imágenes y detecciones.
- Se elaboraron las rutinas de fragmentación y envío de imágenes en paquetes, optimizando tanto el uso de memoria como la estabilidad en conexiones prolongadas. Esto representó una funcionalidad de complejidad significativa.
- Se adaptaron los esquemas de serialización de los resultados inferidos (coordenadas, clases y tiempos de inferencia) al formato JSON planteado y esperado por el *Gateway*, asegurando compatibilidad con los flujos de datos que posteriormente serían reenviados al servidor central vía internet.

Posteriormente, se avanzó sobre aspectos de robustez y usabilidad del sistema, introduciendo mejoras que elevaron el grado de autonomía del dispositivo. Entre ellas se destacan:

- Un mecanismo de *hard reset* desencadenable de forma remota vía BLE, útil para restaurar el estado del sistema en caso de fallos persistentes.
- Un *timer* de reinicio automático —funcionalmente equivalente a un *watchdog*—, que aseguraba la recuperación ante bloqueos inesperados.
- Mejoras en el control de la cámara embebida, permitiendo modificar parámetros de captura, logrando probar diferentes configuraciones que se adapten mejor en función de mejorar el restulado.

Finalmente, con el pipeline de inferencia estable, se procedió a optimizar el modelo cargado mediante un nuevo proceso de entrenamiento con un dataset mas grande, mayor resolución de entrada y configuraciones adaptadas a las restricciones del microcontrolador. Las pruebas de funcionamiento confirmaron no solo la compatibilidad con el *Gateway* de comunicaciones, sino también la correcta recepción y reenvío de datos al servidor central.

Este retorno a Edge Impulse, lejos de ser un retroceso, representó una consolidación del proyecto, donde la experiencia adquirida en el desarrollo de firmware personalizado permitió aprovechar la plataforma con un criterio técnico más afinado. El resultado fue un sistema funcional, robusto y validado, capaz de ejecutar inferencias en tiempo real y transmitir los resultados bajo un esquema inalámbrico eficiente.

4.7. Integración de la Cámara

Tal como se introdujo en la Sección 4.1, la cámara OV2640 constituye un componente esencial para el funcionamiento del sistema de visión embebida. Su integración permitió capturar imágenes de baja resolución adaptadas a los requerimientos del modelo de inferencia y validar el correcto funcionamiento del pipeline. La Figura 4.4 muestra un ejemplo real de captura, en el que se observan automóviles detectados como objetos de interés mediante el modelo de visión artificial.

Figura 4.4: Ejemplo de captura de imagen 320x240 realizada por la cámara embebida. Se observa la presencia de automóviles como objetivo del sistema.

El acceso a la cámara fue un hito clave en los comienzos de la implementación del firmware, ya que permitió realizar pruebas preliminares para validar el funcionamiento del hardware y el framework, y verificar la viabilidad de usar una cámara embebida. Este paso fue fundamental para asegurar que la cámara pudiera capturar imágenes, necesarias para el sistema de visión por computadora y la transmisión inalámbrica para depuración y validación.

4.7.1. Desafíos Técnicos y Soluciones Implementadas

La integración de la cámara representó uno de los primeros grandes desafíos del proyecto. Inicialmente, no se logró ejecutar con éxito ningún ejemplo funcional en ESP-IDF ni en frameworks alternativos. La primera validación efectiva se obtuvo gracias a un ejemplo de Arduino desarrolldo por "Mjrovai" [33], el cual permitió verificar el correcto funcionamiento físico del módulo OV2640.

Posteriormente, se migró la implementación al entorno ESP-IDF, lo cual implicó una curva de aprendizaje considerable. A diferencia de Arduino, que provee abstracciones de alto nivel, además configurar de forma explícita los pines del periférico en ESP-IDF fue necesario activar el uso de PSRAM desde el archivo de configuración `sdkconfig`, e inicializar el controlador de cámara a bajo nivel.

Este proceso supuso además una introducción práctica al sistema de configuración de ESP-IDF, incluyendo el uso del menú interactivo de compilación y la comprensión de parámetros importantes como el tamaño del búfer de captura, el formato de salida y la asignación de DMA para transmisión eficiente.

4.7.2. Importancia de Librerías Preexistentes

La elección del framework ESP-IDF resultó decisiva. Si bien se exploraron alternativas como Zephyr RTOS, su falta de soporte nativo para la OV2640 descartó su utilización en esta etapa. Arduino, por su parte, permitió una rápida puesta en marcha, pero su escasa capacidad de configuración lo hizo inadecuado para los requerimientos del proyecto.

ESP-IDF proporcionó bibliotecas optimizadas y estables para el manejo de periféricos como la cámara, incluyendo control sobre el reloj del sensor, buffers de lectura, configuración de frame size y operaciones DMA. Estas librerías permitieron alcanzar un mayor control sobre el flujo de datos y la gestión de memoria, esenciales para el funcionamiento continuo del sistema bajo un sistema operativo en tiempo real (RTOS).

4.7.3. Conversión RGB888 y Configuración Final

En las versiones finales del sistema, se optó por capturar imágenes en escala de grises (1 canal), lo que facilitaba su transmisión vía BLE sin conversiones intermedias. Sin embargo, el modelo de inferencia desarrollado en Edge Impulse requiere imágenes en formato RGB888 (3 canales de 8 bits), sin posibilidad de modificar el formato de entrada del modelo desde la plataforma.

Para resolver esta incompatibilidad, se implementó una función de conversión directa de escala de grises a RGB888. Esta operación replica el valor de luminancia de una imagen blanco y negro en los tres canales de color mediante una operación lógica por píxel.

Esta sección concluye la descripción del proceso de integración de la cámara embebida, remarcando la importancia de los recursos disponibles en el framework, la validación paso a paso de los datos capturados y las adaptaciones necesarias para compatibilizar el flujo de datos con los requisitos del modelo de inferencia utilizado.

Capítulo 5

Comunicación inalámbrica

5.1. Introducción General

El correcto funcionamiento de la solución depende de un conjunto de comunicaciones inalámbricas, que permiten enviar eventos detectados en los Dispositivos Parkwise hacia un Servidor Central, con una arquitectura escalable. Además, estas comunicaciones permiten llevar adelante tareas de configuración remota por parte de instaladores u operadores autorizados.

El flujo de información del sistema se muestra en la figura 5.1. Este parte desde un Dispositivo Parkwise (basado en ESP32-S3), continúa hacia un Gateway ubicado físicamente en cercanía —por ejemplo, en el alumbrado público que alimenta al dispositivo— y de allí hacia un broker MQTT en la nube. Finalmente, la información es recibida y procesada por un servidor central.

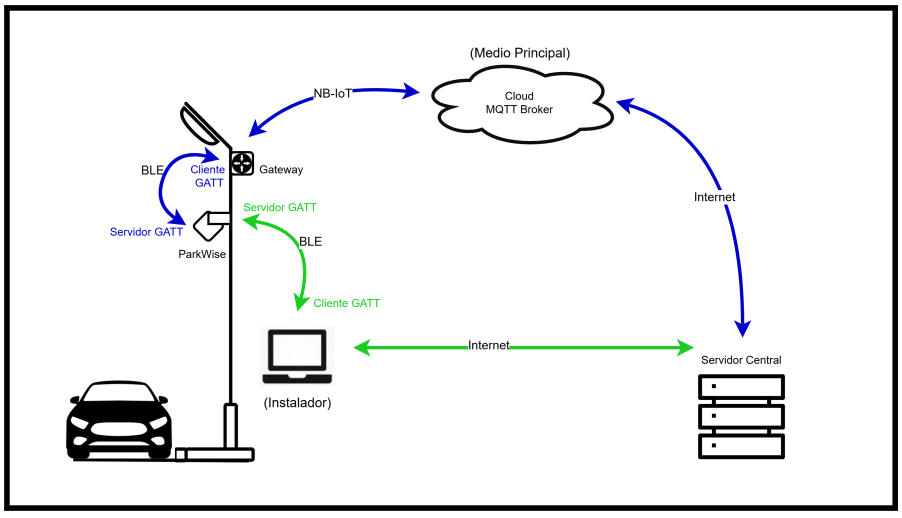


Figura 5.1: Mapa general del flujo de datos. Se muestran los vínculos inalámbricos entre el nodo embebido (Dispositivo Parkwise), el Gateway y el Servidor Central.

El Dispositivo Parkwise actúa como servidor GATT, mientras que los dispositivos

que lo interrogan (el Gateway o el instalador) funcionan como clientes GATT. Los mensajes transmitidos pueden incluir imágenes, detecciones, estados del sistema y comandos de reinicio.

El sistema contempla dos vínculos principales:

- **Vínculo de operación normal:** conecta Parkwise con el Gateway mediante BLE (protocolo NimBLE), y desde el Gateway al broker MQTT a través de NB-IoT.
- **Vínculo de configuración:** permite al instalador conectarse localmente a Parkwise por BLE para configurar parámetros iniciales. Esta conexión se realiza desde un computador móvil o PC actuando como cliente GATT.

El Gateway actúa como traductor de protocolos entre GATT (sobre BLE) y el MQTT (sobre NB-IoT). Para ello, se utiliza un esquema basado en tópicos, donde las características del protocolo GATT se mapean a mensajes JSON publicados bajo distintos tópicos MQTT. Esta decisión de diseño permite una integración sencilla y escalable con el servidor central, que actúa como cliente suscriptor de estos mensajes.

Se contempló además un vínculo mediante BLE entre Parkwise y un dispositivo del operario. Esta vía permite a un operario instalar y poner en marcha el dispositivo o realizar pruebas de funcionamiento cuando lo considere oportuno.

Tecnologías en el sistema

Cada una de las tecnologías empleadas responde a una necesidad específica del sistema y a las comunicaciones del hardware elegido:

- **BLE con NimBLE:** protocolo de baja energía (Low Energy) ideal para comunicaciones de corto alcance. La implementación NimBLE, utilizada sobre ESP-IDF, ofrece una serie de herramientas robusta para el Dispositivo Parkwise con el Gateway.
- **NB-IoT:** red celular de bajo consumo y largo alcance, ideal para entornos urbanos con poca infraestructura de red local.
- **MQTT + Mosquitto:** MQTT es un protocolo ligero de publicación/suscripción que, por su bajo *overhead* y simplicidad de implementación, resulta ideal para dispositivos con recursos limitados. Mosquitto es un broker MQTT *open source* ampliamente adoptado en sistemas IoT, para pruebas en ambiente de test. En un entorno de producción, será necesario hostear un broker dedicado para garantizar disponibilidad y control sobre la infraestructura de comunicación.

Desafíos técnicos del sistema

El diseño del sistema presentó múltiples desafíos:

- **Conectividad intermitente:** dado que tanto la comunicación BLE como NB-IoT pueden sufrir desconexiones, se implementaron mecanismos de reconexión automática y almacenamiento temporal.
- **Latencia tolerable:** el sistema debe reaccionar en segundos a eventos como detecciones o solicitudes del servidor.
- **Separación de responsabilidades:** mediante la creación de tareas virtuales independientes (VTasks) se logró desacoplar la ejecución de inferencias, la gestión del stack BLE y los procesos de monitoreo.

5.2. Introducción al Stack NimBLE

5.2.1. GAP y GATT: Arquitectura Funcional y Modelo de Datos en BLE

Para comprender el diseño de las comunicaciones en el sistema Parkwise, es fundamental distinguir entre las dos principales abstracciones de alto nivel del protocolo BLE: GAP y GATT. Estas capas, aunque complementarias, cumplen roles bien diferenciados dentro del stack de protocolos BLE que implementa NimBLE sobre ESP-IDF.

GAP se encarga de definir los procedimientos de descubrimiento, publicidad, emparejamiento y establecimiento de conexiones entre dispositivos. En otras palabras, determina cómo se inicia una conexión y bajo qué condiciones. Por ejemplo, define si un dispositivo actúa como periférico (advertising) o central (escaneo y conexión), y cómo se intercambian los roles durante la negociación inicial. Este es el nivel donde se configuran los parámetros de conexión, como el intervalo de conexión, latencia, tiempo de supervisión y el uso de direcciones públicas o aleatorias.

GATT, en cambio, es la capa encargada de estructurar y transportar los datos una vez que la conexión BLE ha sido establecida. Basado sobre ATT (Attribute Protocol), GATT define una jerarquía lógica compuesta por servicios, características y descriptores. Esta jerarquía organiza los datos que un dispositivo BLE expone (como servidor) o consume (como cliente), y es la que se utiliza en Parkwise para transmitir inferencias, imágenes, configuraciones y comandos entre nodos.

La Figura 5.2 ilustra la arquitectura de capas que componen a BLE, que se separa en dos conjuntos, "host" y "controller" separadas por la interfaz estándar HCI (Host Controller Interface). En el "host" se ubican GAP y GATT. GAP se encarga de *cómo nos conectamos* y GATT de *qué datos intercambiamos una vez conectados*. Ambos son indispensables para lograr una comunicación BLE eficiente y estructurada, y su correcta implementación fue fundamental para el funcionamiento estable del sistema.

La Figura 5.3 muestra la estructura jerárquica del modelo de datos GATT, donde los datos se organizan en servicios, características y descriptores.

5.2.2. Diseño e Implementación del Servidor GATT con NimBLE

El Dispositivo Parkwise toma el rol de servidor GATT debido a que éste es el que produce datos, sean imágenes o inferencias de vehículos. El protocolo está sobre la implementación de NimBLE y del servidor GATT adaptadas del ejemplo de Espressif [40].

El Dispositivo Parkwise como servidor GATT, anuncia una serie de **servicios** y **características** que encapsulan las principales funcionalidades del sistema. En el proyecto se utilizó un único servicio, y bajo el mismo diferentes características, entre ellas algunas implementadas fueron:

- Transmisión de imágenes
- Detecciones
- Mapas de calor
- Recepción de comandos externos

El servicio cuenta con un identificador único universal (**UUID**), y se registra en una tabla que permite su descubrimiento automático por parte de cualquier cliente compatible con BLE.

El diseño se basa en funciones que se ejecutan en respuesta a operaciones iniciadas por el cliente, llamadas *callback functions*. Estas funciones reciben como parámetro un conjunto de estructuras de contexto, que contienen toda la información relevante sobre

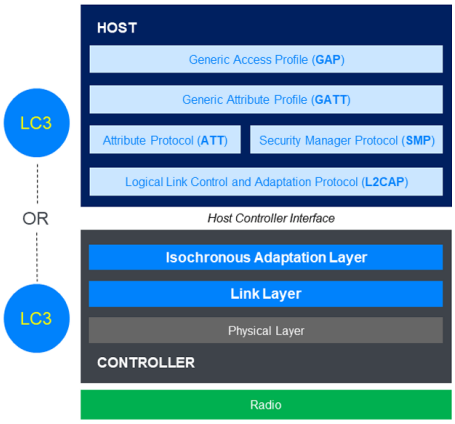


Figura 5.2: Arquitectura de Bluetooth LE (Stack de BLE). Fuente [6]

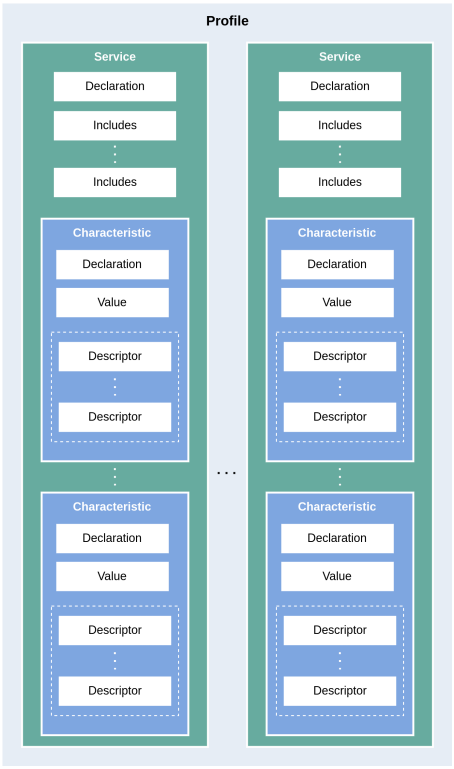


Figura 5.3: Estructura de datos en GATT fuente [37]

el evento ocurrido, incluyendo los metadatos *handle*. Los *handle* permiten determinar el tipo de operación realizada y acceder a los datos transmitidos. operación solicitada y los datos transmitidos.

Cada característica del servidor tiene asociado un *callback* que se activa ante alguno de los siguientes tres tipos de operación: lectura, escritura o suscripción. Para estos últimos, GATT ofrece un mecanismo flexible que distingue entre dos tipos de eventos asincrónicos:

- **Notificaciones** (sin confirmación): Son mensajes unidireccionales que el servidor envía al cliente. El cliente no confirma la recepción de datos, se pueden utilizar para actualizaciones de estado en tiempo real.
- **Indicaciones** (con acuse de recibo): Son mensajes que esperan una confirmación de recepción por el cliente. Esto las hace más confiables, pero también más lentas. Se utilizan para datos críticos que no pueden perderse.

Esta diferenciación permite adaptar el comportamiento del sistema según las particularidades del mensaje. En Parkwise, se utilizan notificaciones, debido a que son muchos paquetes enviados en poco tiempo, tanto con una imagen como con detecciones, por lo que mejorar la velocidad es relevante.

Cabe señalar que las detecciones y los comandos se gestionan directamente dentro del contexto de sus respectivos *callbacks*, mientras que la transmisión de imágenes se realiza de forma periódica a través de una tarea programada en una rutina independiente que recorre los clientes suscritos y les envía los fragmentos correspondientes.

Esta separación entre *callbacks* y tareas programadas permite un control preciso de los recursos del sistema.

El dipositivo Parkwise al reiniciarse estará esperando una conexión, mientras que el Gateway está buscando dispositivos por nombre (en este caso, programados como `Parkwise Device`).

En caso de que la conexión se bloquee, al pasar determinado tiempo la tarea virtual dedicada `reset_timer_task` puede forzar un reinicio seguro (Cap. 4.2). Este mecanismo refuerza la robustez del sistema, que se vuelve a conectar con el Gateway, procurando un servicio fluido.

De este modo, el diseño del servidor GATT bajo NimBLE no solo responde a requerimientos funcionales, sino que también refleja decisiones de arquitectura pensadas para asegurar modularidad, eficiencia y tolerancia a fallos.

5.2.3. Proceso de Conexión y Suscripción

La inicialización del stack BLE y la activación del servidor GATT están estrechamente ligadas al módulo GAP, que define los parámetros fundamentales de conectividad:

- Dirección MAC del dispositivo
- Nombre de difusión (*advertising name*)
- Intervalos de *advertising*
- Tipo de dirección (pública o aleatoria)
- Configuración de seguridad básica
- Latencia
- Tamaño máximo de paquetes
- Tiempo de *timeout*

El proceso de conexión BLE en Parkwise sigue la lógica estándar definida por el perfil GAP [36], donde el Dispositivo Parkwise actúa como periférico (*Peripheral*) y los clientes (Gateway o instalador) como dispositivos centrales (*Central*). El Dispositivo Parkwise comienza anunciando su presencia mediante paquetes de *Advertising*, los cuales contienen información básica como el nombre del dispositivo, su dirección MAC y los servicios disponibles.

Una vez detectado el periférico, el cliente puede iniciar el proceso de conexión. Este implica una negociación de los ya mencionados parámetros como el intervalo de conexión, la latencia de conexión y el tiempo de supervisión (*supervision timeout*), configurados en este proyecto para lograr un equilibrio entre estabilidad de enlace, y velocidad de respuesta.

Tras la conexión, el cliente realiza una exploración de servicios (*service discovery*), donde obtiene la lista de UUIDs registrados por el servidor GATT. Cada característica publicada puede incluir permisos de lectura, escritura y suscripción. En particular, la suscripción permite al cliente recibir notificaciones o indicaciones sin necesidad de solicitar activamente cada dato.

En el Dispositivo Parkwise, la suscripción se gestiona a través de las llamadas *subscribe event*, que activan una bandera interna cuando un cliente se vincula a una característica. El sistema mantiene **tablas de conexión y suscripción** específicas para cada característica, que son consultadas por las funciones de envío antes de transmitir los datos. De esta manera sólo los clientes correctamente conectados y suscritos reciben los datos.

Este mecanismo no solo garantiza una comunicación eficiente, sino que también permite la coexistencia de múltiples clientes, facilitando escenarios como la conexión simultánea de un instalador y un Gateway durante el proceso de configuración.

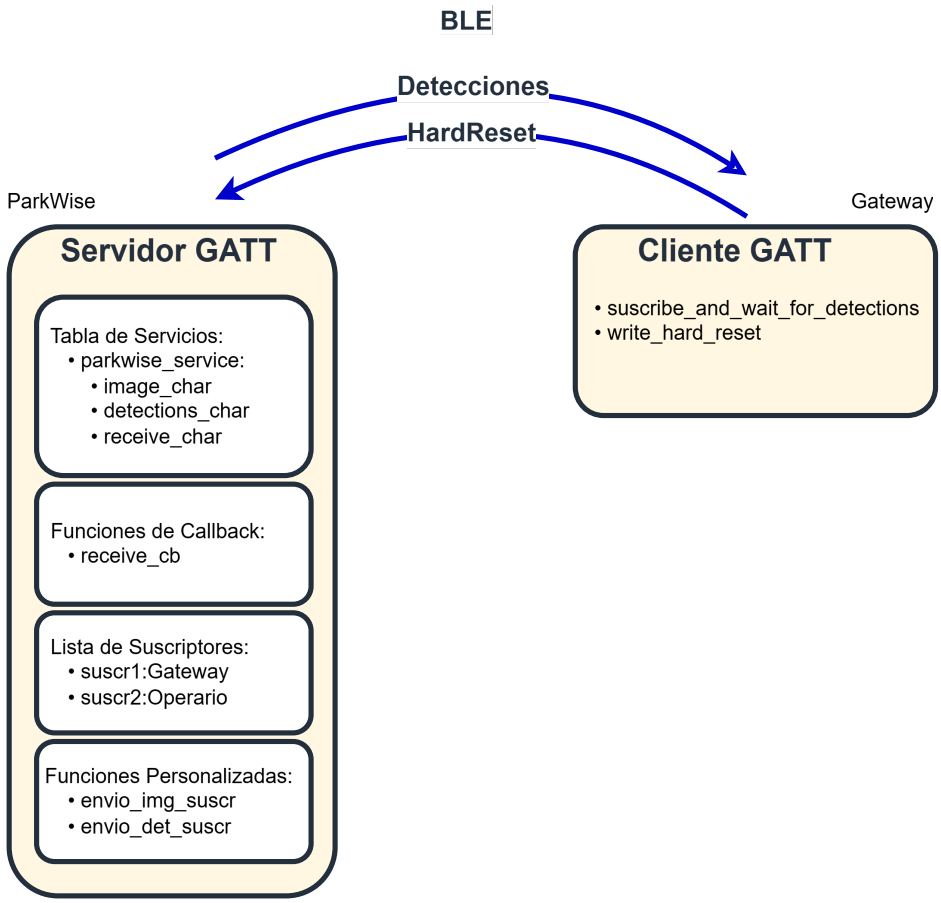


Figura 5.4: Interacción entre Parkwise y Gateway mediante GATT sobre BLE.

5.3. Comunicación BLE entre Dispositivo Parkwise y Gateway

5.3.1. Actores en BLE

El vínculo entre el Dispositivo Parkwise y el Gateway tiene tres operaciones: el envío de detecciones o imágenes desde el Dispositivo Parkwise hacia el Gateway y el envío de comandos de reinicio (hard reset) desde el Gateway hacia el Dispositivo Parkwise. Esta relación se implementa completamente sobre BLE utilizando el perfil GATT, donde el Dispositivo Parkwise actúa como servidor y el Gateway como cliente. La Figura 5.4 resume esta arquitectura simplificada de interacción.

Del lado del Servidor GATT corriendo en el Dispositivo Parkwise (ESP32S3), se aprecia la mencionada tabla de servicios que guarda jerárquicamente las UUIDs de los servicios y características.

Seguido a eso están las funciones de *callback*, en este caso solo hizo falta una, las cual fue configurada para responder a operaciones de escritura.

Parkwise mantiene una lista interna de suscriptores activos, actualizada dinámi-

camente con los identificadores del cliente BLE (como el Gateway o el Operario).

Estas son llamadas en momentos particulares del *pipeline* principal de la *vision task*, por ejemplo luego de una nueva inferencia; y no desencadenadas por eventos externos.

Se aprecian dos funciones personalizadas, en el caso de `envio_img_suscr` se usa únicamente con el operario, que define la Homografía a partir de los puntos en el mapa.

Por otra parte, del lado derecho se aprecia el cliente GATT, el cual tiene dos funciones en lo que respecta a su interacción con el servidor GATT. La primera es para suscribirse a las detecciones, y otra para escrituras donde se envía el comando para reiniciar el dispositivo.

Resumiendo, la relación entre los dos dispositivos sigue un esquema en el que el Gateway inicia la comunicación con un evento de conexión seguido de uno de suscripción en el servicio Parkwise y más puntualmente en la característica image_char. Los datos del cliente (identificación, servicio y característica a la que se suscribe) son almacenados en la lista de suscriptores que tiene el servidor, para que luego las envíe con indicaciones paquetes de datos mediante la `función envio_det_suscr`.

5.3.2. Serialización a Formato JSON

Una vez completada cada inferencia sobre la imagen capturada, el modelo genera una o más estructuras de datos conocidas como *bounding boxes*, cada una de las cuales representa un objeto detectado, junto con sus coordenadas, tamaño, etiqueta y tiempo requerido para su inferencia.

La información se convierte a formato **JSON** para enviarla al Gateway y posteriormente transmitirla al servidor central mediante MQTT. Cada detección se encapsula como una lista de detecciones en un diccionario JSON. Los campos incluidos son los siguientes:

- **score**: del modelo para la detección (float).
- **x, y**: coordenadas del vértice superior izquierdo del objeto detectado.
- **width, height**: dimensiones del *bounding box*.

El resultado de la serialización se ilustra en la Figura 5.5, donde se aprecia cómo cada *bounding box* es transformada en una cadena JSON antes de ser almacenada temporalmente en un búfer de salida.

Como resultado, se obtiene un mensaje como el siguiente:

```
{
  "d": [
    [1.517732, 296, 224, 8, 8]
  ]
}
```

Figura 5.5: Ejemplo de paquete serializado de una bounding box

Este formato compacto y estructurado permite un tratamiento eficiente por parte del cliente GATT (el Gateway), que puede reenviarlo directamente en su forma serializada al broker MQTT agregándole la mac del dispositivo que recibe con el formato: `"m": "74:4D:BD:95:A3:0A"`.

Figura 5.6: Dispositivos Arduino Nano 33 BLE, placa FIng y MikroE LTE IOT 2 Click

5.3.3. Actualización del Búfer de Detecciones

Una vez generada la detección y serializada en formato JSON, el sistema no transmite inmediatamente la información al cliente BLE, sino que primero la almacena temporalmente en un búfer de memoria RAM. Esta actualización se realiza mediante una función específica denominada `update_gatt_detections()`, la cual copia el contenido de la detección al área de memoria que luego será utilizada por el mecanismo de envío.

Esta decisión permite darle robustez y claridad al diseño en los siguientes aspectos:

- **Desacoplamiento entre etapas:** almacenar el JSON de forma intermedia permite separar lógicamente la etapa de inferencia del proceso de transmisión inalámbrica. De este modo, el ciclo de inferencia puede continuar sin depender de la disponibilidad del stack BLE o del estado del cliente suscriptor.
- **Modularidad y claridad en la gestión de variables:** desde una perspectiva de buenas prácticas de programación embebida, la función `update_gatt_detections()` actúa como una interfaz controlada para modificar el estado del búfer, limitando la visibilidad de la variable global a un único punto de escritura. Luego, otras funciones como `send_detections_to_subscribers()` acceden a dicho búfer en modo de solo lectura, respetando así el principio de separación de responsabilidades.

5.3.4. Recepción desde el Gateway

Incorporación a sistema de luminarias

Inicialmente se consideró la posibilidad de incorporar al proyecto una iniciativa paralela desarrollada por Leonardo Steinfield en colaboración con la Intendencia de Montevideo. Esta consistía en medir el envejecimiento de luminarias viales mediante un Arduino Nano 33 BLE, el cual recolectaba datos y los transmitía vía MQTT utilizando el módulo MikroE LTE IoT 2 Click, con conectividad basada en NB-IoT. En nuestro caso, se intentó reutilizar esta plataforma como base para el diseño del Gateway. Para ello, se utilizó una placa de interconexión que integraba un footprint mikroBUS y otro compatible con Arduino, permitiendo la comunicación entre ambos dispositivos. La configuración del módulo LTE se realizaba mediante comandos AT enviados a través de uno de los puertos seriales del Arduino Nano 33 BLE. La figura 5.6 muestra los dispositivos utilizados en esta etapa.

Ambos módulos funcionaban correctamente de forma independiente, pero la comunicación entre ellos no se logró resolver a nivel de hardware. Esto implicó utilizar una

simulación del Gateway en Python a través de una PC. A continuación se detallan las pruebas realizadas en Arduino y MIKROE LTE IoT 2 Click en el intento de concretar el punta a punta.

- **Passthrough por los puertos seriales de Arduino Zero con placa de interconexión Arduino para mikroBUS y LTE IoT 2 Click** esta prueba funcionó.
- **Transmisión desde terminal comandos por MQTT desde Arduino Zero y LTE IoT 2 click interconectados con placa de interconexión de Arduino** Funcionó. Permitió configurar el LTE IoT 2 Click y transmitir por Mosquitto.
- **Transmisión desde terminal a comandos por BLE desde Arduino nano 33 BLE, visualizados a través de nrfConnect en el Celular.** Funcionó el ida y vuelta, tanto para enviar como para recibir en terminal.
- **Passthrough por los puertos seriales de Nano 3 BLE con placa de interconexión FIng y LTE IoT 2 Click** esta prueba no funcionó, se sospecha de la placa o del Nano 33 BLE.

Para que el Gateway cumpliera su rol, era necesario lograr que los datos recibidos por BLE fueran reenviados vía MQTT y, a su vez, que mensajes provenientes del servidor pudieran enviarse por BLE al Dispositivo Parkwise correcto. Además, como el Gateway debe interactuar con varios nodos Parkwise en distintos momentos, era clave implementar un mecanismo para almacenar temporalmente los mensajes dirigidos a cada dispositivo hasta que estuvieran disponibles para recibirlos. Esta lógica de intercambio dinámico no pudo completarse en el entorno Arduino debido a las limitaciones encontradas en el hardware.

Gateway Simulado con un Script

Ante las dificultades con la solución embebida, se optó por simular el Gateway con una laptop usando un script en Python. Este script, basado en la biblioteca `Bleak` [17], se encarga de conectarse por BLE a los Dispositivos Parkwise, recibir las detecciones en formato JSON, y reenviarlas automáticamente al broker MQTT utilizando la biblioteca `paho-mqtt` [8].

El proceso completo puede dividirse conceptualmente en tres etapas:

1. **Descubrimiento y conexión BLE:** se escanean los dispositivos disponibles. Al encontrar un dispositivo cuyo nombre coincida con el valor de `DEVICE_NAME` (definido en un archivo `.env`), se obtiene su dirección MAC y se procede a establecer una conexión BLE.
2. **Suscripción a notificaciones:** una vez conectado, el script comienza a escuchar las notificaciones sobre la característica identificada por `BBOX_CHARACTERISTIC`. A partir de ese momento, cada vez que Parkwise envía una detección como notificación, el manejador de notificaciones la decodifica desde bytes a string y la almacena temporalmente como `data_json`.
3. **Reenvío al broker MQTT:** cuando una nueva detección es recibida, el cliente publica el mensaje decodificado en un tópico MQTT predefinido. Este paso garantiza que el servidor central, suscrito a dicho tópico, pueda consumir la información. El flujo completo de esta transmisión se desarrolla en la Sección 5.7.

Este flujo asegura un puente continuo entre BLE y MQTT, actuando como una pasarela eficiente para llevar la información desde el sensor Parkwise hacia el backend del sistema.

5.4. Comunicación BLE entre Parkwise (ESP32-S3) y Operario

Durante el proceso de configuración inicial del sistema, el operario solicita una imagen a cada Dispositivo Parkwise para realizar posteriormente el ajuste de homografía desde la interfaz web. Esta interacción forma parte del denominado *modo setup* (ver Sección 6.1.2) y se implementa mediante la suscripción del cliente (el software del operario) a la característica `image_char` del servicio GATT.

5.4.1. Servicio de envío de imágenes

La transmisión de imágenes capturadas por la cámara del Dispositivo Parkwise impone un desafío técnico importante: el tamaño típico de las imágenes excede la unidad máxima de transmisión (MTU) permitida por BLE, de 256 bytes. Por este motivo, se desarrolló un mecanismo de fragmentación controlada de imágenes que permite dividir los datos en múltiples paquetes más pequeños y enviarlos de forma secuencial al cliente.

Trayectoria de desarrollo y evolución de versiones

El proceso de fragmentación pasó por múltiples iteraciones. Las primeras versiones intentaban enviar la imagen como un bloque monolítico, lo que generaba errores recurrentes de transmisión y pérdida de datos. Posteriormente, se ensayaron esquemas de división sin encabezado, lo que resultó en una mejora parcial, aunque con dificultades para la reconstrucción en el lado del cliente.

Finalmente, se adoptó una arquitectura basada en fragmentos numerados, en la que cada paquete contiene un encabezado de 4 bytes que indica:

- Número de secuencia del fragmento (2 bytes).
- Total de fragmentos esperados (2 bytes).

Este diseño facilita el reensamblado en el cliente, mejora la trazabilidad del flujo de datos y permite detectar pérdidas o duplicados. Además, el servidor BLE ajusta parámetros como el tamaño de MTU, el intervalo de conexión y el tiempo de supervisión para optimizar el rendimiento y la estabilidad del vínculo.

Estrategia final de fragmentación y encabezado

El esquema de fragmentación controlada divide la imagen en paquetes pequeños compatibles con la unidad máxima de transmisión (MTU). Este proceso se detalla en la Figura 5.7, que ilustra la estructura del búfer de imagen, el cálculo de fragmentos y la composición de cada paquete enviado.

La imagen, de resolución 320×240, se guarda como un *flatbuffer* en memoria, dando lugar a un total de `img_size` = 76.800 bytes. Este búfer es recorrido con un puntero llamado `offset`, y en cada iteración se extrae un fragmento de longitud máxima `chunk_size`, definido como el mínimo entre `MAX_PACKET_SIZE - 4` y `img_size - offset - 4`, reservando 4 bytes para el encabezado.

El encabezado de cada fragmento sigue la siguiente estructura:

- Bytes 0–1: número de secuencia (LSB y MSB).
- Bytes 2–3: número total de fragmentos esperados (LSB y MSB).

Este encabezado cumple una función crítica: permite al cliente reensamblar la imagen completa, detectar fragmentos perdidos y gestionar la visualización del progreso.

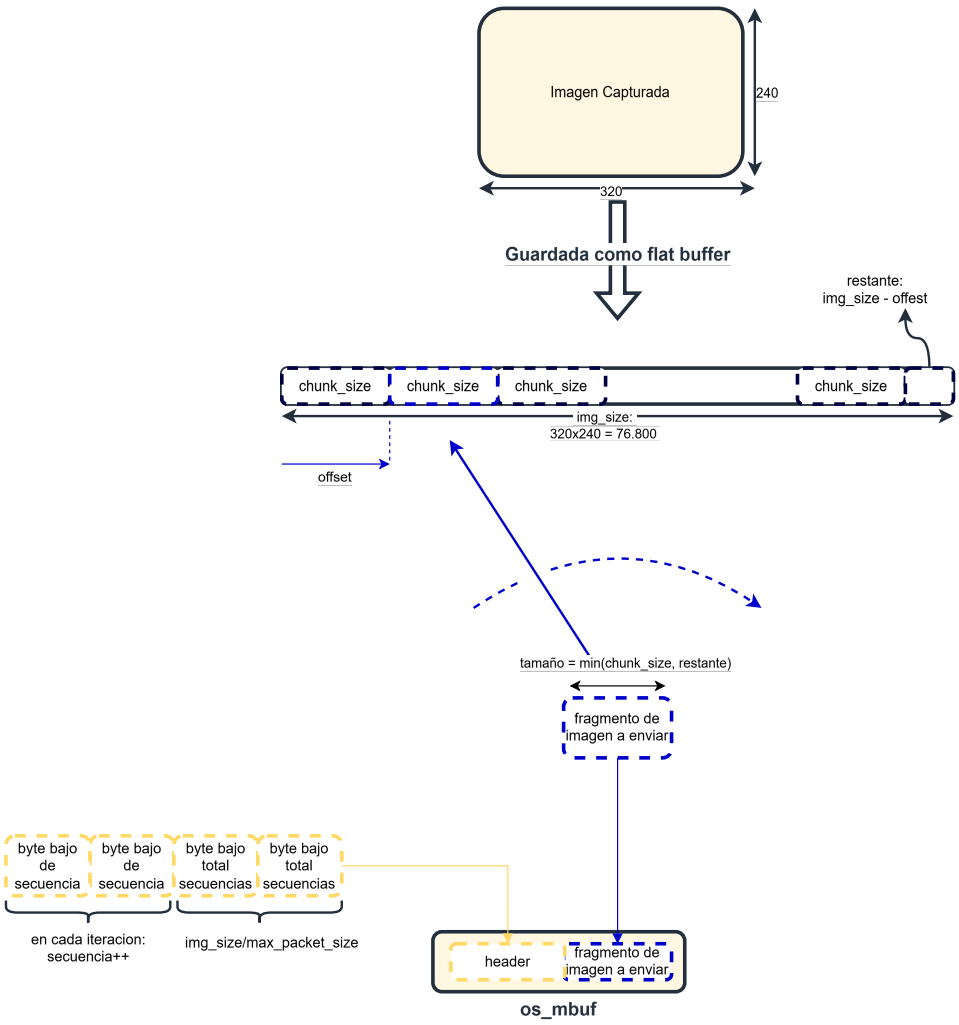


Figura 5.7: Esquema del proceso de fragmentación de imágenes para transmisión vía BLE. La imagen capturada se divide en fragmentos secuenciales, cada uno precedido por un encabezado de 4 bytes que informa su posición y permite la reconstrucción del conjunto.

A diferencia de versiones anteriores, que no incluían este nivel de control, este diseño introdujo mejoras claras en estabilidad, trazabilidad y compatibilidad entre módulos.

Cada paquete BLE es ensamblado utilizando una estructura del sistema llamada `os_mbuf`, la cual forma parte del sistema de búfers de NimBLE. Este tipo de búfer es gestionado por el sistema operativo embebido (RTOS) y está optimizado para operaciones de red, como el envío de notificaciones o indicaciones BLE. La función `send_rawimage_UINT_to_suscribers()` se encarga de todo este flujo, concatenando dinámicamente el encabezado con el fragmento correspondiente del búfer de imagen y asegurando que:

- El búfer se asigne correctamente al `os_mbuf`.
- El cliente BLE permanezca conectado haciendo consultas de conexión periódicas.

- No se exceda el tamaño máximo permitido por paquete.

El proceso se repite hasta que el offset alcanza el tamaño total de la imagen. En cada iteración, el índice de secuencia se incrementa y se actualiza la cantidad restante por enviar. Gracias a esta estrategia, el sistema logra una transmisión robusta en entornos reales con conectividad limitada, manteniendo la fluidez incluso con enlaces inestables.

Flujo de ejecución.

Antes de iniciar el envío, el sistema verifica que exista al menos un cliente suscrito y que el dispositivo se encuentre en estado operativo. Luego se procede con el siguiente flujo:

- Se calcula el número total de fragmentos a enviar.
- Se ajustan los parámetros de conexión BLE (intervalo de conexión, tiempo de supervisión, MTU).
- En cada iteración:
  - Se genera el encabezado.
  - Se asigna el búfer (`os_mbuf`).
  - Se copian encabezado y fragmento al búfer.
  - Se envía el paquete con `ble_gatts_notify_custom()` como una notificación (mayor eficiencia en velocidad de transmisión que las indicaciones).
  - Se actualiza `offset` y `sequence`.
- Al finalizar, se registra un mensaje de éxito o se aborta en caso de error de asignación o envío.

5.4.2. Reconstrucción de imágenes

La reconstrucción de imágenes transmitidas mediante BLE fue implementada en el operario como parte del proceso de configuración inicial del Dispositivo Parkwise. Una vez suscrito a la característica `image_char`, el cliente comienza a recibir notificaciones con los fragmentos de imagen, cada uno acompañado de un encabezado de 4 bytes que permite su ordenamiento y ensamblado.

El cliente mantiene una lista de fragmentos recibidos en memoria, los cuales son almacenados temporalmente en una estructura interna. Cada vez que se recibe un nuevo fragmento, se extrae el número de secuencia y el total de fragmentos esperados desde el encabezado. Esto permite:

- Insertar cada fragmento en la posición correspondiente.
- Verificar cuántos fragmentos faltan.
- Detectar reinicios del envío (cuando el número de secuencia vuelve a 0).

Una vez que se recibe el último fragmento (identificado cuando el índice vuelve a 0 o se completa la cantidad esperada), los fragmentos se concatenan en orden para formar un único búfer de bytes. Este búfer se convierte luego en una imagen en escala de grises de 320×240 píxeles. Para asegurar compatibilidad, si el tamaño recibido es menor al esperado, se completa con ceros.

Durante este proceso, se implementaron herramientas auxiliares para:

- Visualizar el avance de la recepción mediante una barra de progreso (`tqdm`).

- Registrar eventos y errores durante la comunicación (`logging`).
- Guardar la imagen recibida como archivo PNG en una carpeta específica del sistema.
- Inicializar al Dispositivo Parkwise en una base de datos local para su posterior configuración.

Los detalles de dichas funcionalidades de recontruccion y su estructura de código en Python son analizadas en el Capítulo 6.

El esquema completo de recepción opera de forma asincrónica, permitiendo al sistema procesar múltiples dispositivos en paralelo y recibir imágenes de varios nodos durante una misma sesión de configuración. Esta capacidad es clave para el escalado del sistema y para mantener tiempos bajos en entornos con múltiples sensores.

En resumen, el cliente implementado logra:

- Detectar e interpretar correctamente los encabezados de cada fragmento.
- Reensamblar imágenes completas en memoria.
- Persistirlas en disco y registrar el dispositivo automáticamente para su configuración posterior.

Este módulo de recepción y reconstrucción fue validado extensamente en pruebas de campo, demostrando un comportamiento robusto incluso con conexiones BLE de baja calidad.

5.4.3. Posibles mejoras a futuro

Si bien la implementación actual cumple con los requisitos funcionales, existen posibles líneas de mejora:

- Implementar confirmaciones a nivel de aplicación por parte del cliente.
- Incorporar mecanismos de retransmisión selectiva para fragmentos perdidos.
- Estudiar la factibilidad de utilizar indicaciones BLE en lugar de notificaciones, priorizando confiabilidad por sobre velocidad.

El resultado final es un módulo de transmisión de imágenes eficiente, extensible y operativo en escenarios reales de instalación, permitiendo al operario realizar el setup de homografía con fiabilidad desde un dispositivo móvil o PC conectado por BLE.

5.5. Comunicación BLE entre el Dispositivo Parkwise y Servidor Central

Una vez configurado el sistema por el operario en el modo Setup, tanto en la búsqueda y solicitud de imágenes de los Parkwise, como la configuración desde la web (ver Sección 6.1.2); Parkwise comienza a operar en régimen normal. En este modo, una de las opciones disponibles es enviar detecciones directamente al Servidor Central utilizando Bluetooth Low Energy. Esta funcionalidad actúa como una vía alternativa —o complementaria— al canal NB-IoT, permitiendo mantener comunicación directa con el servidor en situaciones específicas, como pruebas de laboratorio, entornos sin conectividad celular, o simplemente para acelerar el flujo en instalaciones reducidas.

La implementación de este vínculo se realiza mediante el perfil GATT, donde Parkwise opera como servidor y el Servidor Central como cliente. Este último se suscribe a la característica `detections_char` del único servicio ofrecido (`parkwise_service`),

recibiendo así en tiempo real las detecciones generadas por el nodo embebido. Estas detecciones están previamente serializadas en formato JSON, siguiendo el mismo procedimiento descrito en la Sección 5.3.2.

Además, el Servidor Central tiene la capacidad de emitir comandos mediante escritura sobre la característica `receive_char`. Uno de los más importantes es el *hard reset*, utilizado para forzar una reinicialización segura del sistema embebido en caso de error o necesidad de reconfiguración.

La Figura 5.8 resume esta arquitectura de interacción directa.

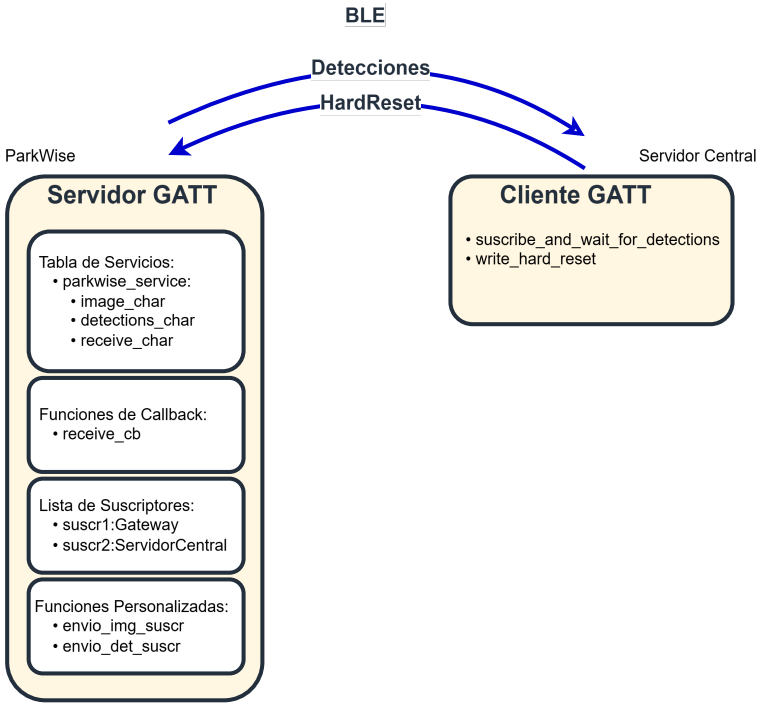


Figura 5.8: Interacción directa entre Parkwise y el Servidor Central mediante GATT sobre BLE. El servidor GATT (Parkwise) expone características que permiten enviar detecciones y recibir comandos. El Servidor Central actúa como cliente GATT y se suscribe a las detecciones mientras que puede enviar instrucciones como el hard reset.

Este canal BLE forma parte del mismo módulo de software que implementa la lógica de configuración y administración del sistema desde una PC portátil. Tanto el intercambio de imágenes para el modo Setup (cubierto en la siguiente sección), como el flujo continuo de detecciones en tiempo real, comparten una base de datos local y estructuras de código comunes. Esta decisión arquitectónica facilita el mantenimiento del código, la persistencia de configuraciones, y la trazabilidad del comportamiento del sistema embebido.

5.6. Introducción al Broker Mosquito y NB-IoT

En esta sección se introducen los conceptos básicos necesarios para comprender el vínculo entre el Gateway y el Servidor Central, que se desarrolla en detalle en las secciones 5.6.1, 5.6.2 y 5.6.3. Se describen los protocolos y tecnologías que permiten

establecer una comunicación eficiente, confiable y escalable entre dispositivos capaces de sensar e intercambiar información a través de internet sin intervención humana, lo que se conoce como Internet de las Cosas (IoT, por sus siglas en inglés).

5.6.1. MQTT: un protocolo liviano para IoT

MQTT (Message Queuing Telemetry Transport) es un protocolo de mensajería diseñado para dispositivos con recursos limitados y redes de alta latencia o intermitentes. Funciona bajo un modelo de publicación/suscripción, donde los *publishers* publican mensajes en *topics* (tópicos), y los *subscribers* (suscriptores) reciben aquellos mensajes de los *topics* a los que están suscritos. Su simplicidad y bajo overhead lo hacen ideal para sistemas embebidos que necesitan enviar datos periódicamente con bajo consumo energético [4].

5.6.2. Mosquitto como broker MQTT

Eclipse Mosquitto es una implementación libre, ligera y altamente confiable del protocolo MQTT, compatible con la versión 5.0 del estándar [4]. Actúa como intermediario entre dispositivos publicadores y suscriptores, gestionando la recepción y distribución de mensajes. Entre sus funcionalidades clave destacan:

- Soporte para múltiples clientes concurrentes.
- Retención de mensajes, permitiendo que nuevos suscriptores reciban la última información publicada en un *topic*.
- Bajo consumo de recursos y alta escalabilidad.

En este proyecto, Mosquitto opera como el núcleo de la infraestructura IoT, siendo responsable de recibir los datos publicados por el Gateway y distribuirlos al Servidor Central.

5.6.3. NB-IoT: comunicación celular de bajo consumo

Narrowband IoT (NB-IoT) es una tecnología de comunicación basada en redes celulares (4G/LTE) especialmente diseñada para dispositivos IoT que requieren transmisión ocasional de pequeños volúmenes de datos. Opera sobre un ancho de banda de apenas 180–200 kHz, y ofrece:

- Alta penetración en interiores y subterráneos.
- Larga duración de batería (hasta varios años).
- Compatibilidad con la infraestructura de las operadoras móviles.

NB-IoT puede ser utilizado en este sistema para conectar el Gateway físico con Mosquitto, permitiendo una comunicación ubicua sin depender de redes Wi-Fi locales. En cambio, no se utiliza NB-IoT entre el Dispositivo Parkwise y el Gateway, donde se prioriza el uso de BLE por su bajo consumo y simplicidad de configuración local.

5.6.4. Resumen de integración

La elección de MQTT sobre NB-IoT con Mosquitto como broker responde a criterios de eficiencia, modularidad y robustez. El Gateway, inicialmente pensado para implementarse en Arduino, se conecta a la red celular mediante un módem NB-IoT y publica los mensajes formateados en JSON bajo diferentes *topics*.

5.7. Comunicación IoT entre Gateway y Servidor Central a través del Broker Mosquito

En este apartado se describe el vínculo entre el Gateway y el Servidor Central, establecido mediante el protocolo MQTT sobre una red NB-IoT. Esta conexión permite el reenvío de detecciones captadas por los nodos Parkwise hacia un backend centralizado, así como la recepción de comandos remotos, como el reinicio forzado de nodos.

5.7.1. Estructura general de la comunicación

El flujo de datos se basa en el modelo de publicación/suscripción provisto por MQTT. El Gateway actúa como *publisher*, enviando mensajes JSON al broker Mosquitto bajo *topics* específicos según el tipo de evento o dispositivo emisor. El Servidor Central, en cambio, opera como *subscriber*, escuchando aquellos *topics* de interés y procesando los datos recibidos. Cada mensaje contiene una detección codificada en formato JSON, como se detalla en la Sección 5.3.2.

La estructura típica de un mensaje publicado es la siguiente:

- **Topic**: `/test/parkwise/`
- **Payload**: objeto JSON conteniendo los datos de la detección, incluyendo coordenadas, etiqueta, timestamp y dirección MAC del nodo.

La Figura 5.9 resume gráficamente esta arquitectura. El Servidor GATT embebido en Parkwise transmite detecciones al Cliente GATT (el Gateway), que utiliza un `notification_handler()` para capturar las notificaciones BLE entrantes y almacenarlas en un búfer temporal. Un bucle asíncrono monitorea este búfer y publica los datos en el broker Mosquitto cuando están disponibles. Finalmente, el Servidor Central consume estos mensajes desde el broker para procesarlos.

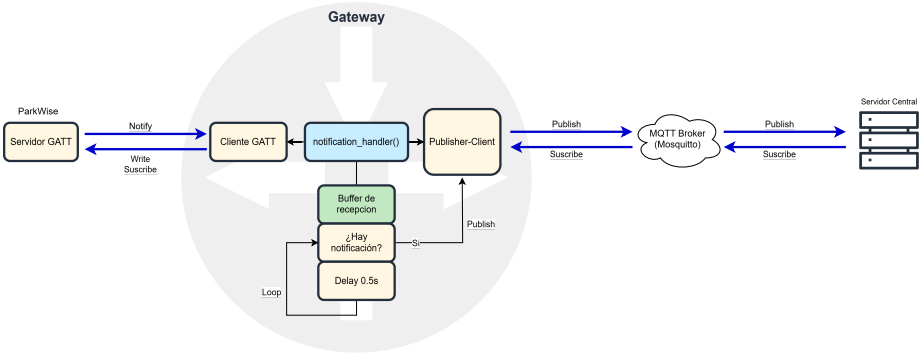


Figura 5.9: Estructura general del flujo de datos entre Parkwise, Gateway y el Servidor Central. La detección BLE es almacenada temporalmente en un búfer de recepción y luego publicada al broker Mosquitto mediante MQTT. El servidor se suscribe a los tópicos correspondientes para recibir los datos.

5.7.2. Plataforma de desarrollo y bibliotecas utilizadas

Inicialmente, el desarrollo del Gateway se realizó sobre una arquitectura de doble microcontrolador en Arduino: uno para manejar BLE y otro para NB-IoT, comunicados mediante un shield intermedio. Sin embargo, como se mencionó en la sección 5.3.4, debido a un problema de hardware que impedía la comunicación entre ambos

módulos —a pesar de que funcionaban correctamente por separado— se optó por una implementación alternativa en PC. Esta decisión se justifica en la Sección 5.3.4, donde se explican los motivos del cambio y los beneficios de esta solución.

El código final se desarrolló en Python utilizando las siguientes bibliotecas:

- `bleak` [5] para la comunicación BLE.
- `paho-mqtt` [14] para la publicación de mensajes MQTT.
- `dotenv` [13] y `asyncio` [15] para la configuración y ejecución asincrónica del sistema.

5.7.3. Gestión de publicaciones y calidad de servicio

Cada detección recibida por BLE se almacena brevemente en memoria como lista de JSONs. Luego, se publica en el broker. Una vez enviado el mensaje, el búfer se limpia para esperar nuevos datos.

Además, Mosquitto permite la retención del último mensaje publicado en cada *topic*, de modo que nuevos suscriptores puedan acceder inmediatamente al estado más reciente, incluso si se conectan más tarde.

5.7.4. Gestión de reconexión y latencia

El sistema fue diseñado para tolerar fallos momentáneos de conectividad, permitiendo la reconexión automática al broker Mosquitto si la red NB-IoT se interrumpe. A su vez, la latencia típica observada fue inferior a 2 segundos en condiciones de campo, lo cual resulta adecuado para el tipo de aplicación.

5.7.5. Recepción de detecciones en el Servidor Central

En el otro extremo del vínculo MQTT, el Servidor Central recibe las detecciones publicadas por el Gateway a través del tópico correspondiente. Este mecanismo está gestionado por la clase `NBIoTManager`, que se encarga de la conexión con el broker, la suscripción y el procesamiento de los mensajes entrantes.

Al iniciarse, el servidor establece una conexión con Mosquitto mediante `paho-mqtt` y se suscribe al tópico correspondiente (por ejemplo, `/parkwise/{mac_address}/detections`). Cada vez que se recibe un nuevo mensaje, se activa el método de la clase llamado `on_message()`, el cual:

1. Decodifica el *payload* desde formato UTF-8.
2. Carga el contenido como objeto JSON.
3. Extrae los campos relevantes: dirección MAC, coordenadas `x/y`, dimensiones de la detección (`width/height`).
4. Registra la detección en la base de datos local agregándole un timestamp asociado.

Este mecanismo permite mantener un registro estructurado de los eventos detectados en campo, asociando cada detección con el nodo que la generó. Se implementan controles de integridad para asegurar que todos los campos requeridos estén presentes, y se reportan advertencias en caso de mensajes mal formados o incompletos.

Además, se utiliza un bucle persistente (`loop_forever()`) que garantiza que el servidor se mantenga escuchando continuamente el flujo de mensajes publicados, incluso ante reconexiones transitorias. Esta configuración favorece la estabilidad y robustez del sistema en despliegues reales.

El presente capítulo se limita a describir los aspectos vinculados a la comunicación inalámbrica. El diseño completo de clases, organización en el manejo de base de datos y arquitectura de software asociada se desarrolla en detalle en el Capítulo 6.

5.7.6. Reenvío de comandos desde el servidor

Además de publicar detecciones, el Gateway está habilitado para recibir comandos desde el Servidor Central mediante MQTT. Uno de los comandos críticos es el *hard reset*, que puede ser enviado por el servidor hacia un topic reservado (por ejemplo, `/parkwise/{mac_address}/commands/reset`). Al recibirlo, el Gateway reenvía el comando hacia el Dispositivo Parkwise correspondiente mediante BLE, utilizando la característica de escritura definida en el servidor GATT.

Este mecanismo cierra el ciclo de control remoto, permitiendo tanto la recolección de eventos como la gestión remota de los nodos desplegados.

5.8. Hosteo del Servidor Local y Acceso desde la Red

Durante el proceso de instalación y puesta en marcha, se ejecuta una aplicación web sobre una PC o laptop local que actúa como servidor central del sistema Parkwise. Esta decisión técnica respondió a la necesidad de integrar múltiples interfaces (BLE, NB-IoT y web) con flexibilidad y capacidad de depuración. Se optó por una computadora convencional en lugar de una Raspberry Pi u otros dispositivos embebidos debido a su mayor capacidad de procesamiento, conectividad y facilidad de desarrollo. Alternativas como servicios en la nube o infraestructura remota fueron descartadas para esta etapa, priorizando un entorno controlado de red local.

La aplicación está implementada en Python utilizando el framework `Flask`, y su despliegue se realiza mediante el servidor `waitress` en modo multicliente. Para permitir el acceso desde cualquier dispositivo conectado a la misma red Wi-Fi, el servidor se configura con la opción `host=''0.0.0.0''`, habilitando así la navegación mediante la IP del host (por ejemplo, `http://192.168.106.1:5000`).

Desde el punto de vista de las comunicaciones inalámbricas, el servidor actúa como nodo convergente: se lanzan hilos independientes para escuchar detecciones BLE y NB-IoT en paralelo, manteniendo una recepción constante de eventos. Esta arquitectura permite que el sistema funcione en tiempo cuasi-real sin interferir con la interfaz web, que se utiliza para configurar homografías, visualizar detecciones y emitir comandos de reinicio.

En el futuro, podría explorarse el despliegue remoto del sistema, habilitando el acceso a la interfaz desde cualquier localidad mediante un dominio público y protocolos seguros. Para ello, sería necesario implementar mecanismos adicionales como autenticación, control de acceso y configuración de puertos o túneles seguros (por ejemplo, mediante VPN o configuración NAT en el router).

La lógica completa del servidor, su organización modular, y las rutas REST implementadas serán tratadas en profundidad en el Capítulo 6.

5.9. Integración y Flujo Completo

Esta sección ofrece una visión holística del sistema Parkwise, integrando todos los módulos desarrollados desde la inferencia local hasta la visualización de eventos en la página web. El sistema fue concebido como una arquitectura modular en la que cada componente cumple una función específica dentro del flujo completo de datos,

y la interoperabilidad entre ellos se garantiza mediante protocolos estandarizados y buenas prácticas de diseño embebido.

A modo de resumen, el proceso comienza con la instalación de un nuevo Dispositivo Parkwise. Esta instancia se divide en dos etapas. En primer lugar, el operario, desde su computadora, se conecta al nuevo nodo vía BLE y le solicita una captura de imagen del entorno. Al recibir la imagen, el sistema registra localmente tanto un identificador del nodo como la imagen recibida.

Continuando con la segunda etapa de la instalación por parte del operario, este debe acceder a la interfaz web servida por el servidor central. Una vez dentro, debe navegar hasta el apartado de configuración para el nuevo nodo, registra la ubicación geográfica de la cámara, define los límites de la vereda observada, y selecciona los puntos de referencia necesarios para calcular la homografía.

Una vez que el nuevo dispositivo Parkwise está configurado, entra en régimen de funcionamiento continuo (*live inferencing*). En este estado, el dispositivo ejecuta inferencias de forma periódica sobre imágenes capturadas localmente. Cuando se detecta un objeto, se genera un mensaje JSON con las coordenadas y propiedades de la detección. Este mensaje puede ser transmitido por dos vías: vía BLE hacia un Gateway cercano, o directamente vía BLE hacia el servidor central. Esta ultima tiene el propósito de ser una vía secundaria o alternativa. Ambas rutas culminan en el servidor central, donde las detecciones son almacenadas en una base de datos local para su posterior visualización y análisis.

El sistema contempla además mecanismos de resiliencia. En particular, la tarea `reset_timer_task` se encarga de monitorear la actividad de la pila de comunicaciones. Si se detecta un período prolongado de inactividad, se activa un reinicio forzado del dispositivo, permitiendo recuperar el funcionamiento sin requerir intervención humana.

La Figura 5.1 resume gráficamente este flujo completo, desde la instalación inicial hasta la recepción y despliegue de eventos en el sistema central, brindando una representación visual del recorrido de datos en el sistema Parkwise.

5.10. Conclusión del capítulo

El presente capítulo abordó en detalle la arquitectura de comunicaciones inalámbricas del sistema Parkwise, recorriendo todos los vínculos que conectan a los nodos embebidos con el servidor central, pasando por Gateways, redes celulares y enlaces BLE. Cada decisión técnica fue evaluada en función del consumo energético, la cobertura, la confiabilidad y la facilidad de implementación en entornos urbanos reales.

Uno de los principales desafíos enfrentados fue el diseño de mecanismos robustos de transmisión de datos sobre BLE, especialmente para el envío de imágenes. Este obstáculo fue superado mediante una fragmentación eficiente, encabezados personalizados y validación progresiva en el cliente. Asimismo, la falla del Gateway desarrollado en Arduino marcó un punto de inflexión, impulsando una migración hacia una solución más flexible y confiable en PC.

El sistema demostró ser altamente modular: los dispositivos pueden operar en modo autónomo o integrados a una red completa, el servidor puede escalar a múltiples clientes, y los módulos de transmisión pueden conmutar entre BLE y NB-IoT sin cambios en la lógica de negocio.

Entre las posibles mejoras a futuro, se identifican:

- Incorporación de capas de seguridad (TLS en MQTT, autenticación en la interfaz web).
- Optimización del consumo energético mediante BLE advertising más agresivo o modos sleep extendidos.

- Despliegue en la nube con acceso remoto al servidor desde cualquier localidad.
- Separación del módulo del operario y el servidor central, manteniendo una base de datos compartida mediante sincronización inalámbrica. Este cambio permitiría desplegar múltiples instancias simultaneas de instalación remotas y colaborativas con mínimos ajustes al código actual.

En conjunto, el sistema alcanza los objetivos de diseño propuestos, demostrando que es posible integrar visión embebida, comunicaciones inalámbricas y configuración geoespacial en una arquitectura liviana y extensible para el monitoreo urbano descentralizado.

Capítulo 6

Desarrollo e Implementación del Software de Alto Nivel

6.1. Introducción

El software desarrollado en el marco del proyecto Parkwise cumple una función esencial como nexo entre los distintos módulos embebidos y el usuario final, tanto durante la instalación como en la operación continua del sistema. Su diseño combina técnicas de desarrollo web, comunicaciones inalámbricas y gestión de datos, integrando funcionalidades de configuración, monitoreo y control en una única interfaz accesible desde cualquier dispositivo conectado a la red local.

El objetivo de este capítulo es presentar en detalle el diseño, la implementación y la arquitectura del sistema de software de alto nivel que respalda a Parkwise. Se abordan los aspectos relativos a las buenas prácticas de ingeniería de software, incluyendo modularidad, control de versiones, programación orientada a objetos, y uso de patrones arquitectónicos y herramientas gráficas típicas como diagramas de Lenguaje Unificado de Modelado (UML) y diagramas de secuencia.

6.1.1. Propósito del Software desarrollado

El sistema de software desarrollado funciona como la principal interfaz de interacción entre el sistema Parkwise y sus usuarios, ya sean operarios técnicos o supervisores remotos. Cumple dos roles fundamentales:

- Durante el **modo de instalación (setup)**, permite configurar cada nodo Parkwise de forma individual: obtener una imagen de referencia, definir los puntos para el cálculo de homografía y registrar su ubicación en el mapa.
- En el **modo de live-inferencing (régimen)**, presenta detecciones vehiculares georreferenciadas en tiempo real sobre un mapa, permite forzar reinicios remotos a los nodos y visualizar información técnica relevante.

Desde el punto de vista funcional, el software actúa como *frontend* y *backend* del sistema, gestionando tanto la visualización en la interfaz web como las tareas de procesamiento, almacenamiento y retransmisión de datos. Internamente, también cumple el rol de servidor local y coordinador de los hilos dedicados a la recepción de datos vía NB-IoT (medio principal) y BLE (medio alternativo).

En la práctica, esta aplicación es la única parte visible para el operario o cliente final. A través de ella se accede a todas las funcionalidades críticas del sistema de forma centralizada y visual, desde cualquier dispositivo con navegador compatible.

Figura 6.1: Vista general de la interfaz web en modo live-inferencing. Se visualizan los controles de detecciones, botones de configuración y un mapa con nodos activos.

6.1.2. Modos de funcionamiento y Guía de Uso de la Interfaz

Como se mencionó, el sistema Parkwise puede ser utilizado en dos modos claramente diferenciados: modo de configuración (setup) y modo live-inferencing (régimen). Ambos se articulan a través de la misma interfaz web, adaptándose al estado del sistema y al rol del usuario (operario o supervisor). Esta dualidad funcional permite facilitar tanto la instalación inicial de los nodos como su monitoreo continuo.

Interfaz en el setup por BLE del operario

La instalación inicial de los nodos Parkwise por parte del operario se divide en dos etapas sucesivas: detección e incorporación del dispositivo, seguida de su configuración geográfica.

1. Detección e incorporación automática del nodo

En primer lugar, el operario ejecuta en la terminal el siguiente comando para iniciar el proceso de escaneo y solicitud de imágenes:

```
python .\operario.py
```

Este script se encarga de detectar, vía BLE, los dispositivos cercanos cuyo nombre o UUID identifica al nodo como un *"Parkwise Device"*. A cada uno se le solicita automáticamente una imagen de referencia, la cual se reconstruye, se guarda localmente y se registra en la base de datos junto con la dirección MAC del dispositivo. Este procedimiento no requiere interacción manual y garantiza que todos los nodos nuevos queden correctamente ingresados en el sistema.

2. Configuración geográfica del nodo

Para pasar a la siguiente etapa es necesario que esté en ejecución la aplicación principal del servidor. La misma ejecutando con el siguiente comando en terminal:

```
python .\app.py
```

Al acceder al navegador, se carga una interfaz web donde se visualiza un mapa interactivo. En él aparecen los nodos registrados ubicados en una posición por defecto si aún no ha sido configurado. Al hacer clic sobre el nodo deseado, se lo selecciona, y mediante el botón `Configure selected ESP` se accede a la **interfaz de configuración** que permite asignarle sus coordenadas reales mediante una correspondencia visual entre la imagen capturada y el mapa.

La Figura 6.2 muestra esta interfaz en acción. A la izquierda se visualiza la imagen capturada por el Dipositivo Parkwise, en escala de grises, donde el operario debe reubicar cuatro puntos clave. Estos puntos están marcados con colores (azul, rojo, verde y naranja), y sus coordenadas en píxeles se muestran en la parte inferior izquierda. A la derecha se encuentra la vista del mapa, donde los mismos cuatro puntos deben ser ubicados geográficamente arrastrándolos interactivamente. Para lograr una buena correspondencia entre ambas vistas, se recomienda seleccionar puntos de referencia bien definidos sobre el plano del suelo —como árboles, columnas o esquinas de pavimento— que puedan ayudar a localizarse con claridad tanto en la imagen como en el mapa. Es importante que estos puntos no estén alineados entre sí y que idealmente conformen un cuadrilátero amplio.

Aunque los puntos se colocan sobre un mapa, se recomienda usar la vista satelital con el mayor nivel de zoom posible para ubicar los objetos con mayor precisión. Incluso puede ser útil abrir "Google Street View" para verificar los detalles del entorno que ayuden a elegir bien los puntos.

Figura 6.2: Interfaz de configuración de un nodo Parkwise. A la izquierda, imagen capturada con selección de puntos clave. A la derecha, vista de mapa con asignación geográfica de los mismos puntos.

Homografía como solución geométrica

Para resolver el problema de alineación entre la perspectiva de la cámara y el mapa georreferenciado, se utiliza una homografía, que es una transformación proyectiva representada mediante una matriz 3×3. Esta matriz se calcula a partir de los cuatro pares de puntos seleccionados (en píxeles y coordenadas geográficas) y permite transformar cualquier punto dentro de la imagen en una ubicación real sobre el mapa.

Por ello se enfatiza la importancia de registrar pares de puntos con la mayor correspondencia posible.

Si bien la aplicación actual utiliza exactamente cuatro pares de puntos, en teoría podrían usarse más para obtener una homografía más precisa.

Este procedimiento es fundamental para garantizar que las detecciones de vehículos futuras puedan ser proyectadas con precisión sobre las veredas y calles. El botón en color verde `Save Configuration` de la figura 6.2 guarda estos valores en la base de datos local y deja el nodo habilitado para operar en modo régimen.

Interfaz en uso normal (régimen)

Una vez que los Dispositivos Parkwise han sido correctamente configurados, el sistema entra en su modo live-inferencing habitual, denominado **modo régimen**. En este estado, los nodos comienzan a enviar detecciones vehiculares en tiempo real, las cuales son proyectadas geográficamente mediante la homografía definida durante la etapa de setup. Estas detecciones pueden recibirse tanto por BLE como por NB-IoT, y se visualizan automáticamente sobre un mapa interactivo en la interfaz web.

Figura 6.3: Vista inicial en modo régimen. Se muestra el mapa de la ciudad con los Dispositivos Parkwise desplegados y el panel lateral con controles de operación.

La Figura 6.3 muestra la vista general del sistema en funcionamiento. A la derecha se encuentra el panel de control, que permite:

- Recargar las detecciones (`Reload Detections`).
- Ocultar temporalmente las detecciones (`Hide Detections`).
- Acceder al menú de configuración de un nodo (`Configure selected ESP`).
- Enviar comandos de reinicio forzado a un nodo (`Reset MAC`).

Al seleccionar un nodo sobre el mapa, se despliega una vista ampliada con información adicional, como se muestra en la Figura 6.4. Esta incluye su dirección MAC, la ubicación registrada de la cámara, los puntos de homografía y los segmentos verdes que representan la vereda configurada. Esta herramienta resulta útil para validar visualmente que el nodo esté correctamente alineado con su entorno físico.

Una vez cargadas las detecciones con sus coordenadas geográficas desde la base de datos, la interfaz despliega cada evento detectado como una silueta de vehículo. Y estas detecciones son proyectadas ortogonalmente sobre el segmento de vereda más cercano. La Figura 6.5 ilustra este resultado final, donde las líneas rojas simulan la longitud de un vehículo genérico estacionado, orientado paralelo a la acera.

Este modo operativo representa la funcionalidad central del sistema Parkwise. Está diseñado con una interfaz clara y amigable para facilitar el monitoreo a usuarios no técnicos, como encargados de tránsito, supervisores municipales o responsables de instalaciones.

Figura 6.4: Detalle de un nodo seleccionado. Se visualizan los puntos de homografía y la vereda proyectada en verde sobre el mapa.

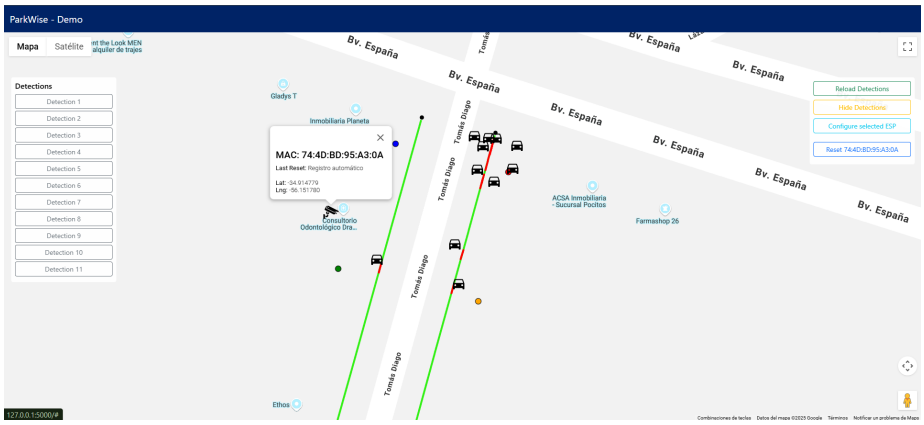


Figura 6.5: Visualización completa de detecciones proyectadas. Cada evento se representa como un vehículo posicionado ortogonalmente a la vereda más cercana.

6.1.3. Metodología de desarrollo

El desarrollo del software de alto nivel para Parkwise se abordó con un enfoque ágil y modular, priorizando la iteración rápida, la validación en campo y la integración temprana de funcionalidades mínimas viables. Se partió de un esquema de desarrollo incremental, donde cada módulo —interfaz web, comunicación BLE, comunicación MQTT, procesamiento de detecciones, persistencia de datos, entre otros— fue construido, probado y conectado de manera progresiva.

El sistema fue inicialmente concebido como una solución distribuida basada en edge computing, capaz de operar sin conexión a servicios en la nube. Esto facilitó su despliegue local y su funcionamiento en entornos con conectividad limitada o intermitente, lo que representa una ventaja clave para aplicaciones urbanas reales.

Las pruebas se realizaron tanto de forma manual (validación visual y pruebas funcionales desde la interfaz web) como automática mediante registros, impresiones de estado y simulaciones de entrada/salida controlada. Estas sesiones de prueba se llevaron a cabo principalmente en entornos reales (barrios residenciales, entornos urbanos,

estacionamientos), permitiendo verificar la robustez del sistema frente a condiciones variables de red, errores de sensado o reinicios inesperados de los nodos. Este enfoque de validación directa permitió mantener un ciclo de iteración corto entre prueba, diagnóstico y corrección.

A nivel de gestión, se utilizó Git como sistema de control de versiones, tanto para el backend en Python como para los scripts de frontend en JavaScript. Esto permitió mantener trazabilidad de cambios, organizar ramas para pruebas o funciones experimentales, y facilitar la recuperación ante errores sin perder estabilidad en las versiones operativas.

La arquitectura del software fue dividida en módulos independientes con responsabilidades bien definidas: captura de datos, almacenamiento, visualización y configuración. Esta decisión buscó facilitar la depuración, el mantenimiento y la futura escalabilidad del sistema, permitiendo, por ejemplo, reemplazar la interfaz web o el módulo de comunicación sin alterar la lógica de base.

6.1.4. Tecnologías utilizadas

El sistema fue desarrollado utilizando una combinación de tecnologías ligeras, portables y compatibles con dispositivos embebidos, priorizando la simplicidad de despliegue, la depuración en tiempo real y la posibilidad de extender sus capacidades en el futuro.

- **Backend (Python)**:
  - `Flask`: framework principal para servir la interfaz web, renderizar plantillas HTML (Jinja) y exponer APIs REST.
  - `Waitress`: servidor WSGI robusto utilizado para el despliegue local en producción.
  - `asyncio` y `threading`: para ejecutar tareas concurrentes de recepción BLE y NB-IoT sin bloquear el hilo del servidor.
  - `dotenv`: para la gestión de claves y variables sensibles.
  - `NumPy`, `OpenCV`, `PIL`: para operaciones de homografía, visualización y procesamiento de imágenes.
- **Comunicación inalámbrica**:
  - `Bleak`: cliente BLE para la recepción de imágenes y bounding boxes desde los Dispositivos Parkwise.
  - `paho-mqtt`: biblioteca para manejo de tópicos MQTT (detecciones y comandos).
  - `Mosquitto`: broker MQTT utilizado tanto en modo local como remoto.
- **Frontend (Web)**:
  - `HTML`, `JavaScript` y `CSS`: estructura, lógica e interacción desde el navegador.
  - `Google Maps API`: visualización georreferenciada de dispositivos, detecciones y proyecciones.
  - `Bootstrap`: estilos visuales responsivos y estructura de componentes reutilizables.
  - `SVG` y `Canvas`: para representación de íconos, polígonos y líneas sobre el mapa.

- **Base de datos**:
  - `SQLite3`: motor embebido para almacenamiento estructurado de configuraciones, detecciones y parámetros de cada Dispositivo Parkwise.

Este conjunto tecnológico fue seleccionado para lograr una solución liviana, funcional y fácilmente adaptable a nuevas funcionalidades. La estructura modular de la aplicación permite migrar hacia una arquitectura cliente-servidor distribuida o integrar servicios en la nube en caso de escalar a un entorno de producción más amplio.

6.2. Arquitectura General del Sistema

6.2.1. Diagrama de bloques funcionales

La Figura 6.6 presenta una visión general del sistema Parkwise desde el punto de vista del software de alto nivel. El diagrama expone los principales componentes y su interacción, incluyendo las capas de frontend y backend, los mecanismos de comunicación inalámbrica, la base de datos local y el acceso del usuario final a través de un navegador web.

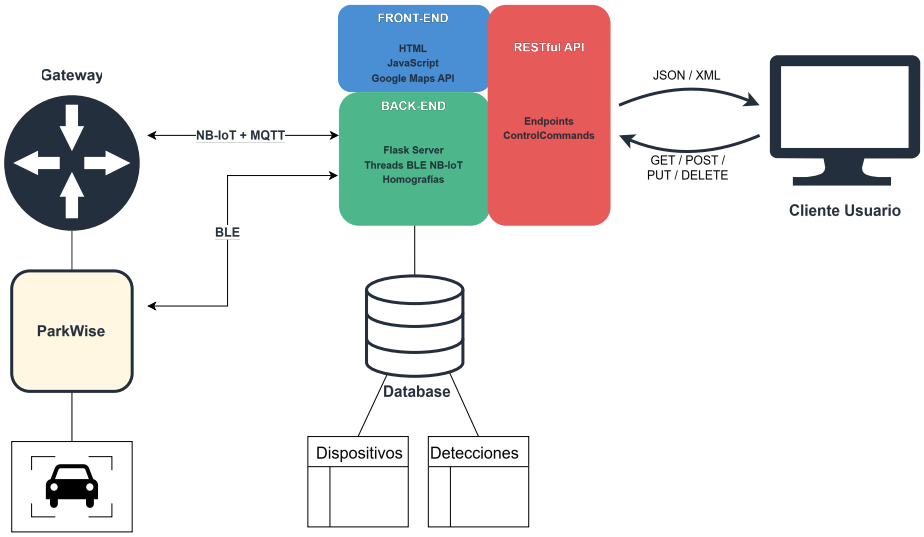


Figura 6.6

Una API RESTful (Representational State Transfer) es un estilo de arquitectura para diseñar servicios web. Se basa en la exposición de recursos identificables mediante URLs y manipulables a través de endpoints, que responden a verbos del protocolo HTTP como GET, POST, PUT y DELETE. Cada recurso representa una entidad del sistema (por ejemplo, una detección o un dispositivo), y las operaciones permiten leer, modificar o eliminar su estado. Estas interacciones suelen codificarse en formatos ligeros como JSON o XML, facilitando la interoperabilidad entre sistemas y lenguajes distintos.

En el extremo izquierdo, el Dispositivo Parkwise se comunica vía BLE con un Gateway para transmitirle las detecciones, el cual retransmite las detecciones al Servidor Central utilizando la red NB-IoT (mediante el mencionado broker MQTT Mosquitto).

El servidor corre sobre un backend construido en Python con el framework Flask, encargado de gestionar tanto la lógica web como las interfaces de comunicación con los dispositivos y la base de datos. La arquitectura incluye módulos concurrentes (mediante hilos y asyncio) para escuchar detecciones en tiempo real por ambas vías (BLE y NB-IoT), procesarlas y almacenarlas en una base de datos embebida (SQLite).

A través de una API RESTful, el servidor expone endpoints para enviar comandos, cargar configuraciones y acceder a las detecciones. Estos endpoints son consumidos por un frontend web implementado en HTML y JavaScript, con integración a Google Maps API para visualizar geográficamente las detecciones, dispositivos y zonas configuradas.

Finalmente, desde el navegador, el usuario puede acceder a la interfaz gráfica, reiniciar dispositivos, configurar puntos de homografía y monitorear la actividad del sistema.

6.2.2. Módulos principales: Frontend, Backend, Comunicaciones, Base de Datos

El sistema se estructura en torno a cuatro grandes módulos funcionales, reflejados en la siguiente estructura de archivos del proyecto:

```
C:.
|   app.py
|   config.py
|   database.py
|   mock_data.py
|   parkwise.db
|   save_db_as_csv.py
|   wireless_comm.py
|
|
|---static
|   |   ._images
|   |
|   |---imagenes_recibidas
|   |
|   |---images
|           camera_icon.png
|           car_icon.png
|
|---templates
        index.html
```

Backend (app.py, database.py, config.py): es el núcleo del sistema, construido en Python con Flask. Define los endpoints HTTP de la API, maneja la lógica de negocio, coordina los hilos de recepción BLE y NB-IoT, y administra el ciclo completo de las detecciones (recepción, transformación, persistencia y visualización). También expone funciones auxiliares como el reseteo de dispositivos o la configuración de homografías.

Frontend (templates/index.html + static/): se sirve desde Flask mediante el patrón de templates. Utiliza HTML, JavaScript, Bootstrap y la API de Google Maps para ofrecer una interfaz gráfica interactiva. El mapa permite visualizar las detecciones en tiempo real, junto con sus proyecciones, configuraciones asociadas y controles remotos.

Comunicaciones (wireless_comm.py): agrupa las funciones de comunicación con los Dispositivos Parkwise. Se encarga de establecer conexiones BLE para la recepción de imágenes y bounding boxes, así como de suscribirse a un broker MQTT para recibir mensajes desde Gateways NB-IoT. Los datos entrantes son manejados

mediante callbacks asincrónicos, permitiendo al servidor procesar múltiples fuentes simultáneamente.

Base de datos (parkwise.db): es una base de datos SQLite local que almacena configuraciones de cada dispositivo (MAC, homografía, ubicación, etc.) y registros históricos de detecciones. Su integración es directa mediante el módulo `sqlite3`, y las operaciones de lectura/escritura son coordinadas por funciones del archivo `database.py`.

Esta modularidad buscó mantener bajo acoplamiento entre componentes y facilitar tanto la depuración como la escalabilidad futura. Cada módulo puede evolucionar de manera independiente (por ejemplo, reemplazar la interfaz web, cambiar el backend a FastAPI o migrar la base de datos a PostgreSQL) sin alterar la lógica general del sistema.

6.2.3. Explicación del Flujo General de datos y responsabilidades

El flujo de datos en Parkwise está orientado a eventos, y combina comunicaciones inalámbricas, almacenamiento local y visualización web sincronizada. En la Figura 6.6 se representa esquemáticamente este recorrido, que puede dividirse en seis etapas principales:

1. Setup y solicitud de imagen: En el mencionado modo de setup, el servidor hace un escaneo por BLE de todos los dispositivos, y a cada dispositvo Parkwise encontrado se le solicita una imagen. Estas son guardadas en una carpeta, con una extension png y cada archivo lleva como nombre la direccion MAC del Parkwise del cual fue enviado. Seguido a eso, se guarda cada dispostivo Parkwise de los encontrados en la tabla de dispostivos de la base de datos SQLite regitrandolos con su direccion MAC.

2. Almacenamiento en la base de datos: Con cada recepcion de detecciones, el servidor registra los datos recibidos en una base de datos SQLite. Cada detección queda vinculada a su dispositivo correspondiente mediante la MAC. Todos los datos estructurados (detecciones, configuraciones de ESP32, homografías, timestamps, etc.) se almacenan en esta base de datos local. Funciona como fuente de verdad del sistema y es accedida tanto por el backend como por los endpoints REST.

3. Procesamiento y transformación: Se recuerda que en la pantalla de setup el instalador puede configurar la homografia para la transformacion de puntos. En caso que el dispositivo tiene definida una matriz de homografía válida, el sistema transforma las coordenadas del centroide detectado en píxeles a coordenadas geográficas utilizando dicha matriz, y guarda los puntos transformados en la base de datos. Esta información es escencial para representaciones georreferenciadas en el mapa.

4. Carga y visualización web: Cuando un usuario accede al sistema vía navegador, el frontend consulta al backend por:

- Coordenadas geográficas y homografías de los dispositivos.
- Detecciones ya proyectadas.
- Archivos de imagen asociados.

Todos estos datos son servidos por la RESTful API y renderizados con Google Maps API en una interfaz visual.

5. Rol del frontend: El frontend actúa como interfaz interactiva entre el usuario y el sistema. Permite:

- Visualizar en el mapa los dispositivos y las detecciones georreferenciadas.
- Realizar acciones sobre los nodos (ej. enviar un reset).

- Reconfigurar homografías y zonas de vereda, o en su defecto configurarlas por primera vez.
- Acceder a imágenes, visualizar bounding boxes o simular detecciones.

Este diseño de flujo modular y asincrónico permite robustez ante errores, fácil trazabilidad, y posibilidad de incorporar múltiples dispositivos simultáneamente, sin necesidad de interrupciones en el sistema.

6.3. Base de Datos Local (parkwise.db)

6.3.1. Estructura de la base `parkwise.db` (tablas, claves foráneas)

La base de datos local `parkwise.db`, implementada mediante SQLite, actúa como repositorio central para todos los datos relacionados con el sistema: detecciones entrantes, configuración de dispositivos y parámetros de homografía. Está compuesta por dos tablas principales:
Tabla 1: Devices Esta tabla almacena los dispositivos registrados, su configuración georreferenciada y su historial de actividad.

Campo	Tipo	Descripción
id	INTEGER PRIMARY KEY	Identificador único interno
DVC_MAC_ADDRESS	TEXT UNIQUE	Dirección MAC del dispositivo
DVC_HOMOGRAFÍA	TEXT (JSON)	Matriz de homografía 3×3
DVC_HOMOGRAFIA_PUNTOS_PIXELES	TEXT (JSON)	Lista de puntos en píxeles
DVC_HOMOGRAFIA_PUNTOS_GEO	TEXT (JSON)	Lista de puntos geográficos
DVC_COORDENADAS_GEO_CAMARA	TEXT (JSON)	Ubicación GPS de la cámara
DVC_COORDENADAS_GEO_VEREDA	TEXT (JSON)	Cuatro vértices de vereda visible
DVC_LAST_RESET	TEXT	Último tipo de reset aplicado
DVC_LAST_SEEK_TIME_STAMP	TEXT	Último timestamp de búsqueda o actividad

Tabla 2: Detection Contiene las detecciones realizadas por los Dispositivos Parkwise, almacenadas con su MAC de origen, el centroide en píxeles, la proyección geográfica (si aplica) y un timestamp.

Campo	Tipo	Descripción
id	INTEGER PRIMARY KEY	Identificador único interno
DET_MAC_ADDRESS	TEXT	Dirección MAC del dispositivo que detectó
DET_DETECCIONES_PIXEL	TEXT (JSON)	Coordenadas del centroide en píxeles
DET_DETECCIONES_GEO	TEXT (JSON)	Coordenadas proyectadas mediante homografía
DET_TIME_STAMP	TEXT	Marca de tiempo (UTC) de la detección

Relaciones Actualmente, la base no define claves foráneas explícitas, ya que SQLite no las impone por defecto a menos que se habiliten. Sin embargo, de forma implícita, cada entrada en la tabla `Detection` está vinculada a un dispositivo de la tabla `Devices` mediante su dirección MAC. Esta clave actúa como nexo para la trazabilidad entre detecciones y configuraciones de origen.

En la siguiente subsección se presentará el esquema relacional en formato visual para facilitar su interpretación.

6.3.2. Esquema Entidad-Relación

El diseño de la base de datos `parkwise.db` se estructura en torno a dos entidades principales: `Devices` y `Detection`. La Figura 6.7 ilustra el esquema entidad-relación (ER) entre ambas, donde puede observarse que cada dispositivo ESP32 puede asociarse con múltiples detecciones (relación 1:N implícita).

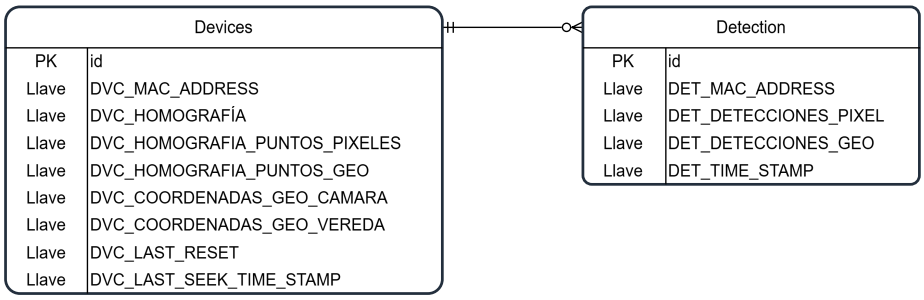


Figura 6.7: Esquema ER simplificado de la base de datos `parkwise.db`

Es importante aclarar que esta relación entre tablas no está implementada mediante claves foráneas explícitas en el código. Esto se debe a que SQLite no fuerza integridad referencial por defecto y, en este sistema, la coherencia se mantiene de forma lógica mediante el campo `MAC_ADDRESS` compartido por ambas tablas. La cardinalidad se representa únicamente con fines ilustrativos.

6.3.3. Módulo de acceso: consultas, inserciones y actualizaciones

Uso en inicializacion

Al iniciar el sistema, se verifica automáticamente si la base de datos `parkwise.db` existe. En caso contrario, se crea utilizando el módulo `init_db()`, encargado de inicializar las tablas `Devices` y `Detection`. Esto asegura que el sistema esté listo para operar sin intervención manual.

Durante el flujo de trabajo, los Dispositivos Parkwise se registran en la tabla `Devices` al momento de enviar por primera vez una imagen. En esa inserción inicial, únicamente se completa el campo `DVC_MAC_ADDRESS`, y el resto de los atributos se inicializa en `NULL`. Posteriormente, durante el proceso de configuración (modo setup) desde la interfaz web, se actualizan los siguientes campos:

- `id` (autoincremental)
- `DVC_HOMOGRAFIA_PUNTOS_PIXELES`
- `DVC_HOMOGRAFIA_PUNTOS_GEO`
- `DVC_COORDENADAS_GEO_CAMARA`
- `DVC_COORDENADAS_GEO_VEREDA`
- `DVC_LAST_RESET`
- `DVC_LAST_SEEK_TIME_STAMP`

Una vez finalizada la configuración al cliquear el botón “Save Configuration”, se corre un proceso donde se recalculan las matrices de homografía (`DVC_HOMOGRAFÍA`) para todos los dispositivos que cuenten con ambos conjuntos de puntos (en píxeles y coordenadas geográficas). Estas matrices se almacenan en la base de datos para permitir la transformación de futuras detecciones.

Uso normal (en régimen)

Cada vez que se recibe una nueva detección, se inserta una fila en la tabla `Detection`, con los siguientes campos:

- `id` (autoincremental)
- `DET_MAC_ADDRESS`
- `DET_DETECCIONES_PIXEL`
- `DET_DETECCIONES_GEO = NULL`
- `DET_TIME_STAMP`

La transformación de las detecciones a coordenadas geográficas se realiza de forma diferida. Al presionar el botón `Reload Detections` desde la interfaz web, se recorre la tabla `Detection` y se identifican aquellas entradas donde `DET_DETECCIONES_GEO` está vacío. Para cada una de ellas, se consulta la matriz de homografía correspondiente en la tabla `Devices`, utilizando la dirección MAC como llave, y se realiza la proyección. Los valores resultantes se almacenan en la base de datos y se actualiza la interfaz.

Cabe destacar que esta actualización de detecciones se da también de forma automática, siendo llamada al final de cada ciclo de un thread del cual se hablara en la sección 6.4.2.

El flujo completo de inserción, transformación y visualización de datos puede verse resumido en la Figura 6.8.

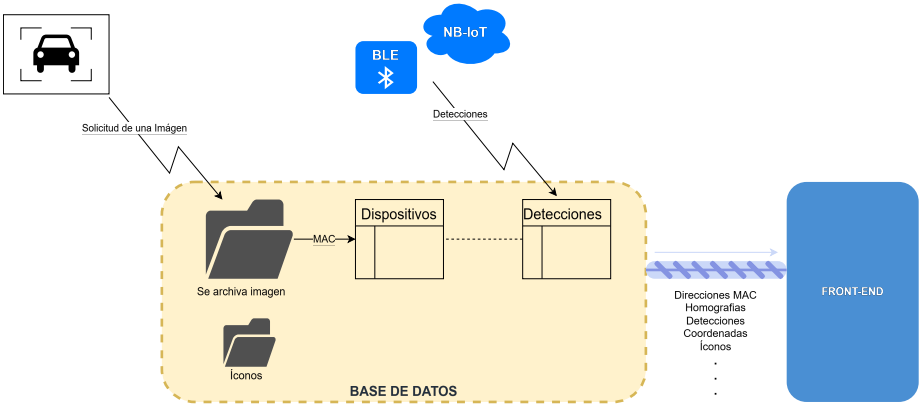


Figura 6.8: Flujo general de datos y responsabilidades del sistema

6.3.4. Consideraciones de Acceso Remoto, Persistencia, Sincronización y posibles mejoras

El sistema actual utiliza `SQLite3` como motor de base de datos local, ubicado en el mismo entorno físico que el software del operario. Esta decisión permitió centralizar la lógica del sistema en un único proyecto, facilitando las pruebas en campo y reduciendo complejidad durante la etapa de desarrollo. Tanto el servidor Flask que sirve la interfaz web como los scripts utilizados por el operario acceden directamente a la base `parkwise.db` mediante una estructura común de acceso como:

```
conn = get_db()
c = conn.cursor()
c.execute(...)
conn.commit()
conn.close()
```

Siendo `get_db()` una función que establece la conexión usando SQLite3:

```
def get_db():
    conn = sqlite3.connect(DATABASE_PATH)
    conn.row_factory = sqlite3.Row
    return conn
```

Esta implementación no solo simplifica la gestión del sistema local, sino que también permite una futura modularización sencilla. Si la base de datos se alojara en un servidor remoto, no sería necesario modificar la lógica de alto nivel del sistema: bastaría con ajustar la función de conexión `get_db()` para apuntar a un motor remoto (por ejemplo, PostgreSQL), manteniendo inalteradas las funciones de consulta, inserción o actualización.

La separación entre los módulos del *operario* y el *servidor central* podría entonces reflejarse también a nivel de proyecto, permitiendo dividir el código en dos componentes independientes que se comunican a través de una base de datos accesible por red. Esto permitiría, por ejemplo, que el operario actúe desde una tablet conectada al mismo servidor que ejecuta la interfaz web y almacena la información de configuración y detecciones, sin necesidad de compartir directorios locales ni sincronizar archivos.

Además de facilitar la modularización, esta arquitectura abriría la puerta a la escalabilidad, ya que múltiples Gateways podrían enviar sus datos a una misma base remota. En ese contexto, se vuelve pertinente considerar la migración hacia un motor más robusto como `PostgreSQL`, que soporta:

- Acceso concurrente por múltiples procesos o dispositivos.
- Gestión de usuarios, permisos y autenticación avanzada.
- Replicación, respaldo automático y alta disponibilidad.
- Integración con servicios en la nube o entornos distribuidos.

En resumen, el diseño actual del sistema fue pensado con modularidad y portabilidad en mente: aunque actualmente opera en modo local, se encuentra preparado para una transición natural hacia una arquitectura remota y escalable, minimizando cambios estructurales en el código existente.

6.4. Backend: Lógica del Servidor Central (app.py)

6.4.1. Descripción de módulos clave: Flask app, manejo de rutas, persistencia

El archivo `app.py` actúa como punto de entrada principal del sistema Parkwise. Dentro de él se inicializan todos los componentes críticos del software, incluyendo la lógica del servidor, los procesos de recepción de datos y la interfaz web.

El módulo utiliza el *framework* `Flask` para levantar un servidor local capaz de atender peticiones HTTP. El método `@app.route('/')` responde con la plantilla principal del frontend (`index.html`), cargando dinámicamente los datos desde la base local. Además, se definen rutas auxiliares para:

- `/api/detections`: actualiza y expone las detecciones procesadas.
- `/reset/<mac>`: envía comandos de reinicio a dispositivos específicos.
- `/api/save_config`, `/api/get_config/<mac>`: gestionan el almacenamiento de configuraciones de cada dispositivo.

Para la persistencia de datos, se utiliza SQLite mediante funciones auxiliares como `get_db()`, que aseguran la conexión, ejecución y cierre de cada transacción. La lógica para insertar, actualizar o recuperar información se encuentra distribuida en funciones dentro del mismo archivo y en el módulo externo `database.py`.

La *app* es servida a través del servidor WSGI `Waitress`, lo que permite un despliegue eficiente y estable en el entorno local o en pruebas de campo. Su arquitectura modular y desacoplada permite una futura migración a entornos más robustos sin alterar la estructura general del sistema.

6.4.2. Diagrama de Secuencia de Sistema (DSS) y casos de uso principales

Uno de los aspectos más importantes del servidor central desarrollado en `app.py` es su capacidad para gestionar, en paralelo, la recepción de datos desde diferentes tecnologías de comunicación inalámbrica: BLE y NB-IoT. Esto se logra mediante la creación de hilos de ejecución independientes, uno para cada tipo de enlace.

- El hilo dedicado a BLE es iniciado mediante la función `start_ble_loop()`, la cual ejecuta un bucle asincrónico que escanea dispositivos ESP32 en busca de notificaciones de `bounding boxes`. Estas detecciones son recibidas, procesadas y almacenadas en la base de datos. El mismo hilo también invoca periódicamente a la función `api_get_detections()` para actualizar los puntos geográficos aplicando la homografía.
- De manera análoga, el hilo `start_nb_iot_loop()` gestiona la recepción de detecciones enviadas a través de la red NB-IoT. Aunque este canal aún se encuentra en fase de pruebas, su estructura está preparada para integrarse sin fricciones al flujo general del sistema.

Ambos hilos son lanzados desde el bloque principal `if __name__ == '__main__'` mediante llamadas a `threading.Thread(...).start()`, asegurando que sus bucles puedan ejecutarse de forma no bloqueante en paralelo al servidor web. Además, al ejecutarse dentro del contexto de aplicación de Flask (`with app.app_context():`), se garantiza que el acceso a la base de datos y las funciones del servidor se mantenga estable y coherente entre hilos concurrentes.

Esta estrategia permitió lograr un sistema verdaderamente reactivo, capaz de recibir datos entrantes sin necesidad de reinicios manuales ni bloqueo del servidor web.

6.4.3. Diagrama de secuencia y casos de uso principales

El backend de Parkwise responde a una arquitectura cliente-servidor basada en Flask, estructurada en torno a operaciones RESTful y controlada por eventos en tiempo real provenientes de los hilos de comunicación BLE y NB-IoT. La Figura 6.9 ilustra los cuatro casos de uso centrales que permiten la operación básica del sistema:

- **Carga inicial de datos (GET /):** al ingresar al sistema, el usuario accede a la ruta raíz del servidor, donde se consulta la base de datos para obtener la lista de dispositivos ESP32, las detecciones previas, y las imágenes almacenadas localmente. Esta información se combina y se renderiza en el front-end mediante una plantilla HTML enriquecida con datos en formato JSON.
- **Configuración de un nuevo nodo (GET /api/get_config + POST /api/save_config):** al seleccionar una ESP32 desde el mapa y acceder al modo de configuración, se recuperan los parámetros actuales del nodo desde la base de datos. El usuario puede editar los puntos de homografía, ubicación de la cámara

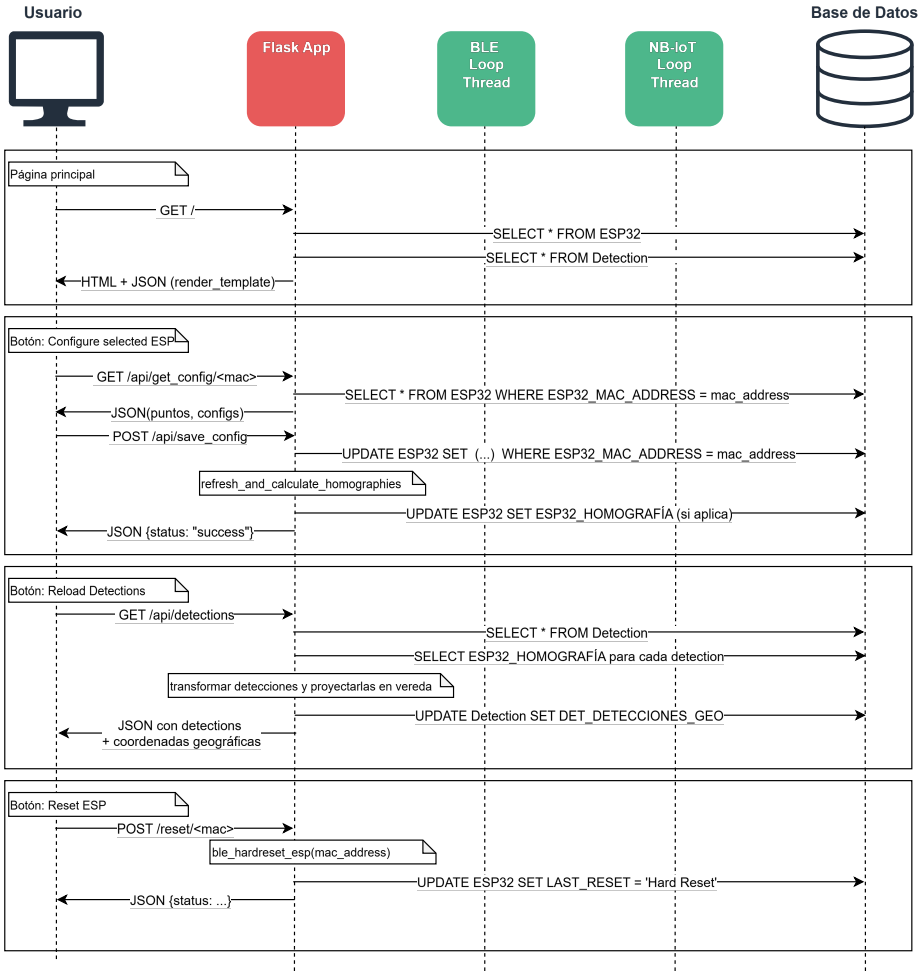


Figura 6.9: Diagrama de secuencia de sistema (DSS) para los principales casos de uso del backend de Parkwise

y puntos de vereda. Al guardar, el sistema actualiza los registros correspondientes y recalcula automáticamente la matriz de homografía, asegurando que las nuevas detecciones puedan ser proyectadas correctamente.

- **Recarga de detecciones proyectadas (GET /api/detections):** esta operación permite obtener las detecciones existentes con sus coordenadas geográficas transformadas mediante homografía. Se consulta la base de datos, se aplica la transformación, se actualizan los valores en cada registro y se devuelven los datos listos para su visualización y proyección sobre las veredas correspondientes.
- **Comando de reinicio remoto (POST /reset/¡mac¿):** al presionar el botón de reinicio, se ejecuta una rutina BLE que envía un comando al dispositivo correspondiente. Luego se actualiza su estado en la base de datos, marcándolo como reiniciado. Este mecanismo es útil para recuperar nodos inactivos o fuera de sincronía.

Este conjunto de casos de uso constituye el núcleo funcional del sistema, alineado

con el concepto de producto mínimo viable (MVP). Su diseño busca garantizar una operación autónoma, confiable y fácilmente replicable en entornos reales.

6.5. Frontend Web (index.html)

6.5.1. Estructura del `index.html` y componentes

El archivo `index.html` constituye el punto de entrada de la interfaz web de Parkwise. Su estructura está organizada en tres capas principales: la cabecera (`<head>`), el cuerpo (`<body>`) y los scripts (`<script>`), todos diseñados para trabajar de forma fluida con el servidor Flask y la base de datos local.

El cuerpo principal incluye:

- **Navbar superior:** barra fija en la parte superior con el título de la aplicación.
- **Mapa central:** visualización de Google Maps que ocupa toda la pantalla y constituye la vista principal.
- **Tres paneles superpuestos sobre el mapa:**
  - **Panel de controles (derecha):** contiene botones para reiniciar nodos, recargar u ocultar detecciones, y activar el modo de configuración.
  - **Panel de detecciones (izquierda):** muestra una lista de detecciones activas con botones individuales para centrar la vista en cada una.
  - **Panel de configuración (oculto por defecto):** permite al operario ajustar manualmente los puntos de homografía, la posición de la cámara y los vértices de la vereda. Está dividido en dos vistas: una vista de imagen (canvas interactivo) y una vista de mapa (Google Maps).

El diseño visual se define mediante estilos CSS internos, que regulan colores, tamaños y comportamiento responsivo. Finalmente, una serie de scripts en JavaScript gestionan la lógica de interacción con el usuario, el control del mapa y la comunicación asíncrona con el servidor a través de llamadas `fetch()` al backend Flask.

6.5.2. Diagrama de dependencias de JavaScript

El archivo `index.html` incluye una única sección de `<script>` al final del documento, donde se centraliza toda la lógica de interacción del frontend con el usuario, el mapa y el servidor. El código JavaScript se organiza de manera modular y temática, pero no utiliza un sistema formal de módulos ni separa archivos.

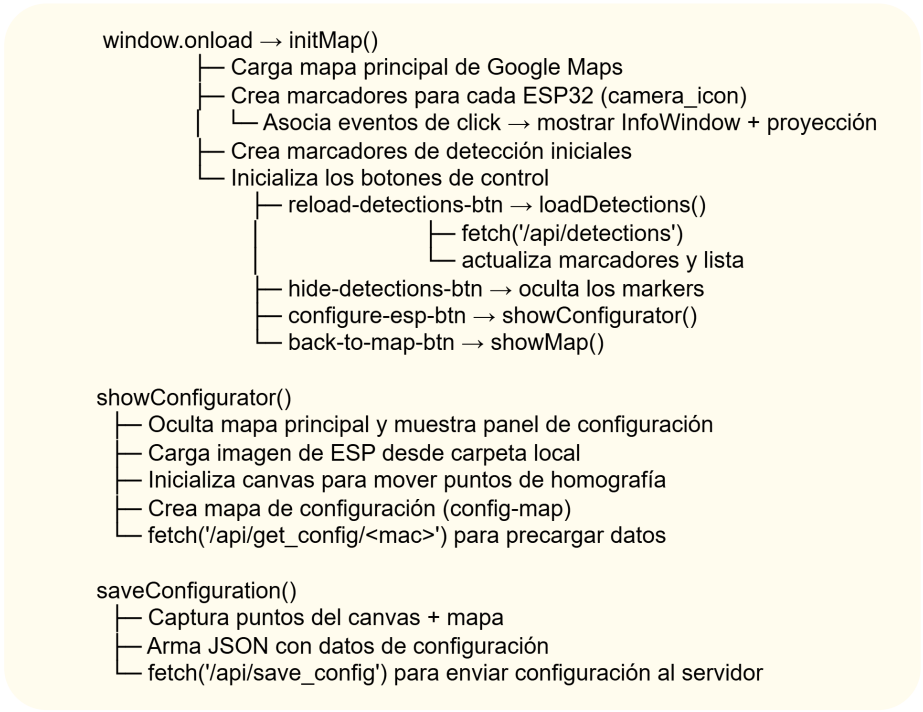
A continuación se presenta una representación jerárquica de las principales dependencias y responsabilidades del código:

Este esquema permite visualizar cómo las distintas funciones y eventos del frontend se encadenan entre sí, destacando especialmente las interacciones asíncronas que conectan con la API RESTful del backend.

6.5.3. Interacción dinámica con el usuario y el servidor

El comportamiento interactivo del frontend se logra mediante funciones en JavaScript que operan sobre el DOM y el objeto `google.maps.Map`. Estas funciones permiten actualizar la interfaz en tiempo real, responder a eventos del usuario y comunicarse con el backend de Flask a través de peticiones `fetch()` (AJAX).

A través de estas llamadas asíncronas, se consulta información georreferenciada (`/api/detections`), se recuperan configuraciones guardadas (`/api/get_config/<mac>`),



se actualizan parámetros (`/api/save_config`) y se envían comandos como el reinicio de nodos (`/reset/<mac>`).

Cada acción del usuario —por ejemplo, hacer clic sobre un nodo o presionar el botón `Reload Detections`— desencadena una cadena de funciones que consulta, procesa y renderiza la información correspondiente en el mapa. Gracias a esta lógica desacoplada, la interfaz reacciona sin necesidad de recargar la página, manteniendo una experiencia fluida y responsiva.

La integración con Google Maps API es central para la visualización geográfica. A partir de las coordenadas almacenadas en la base de datos, se generan marcadores para representar nodos ESP32, detecciones vehiculares y configuraciones proyectadas sobre la vereda. También se construyen elementos gráficos como polígonos, líneas ortogonales y etiquetas dinámicas. El mapa funciona así como núcleo visual de la aplicación, articulando los distintos subsistemas mediante una interfaz intuitiva para el usuario final.

6.6. Módulo de Comunicación Inalámbrica (`wireless_comm.py`)

Este módulo encapsula todas las operaciones relacionadas con la comunicación inalámbrica, tanto vía BLE como a través de la red NB-IoT. Su diseño orientado a objetos permite una separación clara de responsabilidades entre los distintos tipos de enlaces, facilitando el mantenimiento, la extensibilidad y la reutilización de componentes.

El módulo centraliza dos tipos de tareas:

- La recepción de imágenes y detecciones en tiempo real desde los Dispositivos

Parkwise mediante BLE.

- La recepción de detecciones transmitidas mediante MQTT por la red NB-IoT, en formato JSON.

Ambos flujos de datos son procesados y persistidos en la base de datos local, desde donde luego serán transformados y visualizados en la interfaz web.

6.6.1. Diagrama de clases

El diseño orientado a objetos de este módulo se apoya en dos clases principales: `BLEManager` y `NBIoTManager`. La Figura 6.10 presenta el diagrama de clases correspondiente.

- **BLEManager**: gestiona el flujo completo de comunicación BLE. Administra la reconstrucción de imágenes fragmentadas, el almacenamiento en disco, la suscripción a notificaciones GATT y la inserción de detecciones en la base de datos.
- **NBIoTManager**: encapsula la conexión al broker MQTT, la suscripción a tópicos específicos, la interpretación de mensajes entrantes y su registro como detecciones.

Ambas clases comparten una responsabilidad común: recibir detecciones y garantizar que los datos ingresen al sistema con la menor latencia y la mayor integridad posible. Se complementan con funciones auxiliares asincrónicas que escanean dispositivos, ejecutan comandos de reinicio o lanzan los receptores en modo continuo.

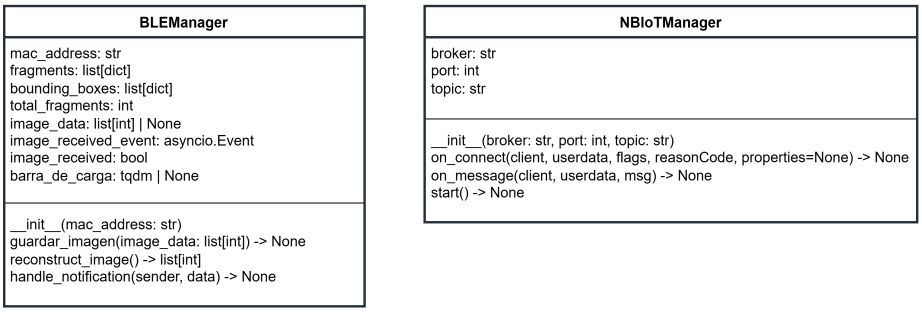


Figura 6.10: Diagrama de clases del módulo `wireless_comm.py`. Se observa la separación de responsabilidades entre los manejadores de BLE y de NB-IoT, junto con sus principales atributos y métodos.

6.6.2. Diagrama de comunicación

Para complementar la estructura orientada a objetos del módulo **wireless_comm.py**, se diseñaron una serie de diagramas de flujo (comunicación) simplificados que ilustran las interacciones clave entre componentes y dispositivos externos. Aunque no siguen formalmente la notación UML estándar, estos diagramas ofrecen una representación clara y secuencial de los procesos internos más relevantes.

1. Proceso de escaneo y configuración inicial (scan_and_request_images())

Este flujo representa cómo el operario ejecuta el comando de escaneo, se detectan los Dispositivos Parkwise vía BLE y se solicitan imágenes de referencia a cada nodo. Estas imágenes se reconstruyen y almacenan en la base de datos, sentando las bases para la posterior configuración de homografías.

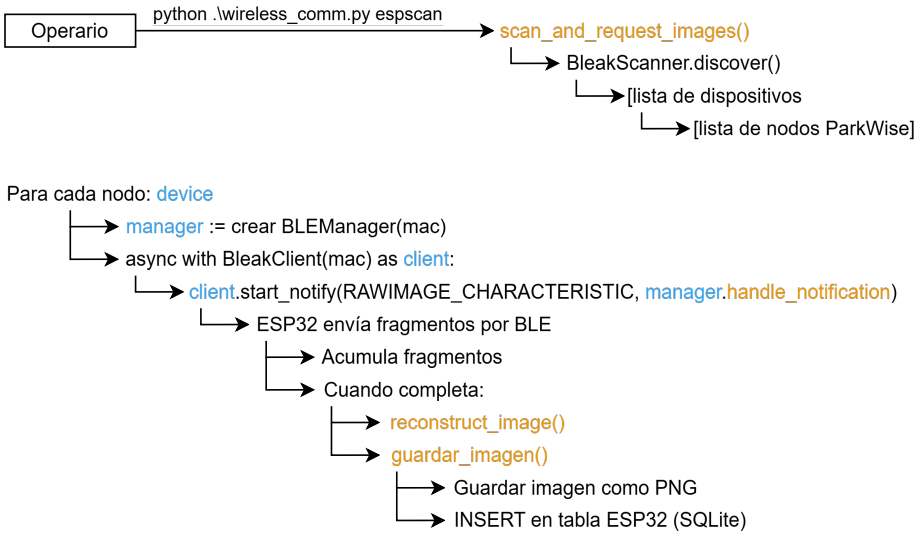


Figura 6.11: Flujo de escaneo de nodos Parkwise y solicitud de imagen inicial por BLE.

2. Recepción de detecciones en régimen (receive_bounding_boxes() y nb_iot_bounding_boxes())

En esta etapa, dos hilos paralelos lanzados desde **app.py** se encargan de recibir bounding boxes desde los nodos. Uno escucha por BLE aquellos dispositivos con homografía registrada, mientras que el otro se conecta a un broker MQTT y recibe mensajes NB-IoT. En ambos casos, los datos se parsean y se insertan en la tabla **Detection**.

3. Envío remoto de comandos de reinicio forzado (ble_hardreset_esp())

Este diagrama describe cómo un usuario puede forzar el reinicio de un nodo Parkwise a través de la interfaz web. El servidor convierte el comando en un paquete JSON, lo transmite por BLE y lo escribe en la característica GATT correspondiente del dispositivo seleccionado.

Estos diagramas no solo documentan las rutas de comunicación y los puntos de interacción con la base de datos, sino que también evidencian la separación de responsabilidades entre BLE y NB-IoT, facilitando la escalabilidad del sistema a futuro.

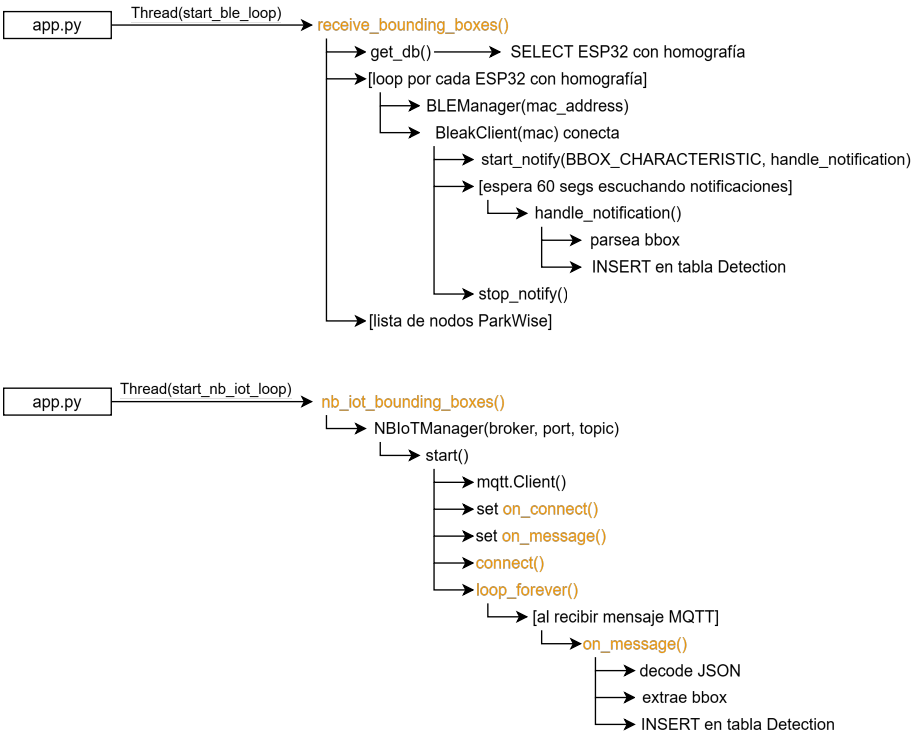


Figura 6.12: Flujo de comunicación en modo régimen: detecciones BLE y NB-IoT procesadas en hilos concurrentes.

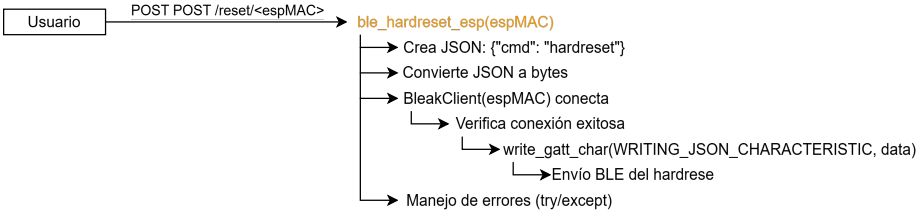


Figura 6.13: Flujo simplificado de envío de comandos de hard reset a nodos Parkwise vía BLE.

6.6.3. Principios aplicados: SOLID, DRY, encapsulamiento, etc

El desarrollo del módulo `wireless_comm.py` se guió por principios fundamentales de diseño de software, que contribuyeron a mejorar la mantenibilidad, la legibilidad y la escalabilidad del código. A continuación se describen brevemente los más relevantes:

- **Encapsulamiento:** la lógica específica de cada tipo de comunicación (BLE y NB-IoT) fue encapsulada en clases independientes (`BLEManager` y `NBIoTManager`), evitando que detalles de implementación se propaguen a otras partes del sistema.
- **Separación de responsabilidades (SRP):** cada clase y función tiene una responsabilidad única bien definida. Por ejemplo, la función `ble_hardreset_esp()` sólo se encarga de enviar un comando GATT, mientras que el hilo principal de recepción BLE ejecuta exclusivamente el ciclo de escaneo y suscripción.

- **DRY (Don't Repeat Yourself):** se evitaron duplicaciones de lógica en funciones de decodificación, almacenamiento y parsing de detecciones, centralizando dichas tareas en funciones reutilizables que actúan sobre ambos protocolos.
- **Abstracción y cohesión:** se definieron métodos que operan en un nivel de abstracción coherente con la clase que los contiene (por ejemplo, `subscribe_to_notifications()` en `BLEManager`), evitando acoplamientos innecesarios.
- **Flexibilidad para extensión futura:** el diseño actual permite incorporar nuevos métodos de comunicación (como WiFi o LoRa) siguiendo el mismo patrón orientado a objetos, sin necesidad de modificar los módulos existentes (principio de abierto/cerrado).

La aplicación de estas buenas prácticas permitió construir un módulo robusto y extensible, clave para un sistema distribuido y dependiente de múltiples flujos de comunicación asincrónica como Parkwise.

6.7. Comunicación entre Módulos

El software de alto nivel de Parkwise está estructurado como un conjunto de módulos interdependientes, cada uno con responsabilidades bien definidas. Esta separación lógica facilita el mantenimiento, la escalabilidad y la trazabilidad del sistema en su conjunto. La Figura 6.14 resume los principales componentes involucrados en el flujo de datos desde la recepción en los nodos hasta la visualización en la interfaz web.

6.7.1. Diagrama de Módulos

A continuación, se describen brevemente las funciones principales de cada módulo:

- `app.py`: núcleo del sistema. Inicia el servidor Flask, lanza los hilos de comunicación BLE y NB-IoT, define los endpoints REST y coordina el acceso a la base de datos.
- `wireless_comm.py`: módulo orientado a objetos que gestiona las comunicaciones con los nodos Parkwise, incluyendo la recepción detecciones (MQTT), así como la ejecución remota de comandos de reinicio.
- `index.html`: archivo HTML que representa la interfaz de usuario. Integra Google Maps, renderiza detecciones y veredas, y permite configurar dispositivos ESP32 mediante eventos e interacciones gráficas.
- `database.py`: contiene funciones para inicializar y consultar la base de datos SQLite, encapsulando operaciones CRUD sobre tablas de dispositivos y detecciones.
- `config.py`: archivo auxiliar donde se declaran constantes, como claves de API y parámetros de conexión.
- `operario.py`: script utilizado por el operario para conectarse via ble y recibir la imagen, la misma es enviada al servidor, que la guarda en la base de datos.
- `parkwise.db`: base de datos embebida que almacena configuraciones, detecciones, imágenes y homografías para cada ESP32 registrado.

6.7.2. Sincronización y Coordinación entre Subsistemas

Una característica distintiva del sistema Parkwise es su capacidad para ejecutar tareas concurrentes en distintos hilos de ejecución, sin sacrificar coherencia ni estabilidad. Al lanzar la aplicación, el script `app.py` inicia dos hilos independientes que ejecutan funciones asincrónicas del módulo `wireless_comm.py`:

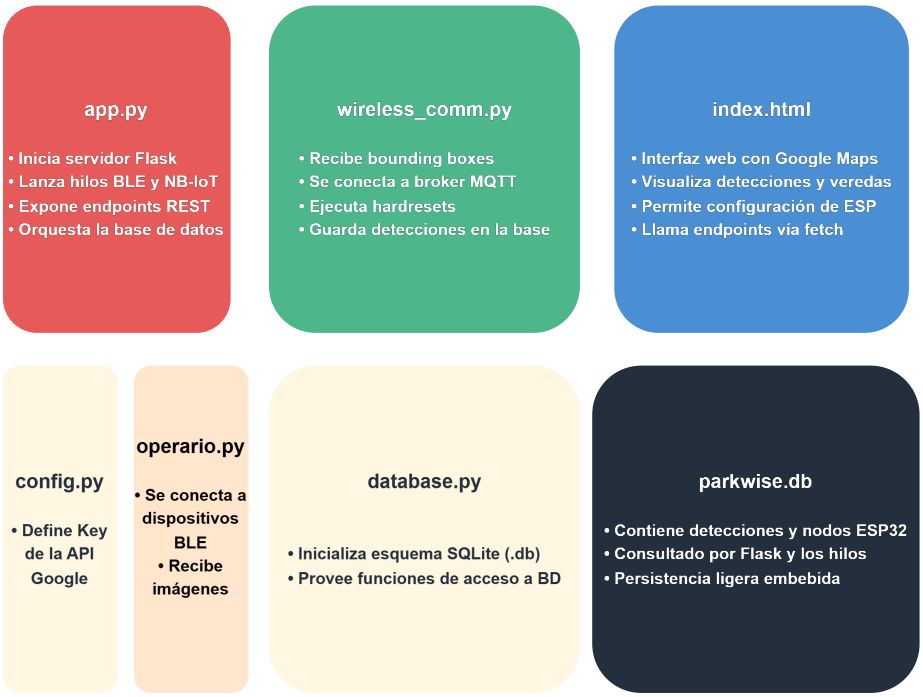


Figura 6.14: Diagrama de módulos del sistema de software de Parkwise. Cada componente cumple una función específica dentro del flujo general: desde el servidor web y los hilos de comunicación hasta la base de datos embebida y la interfaz gráfica.

- `start_ble_loop()`: escucha continuamente por notificaciones BLE de los dispositivos, reconstruye imágenes en tiempo real, y registra detecciones en la base.
- `start_nb_iot_loop()`: se conecta a un broker MQTT local o remoto y escucha por detecciones NB-IoT transmitidas por los nodos.

Ambos hilos son demonios del sistema, lo que significa que se ejecutan en segundo plano durante todo el ciclo de vida del servidor. Sus operaciones no bloquean la ejecución principal ni interfieren con los endpoints RESTful atendidos por Flask.

La sincronización entre módulos se logra mediante el uso compartido de la base de datos SQLite (`parkwise.db`). Cada detección recibida, independientemente del canal (BLE o NB-IoT), se almacena en la tabla `Detection`, con su correspondiente `mac_address`, coordenadas en píxeles y timestamp. Posteriormente, al invocar el endpoint `/api/detections`, Flask consulta dicha tabla, transforma las coordenadas mediante la matriz de homografía almacenada, y genera datos geo-referenciados que son renderizados en el frontend.

Este modelo desacoplado permite que:

- La recepción de datos y la visualización funcionen de forma independiente.
- Los errores en la interfaz web no afecten el almacenamiento de detecciones.
- Sea posible integrar nuevas fuentes de datos (ej. WiFi o LoRa) sin modificar la lógica de visualización.

En resumen, el sistema implementa una arquitectura reactiva basada en eventos y almacenamiento intermedio, donde cada subsistema cumple un rol bien definido y se

comunica mediante estructuras compartidas persistentes, garantizando la robustez y modularidad del software.

6.8. Conclusiones del capítulo

El desarrollo del software de alto nivel de Parkwise representó un desafío integral que combinó múltiples disciplinas: comunicaciones inalámbricas, diseño de interfaces web, sincronización de procesos concurrentes y persistencia de datos. A lo largo de este capítulo se documentaron las decisiones arquitectónicas más importantes, los diagramas funcionales y los principios aplicados para lograr un sistema robusto, modular y operativo en entornos reales.

6.8.1. Resumen y fortalezas del diseño

El sistema desarrollado logró cumplir con sus objetivos funcionales en dos modos de operación diferenciados: *setup* y *régimen*. Durante el *setup*, el operario puede configurar los nodos Parkwise de forma visual e interactiva, estableciendo puntos de homografía y vereda. En *régimen*, las detecciones vehiculares se reciben en tiempo real por BLE o NB-IoT, se proyectan sobre el mapa y se visualizan automáticamente en una interfaz web fluida y amigable.

Entre las principales fortalezas del diseño se destacan:

- **Modularidad**: cada componente (backend, frontend, comunicaciones, base de datos) fue construido como módulo autónomo, lo que facilita el mantenimiento y la evolución del sistema.
- **Bajo acoplamiento entre capas**: el servidor, los hilos de comunicación y el frontend pueden operar en paralelo sin interferencia directa, gracias al uso de una base de datos intermedia y una API RESTful.
- **Simetría entre prototipado y producción**: la implementación fue pensada para escalar, permitiendo migrar fácilmente a entornos con múltiples nodos, servidores remotos o despliegues en la nube.

6.8.2. Lecciones aprendidas y desafíos enfrentados

Durante el proceso de implementación surgieron importantes aprendizajes técnicos y metodológicos:

- **Valor del diseño incremental**: validar cada módulo por separado en condiciones reales permitió detectar errores de integración tempranamente y adaptar la arquitectura sin grandes refactorizaciones.
- **Importancia de la asincronía y la tolerancia a fallos**: especialmente en entornos urbanos con conectividad variable, el uso de hilos y buffers de almacenamiento resultó clave para no perder datos ante cortes o reinicios.
- **Necesidad de depuración visual y feedback inmediato**: tanto en la etapa de configuración como en la visualización de detecciones, el diseño de una interfaz intuitiva simplificó enormemente el diagnóstico de errores y el uso por parte de operarios no técnicos.

No obstante, también se identificaron desafíos relevantes:

- La coordinación entre múltiples protocolos inalámbricos, con sus respectivas bibliotecas, requirió un cuidado especial en la sincronización de procesos.
- La integración con Google Maps y la gestión de datos binarios (imágenes) implicó complejidades adicionales en el manejo del frontend.

6.8.3. Consideraciones para seguridad y escalabilidad futura

El sistema fue diseñado para funcionar en red local durante la etapa de prototipado, pero su arquitectura es compatible con una futura migración a un entorno de despliegue más robusto. Para ello se recomienda:

- **Restringir el acceso al servidor**: ya sea mediante autenticación por IP, contraseñas o tokens, para evitar accesos no autorizados.
- **Proteger los endpoints críticos**: implementar medidas para validar y filtrar peticiones maliciosas, especialmente aquellas que modifican el estado del sistema (`POST`, `PUT`).
- **Cifrado de datos en tránsito**: utilizar HTTPS en lugar de HTTP, junto con servidores de producción como Gunicorn o nginx.
- **Migración de la base de datos a PostgreSQL**: para permitir múltiples conexiones concurrentes, respaldos remotos, gestión de usuarios y seguridad reforzada.

6.8.4. Proyecciones de mejora y líneas de trabajo futuro

El código fue estructurado siguiendo principios de programación como DRY (Don't Repeat Yourself), SRP (Single Responsibility Principle) y encapsulamiento, lo que deja el camino abierto para múltiples mejoras futuras:

- Incorporación de nuevas interfaces de comunicación (WiFi, LoRa).
- Reemplazo del *backend* por FastAPI u otro *framework* más escalable.
- Implementación de autenticación de usuarios y gestión de sesiones.
- Sincronización de múltiples Gateways con una base de datos centralizada.

En síntesis, el sistema Parkwise demostró ser técnicamente viable, funcional y flexible, brindando una base sólida para su evolución hacia versiones más complejas, seguras y distribuibles a escala urbana.

Esta página ha sido intencionalmente dejada en blanco.

Capítulo 7

Pruebas y Validación

En este capítulo se detallan las pruebas realizadas para verificar y validar el correcto funcionamiento del sistema Parkwise en sus distintos niveles: adquisición, inferencia, comunicación, procesamiento, visualización e integración. Las pruebas se diseñaron en función de los requisitos técnicos establecidos en el capítulo de Diseño General (capítulo 1) y buscan demostrar la viabilidad técnica del sistema así como su desempeño bajo condiciones reales de uso.

Las correspondientes referencias al código fuente, scripts de prueba o notebooks utilizados, se encuentran disponibles en el repositorio *park-wise-pfc* [25], como forma de respaldo técnico y trazabilidad de los resultados presentados.

7.1. Pruebas del Modelo de Inferencia FOMO

El modelo fue entrenado utilizando un conjunto total de 557 imágenes, que incluyen 2981 vehículos etiquetados manualmente. Las imágenes provienen de una combinación de tres fuentes: el *Edge Impulse Toy Dataset*, el *Tomas Diago Dataset* y el *Drone Dataset*, lo que permitió abarcar una diversidad de ángulos, condiciones lumínicas y perspectivas urbanas.

Para el entrenamiento y evaluación del modelo, se empleó una división estratificada del conjunto de datos en un 80 % para entrenamiento (446 imágenes) y un 20 % para testeo (111 imágenes), asegurando una distribución representativa de ejemplos en ambos subconjuntos.

El dataset junto con el pipeline de procesamiento de datos puede encontrarse repositorio de Edge Impulse desarrollado por el equipo [32].

Resultados del Modelo en Precisión FLOAT32

El modelo original, sin optimización para ejecución en dispositivos embebidos, fue entrenado utilizando precisión **FLOAT32**. Los resultados sobre el conjunto de testeo fueron los siguientes:

- **Centroid Scoring**: 97.12 %
- **Precision (non-background)**: 0.98
- **Recall (non-background)**: 0.94
- **F1 Score (non-background)**: 0.96

Estos valores reflejan una excelente capacidad del modelo para distinguir correctamente los vehículos en las imágenes, con una alta precisión y un muy buen equilibrio entre la sensibilidad (recall) y la especificidad.

Resultados del Modelo Cuantizado en Precisión INT8

Con el objetivo de reducir el tamaño del modelo y permitir su despliegue en hardware de bajo consumo energético (como el ESP32S3), se aplicó un proceso de *cuantización post-entrenamiento* a precisión **INT8**. A pesar de la pérdida de precisión numérica inherente a esta técnica, el modelo mantuvo un rendimiento notable:

- **Centroid Scoring**: 95.19 %
- **Precision (non-background)**: 0.98
- **Recall (non-background)**: 0.93
- **F1 Score (non-background)**: 0.95

La leve disminución en la métrica de *accuracy* y *recall* se encuentra dentro de lo esperable para este tipo de optimización, y valida la viabilidad del modelo para aplicaciones en tiempo real sobre dispositivos TinyML.

Tabla 7.1: Comparación de métricas entre modelo FLOAT32 e INT8

Métrica	**Modelo FLOAT32**	**Modelo INT8**
Accuracy	97,12 %	95,19 %
Precision (non-background)	0,98	0,98
Recall (non-background)	0,94	0,93
F1 Score (non-background)	0,96	0,95

Los resultados demuestran que la cuantización no afectó significativamente la precisión del modelo. Esta estabilidad en el desempeño sugiere que el modelo es robusto ante transformaciones de representación numérica y adecuado para ser ejecutado en microcontroladores con limitaciones computacionales y de memoria.

Conclusión de los Resultados

Los resultados obtenidos validan la eficiencia del modelo propuesto tanto en precisión completa (FLOAT32) como en su versión comprimida (INT8). La pérdida de rendimiento asociada a la cuantización resulta marginal frente a la ventaja de ejecutarlo en dispositivos embebidos.

No obstante, un análisis más profundo del espacio de características aprendido por el modelo revela ciertas limitaciones sobre el conjunto de entrenamiento. En la Figura 7.1, se presenta una visualización bidimensional del espacio de embeddings extraídos de las imágenes, en la que se evidencia una fuerte concentración de las características en unos pocos conglomerados bien definidos. Esta distribución sugiere una baja diversidad en las escenas, lo que podría limitar la capacidad del modelo para generalizar ante escenarios urbanos menos representados o condiciones visuales no vistas durante el entrenamiento.

En línea con esta observación, se identificó que muchas de las imágenes utilizadas comparten configuraciones similares: alturas, puntos de vista, distribución de objetos y condiciones de iluminación. Esta homogeneidad en los datos limita el rango de variabilidad que el modelo puede aprender, generando un sesgo hacia ciertos contextos visuales. Por ello, resulta fundamental ampliar el conjunto de entrenamiento con

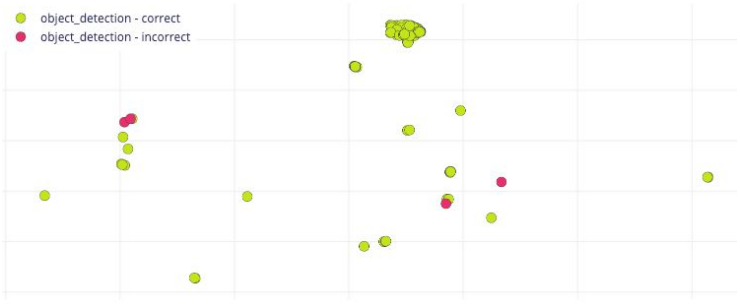


Figura 7.1: Visualización bidimensional del espacio de embeddings extraídos de las imágenes de test.

nuevas capturas que contemplen diferentes alturas, ángulos oblicuos, mayor densidad vehicular, diferentes horarios del día y condiciones climáticas adversas.

A continuación se incluyen en las Figuras 7.2 y 7.3 ejemplos concretos de predicciones del modelo, que ilustran tanto casos exitosos como fallidos.

Figura 7.2: Detecciones correctas efectuadas por el modelo entrenado en presición INT8

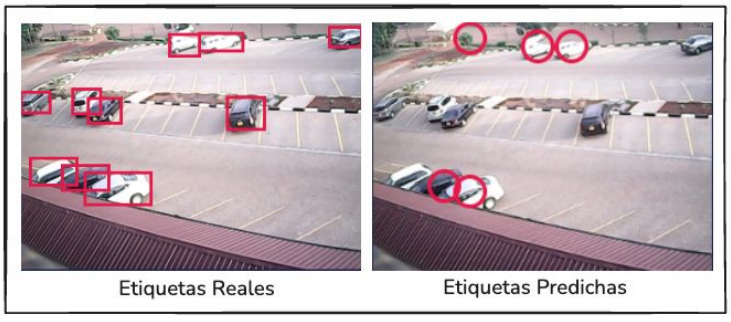


Figura 7.3: Detecciones incorrectas efectuadas por el modelo entrenado en presición INT8

Una vía prometedora para continuar mejorando el desempeño del sistema consiste en reentrenar el modelo con un conjunto de datos más amplio y diverso. Alternativamente, se podrían aplicar técnicas de *fine-tuning* o *data augmentation* dirigidas

específicamente a los contextos donde el modelo ha demostrado mayor debilidad. Estas estrategias permitirían extender la capacidad de generalización del modelo sin comprometer su eficiencia en ejecución.

7.2. Pruebas del Dispositivo Parkwise

Esta sección describe las pruebas realizadas sobre el Dispositivo Parkwise, con el objetivo de verificar sus capacidades básicas de adquisición de imágenes, ejecución local del modelo de inferencia y estabilidad del sistema bajo las condiciones previstas de operación. Todas las pruebas se realizaron sobre el microcontrolador Seeed XIAO ESP32S3 Sense, ejecutando firmware propio desarrollado en C++ con FreeRTOS y la herramienta Edge Impulse. Dicho código fue desarrollado en el framework de Espressif-IDF, por lo que los archivos con formato .c, .h, y .cc que se mencionarán a continuación hacen referencia al proyecto de firmware, disponible en el repositorio `park-wise-pfc` [25] ubicado en .\`src`\`app`\`parkwise`

7.2.1. Verificación de la captura periódica de imágenes mediante la cámara integrada.

La adquisición de imágenes en el Dispositivo Parkwise se realiza a través de la cámara integrada en el módulo Seeed XIAO ESP32-S3 Sense. Esta funcionalidad está implementada en la función `GetImage()`, definida en el archivo `image_provider.cc` del repositorio del proyecto [25]. La función es invocada periódicamente dentro del ciclo principal de la tarea `inference_task()`, también definida en el mismo archivo, en la cual se realiza la captura de la imagen, su conversión y posterior procesamiento por el modelo de inferencia.

Para verificar la adquisición periódica de imágenes, se utilizó un Notebook Jupyter en Python, en el repositorio `park-wise-pfc` [25] llamado `suscribe-to-charasteristics`. La celda que ejecuta la prueba se titula "LiveCam with Detections (imágenes en uint8_t y bb en json)".

Este notebook establece una conexión BLE con el dispositivo, se suscribe a la característica correspondiente al envío de imágenes crudas (`rawimage`), y las visualiza cada vez que llegan en tiempo real en un ciclo sin fin. Durante la sesión de prueba, se observó una secuencia continua de imágenes, con una frecuencia de actualización aproximada de cinco segundos. Este intervalo incluye el tiempo de adquisición, preprocesamiento y ejecución de inferencia, seguido de un retardo de un segundo.

La figura 7.4 muestra una captura de pantalla tomada durante la ejecución del notebook, donde se aprecian las imágenes recibidas por BLE en forma periódica.

La prueba se mantuvo en ejecución durante más de diez minutos sin interrupciones ni errores de captura. Esto permite concluir que la funcionalidad de adquisición periódica está implementada correctamente y que las imágenes pueden ser visualizadas y verificadas mediante herramientas disponibles en el repositorio.

7.2.2. Validación de la ejecución local del modelo FOMO para detección de vehículos.

El modelo de inferencia utilizado en este proyecto está basado en la arquitectura FOMO (*Faster Objects, More Objects*) y fue desplegado en el microcontrolador Seeed XIAO ESP32S3 Sense utilizando el SDK de Edge Impulse. La ejecución local del modelo se lleva a cabo dentro de la tarea `inference_task()` (archivo `main_functions.cc`),

Figura 7.4: Captura de pantalla del Jupyter notebook mostrando imágenes recibidas en tiempo real desde el dispositivo vía BLE.

donde se genera la señal de entrada a partir de la imagen cruda capturada y se invoca la función `run_classifier()`.

El resultado de la inferencia es procesado mediante la función `print_inference_result()`, ubicada en el mismo archivo, la cual extrae y formatea las *bounding boxes* detectadas y las envía por BLE a través de las funciones `update_gatt_detections()` y `send_detections_to_suscribers()`.

La validación de esta funcionalidad se realizó visualmente mediante el mismo Notebook de Jupyter `suscribe-to-charasteristics`. En la celda titulada "LiveCam with Detections (imágenes en uint8_t y bb en json)", se visualizan las imágenes recibidas superpuestas con las *bounding boxes* detectadas por el modelo.

La ultima figura 7.4 muestra un ejemplo de esta superposición, donde se pueden observar claramente los objetos detectados por el sistema, validando el correcto funcionamiento del modelo FOMO embebido.

7.2.3. Medición de latencia entre adquisición e inferencia.

El SDK de Edge Impulse permite obtener métricas detalladas de tiempo durante el proceso de inferencia. Estas métricas son recolectadas automáticamente y disponibles en la estructura `ei_impulse_result_t`, devuelta por la función `run_classifier()`. Los valores relevantes se imprimen en consola mediante la función `print_inference_result()`, ubicada en el archivo `main_functions.cc`.

Una salida típica de esta función en la terminal puede verse a continuación:

```
Reportes de la inferencia: Timing: DSP 54 ms, inference 843 ms,
↪  anomaly 0 ms.
Cantidad de bounding boxes en la salida del modelo: 2
JSON compacto: {"d":[[0.98,130,85,48,38],[0.91,220,91,50,36]]}
```

Estos reportes permiten descomponer el tiempo total del ciclo de inferencia en tres componentes:

- `DSP`: tiempo dedicado al preprocesamiento y extracción de características.
- `inference`: tiempo neto de ejecución del modelo neuronal.

- `anomaly`: reservado para modelos con salida de anomalía (no usado en este caso).

En el ejemplo presentado, el tiempo total de inferencia local es menor a 1 segundo, lo cual es coherente con la frecuencia de actualización observada en la sección anterior. Esta información fue clave para validar que el ciclo completo de adquisición e inferencia se mantiene dentro de márgenes adecuados para el objetivo del sistema.

7.2.4. Evaluación del uso eficiente de recursos embebidos (CPU, memoria).

Para monitorear el consumo de recursos del dispositivo, se implementó una función de diagnóstico denominada `imprimir_memstatus()`, definida en el archivo `main_functions.cc`. Esta función permite imprimir por UART los siguientes valores:

- Espacio libre en el `stack` de la tarea actual (`uxTaskGetStackHighWaterMark()`).
- Memoria libre total del heap (`esp_get_free_heap_size()`).
- Memoria libre del heap con capacidad de acceso de 8 bits

Durante las pruebas, se registraron periódicamente estas salidas en la terminal. A modo de ejemplo, se muestra una de ellas:

```
Espacio libre en el stack: 1452
Espacio libre en el heap: 277320 bytes
Espacio libre en varias partes del heap: 239104 bytes
```

Estas métricas se mantuvieron dentro de márgenes seguros durante toda la prueba, sin provocar reinicios ni cuelgues. Esto valida que la carga impuesta por el modelo y las tareas concurrentes no supera las capacidades del sistema.

7.2.5. Verificación del estado de estacionamiento: Detección de ocupación instantánea

Una consideración importante del sistema es la distinción entre un vehículo en movimiento y uno efectivamente estacionado. Como se definió en los criterios de éxito del proyecto (Apéndice A), un 'vehículo estacionado' es aquel que permanece inmóvil por al menos un minuto.

La implementación actual, concebida como una prueba de concepto, realiza una detección de ocupación instantánea. Es decir, el modelo FOMO identifica la presencia de vehículos en cada imagen capturada periódicamente, lo cual es un indicador directo del uso de un espacio. Este enfoque es funcional para mapear la disponibilidad en tiempo real, asumiendo que los vehículos detectados cerca de la vereda en capturas sucesivas probablemente están estacionados.

Para una futura versión productiva, la verificación estricta del estado 'estacionado' podría implementarse a nivel del Servidor Central. El servidor podría analizar el historial de detecciones de un mismo nodo, y si un vehículo (identificado por un centroide en una posición geográfica similar) permanece detectado en la misma zona por un período superior a un minuto, se confirmaría su estado como 'estacionado'. Esta mejora permitiría filtrar el tráfico pasajero y aumentaría la precisión del mapa de ocupación, representando una línea de trabajo futuro clara para el proyecto.

7.3. Pruebas de Comunicación BLE (Parkwise → Gateway)

El presente experimento tuvo como objetivo verificar el establecimiento correcto de la conexión BLE entre el Dispositivo Parkwise (ESP32-S3) y un cliente (Gateway), así como la recepción de inferencias transmitidas a través de la característica GATT correspondiente a detecciones. La funcionalidad completa de conexión, suscripción y transmisión se encuentra implementada en el archivo `gatt_svc.c` del repositorio [25].

7.3.1. Estado inicial sin suscripciones

Previo a la conexión, el Dispositivo Parkwise permanece en estado inactivo, evaluando periódicamente si existen clientes suscritos a las características GATT configuradas. En ausencia de suscriptores, se imprimen mensajes como los siguientes en la consola del dispositivo:

```
I (13041) gatt_svc: No hay ningun suscriptor.
I (16161) gatt_svc: No hay ningun suscriptor.
I (19281) gatt_svc: No hay ningun suscriptor.
I (22401) gatt_svc: No hay ningun suscriptor.
I (25521) gatt_svc: No hay ningun suscriptor.
...
```

7.3.2. Conexión y suscripción desde el Gateway

Al ejecutar el script `gateway.py` ubicado en la ruta `src/app/gateway`, el cliente BLE escanea dispositivos cercanos, identifica el Dispositivo Parkwise, y establece una conexión segura. La siguiente salida de terminal corresponde al lado del cliente:

```
2025-06-26 19:20:58,628 - Gateway - INFO - Escaneando dispositivos...
2025-06-26 19:21:03,741 - Gateway - INFO - Dispositivo 'ParkWise
↪  Device' encontrado: 74:4D:BD:95:A3:0A
2025-06-26 19:21:03,742 - Gateway - INFO - Conectando al dispositivo
↪  74:4D:BD:95:A3:0A...
2025-06-26 19:21:12,699 - Gateway - INFO - Dispositivo ParkWise Device
↪  (74:4D:BD:95:A3:0A) conectado
```

En paralelo, el Dispositivo Parkwise reporta en su consola los eventos correspondientes al establecimiento de conexión y configuración de los parámetros GATT:

```
I (27321) GAP: A new connection was established or a connection attempt
↪  failed.
I (27321) GAP: In this case is: connection established; status=0
I (27331) GAP:
        Imprimiendo a continuacion caracteristicas del "connection
        ↪  descriptor":
I (27341) GAP: connection handle: 1
I (27341) GAP: This (ESP32) device id address: type=0,
↪  value=74:4D:BD:95:A3:0A
I (27351) GAP: Peer (Luminaria) id address: type=0,
↪  value=1C:BF:C0:7D:5C:48
```

```
...
I (27831) GAP: Subscribe event; conn_handle=1 attr_handle=8 reason=1
↪  ...
I (27831) gatt_svc: Subscribe event; conn_handle=1 attr_handle=8
...
I (28491) GAP: Subscribe event; conn_handle=1 attr_handle=21 ...
I (28491) gatt_svc: Subscribe event; conn_handle=1 attr_handle=21
```

7.3.3. Recepción de detecciones por parte del Gateway

Una vez suscrito a la característica de detecciones, el Gateway comienza a recibir notificaciones periódicas. La estructura de los datos sigue el formato JSON compacto implementado en la función `print_inference_result()` ubicada en `main_functions.cc`, donde se construye y transmite el mensaje mediante llamadas a `update_gatt_detections()` y `send_detections_to_suscribers()`.

La consola del Gateway refleja la recepción de dichos mensajes de la siguiente manera:

```
2025-06-26 19:21:12,892 - Gateway - DEBUG - Notificacion BLE recibida:
↪  {"d":[[4.25,96,152,16,16]]}
2025-06-26 19:21:14,752 - Gateway - INFO - Datos recibidos del
↪  dispositivo: {"d":[[4.25,96,152,16,16]]}
...
2025-06-26 19:21:19,177 - Gateway - DEBUG - Notificacion BLE recibida:
↪  {"d":[[0.61,272,160,8,8],[6.07,128,224,8,8]]}
2025-06-26 19:21:19,233 - Gateway - INFO - Datos recibidos del
↪  dispositivo: {"d":[[0.61,272,160,8,8],[6.07,128,224,8,8]]}
```

7.3.4. Confirmación de transmisión en el Dispositivo Parkwise

La consola del Dispositivo Parkwise confirma cada transmisión exitosa de detecciones:

```
I (28661) gatt_svc: Buffer de notificacion asignado correctamente!
I (28661) NimBLE: GATT procedure initiated: notify;
I (28661) NimBLE: att_handle=21
I (28671) gatt_svc: Transmisión de detecions completada.
I (28671) main_functions: Reset Timer Reiniciado!
...
I (34941) gatt_svc: Buffer de notificacion asignado correctamente!
I (34941) NimBLE: GATT procedure initiated: notify;
I (34941) NimBLE: att_handle=21
I (34951) gatt_svc: Transmisión de detecions completada.
```

7.3.5. Desconexión y retorno al estado inactivo

Al finalizar la prueba, se observa la interrupción de las suscripciones desde el lado del Gateway. El dispositivo retorna a estado de espera, imprimiendo nuevamente:

```
I (41221) gatt_svc: No hay ningun suscriptor.
I (44341) gatt_svc: No hay ningun suscriptor.
...
```

7.3.6. Medición del alcance efectivo de la comunicación BLE.

Se realizaron pruebas de alcance con un cliente BLE (notebook con adaptador BLE) situado a distintas distancias del Dispositivo Parkwise en un entorno interior sin obstrucciones severas.

Durante estas pruebas se mantuvo la conexión estable a distancias de hasta aproximadamente 10 metros. Más allá de esa distancia, comenzaron a observarse pérdidas esporádicas de fragmentos o desconexiones. En entornos exteriores con visibilidad directa se espera que el alcance mejore, aunque factores como interferencias electromagnéticas o presencia de objetos metálicos pueden afectar la calidad del enlace.

7.3.7. Evaluación del resultado

La conclusión de esta prueba es que el alcance efectivo en condiciones normales supera los 7 u 8 metros, lo cual es suficiente para el escenario previsto: comunicación local con el Gateway instalado físicamente en la misma luminaria urbana o estructura soporte. La prueba confirma el comportamiento esperado: el dispositivo detecta correctamente los eventos de conexión y suscripción, transmite periódicamente los resultados de inferencia, y finaliza la transmisión cuando el cliente cancela su suscripción o se desconecta. Este flujo puede ser reproducido íntegramente utilizando los *scripts* proporcionados en el repositorio `park-wise-pfc` [25].

7.4. Pruebas de Comunicación NB-IoT (Gateway → Servidor Central)

7.4.1. Publicación y suscripción a tópicos MQTT sobre la red NB-IoT

Esta prueba tuvo como objetivo verificar el flujo de datos desde el Gateway hacia el Servidor Central utilizando el protocolo MQTT sobre la red NB-IoT. La lógica de publicación y suscripción está implementada en los *scripts* `gateway.py` (ubicado en `src/app/gateway`) y `app.py` (ubicado en `src/app/servidor/front_end`), ambos disponibles en el repositorio del proyecto [25].

Inicialización del servidor central Al iniciar el servidor central, se ejecuta el código que inicializa la base de datos y los hilos de escucha para las conexiones entrantes. Entre ellos, se encuentra el hilo encargado de suscribirse al broker público `test.mosquitto.org` al tópico `test/parkwise`, como se refleja en la salida de terminal:

```
NB-IoT: Conectando a broker test.mosquitto.org:1883...
Conectado al broker MQTT test.mosquitto.org:1883,  suscripto al topic:
↪  test/parkwise
Suscripto al topic de comandos: cmd/parkwise
```

Recepción de datos Posteriormente, el servidor central comienza a recibir mensajes con datos de detección provenientes del Gateway, en el tópico `test/parkwise`. A modo de ejemplo, se muestran los siguientes registros que reflejan correctamente la estructura del mensaje y la identificación del emisor (MAC del ESP32):

```
NB-IoT: Mensaje recibido en test/parkwise: {'d': [[4.25, 176, 64, 8,
↪  8]], 'm': '74:4D:BD:95:A3:0A'}
...
NB-IoT: Mensaje recibido en test/parkwise: {'d': [[1.21, 304, 224, 8,
↪  8]], 'm': '74:4D:BD:95:A3:0A'}
...
```

Publicación desde el Gateway Desde el lado del Gateway, se observa cómo los datos inferidos por el Dispositivo Parkwise (recibidos previamente por BLE) son reenviados al broker MQTT utilizando el tópico definido. El script `gateway.py` utiliza la librería `paho.mqtt` para establecer la conexión y enviar los mensajes construidos en formato JSON.

La siguiente traza muestra el envío de detecciones:

```
2025-06-26 20:28:10,780 - Gateway - DEBUG - '{"d": [[4.25, 176, 64, 8,
↪  8]], "m": "74:4D:BD:95:A3:0A"}' enviado al topico 'test/parkwise'
...
2025-06-26 20:28:14,032 - Gateway - DEBUG - '{"d": [[1.21, 288, 64, 8,
↪  8], [1.52, 296, 96, 8, 8]], "m": "74:4D:BD:95:A3:0A"}' enviado al
↪  topico 'test/parkwise'
```

Resultado y consistencia Las pruebas confirman que todos los mensajes enviados desde el Gateway fueron recibidos correctamente por el Servidor Central, sin pérdida ni alteración de formato. El vínculo MQTT se comportó de forma confiable durante todo el periodo de observación, lo que valida el correcto establecimiento del canal de comunicación NB-IoT a través de un broker público.

Esta funcionalidad puede ser reproducida ejecutando los scripts mencionados en paralelo y monitoreando los *logs* generados en la consola.

7.4.2. Medición de latencia y confiabilidad en la transmisión de detecciones

Para evaluar la latencia de la comunicación NB-IoT y la confiabilidad en la entrega de mensajes, se analizaron los registros de consola generados tanto por el Gateway como por el Servidor Central, ejecutando los scripts `gateway.py` y `app.py` respectivamente, disponibles en el repositorio [25].

Método de medición Se utilizaron las marcas de tiempo generadas automáticamente por el `logger` de Python en ambos scripts. Al tratarse de una comunicación unidireccional (Gateway → Servidor Central), se calculó la latencia como la diferencia entre el instante de publicación (en el Gateway) y el de recepción (en el Servidor Central) para un mismo paquete.

Ejemplo de cálculo de latencia A continuación se presenta un ejemplo representativo. El siguiente mensaje fue publicado desde el Gateway a las 19:53:51:

```
2025-07-03 19:53:51,205 - Gateway - DEBUG - '{"d": [[0.61, 288, 88,
↪  8, 8], [1.52, 288, 152, 8, 8]], "m": "74:4D:BD:95:A3:0A"}'
↪  enviado al topico 'test/parkwise'
```

El mismo mensaje fue recibido por el Servidor Central aproximadamente a las 19:53:57 (según el orden secuencial en el log):

```
[19:53:57] NB-IoT: Mensaje recibido en test/parkwise: {'d': [[0.61,
↪  288, 88, 8, 8], [1.52, 288, 152, 8, 8]], 'm':
↪  '74:4D:BD:95:A3:0A'}
```

La latencia estimada para este mensaje fue entonces inferior a 600 ms. Esta estimación se mantuvo consistente a lo largo de múltiples envíos, con oscilaciones menores típicas de una red móvil.

Evaluación de confiabilidad Durante la sesión de prueba, se registraron más de cinco mensajes consecutivos sin pérdida ni duplicación, como puede verificarse comparando los registros del Gateway con los del Servidor Central. La siguiente tabla resume un fragmento de esta secuencia:

Timestamp (Gateway)	**Contenido**	**Recibido en Servidor**
20:28:10,780	[[4.25, 176, 64, 8, 8]]	Sí
20:28:14,032	[[1.21, 288, 64, 8, 8], [1.52, 296, 96, 8, 8]]	Sí
20:28:16,272	[[0.61, 176, 64, 8, 8]]	Sí
20:28:18,543	[[0.61, 128, 88, 8, 8]]	Sí
20:28:25,817	[[0.61, 240, 224, 8, 8]]	Sí
20:28:28,061	[[1.21, 304, 224, 8, 8]]	Sí

No se observaron fallos de transmisión ni retrasos significativos que superaran el segundo, lo que valida el correcto funcionamiento del canal NB-IoT en términos de fiabilidad y desempeño para el volumen de datos manejado.

Observación adicional Es importante destacar que el formato JSON utilizado en los mensajes y la arquitectura asincrónica del sistema contribuyeron a una entrega ágil y estructurada de los datos, facilitando su procesamiento inmediato en el Servidor Central.

7.5. Pruebas del Servidor Central

7.5.1. Verificación del almacenamiento de detecciones en la base de datos

Una vez recibidas las detecciones vehiculares por parte del servidor central mediante MQTT, se valida que los datos sean correctamente almacenados en la base de datos local SQLite (‘parkwise.db‘).

Durante la prueba, el servidor imprime en consola el siguiente mensaje indicando la correcta recepción de detecciones múltiples desde el Gateway:

```
NB-IoT: Mensaje recibido en test/parkwise: {'d': [[0.91, 144, 40, 8,
↪  8], [2.73, 160, 144, 8, 8], [2.73, 272, 168, 8, 8]], 'm':
↪  '74:4D:BD:95:A3:0A'}
```

Estas detecciones quedan registradas en la tabla `Detection` junto con su MAC address, coordenadas en píxeles y coordenadas geográficas. La veracidad de este almacenamiento puede verificarse mediante el script `save_db_as_csv.py` ubicado en el repositorio `park-wise-pfc/src/app/servidor/utils/`, el cual exporta el contenido de las tablas en formato CSV usando pandas y SQLite.

Una captura de pantalla del contenido exportado puede observarse en la figura 7.5, donde se aprecian tres entradas nuevas correspondientes a las detecciones mencionadas.

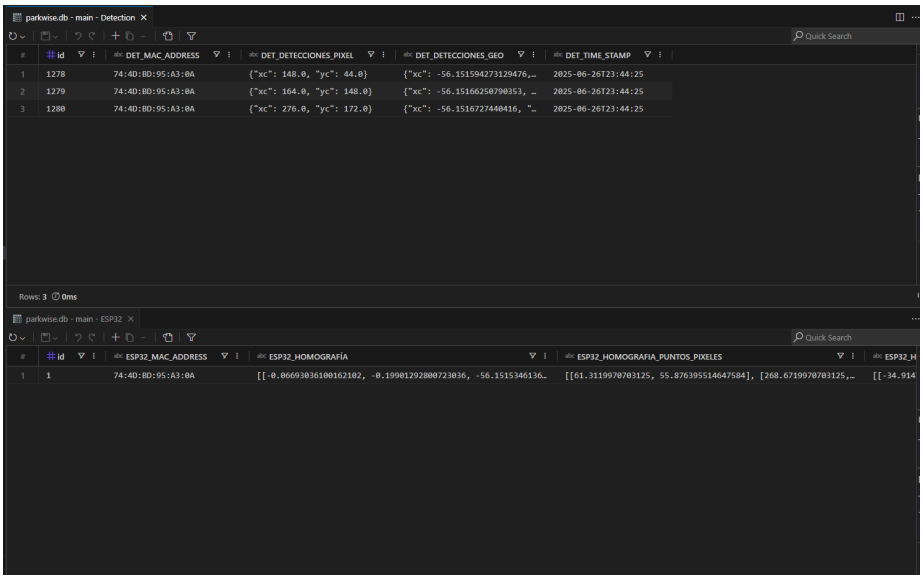


Figura 7.5: Visualización de la tabla `Detection` y `Devices` en la base de datos `parkwise.db` usando DBCode.

7.5.2. Aplicación correcta de la homografía para georreferenciación

Tal como se detalló en la sección 6.1.2, la georreferenciación de las detecciones se logra mediante una homografía, que es una transformación proyectiva (matriz 3x3) calculada a partir de cuatro puntos de correspondencia entre la imagen capturada por la cámara y su ubicación en el mapa. En esta sección se valida que, una vez configurada dicha matriz para un dispositivo, la proyección de los centroides detectados en píxeles a coordenadas geográficas se realiza de manera correcta y visualmente coherente en la interfaz.

Las detecciones almacenadas en píxeles son transformadas en coordenadas geográficas mediante la matriz de homografía previamente configurada por el operario durante la etapa de instalación del dispositivo. Esta matriz se encuentra almacenada en la tabla `Devices` bajo la columna `DVC_HOMOGRAFIA`.

Para esta prueba, se utilizó un dispositivo previamente configurado con una homografía válida, como se observa en la tabla inferior de la figura 7.5, donde el campo `DVC_HOMOGRAFIA` está presente.

La correcta aplicación de la homografía se verifica visualmente en la interfaz web, donde las detecciones transformadas aparecen proyectadas sobre la vereda correspondiente en el mapa. En la figura 7.6, se observa la visualización de tres detecciones alineadas correctamente en relación a la posición de la cámara y las veredas.

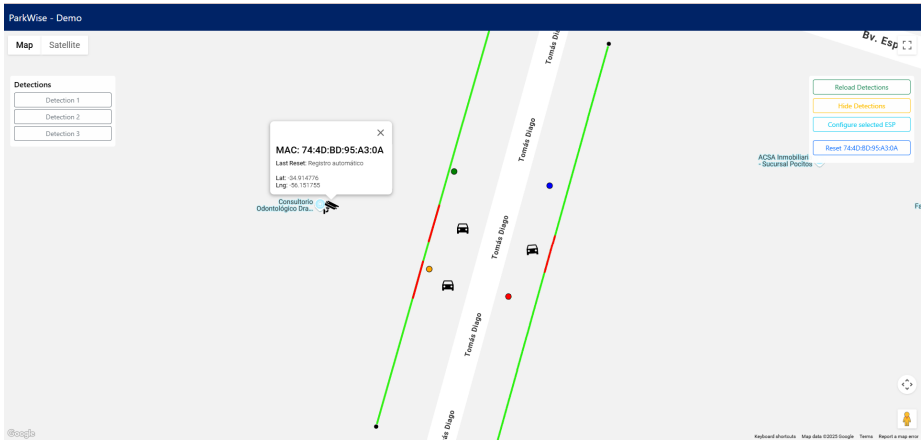


Figura 7.6: Visualización en tiempo real de detecciones proyectadas mediante homografía en el frontend web. Las calles presentan un grosor menor debido al zoom aplicado desde el mapa, sin embargo, la vista satelital coincide con el ancho de las veredas en verde.

7.6. Pruebas del Proceso de Instalación

7.6.1. Funcionamiento del script `operario.py` para captura y envío de imagen

Con el objetivo de simular la fase de instalación de un nuevo Dispositivo Parkwise en campo, se ejecutó el script `operario.py` ubicado en `src/app/servidor/front_end`. Este script automatiza la conexión BLE con el ESP32-S3, solicita la captura de una imagen y la guarda localmente para permitir la configuración posterior de homografía.

Durante su ejecución, se desplegaron mensajes en la terminal que confirmaron el correcto funcionamiento del proceso:

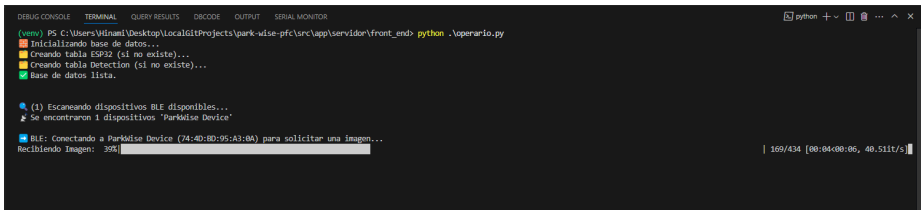


Figura 7.7: Captura de la terminal al ejecutar la recepción de imagen corriendo el script operario.py. Muestra como carga la barra de porcentaje de recepción de la imagen.

El archivo de imagen se almacenó en la ruta relativa `./static/imagenes_recibidas/744DBD95A30A_ac36779f.png`, donde el sufijo alfanumérico es generado aleatoriamente para cada sesión de captura. Además, se insertó correctamente una nueva fila en la tabla `Devices` de la base de datos con únicamente con la MAC registrada y todo el resto de parámetros vacios, cumpliendo con el funcionamiento esperado.

7.6.2. Configuración de homografía y veredas desde la interfaz web

Luego de haber registrado un nuevo Dispositivo Parkwise mediante el script `operario.py`, se procedió a la configuración de la homografía y de los límites geográficos de la vereda directamente desde la interfaz web del sistema. Para ello, se accedió a la pantalla de configuración específica del dispositivo, que presenta dos paneles complementarios: una vista de la imagen capturada por la cámara (panel izquierdo) y un mapa interactivo basado en Google Maps (panel derecho).

En la vista de imagen, el operario seleccionó cuatro puntos clave en la escena —normalmente esquinas del área de detección— que luego son transformados a coordenadas geográficas mediante una homografía. Simultáneamente, en el panel del mapa, se marcaron cuatro puntos correspondientes sobre la vereda real, así como la ubicación aproximada de la cámara. Este procedimiento queda ilustrado en la Figura 7.8, donde pueden verse claramente los puntos seleccionados y los valores registrados automáticamente en pantalla para cada uno de ellos.

Una vez completadas ambas selecciones, se hizo clic en el botón `Save Configuration`, lo cual desencadena el almacenamiento automático de toda la información en la base de datos interna. Como resultado, la fila correspondiente al dispositivo en la tabla `Devices` es actualizada con los nuevos valores en los campos:

- `DVC_HOMOGRAFÍA`: matriz 3×3 calculada a partir de los cuatro pares de puntos imagen–mapa:
  [[-0.063, 0.113, -56.152],
  [-0.039, 0.070, -34.915],
  [0.001, -0.002, 1.000]]
- `DVC_HOMOGRAFIA_PUNTOS_PIXELES`:
  [[256.64, 165.159],
  [169.6, 152.376],
  [264.96, 64.174],
  [156.8, 41.804]]

  `DVC_HOMOGRAFIA_PUNTOS_GEO`: listas de puntos de entrada y destino:
  [[-34.915, -56.152],
  [-34.915, -56.152],
  [-34.915, -56.152],
  [-34.915, -56.152]]
- `DVC_COORDENADAS_GEO_CAMARA`: ubicación estimada de la cámara:
  [-34.915, -56.152]
- `DVC_COORDENADAS_GEO_VEREDA`: vértices del área proyectada de la vereda:
  [[-34.915, -56.152],
  [-34.915, -56.152],
  [-34.915, -56.152],
  [-34.915, -56.152]]
- `DVC_LAST_SEEK_TIME_STAMP`: marca temporal del guardado: 2025-06-27T00:21:45

Esta información permitió al sistema aplicar correctamente la transformación de coordenadas para todas las detecciones futuras realizadas por dicho dispositivo. Asimismo, asegura que las detecciones se proyecten adecuadamente sobre la vereda configurada en el visor de mapa del sistema.

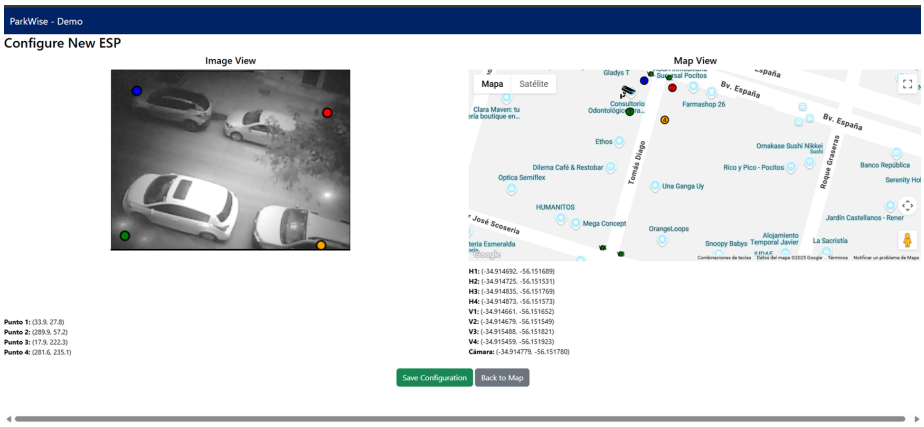


Figura 7.8: Pantalla de configuración web del Dispositivo Parkwise. A la izquierda se muestran los puntos seleccionados sobre la imagen capturada; a la derecha, los puntos correspondientes en el mapa georreferenciado.

7.7. Pruebas de Integración Punta a Punta

7.7.1. Flujo completo

La instalación del Dispositivo Parkwise se realiza colocando el Dispositivo Parkwise sobre una columna en la vía pública. Con la cámara apuntando a los vehículos estacionados frente a ella.

El instalador u operario ejecuta el script encargado de conectarse al Dispositivo Parkwise, solicitarle una fotografía y enviarla por internet al Servidor Central.

En una segunda etapa, el operario accede a la interfaz web del servidor, donde puede seleccionar el Dispositivo Parkwise para configurar su homografía y ubicarlo en el mapa. También deja configurar las veredas, que mostrarán dentro del rango que permite ver la cámara, una proyección de las detecciones, para marcar los lugares ocupados. La figura 7.11 muestra estos ocupados en rojo y los libres en verde.

A partir de dos imágenes se presenta la pantalla del modo instalador en la interfaz web del servidor: la imagen capturada por el operario (fig. 7.9) y el mapa (fig. 7.10). Una vez definidos los puntos de correspondencia, el dispositivo queda configurado para la correcta recepción de sus detecciones.

Una vez configurado el Dispositivo Parkwise, se pueden apreciar sus detecciones en la interfaz web de usuario, donde se muestran además los espacios ocupados, la cámara y los puntos que definen la homografía.

7.8. Pruebas de Tolerancia y Escalabilidad del Sistema

Con el fin de evaluar el comportamiento del sistema Parkwise en situaciones reales y desafiantes, se realizaron pruebas orientadas tanto a la tolerancia a fallos como

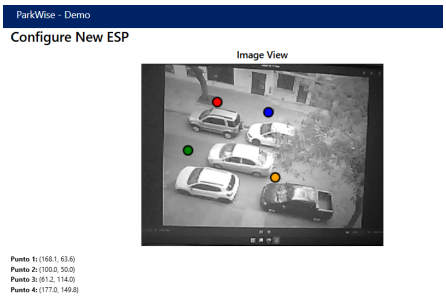


Figura 7.9: Image View en modo instalador: despliega la imagen recibida del Operario, con los puntos de referencia en la imagen, que se corresponderán en el mapa.

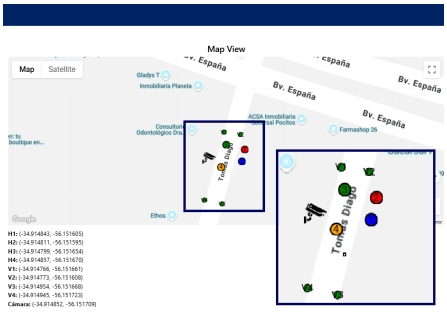


Figura 7.10: Vista del mapa en modo Instalador, con la correspondencia entre puntos de la imagen y del entorno. En el recuadro azul (abajo a la derecha) se amplía la zona de interés.

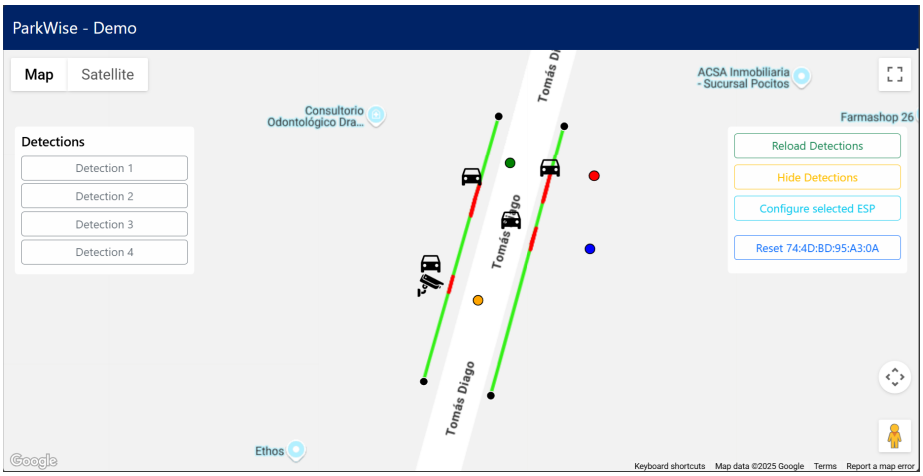


Figura 7.11: Interfaz de usuario: Mapa que ven los usuarios.

a la escalabilidad. A continuación, se describen los resultados obtenidos al operar simultáneamente múltiples Dispositivos Parkwise y Gateways asociados, así como las observaciones ante fallos parciales o condiciones atípicas de red.

7.8.1. Escenario con múltiples dispositivos y Gateways

Para esta prueba se configuraron exitosamente dos Dispositivos Parkwise localizados en dos ubicaciones completamente diferentes, y cada uno vinculado a un Gateway independiente. La interfaz web permitió verificar visualmente que ambos dispositivos estaban activos, con detecciones recibidas correctamente y proyectadas sobre sus respectivas veredas. En la Figura 7.12 puede observarse la visualización de ambos nodos sobre el mapa, mientras que en la Figura 7.13 se muestra el despliegue correcto de las detecciones proyectadas.

A pesar de que cada Gateway sólo establece conexión BLE con un único Parkwise por vez (limitación impuesta por el stack NimBLE), se demostró que el sistema puede escalar horizontalmente gracias al protocolo MQTT. Cada Gateway actúa como publicador independiente en un tópico común, e incluye la dirección MAC del Park-

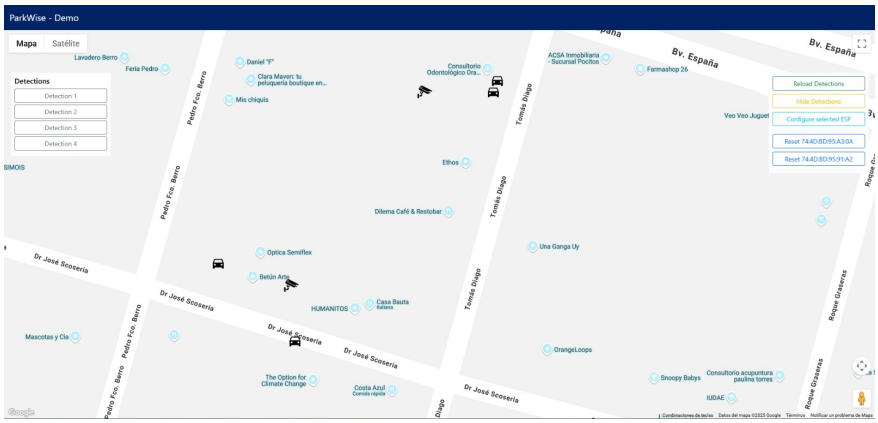


Figura 7.12: Visualización de múltiples Dispositivos Parkwise activos sobre el mapa.

Figura 7.13: Detecciones proyectadas correctamente desde uno de los Dispositivos Parkwise configurados.

wise en los mensajes JSON. Esta estructura permite que múltiples Gateways actúen en paralelo sin conflictos, y que un servidor centralizado procese eventos provenientes de diversos nodos de forma robusta y eficiente. Dado que MQTT soporta múltiples publicadores y múltiples suscriptores de forma asíncrona y distribuida, se considera que el sistema es escalable por diseño.

7.8.2. Tolerancia ante fallos parciales

Durante las pruebas, se simularon condiciones de fallo tales como reinicio forzado del Dispositivo Parkwise, desconexión intermitente del Gateway y corte temporal de energía. En todos los casos, se observó que el sistema respondía de forma controlada:

- **Reinicio del Parkwise:** Al reiniciar un nodo, su estado vuelve a inicializarse. Si se había registrado previamente una imagen mediante `operario.py`, el nodo aún aparece en la base de datos y puede ser reconfigurado desde la interfaz web.
- **Reconexión automática BLE:** Aunque no se implementó reconexión automática avanzada en los Gateways, estos pueden volver a vincularse manual-

mente con el Parkwise tras un reinicio, conservando su funcionalidad sin necesidad de reconfiguración desde cero.

- **Mensajes corruptos o ausentes:** Gracias al diseño minimalista y robusto del protocolo de transmisión (basado en JSON y BLE notifications), los errores de transmisión no detuvieron el funcionamiento general del sistema. Los mensajes con errores simplemente se descartan en el *backend*, sin colapsar el flujo general.

Estos resultados permiten concluir que, aún en su fase prototipo, Parkwise exhibe un comportamiento resiliente frente a fallos individuales de sus componentes, y una arquitectura que favorece la expansión escalonada de dispositivos en un entorno urbano distribuido.

Capítulo 8

Conclusiones

El desarrollo del sistema Parkwise permitió demostrar la viabilidad técnica de un sistema distribuido de detección vehicular en entornos urbanos, utilizando dispositivos de bajo consumo, inteligencia artificial embebida y tecnologías de comunicación inalámbrica. A lo largo de este trabajo se diseñaron, implementaron y validaron los distintos módulos del sistema, desde la captura de imágenes y detección en tiempo real, hasta la transmisión, visualización y configuración remota de los dispositivos.

La elección del microcontrolador ESP32-S3 y el uso de TinyML resultaron apropiados para realizar inferencias en el borde sin depender de conectividad constante o recursos externos. El modelo FOMO embebido demostró buen rendimiento en términos de velocidad y precisión, con tiempos de inferencia dentro de márgenes aceptables para aplicaciones en tiempo real. El sistema de tareas concurrentes bajo FreeRTOS permitió coordinar eficientemente los procesos de captura, inferencia, transmisión y control, mostrando estabilidad incluso bajo condiciones de operación continuada.

Desde el punto de vista de la comunicación, se logró integrar con éxito dos tecnologías complementarias: BLE para la configuración inicial desde un dispositivo cercano (operario o servidor local) y para la comunicación con un Gateway; y NB-IoT para la transmisión de eventos hacia un Servidor Central en la nube. Esta combinación permitió separar los flujos de datos según sus necesidades de alcance, consumo y latencia. Además, se validó la compatibilidad del sistema con el protocolo MQTT, lo que habilita un modelo escalable y flexible de publicación-suscripción para múltiples nodos funcionando simultáneamente.

La arquitectura web desarrollada facilitó tanto el monitoreo en tiempo real como la configuración remota de cada dispositivo. Se diseñó una interfaz gráfica intuitiva que permite definir las transformaciones necesarias (homografía) a partir de una imagen enviada por el dispositivo y establecer los segmentos de vereda a monitorear. Esta interfaz resultó clave para la usabilidad y correcta georreferenciación del sistema.

Las pruebas realizadas en campo confirmaron que Parkwise es capaz de detectar vehículos correctamente y proyectar las detecciones sobre segmentos de vereda definidos, utilizando coordenadas geográficas transformadas por la homografía. Asimismo, se evaluaron situaciones de tolerancia a fallos, como reinicios, pérdidas de conexión o latencias altas en la red NB-IoT. En todos los casos, el sistema mostró un comportamiento robusto y predecible, registrando eventos clave y recuperando el funcionamiento sin intervención manual. También se verificó que múltiples dispositivos pueden operar en paralelo sin interferencias, gracias al uso de MACs únicas y tópicos compartidos en MQTT, lo que refuerza el potencial del sistema para escalar a nivel urbano.

A partir de este trabajo, se puede continuar agregando funcionalidades que permi-

tan al dispositivo detectar e informar otros eventos, como estacionamiento indebido, contenedores de basura colapsados o incendiándose, detección de accidentes de tránsito o inundaciones. También se puede buscar incorporar otras fuentes de información para lograr detectar aún mejor los espacios libres, por ejemplo información de ferias vecinales y calles cerradas o incorporar zonas donde no se puede estacionar.

Existe además,la posibilidad de devolver esta información como un servicio para aplicaciones que indican cómo llegar a un lugar, de esta manera, en lugar de marcar a donde queremos ir, pueden marcar el lugar mas cercano a destino para estacionar.

En resumen, la implementación de Parkwise permitió consolidar múltiples tecnologías en una solución funcional y replicable. La modularidad del sistema, su capacidad de configuración remota y su bajo consumo lo convierten en una alternativa viable para aplicaciones de monitoreo urbano inteligente, con potencial de expansión hacia múltiples funcionalidades futuras.

Apéndice A

Curso de Proyecto

En el presente apéndice se describe la planificación del proyecto realizada al comienzo de la tesis, durante el curso de Gestión. En él se presentan el objetivo, alcance, criterios de éxito, supuestos y restricciones, así como algunas definiciones iniciales que sirvieron para delimitar el proyecto.

Se pueden apreciar detalles que reflejan el entusiasmo del equipo en esa etapa, cuando se concebía una idea interesante y ambiciosa, con la expectativa de desarrollar un modelo punta a punta capaz de ser presentado a actores interesados en su adquisición.

Si bien aún quedan desafíos por resolver para llegar a ese punto, el avance logrado a lo largo del proyecto permite afirmar que dichas metas son técnicamente viables. Este progreso confirma la validez del enfoque inicial y sienta las bases para próximos desarrollos y aplicaciones prácticas.

A.1. Objetivo del proyecto

Construir la prueba de concepto de un sistema capaz de medir y reportar espacios libres entre vehículos estacionados en la vía pública. El sistema, desplegado en un conjunto mayor, será capaz de relevar en un mapa o interfaz web la representación de los tramos de espacios libres y ocupados para cada vereda.

A.2. Alcance

El proyecto abarca la elección, desarrollo y programación del sistema embebido junto a sus periféricos, así como la implementación y testeo de rendimiento de algoritmos de "Computer Vision" para el reconocimiento de vehículos y espacios libres. También se incluirá la comunicación para transmitir los datos a un servidor en tiempo real. El dispositivo debe estar en un rango de precios que justifique el despliegue de muchos de ellos distribuidos en una zona urbana.

A.3. Criterios de Éxito

Se define "vehículo estacionado" como aquel que permanece próximo a la vereda o región de estacionamiento sin estar en movimiento por al menos 1 minuto.

A.3.1. Software

Precisión razonable para detección de vehículos y espacios libres, esto es:

- Lograr detectar el número vehículos estacionados con el algoritmo con accuracy del 90 % o superior.
- Deberá poder detectar autos estacionados a ambos lados de la calle.
- Adaptabilidad a diferentes entornos geográficos urbanos.
- Diseño de un Frontend capaz de desplegar estas detecciones en un mapa, que eventualmente pueda ser accesible para los clientes.

A.3.2. Comunicación

- Transmisión efectiva de datos.
- Transmisión en tiempo real eficiente, esto es:
  - Reporte de la posición de los vehículos estacionados en un tiempo menor a 2 minutos, permitiendo además recibir en el servidor las imágenes capturadas por el dispositivo para dicho reporte.
  - Latencia menor a 2 minutos entre reportes.
  - Será capaz de reportar y transmitir los datos (posiciones ocupadas e imágenes) de estacionamiento post procesados de al menos un nodo en tiempo real.
- Comunicación estable tal que permita el cumplimiento de los puntos de eficiencia previamente dichos.
- La distancia de transmisión deberá ser de al menos 25 metros en línea recta sin obstrucciones para las imágenes y de 600 metros para el reporte de las posiciones.

A.3.3. Hardware

- Funcionamiento estable y confiable del sistema embebido.
- Carcasa capaz de resistir condiciones exteriores.
- Soporte adherido a la columna o pared estable, firme, que permita el correcto funcionamiento del sistema.

A.4. Supuestos y Restricciones

- Disponibilidad de espacios públicos con flujo de estacionamientos para realizar pruebas.
- Existencia de algoritmos y dispositivos de hardware con arquitecturas que permitan ejecutarlos.
- Acceso permanente y gratuito a las bases de datos y modelos existentes para identificación de vehículos automovilísticos.
- Inexistencia de normativas de privacidad que limiten la realización del proyecto.
- Limitaciones de presupuesto.
- Limitación temporal a 1 año.
- Cumplimiento de regulaciones y normativas de seguridad y privacidad actuales.

- Disponibilidad de espacio y recursos para la instalación del sistema en la vía pública.

El proyecto consta en desarrollar un dispositivo partiendo de un sistema embebido con recursos limitados, que a través de computer vision, sea capaz de detectar vehículos estacionados contra las veredas de las calles, y reportar los espacios libres entre ellos. Es importante aclarar, que esta iniciativa busca construir una prueba de concepto para validar la viabilidad técnica y funcional del sistema propuesto, en lugar de desarrollar un dispositivo completo desde cero hasta su etapa final en producción. Se detallan a continuación una serie de especificaciones funcionales:

- Detección de vehículos estacionados: Utilizando algoritmos de computer vision, el dispositivo deberá identificar la presencia de vehículos estacionados en las veredas de ambos lados de las calles.
- Identificación de espacios libres: Una vez detectados los vehículos estacionados, el dispositivo deberá delimitar y reportar los espacios libres entre ellos de manera precisa y efectiva.
- Procesamiento en tiempo real: Se requerirá que el sistema realice el procesamiento de las imágenes en tiempo real, para proporcionar información actualizada sobre la disponibilidad de espacios de estacionamiento.
- Comunicación de datos: El dispositivo deberá comunicar eficientemente los datos sobre la disponibilidad de espacios de estacionamiento, ya sea mediante una conexión inalámbrica u otro medio de comunicación adecuado. En caso de ser necesario enviar una imagen sin procesar para verificación manual de datos.
- Robustez y precisión: Se espera que el sistema sea robusto y preciso en la detección y delimitación de vehículos y espacios libres, incluso bajo condiciones adversas como variaciones de iluminación o cambios en el entorno.

Esta página ha sido intencionalmente dejada en blanco.

Apéndice B

Material de Referencia

En este apéndice se describen los distintos materiales de referencia que, si bien no encontraron su lugar en el cuerpo principal de la tesis, merecen ser mencionados. Se trata, en su mayoría, de recursos que ayudaron a entender cómo llevar una red neuronal entrenada en una computadora hasta un dispositivo embebido que se programa en `C` o `Arduino`, por ejemplo. En su momento, resultaba casi mágico ver cómo todas las piezas, comenzaban a encajar unas con otras, el universo embebido dejaba ver su propio orden secreto. Por eso, estimado lector, dejamos aquí este compendio de materiales, con la esperanza de que, en algún momento, alguien curioso los redescubra y los pueda aprovechar tanto como nosotros.

B.1. Person Detection

Tras dominar los ejemplos básicos, identificamos el repositorio `esp-tflite-micro` de Espressif, que incluye un modelo pre-entrenado para detección de personas (*Person Detection*). Este proyecto, desarrollado específicamente para chipsets Espressif, resultó de especial relavancia por lo siguiente:

- Su arquitectura Optimizada:
  - Integración nativa con ESP-IDF mediante el componente `esp-tflite-micro`
  - Uso de ESP-NN, una biblioteca de Kernel acelerados para operaciones de ML (reduce el tiempo de Inferencia de 2300 ms a 54 ms en la ESP32-S3). [39]
  - Soporte para memoria PSRAM y cuantización a 8 bits, esencial para manejar imágenes de la cámara
- Flujo de Trabajo Reproducible:
  El repositorio incluye una secuencia de comandos para instalar dependencias, crear un proyecto de ejemplo, configurar el objetivo de compilación y ejecutar el programa en el dispositivo. Esto nos permitió ejecutar el modelo con solo ajustar parámetros básicos.

Limitaciones y Adaptaciones:
Si bien el modelo original funcionó correctamente, tuvimos que:

- Compatibilizarlo con la cámara OV2640 (el ejemplo usa imágenes de 96×96 píxeles en escala de grises)

- Reducir su consumo de memoria (∼150 KB de *tensor arena*), reasignando buffers dinámicamente

Este repositorio no solo aceleró nuestro desarrollo, sino que también sirvió como referencia para diseñar soluciones personalizadas (ver Capítulo ...referenciar el capitulo que corresponda cuando terminemos de escribirlo).

Apéndice C

Avance en la Implementación del Gateway

A continuación se describe el avance en las pruebas de implementación del Gateway. En este anexo se deja constancia, de forma breve, de los intentos realizados para darle un poco más de "color" a la tesis, incorporando un puñado de dispositivos que tendrían la tarea de personificar al Gateway.

Este proceso fue, en muchos momentos, trabajoso. Pero más trabajoso aún fue dejarlo en pausa. Por eso, a nuestro pesar, describimos en los próximos párrafos las pruebas realizadas, con la esperanza de que, en algún momento, podamos retomarlas y completarlas como se merecen.

En este trabajo se lograron implementar correctamente las dos primeras funciones: la lectura y escritura sobre características GATT mediante BLE. Sin embargo, las funciones asociadas al uso de comandos AT para enviar y recibir datos por MQTT a través del módulo *LTE IoT 2 Click* no pudieron ser ejecutadas exitosamente sobre el Arduino Nano 33 BLE.

Con el objetivo de identificar la causa del problema (y resistiendo la tentación de culpar directamente al hardware), se realizó una prueba alternativa utilizando un Arduino Zero junto con una placa adaptadora *mikroBUS*, comercializada por Arduino. Tanto el Arduino Zero como el Nano 33 BLE cuentan con dos interfaces de comunicación: UART (utilizada para la conexión por USB) e I2C (utilizada para la conexión con el módulo LTE mediante la placa adaptadora).

La prueba con el Arduino Zero fue exitosa: se logró establecer comunicación con el módulo mediante comandos AT y publicar datos en un tópico MQTT, demostrando así que las funciones requeridas podían ejecutarse correctamente dentro del entorno Arduino. Esto sugiere que las dificultades encontradas con el Nano 33 BLE podrían deberse a limitaciones específicas de hardware, conflictos con las interfaces de comunicación internas, o algún detalle que —como suele pasar— no figura en la hoja de datos.

Esta página ha sido intencionalmente dejada en blanco.

Apéndice D

Exploración Inicial con TinyML

D.0.1. Introducción: Predecir Valores de la Función Seno

Como parte de la investigación preliminar, se revisó el libro TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers, de Pete Warden y Daniel Situnayake [42]. Este libro introduce conceptos fundamentales y ejemplos prácticos para el entrenamiento y despliegue de modelos de aprendizaje automático en dispositivos con recursos altamente limitados. En esta sección, se toma ese ejemplo a modo de introducción al proceso de realizar aprendizaje automático en sistemas embebidos.

En el capítulo 4, se presenta un ejemplo introductorio que consiste en predecir los valores de la función seno mediante una red neuronal completamente conectada. Aunque el problema posee una solución analítica exacta, su implementación sirve como punto de partida para ilustrar el flujo completo del desarrollo en el contexto de TinyML: desde el entrenamiento y evaluación del modelo utilizando la API Keras, hasta su conversión con TensorFlow Lite y posterior ejecución en un microcontrolador.

Para este experimento, se generó un conjunto de valores aleatorios para la variable x, a partir de los cuales se calcularon sus correspondientes valores de seno. Dado que estos datos fueron generados a partir de una función continua conocida, la distribución resultante presentó una curva suave y bien definida, lo que la convierte en un escenario teóricamente ideal.

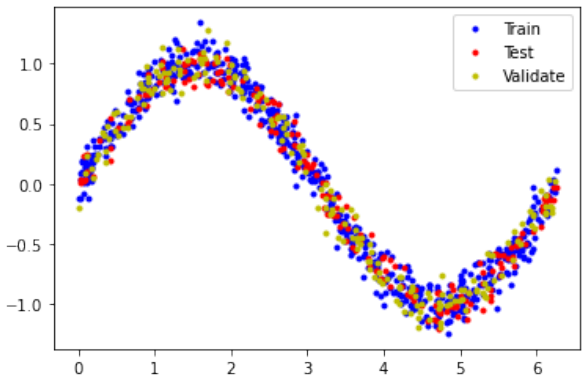


Figura D.1: Distribución de los datos de entrenamiento, validación y prueba.

Sin embargo, con el objetivo de simular condiciones más representativas de aplicaciones reales —donde los datos suelen estar afectados por ruido e incertidumbre— se introdujo una perturbación aleatoria controlada sobre los valores originales. Esto permitió crear un conjunto de datos más "realista" que posibilita evaluar la capacidad del modelo para extraer patrones subyacentes en presencia de distorsiones.

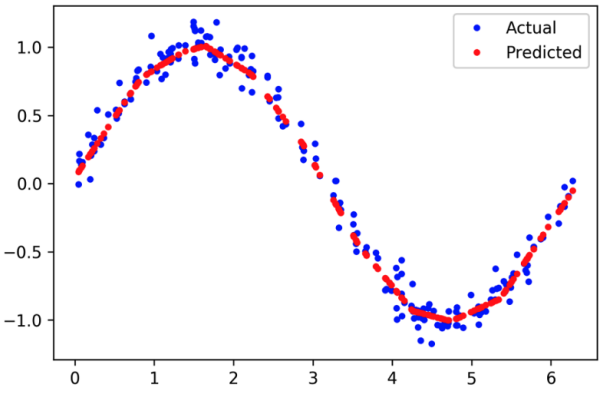


Figura D.2: Comparación entre los valores reales y las predicciones del modelo entrenado.

Desde el punto de vista del aprendizaje automático, este ejemplo resulta sencillo: crear, entrenar y evaluar una red densa completamente conectada con la API Keras requiere apenas unas pocas líneas de código en Python. En el siguiente fragmento se muestra un ejemplo básico de red neuronal entrenada para aproximar la función seno:

```
import numpy as np
from tensorflow import keras

# Datos de entrenamiento
x_train = np.linspace(-2 * np.pi, 2 * np.pi, 1000)
y_train = np.sin(x_train)

# Definicion del modelo
model = keras.Sequential([
    keras.layers.Input(shape=(1,)),
    keras.layers.Dense(16, activation='relu'),
    keras.layers.Dense(1)
])

# Compilacion y entrenamiento
model.compile(optimizer='adam', loss='mse')
model.fit(x_train, y_train, epochs=50)

```

Listing D.1: Red neuronal totalmente conectada en Keras para aproximar la función seno

Lo novedoso de este ejercicio reside no tanto en la complejidad del modelo, sino en la introducción de herramientas diseñadas para ejecutar modelos en dispositivos al borde (*edge devices*), en particular dos componentes fundamentales: el *TensorFlow Lite Converter* y el *TensorFlow Lite Interpreter*. Estas herramientas permiten transformar el modelo entrenado en un formato optimizado para microcontroladores, facilitando su implementación en entornos con recursos limitados.

D.0.2. Funcionamiento de TensorFlow Lite Converter e Interpreter

Antes de utilizar un modelo en un entorno con TensorFlow Lite, es necesario convertirlo a un formato compatible. Este proceso se realiza mediante la API o la interfaz de línea de comandos de *TensorFlow Lite Converter*, que toma como entrada un modelo entrenado y lo transforma en un archivo `.tflite`, basado en el formato binario *FlatBuffer*. Este formato está diseñado para reducir el uso de memoria y facilitar la integración en dispositivos embebidos.

Durante la conversión es posible aplicar técnicas de optimización. Una de las más relevantes es la cuantización, que reduce la precisión de los parámetros del modelo, pasando de representaciones en punto flotante de 32 bits a enteros de 8 bits. Esta transformación no solo disminuye el tamaño del modelo, sino que también mejora el tiempo de inferencia en hardware optimizado para operaciones enteras. Para aplicar cuantización posterior al entrenamiento de manera efectiva, se requiere un conjunto de datos representativo que abarque el rango de entradas esperado.

Una vez convertido y optimizado, el modelo es ejecutado mediante el *TensorFlow Lite Interpreter*. A diferencia de la interfaz de alto nivel de Keras, que permite realizar predicciones con una simple llamada a `predict()`, el uso del *interpreter* implica una secuencia más detallada de pasos. Esta estructura, sin embargo, está pensada para permitir la ejecución eficiente en entornos con recursos limitados, sin comprometer significativamente la precisión del modelo.

D.0.3. Conversión Final para uso en Microcontroladores

El paso final para permitir la ejecución del modelo en microcontroladores consiste en integrarlo como parte del código fuente de la aplicación. Hasta este punto, se ha trabajado con el modelo como un archivo externo, cargado en memoria por el Interpreter. No obstante, en muchos microcontroladores no existe un sistema de archivos disponible, o bien, su implementación es costosa desde el punto de vista del espacio y el rendimiento.

Para superar esta limitación, se recurre a una técnica que permite incluir el modelo directamente como un arreglo de bytes en lenguaje C. Una opción común consiste en utilizar la herramienta de Unix `xxd`, que transforma el archivo binario resultante de la conversión en una definición en `C` que puede ser compilada junto al resto del firmware. Alternativamente, existen mecanismos específicos para microprocesadores, como la macro INCBIN, que permiten incluir directamente archivos binarios —como un modelo en formato `.tflite` — sin necesidad de convertirlos previamente.

Estos enfoques eliminan la necesidad de realizar operaciones de lectura en tiempo de ejecución, reducen el uso de recursos y permiten una integración más eficiente del modelo dentro del entorno embebido.

Esta página ha sido intencionalmente dejada en blanco.

Referencias

[1]

[2] Giuseppe Amato, Fabio Carrara, Fabrizio Falchi, Claudio Gennaro, Carlo Meghini, and Claudio Vairo. Deep learning for decentralized parking lot occupancy detection. *Expert Systems with Applications*, 72:327–334, 2017.

[3] Arduino. Arduino nano 33 ble, 2025. Consultado el 20 de agosto de 2025. URL: `https://store.arduino.cc/products/nano-33-ble`.

[4] Andrew Banks, Ed Briggs, Ken Borgendale, and Rahul Gupta. *MQTT Version 5.0*. OASIS, March 2019. OASIS Standard, Accesed: 2024-12-15. URL: `https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html`.

[5] Bleak contributors. Bleak — bluetooth low energy platform agnostic client for python, 2025. Consultado el 22 de mayo de 2025. URL: `https://bleak.readthedocs.io`.

[6] Bluetooth Special Interest Group. *The Bluetooth® Low Energy Primer*, May 2022. Accessed: 2024-12-12. URL: `https://www.bluetooth.com/wp-content/uploads/2022/05/the-bluetooth-le-primer-v1.2.0.pdf`.

[7] Baiyu Cai, Roberto Álvarez, Michael Sit, Fabio Duarte, and Carlo Ratti. Deep learning based video system for accurate and real-time parking measurement. arXiv preprint arXiv:1902.07401, 2019. URL: `https://arxiv.org/abs/1902.07401`.

[8] Eclipse Foundation. Eclipse paho mqtt python client library, 2024. Accessed: 2025-06-24. URL: `https://github.com/eclipse/paho.mqtt.python`.

[9] Edge Impulse. Edge impulse: The leading development platform for ml on edge devices, 2025. Consultado el 20 de agosto de 2025. URL: `https://www.edgeimpulse.com/`.

[10] Edge Impulse. Fomo object detection for constrained devices, 2025. URL: `https://docs.edgeimpulse.com/docs/edge-impulse-studio/learning-blocks/object-detection/fomo-object-detection-for-constrained-devices`.

[11] Espressif Systems. Arduino core for the esp32. `https://github.com/espressif/arduino-esp32`, 2025. Consultado el 20 de agosto de 2025.

[12] Espressif Systems. Esp-idf programming guide, 2025. Consultado el 20 de agosto de 2025. URL: `https://docs.espressif.com/projects/esp-idf/`.

[13] Saurabh Kumar et al. python-dotenv, 2025. Consultado el 22 de mayo de 2025. URL: `https://pypi.org/project/python-dotenv/`.

[14] Eclipse Foundation. Eclipse paho mqtt python client, 2025. Consultado el 22 de mayo de 2025. URL: `https://www.eclipse.org/paho/index.php?page=clients/python/index.php`.

[15] Python Software Foundation. asyncio — asynchronous i/o, 2025. Consultado el 22 de mayo de 2025. URL: https://docs.python.org/3/library/asyncio.html.

[16] F. Fresard. Efecto real de la restricción vehicular en santiago de chile. In *Ingeniería de Tránsito y Transporte. Actas del X Congreso Panamericano*, pages 299–308, Santiago, Chile, 1998. Ministerio de Fomento, Secretaría General Técnica. URL: https://trid.trb.org/View/964049.

[17] H.B. Møller et al. Bleak: Bluetooth low energy platform agnostic klient, 2024. Accessed: 2025-06-24. URL: https://github.com/hbldh/bleak.

[18] Meng-Ru Hsieh, Yen-Liang Lin, and Winston H Hsu. Drone-based object counting by spatially regularized regional proposal network. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4145–4153, 2017.

[19] Edge Impulse. Ai labeling, 2023. URL: https://docs.edgeimpulse.com/docs/edge-impulse-studio/data-acquisition/ai-labeling.

[20] Edge Impulse. Labeling queue, 2023. URL: https://docs.edgeimpulse.com/docs/edge-impulse-studio/data-acquisition/labeling-queue.

[21] Instituto Nacional de Estadística (Uruguay). Censo 2023: Crecen los hogares unipersonales. https://www.gub.uy/instituto-nacional-estadistica/comunicacion/noticias/censo-2023-crecen-hogares-unipersonales, 2024. Accedido el 24 de junio de 2025.

[22] Glenn Jocher et al. Yolov5 - you only look once version 5. https://github.com/ultralytics/yolov5, 2020.

[23] Ashok Kumar and Abhishek Singh. Parking management system based on key points detection. *Applied Engineering Letters*, 8(1):15–20, 2023.

[24] Akihiko Matsuda and Takashi Matsui. A system for real-time on-street parking detection and visualization on an edge device. In *2021 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 1–4. IEEE, 2021.

[25] Santiago Pérez Michael Dickinson, Andrés Guido. Sistema inalámbrico distribuido para la detección y mapeo urbano de vehículos (parkwise). https://gitlab.fing.edu.uy/parkwiseguys/park-wise-pfc, 2025. Repositorio oficial del proyecto de tesis de grado - Facultad de Ingeniería, Universidad de la República (Uruguay).

[26] MikroElektronika. Lte iot 2 click, 2025. Consultado el 20 de agosto de 2025. URL: https://www.mikroe.com/lte-iot-2-click.

[27] Ministerio de Industria, Energía y Minería (Uruguay). Parque automotor. https://www.gub.uy/ministerio-industria-energia-mineria/datos-y-estadisticas/estadisticas/parque-automotor, 2024. Accedido el 24 de junio de 2025.

[28] Narrowband.com. What is narrowband iot: a comprehensive guide, 2024. Accesed: 2024-12-15. URL: https://www.narrowband.com/what-is-narrowband-iot.

[29] Neuton.AI. Neuton: Tinyml platform without code. https://neuton.ai, 2023. Accessed: 2025-06-14.

[30] OASIS. Mqtt version 5.0, 2019. Estándar oficial del protocolo MQTT. Consultado el 20 de agosto de 2025. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html.

[31] OpenAI, Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, and et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

[32] Parkwise. Adaptation of car parking occupancy detection - fomo, 2025. Accessed: 31 August 2025. URL: https://studio.edgeimpulse.com/studio/637423.

[33] Marcelo Rovai. Xiao-esp32s3-sense, 2024. Consultado el 20 de mayo de 2025. URL: https://github.com/Mjrovai/XIAO-ESP32S3-Sense.

[34] San Francisco Municipal Transportation Agency (SFMTA). San francisco adopts demand-responsive pricing program to make parking easier. https://www.sfmta.com/blog/san-francisco-adopts-demand-responsive-pricing-program-make-parking-easier, 2023. Accessed: 2025-06-24.

[35] SensiML Corporation. Sensiml analytics toolkit. https://sensiml.com, 2023. Accessed: 2025-06-14.

[36] Bluetooth SIG. Generic access profile (gap) – bluetooth core specification v5.4, 2023. Consultado el 21 de mayo de 2025. URL: https://www.bluetooth.com/wp-content/uploads/Files/Specification/HTML/Core-54/out/en/host/generic-access-profile.html.

[37] Espressif Systems. Bluetooth le introduction - esp-idf programming guide, 2024. Accessed: 2024-12-18. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/ble/get-started/ble-introduction.html.

[38] Espressif Systems. esp-nn: Optimized neural network functions for esp32, 2024. Consultado el 19 de mayo de 2025. URL: https://github.com/espressif/esp-nn.

[39] Espressif Systems. Esp-nn: Optimized neural network kernels for esp32, 2024. Última consulta el 28 de marzo de 2025. URL: https://github.com/espressif/esp-tflite-micro.

[40] Espressif Systems. Nimble gatt server example (esp-idf), 2024. Consultado el 21 de mayo de 2025. URL: https://github.com/espressif/esp-idf/tree/27d68f57/examples/bluetooth/ble_get_started/nimble/NimBLE_GATT_Server.

[41] TensorFlow. Tensorflow lite para microcontroladores, 2025. URL: https://www.tensorflow.org/lite/microcontrollers?hl=es-419.

[42] Pete Warden and Daniel Situnayake. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media, Inc., 2019. URL: https://www.oreilly.com/library/view/tinyml/9781492051992/.

[43] Yuxuan Zhou and Hui Wang. Deep learning solutions for smart city challenges in urban development. *Scientific Reports*, 13(1):1–12, 2023.

Esta página ha sido intencionalmente dejada en blanco.

Glosario

al borde Término que hace referencia al procesamiento de datos realizado directamente en dispositivos cercanos al origen de los datos, como sensores o microcontroladores con capacidades TinyML, evitando el envío a la nube. 124

argmax Operador que devuelve el conjunto de valores del dominio de una función donde esta alcanza su valor máximo. . 34

BLE Bluetooth Low Energy (BLE) es una tecnología de comunicación inalámbrica diseñada para operar con un consumo de energía extremadamente bajo. Utiliza 40 canales de datos en la banda ISM de 2.4 GHz y se divide en dos bloques principales: el *host* y el *controller*. Estos bloques están separados lógicamente para permitir su implementación por distintos fabricantes, lo que facilita la interoperabilidad entre dispositivos. La arquitectura de BLE se compone de varias capas, como se muestra en la Figura 5.2. . VII, 4, 6, 85

centroide Punto que representa el centro geométrico de una figura o conjunto de puntos, calculado como la media aritmética de sus coordenadas. `https://es.wikipedia.org/wiki/Centroide` . 34

Edge AI Inteligencia artificial que se ejecuta en dispositivos locales (en el "edge" de la red) en lugar de en la nube. . VII

embeddings Representaciones vectoriales de datos (como palabras, imágenes o categorías) en un espacio de menor dimensión, diseñadas para capturar relaciones semánticas o estructurales relevantes para tareas de aprendizaje automático. 96, 97, 143

ESP32-S3 Microcontrolador de la familia ESP32 desarrollado por Espressif Systems. Incorpora conectividad Wi-Fi y Bluetooth Low Energy, y está optimizado para aplicaciones de visión artificial, incluyendo una unidad de aceleración vectorial para operaciones de inferencia. . VII

Homografía Transformación geométrica que mapea puntos de una imagen a otra, utilizada para corregir perspectivas en visión por computadora. . VII

Inferencia Proceso en el que un modelo de machine learning realiza predicciones basadas en datos de entrada. . 119

Keras Una biblioteca de código abierto para redes neuronales escrita en Python. Proporciona una interfaz simple y modular para construir y entrenar modelos de aprendizaje profundo, y es compatible con múltiples backends como TensorFlow. 20, 124

Kernel Función optimizada que realiza cálculos específicos en algoritmos de machine learning. . 119

LoRaWAN Es una especificación de red de bajo consumo y cobertura amplia (LPWA por siglas en inglés) permite comunicaciones a larga distancia para dispositivos alimentados por batería, con bajo impacto en su consumo. La banda en la que opera no tiene licencia y permite comunicar múltiples dispositivos con distancias de hasta 15 km en zonas despejadas y sin interferencias. . 15

mapa de segmentación Representación visual que asigna una etiqueta o clase a cada píxel de una imagen, indicando a qué objeto o región pertenece. Es el resultado de aplicar técnicas de segmentación de imágenes, como la segmentación semántica o de instancias, y se utiliza en tareas de visión por computadora para identificar y diferenciar elementos dentro de una escena. . 34

MCU Unidad de Control de Microprocesador (por sus siglas en inglés, *Microcontroller Unit*), un circuito integrado que contiene un procesador, memoria y periféricos de entrada/salida, utilizado para controlar dispositivos electrónicos en sistemas embebidos. 22

Microcontrolador Circuito integrado programable que contiene una unidad de procesamiento, memoria y periféricos de entrada/salida. Es ampliamente utilizado en sistemas embebidos para controlar tareas específicas. . vii

MQTT MQTT es un protocolo abierto para transmisión de mensajes entre un Cliente y un Servidor en la modalidad publicar/suscribir. Es muy popular para aplicaciones IoT. El protocolo se lleva realiza sobre protocolos de red bidireccionales sin pérdida como TCP/IP. Entre sus características, permite elegir la cantidad de veces que cada mensaje es enviado, tiene poco *overhead* en el protocolo, por lo que el tráfico de red es reducido, haciéndolo muy útil para conexiones costosas a distancia, también permite avisar cuando hay una desconexión inesperada. [4] MQTT funciona con *brokers* que ofician de intermediarios para permitir a los clientes MQTT comunicarse entre si. Existen dos tipos de clientes: *Publishers*, que publican información y *Suscribers*, que reciben los datos. . 6

NB-IoT Narrowband IoT (NB-IoT) es una tecnología de bajo consumo y área amplia (LWPA) que permite la conexión de dispositivos y servicios en IoT. Tiene un ancho de banda de 200kHz alrededor de las unidades de MHz, ofrece mas canales que LTE o Wi-Fi. Es ideal para dispositivos que funcionan fijos en un lugar determinado, que deben mandar pequeñas cantidades de información. [28]. . vii, 15, 85

RTOS (Real Time Operating System) Sistema operativo en tiempo real que se ejecuta en el dispositivo para gestionar y ejecutar aplicaciones, considerado un tipo de Program Layer. . 12

SoC System on a Chip. Refiere a un sistema integrado, en el mismo se encuentran todos o casi todos los componentes con los que interactúa el microprocesador. . 4

TensorFlow Lite Micro TensorFlow Lite Micro es la versión ligera de TensorFlow diseñada para dispositivos embebidos con recursos muy restrictivos. . vii, 12, 13, 19

TinyML Tiny Machine Learning (TinyML) es una rama del aprendizaje automático que se enfoca en ejecutar modelos directamente en dispositivos con recursos muy limitados, como microcontroladores. Permite realizar tareas de inferencia sin conexión a la nube, con bajo consumo energético. . VII, 20

Esta página ha sido intencionalmente dejada en blanco.

Acrónimos

al borde Término que hace referencia al procesamiento de datos realizado directamente en dispositivos cercanos al origen de los datos, como sensores o microcontroladores con capacidades TinyML, evitando el envío a la nube. 124

argmax Operador que devuelve el conjunto de valores del dominio de una función donde esta alcanza su valor máximo. . 34

BLE Bluetooth Low Energy (BLE) es una tecnología de comunicación inalámbrica diseñada para operar con un consumo de energía extremadamente bajo. Utiliza 40 canales de datos en la banda ISM de 2.4 GHz y se divide en dos bloques principales: el *host* y el *controller*. Estos bloques están separados lógicamente para permitir su implementación por distintos fabricantes, lo que facilita la interoperabilidad entre dispositivos. La arquitectura de BLE se compone de varias capas, como se muestra en la Figura 5.2. . VII, 4, 6, 85

centroide Punto que representa el centro geométrico de una figura o conjunto de puntos, calculado como la media aritmética de sus coordenadas. `https://es.wikipedia.org/wiki/Centroide` . 34

Edge AI Inteligencia artificial que se ejecuta en dispositivos locales (en el "edge" de la red) en lugar de en la nube. . VII

embeddings Representaciones vectoriales de datos (como palabras, imágenes o categorías) en un espacio de menor dimensión, diseñadas para capturar relaciones semánticas o estructurales relevantes para tareas de aprendizaje automático. 96, 97, 143

ESP32-S3 Microcontrolador de la familia ESP32 desarrollado por Espressif Systems. Incorpora conectividad Wi-Fi y Bluetooth Low Energy, y está optimizado para aplicaciones de visión artificial, incluyendo una unidad de aceleración vectorial para operaciones de inferencia. . VII

Homografía Transformación geométrica que mapea puntos de una imagen a otra, utilizada para corregir perspectivas en visión por computadora. . VII

Inferencia Proceso en el que un modelo de machine learning realiza predicciones basadas en datos de entrada. . 119

Keras Una biblioteca de código abierto para redes neuronales escrita en Python. Proporciona una interfaz simple y modular para construir y entrenar modelos de aprendizaje profundo, y es compatible con múltiples backends como TensorFlow. 20, 124

Kernel Función optimizada que realiza cálculos específicos en algoritmos de machine learning. . 119

LoRaWAN Es una especificación de red de bajo consumo y cobertura amplia (LPWA por siglas en inglés) permite comunicaciones a larga distancia para dispositivos alimentados por batería, con bajo impacto en su consumo. La banda en la que opera no tiene licencia y permite comunicar múltiples dispositivos con distancias de hasta 15 km en zonas despejadas y sin interferencias. . 15

mapa de segmentación Representación visual que asigna una etiqueta o clase a cada píxel de una imagen, indicando a qué objeto o región pertenece. Es el resultado de aplicar técnicas de segmentación de imágenes, como la segmentación semántica o de instancias, y se utiliza en tareas de visión por computadora para identificar y diferenciar elementos dentro de una escena. . 34

MCU Unidad de Control de Microprocesador (por sus siglas en inglés, *Microcontroller Unit*), un circuito integrado que contiene un procesador, memoria y periféricos de entrada/salida, utilizado para controlar dispositivos electrónicos en sistemas embebidos. 22

Microcontrolador Circuito integrado programable que contiene una unidad de procesamiento, memoria y periféricos de entrada/salida. Es ampliamente utilizado en sistemas embebidos para controlar tareas específicas. . vii

MQTT MQTT es un protocolo abierto para transmisión de mensajes entre un Cliente y un Servidor en la modalidad publicar/suscribir. Es muy popular para aplicaciones IoT. El protocolo se lleva realiza sobre protocolos de red bidireccionales sin pérdida como TCP/IP. Entre sus características, permite elegir la cantidad de veces que cada mensaje es enviado, tiene poco *overhead* en el protocolo, por lo que el tráfico de red es reducido, haciéndolo muy útil para conexiones costosas a distancia, también permite avisar cuando hay una desconexión inesperada. [4] MQTT funciona con *brokers* que ofician de intermediarios para permitir a los clientes MQTT comunicarse entre si. Existen dos tipos de clientes: *Publishers*, que publican información y *Suscribers*, que reciben los datos. . 6

NB-IoT Narrowband IoT (NB-IoT) es una tecnología de bajo consumo y área amplia (LWPA) que permite la conexión de dispositivos y servicios en IoT. Tiene un ancho de banda de 200kHz alrededor de las unidades de MHz, ofrece mas canales que LTE o Wi-Fi. Es ideal para dispositivos que funcionan fijos en un lugar determinado, que deben mandar pequeñas cantidades de información. [28]. . vii, 15, 85

RTOS (Real Time Operating System) Sistema operativo en tiempo real que se ejecuta en el dispositivo para gestionar y ejecutar aplicaciones, considerado un tipo de Program Layer. . 12

SoC System on a Chip. Refiere a un sistema integrado, en el mismo se encuentran todos o casi todos los componentes con los que interactúa el microprocesador. . 4

TensorFlow Lite Micro TensorFlow Lite Micro es la versión ligera de TensorFlow diseñada para dispositivos embebidos con recursos muy restrictivos. . vii, 12, 13, 19

TinyML Tiny Machine Learning (TinyML) es una rama del aprendizaje automático que se enfoca en ejecutar modelos directamente en dispositivos con recursos muy limitados, como microcontroladores. Permite realizar tareas de inferencia sin conexión a la nube, con bajo consumo energético. . VII, 20

Esta página ha sido intencionalmente dejada en blanco.

Índice de tablas

2.1. Comparativa de dispositivos 11
2.2. Funciones requeridas para la operación del Gateway 17

3.1. Comparación entre TensorFlow Lite y TensorFlow Lite for Microcontrollers 29
3.2. Comparación de métricas entre modelo FLOAT32 e INT8 35

7.1. Comparación de métricas entre modelo FLOAT32 e INT8 96

Esta página ha sido intencionalmente dejada en blanco.

Índice de figuras

1.1. Acumulado de vehículos en Uruguay y la proporción de vehículos cada mil habitantes.Fuente [27]. 1
1.2. Diagrama de la arquitectura física. Se muestran los vínculos inalámbricos entre los componentes. El instalador, con conexiones en verde, estará únicamente al inicio. 3
1.3. Seeed XIAO ESP32S3 (Derecha) y el módulo Sense (Izquierda) 4
1.4. Representación entrada-salida de la red. 5
1.5. Diagrama de flujo de información del Sistema Parkwise. 7
1.6. Image View en modo instalador: despliega la imagen recibida del Operario, con los puntos de referencia en la imagen, que se corresponderán en el mapa. 7
1.7. Vista del mapa en modo Instalador, con la correspondencia entre puntos de la imagen y del entorno. En el recuadro azul (abajo a la derecha) se amplía la zona de interés. 7
1.8. Interfaz de usuario: Mapa que ven los usuarios. 8

2.1. Seeed XIAO ESP32S3 desarmado, vista de sus componentes 12
2.2. Dimensiones de Seeed XIAO ESP32S3 12
2.3. Diagrama del Gateway y sus conexiones 16
2.4. Hardware del Gateway 16

3.1. Ejemplo de imágenes de entrenamiento junto con sus anotaciones en formato `xywh` (Edge Impulse Toy Dataset) 22
3.2. Ejemplos de imágenes del conjunto de datos PUCPR+ (izquierda) y CARPK (derecha), con sus respectivas anotaciones de cajas delimitadoras. 23
3.3. Ejemplo de imágenes de entrenamiento junto con sus anotaciones en formato `xywh` (Tomas Diago Dataset) 26
3.4. Ejemplo de imágenes de entrenamiento junto con sus anotaciones en formato `xywh` (Dron Dataset) 27
3.5. Plataforma Edge impulse 30
3.6. Representación entrada-salida de la red. 32
3.7. Centroides 'ground-truth' y 'pred' junto con el umbral 33

4.1. Diagrama de bloques del pipeline del firmware: etapa set-up. 38
4.2. Diagrama de bloques del pipeline del firmware: etapa creación de tareas virtuales. 39
4.3. Pipeline de inferencia embebido con TFLite Micro: captura, ejecución y transmisión de resultados optimizados. 41
4.4. Ejemplo de captura de imagen 320x240 realizada por la cámara embebida. Se observa la presencia de automóviles como objetivo del sistema. 47

Índice de figuras

5.1. Mapa general del flujo de datos. Se muestran los vínculos inalámbricos entre el nodo embebido (Dispositivo Parkwise), el Gateway y el Servidor Central. 49
5.2. Arquitectura de Bluetooth LE (Stack de BLE). Fuente [6] 52
5.3. Estructura de datos en GATT fuente [37] 52
5.4. Interacción entre Parkwise y Gateway mediante GATT sobre BLE. . . 54
5.5. Ejemplo de paquete serializado de una bounding box 55
5.6. Dispositivos Arduino Nano 33 BLE, placa FIng y MikroE LTE IOT 2 Click 56
5.7. Esquema del proceso de fragmentación de imágenes para transmisión vía BLE. La imagen capturada se divide en fragmentos secuenciales, cada uno precedido por un encabezado de 4 bytes que informa su posición y permite la reconstrucción del conjunto. 59
5.8. Interacción directa entre Parkwise y el Servidor Central mediante GATT sobre BLE. El servidor GATT (Parkwise) expone características que permiten enviar detecciones y recibir comandos. El Servidor Central actúa como cliente GATT y se suscribe a las detecciones mientras que puede enviar instrucciones como el hard reset. 62
5.9. Estructura general del flujo de datos entre Parkwise, Gateway y el Servidor Central. La detección BLE es almacenada temporalmente en un búfer de recepción y luego publicada al broker Mosquitto mediante MQTT. El servidor se suscribe a los tópicos correspondientes para recibir los datos. 64

6.1. Vista general de la interfaz web en modo live-inferencing. Se visualizan los controles de detecciones, botones de configuración y un mapa con nodos activos. 70
6.2. Interfaz de configuración de un nodo Parkwise. A la izquierda, imagen capturada con selección de puntos clave. A la derecha, vista de mapa con asignación geográfica de los mismos puntos. 71
6.3. Vista inicial en modo régimen. Se muestra el mapa de la ciudad con los Dispositivos Parkwise desplegados y el panel lateral con controles de operación. 72
6.4. Detalle de un nodo seleccionado. Se visualizan los puntos de homografía y la vereda proyectada en verde sobre el mapa. 73
6.5. Visualización completa de detecciones proyectadas. Cada evento se representa como un vehículo posicionado ortogonalmente a la vereda más cercana. 73
6.6. 75
6.7. Esquema ER simplificado de la base de datos `parkwise.db` 79
6.8. Flujo general de datos y responsabilidades del sistema 80
6.9. Diagrama de secuencia de sistema (DSS) para los principales casos de uso del backend de Parkwise 83
6.10. Diagrama de clases del módulo `wireless_comm.py`. Se observa la separación de responsabilidades entre los manejadores de BLE y de NB-IoT, junto con sus principales atributos y métodos. 86
6.11. Flujo de escaneo de nodos Parkwise y solicitud de imagen inicial por BLE. 87
6.12. Flujo de comunicación en modo régimen: detecciones BLE y NB-IoT procesadas en hilos concurrentes. 88
6.13. Flujo simplificado de envío de comandos de hard reset a nodos Parkwise vía BLE. 88

6.14. Diagrama de módulos del sistema de software de Parkwise. Cada componente cumple una función específica dentro del flujo general: desde el servidor web y los hilos de comunicación hasta la base de datos embebida y la interfaz gráfica. 90

7.1. Visualización bidimensional del espacio de embeddings extraídos de las imágenes de test. 97
7.2. Detecciones correctas efectuadas por el modelo entrenado en presición INT8 97
7.3. Detecciones incorrectas efectuadas por el modelo entrenado en presición INT8 97
7.4. Captura de pantalla del Jupyter notebook mostrando imágenes recibidas en tiempo real desde el dispositivo vía BLE. 99
7.5. Visualización de la tabla `Detection` y `Devices` en la base de datos `parkwise.db` usando DBCode. 106
7.6. Visualización en tiempo real de detecciones proyectadas mediante homografía en el frontend web. Las calles presentan un grosor menor debido al zoom aplicado desde el mapa, sin embargo, la vista satelital coincide con el ancho de las veredas en verde. 107
7.7. Captura de la terminal al ejecutar la recepción de imagen corriendo el script operario.py. Muestra como carga la barra de porcentaje de recepción de la imagen. 107
7.8. Pantalla de configuración web del Dispositivo Parkwise. A la izquierda se muestran los puntos seleccionados sobre la imagen capturada; a la derecha, los puntos correspondientes en el mapa georreferenciado. . . . 109
7.9. Image View en modo instalador: despliega la imagen recibida del Operario, con los puntos de referencia en la imagen, que se corresponderán en el mapa. 110
7.10. Vista del mapa en modo Instalador, con la correspondencia entre puntos de la imagen y del entorno. En el recuadro azul (abajo a la derecha) se amplía la zona de interés. 110
7.11. Interfaz de usuario: Mapa que ven los usuarios. 110
7.12. Visualización de múltiples Dispositivos Parkwise activos sobre el mapa. 111
7.13. Detecciones proyectadas correctamente desde uno de los Dispositivos Parkwise configurados. 111

D.1. Distribución de los datos de entrenamiento, validación y prueba. . . . 123
D.2. Comparación entre los valores reales y las predicciones del modelo entrenado. 124

Esta es la última página.
Compilado el martes 2 septiembre, 2025.
http://iie.fing.edu.uy/

Parkwise: Sistema Inalámbrico para el Mapeo de Vehículos Estacionados utilizando TinyML y Computer Vision en el Borde

Michael Dickinson
Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay
michael.dickinson@fing.edu.uy

Andrés Guido
Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay
andres.guido@fing.edu.uy

Santiago Pérez
Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay
santiago.perez@fing.edu.uy

***Abstract*—Esta tesis presenta el diseño, desarrollo e implementación de ParkWise, un sistema que permite la detección y mapeo de vehículos en entornos urbanos empleando inteligencia artificial y comunicación inalámbrica distribuida. El objetivo del proyecto es construir una plataforma para el monitoreo de la ocupación vehicular en la vía pública. Se adopta un enfoque descentralizado donde la detección de objetos se realiza directamente en el borde, utilizando microcontroladores de bajo consumo que ejecutan modelos de inferencia embebidos mediante técnicas de TinyML. Cada dispositivo, compuesto por un microcontrolador XIAO ESP32-S3 y una cámara, captura imágenes de forma periódica. Dicho proceso, junto a la comunicación inalámbrica, es gestionado por tareas concurrentes sobre el sistema operativo en tiempo real (RTOS) FreeRTOS. Posteriormente, las imágenes son procesadas localmente con un modelo de visión por computadora (FOMO), optimizado mediante TensorFlow Lite para microcontroladores, que alcanza un 95% de exactitud en la detección. Las detecciones se transmiten de forma inalámbrica a un Gateway, que las reenvía vía MQTT IoT a un Servidor Central para su procesamiento y visualización en una interfaz web interactiva. El sistema incluye un procedimiento de calibración visual que, a través de una matriz de homografía, permite proyectar las detecciones sobre un plano georreferenciado. El resultado es un sistema punta a punta, adaptable y de bajo costo que demuestra la viabilidad de una implementación a gran escala para el monitoreo de estacionamientos en tiempo real, ofreciendo una nueva herramienta para la gestión urbana inteligente.**

***Index Terms*—Edge AI, TinyML, TensorFlow Lite Micro, Microcontroller, ESP32-S3, BLE, MQTT, Homografía**

I. Introducción

En la mayoría de los centros urbanos, encontrar un espacio de estacionamiento disponible se ha convertido en un desafío diario que no solo consume tiempo, sino que también incrementa la congestión vehicular y el estrés de los conductores [10]. La ciudad de Montevideo, Uruguay, no es una excepción a esta problemática, impulsada por un crecimiento sostenido del parque automotor en los últimos años, como se evidencia en la Figura 1.1 [11], [12]. Este fenómeno, conocido como *cruising*, aumenta los tiempos de traslado y contribuye directamente a la saturación del tránsito, afectando a todos los usuarios de la vía pública [10].

Las soluciones tradicionales, como la expansión de infraestructura o las restricciones de circulación, han demostrado ser costosas, insostenibles a largo plazo o poco efectivas [13]. Sin embargo, enfoques innovadores como el tarifado dinámico, implementado en ciudades como San Francisco, han logrado reducir significativamente los tiempos de búsqueda de estacionamiento [14]. Inspirado en estos antecedentes, este trabajo presenta una solución de bajo costo orientada a optimizar la gestión del estacionamiento mediante la monitorización en tiempo real de la ocupación vehicular [15].

Para abordar este problema, se propone **Parkwise**, un sistema distribuido que utiliza inteligencia artificial en el borde (Edge AI) para detectar vehículos estacionados y mapear su ubicación en tiempo real [16]. La solución se basa en dispositivos de bajo consumo equipados con una cámara y un microcontrolador (Seeed XIAO ESP32-S3 Sense) [17], que ejecutan localmente un modelo de visión por computadora optimizado (FOMO) para identificar la presencia o ausencia de vehículos [18]. Las detecciones son transmitidas de forma inalámbrica a un servidor central, que las procesa y las presenta sobre un mapa digital interactivo, permitiendo a los usuarios visualizar las plazas disponibles [16].

La principal contribución de este trabajo es el diseño y la validación de un sistema completo y escalable que integra las siguientes tecnologías:

- **Detección en el borde con TinyML:** El procesamiento de imágenes se realiza localmente en microcontroladores con recursos limitados, reduciendo la latencia y el volumen de datos transmitidos [4], [19].
- **Arquitectura distribuida y de bajo costo:** El sistema utiliza un Gateway como intermediario para que múltiples dispositivos compartan una única conexión a la red de internet (IoT), optimizando los costos de conectividad e implementación [6], [20].
- **Calibración georreferenciada:** Se implementa un método de calibración basado en homografía que permite proyectar con precisión las detecciones de la cámara sobre un mapa, facilitando una visualización intuitiva y útil para el usuario final [8].

Este enfoque no solo ofrece una solución técnica viable para la gestión inteligente del estacionamiento, sino que también genera una valiosa fuente de datos para el análisis de la movilidad urbana [9].

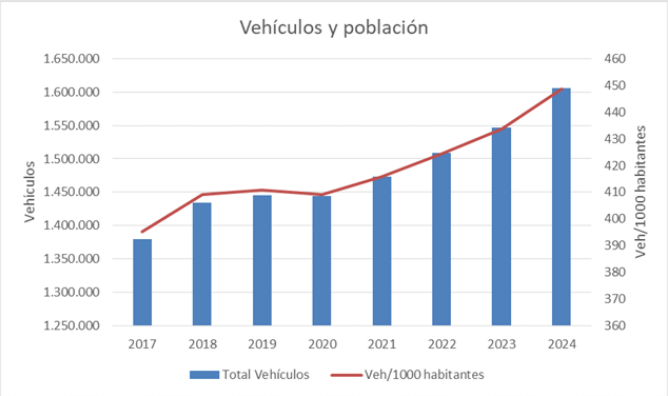


Fig. 1. Acumulado de vehículos en Uruguay y la proporción de vehículos cada mil habitantes. Fuente [11].

II. Arquitectura del Sistema

La solución propuesta se estructura en tres componentes principales interconectados: el **Dispositivo Parkwise**, el **Servidor Central** y el **Gateway**, cuya interacción permite generar y transmitir información en tiempo real sobre la disponibilidad de espacios de estacionamiento. El flujo de datos general del sistema se ilustra en la Figura 2.

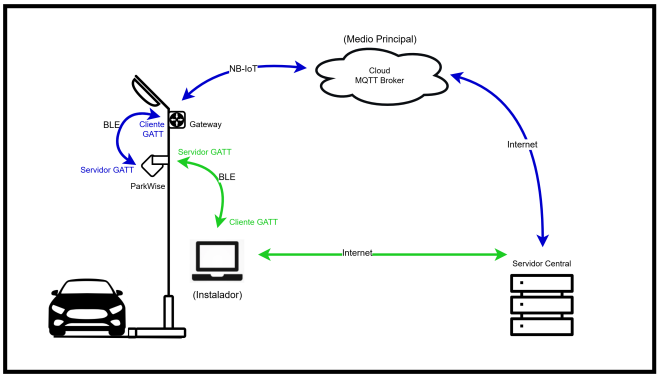


Fig. 2. Mapa general del flujo de datos. Se muestran los vínculos inalámbricos entre los componentes. La conexión del instalador (en verde) se utiliza únicamente durante la configuración inicial.

A. Dispositivo Parkwise

El Dispositivo Parkwise es un sistema embebido diseñado para operar en exteriores, encargado de la detección de vehículos. Está construido sobre el módulo **Seeed XIAO ESP32-S3 Sense**, que integra un microcontrolador, una cámara OV2640 y un módulo de comunicación inalámbrica BLE. Periódicamente, el dispositivo adquiere una imagen y ejecuta localmente un modelo de inteligencia artificial basado en la red neuronal **FOMO (Faster Objects, More Objects)**, optimizada para dispositivos con recursos limitados. Este modelo recibe la imagen como entrada y genera un conjunto de cajas delimitadoras (*bounding boxes*) que indican la ubicación de los vehículos presentes. Finalmente, estas coordenadas se empaquetan y se transmiten vía BLE al Gateway.

B. Servidor Central

El Servidor Central es el componente de *software* que centraliza y procesa la información proveniente de todos los dispositivos desplegados. Recibe las detecciones a través de los Gateways mediante un *broker* MQTT y las almacena en una base de datos local [20]. Una de sus funciones clave es aplicar una transformación de **homografía**, calculada durante la instalación, para proyectar las coordenadas de las detecciones desde la perspectiva de la cámara a un sistema de referencia geográfico. Esta información se presenta en una interfaz web con un mapa interactivo que permite a los usuarios visualizar los espacios de estacionamiento disponibles. Además, el servidor gestiona el envío de comandos remotos, como reinicios, a los dispositivos.

C. Gateway

El Gateway actúa como un puente de comunicación entre los Dispositivos Parkwise y el Servidor Central, traduciendo protocolos para garantizar la interoperabilidad. Se comunica localmente con los dispositivos utilizando el protocolo **GATT sobre BLE** y se conecta remotamente con el Servidor Central empleando **MQTT sobre internet (IoT)**. Esta arquitectura permite que múltiples Dispositivos Parkwise compartan un único punto de acceso a la red, lo que reduce significativamente los costos de conectividad y simplifica la implementación del sistema en entornos urbanos con infraestructura limitada. El Gateway se suscribe a las características GATT de los dispositivos para recibir las detecciones y las publica en un tópico MQTT específico para que el Servidor Central las consuma.

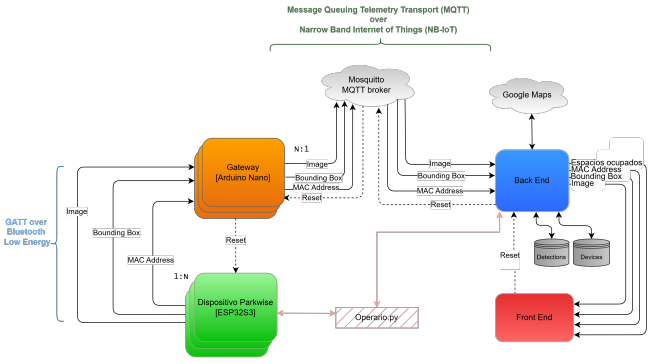


Fig. 3. Diagrama de protocolos, medios y flujo de información del sistema completo.

III. Metodología

La implementación del sistema Parkwise se basa en una cuidadosa selección de hardware y software, junto con el

desarrollo de un modelo de aprendizaje automático optimizado para operar en dispositivos con recursos limitados. A continuación, se detallan los componentes y procesos clave de esta metodología.

A. Selección de Hardware y Framework

La elección del hardware fue un factor determinante, imponiendo restricciones directas sobre el software [21]. El dispositivo debe capturar imágenes, ejecutar modelos de detección de objetos y comunicarse vía BLE [7]. Se seleccionó el módulo **Seeed XIAO ESP32-S3 Sense** por su equilibrio entre costo, capacidad de procesamiento y comunidad activa [17]. Este módulo integra un procesador dual-core Xtensa LX7, 512KB de SRAM, 8MB de PSRAM, y una cámara OV2640, cumpliendo con todos los requisitos funcionales del proyecto.

Para el desarrollo del firmware, se utilizó el framework **ESP-IDF**, el entorno oficial de Espressif [22]. Esto permitió un control profundo sobre los periféricos y la gestión de memoria, a la vez que mantuvo compatibilidad con el ecosistema de Arduino, ampliamente utilizado en proyectos de TinyML [23]. La arquitectura del firmware se construyó sobre el sistema operativo en tiempo real **FreeRTOS**, que facilita la gestión de tareas concurrentes que son la captura de imágenes, la ejecución de inferencias y la comunicación inalámbrica.

1) Preparación de Datasets: Para entrenar un modelo robusto y adaptado al contexto local, se desarrollaron dos conjuntos de datos propios que fueron complementados con un dataset público de exploración. La creación de datos locales fue fundamental para robustecer el modelo, asegurando que pudiera generalizar correctamente en las condiciones urbanas de Montevideo.

Los conjuntos de datos utilizados para el entrenamiento y la evaluación final fueron:

- **Tomas Diago Dataset**: Un conjunto de 312 imágenes capturadas manualmente por el equipo desde una vivienda en Montevideo[cite: 643, 647]. Su objetivo es reflejar fielmente las condiciones de iluminación, el entorno urbano y los modelos de vehículos que circulan localmente[cite: 655, 661, 663].
- **Drone Dataset**: Compuesto por 85 imágenes tomadas con un dron en una zona de alta circulación de Montevideo (inmediaciones del Shopping Punta Carretas)[cite: 666, 667, 673]. Este dataset aporta una perspectiva cenital, similar a la que tendría el dispositivo instalado en una luminaria, lo cual es clave para el caso de uso del sistema[cite: 700, 702].
- **Edge Impulse Toy Dataset**: Conjunto de datos público que fue utilizado en las fases iniciales para validar la arquitectura del modelo FOMO [cite: 521, 588] y que también formó parte de la mezcla final de datos para el entrenamiento.

En total, el conjunto de datos combinado para el entrenamiento y la evaluación del modelo consistió en **557 imágenes**, que contienen **2981 vehículos** etiquetados manualmente. Durante la investigación preliminar también se evaluó el **Car Parking Lot Dataset (CARPK)**, pero finalmente no fue empleado en el desarrollo del modelo definitivo.

El etiquetado de los datasets propios se realizó con **Edge Impulse Studio**, utilizando su funcionalidad de etiquetado asistido por IA con un modelo YOLOv5 pre-entrenado para agilizar el proceso[cite: 624, 626].

2) Arquitectura del Modelo y Entrenamiento: Se optó por la arquitectura **FOMO (Faster Objects, More Objects)**, desarrollada por Edge Impulse, debido a su alta eficiencia en dispositivos con recursos limitados [18]. FOMO es hasta 30 veces más rápido que arquitecturas como MobileNet-SSD y puede operar con menos de 200 KB de RAM [18]. A diferencia de los modelos que generan directamente cajas delimitadoras, FOMO produce un mapa de calor (*heatmap*) que indica los centroides de los objetos, un enfoque ideal para aplicaciones donde la ubicación es prioritaria sobre el tamaño exacto del objeto [18]. El entrenamiento se realizó en la plataforma de Edge Impulse, que abstrae la complejidad del proceso y genera una librería C++ optimizada para microcontroladores [19].

3) Cuantización y Despliegue: Para reducir el tamaño del modelo y permitir su ejecución en el ESP32-S3, se aplicó un proceso de **cuantización post-entrenamiento a precisión INT8** [27]. A pesar de la reducción en la precisión numérica, el modelo mantuvo un rendimiento robusto, con una mínima degradación en las métricas de evaluación. El modelo final, junto con el intérprete de **TensorFlow Lite for Microcontrollers (TFLM)**, se integró directamente en el firmware como un arreglo de bytes en C, eliminando la necesidad de un sistema de archivos y optimizando el uso de memoria.

IV. Implementación

La implementación de Parkwise abarcó dos dominios principales: el desarrollo de un firmware robusto para el dispositivo embebido y la creación de una aplicación de software de alto nivel que actúa como servidor central y como interfaz de usuario. Ambos componentes fueron diseñados de manera modular para garantizar la escalabilidad y el mantenimiento del sistema.

A. Firmware del Dispositivo

El firmware del Dispositivo Parkwise fue desarrollado en C++ sobre el framework **ESP-IDF** y el sistema operativo en tiempo real **FreeRTOS** [22]. Esta elección permitió un control detallado del hardware y una gestión eficiente de procesos concurrentes, clave para una operación fluida y no bloqueante [21].

La arquitectura del firmware se organiza en torno a tres tareas virtuales principales que se ejecutan de forma concurrente:

- **Vision Task:** Es el núcleo operativo del dispositivo. De forma cíclica, esta tarea captura una imagen de la cámara OV2640, realiza la conversión de formato, ejecuta el modelo de inferencia FOMO, serializa las cajas delimitadoras (*bounding boxes*) resultantes a formato JSON y las transmite vía BLE si existen clientes suscritos [16].

- **NimBLE Host Task:** Gestiona el stack de comunicación Bluetooth Low Energy (BLE), incluyendo la publicidad del dispositivo, el manejo de conexiones y la gestión de eventos de los perfiles GAP y GATT [29], [30]. Opera en paralelo a la tarea de visión, asegurando que la comunicación inalámbrica sea siempre responsiva [31].
- **Reset Timer Task:** Actúa como un mecanismo de *watchdog* personalizado, reiniciando periódicamente el ciclo de inferencia para garantizar la recuperación automática del sistema ante bloqueos o fallos silenciosos.

El flujo de inferencia se optimizó para recursos limitados, utilizando el intérprete de **TensorFlow Lite for Microcontrollers (TFLM)** [5]. Se asignó una región de memoria estática en PSRAM (*Tensor Arena*) y se utilizó un *Op Resolver* que incluye únicamente las cuatro operaciones matemáticas necesarias para el modelo (Add, Conv2D, DepthwiseConv2D y Pad), minimizando así la huella de memoria del binario [28].

B. Software de Alto Nivel

El software de alto nivel funciona como Servidor Central y como interfaz de usuario. Fue desarrollado en **Python** utilizando un conjunto de tecnologías ligeras y portables [32], [33].

El **backend**, construido con el framework **Flask**, se encarga de servir la interfaz web, exponer una API RESTful para la gestión de datos y coordinar los procesos de comunicación. Se utilizan hilos de ejecución concurrentes para escuchar de forma continua las detecciones provenientes tanto de la vía BLE como internet (a través de un broker MQTT), asegurando una operación no bloqueante [20], [34], [35].

El **frontend** es una aplicación web de una sola página (SPA) implementada con **HTML, JavaScript y CSS**, utilizando **Bootstrap** para los estilos y la **API de Google Maps** para la visualización geoespacial. La interfaz opera en dos modos principales:

- **Modo de Instalación (Setup):** Un script de operario (‘operario.py‘) se conecta por BLE a los nuevos dispositivos para solicitar una imagen de referencia. Posteriormente, desde la interfaz web, el operario utiliza esta imagen y el mapa para seleccionar cuatro pares de puntos de correspondencia, a partir de los cuales el sistema calcula y almacena la **matriz de homografía** 3x3 del dispositivo.
- **Modo de Régimen (Live-Inferencing):** La interfaz muestra en tiempo real las detecciones de vehículos recibidas, proyectándolas sobre la vereda correspondiente en el mapa gracias a la homografía previamente calculada. El usuario puede recargar las detecciones, ocultarlas o enviar comandos de reinicio a los dispositivos.

Toda la información de configuración (dispositivos, matrices de homografía, etc.) y los registros históricos de detecciones se almacenan en una base de datos local **SQLite**. Esta arquitectura modular permite que el sistema opere de forma autónoma en una red local, con potencial para ser migrado a un entorno distribuido en la nube con mínimos ajustes [16].

V. Experimentos y Resultados

Para validar el correcto funcionamiento del sistema Parkwise, se realizaron pruebas exhaustivas en cada uno de sus componentes: el modelo de inferencia, el dispositivo embebido, los canales de comunicación y la integración completa del sistema.

A. Rendimiento del Modelo de Inferencia

El modelo FOMO, basado en MobileNetV2, fue entrenado con un conjunto de datos combinado de 557 imágenes (conteniendo 2981 vehículos) de tres fuentes distintas para asegurar diversidad de escenarios. Se utilizó una partición del 80% para entrenamiento y 20% para pruebas [36].

1) Modelo en Precisión FLOAT32: La versión original del modelo, sin cuantización, alcanzó un alto rendimiento, demostrando su capacidad para identificar vehículos con gran precisión. Las métricas clave fueron:

- **Accuracy:** 97.12%
- **Precision (no-background):** 0.98
- **Recall (no-background):** 0.94
- **F1 Score (no-background):** 0.96

2) Modelo Cuantizado a Precisión INT8: Para optimizar el modelo para su despliegue en el microcontrolador ESP32-S3, se aplicó una cuantización post-entrenamiento a precisión INT8. A pesar de la reducción en la precisión numérica, el rendimiento se mantuvo robusto, validando su viabilidad para aplicaciones TinyML [27].

- **Accuracy:** 95.19%
- **Precision (no-background):** 0.98
- **Recall (no-background):** 0.93
- **F1 Score (no-background):** 0.95

La leve disminución en las métricas (Tabla I) fue marginal en comparación con los beneficios en tamaño y velocidad de ejecución en el hardware embebido.

TABLE I
Comparación de métricas entre modelo FLOAT32 e INT8.

Métrica	Modelo FLOAT32	Modelo INT8
Accuracy	97.12%	95.19%
Precision	0.98	0.98
Recall	0.94	0.93
F1 Score	0.96	0.95

B. Validación del Dispositivo Parkwise

Las pruebas sobre el dispositivo confirmaron la correcta implementación del firmware y su estabilidad operativa.

1) Captura de Imágenes y Ejecución de Inferencia: Se verificó la captura periódica de imágenes y la ejecución local del modelo FOMO. Mediante un cliente BLE, se recibieron en tiempo real las imágenes junto con las cajas delimitadoras (*bounding boxes*) superpuestas, validando visualmente que el pipeline completo —desde la captura hasta la detección— funcionaba correctamente [32].

2) Latencia y Uso de Recursos: El SDK de Edge Impulse permitió medir los tiempos de ejecución directamente en el dispositivo. El tiempo total del ciclo, incluyendo preprocesamiento (DSP) e inferencia, fue consistentemente inferior a 1 segundo (ej: DSP 54 ms, inferencia 843 ms) [37]. Adicionalmente, se monitoreó el uso de memoria RAM y de la pila de tareas (stack), confirmando que se mantenía dentro de márgenes seguros durante la operación continua, sin causar desbordamientos ni reinicios inesperados.

C. Pruebas de Comunicación e Integración

Se validaron los canales de comunicación y el flujo de datos de extremo a extremo.

1) Comunicación BLE e Intrnet: La comunicación entre el Dispositivo Parkwise y el Gateway (simulado en una PC) vía BLE fue estable en un radio de hasta 10 metros, suficiente para el caso de uso previsto [7]. Se verificó el ciclo completo de conexión, suscripción y recepción de detecciones en formato JSON. Posteriormente, se confirmó que el Gateway reenviaba correctamente estos mensajes al broker MQTT, con una latencia extremo a extremo (BLE + IoT) inferior a 2 segundos y sin pérdida de paquetes [20].

2) Pruebas de Integración y Proceso de Instalación: Se validó el flujo completo del sistema, desde la instalación de un nuevo dispositivo hasta la visualización de sus detecciones. Se comprobó que el script del operario (‘operario.py‘) solicitaba y recibía correctamente la imagen de referencia. Posteriormente, desde la interfaz web, fue posible configurar la homografía y las veredas, y los datos se almacenaron correctamente en la base de datos SQLite. Finalmente, las detecciones recibidas en modo de régimen fueron proyectadas con éxito sobre el mapa, como se muestra en la Figura 4.

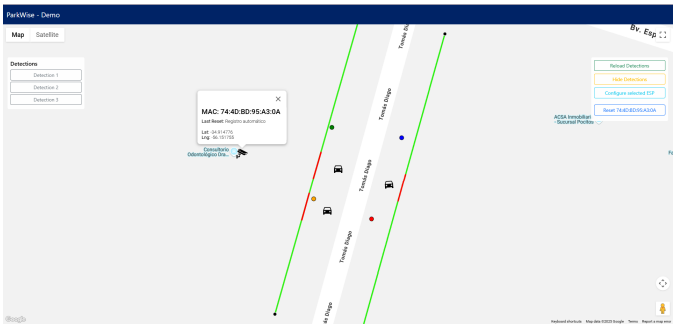


Fig. 4. Visualización en tiempo real de detecciones proyectadas mediante homografía en la interfaz web.

D. Escalabilidad y Tolerancia a Fallos

Para evaluar la escalabilidad, se configuraron dos Dispositivos Parkwise en ubicaciones diferentes, cada uno asociado a un Gateway independiente. Ambos nodos reportaron sus detecciones en paralelo y sin conflictos, demostrando que la arquitectura basada en MQTT escala horizontalmente [20]. También se simularon fallos como reinicios forzados y desconexiones, observando que el sistema respondía de forma resiliente, recuperando la operación sin intervención manual.

VI. Conclusión y Trabajo Futuro

El desarrollo del sistema Parkwise ha permitido demostrar la viabilidad técnica de una solución distribuida para la detección de vehículos en entornos urbanos, combinando dispositivos de bajo consumo, inteligencia artificial en el borde y tecnologías de comunicación inalámbrica. A lo largo de este trabajo, se diseñaron, implementaron y validaron con éxito todos los módulos del sistema, desde la captura y el procesamiento de imágenes en tiempo real hasta la transmisión, visualización y configuración remota de los nodos.

La elección del microcontrolador ESP32-S3 y el enfoque basado en TinyML resultaron ser acertados, permitiendo realizar inferencias directamente en el dispositivo sin depender de una conectividad constante. El modelo FOMO embebido demostró un rendimiento notable en términos de velocidad y precisión, mientras que la arquitectura de tareas concurrentes sobre FreeRTOS garantizó una operación estable y eficiente. La integración de BLE y IoT (a través de MQTT) proporcionó una solución de comunicación flexible y escalable, adaptada a las necesidades de alcance y consumo del sistema. Las pruebas de campo confirmaron la robustez del sistema, su capacidad para proyectar detecciones de forma georreferenciada mediante homografía y su resiliencia ante fallos, validando su potencial para un despliegue a escala urbana.

Como trabajo futuro, se proponen varias líneas de expansión. El sistema podría enriquecerse para detectar otros eventos de interés urbano, como estacionamientos indebidos, contenedores de basura desbordados, o incluso incidentes de tráfico. Asimismo, la plataforma podría integrar fuentes de datos externas, como calendarios de ferias vecinales o información sobre calles cerradas, para mejorar la precisión en la identificación de espacios disponibles. Finalmente, la información generada por Parkwise podría ofrecerse como un servicio a aplicaciones de navegación, guiando a los conductores no solo a su destino, sino al lugar de estacionamiento libre más cercano, optimizando así la movilidad urbana de manera integral.

Agradecimientos

Los autores desean expresar su agradecimiento a la Facultad de Ingeniería de la Universidad de la República por el apoyo brindado durante el desarrollo de este proyecto. En particular, por facilitar el acceso a sus instalaciones, así como a las plataformas y recursos tecnológicos que fueron esenciales para la realización de esta investigación. Se extiende también un reconocimiento a los docentes y compañeros por sus valiosas recomendaciones técnicas.

References

[1] Y. Zhou and H. Wang, ”Deep learning solutions for smart city challenges in urban development,” *Scientific Reports*, vol. 13, no. 1, pp. 1–12, 2023.

[2] B. Cai, R. Álvarez, M. Sit, F. Duarte, and C. Ratti, ”Deep Learning Based Video System for Accurate and Real-Time Parking Measurement,” *arXiv preprint arXiv:1902.07401*, 2019.

[3] G. Amato, F. Carrara, F. Falchi, C. Gennaro, C. Meghini, and C. Vairo, ”Deep learning for decentralized parking lot occupancy detection,” *Expert Systems with Applications*, vol. 72, pp. 327–334, 2017.

[4] P. Warden and D. Situnayake, *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media, Inc., 2019.

[5] TensorFlow, "TensorFlow Lite para Microcontroladores," 2025. [Online]. Available: https://www.tensorflow.org/lite/microcontrollers?hl=es-419

[6] Narrowband.com, "What is Narrowband IoT: a Comprehensive Guide," 2024. [Online]. Available: https://www.narrowband.com/what-is-narrowband-iot

[7] Bluetooth Special Interest Group, "The Bluetooth Low Energy Primer," Version 1.2.0, May 2022. [Online]. Available: https://www.bluetooth.com/wp-content/uploads/2022/05/the-bluetooth-le-primer-v1.2.0.pdf

[8] M.-R. Hsieh, Y.-L. Lin, and W. H. Hsu, "Drone-based object counting by spatially regularized regional proposal network," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 4145–4153.

[9] A. Kumar and A. Singh, "Parking Management System Based on Key Points Detection," *Applied Engineering Letters*, vol. 8, no. 1, pp. 15–20, 2023.

[10] G. Cookson, "INRIX 2019 Parking Pain Survey," INRIX Research, 2019. [Online]. Available: https://inrix.com/blog/parkingsurvey/

[11] Ministerio de Industria, Energía y Minería (Uruguay), "Parque Automotor," 2024. [Online]. Available: https://www.gub.uy/ministerio-industria-energia-mineria/datos-y-estadisticas/estadisticas/parque-automotor

[12] Instituto Nacional de Estadística (Uruguay), "Censo 2023: Crecen los hogares unipersonales," 2024. [Online]. Available: https://www.gub.uy/instituto-nacional-estadistica/comunicacion/noticias/censo-2023-crecen-hogares-unipersonales

[13] F. Fresard, "Efecto real de la restricción vehicular en Santiago de Chile," in *Ingeniería de Tránsito y Transporte. Actas del X Congreso Panamericano*, Santiago, Chile, 1998, pp. 299–308.

[14] San Francisco Municipal Transportation Agency (SFMTA), "San Francisco Adopts Demand-Responsive Pricing Program to Make Parking Easier," 2023. [Online]. Available: https://www.sfmta.com/blog/san-francisco-adopts-demand-responsive-pricing-program-make-parking-easier

[15] A. Matsuda and T. Matsui, "A System for Real-time On-street Parking Detection and Visualization on an Edge Device," in *2021 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, 2021, pp. 1–4.

[16] M. Dickinson, A. Guido, and S. Pérez, "Sistema inalámbrico distribuido para la detección y mapeo urbano de vehículos (Parkwise)," 2025. [Online]. Available: https://gitlab.fing.edu.uy/parkwiseguys/park-wise-pfc

[17] M. Rovai, "TinyML Made Easy: Object Detection with XIAO ESP32S3 Sense," Hackster.io, 2022. [Online]. Available: https://www.hackster.io/mjrobot/tinyml-made-easy-object-detection-with-xiao-esp32s3-sense-6be28d

[18] Edge Impulse, "FOMO: Object detection for constrained devices," 2025. [Online]. Available: https://docs.edgeimpulse.com/docs/edge-impulse-studio/learning-blocks/object-detection/fomo-object-detection-for-constrained-devices

[19] S. Hymel et al., "Edge Impulse: An MLOps Platform for Tiny Machine Learning," *arXiv preprint arXiv:2212.03332*, 2023.

[20] A. Banks, E. Briggs, K. Borgendale, and R. Gupta, "MQTT Version 5.0," OASIS Standard, Mar. 2019. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html

[21] T. Novosel and L. Jelenković, "Frameworks for embedded system development," in *2011 Proceedings of the 34th International Convention MIPRO*, 2011, pp. 825–828.

[22] Espressif Systems, "ESP-IDF Programming Guide," 2025. [Online]. Available: https://docs.espressif.com/projects/esp-idf/

[23] Espressif Systems, "Arduino core for the ESP32," 2025. [Online]. Available: https://github.com/espressif/arduino-esp32

[24] Edge Impulse, "Labeling queue," 2023. [Online]. Available: https://docs.edgeimpulse.com/docs/edge-impulse-studio/data-acquisition/labeling-queue

[25] G. Jocher et al., "YOLOv5 - You Only Look Once Version 5," 2020. [Online]. Available: https://github.com/ultralytics/yolov5

[26] TensorFlow Authors, "Post-training quantization," 2024.[Online]. Available: https://www.tensorflow.org/model_optimization/guide/quantization/post_training

[27] TensorFlow Authors, "TensorFlow Lite Micro: Ejemplo de Detección de Personas," 2020. [Online]. Available: https://github.com/tensorflow/tensorflow/tree/9837eceb39171aba9e28dc1f120f53271b6b1ef0/tensorflow/lite/experimental/micro/examples/person_detection

[28] Apache Software Foundation, "Apache Mynewt NimBLE Bluetooth Low Energy Stack," 2024. [Online]. Available: https://mynewt.apache.org/latest/network/ble/ble_intro/

[29] Bluetooth SIG, "Generic Access Profile (GAP) Specification," 2024. [Online]. Available: https://www.bluetooth.com/specifications/assigned-numbers/generic-access-profile/

[30] Bluetooth SIG, "Generic Attribute Profile (GATT) Specification," 2024. [Online]. Available: https://www.bluetooth.com/specifications/gatt/

[31] Bleak contributors, "Bleak — Bluetooth Low Energy platform agnostic Client for Python," 2025. [Online]. Available: https://bleak.readthedocs.io

[32] Eclipse Foundation, "Eclipse Paho MQTT Python Client," 2025. [Online]. Available: https://www.eclipse.org/paho/index.php?page=clients/python/index.php

[33] Python Software Foundation, "asyncio — Asynchronous I/O," 2025. [Online]. Available: https://docs.python.org/3/library/asyncio.html

[34] F. Chollet et al., "Keras," 2015. [Online]. Available: https://keras.io

[35] A. G. Howard et al., "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv preprint arXiv:1704.04861*, 2017.

[36] Edge Impulse, "C++ Inferencing SDK," 2024. [Online]. Available: https://docs.edgeimpulse.com/reference/inferencing-sdks/cpp-inferencing-sdk

Parkwise: Sistema Inálambrico para el Mapeo de Vehículos Estacionados utilizando TinyML y Computer Vision en el Borde

Michael Dickinson, Andrés Guido, Santiago Pérez
[michael.dickinson, andres.guido, santiago.perez]@fing.edu.uy
Facultad de Ingeniería, Montevideo, UY.

gitlab.fing.edu.uy/parkwiseguys/park-wise-pfc

Introducción

Esta tesis presenta Parkwise, un sistema que detecta y mapea vehículos estacionados en tiempo real. Los dispositivos, basados en microcontroladores XIAO ESP32-S3 Sense con cámara, capturan imágenes y ejecutan una red neuronal FOMO directamente en el dispositivo.
Las detecciones se envían de manera inalámbrica a un Gateway, que concentra la información y la reenvía a un Servidor Central, donde se visualizan en un mapa digital interactivo con las plazas ocupadas. El sistema ofrece así una solución escalable y adaptable para la gestión inteligente del estacionamiento urbano, contribuyendo a reducir la congestión asociada a la búsqueda de espacios de estacionamiento (*cruising*).

Arquitectura del Sistema

La solución propuesta se estructura en tres componentes principales interconectados: el Dispositivo Parkwise, el Servidor Central y el Gateway, cuya interacción permite generar y transmitir información en tiempo real sobre la disponibilidad de espacios de estacionamiento. El flujo de datos general del sistema se ilustra en la Fig. 1.

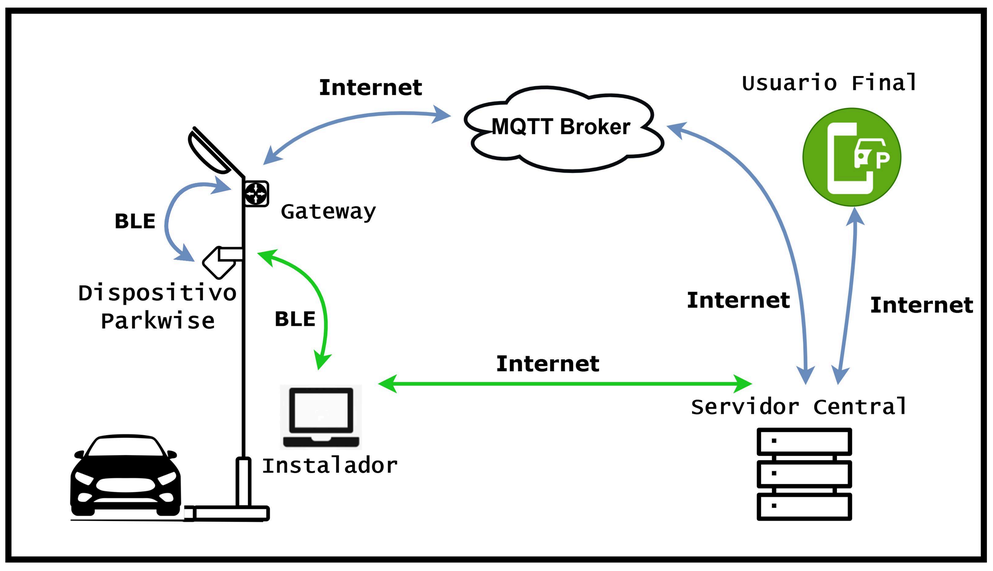


Fig. 1. Diagrama de la arquitectura física y los vínculos inalámbricos entre los componentes

Dispositivo Parkwise: Es un sistema embebido diseñado para detectar vehículos estacionados. Captura imágenes periódicamente y ejecuta localmente un modelo de Inteligencia Artificial basado en la red neuronal FOMO, que detecta los vehículos y envía las detecciones de manera inalámbrica por Bluetooth BLE.

Gateway: Actúa como puente entre los Dispositivos Parkwise y el Servidor Central. Se comunica con los dispositivos mediante BLE y disponibiliza los datos por internet, centralizando las detecciones cercanasy facilitando la interoperabilidad entre múltiples dispositivos.

Sesvidor Central: Recibe las detecciones a través del Gateway, las procesa y almacena en una base de datos. Aplica transformaciones geográficas para proyectar las detecciones en un mapa interactivo que permite visualizar los espacios de estacionamiento disponibles. Además, gestiona el envío de comandos remotos a los dispositivos, como reinicios.

Metodología

El Dispositivo Parkwise corresponde al Seeed XIAO ESP32-S3 con módulo Sense, que integra una cámara. Este dispositivo ofrece un equilibrio entre costo, capacidad de procesamiento y soporte de la comunidad del framework ESP-IDF.

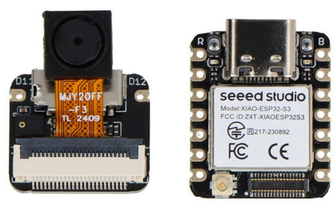


Fig. 2. Dispositivo Parkwise

Datasets utilizados: Se utilizaron tres datasets para el etrenamiento del modelo final.

- **Tomas Diago Dataset:** un dataset propio de 312 imágenes con sus etiquetas desde un apartamento en la calle Tomás Diago, Montevideo.
- **Drone Dataset:** conjunto de datos propio de 85 imágenes etiquetadas con un drone en el estacionamiento de Punta Carretas Shopping.
- **Edge Impulse Toy Dataset:** un dataset público que se encuentra en unun proyecto de detección de vehículos en la plataforma de Edge Impulse.

El conjunto de entrenamiento y validación totalizó 557 imágenes con una partición estratificada del 80% para entrenamiento y 20% para test.
Los mejores resultados obtrenidos fueron con una red neuronal FOMO que toma imágenes de entrada de 320x240. Se realizó una cuantización a entero de 8 bits para el modelo final. Los resultados se muestran en la Fig. 3.

Métrica	Modelo Float32	Modelo Int8
Accuracy	0.97	0.95
Precision	0.98	0.98
Recall	0.94	0.93
F1 Score	0.96	0.95

Fig. 3. Métricas de los modelos en punto flotante y cuantizado

Validaciones realizadas:

- **Captura de imágenes e inferencia de vehículos.** Recibidas por BLE en otro dispositivo.
- **Pruebas de comunicación e integración entre los módulos:** validada desde la comunicación Dispositivo Parkwise con Gateway y del Gateway al Servidor Central, de manera concatenada, para recibir las detecciones en el servidor.
- **Escalabilidad:** Se probaron dos dispositivos Parkwise en simultaneo enviando datos al servidor central desde sus Gateway, validándose que ambos desplegaban sus detecciones en el servidor central.

Fig. 4. Sistema funcionando con dos dispositivos en simultáneo.

Conclusiones:

- Se demostró la viabilidad técnica de un sistema distribuido capaz de detectar vehículos estacionados realizando inferencias en el borde.
- Se desarrolló y embebió un modelo FOMO con exactitud del 95% y predicciones en tiempo real.
- Se desplegaron las detecciones sobre un mapa digital que permite escalar la solución al sensado de múltiples puntos en simultáneo.

Referencias

[1] Zhou, Y. & Wang, H. (2023). Deep learning solutions for smart city challenges in urban development. Scientific Reports, 13(1), 1–12.
[2] Cai, B., Álvarez, R., Sit, M., Duarte, F. & Ratti, C. (2019). Deep Learning Based Video System for Accurate and Real-Time Parking Measurement. arXiv:1902.07401.
[3] Amato, G., Carrara, F., Falchi, F., Gennaro, C., Meghini, C. & Vairo, C. (2017). Deep learning for decentralized parking lot occupancy detection. Expert Systems with Applications, 72, 327–334.
[4] Warden, P. & Situnayake, D. (2019). TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers.
[5] TensorFlow (2025). TensorFlow Lite para Microcontroladores. https://www.tensorflow.org/lite/microcontrollers
[6] Narrowband.com (2024). What is Narrowband IoT: a Comprehensive Guide. https://www.narrowband.com/what-is-narrowband-iot
[7] Bluetooth SIG (2022). The Bluetooth Low Energy Primer, v1.2.0.
[8] Hsieh, M.-R., Lin, Y.-L. & Hsu, W. H. (2017). Drone-based object counting by spatially regularized regional proposal network. Proc. ICCV, 4145–4153.
[9] Kumar, A. & Singh, A. (2023). Parking Management System Based on Key Points Detection. Applied Engineering Letters, 8(1), 15–20.
[10] Cookson, G. (2019). INRIX 2019 Parking Pain Survey. INRIX Research. https://inrix.com/blog/parkingsurvey/
[11] Ministerio de Industria, Energía y Minería (Uruguay) (2024). Parque Automotor. https://www.gub.uy/ministerio-industria-energia-mineria/datos-y-estadisticas/estadisticas/parque-automotor
[12] Instituto Nacional de Estadística (Uruguay) (2024). Censo 2023: Crecen los hogares unipersonales. https://www.gub.uy/instituto-nacional-estadistica
[13] Fresard, F. (1998). Efecto real de la restricción vehicular en Santiago de Chile. Actas del X Congreso Panamericano de Ingeniería de Tránsito y Transporte, 299–308.
[14] SFMTA (2023). San Francisco Adopts Demand-Responsive Pricing Program to Make Parking Easier. https://www.sfmta.com/blog
[15] Matsuda, A. & Matsui, T. (2021). A System for Real-time On-street Parking Detection and Visualization on an Edge Device. Proc. IEEE PerCom Workshops, 1–4.
[17] Rovai, M. (2022). TinyML Made Easy: Object Detection with XIAO ESP32S3 Sense. Hackster.io. https://www.hackster.io/mjrobot/tinyml-made-easy-object-detection
[18] FOMO: https://docs.edgeimpulse.com/docs/edge-impulse-studio/learning-blocks/object-detection/fomo-object-detection
[19] Hymel, S. et al. (2023). Edge Impulse: An MLOps Platform for Tiny Machine Learning. arXiv:2212.03332.
[20] Banks, A., Briggs, E., Borgendale, K. & Gupta, R. (2019). MQTT Version 5.0. https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html
[21] Novosel, T. & Jelenković, L. (2011). Frameworks for embedded system development. Proc. MIPRO, 825–828.
[22] Espressif Systems (2025). ESP-IDF Programming Guide. https://docs.espressif.com/projects/esp-idf/
[23] Espressif Systems (2025). Arduino core for the ESP32. https://github.com/espressif/arduino-esp32
[24] Edge Impulse (2023). Labeling queue. https://docs.edgeimpulse.com/docs/edge-impulse-studio/data-acquisition/labeling-queue
[25] Jocher, G. et al. (2020). YOLOv5 – You Only Look Once Version 5. https://github.com/ultralytics/yolov5
[26] Apache Software Foundation (2024). Apache Mynewt NimBLE Bluetooth Low Energy Stack. https://mynewt.apache.org/latest/network/ble/
[27] Bluetooth SIG (2024). Generic Access Profile (GAP) Specification.
[28] Bluetooth SIG (2024). Generic Attribute Profile (GATT) Specification. https://www.bluetooth.com/specifications/gatt/
[29] Bleak contributors (2025). Bleak — Bluetooth Low Energy Client for Python. https://bleak.readthedocs.io
[30] Eclipse Foundation (2025). Eclipse Paho MQTT Python Client. https://www.eclipse.org/paho
[31] Python Software Foundation (2025). asyncio — Asynchronous I/O. https://docs.python.org/3/library/asyncio.html
[32] Chollet, F. et al. (2015). Keras. https://keras.io
[33] Howard, A. G. et al. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861.
[34] Edge Impulse (2024). C++ Inferencing SDK. https://docs.edgeimpulse.com/reference/inferencing-sdks/cpp