Documento Técnico Integral – Sistema de Seguridad IoT

1. Introducción

Este documento describe en detalle el diseño, la arquitectura, los requisitos y la operación de un sistema de Seguridad IoT basado en Raspberry Pi 4, ESP32 y cámaras, desplegado mediante contenedores Docker. Combina especificación de requisitos de software (SRS), documento de arquitectura (SAD) y guía operacional (DevOps).

2. Alcance

El sistema debe integrar sensores PIR, cámaras, relés de iluminación y otros sensores IoT para proveer monitoreo y reacción automática ante eventos de movimiento en distintas zonas. La solución se ejecuta localmente pero está pensada para ser migrable a nubes como AWS IoT Core sin rediseñar la arquitectura central.

3. Requisitos Funcionales

RF1. El sistema debe recibir eventos de sensores PIR a través de MQTT.

RF2. El sistema debe asociar cada evento PIR a una zona lógica configurada en la base de datos.

RF3. El sistema debe activar automáticamente los relés de iluminación pertenecientes a la zona donde se produjo el evento PIR.

RF4. El sistema debe activar las cámaras asociadas a la misma zona para capturar imágenes o vídeo del evento.

RF5. Las cámaras deben poder ser activadas también de forma manual mediante API REST.

RF6. Las imágenes capturadas deben ser enviadas al backend y almacenadas como evidencias en disco y registradas en la base de datos.

RF7. Un servicio de IA debe procesar las evidencias y agregar metadatos (por ejemplo detección de personas).

RF8. El sistema debe registrar eventos y telemetría en PostgreSQL en formato JSONB.

RF9. El dashboard web debe permitir visualizar dispositivos, zonas, eventos recientes y evidencias, además de consultar mediciones históricas.

RF10. Todos los componentes lógicos (broker MQTT, backend REST, worker MQTT, IA, frontend y base de datos) deben poder ejecutarse como contenedores Docker.

4. Requisitos No Funcionales

RNF1. Disponibilidad: el sistema debe operar 24/7 con mínima intervención manual.

RNF2. Rendimiento: la latencia entre un evento PIR y la activación de relés no debe superar los 300 ms bajo carga nominal.

RNF3. Seguridad: el broker MQTT debe requerir autenticación, y la API REST debe estar protegida mediante autenticación basada en tokens.

RNF4. Escalabilidad: debe ser posible agregar nuevas zonas y dispositivos sin cambios en la arquitectura central.

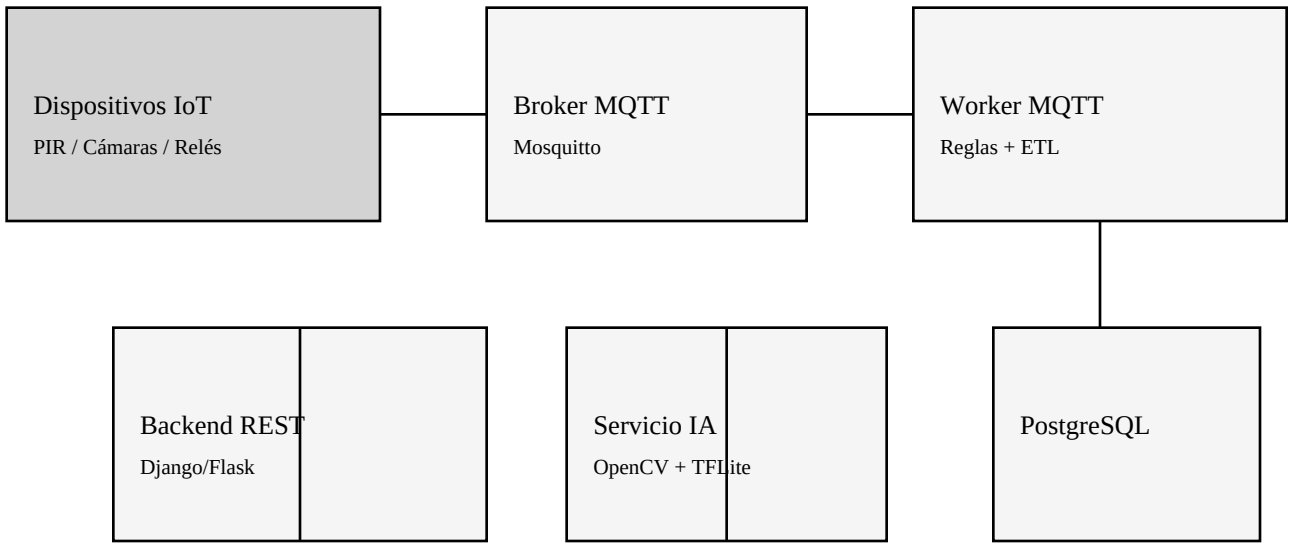
RNF5. Portabilidad: toda la solución debe poder levantarse mediante Docker Compose en una Raspberry Pi 4, y en un futuro en una VM o servidor en la nube.

RNF6. Observabilidad: se deben generar logs de los servicios principales, con rotación controlada y posibilidad de inspección remota.

RNF7. Mantenibilidad: el código debe estar organizado por servicios (microservicios ligeros) con responsabilidades claras.

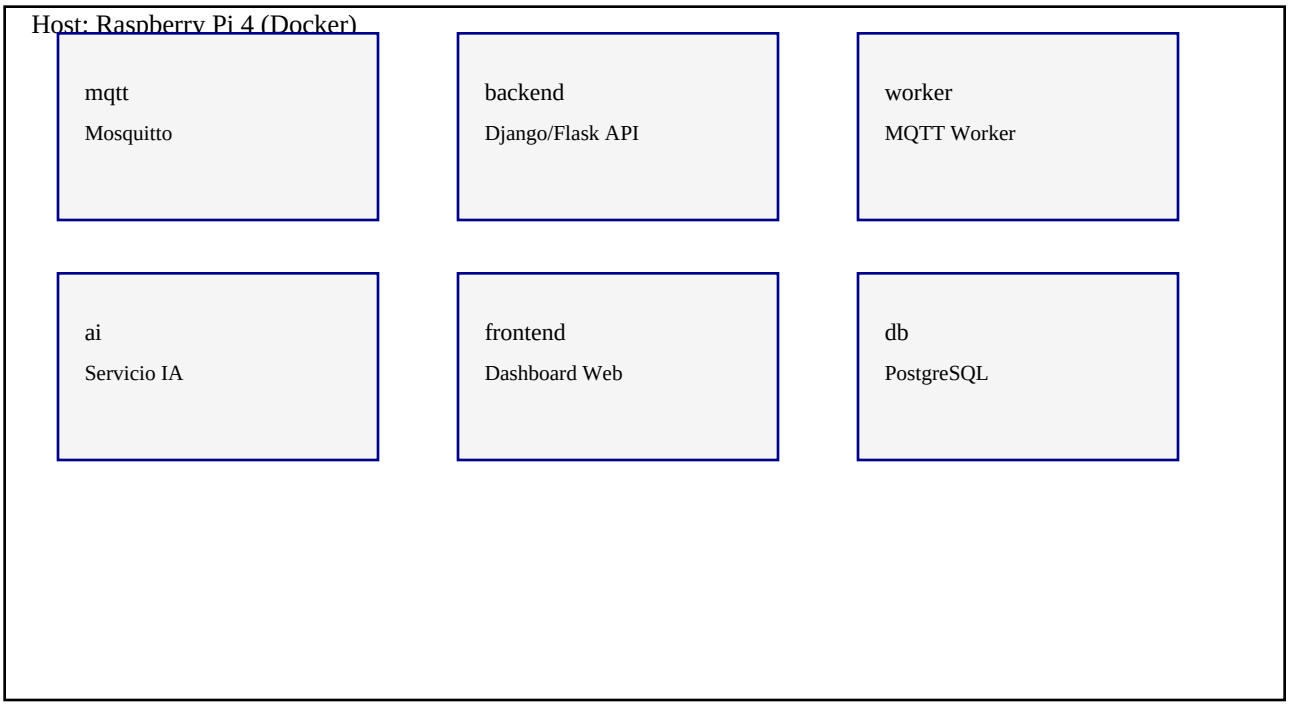
5. Arquitectura Lógica del Sistema

La arquitectura se compone de varios bloques lógicos: dispositivos IoT (PIR, cámaras, relés), un broker MQTT, un worker MQTT que actúa como motor de reglas y ETL, un backend REST (Django/Flask), un servicio de IA para reconocimiento de imágenes, un frontend web y la base de datos PostgreSQL.



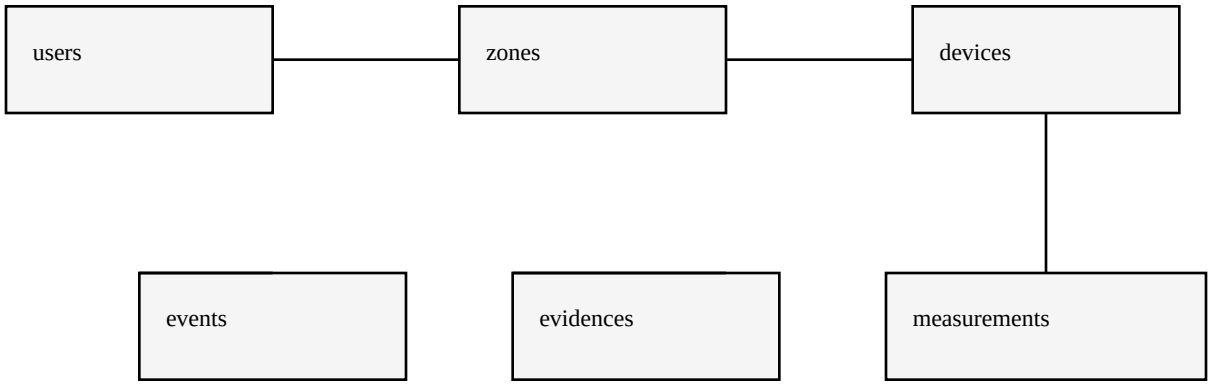
6. Arquitectura de Contenedores Docker

Cada componente lógico del sistema se despliega en un contenedor independiente. La Raspberry Pi 4 actúa como host Docker y orquesta los servicios mediante Docker Compose.



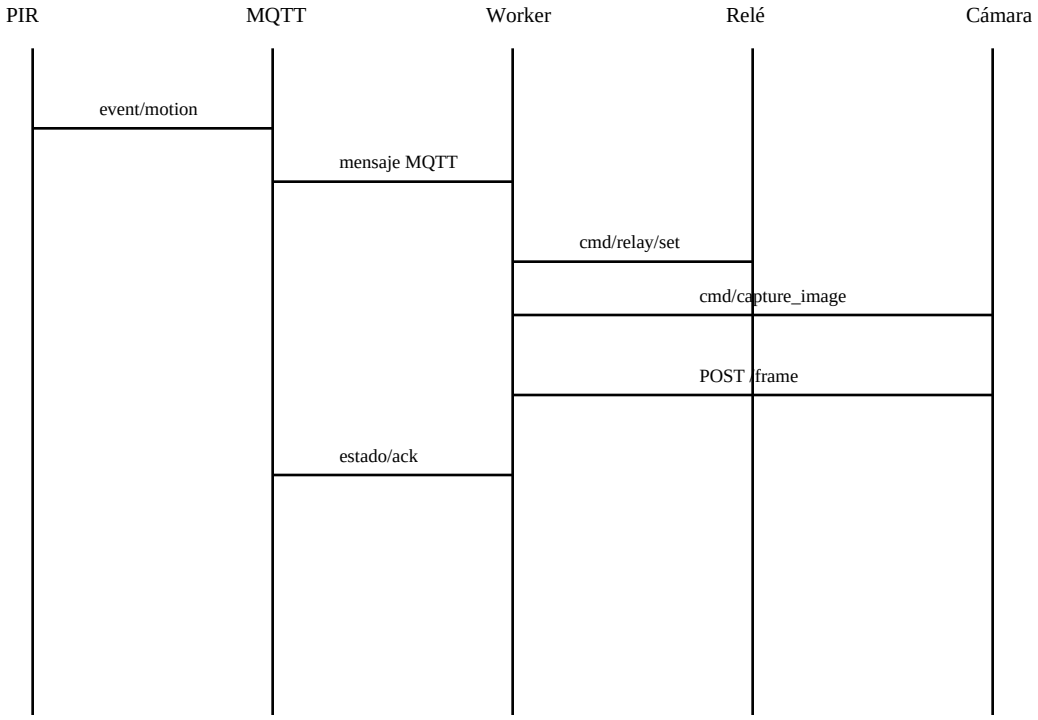
7. Diagrama Entidad–Relación (Simplificado)

El siguiente diagrama en alto nivel muestra las relaciones principales entre usuarios, zonas, dispositivos, eventos, evidencias y mediciones.



8. Diagrama de Secuencia: PIR → Luces y Cámaras

Este diagrama representa el flujo de mensajes cuando un sensor PIR detecta movimiento. Se muestran las interacciones básicas entre el PIR, el broker MQTT, el worker, los relés de iluminación y las cámaras.



9. Operación General y Casos de Uso

CASO DE USO 1: Detección automática.

1. Un PIR detecta movimiento en la Zona A y publica un mensaje MQTT. 2. El worker MQTT identifica la zona y registra un evento en la base de datos. 3. El worker busca todos los dispositivos tipo 'relay' asociados a la zona y envía comandos para encender las luces. 4. El worker también envía comandos a las cámaras de la zona para capturar imágenes o activar streaming. 5. Las cámaras envían las imágenes al backend, donde se almacenan como evidencias. 6. El servicio de IA analiza las evidencias y anota metadatos de detección. 7. El dashboard actualiza la vista de eventos y muestra notificaciones al usuario.

CASO DE USO 2: Visualización de telemetría.

1. Un ESP32 de telemetría publica periódicamente datos en tópicos 'devices/<id>/telemetry'. 2. El worker MQTT captura y almacena los datos en la tabla 'measurements'. 3. El usuario abre el dashboard y consulta gráficos históricos de temperatura, humedad u otros parámetros.

10.10 Detalles Adicionales de Implementación

Esta sección describe lineamientos de calidad de código, manejo de errores, política de reconexión MQTT, y mejores prácticas para la implementación en Raspberry Pi. Los servicios deben manejar reconexiones automáticas al broker y a la base de datos, así como registrar adecuadamente los fallos de comunicación. Los contenedores se gestionan mediante Docker Compose, permitiendo reinicios automatizados y despliegues consistentes. Es recomendable utilizar archivos .env para parametrizar credenciales y endpoints.

10.11 Detalles Adicionales de Implementación

Esta sección describe lineamientos de calidad de código, manejo de errores, política de reconexión MQTT, y mejores prácticas para la implementación en Raspberry Pi. Los servicios deben manejar reconexiones automáticas al broker y a la base de datos, así como registrar adecuadamente los fallos de comunicación. Los contenedores se gestionan mediante Docker Compose, permitiendo reinicios automatizados y despliegues consistentes. Es recomendable utilizar archivos .env para parametrizar credenciales y endpoints.

10.12 Detalles Adicionales de Implementación

Esta sección describe lineamientos de calidad de código, manejo de errores, política de reconexión MQTT, y mejores prácticas para la implementación en Raspberry Pi. Los servicios deben manejar reconexiones automáticas al broker y a la base de datos, así como registrar adecuadamente los fallos de comunicación. Los contenedores se gestionan mediante Docker Compose, permitiendo reinicios automatizados y despliegues consistentes. Es recomendable utilizar archivos .env para parametrizar credenciales y endpoints.

10.13 Detalles Adicionales de Implementación

Esta sección describe lineamientos de calidad de código, manejo de errores, política de reconexión MQTT, y mejores prácticas para la implementación en Raspberry Pi. Los servicios deben manejar reconexiones automáticas al broker y a la base de datos, así como registrar adecuadamente los fallos de comunicación. Los contenedores se gestionan mediante Docker Compose, permitiendo reinicios automatizados y despliegues consistentes. Es recomendable utilizar archivos .env para parametrizar credenciales y endpoints.

10.14 Detalles Adicionales de Implementación

Esta sección describe lineamientos de calidad de código, manejo de errores, política de reconexión MQTT, y mejores prácticas para la implementación en Raspberry Pi. Los servicios deben manejar reconexiones automáticas al broker y a la base de datos, así como registrar adecuadamente los fallos de comunicación. Los contenedores se gestionan mediante Docker Compose, permitiendo reinicios automatizados y despliegues consistentes. Es recomendable utilizar archivos .env para parametrizar credenciales y endpoints.

10.15 Detalles Adicionales de Implementación

Esta sección describe lineamientos de calidad de código, manejo de errores, política de reconexión MQTT, y mejores prácticas para la implementación en Raspberry Pi. Los servicios deben manejar reconexiones automáticas al broker y a la base de datos, así como registrar adecuadamente los fallos de comunicación. Los contenedores se gestionan mediante Docker Compose, permitiendo reinicios automatizados y despliegues consistentes. Es recomendable utilizar archivos .env para parametrizar credenciales y endpoints.

10.16 Detalles Adicionales de Implementación

Esta sección describe lineamientos de calidad de código, manejo de errores, política de reconexión MQTT, y mejores prácticas para la implementación en Raspberry Pi. Los servicios deben manejar reconexiones automáticas al broker y a la base de datos, así como registrar adecuadamente los fallos de comunicación. Los contenedores se gestionan mediante Docker Compose, permitiendo reinicios automatizados y despliegues consistentes. Es recomendable utilizar archivos .env para parametrizar credenciales y endpoints.

10.17 Detalles Adicionales de Implementación

Esta sección describe lineamientos de calidad de código, manejo de errores, política de reconexión MQTT, y mejores prácticas para la implementación en Raspberry Pi. Los servicios deben manejar reconexiones automáticas al broker y a la base de datos, así como registrar adecuadamente los fallos de comunicación. Los contenedores se gestionan mediante Docker Compose, permitiendo reinicios automatizados y despliegues consistentes. Es recomendable utilizar archivos .env para parametrizar credenciales y endpoints.

10.18 Detalles Adicionales de Implementación

Esta sección describe lineamientos de calidad de código, manejo de errores, política de reconexión MQTT, y mejores prácticas para la implementación en Raspberry Pi. Los servicios deben manejar reconexiones automáticas al broker y a la base de datos, así como registrar adecuadamente los fallos de comunicación. Los contenedores se gestionan mediante Docker Compose, permitiendo reinicios automatizados y despliegues consistentes. Es recomendable utilizar archivos .env para parametrizar credenciales y endpoints.

10.19 Detalles Adicionales de Implementación

Esta sección describe lineamientos de calidad de código, manejo de errores, política de reconexión MQTT, y mejores prácticas para la implementación en Raspberry Pi. Los servicios deben manejar reconexiones automáticas al broker y a la base de datos, así como registrar adecuadamente los fallos de comunicación. Los contenedores se gestionan mediante Docker Compose, permitiendo reinicios automatizados y despliegues consistentes. Es recomendable utilizar archivos .env para parametrizar credenciales y endpoints.