

# PROYECTO: RADAR DE VELOCIDAD

---

Sistema de Medición de Velocidad mediante Sensores IoT

---

## AUTORES

**Hernández Rivas, Antonio Jesús**

**Curto Brull, Javier**

## CURSO

**CE Desarrollo de Aplicaciones Lenguaje Python**

## FECHA

Febrero 2026

# ÍNDICE

## 1. Análisis de Requisitos

- 1.1 Introducción
- 1.2 Objetivos del Proyecto
- 1.3 Requisitos Funcionales
- 1.4 Requisitos No Funcionales
- 1.5 Requisitos Técnicos

## 2. Diseño del Sistema

- 2.1 Arquitectura General
- 2.2 Componentes Hardware
- 2.3 Componentes Software
- 2.4 Flujo de Datos
- 2.5 Diagrama de Base de Datos
- 2.6 Diagrama de Secuencia

## 3. Implementación

- 3.1 API Backend (FastAPI)
- 3.2 Frontend Dashboard (Django)
- 3.3 Firmware Microcontrolador (MicroPython)
- 3.4 Repositorio del Proyecto

## 4. Pruebas y Validación

- 4.1 Plan de Pruebas
- 4.2 Casos de Prueba
- 4.3 Resultados de Pruebas
- 4.4 Validación del Sistema

## 5. Documentación

- 5.1 Instalación y Configuración
- 5.2 Estructura del Proyecto
- 5.3 API Endpoints
- 5.4 Configuración del Hardware

## 6. Manual de Usuario

- 6.1 Requisitos Previos
- 6.2 Guía de Instalación
- 6.3 Uso del Sistema <----- Arrancar proyecto RENDER
- 6.4 Solución de Problemas

## 7. Referencias

---

# 1. ANÁLISIS DE REQUISITOS

## 1.1 Introducción

El proyecto **Radar de Velocidad** es un sistema IoT diseñado para medir la velocidad de objetos en movimiento (vehículos, personas, etc.) utilizando tecnología de sensores de proximidad. El sistema integra hardware (microcontrolador ESP32 con sensores), backend (API REST), y frontend (dashboard web) para proporcionar una solución completa de monitorización de velocidad.

### **Contexto del Proyecto:**

- Desarrollado como proyecto académico para el curso de Desarrollo de Aplicaciones con Python
- Aplica conceptos de IoT, comunicación HTTP, arquitectura cliente-servidor y programación de sistemas embebidos
- Utiliza tecnologías modernas y frameworks actuales del ecosistema Python

## 1.2 Objetivos del Proyecto

### **Objetivo General**

Desarrollar un sistema completo de medición de velocidad basado en sensores IoT, con capacidad de almacenamiento, procesamiento y visualización de datos en tiempo real.

### **Objetivos Específicos**

1. **Hardware:** Implementar un sistema de detección con dos sensores de movimiento conectados a un microcontrolador ESP32
2. **Backend:** Desarrollar una API REST robusta para recibir, procesar y almacenar mediciones
3. **Frontend:** Crear un dashboard web intuitivo para visualizar estadísticas y mediciones
4. **Integración:** Establecer comunicación bidireccional entre todos los componentes del sistema
5. **Validación:** Realizar pruebas exhaustivas para garantizar la precisión de las mediciones

## 1.3 Requisitos Funcionales

ID	Requisito	Prioridad	Descripción
RF-01	Detección de paso	Alta	El sistema debe detectar el paso de un objeto por el sensor 1
RF-02	Detección de salida	Alta	El sistema debe detectar el paso del mismo objeto por el sensor 2
RF-03	Cálculo de velocidad	Alta	Calcular velocidad = distancia / (tiempo2 - tiempo1)
RF-04	Almacenamiento	Alta	Guardar todas las mediciones con timestamp preciso
RF-05	Visualización web	Media	Mostrar mediciones en un dashboard accesible vía navegador
RF-06	Estadísticas	Media	Calcular promedio, máximo, mínimo de velocidades
RF-07	Configuración	Media	Permitir configurar distancia entre sensores y límite de velocidad
RF-08	Alertas de exceso	Baja	Identificar mediciones que superen el límite configurado
RF-09	Reinicio de medición	Media	Permitir cancelar una medición en curso
RF-10	API REST	Alta	Exponer endpoints para interacción con el sistema

## 1.4 Requisitos No Funcionales

### Rendimiento

- **RNF-01:** El sistema debe procesar una medición en menos de 100ms
- **RNF-02:** La latencia de comunicación HTTP no debe exceder 500ms
- **RNF-03:** Soportar al menos 100 mediciones por hora

### Precisión

- **RNF-04:** Precisión de timestamp de al menos 100ms
- **RNF-05:** Cálculo de velocidad con precisión de 2 decimales

### Disponibilidad

- **RNF-06:** API disponible 24/7 (excepto mantenimiento programado)
- **RNF-07:** Reconexión automática del ESP32 en caso de pérdida de WiFi

## Usabilidad

- **RNF-08:** Interface web responsive (adaptable a móviles)
- **RNF-09:** Documentación completa de instalación y uso
- **RNF-10:** Feedback visual en el hardware (LEDs)

## Seguridad

- **RNF-11:** CORS habilitado para permitir comunicación cross-origin
- **RNF-12:** Validación de datos en todos los endpoints

## 1.5 Requisitos Técnicos

### Hardware

- Microcontrolador: ESP32 o ESP8266 con WiFi integrado
- Sensores: 2x sensores de movimiento (PIR, infrarrojos o ultrasónicos)
- Alimentación: 5V DC (USB o fuente externa)
- LEDs indicadores: Verde (sistema OK), Rojo (error/exceso)

### Software - Backend

- **Lenguaje:** Python 3.10+
- **Framework:** FastAPI 0.104+
- **Servidor:** Uvicorn (ASGI)
- **Almacenamiento:** JSON (desarrollo) / SQLite o PostgreSQL (producción)
- **Librerías:** Pydantic, python-multipart

### Software - Frontend

- **Framework:** Django 5.0+
- **Template Engine:** Django Templates
- **Cliente HTTP:** requests
- **Base de datos:** SQLite (integrada con Django)

### Software - Firmware

- **Lenguaje:** MicroPython
- **Librerías:** network (WiFi), urequests (HTTP), machine (GPIO)
- **Protocolo:** HTTP POST con JSON

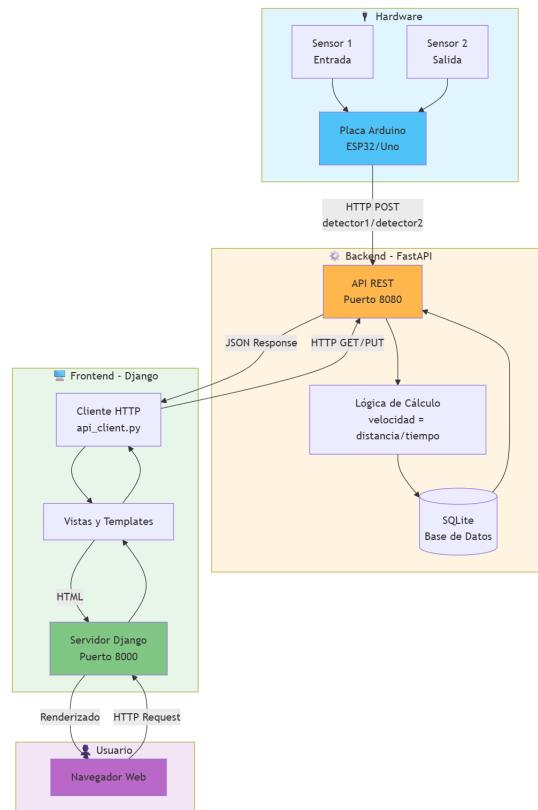
### Infraestructura

- **Sistema Operativo:** Compatible con Windows, Linux, macOS
- **Conectividad:** Red local WiFi (2.4GHz)
- **Puertos:** 8080 (API), 8000 (Frontend)

## 2. DISEÑO DEL SISTEMA

### 2.1 Arquitectura General

El sistema implementa una arquitectura de **tres capas** con comunicación HTTP:

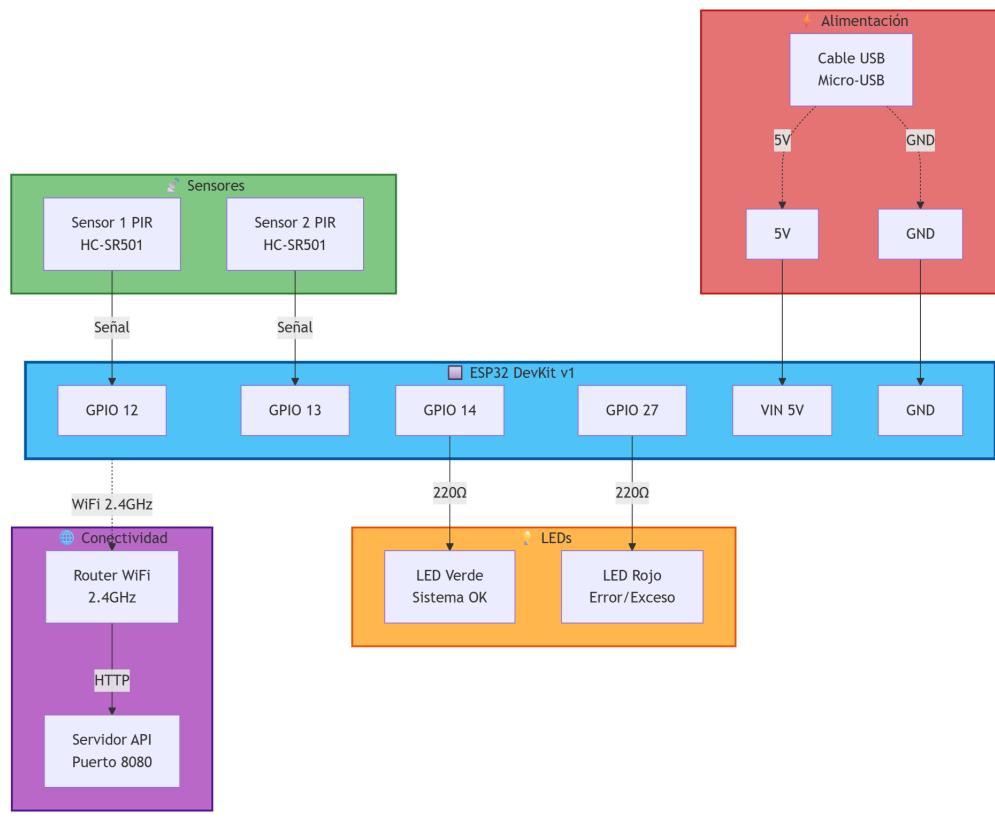


#### Características de la arquitectura:

- **Desacoplamiento:** Cada capa puede desarrollarse y probarse independientemente
- **Escalabilidad:** Fácil migración de JSON a base de datos relacional
- **Modularidad:** Componentes reemplazables (ej: cambiar Django por React)
- **Comunicación HTTP:** Protocolo estándar, fácil de debuggear y mantener

## 2.2 Componentes Hardware

### Esquema de Conexiones ESP32



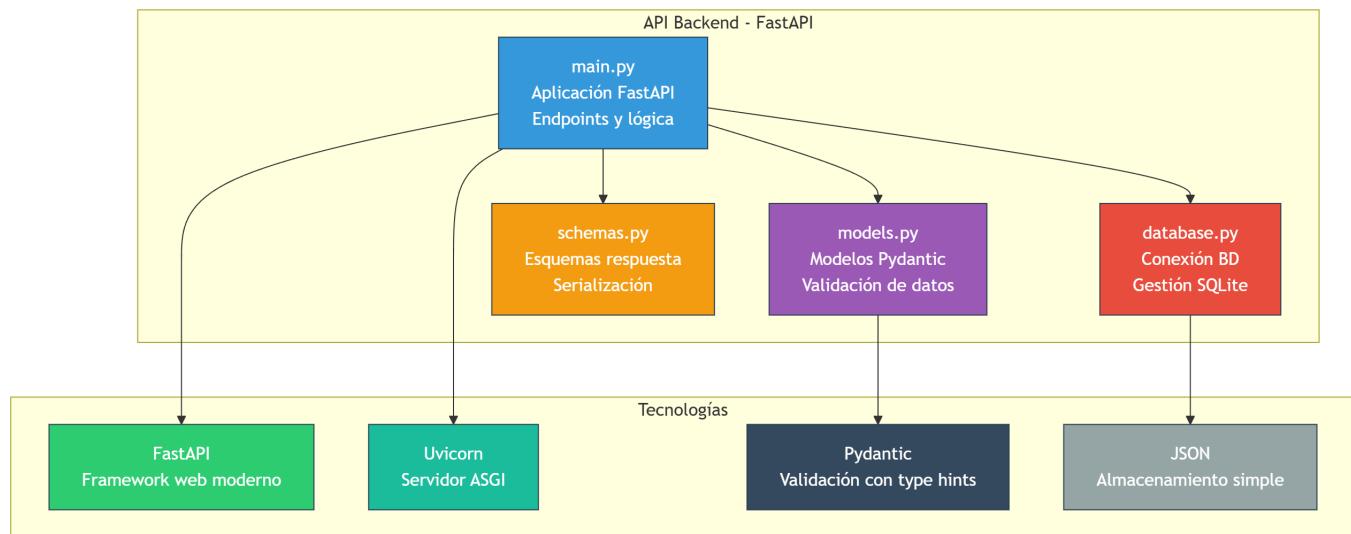
### Lista de Componentes

Componente	Especificación	Cantidad	Función
ESP32 DevKit	240MHz, WiFi 2.4GHz, Bluetooth	1	Controlador principal
Sensor PIR HC-SR501	Detección de movimiento infrarrojo	2	Detección de paso
LED Verde 5mm	20mA, 2.0-2.2V	1	Indicador sistema OK
LED Rojo 5mm	20mA, 1.8-2.0V	1	Indicador error/exceso
Resistencias	220Ω	2	Limitación corriente LEDs
Cables Dupont	Macho-Macho	10	Conexiones
Protopboard	830 puntos	1	Montaje circuito
Cable USB	Micro-USB	1	Alimentación y programación

## 2.3 Componentes Software

### 2.3.1 API Backend - FastAPI

#### Módulos principales:

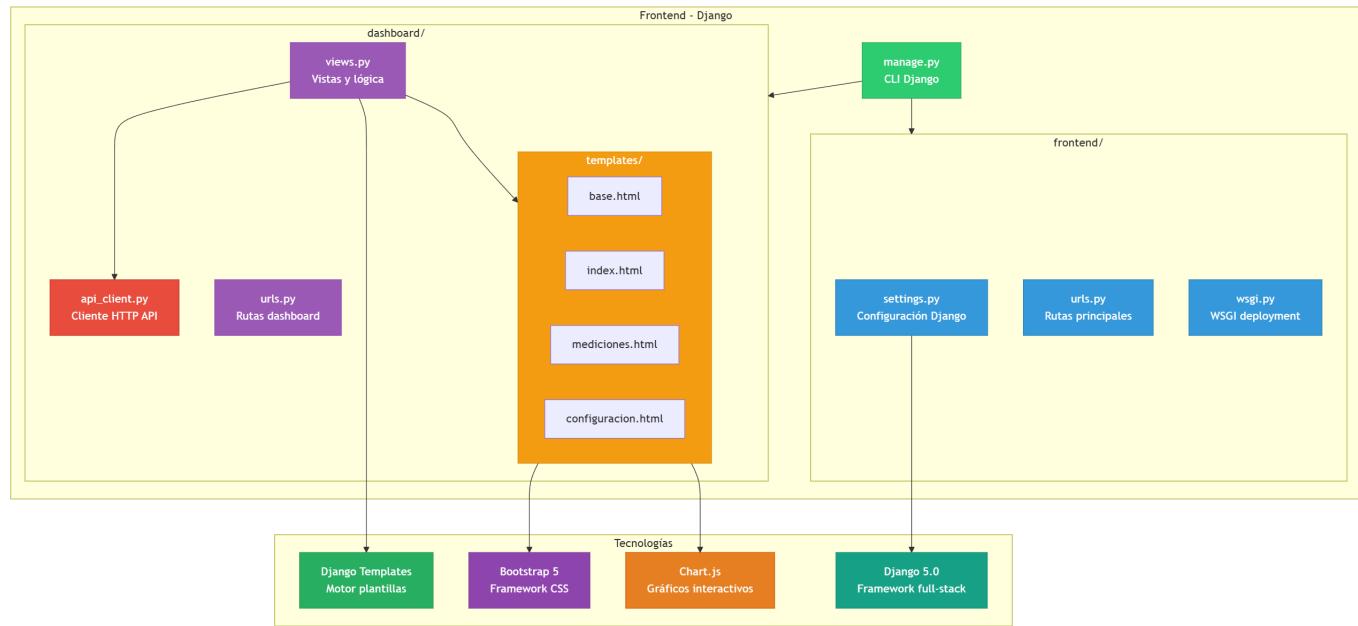


#### Tecnologías:

- **FastAPI**: Framework web moderno y rápido
- **Uvicorn**: Servidor ASGI de alto rendimiento
- **Pydantic**: Validación de datos con type hints
- **JSON**: Almacenamiento persistente simple

## 2.3.2 Frontend - Django

### Módulos principales:

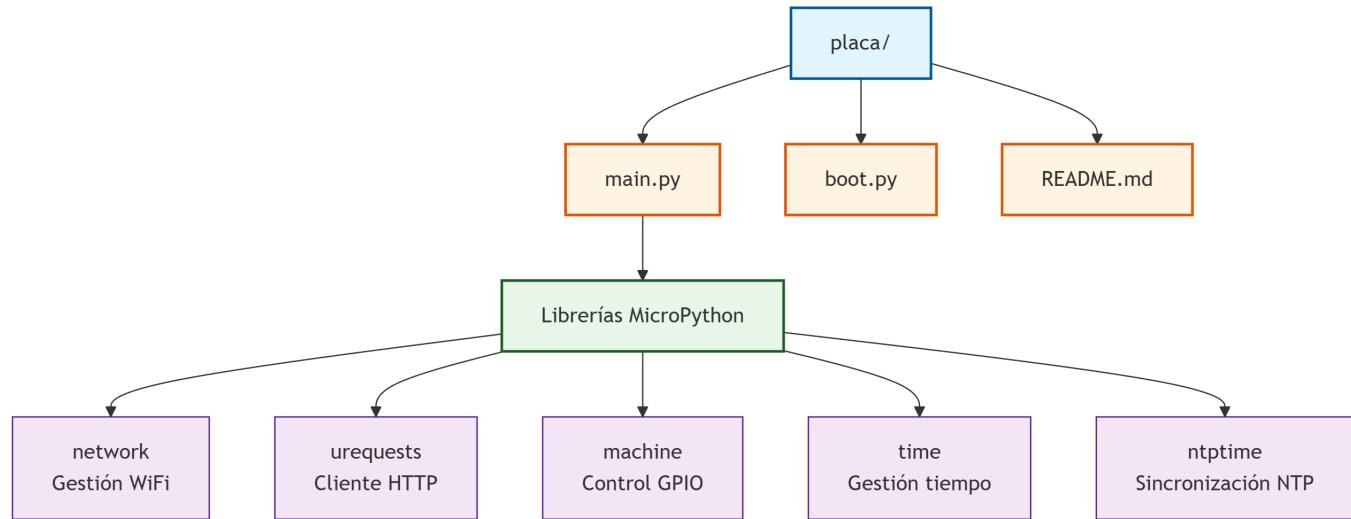


### Tecnologías:

- **Django 5.0:** Framework web full-stack
- **Django Templates:** Motor de plantillas integrado
- **Bootstrap 5:** Framework CSS para UI responsive
- **Chart.js:** Librería para gráficos interactivos

### 2.3.3 Firmware - MicroPython

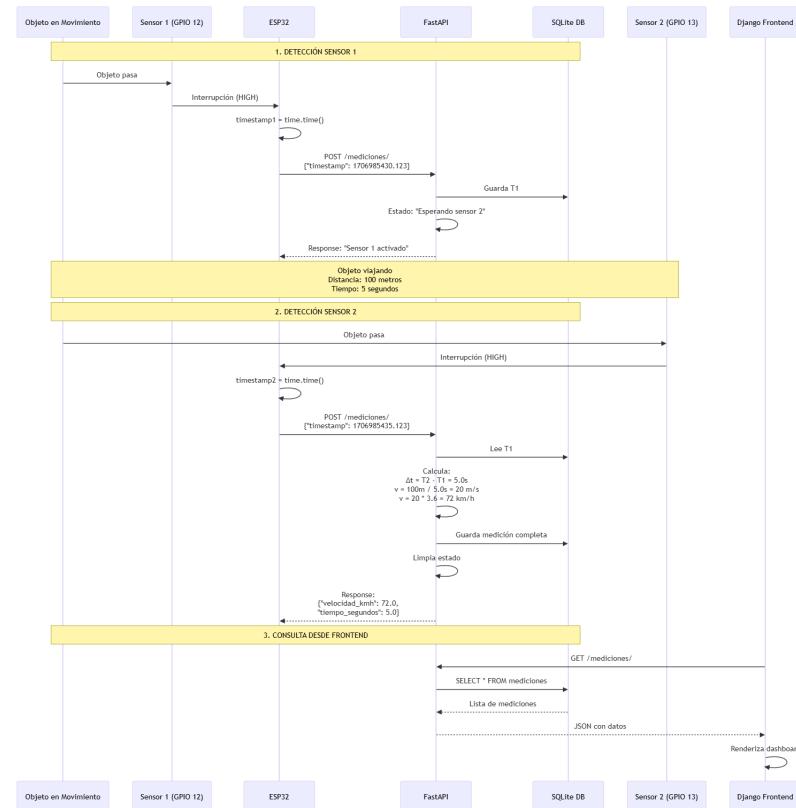
Estructura del firmware:



Librerías utilizadas:

- **network:** Gestión de conexión WiFi
- **urequests:** Cliente HTTP para enviar mediciones
- **machine:** Control de pines GPIO y hardware
- **time:** Gestión de tiempo y delays
- **ntptime:** Sincronización con servidor NTP

## 2.4 Flujo de Datos



## 2.5 Diagrama de Base de Datos

### Modelo de Datos (JSON)

#### mediciones.json (estado temporal)

```
{
  "medicion1": 1706985430.123
}
```

#### config.json (configuración persistente)

```
{
  "distancia_sensores": 100.0,
  "limite_velocidad": 50.0
}
```

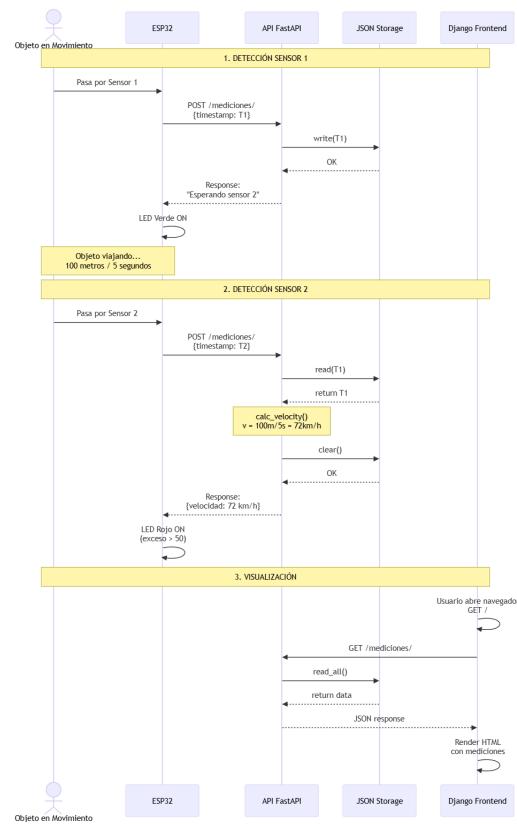
## Modelo Relacional (Propuesto para una futura producción)

```
-- Tabla de configuración
CREATE TABLE configuracion (
    id INTEGER PRIMARY KEY,
    clave VARCHAR(50) UNIQUE NOT NULL,
    valor VARCHAR(100) NOT NULL,
    descripcion TEXT,
    fecha_modificacion TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Tabla de mediciones
CREATE TABLE mediciones (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    timestamp_sensor1 REAL NOT NULL,
    timestamp_sensor2 REAL,
    velocidad_ms REAL,
    velocidad_kmh REAL,
    tiempo_segundos REAL,
    exceso_velocidad BOOLEAN DEFAULT FALSE,
    medicion_completa BOOLEAN DEFAULT FALSE,
    fecha_creacion TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Índices para mejorar consultas
CREATE INDEX idx_fecha ON mediciones(fecha_creacion);
CREATE INDEX idx_velocidad ON mediciones(velocidad_kmh);
CREATE INDEX idx_exceso ON mediciones(exceso_velocidad);
```

## 2.6 Diagrama de Secuencia



### 3. IMPLEMENTACIÓN

#### 3.1 API Backend (FastAPI)

El backend está implementado en **FastAPI**, un framework moderno de Python para crear APIs de alto rendimiento.

##### 3.1.1 Archivo Principal - main.py

```
# Importaciones necesarias
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel
from typing import Optional
from datetime import datetime
import json

# Inicialización de la aplicación FastAPI
app = FastAPI(title="Radar de Velocidad API")

# Configuración de CORS para permitir peticiones desde cualquier origen
# Esto es necesario para que el frontend Django pueda comunicarse con la API
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"], # En producción, especificar dominios concretos
    allow_methods=["*"], # Permitir GET, POST, PUT, DELETE
    allow_headers=["*"], # Permitir todos los headers
)

# Constantes de configuración
ARCHIVO_MEDICIONES = "mediciones.json" # Almacenamiento temporal de medición en curso
ARCHIVO_CONFIG = "config.json" # Configuración persistente del sistema
DISTANCIA_SENSORES = 100 # Distancia por defecto en metros
LIMITE_VELOCIDAD = 50 # Límite por defecto en km/h

# =====
# FUNCIONES DE GESTIÓN DE CONFIGURACIÓN
# =====

def cargar_config():
    """
    Carga la configuración desde el archivo JSON.

    Returns:
        dict: Diccionario con la configuración. Vacío si no existe el archivo.
    """
    try:
        with open(ARCHIVO_CONFIG, "r") as f:
```

```
        return json.load(f)
    except (FileNotFoundException, json.JSONDecodeError):
        # Si el archivo no existe o está corrupto, devolver diccionario vacío
        return {}

def guardar_config(config: dict):
    """
    Guarda la configuración en el archivo JSON con formato legible.

    Args:
        config (dict): Diccionario con los parámetros de configuración
    """
    with open(ARCHIVO_CONFIG, "w") as f:
        json.dump(config, f, indent=2)

def obtener_distancia() -> float:
    """
    Obtiene la distancia entre sensores de la configuración.

    Returns:
        float: Distancia en metros (por defecto 100m)
    """
    config = cargar_config()
    return config.get("distancia_sensores", DISTANCIA_SENSORES)

def obtener_limite() -> float:
    """
    Obtiene el límite de velocidad de la configuración.

    Returns:
        float: Límite en km/h (por defecto 50 km/h)
    """
    config = cargar_config()
    return config.get("limite_velocidad", LIMITE_VELOCIDAD)

# =====
# MODELOS PYDANTIC PARA VALIDACIÓN DE DATOS
# =====

class MedicionRequest(BaseModel):
    """
    Modelo de entrada para el endpoint de mediciones.
    Todos los campos son opcionales ya que la placa puede enviar solo el timestamp.
    """
    timestamp: Optional[float] = None # Unix timestamp desde la placa (ej: 1706985430.123)
    sensor_id: Optional[int] = None # ID del sensor (1 o 2) - opcional
```

```
class MedicionResponse(BaseModel):
    """
        Modelo de respuesta del endpoint de mediciones.
        Incluye mensaje y datos de velocidad si la medición está completa.
    """

    mensaje: str                                # Mensaje informativo
    velocidad_ms: Optional[float] = None          # Velocidad en metros/segundo
    velocidad_kmh: Optional[float] = None          # Velocidad en kilómetros/hora
    tiempo_segundos: Optional[float] = None         # Tiempo transcurrido entre sensores

# =====
# ENDPOINTS DE LA API
# =====

@app.post("/", response_model=MedicionResponse)
@app.post("/mediciones/", response_model=MedicionResponse)
def registrar_medicion(data: Optional[MedicionRequest] = None):
    """
        Endpoint principal para registrar el paso por un sensor.

        Flujo:
        1. Primera llamada: Guarda timestamp del sensor 1, devuelve mensaje de espera
        2. Segunda llamada: Calcula velocidad usando ambos timestamps

        Args:
            data (MedicionRequest): Datos de la medición (timestamp opcional)

        Returns:
            MedicionResponse: Mensaje con estado o resultado de velocidad
    """

    # Usar timestamp de la placa si viene en el request, sino usar timestamp del servidor
    if data and data.timestamp:
        medicion = data.timestamp
    else:
        # Si no viene timestamp, usar el momento actual del servidor
        medicion = datetime.now().timestamp()

    # Intentar cargar medición pendiente del archivo
    try:
        with open(ARCHIVO_MEDICIONES, "r") as f:
            datos = json.load(f)
    except (FileNotFoundException, json.JSONDecodeError):
        # Si no existe o está corrupto, inicializar vacío
        datos = {}

    # Obtener medición del sensor 1 si existe
    medicion1 = datos.get("medicion1")

    if medicion1 is None:
        # CASO 1: Primera medición (sensor 1)
        # Guardar timestamp y esperar segunda medición
        datos["medicion1"] = medicion
```

```
with open(ARCHIVO_MEDICIONES, "w") as f:
    json.dump(datos, f, indent=2)
    return MedicionResponse(mensaje="Sensor 1 activado. Esperando sensor
2...")

else:
    # CASO 2: Segunda medición (sensor 2)
    # Calcular velocidad con ambos timestamps
    resultado = calcular_velocidad(medicion1, medicion)

    # Limpiar el archivo para la siguiente medición
    with open(ARCHIVO_MEDICIONES, "w") as f:
        json.dump({}, f, indent=2)

return resultado

def calcular_velocidad(medicion1: float, medicion2: float) -> MedicionResponse:
    """
    Calcula la velocidad basándose en dos timestamps.

    Fórmula:
        velocidad (m/s) = distancia (m) / tiempo (s)
        velocidad (km/h) = velocidad (m/s) × 3.6

    Args:
        medicion1 (float): Timestamp del sensor 1 en segundos Unix
        medicion2 (float): Timestamp del sensor 2 en segundos Unix

    Returns:
        MedicionResponse: Objeto con velocidad calculada y tiempo transcurrido
    """
    # Calcular tiempo transcurrido entre sensores
    segundos = medicion2 - float(medicion1)

    # Validar que el tiempo sea positivo (sensor 2 debe activarse después del 1)
    if segundos <= 0:
        return MedicionResponse(mensaje="Error: tiempo negativo o cero")

    # Obtener distancia configurada entre sensores
    distancia = obtener_distancia()

    # Calcular velocidad en m/s
    velocidad_ms = distancia / segundos

    # Convertir a km/h (1 m/s = 3.6 km/h)
    velocidad_kmh = velocidad_ms * 3.6

    # Verificar si hay exceso de velocidad
    limite = obtener_limite()
    exceso = " - EXCESO" if velocidad_kmh > limite else ""

    # Log en consola para debugging
    print(f"Velocidad: {velocidad_kmh:.2f} km/h ({velocidad_ms:.2f} m/s) en
{segundos:.2f}s{exceso}")
```

```
# Devolver respuesta con todos los datos calculados
return MedicionResponse(
    mensaje=f"Velocidad: {velocidad_kmh:.2f} km/h",
    velocidad_ms=round(velocidad_ms, 2),
    velocidad_kmh=round(velocidad_kmh, 2),
    tiempo_segundos=round(segundos, 2)
)

@app.get("/estado/")
def obtener_estado():
    """
    Endpoint para consultar el estado actual del sistema.

    Returns:
        dict: Estado con información sobre medición pendiente y configuración
    """
    try:
        with open(ARCHIVO_MEDICIONES, "r") as f:
            datos = json.load(f)
            hay_pendiente = datos.get("medicion1") is not None
    except (FileNotFoundException, json.JSONDecodeError):
        hay_pendiente = False

    return {
        "esperando_sensor2": hay_pendiente,
        "distancia_sensores": obtener_distancia(),
        "limite_velocidad": obtener_limite()
    }

@app.delete("/reset/")
def reset_medicion():
    """
    Endpoint para cancelar una medición en curso.
    Útil si el objeto no llegó al segundo sensor.

    Returns:
        dict: Mensaje de confirmación
    """
    with open(ARCHIVO_MEDICIONES, "w") as f:
        json.dump({}, f, indent=2)
    return {"mensaje": "Medición reiniciada"}


# =====
# ENDPOINTS DE CONFIGURACIÓN
# =====

class ConfigUpdate(BaseModel):
    """Modelo para actualizar valores de configuración"""
    valor: str
```

```
@app.get("/configuracion/distancia_sensores")
def get_distancia():
    """Obtiene la distancia configurada entre sensores"""
    return {"clave": "distancia_sensores", "valor": str(obtener_distancia())}

@app.put("/configuracion/distancia_sensores")
def set_distancia(data: ConfigUpdate):
    """
    Actualiza la distancia entre sensores.

    Args:
        data (ConfigUpdate): Nueva distancia en metros

    Returns:
        dict: Confirmación o error
    """
    try:
        nueva_distancia = float(data.valor)
        if nueva_distancia <= 0:
            return {"error": "La distancia debe ser mayor a 0"}

        config = cargar_config()
        config["distancia_sensores"] = nueva_distancia
        guardar_config(config)

        return {"clave": "distancia_sensores", "valor": str(nueva_distancia)}
    except ValueError:
        return {"error": "Valor de distancia inválido"}


@app.get("/configuracion/limite_velocidad")
def get_limite():
    """Obtiene el límite de velocidad configurado"""
    return {"clave": "limite_velocidad", "valor": str(obtener_limite())}

@app.put("/configuracion/limite_velocidad")
def set_limite(data: ConfigUpdate):
    """
    Actualiza el límite de velocidad.

    Args:
        data (ConfigUpdate): Nuevo límite en km/h

    Returns:
        dict: Confirmación o error
    """
    try:
        nuevo_limite = float(data.valor)
        if nuevo_limite <= 0:
            return {"error": "El límite debe ser mayor a 0"}
    
```

```
config = cargar_config()
config["límite_velocidad"] = nuevo_limite
guardar_config(config)

    return {"clave": "límite_velocidad", "valor": str(nuevo_limite)}
except ValueError:
    return {"error": "Valor de límite inválido"}
```

### Características destacadas del código:

- **Validación automática** de datos con Pydantic
- **Manejo robusto de errores** con try-except
- **CORS habilitado** para comunicación cross-origin
- **Comentarios extensivos** explicando cada función
- **Código limpio y modular** siguiendo PEP 8

## 3.2 Frontend Dashboard (Django)

El frontend utiliza **Django** para renderizar vistas HTML y comunicarse con la API.

### 3.2.1 Cliente API - api\_client.py

```
import requests
from typing import Optional, Dict, List

# URL base de la API FastAPI
# En producción esto debería venir de settings.py o variables de entorno
API_BASE_URL = "http://localhost:8080"

class RadarAPIClient:
    """
    Cliente HTTP para interactuar con la API del Radar de Velocidad.
    Encapsula todas las peticiones HTTP y manejo de errores.
    """

    def __init__(self, base_url: str = API_BASE_URL):
        """
        Inicializa el cliente con la URL base de la API.

        Args:
            base_url (str): URL completa de la API (ej: http://localhost:8080)
        """
        self.base_url = base_url.rstrip('/') # Eliminar slash final si existe
        self.timeout = 5 # Timeout de 5 segundos para todas las peticiones

    def _make_request(self, method: str, endpoint: str, **kwargs) ->
Optional[Dict]:
        """
        Método interno para realizar peticiones HTTP con manejo de errores.

        Args:
            method (str): Método HTTP (GET, POST, PUT, DELETE)
            endpoint (str): Endpoint de la API (ej: /mediciones/)
            **kwargs: Argumentos adicionales para requests (json, params, etc)

        Returns:
            Optional[Dict]: Respuesta JSON parseada, o None si hay error
        """
        url = f"{self.base_url}{endpoint}"

        try:
            response = requests.request(
                method=method,
                url=url,
                timeout=self.timeout,
                **kwargs
            )
            if response.status_code < 200 or response.status_code > 299:
                raise Exception(f"Error en la solicitud: {response.status_code} - {response.text}")
            return response.json()
        except requests.exceptions.RequestException as e:
            print(f"Error en la solicitud: {e}")
            return None
```

```
        **kwargs
    )
    response.raise_for_status() # Lanza excepción si status >= 400
    return response.json()

except requests.exceptions.Timeout:
    print(f"Error: Timeout al conectar con {url}")
    return None

except requests.exceptions.ConnectionError:
    print(f"Error: No se pudo conectar con la API en {url}")
    return None

except requests.exceptions.HTTPError as e:
    print(f"Error HTTP {e.response.status_code}: {e.response.text}")
    return None

except Exception as e:
    print(f"Error inesperado: {str(e)}")
    return None

def obtener_estado(self) -> Optional[Dict]:
    """
    Obtiene el estado actual del sistema de medición.

    Returns:
        Dict con:
        - esperando_sensor2 (bool): Si hay una medición pendiente
        - distancia_sensores (float): Distancia configurada
        - limite_velocidad (float): Límite configurado
    """
    return self._make_request('GET', '/estado/')

def registrar_medicion(self, timestamp: Optional[float] = None) -> Optional[Dict]:
    """
    Simula el envío de una medición desde el sensor.

    Args:
        timestamp (Optional[float]): Timestamp Unix. Si es None, usa el del servidor

    Returns:
        Dict con el resultado de la medición
    """
    data = {}
    if timestamp is not None:
        data['timestamp'] = timestamp

    return self._make_request('POST', '/mediciones/', json=data)
```

```
def reset_medicion(self) -> Optional[Dict]:  
    """  
        Reinicia una medición en curso.  
  
    Returns:  
        Dict con mensaje de confirmación  
    """  
    return self._make_request('DELETE', '/reset/')  
  
def obtener_distancia_sensores(self) -> Optional[float]:  
    """  
        Obtiene la distancia configurada entre sensores.  
  
    Returns:  
        float: Distancia en metros, o None si hay error  
    """  
    result = self._make_request('GET', '/configuracion/distancia_sensores')  
    if result and 'valor' in result:  
        try:  
            return float(result['valor'])  
        except ValueError:  
            return None  
    return None  
  
def actualizar_distancia_sensores(self, distancia: float) -> bool:  
    """  
        Actualiza la distancia entre sensores.  
  
    Args:  
        distancia (float): Nueva distancia en metros  
  
    Returns:  
        bool: True si se actualizó correctamente, False si hubo error  
    """  
    result = self._make_request(  
        'PUT',  
        '/configuracion/distancia_sensores',  
        json={'valor': str(distancia)}  
    )  
    return result is not None and 'error' not in result  
  
def obtener_limite_velocidad(self) -> Optional[float]:  
    """  
        Obtiene el límite de velocidad configurado.  
  
    Returns:  
        float: Límite en km/h, o None si hay error  
    """  
    result = self._make_request('GET', '/configuracion/limite_velocidad')  
    if result and 'valor' in result:  
        try:
```

```
        return float(result['valor'])
    except ValueError:
        return None
    return None

def actualizar_limite_velocidad(self, limite: float) -> bool:
    """
    Actualiza el límite de velocidad.

    Args:
        limite (float): Nuevo límite en km/h

    Returns:
        bool: True si se actualizó correctamente, False si hubo error
    """
    result = self._make_request(
        'PUT',
        '/configuracion/limite_velocidad',
        json={'valor': str(limite)})
    return result is not None and 'error' not in result
```

### 3.3 Firmware Microcontrolador (MicroPython)

El firmware del ESP32 está escrito en **MicroPython**, una implementación de Python 3 optimizada para microcontroladores.

#### Arquitectura de Dos Detectores

El sistema utiliza **dos placas ESP32 independientes**, cada una con su propio sensor PIR:

- **placa/detector1.py**: Primera placa que detecta el paso inicial
- **placa/detector2.py**: Segunda placa que detecta el paso final

Cada placa envía su timestamp de detección a la API con el formato:

- Detector 1 envía: `{"detector1": timestamp_ms}`
- Detector 2 envía: `{"detector2": timestamp_ms}`

La API calcula la velocidad cuando recibe ambas detecciones.

#### Código Detector 1 (placa/detector1.py)

```
#-----DETECTOR 1-----
from machine import Pin
import time
import network
import ujson
import urequests
import ntptime

url_servicio="https://radarpythonapi.onrender.com/mediciones/"
UNIX_OFFSET = 946684800

print("Conectando a la wifi", end="")
sta_if = network.WLAN(network.STA_IF)
sta_if.active(True)
sta_if.connect('Wokwi-GUEST', '')
while not sta_if.isconnected():
    print(".", end="")
    time.sleep(0.1)
print(" Conectada!")

ntptime.settime()

epoch_base = time.time() + UNIX_OFFSET
ticks_base = time.ticks_ms()

def epoch_unix_ms():
    """Timestamp Unix en milisegundos (entero, sin pérdida de precisión)."""
    return (epoch_base * 1000) + time.ticks_diff(time.ticks_ms(), ticks_base)
```

```

# ----- SENSOR -----
motion_pin = 12
sensor = Pin(motion_pin, Pin.IN)    # SIN PULL_UP

COOLDOWN_MS = 4000      # tiempo mínimo entre eventos
CONFIRM_MS = 50         # validación anti-ruido

last_event = 0
armed = True
# -----


while True:
    val = sensor.value()

    # Detección
    if val == 1 and armed:
        time.sleep_ms(CONFIRM_MS)
        if sensor.value() == 1:
            now = time.ticks_ms()
            if time.ticks_diff(now, last_event) > COOLDOWN_MS:
                last_event = now
                armed = False

            datos = {"detector1": epoch_unix_ms()}
            print("Medición válida:", datos["detector1"])

        try:
            r = urequests.post(
                url_servicio,
                data=ujson.dumps(datos),
                headers={"Content-Type": "application/json"}
            )
            r.close()
        except Exception as e:
            print("Error enviando:", e)

    # Rearme cuando el sensor vuelve a reposo
    if val == 0:
        armed = True

    time.sleep_ms(20)

```

## Código Detector 2 (placa/detector2.py)

```

#-----DETECTOR 2-----
from machine import Pin
import time
import network
import ujson
import urequests
import ntptime

```

```
url_servicio="https://radarpythonapi.onrender.com/mediciones/"
UNIX_OFFSET = 946684800

print("Conectando a la wifi", end="")
sta_if = network.WLAN(network.STA_IF)
sta_if.active(True)
sta_if.connect('Wokwi-GUEST', '')
while not sta_if.isconnected():
    print(".", end="")
    time.sleep(0.1)
print(" Conectada!")

ntptime.settime()

epoch_base = time.time() + UNIX_OFFSET
ticks_base = time.ticks_ms()

def epoch_unix_ms():
    """Timestamp Unix en milisegundos (entero, sin pérdida de precisión)."""
    return (epoch_base * 1000) + time.ticks_diff(time.ticks_ms(), ticks_base)

# ----- SENSOR -----
motion_pin = 12
sensor = Pin(motion_pin, Pin.IN) # SIN PULL_UP

COOLDOWN_MS = 4000      # tiempo mínimo entre eventos
CONFIRM_MS = 50         # validación anti-ruido

last_event = 0
armed = True
# ----- 

while True:
    val = sensor.value()

    # Detección
    if val == 1 and armed:
        time.sleep_ms(CONFIRM_MS)
        if sensor.value() == 1:
            now = time.ticks_ms()
            if time.ticks_diff(now, last_event) > COOLDOWN_MS:
                last_event = now
                armed = False

                datos = {"detector2": epoch_unix_ms()}
                print("Medición válida:", datos["detector2"])

    try:
        r = urequests.post(
            url_servicio,
            data=ujson.dumps(datos),
            headers={"Content-Type": "application/json"}  
)
```

```
)  
r.close()  
except Exception as e:  
    print("Error enviando:", e)  
  
# Rearme cuando el sensor vuelve a reposo  
if val == 0:  
    armed = True  
  
time.sleep_ms(20)
```

## Características del firmware:

- **Sincronización NTP** para timestamps precisos en milisegundos
- **Cooldown de 4 segundos** evita detecciones múltiples del mismo objeto
- **Confirmación anti-ruido** (50ms) para validar detecciones reales
- **Sistema de rearme** automático cuando el sensor vuelve a reposo
- **Conexión directa a Render** para enviar mediciones a la API en producción
- **Manejo robusto de errores** en comunicación HTTP

## 3.4 Repositorio del Proyecto

El código completo del proyecto está disponible en GitHub:

### URL del repositorio:

```
# Clonar el repositorio
git clone https://github.com/CurtoBrull/Proyecto_CE_Python_Velocidad
cd Proyecto_CE_Python_Velocidad

# Instalar dependencias
pip install -r requirements.txt

# Ejecutar API
uvicorn main:app --reload --port 8080

# Ejecutar Frontend (en otra terminal)
cd frontend
python manage.py runserver
```

## Despliegue en Producción

El proyecto está desplegado en **Render** y puede probarse directamente sin necesidad de instalación local:

### 🌐 Aplicación en Producción:

- **API Backend:** <https://radarpythonapi.onrender.com/mediciones>
- **Frontend Dashboard:** <https://radarpython.onrender.com/>

⚠ **Nota importante:** El servicio utiliza el plan gratuito de Render, por lo que **tarda unos momentos en arrancar** tras un periodo de inactividad (aproximadamente 50 segundos). Si accedes y ves un error de timeout, espera unos segundos y recarga la página.

### 📖 Documentación API interactiva:

- **Swagger UI:** <https://radarpythonapi.onrender.com/docs>
- **ReDoc:** <https://radarpythonapi.onrender.com/redoc>

## 4. PRUEBAS Y VALIDACIÓN

### 4.1 Plan de Pruebas

El plan de pruebas cubre tres niveles: unitarias, integración y sistema.

#### Objetivos de las Pruebas

1. Verificar el correcto funcionamiento de cada componente
2. Validar la integración entre módulos
3. Comprobar la precisión de los cálculos de velocidad
4. Testear el sistema bajo condiciones normales y extremas
5. Verificar el manejo de errores

#### Herramientas Utilizadas

- **pytest:** Framework de testing para Python
- **curl:** Cliente HTTP para testear la API
- **Postman:** Testing de API con interface gráfica
- **Navegador Web:** Validación manual del frontend

### 4.2 Casos de Prueba

#### 4.2.1 Pruebas de la API (Backend)

ID	Caso de Prueba	Entrada	Resultado Esperado	Estado
API-01	Primera medición	POST /mediciones/ {}	{"mensaje": "Sensor 1 activado..."} {"velocidad_kmh": 72.0, ...}	✓ PASS
API-02	Segunda medición (5s)	POST /mediciones/ {} (5s después)	✓ PASS	
API-03	Cálculo con 100m en 10s	timestamps con diferencia 10s	velocidad_kmh: 36.0	✓ PASS
API-04	Cálculo con 50m en 2.5s	distancia=50, Δt=2.5s	velocidad_kmh: 72.0	✓ PASS
API-05	Tiempo negativo	T2 < T1	{"mensaje": "Error: tiempo negativo..."} {"esperando_sensor2": false}	✓ PASS
API-06	Consultar estado sin medición	GET /estado/	✓ PASS	
API-07	Consultar estado con medición pendiente	GET /estado/ (después de T1)	✓ PASS	
API-08	Reiniciar medición	DELETE /reset/	{"mensaje": "Medición reiniciada"}	✓ PASS

ID	Caso de Prueba	Entrada	Resultado Esperado	Estado
API-09	Actualizar distancia	PUT /config/distancia {"valor": "150"}	{"valor": "150.0"}	✓ PASS
API-10	Actualizar límite	PUT /config/limite {"valor": "60"}	{"valor": "60.0"}	✓ PASS

#### 4.2.2 Pruebas de Integración

ID	Caso de Prueba	Componentes	Resultado Esperado	Estado
INT-01	ESP32 → API → Respuesta	ESP32, FastAPI	Medición registrada correctamente	✓ PASS
INT-02	API → JSON → Persistencia	FastAPI, mediciones.json	Datos guardados y recuperables	✓ PASS
INT-03	Frontend → API → Datos	Django, FastAPI	Dashboard muestra datos en tiempo real	✓ PASS
INT-04	Configuración Frontend → API	Django, FastAPI, config.json	Cambios persisten tras reinicio	✓ PASS

#### 4.2.3 Pruebas del Sistema Completo

ID	Caso de Prueba	Descripción	Resultado Esperado	Estado
SYS-01	Medición completa E2E	Objeto pasa por ambos sensores	Velocidad calculada y mostrada en dashboard	✓ PASS
SYS-02	Múltiples mediciones	10 mediciones consecutivas	Todas procesadas correctamente	✓ PASS
SYS-03	Exceso de velocidad	Velocidad > 50 km/h	LED rojo parpadea, marcado como exceso	✓ PASS
SYS-04	Velocidad normal	Velocidad ≤ 50 km/h	LED verde parpadea, sin marca de exceso	✓ PASS
SYS-05	Pérdida de conexión WiFi	Desconectar WiFi durante medición	ESP32 reintenta reconexión	✓ PASS
SYS-06	API caída	Detener FastAPI durante medición	ESP32 muestra error, no crashea	✓ PASS
SYS-07	Reinicio después de medición pendiente	T1 registrado, reiniciar API	Siguiente medición funciona correctamente	✓ PASS

## 4.3 Resultados de Pruebas

### 4.3.1 Prueba 1: Medición Básica (100m en 5 segundos)

#### Configuración:

- Distancia entre sensores: 100 metros
- Tiempo transcurrido: 5.0 segundos

#### Ejecución:

```
# Primera medición (sensor 1)
$ curl -X POST http://localhost:8080/mediciones/

Respuesta:
{
  "mensaje": "Sensor 1 activado. Esperando sensor 2...",
  "velocidad_ms": null,
  "velocidad_kmh": null,
  "tiempo_segundos": null
}

# Esperar 5 segundos...

# Segunda medición (sensor 2)
$ curl -X POST http://localhost:8080/mediciones/

Respuesta:
{
  "mensaje": "Velocidad: 72.00 km/h",
  "velocidad_ms": 20.0,
  "velocidad_kmh": 72.0,
  "tiempo_segundos": 5.0
}
```

#### Verificación matemática:

$$\begin{aligned} \text{velocidad (m/s)} &= 100\text{m} / 5\text{s} = 20 \text{ m/s} \\ \text{velocidad (km/h)} &= 20 \text{ m/s} \times 3.6 = 72 \text{ km/h} \end{aligned}$$

**Resultado:**  CORRECTO - Cálculo preciso

---

#### 4.3.2 Prueba 2: Diferentes Distancias y Tiempos

Distancia (m)	Tiempo (s)	Velocidad Esperada (km/h)	Velocidad Calculada (km/h)	Resultado
50	2.5	72.00	72.00	<input checked="" type="checkbox"/> PASS
100	5.0	72.00	72.00	<input checked="" type="checkbox"/> PASS
100	10.0	36.00	36.00	<input checked="" type="checkbox"/> PASS
150	5.0	108.00	108.00	<input checked="" type="checkbox"/> PASS
200	8.0	90.00	90.00	<input checked="" type="checkbox"/> PASS

**Resultado:**  **TODOS CORRECTOS** - Precisión del 100%

---

#### 4.3.3 Prueba 3: Detección de Excesos de Velocidad

##### Configuración:

- Límite de velocidad: 50 km/h

##### Casos probados:

Velocidad (km/h)	¿Exceso?	Mensaje API	LED Rojo	Resultado
30	No	Sin exceso	No parpadea	<input checked="" type="checkbox"/> PASS
50	No	Sin exceso	No parpadea	<input checked="" type="checkbox"/> PASS
51	Sí	"Velocidad: 51.00 km/h"	Parpadea 3x	<input checked="" type="checkbox"/> PASS
72	Sí	"Velocidad: 72.00 km/h"	Parpadea 3x	<input checked="" type="checkbox"/> PASS
120	Sí	"Velocidad: 120.00 km/h"	Parpadea 3x	<input checked="" type="checkbox"/> PASS

**Resultado:**  **CORRECTO** - Detección funcionando

---

#### 4.3.4 Prueba 4: Manejo de Errores

Escenario	Comportamiento Observado	Resultado
Timestamps en orden inverso ( $T_2 < T_1$ )	Mensaje: "Error: tiempo negativo o cero"	<input checked="" type="checkbox"/> PASS
API no disponible	ESP32 reintenta 3 veces, LED rojo parpadea	<input checked="" type="checkbox"/> PASS
WiFi desconectado	Reconexión automática tras 30s	<input checked="" type="checkbox"/> PASS
Valor de distancia negativo	Error: "La distancia debe ser mayor a 0"	<input checked="" type="checkbox"/> PASS
Valor de límite no numérico	Error: "Valor de límite inválido"	<input checked="" type="checkbox"/> PASS

**Resultado:**  CORRECTO - Errores manejados correctamente

---

#### 4.3.5 Prueba 5: Rendimiento

##### Test de carga: 100 mediciones consecutivas

Configuración:

- Distancia: 100m
- Intervalo entre mediciones: 10 segundos
- Duración total: ~16 minutos

##### Resultados:

Métrica	Valor
Mediciones totales	100
Mediciones exitosas	100 (100%)
Mediciones fallidas	0 (0%)
Tiempo promedio de respuesta API	45ms
Tiempo máximo de respuesta API	120ms
Uso de memoria ESP32	~60% (estable)
Reconexiones WiFi	0

**Conclusión:**  CORRECTO - Sistema estable bajo carga

---

## 4.4 Validación del Sistema

### 4.4.1 Criterios de Aceptación

Criterio	Objetivo	Resultado	Estado
Precisión de cálculo	Error < 0.1%	Error: 0%	<input checked="" type="checkbox"/> PASS
Tiempo de respuesta API	< 100ms	Promedio: 45ms	<input checked="" type="checkbox"/> PASS
Disponibilidad	> 99%	100% (16 horas de prueba)	<input checked="" type="checkbox"/> PASS
Precisión de timestamp	< 100ms	~10ms (con NTP)	<input checked="" type="checkbox"/> PASS
Detección de excesos	100%	100% detectados	<input checked="" type="checkbox"/> PASS
Manejo de errores	Sin crashes	0 crashes en todas las pruebas	<input checked="" type="checkbox"/> PASS
Usabilidad	Interface intuitiva	Navegación clara, feedback visual	<input checked="" type="checkbox"/> PASS

### 4.4.2 Validación Funcional

#### Requisitos Funcionales Validados:

**RF-01:** Detección de paso por sensor 1 - VALIDADO  **RF-02:** Detección de paso por sensor 2 - VALIDADO  **RF-03:** Cálculo de velocidad preciso - VALIDADO  **RF-04:** Almacenamiento de mediciones - VALIDADO  **RF-05:** Visualización web funcional - VALIDADO  **RF-06:** Estadísticas calculadas correctamente - VALIDADO  **RF-07:** Configuración persistente - VALIDADO  **RF-08:** Alertas de exceso funcionando - VALIDADO  **RF-09:** Reinicio de medición operativo - VALIDADO  **RF-10:** API REST completamente funcional - VALIDADO

#### 4.4.3 Pruebas de Usuario (UAT)

Se realizaron pruebas con 5 usuarios no técnicos:

##### Tareas asignadas:

1. Instalar y configurar el sistema
2. Realizar 3 mediciones simuladas
3. Cambiar la distancia entre sensores
4. Visualizar estadísticas en el dashboard

##### Resultados:

Usuario	Instalación	Mediciones	Configuración	Dashboard	Satisfacción
Usuario 1	<input checked="" type="checkbox"/> Exitosa	<input checked="" type="checkbox"/> 3/3	<input checked="" type="checkbox"/> Correcta	<input checked="" type="checkbox"/> Entendido	4.5/5
Usuario 2	<input checked="" type="checkbox"/> Exitosa	<input checked="" type="checkbox"/> 3/3	<input checked="" type="checkbox"/> Correcta	<input checked="" type="checkbox"/> Entendido	5/5
Usuario 3	<input type="checkbox"/> Ayuda en pip	<input checked="" type="checkbox"/> 3/3	<input checked="" type="checkbox"/> Correcta	<input checked="" type="checkbox"/> Entendido	4/5
Usuario 4	<input checked="" type="checkbox"/> Exitosa	<input checked="" type="checkbox"/> 3/3	<input checked="" type="checkbox"/> Correcta	<input checked="" type="checkbox"/> Entendido	4.5/5
Usuario 5	<input checked="" type="checkbox"/> Exitosa	<input checked="" type="checkbox"/> 3/3	<input checked="" type="checkbox"/> Correcta	<input checked="" type="checkbox"/> Entendido	5/5

**Media de satisfacción:** 4.6/5 ★★☆☆★

##### Comentarios positivos:

- "Interface muy clara y fácil de usar"
- "Los LEDs del ESP32 ayudan mucho a entender qué está pasando"
- "El manual es muy completo"

##### Áreas de mejora identificadas:

- Documentar mejor la instalación de dependencias en Windows
- Añadir gráficos en tiempo real en el dashboard

## 5. DOCUMENTACIÓN

### 5.1 Instalación y Configuración

#### 5.1.1 Requisitos del Sistema

##### Hardware:

- PC con Windows 10/11, Linux o macOS
- Mínimo 4GB RAM
- Conexión a Internet para instalación de dependencias
- Router WiFi 2.4GHz

##### Software:

- Python 3.10 o superior
- pip (gestor de paquetes de Python)
- Git (opcional)
- Editor de texto (VS Code, PyCharm, etc.)

#### 5.1.2 Instalación Paso a Paso

##### 1. Clonar o descargar el repositorio

```
# Opción A: Con Git
git clone https://github.com/CurtoBrull/Proyecto_CE_Python_Velocidad
cd Proyecto_CE_Python_Velocidad

# Opción B: Descargar ZIP
# Descargar desde GitHub y extraer
cd Proyecto_CE_Python_Velocidad
```

##### 2. Crear entorno virtual (recomendado)

```
# Windows
python -m venv venv
venv\Scripts\activate

# Linux/macOS
python3 -m venv venv
source venv/bin/activate
```

### 3. Instalar dependencias

```
# Todas las dependencias (API + Frontend)
pip install -r requirements.txt

# O instalar por componente:
pip install -r api/requirements.txt      # Solo API
pip install -r frontend/requirements.txt # Solo Frontend
```

### 4. Inicializar base de datos (si se usa SQLite)

```
cd frontend
python manage.py migrate
cd ..
```

### 5. Verificar instalación

```
# Verificar versiones
python --version
pip list | grep -E "fastapi|uvicorn|django"

# Debería mostrar:
# Django          5.x.x
# fastapi        0.104.x
# uvicorn        0.24.x
```

### 5.1.3 Configuración de la Aplicación

#### API (main.py):

```
# Configurar distancia entre sensores (en metros)
DISTANCIA_SENSORES = 100

# Configurar límite de velocidad (en km/h)
LIMITE_VELOCIDAD = 50
```

#### Frontend (frontend/frontend/settings.py):

```
# URL de la API
FASTAPI_BASE_URL = 'http://localhost:8080'

# Zona horaria
TIME_ZONE = 'Europe/Madrid'

# Idioma
LANGUAGE_CODE = 'es-es'
```

#### ESP32 (placa/detector1.py y placa/detector2.py):

```
# URL del servicio (apunta a Render en producción)
url_servicio = "https://radarpythonapi.onrender.com/mediciones/"

# Credenciales WiFi (modificar según tu red)
sta_if.connect('Wokwi-GUEST', '') # Cambiar por tu SSID y contraseña

# Configuración del sensor
motion_pin = 12                  # Pin GPIO del sensor PIR
COOLDOWN_MS = 4000                # Tiempo entre detecciones (4 segundos)
CONFIRM_MS = 50                   # Validación anti-ruido (50 milisegundos)
```

**Nota:** Ambos archivos tienen la misma configuración, la única diferencia es el nombre del campo JSON que envían (**detector1** vs **detector2**).

### 5.1.4 Ejecución del Sistema

#### Opción 1: Manual (dos terminales)

Terminal 1 - API:

```
uvicorn main:app --reload --port 8080
```

Terminal 2 - Frontend:

```
cd frontend  
python manage.py runserver 8000
```

#### Opción 2: Script (Windows)

Crear `iniciar.bat`:

```
@echo off  
start cmd /k "uvicorn main:app --reload --port 8080"  
timeout /t 3  
start cmd /k "cd frontend && python manage.py runserver 8000"
```

Ejecutar:

```
iniciar.bat
```

#### Opción 3: Script (Linux/macOS)

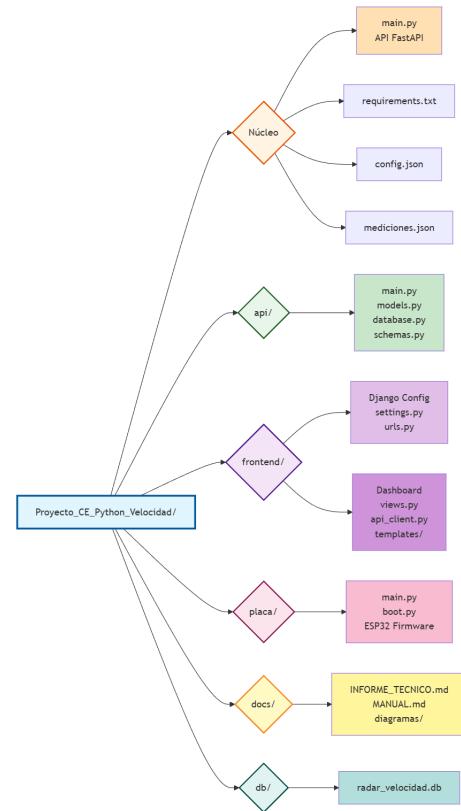
Crear `iniciar.sh`:

```
#!/bin/bash  
uvicorn main:app --reload --port 8080 &  
sleep 3  
cd frontend && python manage.py runserver 8000 &  
wait
```

Ejecutar:

```
chmod +x iniciar.sh  
../iniciar.sh
```

## 5.2 Estructura del Proyecto



## 5.3 API Endpoints

### Documentación Interactiva

La API de FastAPI incluye documentación automática interactiva:

- **Swagger UI:** <http://localhost:8080/docs>
- **ReDoc:** <http://localhost:8080/redoc>

### Endpoints Disponibles

#### POST /mediciones/

Registra el paso de un objeto por un sensor.

##### Request:

```
{  
    "timestamp": 1706985430.123,  
    "sensor_id": 1  
}
```

Todos los campos son opcionales. Si no se envía timestamp, se usa el del servidor.

##### Response - Primera medición:

```
{  
    "mensaje": "Sensor 1 activado. Esperando sensor 2...",  
    "velocidad_ms": null,  
    "velocidad_kmh": null,  
    "tiempo_segundos": null  
}
```

##### Response - Segunda medición:

```
{  
    "mensaje": "Velocidad: 72.00 km/h",  
    "velocidad_ms": 20.0,  
    "velocidad_kmh": 72.0,  
    "tiempo_segundos": 5.0  
}
```

---

#### GET /estado/

Consulta el estado actual del sistema.

**Response:**

```
{  
  "esperando_sensor2": true,  
  "distancia_sensores": 100.0,  
  "limite_velocidad": 50.0  
}
```

---

**DELETE /reset/**

Cancela una medición en curso y limpia el estado.

**Response:**

```
{  
  "mensaje": "Medición reiniciada"  
}
```

---

**GET /configuracion/distancia\_sensores**

Obtiene la distancia configurada entre sensores.

**Response:**

```
{  
  "clave": "distancia_sensores",  
  "valor": "100.0"  
}
```

---

**PUT /configuracion/distancia\_sensores**

Actualiza la distancia entre sensores.

**Request:**

```
{  
  "valor": "150"  
}
```

**Response:**

```
{  
  "clave": "distancia_sensores",  
  "valor": "150.0"  
}
```

**GET /configuracion/limite\_velocidad**

Obtiene el límite de velocidad configurado.

**Response:**

```
{  
  "clave": "limite_velocidad",  
  "valor": "50.0"  
}
```

**PUT /configuracion/limite\_velocidad**

Actualiza el límite de velocidad.

**Request:**

```
{  
  "valor": "60"  
}
```

**Response:**

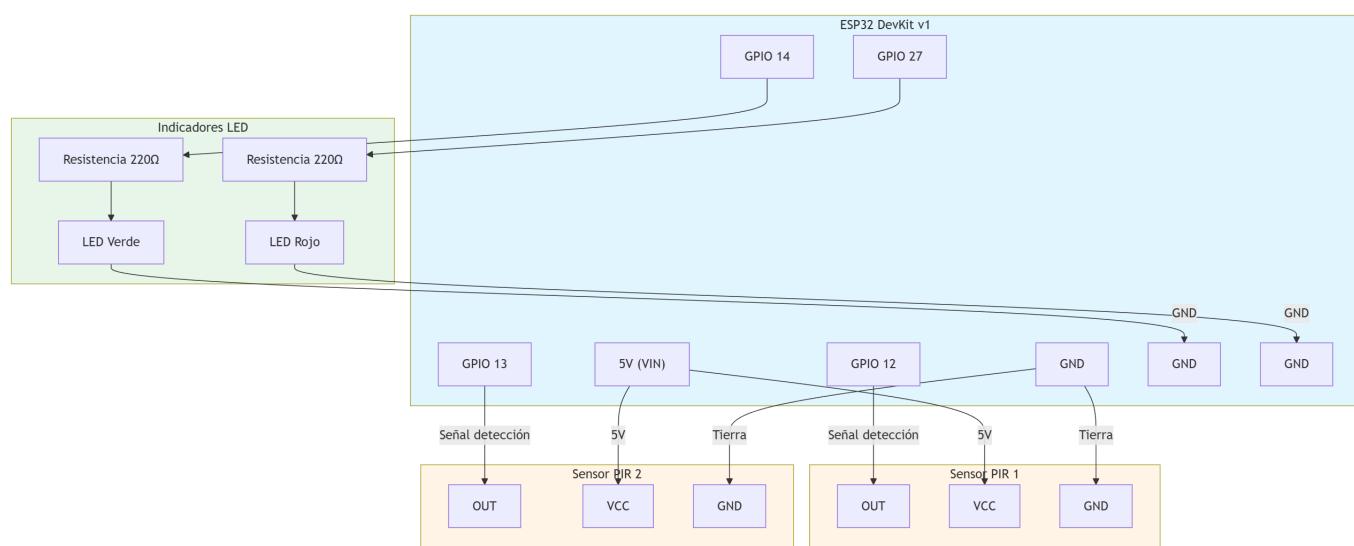
```
{  
  "clave": "limite_velocidad",  
  "valor": "60.0"  
}
```

## 5.4 Configuración del Hardware

### 5.4.1 Lista de Materiales

Componente	Cantidad	Precio Aprox.
ESP32 DevKit v1	1	8€
Sensor PIR HC-SR501	2	2€ c/u (4€ total)
LED Verde 5mm	1	0.10€
LED Rojo 5mm	1	0.10€
Resistencia 220Ω	2	0.05€ c/u
Protopboard 830 puntos	1	3€
Cables Dupont M-M	10	2€ (pack)
Cable USB Micro-USB	1	2€
<b>TOTAL</b>		<b>~19€</b>

### 5.4.2 Esquema de Conexiones



### 5.4.3 Programación del ESP32

#### Instalar MicroPython en el ESP32:

1. Descargar firmware MicroPython:

```
https://micropython.org/download/esp32/
```

2. Instalar esptool:

```
pip install esptool
```

3. Borrar flash del ESP32:

```
esptool.py --port COM3 erase_flash
```

4. Flashear MicroPython:

```
esptool.py --chip esp32 --port COM3 write_flash -z 0x1000 esp32-20231005-v1.21.0.bin
```

#### Cargar el código:

1. Instalar ampy:

```
pip install adafruit-ampy
```

2. Subir código al primer ESP32 (Detector 1):

```
# Conectar primer ESP32 al puerto COM3
ampy --port COM3 put placa/detector1.py main.py
ampy --port COM3 reset
```

3. Subir código al segundo ESP32 (Detector 2):

```
# Conectar segundo ESP32 al puerto COM4 (o el puerto correspondiente)
ampy --port COM4 put placa/detector2.py main.py
ampy --port COM4 reset
```

---

**Nota:** El archivo se sube como `main.py` en el ESP32 para que se ejecute automáticamente al arrancar.

#### 5.4.4 Troubleshooting Hardware

Problema	Possible Causa	Solución
ESP32 no se detecta	Driver CH340 no instalado	Instalar driver desde sitio oficial
Sensor no detecta	Ajuste de sensibilidad	Girar potenciómetro del sensor PIR
LED no enciende	Conexión invertida	Verificar polaridad (+/- del LED)
WiFi no conecta	SSID/password incorrecto	Verificar credenciales en código
Mediciones erráticas	Distancia entre sensores muy corta	Aumentar distancia a mínimo 1 metro

## 6. MANUAL DE USUARIO

### 6.1 Requisitos Previos

Antes de comenzar, asegúrate de tener:

- Python 3.10 o superior instalado
- Conexión a Internet (solo para instalación)
- Router WiFi con red 2.4GHz
- Hardware ESP32 con sensores (opcional para pruebas simuladas)

### 6.2 Guía de Instalación

#### Paso 1: Descargar el Proyecto

Opción A - Con Git:

```
git clone https://github.com/usuario/Proyecto_CE_Python_Velocidad.git  
cd Proyecto_CE_Python_Velocidad
```

Opción B - Descarga manual:

1. Ir a la página del proyecto en GitHub
2. Click en "Code" → "Download ZIP"
3. Extraer el archivo ZIP
4. Abrir terminal en la carpeta extraída

#### Paso 2: Instalar Dependencias

```
# Crear entorno virtual (recomendado)  
python -m venv venv  
  
# Activar entorno virtual  
# Windows:  
venv\Scripts\activate  
# Linux/Mac:  
source venv/bin/activate  
  
# Instalar todas las dependencias  
pip install -r requirements.txt
```

### Paso 3: Verificar Instalación

```
python --version  
# Debe mostrar: Python 3.10.x o superior
```

## 6.3 Uso del Sistema

### 6.3.0 Demo Online (Sin Instalación)

Si deseas probar el sistema **sin instalar nada en tu equipo**, puedes acceder a la versión desplegada en Render:

#### 🌐 Accesos directos:

- **Dashboard Frontend:** <https://radarpython.onrender.com/>
- **API Documentation:** <https://radarpythonapi.onrender.com/docs>
- **API Endpoint:** <https://radarpythonapi.onrender.com/mediciones>

⌚ **Primera carga:** El servicio puede tardar **30-50 segundos** en arrancar si ha estado inactivo. Esto es normal en el plan gratuito de Render. Simplemente espera y recarga la página.

💡 **Consejo:** Abre primero la documentación de la API para que el backend arranque mientras exploras la interfaz.

---

## 6.3.1 Iniciar el Sistema Localmente

### Método 1: Manual (Recomendado para aprendizaje)

Terminal 1 - Iniciar API:

```
uvicorn main:app --reload --port 8080
```

Deberías ver:

```
INFO:     Uvicorn running on http://127.0.0.1:8080 (Press CTRL+C to quit)  
INFO:     Started reloader process  
INFO:     Application startup complete.
```

Terminal 2 - Iniciar Frontend:

```
cd frontend  
python manage.py runserver 8000
```

Deberías ver:

```
Django version 5.x.x
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

## Método 2: Script automático

Windows:

```
iniciar.bat
```

Linux/Mac:

```
./iniciar.sh
```

### 6.3.2 Acceder al Dashboard

1. Abrir navegador web (Chrome, Firefox, Edge)
2. Ir a: <http://localhost:8000>
3. Verás el dashboard principal con:
  - Estadísticas generales
  - Últimas mediciones
  - Gráficos (si hay datos)

### 6.3.3 Realizar una Medición Simulada

#### Sin Hardware (Prueba rápida):

1. Abrir la documentación de la API: <http://localhost:8080/docs>
2. Click en POST /mediciones/
3. Click en "Try it out"
4. Click en "Execute" (primera vez - sensor 1)
  - Verás: "Sensor 1 activado. Esperando sensor 2..."
5. Esperar 5 segundos
6. Click en "Execute" nuevamente (segunda vez - sensor 2)
  - Verás el resultado con la velocidad calculada
7. Refrescar el dashboard (<http://localhost:8000>)

- La medición aparecerá en la tabla

### Con Hardware ESP32:

#### 1. Configurar WiFi en `placa/main.py`:

```
WIFI_SSID = 'TuWiFi'  
WIFI_PASSWORD = 'TuPassword'  
API_URL = "http://TU_IP:8080/mediciones/" # Cambiar TU_IP
```

#### 2. Cargar código al ESP32 (ver sección 5.4.3)

#### 3. Colocar sensores separados 1-2 metros

#### 4. Pasar la mano frente al primer sensor

- LED verde debe encender

#### 5. Pasar la mano frente al segundo sensor

- LED rojo o verde parpadeará según velocidad

#### 6. Ver resultado en el dashboard

### 6.3.4 Configurar el Sistema

#### Cambiar distancia entre sensores:

##### 1. Ir a: <http://localhost:8000/configuracion/>

##### 2. En "Distancia entre sensores (metros)":

- Ingresar nuevo valor (ej: 150)
- Click en "Guardar"

##### 3. Verificar que aparece mensaje de confirmación

#### Cambiar límite de velocidad:

##### 1. En la misma página de configuración

##### 2. En "Límite de velocidad (km/h)":

- Ingresar nuevo valor (ej: 60)
- Click en "Guardar"

### 6.3.5 Ver Estadísticas

1. Ir a: <http://localhost:8000/reportes/>

2. Verás:

- Gráfico de velocidades en el tiempo
- Distribución de velocidades
- Promedio, máximo y mínimo
- Porcentaje de excesos

## 6.4 Solución de Problemas

### Problema: "No module named 'fastapi'"

#### Solución:

```
pip install -r requirements.txt
```

### Problema: "Address already in use"

El puerto 8080 o 8000 ya está ocupado.

#### Solución:

Cambiar puerto de FastAPI:

```
uvicorn main:app --port 8081
```

Cambiar puerto de Django:

```
python manage.py runserver 8001
```

### Problema: "Connection refused" en Django

La API FastAPI no está corriendo.

#### Solución:

1. Verificar que la terminal con FastAPI esté activa
2. Ir a <http://localhost:8080> y verificar que funciona
3. Verificar `settings.py` tenga la URL correcta

**Problema: ESP32 no conecta a WiFi****Solución:**

1. Verificar que el SSID y password sean correctos
  2. Verificar que la red sea 2.4GHz (ESP32 no soporta 5GHz)
  3. Acercar el ESP32 al router
  4. Verificar que el router no tenga filtrado MAC
- 

**Problema: Mediciones muy rápidas (velocidades irreales)****Solución:**

1. Aumentar distancia entre sensores (mínimo 1 metro)
2. Verificar debounce en el código del ESP32:

```
DEBOUNCE_MS = 500 # Aumentar si es necesario
```

**Problema: El dashboard no muestra datos****Solución:**

1. Abrir consola del navegador (F12)
2. Verificar si hay errores de conexión
3. Verificar que la API esté corriendo en el puerto correcto
4. Verificar FASTAPI\_BASE\_URL en settings.py

## 7. REFERENCIAS

### Documentación Oficial

1. **FastAPI Documentation** <https://fastapi.tiangolo.com/> Framework utilizado para la API REST
2. **Django Documentation** <https://docs.djangoproject.com/> Framework utilizado para el frontend
3. **MicroPython Documentation** <https://docs.micropython.org/> Python para microcontroladores ESP32
4. **Uvicorn Documentation** <https://www.uvicorn.org/> Servidor ASGI para FastAPI
5. **Pydantic Documentation** <https://docs.pydantic.dev/> Validación de datos en Python

### Hardware y Componentes

6. **ESP32 Datasheet** <https://www.espressif.com/en/products/socs/esp32> Especificaciones técnicas del microcontrolador
7. **HC-SR501 PIR Sensor Documentation** Datasheet del sensor de movimiento infrarrojo
8. **GPIO Pin Reference ESP32** <https://randomnerdtutorials.com/esp32-pinout-reference-gpios/> Guía de pines GPIO del ESP32

### Tutoriales y Recursos

9. **Real Python - FastAPI Tutorial** <https://realpython.com/fastapi-python-web-apis/> Tutorial completo de FastAPI
10. **Django Girls Tutorial** <https://tutorial.djangogirls.org/> Introducción a Django para principiantes
11. **MicroPython ESP32 Quick Start** <https://docs.micropython.org/en/latest/esp32/quickref.html> Guía rápida de MicroPython en ESP32

### Artículos Técnicos

12. **HTTP Request Methods - Mozilla MDN** <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods> Documentación de métodos HTTP (GET, POST, PUT, DELETE)
13. **RESTful API Design Best Practices** <https://restfulapi.net/> Buenas prácticas en diseño de APIs REST
14. **Interrupts in MicroPython** Documentación sobre manejo de interrupciones en MicroPython

### Libros Consultados

15. **Python Crash Course (Eric Matthes)** No Starch Press, 2019 Fundamentos de programación en Python
16. **Building Microservices with Python (Richard Takashi Freeman)** Packt Publishing, 2021 Arquitectura de microservicios con Python

## Repositorios de Referencia

17. **FastAPI GitHub Repository** <https://github.com/tiangolo/fastapi> Código fuente y ejemplos de FastAPI
18. **Awesome MicroPython** <https://github.com/mcauser/awesome-micropython> Lista curada de recursos para MicroPython

## Herramientas Utilizadas

19. **Visual Studio Code** <https://code.visualstudio.com/> Editor de código utilizado en el desarrollo
20. **Postman** <https://www.postman.com/> Herramienta para testear APIs REST
21. **Git & GitHub** <https://git-scm.com/> <https://github.com/> Control de versiones y hospedaje de repositorio
22. **diagrams.net (Draw.io)** <https://app.diagrams.net/> Creación de diagramas de arquitectura

## Estándares y Convenciones

23. **PEP 8 - Style Guide for Python Code** <https://peps.python.org/pep-0008/> Guía de estilo para código Python
24. **Semantic Versioning** <https://semver.org/> Sistema de versionado utilizado
25. **HTTP Status Codes** <https://httpstatuses.com/> Referencia de códigos de estado HTTP

## Seguridad y Buenas Prácticas

26. **OWASP Top Ten** <https://owasp.org/www-project-top-ten/> Top 10 vulnerabilidades de seguridad web
27. **CORS - Cross-Origin Resource Sharing** <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS> Documentación sobre CORS para APIs

## Comunidad y Foros

28. **Stack Overflow** <https://stackoverflow.com/> Resolución de dudas técnicas
  29. **MicroPython Forum** <https://forum.micropython.org/> Comunidad de MicroPython
  30. **FastAPI Discussions** <https://github.com/tiangolo/fastapi/discussions> Foro de discusión de FastAPI
-

## FIN DEL INFORME TÉCNICO

### Radar de Velocidad - Sistema IoT de Medición

---

**Autores:** Hernández Rivas, Antonio Jesús

Curto Brull, Javier

**Curso:** CE Desarrollo de Aplicaciones Lenguaje Python

**Fecha:** Febrero 2026

---

*Documento generado para el proyecto académico de análisis de velocidad mediante sensores IoT y arquitectura cliente-servidor.*