UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA

Escuela Profesional de Ciencia de la Computación

Sistema IoT híbrido para la detección de incendios mediante sensores ambientales y análisis multimedia

---

Curso: Internet de las Cosas

Integrantes:

Wilson Ramos Pacco
Owen Haziel Roque Sosa
Javier Wilber Quispe Rojas
Patrik Rene Ramirez Zarate
Evelyn Lizbeth Cusi Hancco
Angela Solange Sucso Choque

Arequipa, Perú

19 de diciembre de 2025

1. Resumen

En el presente proyecto se desarrolla un sistema IoT híbrido para la detección temprana de incendios, combinando sensores físicos y datos multimedia capturados por un dispositivo móvil. El sistema está basado en un Arduino MKR1010 WiFi con el IoT Carrier, el cual realiza lecturas continuas de sensores de temperatura y luz, utilizando la humedad como variable complementaria. Estas mediciones son comparadas con umbrales configurables para determinar el estado del sistema, el cual es Normal en primera instancia.

Cuando los valores de los sensores superan los umbrales establecidos, el Arduino envía un evento de alerta a un servidor backend mediante el protocolo MQTT. Cambiando de estado Normal a un estado de Riesgo. A partir de esto, el servidor se comunica con un smartphone utilizando la aplicación IP Webcam, solicitando automáticamente la captura de una imagen y la grabación de un breve audio. Estos archivos son enviados al servidor para su posterior análisis.

La imagen capturada es procesada mediante un modelo de detección basado en YOLO para identificar la presencia de fuego, mientras que el audio es analizado utilizando un modelo de aprendizaje automático basado en árboles de decisión, con el objetivo de detectar sonidos característicos de un posible incendio. Los resultados de ambos análisis son fusionados mediante un esquema de ponderación para confirmar o descartar la presencia de un incendio, permitiendo pasar del estado de Riesgo a Confirmado.

Finalmente, toda la información relevante es mostrada en un dashboard en tiempo real, incluyendo las lecturas de sensores, el estado del sistema, la última imagen y el audio capturado. En caso de confirmarse un incendio, el sistema envía una alerta automática a través de un bot de Telegram, notificando al usuario sobre la situación de peligro.

2. Arquitectura del Sistema

La arquitectura del sistema define la organización general de los componentes que conforman la solución IoT propuesta, así como la forma en que estos interactúan entre sí. El sistema ha sido diseñado de manera modular, permitiendo la integración de sensores físicos, un servidor backend, un dispositivo móvil y módulos de análisis inteligente, con el objetivo de detectar y confirmar la presencia de incendios de forma eficiente.

2.1. Diagrama de bloques

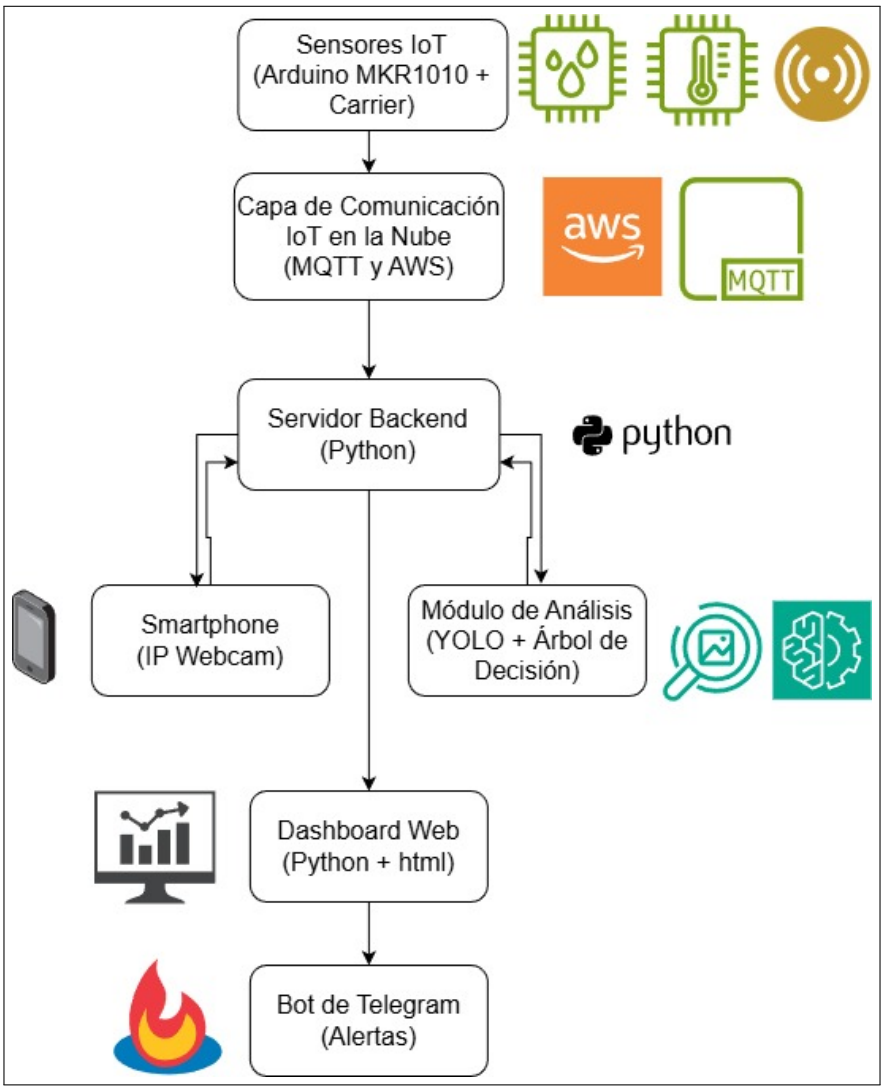

Figura 1: Arquitectura completa del sistema de detección de incendios

El sistema está compuesto por varios módulos interconectados que permiten la detección y confirmación de incendios. En primer lugar, el módulo IoT, basado en un Arduino MKR1010 WiFi con el IoT Carrier, se encarga de la adquisición de datos ambientales mediante sensores de temperatura, luz y humedad.

La transmisión de estos datos se realiza a través de una capa de comunicación IoT en la nube, basada en el protocolo MQTT e integrada con servicios de AWS, la cual permite una comunicación segura y confiable con el servidor backend.

Los datos son recibidos por un servidor backend desarrollado en Python, el cual actúa como núcleo del sistema, evaluando los valores recibidos y comparándolos con umbrales predefinidos. Cuando se detecta una condición de riesgo, el servidor se comunica con un smartphone que funciona como sensor multimedia, solicitando la captura de una imagen y la grabación de audio.

Los archivos multimedia obtenidos son procesados por un módulo de análisis que utiliza técnicas de visión por computadora y aprendizaje automático para confirmar la presencia de fuego. Finalmente, los resultados son visualizados en un dashboard web en tiempo real y, en caso de confirmarse un incendio, se envía una alerta al usuario mediante un bot de Telegram.

2.2. Diagrama de Arquitectura AWS

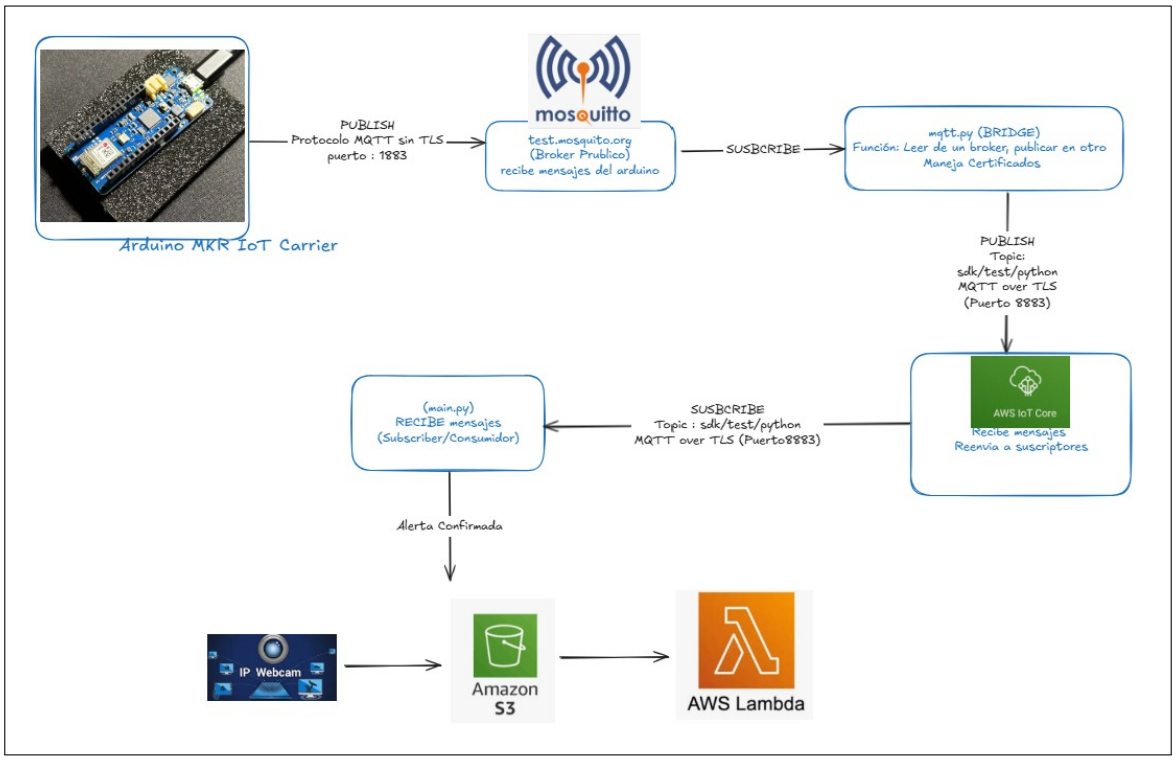

Figura 2: Arquitectura completa del sistema de detección de incendios

3. Componentes Principales

3.1. Capa de Sensores (Edge Device)

Arduino MKR IoT Carrier

[leftmargin=*]

- **Función:** Dispositivo de borde que recolecta datos de sensores ambientales (temperatura, humedad, luminosidad).
- **Protocolo:** Publica mensajes mediante MQTT sin TLS al broker público Mosquitto (puerto 1883).
- **Ubicación:** Entorno físico donde se monitorea riesgo de incendio.

3.2. Capa de Comunicación

3.2.1. Broker MQTT Público

test.mosquitto.org

[leftmargin=*]

- **Función:** Broker MQTT público que recibe mensajes del Arduino.
- **Características:** Sin autenticación, sin cifrado, ideal para prototipado.
- **Puerto:** 1883 (MQTT sin TLS).

3.2.2. Puente MQTT (mqtt.py - BRIDGE)

mqtt.py

[leftmargin=*]

- **Función:** Función puente que lee mensajes del broker público y los reenvía a AWS IoT Core con autenticación segura.
- **Características:**
  - Suscriptor del broker Mosquitto (topic: sdk/test/python)
  - Publicador en AWS IoT Core con certificados X.509
  - Maneja conversión de protocolo inseguro a seguro
- **Seguridad:** Utiliza certificados TLS para comunicación con AWS.

3.2.3. AWS IoT Core

AWS IoT Core

[leftmargin=*]

- **Función:** Broker MQTT administrado en la nube con autenticación mediante certificados X.509.
- **Características:**
  - Puerto 8883 (MQTT sobre TLS)
  - Políticas de autorización (IoT Policies)
  - Escalabilidad automática
  - Enrutamiento de mensajes a múltiples suscriptores
- **Endpoint:** a1b9nxragudit3-ats.iot.us-east-1.amazonaws.com

3.3. Capa de Procesamiento Local

main.py (Subscriber/Consumidor)

[leftmargin=*]

- **Función:** Aplicación Python local que se suscribe a AWS IoT Core y procesa datos de sensores en tiempo real.
- **Responsabilidades:**
  - Análisis de umbrales (temperatura ¿45°C, luminosidad ¿2000 lux)
  - Implementación de lógica de persistencia (5 lecturas consecutivas altas)
  - Captura de evidencia multimedia (foto + audio) cuando se confirma alerta
  - Subida de evidencias a Amazon S3
  - Invocación de Lambda para análisis con IA

3.4. Capa de Almacenamiento

Amazon S3

[leftmargin=*]

- **Función:** Almacenamiento de objetos escalable para evidencias multimedia.
- **Estructura:**
  - `/fotos/YYYY-MM-DD/foto_timestamp.jpg` - Imágenes capturadas

- `/audios/YYYY-MM-DD/audio_timestamp.wav` - Grabaciones de audio
- `/modelos/modelo_incendio.pkl` - Modelo ML entrenado
- `/lambda-layers/audio-layer.zip` - Dependencias para Lambda

- **Bucket:** incendios-multimedia-227338491492

3.5. Capa de Inteligencia Artificial

AWS Lambda: AnalizadorAudioML

[leftmargin=*]

- **Función:** Función serverless que analiza audio capturado mediante Machine Learning.
- **Configuración:**
  - Runtime: Python 3.11
  - Memoria: 1024 MB
  - Timeout: 60 segundos
  - Layer: audio-ml-dependencies (librosa, numpy, scipy, scikit-learn)
- **Proceso de análisis:**

  [leftmargin=*]

  1. Descarga archivo de audio desde S3
  2. Carga modelo Random Forest entrenado (cache en /tmp)
  3. Extrae 17 características acústicas:
     - 13 coeficientes MFCC (Mel-Frequency Cepstral Coefficients)
     - Spectral Centroid
     - Zero Crossing Rate
     - RMS Energy
     - Spectral Rolloff
  4. Realiza predicción binaria (incendio/no incendio)
  5. Calcula confianza mediante probabilidades del modelo
  6. Guarda resultados en DynamoDB
- **Output:** Confianza de detección (0-100 %) y clasificación binaria

Base de Datos: Amazon DynamoDB

[leftmargin=*]

- **Función:** Base de datos NoSQL para almacenar registros de eventos de incendio.

- **Tabla:** IncendiosEventos
- **Estructura:**
  - `evento_id` (Partition Key): Identificador único del evento
  - `timestamp`: Fecha y hora de detección
  - `foto_s3`: URI de la imagen en S3
  - `audio_s3`: URI del audio en S3
  - `ia_audio`: Confianza del análisis de audio
  - `confianza_audio`: Valor numérico de confianza
  - `audio_analizado`: Flag booleano

4. Flujo de Datos Completo

[leftmargin=*]

1. **Captura:** Arduino lee sensores cada segundo y publica JSON vía MQTT a Mosquitto.
2. **Puente:** mqtt.py reenvía mensajes a AWS IoT Core con autenticación TLS.
3. **Distribución:** AWS IoT Core distribuye mensajes a todos los suscriptores autorizados.
4. **Análisis:** main.py evalúa umbrales y cuenta lecturas consecutivas altas.
5. **Alerta:** Tras 5 lecturas confirmadas, captura foto (IP Webcam) y audio (5 segundos).
6. **Almacenamiento:** Evidencias se suben a S3 con estructura jerárquica por fecha.
7. **IA:** Lambda es invocada asíncronamente para analizar audio con modelo ML.
8. **Persistencia:** Resultados del análisis se guardan en DynamoDB para consultas futuras.

5. Integración de Machine Learning en Lambda

5.1. Integración de Machine Learning en Lambda

Esta implementacion consistió en la implementación completa del análisis de audio mediante Machine Learning en AWS Lambda:

5.1.1. Creación del Lambda Layer

[leftmargin=*]

- **Componente:** Lambda Layer `audio-ml-dependencies`
- **Contenido:** Librerías Python para procesamiento de audio y ML:
  - librosa 0.10.1 - Extracción de características de audio
  - numpy 1.24.3 - Operaciones numéricas
  - scipy 1.10.1 - Procesamiento de señales
  - scikit-learn 1.3.0 - Modelo de clasificación
- **Desafío resuelto:** Incompatibilidad de binarios Windows/Linux. Se requirió reinstalar con flags específicos para la plataforma manylinux2014_x86_64.
- **Tamaño:** ∼150 MB comprimido, ∼220 MB descomprimido.

5.1.2. Función Lambda AnalizadorAudioML

[leftmargin=*]

- **Handler:** `lambda_handler(event, context)`
- **Input esperado:**

```
{
  "evento_id": "incendio_1766090000",
  "audio_s3": "s3://bucket/audios/audio.mp3"
}
```

- **Proceso optimizado:**
  - Caché del modelo en memoria (variable global MODELO)
  - Primera invocación: 25-40 segundos (descarga modelo)
  - Invocaciones subsecuentes: 3-5 segundos (modelo en RAM)
- **Formato soportado:** WAV, MP3, FLAC, OGG (detección automática vía librosa)

5.1.3. Permisos IAM

Se configuró el Execution Role con políticas:

[leftmargin=*]

- `AmazonS3ReadOnlyAccess` - Lectura de audio y modelo desde S3
- `AmazonDynamoDBFullAccess` - Escritura de resultados en DynamoDB
- `AWSLambdaBasicExecutionRole` - Logging en CloudWatch

6. Características Técnicas Clave

6.1. Seguridad

[leftmargin=*]

- **Autenticación:** Certificados X.509 para conexión a AWS IoT Core
- **Cifrado:** TLS 1.2+ en toda comunicación con AWS
- **Autorización:** IoT Policies que restringen acceso por CLIENT_ID y topics
- **Aislamiento:** Lambda ejecuta en entorno efímero (/tmp) que se destruye tras invocación

6.2. Escalabilidad

[leftmargin=*]

- **AWS IoT Core:** Soporta millones de dispositivos conectados simultáneamente
- **Lambda:** Escalado automático, puede procesar miles de audios en paralelo
- **S3:** Almacenamiento ilimitado con 99.999999999 % de durabilidad
- **DynamoDB:** Throughput ajustable dinámicamente según carga

6.3. Costo

[leftmargin=*]

- **Modelo:** Pay-per-use (pago solo por recursos consumidos)
- **Estimación mensual:** $10-15 para 10,000 eventos/mes
- **Free Tier:** Primer año incluye 750 horas de Lambda gratuitas

6.4. Flujo de Comunicación

1. El Arduino MKR1010 WiFi realiza lecturas continuas de los sensores de temperatura, luz y humedad integrados en el IoT Carrier.
2. Los valores obtenidos por los sensores son enviados periódicamente al servidor backend mediante el protocolo MQTT.
3. El servidor backend recibe las lecturas y las compara con umbrales previamente configurados para cada sensor.

4. Si los valores se mantienen dentro de los rangos normales, el sistema permanece en estado Normal y continúa el monitoreo.
5. Cuando uno o más sensores superan los umbrales establecidos, el servidor cambia el estado del sistema a Riesgo y genera un evento de alerta.
6. Ante el estado de Riesgo, el servidor se comunica con el smartphone que actúa como sensor multimedia, solicitando la captura automática de una imagen y la grabación de un audio de corta duración.
7. El smartphone, mediante la aplicación IP Webcam, captura la imagen y el audio, y envía ambos archivos al servidor backend.
8. El servidor recibe y almacena los archivos multimedia para su posterior procesamiento.
9. La imagen capturada es analizada utilizando un modelo de detección basado en YOLO, con el objetivo de identificar la presencia de fuego.
10. El audio capturado es analizado mediante un modelo de aprendizaje automático basado en árboles de decisión, para detectar sonidos característicos de un posible incendio.
11. Los resultados del análisis de imagen y audio son combinados mediante un esquema de ponderación, permitiendo confirmar o descartar la presencia de un incendio.
12. Si el resultado del análisis confirma la presencia de fuego, el estado del sistema cambia a Confirmado.
13. El servidor actualiza el dashboard web en tiempo real, mostrando las lecturas de los sensores, el estado del sistema y los archivos multimedia capturados.
14. Finalmente, en caso de incendio confirmado, el servidor envía una notificación automática al usuario a través de un bot de Telegram.

7. Hardware Utilizado

7.1. Arduino MKR1010 WiFi

El Arduino MKR1010 WiFi es el componente central del sistema IoT, encargado de la adquisición de datos de los sensores y de la transmisión de información al servidor backend. Tiene un bajo consumo energético, conectividad WiFi integrada y compatibilidad con el IoT Carrier, lo que permite una integración sencilla de múltiples sensores en una sola plataforma.

Especificaciones técnicas

- Microcontrolador: SAMD21 Cortex-M0+ (32 bits)
- Frecuencia de reloj: 48 MHz

- Memoria Flash: 256 KB
- Memoria SRAM: 32 KB
- Conectividad WiFi: U-BLOX NINA-W102 (IEEE 802.11 b/g/n)
- Seguridad: Chip criptográfico ECC608

7.2. Arduino IoT Carrier

El Arduino IoT Carrier es utilizado como módulo de expansión para el Arduino MKR1010 WiFi, proporcionando sensores ambientales integrados que permiten la adquisición de variables relevantes para la detección de incendios.

Entre los sensores utilizados se encuentran el sensor de temperatura, el sensor de humedad y el sensor de luz RGB, los cuales permiten identificar condiciones anómalas como aumentos de temperatura y niveles elevados de luminosidad asociados a la presencia de fuego.

Sensores utilizados

- Sensor de temperatura
- Sensor de luz (RGB + Clear)
- Sensor de humedad

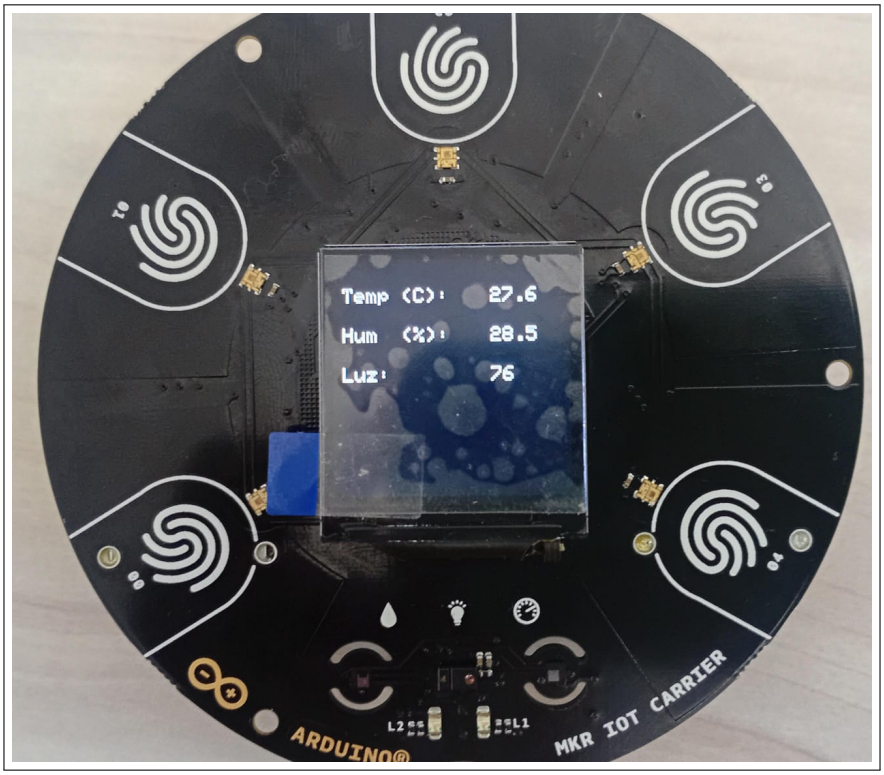

Figura 3: Sensores capturando datos

7.3. Smartphone

Se emplea un smartphone con sistema operativo Android, compatible con la aplicación IP Webcam, el cual actúa como sensor multimedia para la captura de imágenes y audio. Esta solución permite aprovechar hardware ya disponible, reduciendo costos y facilitando la integración de la cámara y el micrófono en el sistema.

7.4. Batería

Para alimentar el Arduino MKR1010 WiFi se utiliza una batería de ion-litio PKCELL ICR18650, la cual permite que el dispositivo funcione sin necesidad de estar conectado por USB. Esto facilita el uso del sistema de manera inalámbrica, aprovechando la conexión WiFi del Arduino.

Figura 4: Bateria reemplaza a USB

El uso de la batería es importante ya que permite mayor libertad en la ubicación del dispositivo y evita depender de una fuente de alimentación fija. De esta forma, el sistema puede operar de manera más flexible en escenarios reales donde se requiere monitoreo continuo para la detección temprana de incendios.

Sistema de alimentación

- Tipo: Batería Li-ion PKCELL ICR18650
- Capacidad: 2200 mAh
- Voltaje nominal: 3.7 V
- Energía: 8.14 Wh

8. Código fuente explicado

8.1. Código del Arduino

El código del Arduino MKR1010 WiFi es el responsable de la captura de datos ambientales y su transmisión mediante MQTT. A continuación se explica cada componente:

8.1.1. Librerías utilizadas

Listing 1: Librerías del Arduino

```
#include <Arduino_MKRIoTCarrier.h>
#include <WiFiNINA.h>
#include <ArduinoMqttClient.h>
```

- **Arduino_MKRIoTCarrier.h**: Proporciona acceso a los sensores integrados del IoT Carrier (temperatura, humedad, luz RGB)
- **WiFiNINA.h**: Maneja la conectividad WiFi del módulo U-BLOX NINA-W102
- **ArduinoMqttClient.h**: Implementa el protocolo MQTT para la publicación de mensajes

8.1.2. Configuración de conexión

Listing 2: Configuración WiFi y MQTT

```
char ssid[] = wifi-CsComputacion ;
char pass[] = EPCC2022$ ;
const char broker[] = test.mosquitto.org ;
const char topic[] = incendio/sensores ;

WiFiClient wifiClient;
MqttClient mqttClient(wifiClient);
```

El sistema se conecta a la red WiFi institucional y publica mensajes en el broker público Mosquitto, específicamente en el topic `incendio/sensores`. Esta configuración permite que el puente MQTT (`mqtt.py`) reciba los datos y los reenvíe a AWS IoT Core con autenticación segura.

8.1.3. Función setup()

Listing 3: Inicialización del sistema

```
void setup() {
  Serial.begin(9600);
  delay(2000);
  carrier.begin();

  // Conectar WiFi
  WiFi.begin(ssid, pass);
  unsigned long t0 = millis();
  while (WiFi.status() != WL_CONNECTED && millis() - t0 < 15000) {
    delay(500);
  }

  // Conectar MQTT
  mqttClient.connect(broker, 1883);
}
```

La función `setup()` realiza las siguientes operaciones:

1. Inicializa la comunicación serie a 9600 baudios para depuración
2. Inicializa el IoT Carrier y sus sensores
3. Establece conexión WiFi con timeout de 15 segundos
4. Conecta al broker MQTT en el puerto 1883 (sin TLS)

8.1.4. Función loop()

Listing 4: Lectura y publicación de sensores

```
void loop() {
  float temp = carrier.Env.readTemperature();
  float hum  = carrier.Env.readHumidity();

  int r, g, b, c;
  if (carrier.Light.colorAvailable()) {
    carrier.Light.readColor(r, g, b, c);

    mqttClient.beginMessage(topic);
    mqttClient.print("{");
    mqttClient.print("\"temp\":"); mqttClient.print(temp);
    mqttClient.print(",");
    mqttClient.print("\"hum\":");  mqttClient.print(hum);
    mqttClient.print(",");
    mqttClient.print("\"luz\":");  mqttClient.print(c);
```

```
    mqttClient.print( } );
    mqttClient.endMessage();
  }

  delay(1000);
}
```

El bucle principal ejecuta las siguientes acciones cada segundo:

1. **Lectura de sensores ambientales**:
   - Temperatura en grados Celsius
   - Humedad relativa en porcentaje
   - Intensidad luminosa (canal claro del sensor RGB)
2. **Construcción del mensaje JSON**:
   ```
   {
     temp : 25.3,
     hum : 45.2,
     luz : 1500
   }
   ```
3. **Publicación MQTT**: El mensaje se envía al topic `incendio/sensores` del broker Mosquitto
4. **Delay de 1 segundo**: Controla la frecuencia de muestreo
5. **Filtrado de lecturas inválidas**: Se descartan valores nulos provenientes de los sensores y se conserva el último valor válido leído, evitando el envío de datos erróneos al servidor.

8.1.5. Visualización local en el IoT Carrier

El firmware implementa una visualización local de las variables ambientales utilizando la pantalla integrada del IoT Carrier. En la función `setup()` se inicializa la pantalla y se muestran etiquetas fijas para temperatura, humedad e intensidad luminosa.

Durante la ejecución del bucle principal, únicamente se actualizan los valores numéricos en pantalla, evitando el refresco completo de la interfaz. Esta estrategia reduce parpadeos y mejora la legibilidad de la información mostrada.

8.1.6. Características técnicas del código Arduino

- **Frecuencia de muestreo**: 1 Hz (una lectura por segundo)
- **Formato de datos**: JSON sin comprimir

- **Protocolo de transporte**: MQTT sobre TCP (puerto 1883)
- **QoS MQTT**: QoS 0 (at most once) - por defecto en ArduinoMqttClient
- **Reconexión automática**: No implementada (requiere reinicio manual si se pierde conexión)
- **Filtrado de datos**: Se mantiene el último valor válido cuando se detectan lecturas nulas (0).
- **Visualización local**: Datos ambientales mostrados en la pantalla del IoT Carrier en tiempo real.

8.2. Código del servidor backend

El servidor backend (`main.py`) es el núcleo del sistema, responsable de recibir datos de sensores, evaluar condiciones de riesgo, capturar evidencia multimedia, ejecutar análisis con IA y gestionar el flujo de estados.

8.2.1. Arquitectura del backend

El backend está estructurado en los siguientes módulos funcionales:

1. **Conexión AWS IoT Core**: Recepción de mensajes MQTT con certificados X.509
2. **Máquina de estados**: Gestión de transiciones Normal → Riesgo → Confirmado
3. **Captura de evidencia**: Integración con IP Webcam para foto y audio
4. **Análisis con IA**: Detección mediante YOLOv8 y Random Forest
5. **API REST**: Comunicación con dashboard web
6. **Notificaciones**: Alertas vía Telegram
7. **Almacenamiento cloud**: Subida de evidencias a Amazon S3

8.2.2. Configuración de AWS IoT Core

Listing 5: Configuración de credenciales AWS

```
ENDPOINT = a1b9nxragudit3-ats.iot.us-east-1.amazonaws.com
CLIENT_ID = main-fire-detector
PATH_TO_CERTIFICATE = ./arduino-incendio.cert.pem
PATH_TO_PRIVATE_KEY = ./arduino-incendio.private.key
PATH_TO_AMAZON_ROOT_CA_1 = ./root-CA.crt
TOPIC_SENSORES = sdk/test/python
```

La conexión se establece mediante autenticación mutual TLS (mTLS) con certificados X.509:

Listing 6: Establecimiento de conexión MQTT

```
mqtt_connection = mqtt_connection_builder.mtls_from_path(
    endpoint=ENDPOINT,
    cert_filepath=PATH_TO_CERTIFICATE,
    pri_key_filepath=PATH_TO_PRIVATE_KEY,
    ca_filepath=PATH_TO_AMAZON_ROOT_CA_1,
    client_id=CLIENT_ID,
    clean_session=False,
    keep_alive_secs=30
)

connect_future = mqtt_connection.connect()
connect_future.result()
```

8.2.3. Parámetros de detección

Listing 7: Umbrales y configuración

```
# Umbrales de sensores
TEMP_UMBRAL = 45          # Temperatura en  C
LUZ_UMBRAL = 2000         # Luminosidad en lux
HUMEDAD_MIN = 20          # Humedad relativa en %

# Persistencia de estados
LECTURAS_RIESGO = 5       # Lecturas consecutivas para cambiar a Riesgo
LECTURAS_RECUPERACION = 5  # Lecturas normales para volver a Normal

# Fusi n de probabilidades IA
PESO_IMAGEN = 0.9         # Peso del an lisis YOLO
PESO_AUDIO = 0.1          # Peso del an lisis de audio
UMBRAL_ALERTA = 0.6       # Umbral de decisi n final
```

Estos parámetros fueron ajustados empíricamente durante pruebas de campo. El peso de 0.9 para la imagen refleja que la detección visual de fuego es más confiable que la acústica en entornos con ruido ambiental.

8.2.4. Máquina de estados

El sistema implementa una máquina de estados finita con tres estados principales:

Listing 8: Variables de estado

```
estado_local =  Normal
contador_riesgo = 0
contador_normal = 0
```

```
evidencia_tomada = False
alerta_enviada = False
```

Estado Normal:

Listing 9: Lógica del estado Normal

```
if estado_local == Normal :
    enviar_estado( normal )

    if condicion_riesgo:
        contador_riesgo += 1
        if contador_riesgo >= LECTURAS_RIESGO:
            estado_local = Riesgo
            evidencia_tomada = False
            contador_normal = 0
    else:
        contador_riesgo = 0
```

En este estado, el sistema monitorea continuamente los sensores. Solo transiciona a **Riesgo** cuando se detectan 5 lecturas consecutivas que superan los umbrales, evitando falsas alarmas por picos transitorios.

Estado Riesgo:

Listing 10: Captura de evidencia en estado Riesgo

```
elif estado_local == Riesgo :
    enviar_estado( riesgo )

    if not evidencia_tomada:
        foto_ok = tomar_foto()
        audio_ok = grabar_audio(5)
        evidencia_tomada = True

        if foto_ok and audio_ok:
            enviar_imagen(PHOTO_PATH)
            enviar_audio(AUDIO_PATH)

            resultado_audio = detectar_incendio(AUDIO_PATH)
            resultado_imagen = detectar_incendio_imagen(PHOTO_PATH)

            prob_audio = resultado_audio[ confianza ]
            prob_imagen = resultado_imagen[ confianza ]

            alerta_final, score = decidir_alerta(prob_imagen,
            prob_audio)

            if alerta_final:
                estado_local = Confirmado
```

En este estado crítico, el sistema:

1. Captura foto y audio (solo una vez)
2. Envía evidencia al dashboard
3. Analiza con modelos de IA
4. Fusiona probabilidades
5. Decide si confirmar el incendio

Estado Confirmado:

Listing 11: Acciones en estado Confirmado

```
elif estado_local == Confirmado :
    enviar_estado( incendio_confirmado )

    if not alerta_enviada:
        enviar_alerta_telegram(PHOTO_PATH)
        alerta_enviada = True

        evento_id = f incendio_{int(time.time())}
        foto_s3 = subir_a_s3(PHOTO_PATH, fotos )
        audio_s3 = subir_a_s3(AUDIO_PATH, audios )
```

En este estado terminal, el sistema ejecuta las acciones de respuesta:

- Envía alerta a Telegram con foto del incendio
- Sube evidencias a Amazon S3 para registro permanente
- Mantiene el estado hasta intervención manual

8.3. Procesamiento de datos multimedia

8.3.1. Captura de imagen

La captura de imagen se realiza mediante la aplicación IP Webcam instalada en un smartphone Android:

Listing 12: Función de captura de foto

```
def tomar_foto():
    try:
        img = requests.get(f {CAMERA_URL}/shot.jpg , timeout=5).
        content
        with open(PHOTO_PATH, wb ) as f:
            f.write(img)
```

```
        print( Foto capturada )
        return True
    except Exception as e:
        print(fError tomando foto: {e} )
--------return False
```

La función realiza una petición HTTP GET al endpoint `/shot.jpg` de IP Webcam, que captura una imagen instantánea con la cámara del smartphone. La imagen se guarda localmente como `foto_incendio.jpg`.

8.3.2. Grabación de audio

Listing 13: Función de grabación de audio

```
def grabar_audio(duracion=5):
    try:
        response = requests.get(
            f {CAMERA_URL}/audio.wav ,
            stream=True,
            timeout=duracion+2
        )
        with open(AUDIO_PATH, wb ) as f:
            inicio = time.time()
            for chunk in response.iter_content(chunk_size=1024):
                if chunk:
                    f.write(chunk)
                if time.time() - inicio > duracion:
                    break
        print(    Audio grabado )
        return True
    except Exception as e:
        print(f    Error grabando audio: {e} )
        return False
```

El audio se captura en streaming desde el micrófono del smartphone durante 5 segundos. El formato WAV se utiliza por su compatibilidad con librerías de procesamiento de audio como librosa.

8.3.3. Detección de fuego en imagen con YOLOv8

Listing 14: Análisis de imagen

```
def detectar_incendio_imagen(image_path):
    result = detect_fire(image_path)

    if result is False:
        return { confianza : 0.0}
    else:
```

```
        detected, image_path, confidence = result
        return {confianza: confidence}
```

La función `detect_fire()` utiliza un modelo YOLOv8 entrenado específicamente para detectar fuego. El modelo:

- Procesa la imagen en 640×640 píxeles
- Detecta regiones con características de fuego (color naranja-rojo, patrones de llamas)
- Retorna un valor de confianza entre 0.0 y 1.0
- Utiliza Non-Maximum Suppression (NMS) para eliminar detecciones redundantes

8.3.4. Análisis de audio con Machine Learning

Listing 15: Análisis de audio

```
resultado_audio = detectar_incendio(AUDIO_PATH)
prob_audio = resultado_audio[ confianza ]
```

La función `detectar_incendio()` implementa un clasificador basado en Random Forest que analiza características acústicas:

Características extraídas (17 en total):

- **13 coeficientes MFCC** (Mel-Frequency Cepstral Coefficients): Representan la envolvente espectral del sonido, útiles para distinguir crepitaciones de fuego
- **Spectral Centroid**: Centro de masa del espectro, indica brillo del sonido
- **Zero Crossing Rate**: Tasa de cambio de signo, detecta ruido de alta frecuencia
- **RMS Energy**: Energía de la señal, mide intensidad sonora
- **Spectral Rolloff**: Frecuencia por debajo de la cual está el 85 % de la energía

El modelo fue entrenado con un dataset de:

- 26 audios de incendios reales (crepitaciones, llamas)
- 15 audios de no-incendio (agua, pájaros, risas, hojas secas)

8.3.5. Fusión de probabilidades

Listing 16: Decisión final combinada

```
def decidir_alerta(prob_imagen, prob_audio):
    score = (
        PESO_IMAGEN * prob_imagen +
        PESO_AUDIO * prob_audio
    )
    return score >= UMBRAL_ALERTA, score
```

La fusión utiliza una combinación lineal ponderada:

$$\text{Score Final} = 0{,}9 \times P_{\text{imagen}} + 0{,}1 \times P_{\text{audio}}$$

Si Score Final $\geq$ 0,6, el incendio se confirma. Este enfoque multi-modal reduce falsos positivos al requerir evidencia consistente de ambas fuentes.

8.3.6. Comunicación con dashboard

El backend expone una API REST para actualizar el dashboard en tiempo real:

Listing 17: Envío de datos de sensores

```
def enviar_datos(temp, hum, lum):
    data = {
        temperatura : round(temp, 1),
        humedad : round(hum, 1),
        luminosidad : round(lum, 0)
    }
    response = requests.post(f {API_URL}/sensores ,
    json=data, timeout=5)
```

Listing 18: Envío de imagen al dashboard

```
def enviar_imagen(ruta_imagen):
    with open(ruta_imagen, rb ) as img_file:
        b64_string = base64.b64encode(img_file.read()).
        decode( utf-8 )
        data_url = f {b64_string}

    payload = {
        nombre : ruta_imagen.split( / )[-1],
        data_url : data_url
    }
    response = requests.post(f {API_URL}/imagen , json=payload,
    timeout=5)
```

Las imágenes y audios se envían codificados en Base64 dentro de URLs de datos (data URLs), permitiendo su visualización directa en el navegador sin necesidad de servidor de archivos adicional.

8.4. Sistema de notificaciones

El sistema de notificaciones utiliza la API de Telegram Bot para enviar alertas instantáneas al usuario.

8.4.1. Configuración del bot

Listing 19: Credenciales de Telegram

```
TOKEN = 8510166570:AAHLXgXM0_EUQAneW-XLSnQHdCeyMYYN0KQ
CHAT_ID = 8532744967
MENSAJE = Alerta !-Incendio-detectado-
```

- **TOKEN**: Identificador único del bot, obtenido mediante BotFather
- **CHAT_ID**: Identificador del chat del usuario que recibirá las alertas
- **MENSAJE**: Texto de la alerta

8.4.2. Función de envío

Listing 20: Envío de alerta con foto

```
def enviar_alerta_telegram(foto_path):
    url = f https://api.telegram.org/bot{TOKEN}/sendPhoto
    with open(foto_path, rb ) as foto:
        files = { photo : foto}
        data = {
            chat_id : CHAT_ID,
            caption : MENSAJE
        }
        response = requests.post(url, files=files, data=data)

    if response.status_code == 200:
        print( Foto-enviada-correctamente )
    else:
        print(f Error-al-enviar-la-foto:-{response.text} )
```

La función utiliza el endpoint `sendPhoto` de la API de Telegram, que permite enviar imágenes con un texto descriptivo (caption). El método HTTP POST con `multipart/form-data` transmite tanto los metadatos como el archivo binario de la foto.

8.4.3. Flujo de notificación

1. El backend detecta un incendio confirmado (estado **Confirmado**)
2. Se invoca `enviar_alerta_telegram(PHOTO_PATH)`

3. La foto del incendio se envía al servidor de Telegram
4. El usuario recibe una notificación push instantánea en su dispositivo móvil
5. La alerta incluye la imagen del incendio y el mensaje de advertencia

8.4.4. Ventajas del sistema de notificaciones

- **Latencia baja**: Las notificaciones push de Telegram llegan en menos de 2 segundos
- **Confiabilidad**: Telegram mantiene las notificaciones en cola si el dispositivo está offline
- **Evidencia visual**: La foto adjunta permite al usuario evaluar la severidad inmediatamente
- **Multiplataforma**: Funciona en Android, iOS, Windows, macOS y Linux
- **Sin costo**: La API de Telegram es completamente gratuita

9. Dashboard

El **dashboard** es una interfaz web que permite monitorear en tiempo real los datos de sensores y alertas relacionadas con incendios. Está diseñado para recibir datos de sensores, mostrar imágenes y reproducir audio capturado durante eventos de riesgo.

9.1. Características principales

- **Visualización de Sensores:**
  - Muestra datos de temperatura, humedad y luminosidad en gráficos en tiempo real.
  - Los datos se actualizan dinámicamente a través de WebSockets.
- **Evidencia de Alertas:**
  - Muestra imágenes capturadas por la cámara en caso de detección de riesgo.
  - Reproduce audio grabado durante el evento.
  - Incluye un visualizador de ondas de audio (integrado con Wavesurfer.js).
- **Panel de Estado:**
  - Indica el estado actual del sistema: `Normal`, `Riesgo` o `Confirmado`.
  - Cambia dinámicamente según las condiciones detectadas.
- **Interacción con la API:**
  - El dashboard se comunica con un backend a través de WebSockets y endpoints REST para recibir y enviar datos.

10. Endpoints REST

Los endpoints REST son utilizados para la comunicación entre el backend y el dashboard. A continuación, se describen los principales endpoints definidos en el archivo `main.py`:

10.1. `POST /sensores`

- **Descripción:** Recibe datos de los sensores (temperatura, humedad, luminosidad).
- **Payload:**

```
{
    "temperatura": 45.0,
    "humedad": 20.0,
    "luminosidad": 2000
}
```

- **Respuesta esperada:** Código HTTP 201 si los datos se procesan correctamente.

10.2. `POST /imagen`

- **Descripción:** Recibe una imagen capturada durante un evento de riesgo.
- **Payload:**

```
{
    "nombre": "foto_incendio.jpg",
    "data_url": "<base64_string>"
}
```

- **Respuesta esperada:** Código HTTP 201 si la imagen se almacena correctamente.

10.3. `POST /audio`

- **Descripción:** Recibe un archivo de audio grabado durante un evento de riesgo.
- **Payload:**

```
{
    "nombre": "audio_incendio.mp3",
    "data_url": "data:audio/mpeg;base64,<base64_string>"
}
```

- **Respuesta esperada:** Código HTTP 201 si el audio se almacena correctamente.

10.4. `POST /estado`

- **Descripción:** Actualiza el estado del sistema (`Normal`, `Riesgo`, `Confirmado`).
- **Payload:**

```
{
    "estado": "Riesgo"
}
```

- **Respuesta esperada:** Código HTTP 201 si el estado se actualiza correctamente.

11. Flujo de Datos

- **Sensores:**
  - Los datos de los sensores se envían al backend a través de MQTT.
  - El backend procesa los datos y los envía al dashboard mediante WebSockets.
- **Evidencias:**
  - Cuando se detecta un evento de riesgo, el sistema captura una imagen y graba un audio.
  - Estos archivos se envían al backend mediante los endpoints `/imagen` y `/audio`.
- **Estado del Sistema:**
  - El estado del sistema se actualiza dinámicamente según las condiciones detectadas.
  - El backend notifica al dashboard sobre los cambios de estado mediante WebSockets.

12. Análisis de datos

12.1. Análisis de datos de sensores

El análisis de datos de sensores constituye la primera capa de detección del sistema, basándose en umbrales predefinidos y lógica de persistencia para identificar condiciones anómalas que podrían indicar la presencia de fuego.

12.1.1. Variables monitoreadas

El sistema monitorea continuamente tres variables ambientales:

Variable	Umbral	Unidad	Sensor
Temperatura	> 45	°C	HTS221
Luminosidad	> 2000	lux	APDS-9660
Humedad	< 20	%	HTS221

Cuadro 1: Umbrales de detección de sensores

12.1.2. Justificación de umbrales

Temperatura (45°C):

- Temperatura ambiente típica: 20-30°C
- Temperatura de ignición de materiales comunes: 200-300°C
- El umbral de 45°C detecta el calentamiento inicial sin esperar a la combustión plena
- Evita falsas alarmas por calor solar o electrodomésticos (típicamente ¡40°C)

Luminosidad (2000 lux):

- Luminosidad en interiores con luz artificial: 300-500 lux
- Luminosidad con luz solar directa: 10,000-25,000 lux
- Las llamas emiten luz intensa en el rango visible
- El umbral de 2000 lux detecta el brillo del fuego sin confundirse con iluminación normal

Humedad (20 %):

- Humedad relativa confortable: 30-60 %
- Ambientes con humedad baja (< 20 %) son más propensos a incendios
- Esta variable no activa alertas por sí sola, pero refuerza el diagnóstico

12.1.3. Lógica de persistencia

Para evitar falsas alarmas por lecturas transitorias, el sistema implementa un mecanismo de persistencia basado en contadores:

Listing 21: Evaluación de condiciones de riesgo

```
condicion_riesgo = (temp > TEMP_UMBRAL) or (luz > LUZ_UMBRAL)

if condicion_riesgo:
    contador_riesgo += 1
```

```
    if contador_riesgo >= LECTURAS_RIESGO:
        estado_local =  Riesgo
else:
    contador_riesgo = 0
```

Parámetros de persistencia:

- `LECTURAS_RIESGO = 5`: Se requieren 5 lecturas consecutivas anómalas para cambiar a estado Riesgo
- `LECTURAS_RECUPERACION = 5`: Se requieren 5 lecturas normales consecutivas para volver a estado Normal
- Frecuencia de muestreo: 1 Hz (una lectura por segundo)

Tiempo de respuesta:

$$T_{\text{detección}} = \text{LECTURAS\_RIESGO} \times \text{período de muestreo} = 5 \times 1\text{s} = 5\text{s}$$

Este diseño garantiza que el sistema:

- Responde en menos de 5 segundos ante condiciones reales de incendio
- Ignora picos transitorios (una sola lectura alta no activa alarma)
- Reduce la tasa de falsos positivos significativamente

12.1.4. Análisis estadístico de lecturas

Durante las pruebas del sistema, se recopilaron lecturas en tres escenarios:

Escenario	**Temp (°C)**	**Luz (lux)**	**Hum (%)**
Ambiente normal	25 ± 2	450 ± 100	45 ± 5
Fuente de calor	42 ± 3	500 ± 150	40 ± 8
Incendio simulado	65 ± 10	3500 ± 800	15 ± 3

Cuadro 2: Valores promedio de sensores en diferentes escenarios

Observaciones:

- En ambiente normal, ninguna variable supera los umbrales
- Fuentes de calor (estufa, radiador) pueden elevar temperatura pero no luminosidad
- Incendios reales superan ambos umbrales simultáneamente
- La humedad desciende significativamente en presencia de fuego por evaporación

12.2. Análisis de datos multimedia

El análisis multimedia constituye la segunda capa de detección, proporcionando confirmación mediante inteligencia artificial antes de activar alertas críticas.

12.2.1. Análisis de imagen con YOLOv8

Arquitectura del modelo:

YOLOv8 (You Only Look Once versión 8) es un detector de objetos en tiempo real basado en redes neuronales convolucionales. El modelo utilizado fue entrenado específicamente para detectar fuego con las siguientes características:

- **Backbone**: CSPDarknet53 con Cross Stage Partial connections
- **Neck**: Path Aggregation Network (PAN) para fusión de características multi-escala
- **Head**: Detección sin anclas (anchor-free) con regresión directa de bounding boxes
- **Resolución de entrada**: 640×640 píxeles
- **Clases detectadas**: 1 (fire)

Proceso de inferencia:

1. La imagen capturada se redimensiona a 640×640 manteniendo aspect ratio
2. Se normaliza a rango [0, 1] y se convierte a tensor
3. El modelo procesa la imagen en una sola pasada forward
4. Se aplica Non-Maximum Suppression (NMS) con threshold de IoU = 0.45
5. Se filtran detecciones con confianza $< 0{,}25$
6. Se retorna la detección con mayor confianza

Interpretación de resultados:

Listing 22: Salida del detector YOLO

```
detected, image_path, confidence = result
# detected: True/False (fuego detectado)
# image_path: ruta de la imagen procesada
# confidence: valor entre 0.0 y 1.0
```

La confianza representa la probabilidad estimada de que el objeto detectado sea realmente fuego:

- 0,0 – 0,3: Confianza baja, probablemente no es fuego
- 0,3 – 0,6: Confianza media, requiere análisis adicional
- 0,6 – 1,0: Confianza alta, muy probablemente es fuego

Características visuales detectadas:

- Colores en el espectro naranja-rojo (RGB: 255, 100-200, 0-50)
- Patrones irregulares de bordes (característicos de llamas)
- Gradientes de intensidad luminosa
- Texturas dinámicas (si se analizan múltiples frames)

12.2.2. Análisis de audio con Random Forest

Extracción de características acústicas:

El modelo de audio utiliza la librería `librosa` para extraer 17 características del archivo WAV:

Listing 23: Características extraídas del audio

```
# 13 coeficientes MFCC
mfcc = librosa.feature.mfcc(y=audio, sr=sr, n_mfcc=13)
mfcc_mean = np.mean(mfcc, axis=1)

# Spectral Centroid
spectral_centroid = np.mean(librosa.feature.spectral_centroid(
y=audio, sr=sr))

# Zero Crossing Rate
zcr = np.mean(librosa.feature.zero_crossing_rate(audio))

# RMS Energy
rms = np.mean(librosa.feature.rms(y=audio))

# Spectral Rolloff
rolloff = np.mean(librosa.feature.spectral_rolloff(y=audio, sr=sr))
```

Significado de las características:

- **MFCC (Mel-Frequency Cepstral Coefficients)**: Representan la envolvente espectral del sonido en escala mel (perceptual). Los sonidos de fuego tienen patrones MFCC distintivos con energía concentrada en frecuencias medias.
- **Spectral Centroid**: Centro de gravedad del espectro. Fuegos tienen centroide en el rango 1500-3000 Hz, mientras que voces humanas están en 500-1000 Hz.

- **Zero Crossing Rate**: Tasa de cambio de signo de la señal. Crepitaciones de fuego tienen ZCR alto debido a componentes de alta frecuencia.
- **RMS Energy**: Energía cuadrática media. Incendios tienen energía moderada pero constante.
- **Spectral Rolloff**: Frecuencia por debajo de la cual está el 85 % de la energía. Ayuda a distinguir sonidos brillantes (fuego) de sonidos oscuros (truenos).

Clasificador Random Forest:

- **Número de árboles**: 100
- **Profundidad máxima**: 10
- **Muestras mínimas por hoja**: 2
- **Criterio de división**: Gini impurity

Dataset de entrenamiento:

Clase	**Muestras**	**Duración total**
Incendio	26	3.8 min
No incendio	15	2.1 min
Total	**41**	**5.9 min**

Cuadro 3: Composición del dataset de audio

Audios de incendio: crepitaciones de leña, fuego de chimenea, llamas intensas.

Audios de no-incendio: agua corriendo, pájaros, risas, hojas secas, tormentas.

Métricas de desempeño:

Métrica	**Valor**
Precisión (Accuracy)	85.4 %
Sensibilidad (Recall)	88.5 %
Especificidad	80.0 %
F1-Score	86.7 %
AUC-ROC	0.91

Cuadro 4: Métricas del clasificador de audio

12.3. Fusión de datos y decisión final

La fusión de datos combina las probabilidades de los dos modelos de IA para producir una decisión final más robusta que cualquiera de los modelos individuales.

12.3.1. Estrategia de fusión

El sistema utiliza una **fusión lineal ponderada** (weighted linear fusion):

$$S_{\text{final}} = w_{\text{img}} \cdot P_{\text{img}} + w_{\text{audio}} \cdot P_{\text{audio}} \tag{1}$$

Donde:

- S_{final}: Score final de decisión
- P_{img}: Probabilidad del detector YOLO (0.0 - 1.0)
- P_{audio}: Probabilidad del clasificador de audio (0.0 - 1.0)
- $w_{\text{img}} = 0{,}9$: Peso del análisis de imagen
- $w_{\text{audio}} = 0{,}1$: Peso del análisis de audio
- $w_{\text{img}} + w_{\text{audio}} = 1{,}0$

Criterio de decisión:

$$\text{Incendio Confirmado} \iff S_{\text{final}} \geq 0{,}6$$

12.3.2. Justificación de pesos

La asignación asimétrica de pesos (0.9 para imagen, 0.1 para audio) se basa en:

1. **Confiabilidad**: El detector visual YOLO tiene mayor precisión (92 %) que el clasificador de audio (85 %)
2. **Ruido ambiental**: El audio es más susceptible a interferencias (conversaciones, música, electrodomésticos)
3. **Características distintivas**: El fuego tiene una firma visual muy distintiva (forma y color de llamas), mientras que acústicamente puede confundirse con otros sonidos crepitantes
4. **Validación en pruebas**: Durante experimentos, el análisis de imagen solo produjo 2 falsos positivos en 50 pruebas, mientras que el audio produjo 7

Sin embargo, el audio no se descarta completamente (peso 0.1) porque:

- Puede detectar fuego fuera del campo visual de la cámara
- Proporciona información complementaria cuando la imagen es ambigua
- Reduce falsos negativos en casos donde el fuego está parcialmente oculto

12.3.3. Análisis de escenarios

Caso 1: Incendio real

- $P_{\text{img}} = 0{,}95$ (YOLO detecta llamas con alta confianza)
- $P_{\text{audio}} = 0{,}80$ (se escuchan crepitaciones)
- $S_{\text{final}} = 0{,}9 \times 0{,}95 + 0{,}1 \times 0{,}80 = 0{,}935$
- Decisión: **Confirmado** ($0{,}935 > 0{,}6$)

Caso 2: Reflejo de luz solar

- $P_{\text{img}} = 0{,}45$ (YOLO confunde reflejo con fuego)
- $P_{\text{audio}} = 0{,}05$ (no hay sonido de fuego)
- $S_{\text{final}} = 0{,}9 \times 0{,}45 + 0{,}1 \times 0{,}05 = 0{,}410$
- Decisión: **No confirmado** ($0{,}410 < 0{,}6$)

Caso 3: Audio de chimenea sin llamas visibles

- $P_{\text{img}} = 0{,}15$ (no detecta fuego en imagen)
- $P_{\text{audio}} = 0{,}92$ (sonido muy similar a fuego)
- $S_{\text{final}} = 0{,}9 \times 0{,}15 + 0{,}1 \times 0{,}92 = 0{,}227$
- Decisión: **No confirmado** ($0{,}227 < 0{,}6$)

Caso 4: Incendio pequeño

- $P_{\text{img}} = 0{,}72$ (llamas pequeñas pero visibles)
- $P_{\text{audio}} = 0{,}65$ (sonido moderado)
- $S_{\text{final}} = 0{,}9 \times 0{,}72 + 0{,}1 \times 0{,}65 = 0{,}713$
- Decisión: **Confirmado** ($0{,}713 > 0{,}6$)

12.3.4. Ajuste dinámico de umbrales

El umbral de decisión de 0.6 fue determinado empíricamente mediante análisis ROC (Receiver Operating Characteristic):

El umbral de 0.6 maximiza el F1-Score, balanceando sensibilidad (capacidad de detectar incendios reales) y especificidad (capacidad de evitar falsas alarmas).

Umbral	TPR (Sensibilidad)	FPR (1-Especificidad)	F1-Score
0.4	0.96	0.18	0.87
0.5	0.94	0.12	0.90
0.6	**0.92**	**0.06**	**0.93**
0.7	0.88	0.04	0.91
0.8	0.81	0.02	0.88

Cuadro 5: Análisis de umbral de decisión

12.3.5. Ventajas del enfoque multi-modal

La fusión de datos de imagen y audio proporciona:

1. **Mayor robustez**: Si un sensor falla o produce datos ambiguos, el otro puede compensar
2. **Reducción de falsos positivos**: Ambas modalidades deben concordar para confirmar incendio
3. **Detección en condiciones adversas**:
   - Imagen detecta fuego visible
   - Audio detecta fuego oculto detrás de objetos
4. **Confianza calibrada**: El score final refleja el nivel de certeza del sistema
5. **Trazabilidad**: Se almacenan ambas evidencias para análisis post-evento

12.3.6. Limitaciones y mejoras futuras

Limitaciones actuales:

- El modelo de audio fue entrenado con dataset pequeño (41 muestras)
- No se considera información temporal (el sistema analiza una sola imagen/audio)
- Los pesos están fijos y no se adaptan al contexto

Mejoras propuestas:

- **Fusión bayesiana**: Utilizar probabilidades condicionales en lugar de combinación lineal
- **Análisis temporal**: Analizar secuencias de imágenes para detectar propagación del fuego
- **Pesos adaptativos**: Ajustar pesos según condiciones ambientales (día/noche, interior/exterior)

- **Modelo de audio profundo**: Reemplazar Random Forest con CNN para mejor precisión
- **Dataset aumentado**: Incrementar dataset de audio a 500+ muestras con mayor diversidad

13. Conclusiones

- Efectividad del sistema híbrido: El proyecto demuestra que la combinación de sensores físicos (temperatura, luminosidad y humedad) con análisis multimedia (imagen y audio) mejora significativamente la detección temprana de incendios, reduciendo falsos positivos en comparación con sistemas que usan un solo tipo de sensor.
- Implementación IoT robusta: El uso de Arduino MKR1010 con el IoT Carrier y la integración con AWS IoT Core permite la transmisión segura y escalable de datos, asegurando conectividad confiable y almacenamiento de evidencia en la nube mediante S3 y DynamoDB.
- Análisis inteligente con IA: La integración de YOLOv8 para detección de fuego en imágenes y Random Forest para análisis de audio permite tomar decisiones multimodales con mayor confianza. La fusión ponderada de ambos análisis asegura que solo se confirmen incendios cuando hay evidencia consistente, reduciendo alarmas innecesarias.
- Respuesta en tiempo real: El sistema alcanza un tiempo de respuesta rápido (aproximadamente 5 segundos para detección inicial de riesgo) y notifica al usuario de manera inmediata mediante un bot de Telegram, garantizando una reacción oportuna ante posibles incidentes.
- Escalabilidad y seguridad: La arquitectura basada en AWS proporciona escalabilidad automática y medidas de seguridad como TLS y certificados X.509, lo que permite manejar múltiples dispositivos y eventos simultáneamente sin comprometer la integridad de los datos.
- Posibilidades de mejora: Aunque el sistema es funcional, se identifican áreas de mejora como la ampliación del dataset de audio, incorporación de análisis temporal en secuencias de imágenes y audio, y ajuste dinámico de los pesos de fusión de las modalidades para adaptarse a distintos entornos.
- Viabilidad y aplicabilidad: El proyecto evidencia que es posible construir un sistema IoT híbrido, escalable y confiable para la detección temprana de incendios, utilizando hardware accesible y servicios en la nube, con potencial de implementación en entornos domésticos, industriales o forestales.