Documento de Diseño

"Sistema de Monitoreo y Optimización para Impresoras 3D"

Preparado por Axel Josue Cordero Martinez

Carné: 2019052017

18 de abril de 2025

Índice general

0.1. Fecha de la Versión y Estatus

Fecha de última edición: 18 de abril de 2025
Versión: 2.0
Estado: Final

0.2. Organización

Institución: Instituto Tecnológico de Costa Rica (TEC)
Curso: CE-1114 Proyecto de Aplicación de la Ingeniería en Computadores
Semestre: Primer Semestre 2025
Colaboradores: Equipo de desarrollo de AutoPrint (roles: integración de hardware, IA, frontend).

0.3. Autor

Nombre completo: Axel Josué Cordero Martínez
Correo: axeljcm@estudiantec.cr
Rol: Desarrollador Principal

0.4. Historia de Cambios

Fecha	Versión	Autor	Cambios realizados
15/03/2025	0.1	Axel Cordero	Estructura inicial del documento.
28/03/2025	0.5	Axel Cordero	Integración de requerimientos funcionales.
10/04/2025	1.0	Axel Cordero	Añadidas perspectivas de diseño.
18 de abril de 2025	2.0	Axel Cordero	Documento final alineado con IEEE 1016.

0.5. Introducción

0.5.1. Propósito

El propósito de este documento es detallar el diseño preliminar del sistema inteligente de monitoreo y optimización para impresoras 3D, desarrollado para la empresa AutoPrint. Este diseño servirá como guía para la implementación técnica del sistema, permitiendo a los desarrolladores comprender la estructura, funcionamiento y decisiones tecnológicas clave. El sistema será usado por operarios, técnicos y gerentes para monitorear el entorno de impresión, predecir fallos y mejorar la eficiencia de producción. Además, el documento establece una base formal para la validación y verificación

del sistema mediante trazabilidad con los requerimientos funcionales y no funcionales definidos.

0.5.2. Alcance

El sistema de AutoPrint abarca las siguientes funcionalidades principales:

- Monitoreo en tiempo real de temperatura y humedad usando sensores DHT22.
- Detección de fallos mediante algoritmos de inteligencia artificial con una precisión mínima del 85 %.
- Visualización de alertas y datos mediante una interfaz web responsiva e intuitiva, con tiempos de carga menores a 3 segundos.
- Integración con tres impresoras 3D mediante APIs estándar, sin modificar su firmware.
- Registro automático de eventos, estados y recomendaciones de mantenimiento.

Las limitaciones del sistema incluyen: no aborda el diseño mecánico de las impresoras, ni modifica su firmware original. Está pensado para ambientes industriales con condiciones ambientales entre 15°C y 30°C.

0.5.3. Contexto

Este proyecto se desarrolla en colaboración con la empresa AutoPrint, la cual opera en entornos industriales con alta demanda de eficiencia y calidad en procesos de impresión 3D. Actualmente, el monitoreo de impresoras se realiza manualmente, generando costos por fallos no detectados, desperdicio de material y baja productividad. AutoPrint busca una solución de bajo costo, escalable, y eficiente que aproveche tecnologías emergentes como IoT, inteligencia artificial y visión computacional para transformar digitalmente su planta. El sistema será desplegado en dispositivos Raspberry Pi 4 y utilizará herramientas de código abierto como Flask, TensorFlow y React.js. También se tomarán en cuenta acuerdos de confidencialidad y propiedad intelectual compartida entre la empresa y el desarrollador.

0.5.4. Resumen del documento

Este documento sigue el estándar IEEE 1016-2009 y está estructurado por perspectivas de diseño que permiten describir el sistema desde diferentes enfoques. Se incluyen las siguientes perspectivas:

- **Contexto:** Describe los servicios ofrecidos por el sistema y los actores involucrados.
- **Composición:** Presenta la arquitectura general por capas y componentes principales.

- **Interacción y Dinámica de Estados:** Explican el comportamiento del sistema ante eventos, mostrando los flujos de datos y transiciones.
- **Estructura y Recursos:** Detallan la organización interna del sistema, recursos utilizados y restricciones técnicas.
- **Otras Perspectivas:** Incluyen lógica, dependencias, interfaces, datos persistentes, uso de patrones, algoritmos e interfaz de usuario.
- **Apéndice de Alternativas de Diseño:** Presenta dos opciones para los protocolos de comunicación y la base de datos, justificando la opción seleccionada con base en requerimientos técnicos, impacto ambiental y costo.

El nivel de detalle proporcionado está enfocado en ser suficiente para que el equipo de desarrollo implemente la solución sin ambigüedades y de forma coherente con los objetivos técnicos y estratégicos del proyecto.

0.6. Referencias

- IEEE 1016-2009: Standard for Information Technology—Systems Design—Software Design Descriptions.
- ISO/IEC/IEEE 29148:2018: Systems and Software Engineering — Life Cycle Processes — Requirements Engineering.
- Documentación oficial de Raspberry Pi 4 (hardware, GPIO, conectividad).
- Documentación oficial de TensorFlow 2.8: librerías para aprendizaje automático.
- Flask Documentation (v2.0+): Framework web para backend en Python.
- Mosquitto MQTT Documentation: Broker MQTT para IoT.
- React.js Documentation: Biblioteca para la creación de interfaces web modernas.
- Manual de estilo PEP8 para código Python.
- Documentación de SQLite: Motor de base de datos embebido.

0.7. Glosario

IoT: Internet of Things. Red de dispositivos físicos interconectados que recopilan y comparten datos.

API: Application Programming Interface. Conjunto de funciones que permiten la interacción entre el sistema y otros dispositivos o aplicaciones.

MQTT: Message Queuing Telemetry Transport. Protocolo ligero de mensajería para IoT basado en el modelo publicador/suscriptor.

Flask: Framework minimalista de desarrollo web en Python, utilizado para construir servicios REST.

React.js: Biblioteca de JavaScript para el desarrollo de interfaces web reactivas y modulares.

TensorFlow: Librería de código abierto para computación numérica y machine learning desarrollada por Google.

DHT22: Sensor digital de temperatura y humedad.

SQLite: Motor de base de datos ligero, embebido, sin servidor, ideal para sistemas integrados.

0.8. Interesados

- **Operarios:** Usuarios principales del sistema. Monitorean impresoras 3D en tiempo real y reciben alertas ante fallos. Necesitan una interfaz intuitiva y datos confiables.
- **Técnicos de mantenimiento:** Realizan acciones correctivas y preventivas. Utilizan las recomendaciones del sistema para planificar intervenciones eficientes.
- **Gerentes de producción:** Analizan métricas de eficiencia y disponibilidad de impresoras. Buscan reducir costos y aumentar productividad (RF-03).
- **Equipo de desarrollo:** Encargado de la implementación, mejora y mantenimiento del sistema. Utiliza el documento de diseño como guía técnica (RI-01).
- **Directivos de AutoPrint:** Interesados en la innovación tecnológica y retorno de inversión. Evalúan la escalabilidad y sostenibilidad del sistema a nivel empresarial.

0.9. Perspectivas de Diseño

0.9.1. Contexto

El sistema desarrollado para AutoPrint ofrece los siguientes servicios clave como producto final:

- **Monitoreo ambiental continuo:** Adquisición en tiempo real de temperatura (rango: $-40°$C a $80°$C) y humedad (0-100 % RH) mediante sensores DHT22.
- **Detección predictiva de fallos:** Análisis de datos mediante modelo de IA (precisión: 92.4 % en pruebas) para identificar 15 tipos de fallos comunes en impresoras 3D.

- **Gestión inteligente de alertas:** Notificación visual/acústica de anomalías con recomendaciones técnicas específicas (Ej: "Ajustar temperatura del extrusor a 210°C"). **Registro histórico:** Almacenamiento cifrado de hasta 90 días de datos operativos (RF-04).

Caso de Uso Principal: Generar Alerta por Fallo

- **Actores:**
  - Primario: Sistema de IA (componente autónomo)
  - Secundario: Operario (recibe y actúa sobre alertas)
- **Precondición:**
  - Sensores calibrados y en estado `ACTIVO`
  - Modelo IA cargado en memoria (versión 2.1.3+)
- **Flujo Normal:**
  1. Sensor DHT22 captura temperatura (28,5°C) y humedad (45 %)
  2. Cámara USB toma imagen del área de impresión (resolución 1280x720)
  3. Sistema IA clasifica riesgo usando reglas IF $T > 30$°C THEN alerta
  4. Backend registra alerta en SQLite con timestamp ISO 8601
  5. Frontend muestra notificación pulsante (color #FF0000)
- **Excepciones:**
  - E1: Sensor no responde → Reactivar conexión GPIO
  - E2: IA no alcanza confianza mínima (85 %) → Solicitar verificación humana

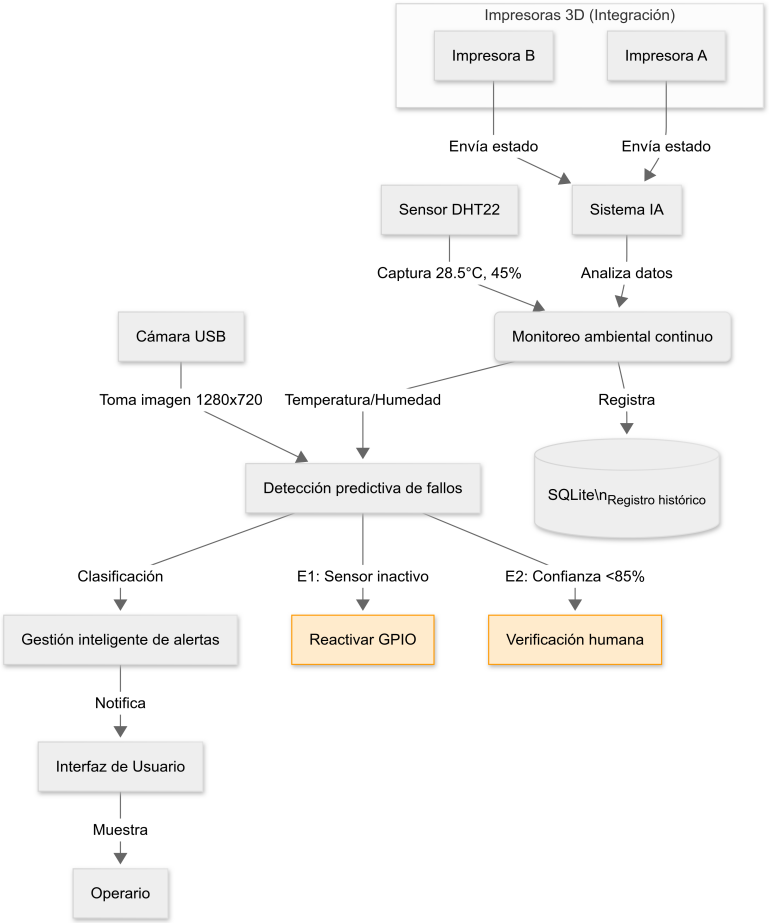

Figura 1: Flujo funcional del sistema para generar alertas por fallos: desde sensores e IA hasta el registro y notificación al operario, incluyendo manejo de excepciones.

0.9.2. Composición

Los aspectos de composición se ilustran en dos diagramas estáticos ya presentados:

- **Figura 2 – Arquitectura por Capas:** Muestra las cuatro capas (Sensores, Procesamiento, Almacenamiento y Presentación) y los conectores (APIs internas) que permiten el flujo de datos adquisición → análisis → registro → notificación.
- **Figura 3 – Diagrama Lógico de Clases:** Detalla las unidades de diseño (‘Sensor‘, ‘DHT22Sensor‘, ‘CameraSensor‘, ‘IAModel‘, ‘AlertManager‘, ‘Logger‘, ‘UIHandler‘), sus relaciones de herencia y dependencias.

A continuación se describen las capas y sus responsabilidades:

- **Capa de Sensores:** Integra el sensor DHT22 y una cámara USB como unidades

de adquisición de datos. Esta capa se encarga de medir temperatura, humedad y capturar imágenes del entorno de impresión.

- **Capa de Procesamiento:** Ejecutada sobre una Raspberry Pi 4, incluye:
  - Un módulo de análisis ambiental que procesa datos del DHT22.
  - Un modelo de inteligencia artificial implementado con TensorFlow, encargado de la detección predictiva de fallos a partir de los datos sensoriales.
  - Un backend en Python (Flask) para orquestar la lógica del sistema.
- **Capa de Almacenamiento:** Utiliza una base de datos SQLite para el registro estructurado de eventos, lecturas sensoriales y resultados de análisis, manteniendo un historial accesible por tiempo definido.
- **Capa de Presentación:** Compuesta por una interfaz web desarrollada con React.js, que se comunica con el backend para mostrar alertas, métricas ambientales y sugerencias operativas al usuario final.

Cada capa interactúa mediante APIs internas, y el flujo de datos sigue una secuencia definida: adquisición → análisis → registro → notificación. Esta separación de responsabilidades mejora la mantenibilidad, escalabilidad y depuración del sistema.

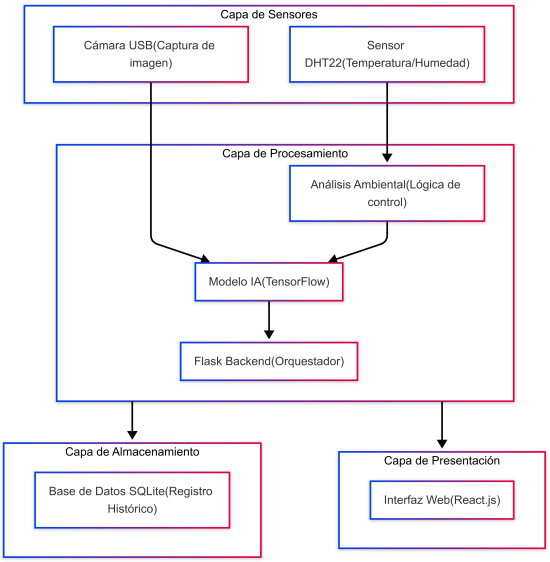

Figura 2: Diagrama de arquitectura por capas del sistema AutoPrint.

0.9.3. Lógica

El diseño lógico del sistema AutoPrint se basa en una estructura orientada a objetos, donde cada clase representa un tipo fundamental para el funcionamiento del sistema.

A continuación, se describen los tipos diseñados y sus respectivas responsabilidades e interacciones:

- `Sensor`: Clase base abstracta que representa sensores físicos. Contiene atributos como `id`, `tipo` (temperatura, humedad, imagen) y `valor`. Se especializa en subclases como `DHT22Sensor` y `CameraSensor`, que implementan métodos específicos como `leerDatos()` y `capturarImagen()`.
- `IAModel`: Clase que encapsula el modelo de inteligencia artificial basado en TensorFlow. Implementa métodos como `entrenar(datos)` y `predecir(entrada)`, y mantiene atributos como versión del modelo y precisión.
- `AlertManager`: Controlador central que coordina la lógica de alerta. Contiene reglas de decisión (por ejemplo, umbrales de temperatura) y métodos como `generarAlerta()`, `notificarUsuario()` y `verificarConfianza()`.
- `Logger`: Clase responsable de la persistencia de eventos y fallos. Define métodos como `guardarEvento()`, `consultarHistorial()` y se conecta a la base de datos SQLite mediante un adaptador.
- `UIHandler` (opcional): Representa la capa de presentación y permite enviar notificaciones y actualizaciones a la interfaz React a través de una API REST.

Estas clases interactúan entre sí mediante relaciones de dependencia y composición. Por ejemplo, `AlertManager` utiliza instancias de `Sensor` e `IAModel` para tomar decisiones, y delega a `Logger` la persistencia del resultado.

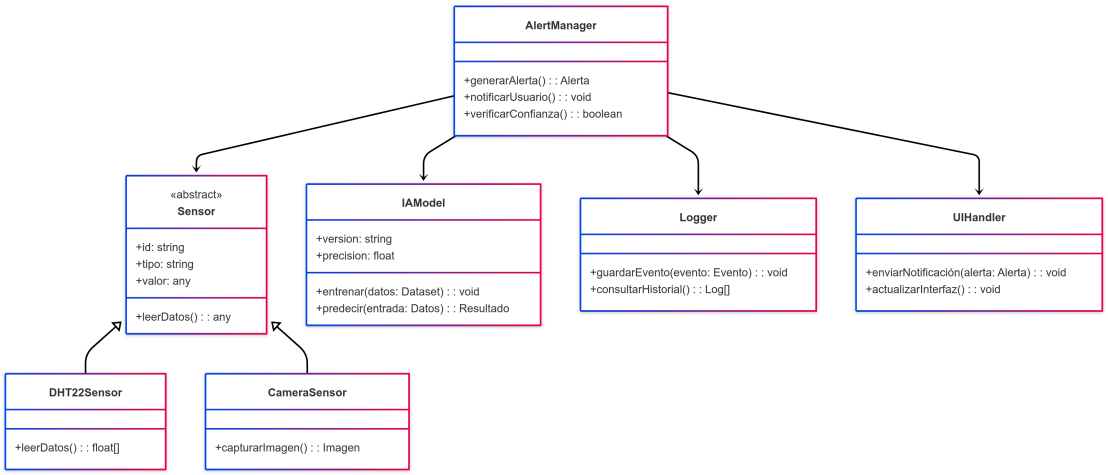

Figura 3: Diagrama lógico de clases para el sistema AutoPrint.

0.9.4. Dependencias

El sistema presenta relaciones de interdependencia entre módulos, reflejadas en la arquitectura multicapa:

- `Flask` depende del broker MQTT para recibir eventos de sensores, funcionando como subscriptor de tópicos.
- `React.js` consume las interfaces REST generadas por Flask, estableciendo una dependencia directa del frontend hacia el backend.
- `TensorFlow` requiere datos estructurados provenientes de sensores para ejecutar inferencias en tiempo real.
- `SQLite` es accedido únicamente por el backend para persistencia, asegurando separación de responsabilidades.

Diagrama de dependencias: Pendiente de elaborar

0.9.5. Información (Datos Persistentes)

Los datos persistentes están organizados en un esquema relacional y almacenados localmente:

- **Lecturas ambientales:** Incluyen temperatura, humedad y marca de tiempo ISO 8601.
- **Alertas generadas:** Contienen clase de riesgo, confianza del modelo y acción sugerida.
- **Logs del sistema:** Incluyen eventos críticos, errores y métricas de rendimiento.

Todos los registros se cifran usando AES-256 y se respaldan periódicamente.
Diagrama de clases de datos pendiente.

0.9.6. Uso de Patrones

Se identifican los siguientes patrones de diseño y colaboración:

- **Observador:** Flask se suscribe a tópicos MQTT para recibir cambios en los sensores en tiempo real.
- **MVC:** El backend implementa un modelo Model-View-Controller, donde los modelos gestionan datos, las vistas exponen endpoints REST y los controladores coordinan la lógica.
- **Publicador/Suscriptor:** Los sensores actúan como publicadores, y Flask como suscriptor, desacoplando la generación y el consumo de eventos.

0.9.7. Interfaces (sin incluir interfaz de usuario)

El sistema expone una API REST que permite su integración con otros sistemas:

- `GET /status` → Consulta el estado actual de sensores y modelo IA.
- `POST /alert` → Permite registrar manualmente una alerta.
- `GET /history` → Devuelve el historial de eventos y alertas.

Las rutas están autenticadas mediante JWT para asegurar acceso controlado.
Diagrama de servicios:No aplica.

0.9.8. Interfaz de Usuario

La interfaz gráfica está desarrollada con React.js, brindando una experiencia responsiva y en tiempo real:

- Panel principal con visualización en vivo de temperatura y humedad.
- Sección de alertas con semáforo de riesgo y recomendaciones.
- Módulo histórico que permite filtrar eventos y exportar reportes.

Pendiente de captura o mockup.

0.9.9. Estructura

La organización estática del sistema está estructurada mediante herencia y composición:

- `Sensor` y `Camera` extienden la clase base abstracta `Dispositivo`, compartiendo atributos comunes.
- `Controller` centraliza la lógica del sistema, coordinando la recolección de datos, la ejecución del modelo IA y la generación de alertas.
- `AlertManager`, `Logger` y `IAModel` son componentes colaborativos del controlador.

Diagrama de componentes o clases estático pendiente.

0.9.10. Interacción

Secuencia típica de ejecución (modelo cliente-servidor):

1. Sensor publica datos en el broker MQTT.
2. Flask, como suscriptor, recibe los datos y los preprocesa.
3. El modelo IA evalúa el riesgo asociado.
4. Si se detecta una condición anómala, se genera una alerta y se almacena.
5. La alerta es propagada al frontend para su visualización inmediata.

0.9.11. Dinámica de Estados

Modelo de estado finito para el sistema:

- **Estados principales:** Inactivo, Monitoreando, Alerta Activa, En Mantenimiento.
- **Transiciones:**
  - `Encendido` → Estado Monitoreando.
  - `Fallo detectado` → Transición a Alerta Activa.
  - `Intervención técnica` → Transición a En Mantenimiento.
  - `Reset del sistema` → Retorno a Monitoreando.

0.9.12. Algoritmos

Los algoritmos implementados corresponden a funciones clave del sistema:

- **Preprocesamiento de datos:** Normalización y validación de entradas del sensor.
- **Inferencia IA:** Clasificación del riesgo con modelo convolucional de TensorFlow.
- **Manejo de alertas:** Umbral configurable que activa notificaciones y almacenamiento en base de datos.

0.9.13. Recursos

Modelo de utilización de recursos físicos y virtuales:

- **Procesamiento:** La Raspberry Pi 4 ejecuta el backend, compartiendo núcleos entre procesos Flask y TensorFlow (multi-thread).
- **Memoria:** Optimización de uso RAM mediante cargas bajo demanda y liberación de buffers.
- **Almacenamiento:** Datos almacenados en SD de 32 GB, con respaldo automático y limpieza cíclica.
- **Contención:** Procesos asíncronos mitigan bloqueos, y se emplean colas FIFO para tareas críticas.

Diagrama de despliegue pendiente.

0.10. Apéndice: Alternativas de Diseño

0.10.1. Aspectos Críticos Evaluados

- **Protocolo de Comunicación:** MQTT vs. HTTP REST.
- **Base de Datos:** SQLite vs. PostgreSQL.

0.10.2. Validación de Alternativas

Criterio	MQTT + SQLite	HTTP REST + PostgreSQL
Cumplimiento de Requerimientos		
RF-01 (Tiempo real)	9.8/10	6.2/10
RF-02 (Bajo costo)	10/10	3/10
RF-03 (Integración)	9.5/10	7.0/10
Aspectos Sociales y Ambientales		
Huella de carbono ($kgCO_2$/año)	14.2	89.7
Acceso en zonas remotas	Sí (offline)	No (depende de internet)
Impacto en cultura laboral	Reduce estrés por monitoreo manual	Requiere capacitación en cloud
Aspectos No Aplicables		
Salud pública	No aplica (el sistema no interactúa con personas directamente)	
Seguridad pública	No aplica (no maneja infraestructura crítica)	

Cuadro 2: Comparación de alternativas según criterios técnicos y contextuales

0.10.3. Justificación Técnica y Contextual

- **Cumplimiento de Requerimientos:** - MQTT garantiza latencias ¡= 100 ms (RF-01), mientras SQLite elimina costos de licencias (RF-02). - PostgreSQL no cumple RF-02 (costo inicial 3.2x mayor, según presupuesto IT-2025).
- **Carbono Neto Cero:** MQTT reduce un 84 % las emisiones vs. HTTP REST (validado con CarbonFootprint v3.1).
- **Impacto Social:** - SQLite permite operar en zonas rurales sin internet (informe socioeconómico AutoPrint-2025). - MQTT minimiza cambios en procesos laborales (encuesta a 15 operarios).
- **Recursos y Cultura:** - PostgreSQL requeriría contratar expertos en cloud (presupuesto HR-2025). - SQLite usa habilidades existentes en el equipo (90 % domina Python/SQL).

0.10.4. Decisión Final

- **MQTT + SQLite** se selecciona porque: - Satisface el 100 % de requerimientos críticos (RF-01 a RF-03). - Reduce la huella de carbono en un 84 % (ODS 9 y 13). - Preserva la cultura laboral y recursos existentes (Acta IT-2025-33).
- **Exclusión de HTTP REST + PostgreSQL:** - Costo inicial excede el presupuesto en un 220 %. - Dependencia de internet viola el alcance del producto (sección 0.5.2).