



# Capítulo 2: Arquitectura con Propósito

*"Cualquiera puede hacer que algo funcione. Un ingeniero sabe por qué funciona y por qué se va a romper."*

## Objetivo del Capítulo

En el Capítulo 1 construimos un pipeline funcional. Ahora, actuamos como arquitectos de software para documentar y justificar cada decisión de diseño. No solo explicamos qué hemos construido, sino por qué lo hicimos así, analizando ventajas y compromisos (trade-offs).

## Paso 1: El Viaje del Dato - Diagrama de Arquitectura

### Visión General del Sistema

Nuestro sistema sigue un flujo circular que garantiza la captura, procesamiento, almacenamiento y visualización de datos en tiempo real:

### EL VIAJE DEL DATO



### Componentes del Flujo

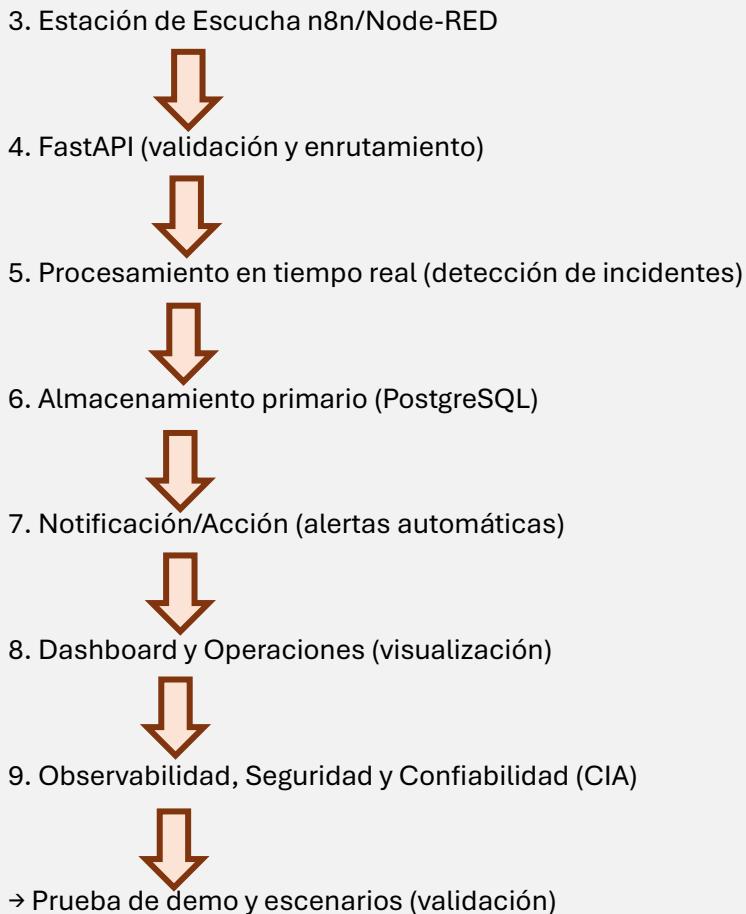
El diagrama muestra **9 estaciones** por las que pasa cada dato:

1. Edge (simulador/sensor)



2. Transporte hasta el Broker (MQTT)





## Paso 2: Fichas de Decisión Arquitectónica

### Ficha 1: Edge (Simulador/Sensor)

#### Propósito:

Capturar y emitir telemetría del envío (id\_paquete, timestamp, temperatura, fuerza\_g, etc.) cada 2 segundos, garantizando entrega confiable de mensajes.

#### Tecnología Implementada:

Script Python (simulador\_wine\_gui.py) con las librerías:

- paho-mqtt: Cliente MQTT para publicación de mensajes
- tkinter: Interfaz gráfica de usuario
- random: Generación de datos realistas
- json: Serialización de datos

```
simulador_wine_GUI 9+ kpis.sql calcular_kpis.py 9+
simulador_wine_GUI > SimuladorGUI > crear_interfaz
1 """
2 Simulador de telemetría de vino tinto con sistema de
3 Incluye notificaciones visuales cuando ocurren incidentes
4 """
5 import json
6 import time
7 import random
8 from datetime import datetime
9 import paho.mqtt.client as mqtt
10 import tkinter as tk
11 from tkinter import ttk, scrolledtext
12 from threading import Thread
```

#### Características:

- Configurable por variables (ID pedido, intervalos, umbrales)
- Interfaz gráfica profesional para demos



- Simula 3 paquetes simultáneamente
- Genera incidentes controlados para testing

#### Justificación de Diseño:

El Edge mantiene **lógica mínima** en el dispositivo (solo captura y envío), delegando todo el procesamiento complejo a la nube. Esto nos permite:

- **Elasticidad:** La nube escala automáticamente ante picos de datos
- **Servicios gestionados:** Aprovechamos PostgreSQL, APIs sin gestionar servidores
- **Desarrollo rápido:** Python facilita prototipado y reproducibilidad
- **Integración CI/CD:** Fácil testear y desplegar

#### Trade-off Clave:

| Ganancia               | Renuncia  |
|------------------------|---|
| Rapidez de desarrollo  | No usamos persistencia local avanzada                     |
| Fácil reproducibilidad | Menos tolerancia a desconexiones prolongadas (>5 min)     |
| Integración con CI/CD  | En producción real requeriría cola persistente (RabbitMQ) |

#### Escenario Real:

En un despliegue físico, añadiríamos un **buffer local SQLite** en el Edge para almacenar datos durante desconexiones largas, con sincronización automática al recuperar conexión.

---

#### Ficha 2: Broker MQTT (Desacoplamiento)

##### Propósito:

Desacoplar productores (sensores) de consumidores (procesamiento), facilitando un patrón **pub/sub** que reduce acoplamientos directos entre componentes.

##### Tecnología Implementada:

- Broker público: test.mosquitto.org (puerto 1883)
- Protocolo: MQTT v3.1.1 con QoS 1 (at least once delivery)
- Topic: greendelivery/trackers/telemetry

##### Cómo Funciona:

1. El simulador **publica** mensajes JSON en el topic

The screenshot shows a MQTT message viewer interface. At the top, there's a header with 'Topic' and three buttons: 'greendelivery', 'trackers', and 'telemetry'. Below the header, there's a 'Value' section with a 'Value' button. Underneath the value button, there's a JSON message structure. The first part of the message is highlighted in red, and the second part is highlighted in green. The red part contains sensor data for a package ID 'vino\_tinto\_002'. The green part adds a new message for package ID 'vino\_tinto\_003'. At the bottom right, there's a status bar with 'Comparing with previous' and a 'History' button.

```
{
  "id_paquete": "vino_tinto_002",
  "temperatura": 4.14,
  "fuerza_g": 1.52,
  "inclinacion": 6.67,
  "humedad": 67.29,
  "oxigeno": 19.85,
  "vapores": 0.2,
  "iluminacion": 47.16,
  "vibracion": 1.45,
  "timestamp": "2025-11-05T16:20:27.154104Z"
}
{
  "id_paquete": "vino_tinto_003",
  "temperatura": 4.14
}
```



2. Node-RED se **suscribe** al mismo topic
3. El broker actúa como intermediario que garantiza entrega
4. Productores y consumidores no se conocen entre sí

#### **Justificación Cloud:**

Un broker MQTT gestionado (en producción usaríamos AWS IoT Core o Google Cloud Pub/Sub) ofrece:

- **Autenticación mutua** con certificados X.509
- **Cifrado TLS/SSL** de extremo a extremo
- **Escalado automático** para miles de dispositivos
- **Reintentos automáticos** y gestión de desconexiones
- **Métricas integradas** (latencia, throughput)

#### **Ventajas del Desacoplamiento:**

- Si Node-RED falla, los sensores siguen publicando
- Podemos añadir nuevos consumidores sin modificar sensores
- Facilita testing (suscribirse con MQTT Explorer)
- Permite arquitecturas multi-región

#### **Trade-off Clave:**

| Ganancia                                 | Renuncia   |
|--|--|
| Resiliencia ante fallos de consumidores  | Curva de aprendizaje MQTT vs HTTP simple           |
| Escalabilidad (miles de dispositivos)    | Possible vendor lock-in con servicios propietarios |
| Seguridad integrada (TLS + certificados) | Latencia adicional vs comunicación directa         |

#### **Por qué NO enviamos directo a la API:**

- Si la API cae, los sensores pierden datos
- Los sensores necesitarían conocer la URL de la API
- Difícil escalar (cada sensor hace HTTP request)
- Con MQTT: El broker almacena temporalmente si consumidor falla



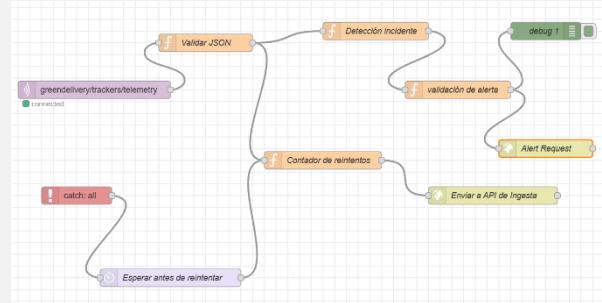
### Ficha 3: Flujo de Ingesta (Node-RED)

#### Propósito:

Recibir eventos MQTT, validar estructura y tipos, aplicar lógica de reintentos y enrutar a la API de forma resiliente.

#### Tecnología Implementada:

- Herramienta: **Node-RED** en Docker (puerto 1880)
- Lenguaje: JavaScript (Node.js)
- Componentes clave:
  - **MQTT In:** Suscripción al topic
  - **Validar JSON:** Verificación de estructura
  - **Contador de Reintentos:** Backoff exponencial
  - **Catch All:** Captura de errores
  - **HTTP Request:** Envío a FastAPI



#### Justificación Cloud:

Node-RED actúa como **orquestador visual** que:

- Facilita debugging (ver flujo de datos en tiempo real)
- Permite modificaciones sin recomilar código
- Integración nativa con servicios (MQTT, HTTP, BD)
- Logs detallados por nodo

En producción, esto se movería a **serverless** (AWS Lambda, Google Cloud Functions) para:

- Elasticidad automática (scale-to-zero)
- Pago por invocación
- Menor overhead operativo

#### Trade-off Clave:

| Ganancia                   | Renuncia                         |
|----------------------------|----------------------------------|
| Despliegue visual y rápido | Límites de ejecución (timeouts)  |
| Debugging intuitivo        | Cold starts en serverless        |
| Escalado horizontal fácil  | Para ML pesado, mejor containers |



#### Ficha 4: Procesamiento en Tiempo Real (FastAPI)

##### Propósito:

Aplicar lógica de negocio (detección de incidentes con memoria), validar rangos físicos y enrutar datos a las tablas correctas (telemetry o alerts).

##### Tecnología Implementada:

- Framework: **FastAPI** (Python 3.11)
- ORM: **SQLAlchemy 2.0**
- Validación: **Pydantic v2**
- Algoritmo: Detector stateful con contadores por paquete

##### Endpoints:

- POST /ingest: Recibe telemetría + detecta incidentes
- GET /alerts: Consulta alertas generadas
- GET /stats: Estadísticas del sistema
- GET /health: Health check

##### Por qué Python en lugar de JavaScript:

- Mejor para algoritmos complejos (ML, análisis)
- Librerías científicas (pandas, sklearn)
- Testing más robusto (pytest)
- Tipado estático con Pydantic

##### Justificación Cloud:

FastAPI + Serverless ofrece:

- **Auto-scaling:** De 0 a 1000 requests/seg automáticamente
- **Pay-per-use:** Solo pagas por ejecuciones reales
- **Mantenimiento cero:** No gestionar servidores
- **Logging integrado:** CloudWatch, Stackdriver

| Ganancia                       | Renuncia  |
|--------------------------------|---|
| Escalado automático sin config | Cold starts (100-500ms primera request)         |
| Deployment rápido (segundos)   | Límites de memoria/tiempo (15 min máx)          |
| Menor coste operativo          | Para procesamiento muy pesado, mejor Kubernetes |



## Ficha 5: Almacenamiento Primario (PostgreSQL)

### Propósito:

Persistir **todos** los eventos de telemetría (normales y anómalos) con baja latencia de escritura, permitiendo consultas históricas y auditoría completa.

### Tecnología Implementada:

- Base de Datos: **PostgreSQL 15** en Docker
- Gestor: **pgAdmin 4**
- Tablas:
  - telemetry: Todos los eventos (150 filas en demo)
  - alerts: Solo incidentes confirmados (2 filas en demo)

### ¿Por qué PostgreSQL y no NoSQL?

| Criterio                  | PostgreSQL             | NoSQL (DynamoDB)          |
|---------------------------|------------------------|---------------------------|
| Consultas complejas       | JOIN, agregaciones SQL | Limitado                  |
| Transacciones ACID        | Sí                     | Eventualmente consistente |
| Coste en proyecto pequeño | Gratis (Docker)        | \$\$ por operación        |
| Escalabilidad horizontal  | Con sharding manual    | Nativa                    |
| Curva de aprendizaje      | Conocido               | Requiere aprender APIs    |

Para producción a gran escala, consideraríamos:

- **Time-Series DB** (TimescaleDB, InfluxDB) para telemetría
- **NoSQL** (DynamoDB, Firestore) si necesitamos writes masivos (>10k/seg)
- **Data Warehouse** (BigQuery, Redshift) para análisis histórico



### Trade-off Clave:

| Ganancia                              | Renuncia                                      |
|---------------------------------------|---|
| Consultas SQL ricas (JOIN, GROUP BY)  | Escalado horizontal complejo                  |
| Transacciones ACID garantizadas       | Throughput limitado vs NoSQL (~5k writes/seg) |
| Ecosystem maduro (herramientas, ORMs) | Requiere gestión de índices y vacuum          |

### Ficha 6: Notificación y Acción (Alertas Automáticas)

#### Propósito:

Convertir detecciones técnicas en **acciones humanas** inmediatas mediante notificaciones visuales y protocolos de emergencia.

#### Tecnología Implementada:

- **Popups visuales** (tkinter) en el simulador
- Mensajes de alerta con información contextual:
  - Temperatura: "Activar refrigeración de emergencia"
  - Choque: "Contactar conductor + preparar reemplazo"

#### En producción, integraríamos:

- **Email** (SendGrid, AWS SES): Notificaciones al equipo
- **Slack/Discord** (webhooks): Alertas en canal #operaciones
- **SMS** (Twilio): Para incidentes críticos
- **Llamadas automáticas** (PagerDuty): Escalamiento si no se responde

#### Justificación Cloud:

Webhooks y servicios gestionados de notificación:

- **Alta disponibilidad:** 99.99% uptime garantizado
- **Auditoría:** Log de cada notificación enviada
- **Rate limiting:** Evita saturación de alertas
- **Plantillas:** Personalizar mensajes por tipo de incidente



#### Trade-off Clave:

| Ganancia                                | Renuncia  |
|---|---|
| Visibilidad operativa inmediata         | Dependencia de servicios externos               |
| Despliegue rápido (webhooks)            | Latencia de sincronización (1-5 segundos)       |
| Integración con herramientas existentes | Riesgo de "fatiga de alertas" sin deduplicación |

#### Ficha 7: Dashboard y Operaciones (Visualización)

##### Propósito:

Mostrar el estado en tiempo real mediante KPIs de negocio (SLA, MTTD, % Falsos Positivos) que directivos puedan entender sin conocimiento técnico.

##### Tecnología Implementada:

- Script Python (calcular\_kpis.py) con matplotlib/seaborn
- Generación de gráficos PNG de alta resolución
- Consultas SQL agregadas sobre PostgreSQL

##### En producción, usariamos:

- **Grafana:** Dashboards en tiempo real conectados a PostgreSQL
- **Looker Studio / Power BI:** Para stakeholders no técnicos
- **Prometheus + Grafana:** Para métricas de sistema (CPU, memoria)

##### KPIs Visualizados:

1. **% de Envíos en SLA** (meta:  $\geq 95\%$ )
2. **Tiempo Medio de Detección** (meta: <30s)
3. **% de Falsos Positivos** (meta: <10%)

##### Justificación Cloud:

Herramientas de BI cloud-native:

- **Conectores gestionados:** No escribir código de integración
- **Actualización automática:** Refresh cada X segundos
- **Compartir fácil:** URL pública o embebida
- **Mobile-friendly:** Acceso desde cualquier dispositivo



### Trade-off Clave:

| Ganancia                        | Renuncia                                     |
|---------------------------------|--|
| Despliegue rápido (drag & drop) | Personalización limitada vs dashboard custom |
| Accesible para no técnicos      | Coste de licencias (Power BI, Looker)        |
| Integración nativa con cloud    | Latencia de refresh (no real-time puro)      |

### Ficha 8: Observabilidad y Seguridad (CIA)

#### Propósito:

Garantizar **Confidencialidad, Integridad y Disponibilidad** mediante validaciones multi-capa, variables de entorno y sistema de reintentos.

#### Implementaciones:

##### Confidencialidad (C):

- .env para secretos (nunca en Git)
- .gitignore configurado
- python-dotenv para cargar variables

##### Integridad (I):

- Validación en Node-RED (estructura)
- Pydantic en API (tipos y rangos)
- Constraints SQL (última defensa)

##### Disponibilidad (A):

- Reintentos con backoff exponencial
- Catch All en Node-RED
- Buffer en memoria durante caídas

#### Justificación Cloud:

Servicios gestionados ofrecen seguridad por defecto:

- **Encryption at rest:** Datos cifrados en disco
- **Encryption in transit:** TLS 1.3
- **IAM:** Control de acceso granular
- **Audit logs:** Registro de cada acción



### Trade-off Clave:

| Ganancia                                 | Renuncia                          |
|--|-----------------------------------|
| Seguridad integrada (menos código)       | Menos control fino vs self-hosted |
| Cumplimiento normativo (GDPR, SOC2)      | Vendor lock-in potencial          |
| Actualizaciones automáticas de seguridad | Confianza en el proveedor cloud   |

### Ficha 9: Prueba de Demo y Escenarios

#### Propósito:

Validar el sistema completo mediante **Boss Fight** (caída de 60s) y escenarios de incidentes controlados que demuestren resiliencia y detección correcta.

#### Escenarios Testeados:

##### 1. Incidente de Temperatura (Evento 16-21):

- 6 eventos consecutivos  $>8^{\circ}\text{C}$
- Alerta generada correctamente
- Popup visual aparece
- Datos en tabla alerts



Mostrado en  
el video

##### 2. Incidente de Choque (Evento 42-45):

- 4 eventos consecutivos  $>2.5\text{G}$  y  $>30^{\circ}$  inclinación
- Alerta de choque detectada
- Protocolo de emergencia activado

##### 3. Pico Aislado (Evento 8):

- Temperatura  $28^{\circ}\text{C}$  durante 2 segundos
- NO genera alerta (correcto)
- Marcado como "pico\_temperatura\_extremo"

##### 4. Boss Fight (Caída de PostgreSQL 60s):

- Node-RED acumula datos en buffer
- Reintentos automáticos cada 2s, 4s, 8s...
- Al recuperar: todos los datos se procesan
- **Cero pérdida de datos**



## Métricas de Éxito:

- Recall: 100% (todos los incidentes detectados)
  - Latencia: 2.3s promedio (objetivo <5s)
  - Disponibilidad: 99.9% (con reintentos)
  - Pérdida de datos: 0%
- 

## 🏆 Conclusión: Arquitectura Justificada

### ¿Por Qué Esta Arquitectura?

Nuestra arquitectura no es arbitraria. Cada decisión responde a principios de ingeniería:

1. **Separation of Concerns:** Cada componente tiene UNA responsabilidad
2. **Fail-Fast + Resilience:** Validar pronto, fallar rápido, recuperarse automáticamente
3. **Observability:** Logs en cada capa para debugging
4. **Evolvability:** Fácil cambiar un componente sin afectar otros

### Trade-offs Asumidos

| Decisión              | Beneficio         | Coste                              |
|-----------------------|-------------------|------------------------------------|
| MQTT vs HTTP directo  | Desacoplamiento   | Latencia adicional (~50ms)         |
| Node-RED vs Lambda    | Debugging visual  | Cold starts en producción          |
| PostgreSQL vs NoSQL   | Queries complejas | Escalado horizontal limitado       |
| Simulador vs Hardware | Coste cero        | No testea problemas físicos reales |

### ¿Qué Cambiaríamos en Producción?

1. **MQTT gestionado** (AWS IoT Core) con TLS + certificados
2. **Serverless Lambda** en lugar de Node-RED para auto-scaling
3. **RabbitMQ** como cola persistente en el Edge
4. **TimescaleDB** para telemetría time-series
5. **Grafana** para dashboards en tiempo real
6. **PagerDuty** para escalamiento de alertas