Universidad Nacional de Educación a Distancia

Escuela Técnica Superior de Ingeniería Informática

Universidad Complutense de Madrid

Facultad de Informática

Despliegue y validación de un controlador PID basado en eventos en una red blockchain

Carlos Caravaca Gallego

Director/a: Prof. Luis de la Torre Cubillo
Co-director/a: Prof. Jesús Chacón Sombría

Trabajo de Fin de Máster

Máster Universitario
en Ingeniería de Sistemas y Control

Junio de 2025

Agradecimientos

Deseo expresar mi sincero agradecimiento a la Universidad Nacional de Educación a Distancia (UNED) y, muy especialmente, a los profesores Luis de la Torre Cubillo y Jesús Chacón Sombría por su rigurosa dirección y valiosas orientaciones durante el desarrollo de este trabajo. Extiendo igualmente mi gratitud a mi familia, cuyo apoyo y comprensión han sido esenciales para culminar con éxito esta etapa académica.

Resumen

Este trabajo demuestra la viabilidad de ejecutar un controlador PID *event-triggered* en una red blockchain al integrar dos conceptos complementarios: (i) la contabilidad distribuida, que aporta inmutabilidad y auditoría, y (ii) el muestreo por eventos, que reduce drásticamente las transacciones al recalcular la acción de control sólo cuando la señal de salida del proceso controlado cumple una cierta condición. La lógica PID se codificó en Solidity y se desplegó sobre la *Ethereum Virtual Machine*; las entradas y salidas se gestionan mediante un oráculo Python que decide si emitir o no la transacción. La planta objeto de estudio es una caldera (configurable de primer o segundo orden), formada por el fluido y las paredes del recipiente, con pérdidas convectivas y radiativas; el instrumento de medida se modela con ruido gaussiano, y la integración del estado se lleva a cabo con un integrador RK4.

La metodología incluyó: (1) adaptación del algoritmo PID a aritmética de 18 decimales y diseño de un mecanismo *anti-windup* equivalente al de la versión en punto flotante; (2) optimización *off-chain* por *grid-search*, que arroja ganancias óptimas en función de los parámetros elegidos para la caldera y la simulación; (3) aplicación de la estrategia *send-on-delta*, seguida de un barrido de umbral que identifica el compromiso óptimo (número de eventos vs rendimiento) en función de los parámetros; (4) simulaciones de 200 s comparando el lazo periódico ($\delta \approx 0$) con el lazo *event-triggered* tanto en `eth-tester` como en la test-net Sepolia.

Para una parametrización de valores típica escogida (que se mantuvo constante durante los experimentos realizados y que está detallada en la sección de resultados), el controlador emitió 44 transacciones en muestreo por eventos frente a 199 del muestreo periódico (-78 %), manteniendo valores aceptables en tiempo de establecimiento y sobrepaso. En la test-net Sepolia el consumo total fue de 2 283 kwei (unos 8,95 USD), mientras que la versión periódica alcanzó 10 323 kwei (40,48 USD). La relación lineal entre número de eventos y gas confirma la sensibilidad a la frecuencia de disparo. La equivalencia numérica entre la versión *off-chain* y el contrato se mantuvo despreciable frente al ruido del sensor gracias al escalado fijo.

Estos resultados muestran que el control PID activado por eventos sobre blockchain es técnicamente factible y económicamente viable para lazos de baja frecuencia donde la trazabilidad multi-actor tiene valor regulatorio. Persisten desafíos, como latencia variable, volatilidad del *gas-price*, actualización segura de parámetros y robustez del oráculo, pero la migración a soluciones *layer-2* (Polygon, Arbitrum, zk-rollups) puede reducir el coste por evento a cifras competitivas, ampliando aún más la viabilidad práctica del enfoque.

Palabras clave: Blockchain, Control PID, Muestreo por eventos, Contratos inteligentes, Ethereum, Control distribuido

// Abstract

This work demonstrates the feasibility of running an *event-triggered* PID controller on a blockchain by integrating two complementary concepts: (i) distributed ledger technology, which provides immutability and auditability, and (ii) event-based sampling, which drastically reduces transactions by recalculating the control action only when the controlled-process output meets a given condition. The PID logic was coded in Solidity and deployed on the *Ethereum Virtual Machine*; inputs and outputs are handled through a Python oracle that decides whether to emit the transaction. The plant under study is a boiler (configurable as first- or second-order), comprising the fluid and vessel walls with convective and radiative losses; the measurement instrument is modelled with Gaussian noise, and state integration is performed with an RK4 integrator.

The methodology included: (1) adapting the PID algorithm to 18-decimal fixed-point arithmetic and designing an *anti-windup* mechanism equivalent to the floating-point version; (2) off-chain optimisation via *grid-search*, yielding optimal gains as a function of the chosen boiler and simulation parameters; (3) application of the *send-on-delta* strategy, followed by a threshold sweep that identifies the optimal compromise (number of events vs. performance) for the parameters; (4) 200-s simulations comparing the periodic loop ($\delta \approx 0$) with the *event-triggered* loop in both `eth-tester` and the Sepolia test-net.

For a typical parameter set (that was kept constant during the experiments performed and that is detailed in the Results section), the controller issued 44 transactions under event sampling versus 199 under periodic sampling (-78%), while maintaining acceptable settling time and overshoot. On the Sepolia test-net the total consumption was 2 283 kwei (about $8.95 USD), whereas the periodic version reached 10 323 kwei ($40.48 USD). The linear relationship between event count and gas confirms sensitivity to trigger frequency. Numerical equivalence between the *off-chain* version and the contract remained negligible relative to sensor noise thanks to fixed scaling.

These results show that event-triggered PID control on blockchain is technically feasible and economically viable for low-frequency loops in which multi-actor traceability has regulatory value. Challenges remain, such as variable latency, gas-price volatility, secure parameter updates, and oracle robustness, but migration to *layer-2* solutions (Polygon, Arbitrum, zk-rollups) can reduce per-event cost to competitive figures, further expanding the practical viability of the approach.

Keywords: Blockchain, PID Control, Event-Triggered Sampling, Smart Contracts, Ethereum, Distributed Control

Glosario

Anti-windup Técnica que evita la integración excesiva del término I en control PID cuando el actuador satura.

Arbitrum Red *layer-2* basada en *optimistic rollups* para escalar Ethereum y abaratar las comisiones.

CPU Unidad Central de Procesamiento; componente principal de cómputo.

ETH Unidad de cuenta nativa de la red Ethereum, utilizada para pagar comisiones de gas.

EVM Máquina virtual replicada en todos los nodos Ethereum donde se ejecutan los contratos inteligentes.

Fixed-point Representación numérica entera que almacena fracciones usando una escala fija; necesaria en la EVM.

Gas Cuota de cómputo que debe pagarse para ejecutar operaciones en la EVM.

Gwei Unidad monetaria igual a 10^{-9} ETH; se usa para cotizar el precio de gas.

IoT Internet de las Cosas; red de objetos físicos con capacidad de comunicación y sensado/actuación.

JSON-RPC Protocolo liviano para llamadas remotas a métodos, usado para interactuar con nodos blockchain.

Layer 2 / L2 Solución de segunda capa sobre una blockchain principal (L1) que mejora rendimiento y reduce costes.

MQTT Protocolo de mensajería publish/subscribe ligero, habitual en IoT.

NCS Sistemas de Control en Red; lazos de control con componentes conectados por redes de comunicación.

Oráculo Componente que conecta el mundo externo con un contrato inteligente, inyectando datos o accionando eventos.

PID Controlador Proporcional-Integral-Derivativo; regula procesos mediante la combinación de tres acciones.

PLC Controlador Lógico Programable; dispositivo industrial para control en tiempo real.

PLCBlox Marco experimental que integra PLCs con blockchain para supervisión segura.

Polygon Ecosistema de cadenas auxiliares y soluciones de escalado (PoS chain, zk-EVM, etc.) compatible con Ethereum.

SCADA Sistema de Supervisión, Control y Adquisición de Datos; arquitectura centralizada de control industrial.

Send-on-delta Estrategia de control que envía una actualización solo cuando el error supera un umbral δ.

Sepolia Red de pruebas pública de Ethereum empleada para validar contratos antes de desplegarlos en mainnet.

Solidity Lenguaje de programación de alto nivel para escribir contratos inteligentes en Ethereum.

TPS Transacciones por segundo; medida de capacidad de una red blockchain.

Zk-rollup Mecanismo de agregación de transacciones que emplea pruebas de conocimiento-cero para reducir costes y aumentar el rendimiento en *layer-2*.

Índice general

Glosario IX

1. Introducción **1**
- 1.1. Motivación . . . 1
- 1.2. Propuesta y objetivos . . . 1
- 1.3. Estructura del documento . . . 2

2. Estado del Arte **3**
- 2.1. Blockchain en el Control de Procesos Industriales . . . 4
  - 2.1.1. Aplicabilidad y beneficios en sistemas de control . . . 4
  - 2.1.2. Retos y limitaciones del control sobre blockchain . . . 5
- 2.2. Controladores PID basados en eventos . . . 7
  - 2.2.1. Lógica y fundamentos del control basado en eventos . . . 7
  - 2.2.2. Ventajas frente al muestreo periódico . . . 9
  - 2.2.3. Integración con sistemas distribuidos e IoT . . . 10
- 2.3. Implementación de controladores PID basados en eventos en Blockchain . . . 12
  - 2.3.1. Contratos inteligentes en Solidity y adecuación del EVM para control 12
  - 2.3.2. Arquitectura de integración: sensores, oráculos y actuadores . . . 14
  - 2.3.3. Validación en testnets y simulación . . . 17
- 2.4. Resumen del estado del arte . . . 19

3. Materiales y métodos **23**
- 3.1. Entorno experimental . . . 23
  - 3.1.1. Pila de software . . . 23
  - 3.1.2. Nodo blockchain y compilación de contratos . . . 24
  - 3.1.3. Simulación del proceso y lógica off-chain . . . 25
  - 3.1.4. Interfaz de usuario . . . 25
- 3.2. Modelo del sistema . . . 25
  - 3.2.1. Parámetros nominales . . . 27
  - 3.2.2. Método de integración . . . 28
  - 3.2.3. Medición y ruido . . . 28

3.2.4. Casos límite y saturación 29
3.2.5. Motivación de la elección 29
3.3. Diseño y optimización del PID 30
3.3.1. Formulación discreta y anti-windup 30
3.3.2. Disparo por eventos 31
3.3.3. Métrica de optimización 31
3.3.4. Procedimiento de *grid-search* 32
3.3.5. Barrido de umbral 32
3.3.6. Resumen de la metodología 33
3.4. Implementación del contrato inteligente 33
3.4.1. Punto fijo y almacenamiento compacto 33
3.4.2. Constructor inmutable 33
3.4.3. Algoritmo `update()` 35
3.4.4. Determinismo y atomicidad 35
3.4.5. Compilación y despliegue 35
3.4.6. Equivalencia con el PID *off-chain* 36
3.5. Desarrollo de la interfaz web 36
3.5.1. Estructura y flujo de trabajo 37
3.5.2. Diseño de la experiencia de usuario 38
3.5.3. Estado actual y posibles extensiones 39

4. Resultados **41**
4.1. Configuración de las pruebas 41
4.1.1. Red blockchain y medición de gas 41
4.1.2. Parámetros de simulación 42
4.1.3. Estrategia de lanzamiento 42
4.2. Pruebas unitarias y validación de la implementación 43
4.2.1. Contrato inteligente 43
4.2.2. Modelo del sistema y PID 43
4.3. Pruebas de integración 45
4.3.1. Protocolo 45
4.3.2. Resultados 45
4.4. Pruebas funcionales 46
4.4.1. Eficiencia del muestreo basado en eventos 46
4.4.2. Coste (gas) y rendimiento del controlador 48
4.4.3. Comparación visual 48
4.5. Análisis de sensibilidad y amenazas a la validez 49
4.5.1. Latencia y variabilidad del tiempo de bloque 50
4.5.2. Congestión y picos de *gas price* 50

4.5.3. Sensor, ruido y cuantización 51
4.5.4. Modelo de planta y variabilidad del proceso 51
4.5.5. Fiabilidad del oráculo y del nodo 52
4.5.6. Amenazas a la validez interna y externa 53
4.6. Discusión de los resultados 54
4.6.1. Ventajas observadas 54
4.6.2. Limitaciones constatadas 54
4.6.3. Comparación con la literatura previa 55
4.6.4. Implicaciones para aplicaciones industriales 56
4.6.5. Líneas de mejora 56

5. Conclusiones y trabajos futuros **59**
5.1. Conclusiones 59
5.2. Trabajos futuros 60

Bibliografía y referencias **61**

A. Smart Contrats **63**
A.1. Código completo de `EventPID.sol` 63

B. Código fuente de la aplicación **67**
B.1. Script de entrada Python/Streamlit 67

Índice de figuras

2.1. Diagrama de flujo de comunicación. 16

3.1. Diagrama del modelo del sistema 27
3.2. Diagrama de flujo de la metodología 34
3.3. Interfaz web para la simulación y validación del PID sobre blockchain. 37
3.4. Interfaz web: visualización de resultados. 38

4.1. Ejecución *off-chain*. 44
4.2. Ejecución *on-chain*. 45
4.3. Barrido de umbral: Eventos, Puntuación y Índice 47
4.4. Control periódico 49
4.5. Control *event-triggered* relajado 49

Índice de tablas

3.1. Herramientas y librerías principales utilizadas en los experimentos. 23
3.2. Parámetros nominales de los modelos de caldera. 28

4.1. Entornos utilizados en la validación *on-chain*. 41
4.2. Equivalencia numérica entre la referencia *off-chain* y el contrato inteligente. . 44
4.3. Pruebas de integración 45
4.4. Comparación de control óptimo frente a extremos 48

Capítulo 1

Introducción

1.1. Motivación

La convergencia entre **tecnologías de control clásico** y **tecnologías distribuidas emergentes** está redefiniendo la forma en que las plantas industriales, las microrredes y los sistemas ciber-físicos gestionan su operación diaria. En particular, dos tendencias han cobrado gran relevancia en la última década:

1. La *blockchain*, concebida originalmente como infraestructura financiera descentralizada, pero ahora explorada como medio para dotar de *transparencia, inmutabilidad y confianza* a entornos multi-agente.

2. El *control basado en eventos*, que optimiza el uso de recursos de cómputo y red al recalcular la acción de control únicamente cuando determinadas condiciones de la planta lo justifican.

Integrar ambas tendencias ofrece ventajas potenciales —auditoría inviolable de las acciones de control, eliminación de puntos únicos de fallo y reducción drástica del tráfico de red— pero también introduce retos: latencias de confirmación, aritmética entera en contratos inteligentes y dependencia de *oráculos* para acceder al mundo físico. El presente trabajo nace de la necesidad de **estudiar la viabilidad práctica** de esta integración en un escenario representativo pero acotado: el control de la temperatura de una caldera mediante un *controlador PID activado por eventos* implementado en Solidity y desplegado sobre una red de pruebas Ethereum.

1.2. Propuesta y objetivos

El proyecto propone **diseñar, desplegar y validar** un lazo de control donde la lógica PID resida dentro de un contrato inteligente y las decisiones de control se desencadenen sólo cuando se cumpla una condición de evento predeterminada. Los objetivos concretos son:

- **Diseño del lazo de control**: Definir la estrategia de disparo por eventos, seleccionar los umbrales adecuados y adaptar el algoritmo PID a aritmética entera de la EVM.
- **Implementación on-chain**: Codificar el controlador en Solidity, incluyendo mecanismos de protección (control de acceso, histéresis temporal y actualización segura de ganancias).
- **Integración con la planta**: Desarrollar los oráculos de sensado y actuación en Python para comunicar la caldera simulada con el contrato inteligente.
- **Validación experimental**: Ejecutar una campaña de pruebas en *testnet* (Sepolia) y en una cadena privada e identificar los límites de desempeño (retardos, coste de gas, estabilidad).
- **Comparación crítica**: Contrastar el enfoque propuesto frente a una arquitectura SCADA clásica y frente a un lazo PID periódico estándar, cuantificando ahorro de transacciones, robustez y trazabilidad.

El logro de estos objetivos permitirá responder a la pregunta central: *¿Hasta qué punto es técnicamente y económicamente viable implementar controladores PID activados por eventos en una red blockchain y qué beneficios tangibles aporta respecto a las soluciones tradicionales?*

1.3. Estructura del documento

El documento se ha organizado para guiar al lector desde los fundamentos teóricos hasta los resultados experimentales: el **Capítulo 1** introduce la motivación, define los objetivos y explica la organización global; el **Capítulo 2** revisa el estado del arte sobre blockchain en control de procesos y sobre controladores PID basados en eventos, identificando las lagunas que aborda esta investigación; el **Capítulo 3** detalla los materiales y métodos, describiendo la implementación del contrato inteligente, la arquitectura de oráculos y la metodología de validación; el **Capítulo 4** presenta y discute los resultados obtenidos, comparando desempeño, latencia y costes con enfoques tradicionales; por último, el **Capítulo 5** resume las conclusiones, señala las limitaciones y plantea líneas de trabajo futuro, seguido de la bibliografía que sustenta el estudio. De este modo, cada capítulo puede leerse de forma independiente, pero en conjunto ofrece una narrativa completa que evidencia la pertinencia y el aporte original del proyecto.

Capítulo 2

Estado del Arte

El presente capítulo expone el estado del arte relacionado con el despliegue y validación de controladores PID basados en eventos en redes blockchain. En los últimos años, ha surgido interés en aprovechar las *blockchains* en sistemas de control industrial y ciber-físicos debido a sus propiedades de **descentralización, inmutabilidad y seguridad**. A su vez, los controladores PID basados en eventos representan una alternativa novedosa al muestreo periódico tradicional, activando el cálculo de la acción de control sólo cuando ocurren ciertos eventos o condiciones. La combinación de ambas tecnologías (control basado en eventos y plataformas blockchain) promete sistemas de control distribuidos más confiables y transparentes, pero conlleva desafíos importantes en cuanto a rendimiento en tiempo real e integración.

En este capítulo se revisan, en primer lugar, las aplicaciones de la tecnología blockchain en el control de procesos industriales, junto con sus beneficios potenciales, retos y trabajos previos relevantes. En segundo lugar, se abordan los controladores PID basados en eventos, detallando su lógica de operación y ventajas frente al muestreo periódico, así como su compatibilidad con entornos distribuidos típicos del *Internet of Things* (IoT) industrial. Posteriormente, se analiza cómo se pueden implementar controladores de este tipo sobre una plataforma blockchain mediante contratos inteligentes (ej. en Solidity sobre la Ethereum Virtual Machine), discutiendo la idoneidad de estas herramientas para ejecutar lógica de control y las consideraciones prácticas (precisión numérica, latencia de bloque, oráculos, etc.). Asimismo, se incluye una comparación con las tecnologías tradicionales de control distribuido, como SCADA e IoT basado en la nube, resaltando diferencias en arquitectura, rendimiento y seguridad. Por último, se explora la viabilidad de usar redes de prueba (testnets) de blockchain y entornos de simulación (en Python, Javascript, MATLAB, etc.) para prototipar y validar estos sistemas de control antes de su despliegue real.

2.1. Blockchain en el Control de Procesos Industriales

Las *blockchains* se concibieron originalmente para aplicaciones financieras (como Bitcoin), pero sus propiedades únicas han motivado su adopción en otros dominios, incluyendo la automatización industrial y los sistemas de control. Una blockchain es esencialmente un libro mayor distribuido y seguro, mantenido por una red de nodos que acuerdan mediante consenso las transacciones registradas en bloques encadenados criptográficamente [Mao and Xiao, 2018]. Además, mediante contratos inteligentes, una blockchain como Ethereum permite ejecutar código de forma autónoma y transparente dentro de la red, lo que abre la puerta a coordinar dispositivos físicos y lógicas de control sin depender de un servidor central. A continuación, se discuten las motivaciones para usar blockchain en control de procesos, sus posibles ventajas, y las limitaciones o desafíos identificados.

2.1.1. Aplicabilidad y beneficios en sistemas de control

En un sistema de control industrial típico (por ejemplo, una planta manufacturera o una red eléctrica), suele haber jerarquías centralizadas de supervisión (sistemas SCADA, PLCs, servidores) que recogen datos de sensores y envían comandos a actuadores. La introducción de una blockchain en este contexto permite **descentralizar la toma de decisiones** y el registro de eventos, eliminando la necesidad de confiar en una única entidad o infraestructura. Diversos autores han propuesto que las propiedades de la blockchain pueden aumentar la confiabilidad, seguridad, eficiencia y reducir costos en sistemas de control industrial. Por ejemplo, Mao and Xiao [2018] argumentan que una blockchain podría mejorar la **resiliencia y ciberseguridad** en redes de control industrial al eliminar puntos únicos de falla y ofrecer un registro inviolable de todas las operaciones.

Un beneficio clave de usar blockchain es la **inmutabilidad y trazabilidad** de los datos: las mediciones de sensores, comandos enviados y otros eventos de control pueden almacenarse en la cadena de bloques de forma que queden auditables y a prueba de manipulación. Esto es especialmente atractivo para industrias con altos requisitos regulatorios o de seguridad, donde un registro confiable de las acciones de control es fundamental (p. ej., industria energética, farmacéutica, nuclear). Augello et al. [2022] destacan que integrar blockchain en sistemas SCADA ofrece **transparencia, trazabilidad y responsabilidad**, contrastando con los sistemas tradicionales donde los datos se recopilan de forma centralizada en entornos confiables pero opacos. Un contrato inteligente en la blockchain puede actuar como *juez* distribuido que verifica condiciones y autoriza acciones de control sólo si las reglas acordadas se cumplen, aumentando la confianza entre múltiples partes.

Otra ventaja es la **resistencia a fallos y ataques**. La replicación de los datos de control en todos los nodos de la red blockchain garantiza que, aunque falle un nodo o haya intentos de ataque (p. ej. ransomware en un servidor SCADA), el sistema global pueda continuar operando o al menos recuperar su estado rápidamente. Una arquitectura de control basada

en blockchain podría seguir operativa incluso ante ataques o caídas de servidores, ya que la lógica de control y los datos están distribuidos en la red. Asimismo, las técnicas criptográficas de la blockchain (firmas digitales, hashes encadenados, algoritmos de consenso) proporcionan autenticación y no-repudio de las órdenes de control. Esto significa que cualquier comando enviado a un actuador puede verificarse como legítimo y originado por la entidad autorizada, resolviendo en parte problemas de autenticación/autorización de dispositivos IoT [Short et al., 2024]. Por ejemplo, en PLCBlox (un marco experimental que integra PLCs con blockchain) el contrato inteligente asegura que sólo un operador autenticado con su monedero digital pueda cambiar un setpoint, y todos esos cambios quedan registrados de forma permanente. Esto dificulta que un atacante pueda ocultar sus acciones o evitar ser rastreado, pues la blockchain actúa como bitácora inmutable de todas las operaciones.

Finalmente, blockchain puede facilitar la colaboración entre múltiples organizaciones en el control de un proceso. En escenarios como microgrids o consorcios industriales, varios agentes (empresas, usuarios) comparten infraestructura de control. Una red blockchain permite que todos ellos verifiquen y acuerden las decisiones de control sin tener que ceder el control a un ente central de confianza. Por ejemplo, en redes eléctricas inteligentes se han propuesto blockchains consorciadas donde generadores y consumidores almacenan medidas de frecuencia y deciden ajustes de forma descentralizada para estabilizar el sistema [Bin Masood et al., 2019]. En suma, la aplicabilidad de blockchain en control reside en mejorar la confianza, transparencia y resiliencia de sistemas distribuidos, especialmente cuando intervienen múltiples partes interesadas o cuando la seguridad de los datos de control es crítica.

2.1.2. Retos y limitaciones del control sobre blockchain

A pesar de sus beneficios teóricos, la adopción de blockchain en lazo de control conlleva desafíos importantes. El primero es la latencia y frecuencia de actualización. Las blockchains públicas típicas (como Ethereum) tienen tiempos de bloque del orden de segundos (por ejemplo, 12-15 s en Ethereum mainnet), lo cual es órdenes de magnitud más lento que los ciclos de control tradicionales que operan en milisegundos. Incluso en blockchains más rápidas o redes permisadas, la sobrecarga de consenso introduce retardos no deterministas. Bin Masood et al. [2019] estudiaron un sistema de control de frecuencia en microrred eléctrica implementado sobre Ethereum y midieron delays de unos pocos segundos entre la captura de datos y la ejecución de la acción de control. Concluyen que estas demoras son aceptables sólo para funciones de control lento, como el control secundario de frecuencia en microrredes, y no causan inestabilidad en ese contexto. Sin embargo, para lazos de control más rápidos (por ejemplo, control de posición de un motor en tiempo real), los retrasos de varios segundos serían inasumibles. Este problema puede mitigarse parcialmente usando blockchains privadas o de propósito específico con tiempos de bloque más bajos (incluso sub-segundo), o algoritmos

de consenso más rápidos. Por ejemplo, el uso de Solana (blockchain de alta velocidad) en una celda de manufactura permitió tiempos de respuesta más acotados; en un experimento [Short et al., 2024] se implementó la lógica de control de un robot industrial en un contrato de Solana y se observaron tiempos variables de ejecución en el orden de cientos de milisegundos, influenciados por la latencia de red y la confirmación de transacciones. Si bien estos tiempos eran mayores que en control local, un análisis de costo-beneficio mostró que el desempeño era aceptable para ofrecer un servicio de control confiable de terceros sobre Solana. En resumen, la escalabilidad temporal es un reto: la blockchain impone un límite en la rapidez con que se pueden muestrear sensores y aplicar actuadores. Esto restringe su uso a procesos inherentemente lentos o control supervisorio, a menos que se utilicen arquitecturas híbridas (ej. control local rápido con sincronización periódica en blockchain para coherencia global).

Otro desafío es la integración con el mundo físico, ya que las blockchains por sí solas no tienen acceso directo a sensores ni actuadores. Se requiere de componentes externos llamados oráculos para introducir la información del entorno en la cadena y para llevar las salidas de control desde la cadena hasta los actuadores reales. Esta capa de integración puede introducir vulnerabilidades o puntos de fallo si no se diseña adecuadamente. En el ejemplo de PLCBlox, se desarrolló un gateway software para conectar un PLC (vía OPC UA) con la blockchain, de forma que el PLC pudiera consultar periódicamente los datos/órdenes almacenados en la cadena. Esta arquitectura invierte el modelo tradicional: en vez de que un sistema central escriba en el PLC, el propio PLC lee de una fuente segura (la blockchain). La desventaja es la complejidad añadida y la necesidad de programar el PLC para ese rol, pero la ventaja es que se puede deshabilitar la escritura remota directa en el PLC, reduciendo la superficie de ataque. En general, cualquier solución de este tipo debe asegurar que los oráculos (ya sean gateways industriales, scripts IoT o servicios tipo Chainlink) sean confiables, pues si un oráculo es comprometido, la seguridad de la lógica on-chain se ve socavada.

Adicionalmente, existen limitaciones técnicas en los contratos inteligentes para la lógica de control. Lenguajes como Solidity (en Ethereum) no manejan números de coma flotante ni ofrecen fácilmente aritmética en tiempo continuo, lo que obliga a implementar los cálculos de control en aritmética entera o de punto fijo con cuidado en la precisión. La EVM (Ethereum Virtual Machine) impone un costo computacional (gas) por cada instrucción, desincentivando algoritmos demasiado complejos o bucles extensos. Un controlador PID es algorítmicamente sencillo, pero si la frecuencia de activación es alta, el costo por transacciones repetitivas puede ser prohibitivo en redes públicas. Por ello, en muchos casos se opta por blockchains privadas o consorciadas para aplicaciones industriales, donde los costos de transacción pueden ignorarse o parametrizarse. Otro punto es la deterministicidad: el contrato debe producir resultados deterministas en todos los nodos, sin depender de fuentes externas de aleatoriedad o datos volátiles. Esto encaja bien con la lógica de un controlador (que es determinista), pero implica que no se puede, por ejemplo, invocar directamente un sensor desde el contrato; siempre deberá recibirse el dato como parámetro de una transacción ya consensuada.

Finalmente, la adopción de blockchain enfrenta barreras prácticas en entornos industriales conservadores: interoperabilidad con sistemas legados, falta de estandarización, requerimientos regulatorios (p.ej., cumplimiento de estándares como ISA/IEC si se altera la arquitectura de control), y preocupaciones de seguridad (ironía: introducir un nodo blockchain requiere abrir ciertos puertos/redes, lo cual en entornos OT tradicionales muy aislados puede verse como aumento de superficie de ataque). Aunque las investigaciones iniciales demuestran factibilidad en laboratorio, pasar a implementaciones a gran escala requerirá superar estos obstáculos técnicos y culturales.

2.2. Controladores PID basados en eventos

Tradicionalmente, los controladores digitales operan con muestreo periódico: el sensor es leído a intervalos fijos (por ejemplo cada T segundos) y el controlador (PID u otro) calcula la acción de control en cada intervalo, enviándola al actuador. En contraposición, un controlador basado en eventos no actúa en intervalos regulares de tiempo, sino que depende de la ocurrencia de ciertos eventos para decidir cuándo calcular una nueva acción de control. En otras palabras, es el suceso de un evento, en lugar del paso constante del tiempo, lo que determina el instante de muestreo y actualización del control [Dormido et al., 2008]. Por ejemplo, un evento puede ser que la diferencia entre la variable medida y su último valor muestreado supere un umbral; sólo entonces se dispara el cálculo controlador, permaneciendo inactivo mientras el sistema esté aproximadamente en régimen.

La idea de control activado por eventos surgió para reducir el uso de recursos en sistemas embebidos y de control en red. Si el sistema no está cambiando significativamente, ¿para qué gastar CPU y ancho de banda recomputando una salida de control prácticamente igual a la anterior? En un PID basado en eventos, la lógica espera hasta detectar una condición significativa (evento) para actualizar la señal de control, logrando en muchos casos un desempeño similar al control periódico pero con muchas menos ejecuciones [Årzén, 1999]. En esta sección se describen los fundamentos de los controladores PID event-driven, sus ventajas frente al muestreo periódico, y consideraciones para integrarlos en sistemas distribuidos (redes de sensores y actuadores). Cabe mencionar que este enfoque también se denomina control disparado por eventos o *Event-Triggered Control* en la literatura anglosajona.

2.2.1. Lógica y fundamentos del control basado en eventos

Un controlador PID (Proporcional-Integral-Derivativo) clásico calcula su acción $u(t)$ en función del error $e(t)$ (diferencia entre el setpoint y la medida) y sus componentes integrales/derivativas. En un sistema de muestreo periódico discreto con período h, esto sucede en cada paso k. En cambio, en un controlador PID basado en eventos, existe un mecanismo de detección de eventos que monitorea ciertas señales del sistema en tiempo continuo o a alta

frecuencia, y decide cuándo requerir una actualización del PID. Este mecanismo típicamente impone una o varias condiciones lógicas. Un esquema sencillo propuesto por Årzén (1999) (pionero en este campo) utiliza como condición principal la variación de la señal medida: "ejecutar el PID cuando el cambio en la medición y exceda un cierto límite predefinido". En otras palabras, si la planta está tranquila e y no ha cambiado lo suficiente desde la última corrección, el controlador permanece inactivo; pero si y fluctúa más allá de un umbral (indicando una perturbación o error crecientes), se genera un evento que activa inmediatamente el cálculo $u(t) = K_p e(t) + K_i \int e(t) + K_d \frac{de(t)}{dt}$, actualizando la salida. Alternativamente, se pueden usar otras señales para el evento, como el propio error $e(t)$ del PID en vez de la salida y. Es común emplear medidas relativas (porcentaje de cambio) en vez de absolutas, para normalizar la condición. Además, suele introducirse una salvaguarda de tiempo máximo entre eventos: si pasó demasiado tiempo desde la última acción de control, se fuerza una actualización aunque no se haya cruzado el umbral, para garantizar estabilidad [Miguel-Escrig et al., 2017]. Esto previene quedarse congelado indefinidamente ante cambios muy pequeños acumulativos. Por otro lado, se suele considerar un tiempo mínimo entre eventos, para evitar disparos sucesivos demasiado próximos que puedan saturar la red de comunicación o provocar *chattering*.

Implementar un controlador PID basado en eventos requiere manejar la variación del período de muestreo. A diferencia del PID periódico que asume un h fijo, el PID event-driven debe considerar en su algoritmo el tiempo h_k transcurrido desde la última ejecución (que puede cambiar en cada evento). Esto se resuelve informando al controlador el valor de h_k en cada activación para que pueda compensar el efecto en la integración y derivación. Por ejemplo, el integrador del PID puede acumular el error multiplicado por h_k (rectángulo de área variable), o el término derivativo dividir la variación de error entre h_k. Årzén propuso enviar al algoritmo PID, junto con la orden de ejecución, el valor del intervalo reciente (h_{recent}) para ajustar los cálculos. De esta forma, el PID event-driven puede lograr un desempeño cercano al de un PID periódico con h promedio, incluso con espaciados irregulares. Es importante definir reglas de evento con histéresis para evitar oscilaciones por disparos excesivamente frecuentes (por ejemplo, usando umbrales diferentes para activación y desactivación, o bloquear eventos muy seguidos).

En resumen, la lógica de un PID basado en eventos combina un detector de eventos (que monitorea condiciones sobre error, salida u otra métrica) y el controlador PID discreto adaptado que se invoca a demanda. Esta arquitectura a veces se describe como cliente-servidor: el detector actúa de cliente solicitando servicio (cálculo de control) al servidor PID cuando lo requiere. Entre eventos, el actuador típicamente mantiene la última salida u constante (o la planta se controla en hold).

2.2.2. Ventajas frente al muestreo periódico

El control basado en eventos ofrece varias ventajas importantes en ciertos escenarios:

- **Uso eficiente de recursos de cómputo y comunicación**: Dado que el controlador no recalcula continuamente cuando el sistema está en calma, se pueden lograr grandes reducciones en la utilización de la CPU y en el tráfico de red, sin degradar el rendimiento significativamente [Coutinho et al., 2023]. Por ejemplo, Årzén reportó reducciones drásticas en la carga de CPU (tiempo de cálculo) de un PID event-driven comparado con su equivalente periódico, con apenas una leve pérdida de desempeño. En sistemas de control en red (NCS) o IoT industrial, esto se traduce en menos paquetes enviados por los sensores y actuadores, aliviando el ancho de banda. Estudios en control distribuido confirman que las estrategias event-triggered disminuyen el consumo de recursos de comunicación y alivian la congestión en la red, al evitar transmisiones redundantes. En redes inalámbricas de sensores/actuadores, esto también prolonga la vida de baterías al transmitir sólo cuando es necesario.

- **Respuesta rápida a perturbaciones significativas**: Un controlador basado en eventos adapta efectivamente su frecuencia de actuación a la dinámica del sistema. Si ocurre una perturbación grande o un cambio rápido en la referencia, los eventos se dispararán con mayor frecuencia (incluso más rápido que el período fijo, si así lo permiten los sensores), dando una respuesta temprana a situaciones críticas. Contrariamente, en control periódico se está limitado por el período fijo; aunque el error crezca rápidamente, el controlador no actuará hasta el siguiente tick. En cambio, en event-driven, un gran error puede disparar un control inmediato sin esperar, reduciendo potencialmente el sobreshoot o tiempo de asentamiento. Esta adaptatividad temporal es deseable en sistemas con comportamientos esporádicos: reacciona rápido cuando se necesita y ahorra esfuerzo cuando no.

- **Menor demanda de ancho de banda determinista**: En algunas aplicaciones, múltiples lazos de control comparten una misma red (p. ej. buses de campo). Usando muestreo periódico sincronizado, el peor caso es que todos envían mensajes cada h, pudiendo saturar la red si h es pequeño. Con control basado en eventos, las transmisiones se distribuyen en el tiempo según la necesidad de cada lazo. Se ha demostrado que esto reduce la probabilidad de colisiones o retrasos en comunicaciones críticas, ya que estadísticamente es poco probable que todos los lazos disparen eventos simultáneamente a alta tasa [Guo et al., 2023]. Así, se mejora la calidad de servicio de la red y se pueden integrar más lazos sin aumentar el ancho de banda nominal.

- **Ahorro de energía**: En dispositivos con capacidad de dormir (sensores inalámbricos, controladores alimentados por batería), el muestreo y transmisión bajo demanda permiten que el dispositivo permanezca en modo de bajo consumo la mayor parte del

tiempo, despertando sólo para eventos relevantes. Esto extiende la autonomía y reduce costos operativos en IoT. Por ejemplo, un sensor de temperatura en una fábrica podría reportar al controlador sólo cuando la temperatura varía ±1°C, en vez de cada segundo inútilmente cuando la temperatura es estable.

- **Mantenimiento de rendimiento aceptable**: Quizá lo más sorprendente es que, bien diseñados, los controladores basados en eventos pueden lograr desempeño casi equivalente al periódico. Investigaciones han encontrado que ajustando apropiadamente los umbrales de evento y compensando la variación de período, las métricas de control (tiempos de subida, sobrepico, error en régimen) se mantienen dentro de rangos tolerables respecto al caso periódico. Gomes da Silva et al. introdujeron mejoras al esquema de Årzén para garantizar cotas de estabilidad incluso con tiempos variables entre eventos, asegurando que el control event-driven no degrade la estabilidad del lazo [da Silva et al., 2014]. En suma, se obtiene "lo mejor de ambos mundos": similar calidad de control pero con mucho menos uso de recursos promedio.

Por supuesto, estas ventajas vienen con compromisos. Existe cierta complejidad en el diseño (hallar las condiciones de evento óptimas no es trivial y puede requerir análisis teóricos de estabilidad basados en sistemas muestreados no uniformes). Asimismo, en el peor caso (muchos eventos en corto tiempo), la carga de CPU/red podría acercarse a la de un sistema periódico rápido. Sin embargo, numerosos estudios en control distribuido señalan que, en condiciones normales, la reducción de comunicaciones es sustancial. Por ello, el control basado en eventos se considera altamente prometedor en entornos con recursos limitados (como redes inalámbricas industriales, *sensor nodes*, etc.). En definitiva, el control basado en eventos es una alternativa muy prometedora particularmente cuando se consideran sistemas con capacidades reducidas de computación y comunicación.

2.2.3. Integración con sistemas distribuidos e IoT

Los controladores event-driven encajan de manera natural en arquitecturas distribuidas y IoT, donde la latencia y el ancho de banda son consideraciones cruciales. En un escenario clásico de control distribuido, un sensor remoto envía sus datos vía una red (Ethernet, inalámbrica) a un controlador central, que procesa y envía comandos a un actuador remoto. Reemplazar el muestreo periódico por uno basado en eventos implica que el sensor enviará paquetes de datos sólo cuando detecte una variación importante, reduciendo el tráfico en la red compartida. Esto es especialmente útil en IoT industrial, donde decenas de dispositivos podrían saturar un servidor central si reportan constantemente a alta frecuencia.

Los esquemas event-driven también mejoran la robustez frente a pérdidas de paquetes o latencias variables. Si la red sufre congestión y un paquete de sensor se retrasa, en un sistema periódico eso puede significar un control obsoleto (usando datos viejos hasta la

siguiente muestra). En cambio, en event-driven, tras superar la congestión, el sensor enviará un nuevo evento si el error sigue presente, recuperando la actualidad. Además, al enviar menos paquetes, se reduce la probabilidad de pérdidas en primer lugar. Algunos autores han extendido estos conceptos a control descentralizado colaborativo (multi-agente): por ejemplo, en control de formaciones de robots, utilizar disparo por eventos en cada robot para enviar su estado al vecino sólo cuando cambia en cierta magnitud. Esto mantuvo la cohesión de la formación al tiempo que bajó dramáticamente la comunicación entre robots [Guo et al., 2023].

En entornos IoT y cloud, el control basado en eventos puede implementarse mediante arquitecturas publish/subscribe donde los dispositivos publican mensajes en un tópico sólo al ocurrir eventos. Servicios en la nube procesan esos eventos (por ejemplo, un lambda que ejecuta el control) y devuelven comandos a los actuadores subscritos. Tal arquitectura es escalable y asíncrona, aprovechando el event-driven de punta a punta.

Al combinar control event-driven con blockchain (tema central de este trabajo), encontramos que la naturaleza es sinérgica: la blockchain opera de forma asíncrona y por transacciones/eventos, no en tiempo continuo. Por tanto, un controlador que sólo necesita actuar cuando hay cambios relevantes se adapta mejor a la cadencia lenta de bloques de una cadena. En un lazo de control sobre blockchain (Capítulo 3), típicamente un sensor actuará como oráculo enviando una transacción sólo cuando detecte un evento. Esta transacción despierta al contrato inteligente controlador, que computa la nueva acción. Si el sensor estuviera enviando cada 10 ms datos a la blockchain, claramente la mayoría se desperdiciarían debido a la baja frecuencia de bloques. El muestreo basado en eventos minimiza las transacciones on-chain necesarias, volviendo factible el control en blockchain. Como indicaron Bin Masood et al. [2019], "las señales de control calculadas por los contratos convergen al valor nominal esperado" en su co-simulación, mientras los delays se mantuvieron de orden segundos, rango manejable gracias a que los eventos eran poco frecuentes (control secundario). Por otro lado, la variabilidad de intervalos en event-driven exige robustez: en blockchain los intervalos dependerán de la congestión de la red y tiempos de bloque, que pueden no ser constantes. Afortunadamente, los controladores event-driven ya están diseñados para lidiar con intervalos variables (como se explicó antes), por lo que esta tolerancia es una ventaja en sí misma cuando se usa blockchain como medio de comunicación.

En resumen, los controladores PID basados en eventos representan una evolución en el paradigma de control digital, optimizando el uso de recursos y adecuándose mejor a los sistemas distribuidos modernos. Su adopción en entornos con conectividad limitada o costo por mensaje (como una blockchain pública) resulta prácticamente necesaria para viabilizar el lazo de control. A continuación, en la Sección 4, veremos cómo se implementa en la práctica un controlador basado en eventos dentro de una red blockchain, atendiendo a los detalles de los contratos inteligentes (Solidity/EVM), orquestación con oráculos y pruebas en redes de prueba.

2.3. Implementación de controladores PID basados en eventos en Blockchain

Tras revisar los fundamentos de blockchain y de control basado en eventos por separado, enfocamos ahora su combinación práctica: ¿Cómo desplegar un controlador PID event-driven en una red blockchain?. Este apartado aborda los aspectos de implementación, desde la elección de la plataforma y lenguajes de smart contracts, hasta la arquitectura software/hardware necesaria para integrar los dispositivos físicos. También se discuten las herramientas de desarrollo y pruebas (testnets, simuladores) disponibles para validar el sistema antes de un despliegue real.

2.3.1. Contratos inteligentes en Solidity y adecuación del EVM para control

La plataforma elegida frecuentemente para prototipos es Ethereum, dada su madurez, amplia adopción y capacidad de ejecutar contratos inteligentes Turing-completos. Ethereum utiliza la Ethereum Virtual Machine (EVM), una máquina virtual replicada en todos los nodos de la red, donde se ejecuta el código de los contratos. Un contrato inteligente en Solidity (el lenguaje de alto nivel de Ethereum) es esencialmente un programa con funciones y estado almacenado en la blockchain. Cada contrato reside en una dirección blockchain y su estado persiste en la cadena, pudiendo ser leído/escrito mediante transacciones o llamadas internas.

Esta infraestructura resulta adecuada para implementar la lógica de un controlador: se puede escribir un contrato inteligente que contenga variables de estado como K_p, K_i, K_d, el último error o salida calculada, el umbral de evento, etc., junto con funciones para actualizar la medida del sensor y computar la nueva acción de control. Cuando un sensor (oráculo) realiza una transacción, la función del contrato verifica la condición de evento (por ejemplo, compara el valor absoluto del diferencial del error con el umbral). Si se cumple, el contrato entonces ejecuta el algoritmo PID y guarda/retorna el nuevo control; si no, simplemente no cambia la salida (o mantiene la previa). De esta forma, el contrato inteligente actúa como el controlador event-driven, recibiendo eventos (transacciones con nuevas mediciones) y generando acciones de control almacenadas en la blockchain.

La idoneidad de Solidity/EVM para esto radica en varias cuestiones:

- **Determinismo y replicación**: Todas las operaciones del contrato se replican en cada nodo de la red y producen el mismo resultado en todos (determinismo), condición necesaria para el consenso. Un algoritmo de control es determinista, así que puede ejecutarse sin problemas. Esto asegura que todos los participantes ven la misma salida de control simultáneamente en la cadena, eliminando ambigüedades. Por ejemplo, si un actuador está monitorizando la dirección del contrato para cambios en la variable

u (salida de control), sabe que dicho valor es consistente en toda la red y aprobado por consenso.

- **Inmutabilidad del código**: Una vez desplegado el contrato PID, su lógica no puede ser modificada arbitrariamente (a menos que se programe una actualización explícita). Esto es positivo desde un punto de vista de certificación: garantiza que el controlador que se está ejecutando es exactamente la versión auditada y probada, y no ha sido alterado por terceros. Esto contrasta con servidores SCADA tradicionales donde un atacante podría intercambiar una librería de control por una maliciosa. En blockchain, cualquier cambio de lógica requeriría una nueva implementación de contrato, visible para todos.

- **Disponibilidad 24/7**: El contrato vive en la red y está siempre accesible mientras la red esté operativa. No depende de un servidor físico único que pueda caer. Para sistemas críticos que requieren alta disponibilidad, esta es una ventaja, aunque limitada por la disponibilidad general de la red blockchain (que en Ethereum es muy alta, con nodos globales redundantes).

- **Lenguajes y herramientas familiares**: Solidity es similar a JavaScript/C++, relativamente fácil de aprender para desarrolladores. Permite estructuras condicionales, bucles, cálculo aritmético entero, etc., suficientes para programar un PID. Adicionalmente, existen librerías matemáticas si se requiere aritmética fija de mayor precisión. El entorno Ethereum cuenta con frameworks de testing (Truffle, Hardhat) que facilitan desarrollar el contrato controlador con pruebas unitarias, e incluso simular llamadas secuenciales (por ejemplo, alimentando el contrato con una serie de errores pregrabados y verificando la respuesta). Todo esto acelera el desarrollo del controlador on-chain.

No obstante, también existen limitaciones a considerar:

- **Precisión numérica**: La EVM maneja enteros de 256 bits pero no tiene coma flotante nativa. Un PID típicamente sumaría errores multiplicados por K_i (ganancia integral), etc., que pueden no ser enteros. Se debe escalar las unidades para trabajar con enteros (por ejemplo, representar la temperatura en décimas de grado para tener una precisión de 0.1°C), o usar alguna biblioteca de punto fijo. Esto agrega complejidad y posible error de redondeo. Por fortuna, muchos procesos industriales operan en rangos acotados donde se puede definir un factor de escala apropiado. Otra opción es implementar el PID en forma incremental (diferencias enteras) para evitar problemas numéricos.

- **Costo en gas**: En Ethereum pública, cada invocación del contrato (cada evento de control) conlleva pagar fees de gas en ether. Para una alta frecuencia de eventos esto sería prohibitivo económicamente. En nuestro caso, aprovechamos que el control es event-driven (menos invocaciones). Aun así, es probable que se opte por usar una

red de prueba o privada durante la etapa experimental (ver Sección 4.3) o incluso en producción (p.ej., una blockchain permissionada dentro de la empresa, o una capa 2 con costos muy bajos). De hecho, en la microrred de Bin Masood et al. [2019] se empleó Ethereum privado para la co-simulación, evitando costos de gas. Si se requiriera usar una red pública, podría entrarse en esquemas de tokenomics donde los participantes del proceso comparten el costo de las transacciones de control, algo aún por explorar.

- **Temporalidad y concurrencia**: En Solidity no hay multithreading ni ejecución continua en background. El contrato actúa cuando es llamado mediante una transacción o mensaje. Esto implica que no podemos programar en Solidity algo como "ejecutar esta función cada 1s" (no hay temporizadores internos). La lógica event-driven naturalmente se alinea con esto: el contrato PID espera hasta que alguien (el sensor/oráculo) lo invoque con nuevos datos. Pero debemos asegurarnos de que siempre habrá un agente externo disparando las llamadas cuando toque. Si el sistema físico requiere un control aunque no haya eventos grandes (por ejemplo, para eliminar error estacionario pequeño), debemos implementar la regla de tiempo máximo con ayuda externa: es decir, tener un timer oráculo que tras cierto tiempo sin eventos fuerce uno (por ejemplo, reenviando el último estado para gatillar la actualización). Algunos servicios como **Chainlink Keepers** o **Gelato** permiten llamadas automatizadas a contratos según temporizadores off-chain. Alternativamente, un nodo local podría asumir esa función.

En conclusión, Solidity/EVM ofrece un entorno viable para implementar la lógica de control event-driven, siempre y cuando se trabajen las limitaciones mencionadas mediante diseños híbridos (ej. compensar en el contrato la variación temporal, pero delegar en oráculos externos el cronometraje de último recurso). En la siguiente subsección veremos cómo se estructura la arquitectura completa incluyendo esos oráculos y componentes off-chain necesarios.

2.3.2. Arquitectura de integración: sensores, oráculos y actuadores

Para desplegar un controlador PID basado en eventos en blockchain es necesario diseñar la arquitectura de integración entre el mundo físico y el contrato inteligente. A grandes rasgos, los componentes involucrados se enumeran a continuación:

- **Contrato inteligente PID**: Reside en la blockchain (por ejemplo, Ethereum). Contiene la lógica de control y estado (último error, última salida, umbrales, etc.). Expone funciones como `actualizarError(e)` que procesa un nuevo error medido y potencialmente actualiza la salida de control. También puede tener funciones para que un operador ajuste los setpoints o ganancias K_p, K_i, K_d de manera segura (con autenticación).

- **Sensor/Planta (Nodo Físico)**: El sistema físico bajo control (planta industrial, proceso) con sus sensores. Este componente por sí solo no puede interactuar con la blockchain; necesitará un middleware.

- **Oráculo/Gateway de Sensado**: Software que toma la lectura del sensor y la envía a la blockchain. Puede ser un programa corriendo en un computador conectado a ambos mundos (red de control y red P2P blockchain). Por ejemplo, un script Python con `Web3.py` (librería Python de comunicación con los protocolos blockchain) podría leer un sensor (vía modbus, OPC UA, etc.) y al ocurrir un evento, ejecutar `contract.functions.actualizarError(e).transact()`. Este oráculo debe tener las credenciales (clave privada) de una cuenta Ethereum autorizada para llamar al contrato. Alternativamente, si el sensor es suficientemente inteligente (un dispositivo IoT con capacidad blockchain), él mismo podría actuar de oráculo, firmando y enviando transacciones directamente.

- **Oráculo de Actuación (o actuador subscrito)**: Una vez que el contrato PID actualiza la salida de control (por ej., una variable u en su estado), hay que aplicarla al actuador físico (válvula, motor, etc.). Aquí se tienen dos opciones: (a) Un oráculo de salida que monitorea eventos de la blockchain (logs o cambios de estado) y cuando detecta una nueva salida, la escribe en el actuador mediante el protocolo correspondiente. Muchas plataformas Ethereum permiten suscribirse a eventos del contrato; por ejemplo, el contrato puede emitir un evento `ControlActualizado(u)` y un programa en Node.js o Python escucha ese evento y entonces envía el comando al actuador. (b) Hacer que el actuador por sí mismo consulte la blockchain periódicamente para ver la última orden dirigida a él (en un esquema pull). Esto último fue lo implementado en PLCBlox: el PLC pregunta a la blockchain por nuevos setpoints cada cierto intervalo [Short et al., 2024]. Esta técnica elimina la necesidad de un proceso externo empujando la salida, a cambio de programar la lógica de consulta en el dispositivo.

- **Red Blockchain y Nodos**: Los propios nodos de la blockchain que mantienen el contrato. En un despliegue real, podrían ser nodos dedicados dentro de la planta o nodos en la nube si se usa una red pública. En entornos privados, a veces los controladores (PLCs, RTUs) pueden ejecutar nodos ligeros de blockchain, aunque esto aún es experimental.

- **Interfaces de usuario / SCADA**: Por último, podría haber un frontend para que ingenieros u operadores supervisen el comportamiento. Una dApp web podría leer del contrato la historia de mediciones y acciones (que quedan grabadas) para mostrarlas gráficamente, brindando una suerte de SCADA descentralizado. Además, mediante transacciones especiales, un operador autorizado puede ajustar parámetros del controlador o cambiar el valor deseado (setpoint) de forma transparente y registrada en la

cadena (por ejemplo, un operador sube la temperatura de consigna de 20°C a 22°C firmando una transacción al contrato).

Una estrategia es implementar los oráculos como microservicios en edge servers dentro de la planta, posiblemente contenedorizados, de forma que sean fáciles de reiniciar o duplicar. Por ejemplo, supongamos un sistema de control de temperatura en una cámara, con un PID on-chain. El sensor es un termómetro IoT. El actuador es un calentador eléctrico controlado por un PLC. El flujo sería:

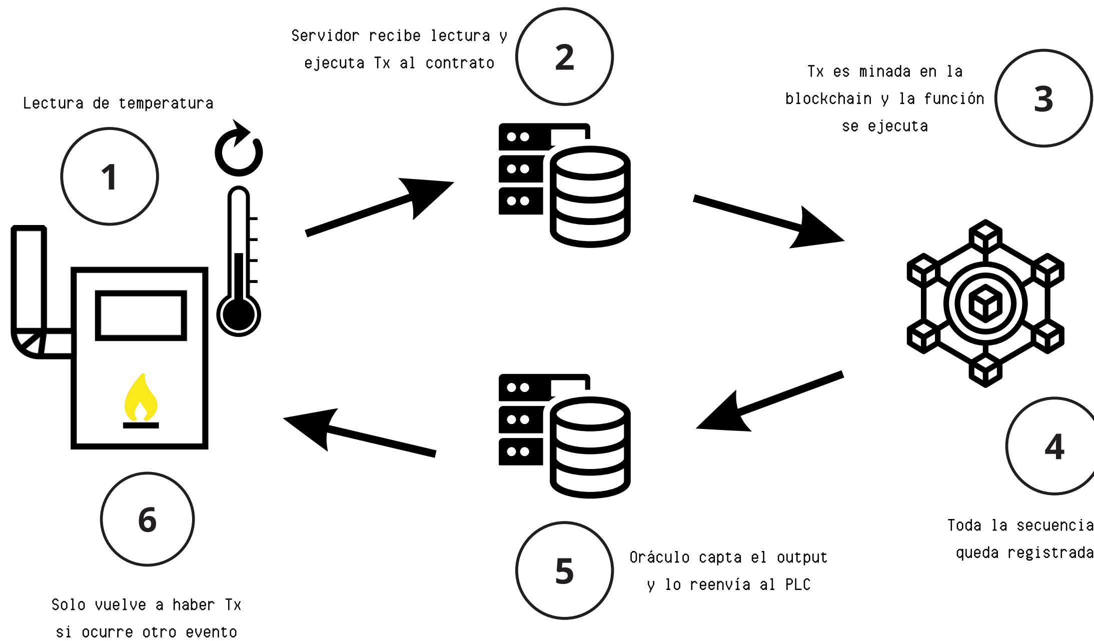

Figura 2.1: Diagrama de flujo de comunicación, numerando los seis pasos explicados.

1. El termómetro lee continuamente la temperatura. Tiene programado que si la diferencia respecto a la última enviada excede 0.5°C, enviará un evento. Cuando ocurre, envía (tal vez vía MQTT a un servidor local) la nueva lectura.

2. Un proceso oráculo en el servidor local recibe la lectura y ejecuta una transacción al contrato `PIDController` con el nuevo error (setpoint - temperatura).

3. La transacción se mina en la blockchain (puede tardar unos segundos[1]). Cuando se confirma, todos los nodos ejecutan la función: el contrato comprueba el umbral, digamos que se superó y entonces recalcula la salida u (por ejemplo, abre la válvula de

[1]El tiempo que transcurre entre el envío de la transacción y su inclusión en un bloque depende de tres factores principales: (i) el *gas price* ofrecido (propina + *base fee* EIP-1559), (ii) la congestión de la red y (iii) el tiempo de bloque de la cadena. Para lazos que no toleren esos retardos es recomendable migrar a **redes *layer-2***, con tiempos de bloque menores. En cualquier caso, el retardo puede modelarse como una pura demora en el diseño del PID o mitigarse con un búfer predictivo, pero es la limitación práctica clave de esta arquitectura.

vapor al 30 %). Esta salida queda almacenada en el estado del contrato y se emite un evento `ControlActualizado(0.30)`.

4. Toda esta secuencia quedó registrada: la transacción con la lectura de temperatura y la nueva salida de control son parte de la cadena, de modo que más tarde se puede auditar qué ocurrió, a qué hora y por qué (qué condición disparó el cambio).

5. Un oráculo de salida, suscrito a ese evento, lo capta y a su vez envía el comando al PLC del calentador: "abrir válvula 30 %". El PLC recibe este comando y lo aplica.

6. Si la temperatura se mantiene estable después, no habrá nuevas transacciones hasta que ocurra otro evento (por ejemplo, bajó 0.5°C más, generando otro ajuste).

La seguridad en esta arquitectura depende en gran medida de que los oráculos estén autenticados y de que el contrato solo acepte datos de fuentes válidas. En Ethereum, esto puede lograrse programando en el contrato que solo ciertas direcciones (las del sensor oráculo) puedan llamar a `actualizarError`. Es posible usar esquemas de múltiples firmas o validación cruzada de varios sensores para mayor confiabilidad, aunque eso excede el alcance de este trabajo. En general, esta arquitectura demuestra que es factible integrar el control on-chain con sistemas industriales existentes (sensores, PLCs) usando estándares abiertos como OPC UA, MQTT, etc., combinados con transacciones blockchain. Los trabajos previos citados (Masood, Vick, PLCBlox) han implementado variantes de este esquema con éxito limitado pero prometedor.

2.3.3. Validación en testnets y simulación

Antes de desplegar un controlador en una red blockchain operativa, es esencial realizar extensas pruebas en entornos controlados. Afortunadamente, el ecosistema Ethereum (y otras plataformas) cuenta con redes de prueba (testnets) públicas y la posibilidad de montar blockchains privadas para ensayo. Asimismo, se pueden utilizar herramientas de simulación y co-simulación para verificar el desempeño del lazo de control incluyendo los retardos de la blockchain. Esta sección describe cómo aprovechar esos recursos para validar el sistema propuesto.

- **Redes de prueba Ethereum**: Una testnet es básicamente una copia de la red Ethereum sin valor real, donde los desarrolladores pueden desplegar contratos y ejecutar transacciones sin costo. Ejemplos actuales son Sepolia y Goerli. En nuestro contexto, usar una testnet nos permite conectar los componentes reales (oráculos, dispositivos) pero operando sobre una blockchain simulada. Por ejemplo, podríamos desplegar el contrato PID en Sepolia, obtener algunos ETH de prueba de un faucet (para pagar gas ficticio) y luego configurar nuestros oráculos para interactuar con Sepolia en vez de mainnet. Esto nos deja observar el comportamiento casi real de las transacciones

(tiempos de bloque, etc.) sin gastar dinero ni arriesgar activos reales. Además, podemos introducir escenarios adversos (como congestionar la testnet con transacciones artificialmente) para ver cómo repercute en el control. Moralis (2023) enfatiza que las testnets proveen un entorno seguro y de bajo costo para iterar en el desarrollo de contratos y aplicaciones Web3 antes de pasar a producción. En ciertos casos, podría convenir montar una blockchain privada/local usando herramientas como Ganache (Ethereum local en memoria) o incluso una red de nodos PoA (prueba de autoridad) dentro de la empresa. Bin Masood et al. [2019] mencionan haber utilizado una co-simulación con Ethereum privada en su prueba de concepto, lo cual les permitió ajustar parámetros sin restricciones. Una red privada ofrece mayor control: podemos fijar el tiempo de bloque a, digamos, 1 segundo para probar un escenario más exigente, o modificar parámetros del consenso. Sin embargo, puede haber sutiles diferencias con la red pública (por ejemplo, en privadas suele no haber congestión ni transacciones inesperadas). Aun así, como paso de validación es muy útil.

- **Simulación y co-simulación**: Además de probar el contrato en sí, debemos probar el lazo de control completo. Dado que integrar hardware real a veces es complejo en primeras etapas, es común realizar simulaciones de la planta controlada. Herramientas como MATLAB/Simulink o simuladores específicos del proceso (ej. simuladores de redes eléctricas, de procesos químicos) pueden conectarse con el contrato blockchain a través de interfaces. Como se comentó, Bin Masood et al. [2019] desarrolló una co-simulación MATLAB + Ethereum: MATLAB calculaba la dinámica de la microrred y cada cierto tiempo invocaba al contrato (simulando mediciones), luego leía la acción devuelta para aplicarla al modelo. Incluso registraron los retardos y confirmaron que eran de pocos segundos. En nuestro laboratorio podríamos hacer algo similar: modelar la planta (ej. un tanque con agua) en Simulink, y escribir un script que enlace con el contrato PID desplegado en Ganache (via API JSON-RPC de Ethereum). Cada vez que Simulink complete un paso y tenga un nuevo valor de sensor, llamaría a una función en el contrato y obtendría la respuesta. MathWorks provee ciertas soluciones de co-simulación, o se puede hacer manualmente con temporización. Otra ruta es usar sólo Python. Con frameworks como Web3.py podemos programar un bucle que simule la planta con ecuaciones diferenciales sencillas (o usando libraries de control) y a la vez interactúe con el contrato. Python también puede modelar eventos asíncronos para imitar la naturaleza event-driven (por ejemplo, usando asyncio o simplemente lógica condicional para llamar al contrato cuando la diferencia supere X). Esto es sumamente flexible: permite evaluar distintas estrategias de event-trigger antes de codificarlas on-chain. Incluso podríamos optimizar el umbral de evento iterativamente corriendo la simulación para minimizar cierto criterio de desempeño vs costo de transacción.

- **Herramientas de desarrollo**: Para implementar el contrato y probar su correcto fun-

cionamiento lógico (independientemente de la planta), se suelen usar entornos como Truffle o Hardhat que permiten escribir test scripts en JavaScript/TypeScript. Podemos escribir pruebas unitarias que, por ejemplo, llamen a la función del contrato con una secuencia de errores y verifiquen que la salida calculada coincide con la que generaría un PID off-chain tradicional. Estas pruebas pueden correr en segundos sobre Ganache (blockchain local instantánea) y dan confianza en que la implementación de Solidity es correcta. Adicionalmente, Hardhat permite simular diferentes cuentas llamando al contrato, asegurando que la restricción de acceso (solo el oráculo sensor puede actualizar, etc.) funciona. Por último, al conectar con hardware real, se recomienda primero hacer pruebas en laboratorio con los dispositivos en entorno controlado. Por ejemplo, conectar un sensor a un Raspberry Pi actuando como oráculo y un actuador pequeño, usando quizás una blockchain local en la Raspberry. Esto ayuda a depurar detalles de comunicaciones (latencias de red local, formatos de datos, etc.) sin la complejidad adicional de la red global. Sólo después de afinar todo, se migraría a una red de prueba pública, y finalmente a una red definitiva si aplica.

En general, la metodología de validación debe contemplar: (1) Verificación funcional del contrato (simulación software); (2) Simulación del lazo completo con la planta modelada; (3) Pruebas con componentes hardware reales en entorno de prueba; (4) Despliegue gradual (quizá en una sección piloto de la planta) en una blockchain operativa. Durante todas estas etapas es valioso aprovechar la característica de log de la blockchain: todos los eventos de control quedan registrados, lo que permite comparar después los resultados reales con los esperados y detectar desviaciones. En conclusión, las herramientas de simulación y las redes de prueba son aliados indispensables para alcanzar un nivel de confianza alto en la solución propuesta, dada la complejidad añadida por la blockchain. Numerosas guías y trabajos reafirman que probar en *testnet* es un paso obligado antes de cualquier despliegue en *mainnet*.

2.4. Resumen del estado del arte

En este capítulo se ha presentado un recorrido exhaustivo por el estado del arte en el tema de controladores PID basados en eventos implementados sobre redes blockchain. Se han identificado las motivaciones que impulsan esta línea de investigación: la necesidad de sistemas de control más flexibles, confiables y seguros en entornos industriales cada vez más conectados e interorganizacionales. Las tecnologías blockchain aportan un medio distribuido e inmutable para orquestar la toma de decisiones de control, mientras que la filosofía de control basado en eventos se adapta mejor a las limitaciones de dichas redes (evitando comunicaciones innecesarias y trabajando de forma asíncrona). A modo de resumen, resaltamos los siguientes hallazgos clave:

- La aplicabilidad de blockchain en control radica principalmente en mejorar la confianza y la resiliencia en sistemas distribuidos. Se han reportado implementaciones exitosas a pequeña escala, como el control descentralizado de microrredes eléctricas o celdas de manufactura robotizadas usando smart contracts, que demuestran que es posible trasladar la lógica de control a una red blockchain y mantener la estabilidad del proceso bajo ciertas condiciones.

- Los controladores PID basados en eventos han emergido como la pieza faltante para casar el control clásico con la naturaleza discreta de las blockchains. Su capacidad de reducir el muestreo y actuación a solo cuando hace falta los convierte en el aliado ideal para operar con las latencias de segundos de una cadena de bloques. Además, tecnológicamente son factibles de implementar en contratos inteligentes sin un sobrecoste prohibitivo, dado que calculan la acción de control sólo esporádicamente.

- Se identificaron desafíos técnicos notables: la latencia de la red blockchain impone la restricción de que este enfoque sólo es aplicable a lazos de control lentos o no estrictamente en tiempo real (control supervisorio, lotes, ajustes de referencia, etc.). Asimismo, la dependencia de oráculos para interacción con el mundo real añade complejidad y posibles vulnerabilidades fuera de la cadena. Superar estos retos requiere diseños cuidadosos (como incorporar backups locales, asegurar criptográficamente la capa de oráculo, y tal vez aprovechar blockchains más rápidas o redes privadas para casos que lo demanden).

- En cuanto a implementación práctica, se destacó la viabilidad de usar Solidity/Ethereum como plataforma, siempre y cuando se realicen pruebas exhaustivas en testnets y simuladores antes del despliegue real. Se recomendó una metodología de co-simulación (ej. MATLAB + Ethereum) que ya ha sido utilizada en literatura para estudiar el rendimiento. La experiencia sugiere que si los retardos se mantienen en unos segundos y la lógica de control es tolerante a ello (posiblemente gracias al control basado en eventos y a la inercia del proceso físico), el sistema puede funcionar correctamente.

Este campo de investigación es aún incipiente. Como trabajo futuro, se vislumbra profundizar en varios frentes: optimizar algoritmos de consenso o usar sidechains especializadas para reducir la latencia en control industrial; desarrollar estándares de oráculos para dispositivos industriales (por ejemplo, drivers blockchain para PLCs de distintos fabricantes); integrar técnicas de control predictivo o adaptativo en contratos inteligentes para contrarrestar los efectos de retardo; y realizar pilotos a escala real en industrias como la energética o manufactura para evaluar el desempeño en entornos operativos. También será importante atender la ciberseguridad integral de estas arquitecturas híbridas, pues aunque la blockchain provee un núcleo seguro, los alrededores (sensores, gateways) deben endurecerse igualmente.

En conclusión, el despliegue de controladores PID basados en eventos sobre blockchain representa una propuesta innovadora alineada con la evolución hacia la Industria 4.0, combinando control automático, IoT y tecnologías de contabilidad distribuida. Los estudios actuales muestran resultados alentadores en cuanto a aplicabilidad y beneficios (transparencia, descentralización, eficiencia en comunicaciones), al tiempo que iluminan los retos a resolver. Con la maduración de las plataformas blockchain (mayor rendimiento, menor costo) y una mayor familiaridad de la industria con estas herramientas, es plausible que en un futuro próximo veamos componentes de infraestructura de control soportados por blockchain operando en entornos reales, especialmente en aquellos que requieran cooperación entre múltiples actores de forma confiable. Este capítulo sienta las bases conceptuales y técnicas para comprender dicha visión y avanzar hacia su realización.

Capítulo 3

Materiales y métodos

3.1. Entorno experimental

El conjunto de experimentos se ejecutó íntegramente en un equipo de sobremesa convencional (CPU de cuatro núcleos, 16 GB de RAM) con *Windows 10 Pro 64-bit*. Dado que el objetivo principal era validar la lógica de control y la interacción *off-chain/on-chain*, los requisitos de cómputo son modestos y no se hizo uso de aceleración por GPU ni de infraestructura en la nube.

3.1.1. Pila de software

La Tabla 3.1 resume las versiones de las herramientas y librerías empleadas. Python 3.13 se gestionó mediante un entorno virtual (`venv`) localizado en el directorio del proyecto; todas las dependencias se instalaron desde `PyPI` usando el fichero `requirements.txt` que acompaña al repositorio.

Tabla 3.1: Herramientas y librerías principales utilizadas en los experimentos.

Componente	**Versión / Descripción**
Python	3.13.0 (64-bit)
NumPy	2.2.5
Matplotlib	3.10.1
Web3.py	7.11.0
Py-solc-x	2.0.3
Eth-tester (+py-EVM)	0.13.0b1
Streamlit	1.33.* (front-end de monitorización)
Solidity compiler	0.8.17 (descargado dinámicamente por Py-solc-x)
Git	2.44 (control de versiones local)

3.1.2. Nodo blockchain y compilación de contratos

El proyecto admite **dos entornos on-chain intercambiables** elegibles desde la interfaz *Streamlit* o la línea de comandos:

1. **Modo local** *Ethereum Tester* (`chain-id 131277322940537`) ejecutado en memoria; suministra diez cuentas precargadas y tiempos de bloque sub-milisegundo, ideales para depuración determinista.
2. **Modo testnet** Sepolia (`chain-id 11155111`) accesible a través de un *endpoint* RPC (Infura, Alchemy, ...); requiere una clave privada y gestiona comisiones reales aunque insignificantes.

Inyección de configuración

Los parámetros de conexión a la red, ya sea un nodo local en memoria o una testnet pública, se gestionan de forma declarativa: la clase `Settings` carga unas credenciales y valores por defecto desde el fichero `.env`, y en el momento de ejecutar la simulación la interfaz decide, según la opción seleccionada por el usuario, qué campos debe sobreescribir (p.,ej. activar el proveedor *in-memory* o introducir la URL RPC y la clave privada). De este modo la lógica de negocio permanece desacoplada de la infraestructura: el resto del programa sólo ve una instancia de `Settings` y una llamada a `init_web3()`, que construye el objeto `Web3` apropiado (tester o RPC) sin que el resto del código tenga que preocuparse de los detalles de transporte.

Flujo de compilación y despliegue

1. **Compilación reproducible**. Antes de cada prueba se invoca la versión indicada del compilador `solc` y se genera nuevamente el binario y el ABI del contrato, de modo que cualquier cambio en la fuente de Solidity queda reflejado al instante y puede rastrearse.
2. **Creación de la transacción**. El script construye y firma localmente la transacción de despliegue con un límite de gas amplio y la tarifa (*gas-price*) que corresponda al entorno: cero en el nodo de memoria y la que dicte la red cuando se opera en Sepolia.
3. **Envío y confirmación**. En el nodo local el contrato queda disponible casi de inmediato, mientras que en la testnet pública se espera el tiempo normal de inclusión en bloque (unos 12–15 s), tras lo cual se devuelve el *receipt* con la dirección del contrato ya inicializado.

Ventajas de la doble estrategia

Trabajar con ambos entornos aporta, en primer lugar, una **iteración muy rápida**, ya que el nodo en memoria elimina la latencia y permite depurar la lógica Solidity sin consumo

de gas; en segundo lugar, ofrece **pruebas realistas**, pues el mismo script ejecutado sobre Sepolia valida el contrato bajo las reglas auténticas de gas, nonce y temporización de una red pública; y, por último, garantiza **portabilidad**, ya que cambiar de un entorno a otro sólo exige seleccionar "on-chain (local)" u "on-chain (testnet)" y facilitar una vez la `RPC_URL` y la `PRIVATE_KEY`, sin alterar el resto de la arquitectura (oráculos, simulador o interfaz) que permanece idéntica en ambos casos.

Esta configuración híbrida acelera el ciclo prueba–medición–optimización y ofrece la confianza adicional de que el controlador funcionará sin modificaciones en redes públicas o incluso en *layer-2* de bajo coste.

3.1.3. Simulación del proceso y lógica off-chain

La biblioteca de simulación ofrece dos plantas térmicas intercambiables:

- **Caldera de primer orden** (`FirstOrderBoilerModel`), que modela la masa de fluido como un único nodo térmico con pérdidas por convección y radiación.

- **Caldera de segundo orden** (`SecondOrderBoilerModel`), que añade un nodo adicional para las paredes del recipiente (T_2) y un flujo interno $k_{12}(T_1 - T_2)$, representando la inercia térmica de las superficies metálicas.

En ambos casos, la integración numérica se puede realizar mediante Euler o RK4 (parámetro `method`) con un paso típico de $\Delta t = 0{,}1$–1 s; el sensor virtual añade ruido gaussiano de $\sigma = 0{,}1\,^{\circ}\mathrm{C}$. La lógica *off-chain* (`onchain_step`) actúa como oráculo: aplica la estrategia *send-on-delta*, detecta eventos cuando $|e|$ supera el umbral δ, firma la transacción `update(error_fixed)` y procesa el evento `ControlOutput` emitido por el contrato inteligente.

3.1.4. Interfaz de usuario

El **front-end** se desarrolló con *Streamlit* (archivo `app.py`). Desde la misma interfaz el usuario puede: (i) lanzar simulaciones totalmente locales (*off-chain*); (ii) compilar y desplegar el contrato, ejecutar la simulación *on-chain* y visualizar temperaturas y señales de control en tiempo real; (iii) realizar *grid search* de ganancias PID y barridos de umbral.

3.2. Modelo del sistema

El proceso empleado para validar el controlador es una **caldera eléctrica** cuya dinámica térmica puede describirse con dos niveles de fidelidad:

Modelo de primer orden

La formulación original representa la masa de fluido mediante una sola temperatura $T\,[K]$. La única entrada manipulada es la potencia del calentador $u\,[\mathrm{W}]$ y las pérdidas combinan convección lineal y radiación no lineal:

$$\frac{\mathrm{d}T}{\mathrm{d}t} = \frac{-\,q_{\mathrm{conv}}(T) - q_{\mathrm{rad}}(T) + u}{C} \tag{3.1}$$

$$q_{\mathrm{conv}}(T) = k_{\mathrm{conv}}\,(T - T_{\mathrm{amb}}) \tag{3.2}$$

$$q_{\mathrm{rad}}(T) = \varepsilon\,\sigma\,A\big(T^4 - T_{\mathrm{amb}}^4\big) \tag{3.3}$$

donde C es la capacidad térmica efectiva [J/K], k_{conv} la conductancia convectiva [W/K], ε la emisividad (–), σ la constante de Stefan–Boltzmann [$\mathrm{W/m^2K^4}$] y A el área radiativa [$\mathrm{m^2}$]. La no linealidad T^4 aporta realismo y fuerza al PID a lidiar con una ganancia dependiente de la temperatura, aunque la planta siga dominada por un único polo.

Extensión a segundo orden

Para evaluar la robustez del controlador frente a dinámicas más complejas se incorporó un **modelo de segundo orden** con dos nodos térmicos:

$$C_1\,\frac{\mathrm{d}T_1}{\mathrm{d}t} = u - k_{12}\,(T_1 - T_2), \tag{3.4}$$

$$C_2\,\frac{\mathrm{d}T_2}{\mathrm{d}t} = k_{12}\,(T_1 - T_2) - q_{\mathrm{loss}}(T_2), \tag{3.5}$$

donde T_1 es la temperatura del fluido, T_2 la temperatura de las paredes, $C_{1,2}$ [J/K] sus respectivas capacidades térmicas y k_{12} [W/K] la conductancia entre ambos nodos. La pérdida exterior $q_{\mathrm{loss}}(T_2)$ se calcula con las mismas expresiones de convección y radiación empleadas en Ecs. (3.2)–(3.3). Este segundo estado introduce una constante de tiempo adicional asociada a la masa metálica, haciendo la dinámica más inercial y poniendo a prueba el esquema *send-on-delta* ante errores prolongados.

En las simulaciones, el usuario puede alternar entre ambos modelos mediante un parámetro de configuración sin modificar el resto de la arquitectura (sensor–oráculo–contrato), lo que permite comparar directamente el impacto de la complejidad de la planta sobre la frecuencia de eventos y el consumo de gas.

La Figura 3.1 resume gráficamente la secuencia completa: el sensor mide la temperatura de la planta (que difiere de la temperatura real por el ruido), y esta medición se compara con el setpoint. Se genera una señal de error, que es enviada al contrato inteligente a través del oráculo de entrada. La lógica PID se aplica en la blockchain, donde se calcula la señal de control y se emite como output. El oráculo de salida, suscrito a ese evento, recibe la señal

de control y la envía al actuador, que finalmente aplica el calor necesario para controlar la temperatura de la caldera.

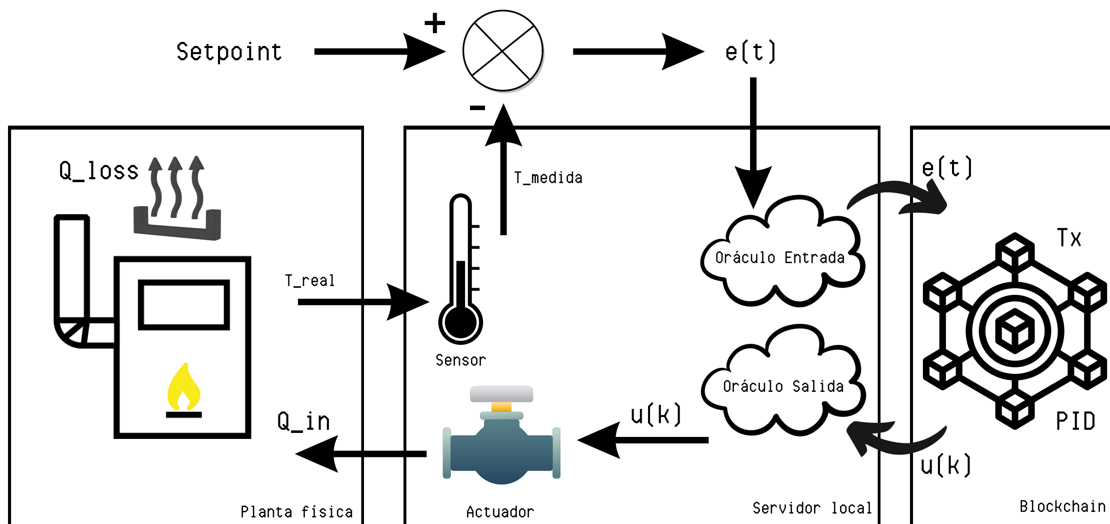

Figura 3.1: Diagrama del modelo del sistema que ilustra la interacción entre: (1) el setpoint y sumador de error, (2) el sensor con ruido gaussiano, (3) el oráculo de entrada, (4) el contrato `PIDController` en la blockchain, (5) el oráculo de salida y (6) el actuador que aplica la potencia u a la caldera.

3.2.1. Parámetros nominales

Los valores de referencia (Tabla 3.2) se seleccionaron para obtener una *constante de tiempo* dominante del orden de decenas de segundos y para que la contribución radiativa fuese apreciable por encima de $40 - 50$ °C. En el modelo de segundo orden se añadió una pared con la mitad de la capacidad térmica del fluido y una conductancia de acoplamiento tal que el segundo polo aparece en la banda de 50–60 s, lo que introduce inercia adicional sin hacer el sistema excesivamente lento. Durante los experimentos principales *no* se modificaron estos parámetros; el *front-end* los expone sólo para exploración interactiva.

Ajuste de la potencia máxima

En una versión preliminar el calentador se limitó a 100 W; al incluir la radiación ($\propto T^4$) se observó que el sistema era incapaz de alcanzar el set-point de 50 °C. Por ello se elevó $u_{\max}$ a 300 W, valor que garantiza un régimen estacionario alcanzable sin forzar saturación continua, a la vez que mantiene tiempos de subida del orden de minutos y por tanto compatibles con la latencia introducida por la lógica on-chain.

Tabla 3.2: Parámetros nominales de los modelos de caldera.

Parámetro	Valor por defecto	Justificación
Comunes a ambos modelos		
k_{conv} (W/K)	0.1	Convección natural aire–metal.
A (m^2)	1.0	Superficie simplificada, escala conveniente.
ε (–)	0.9	Superficie pintada en negro mate.
T_{amb} (°C)	20	Condición ambiente de laboratorio.
$u_{\text{máx}}$ (W)	300	Potencia requerida para contrarrestar pérdidas y alcanzar 50 °C.
Ruido sensor σ (°C)	0.1	RTD/PT-100 típica; realismo para el discriminador de eventos.
Modelo de primer orden		
C (J/K)	10	Constantes de tiempo globales de 10–30 s.
Modelo de segundo orden		
C_1 (J/K)	10	Capacidad térmica del fluido, igual que el modelo de primer orden.
C_2 (J/K)	5	Capacidad de las paredes (50 % del fluido).
k_{12} (W/K)	0.5	Acoplamiento fluido–paredes; sitúa el segundo polo en 50–60 s.

3.2.2. Método de integración

El método de referencia es **Runge–Kutta de cuarto orden** (RK4) con paso Δt configurable (típicamente $0{,}1 - 1$ s). RK4 ofrece mejor estabilidad y menor error local que el avance explícito de Euler cuando se combina la rigidez moderada del término radiativo con la saturación del actuador, evitando oscilaciones numéricas que podrían contaminar los tests de control. Se mantiene la opción `euler` como referencia histórica y para depuración rápida, pero todos los resultados presentados en los capítulos siguientes se obtuvieron con `method = 'rk4'`.

3.2.3. Medición y ruido

Para aproximar la realidad de un sistema industrial se incluye un *sensor virtual* que añade ruido gaussiano blanco $N(0, \sigma^2)$ a la temperatura real antes de entregarla al controlador. La desviación estándar $\sigma = 0{,}1$ °C refleja la resolución típica de un sensor RTD calibrado y confiere significado práctico al umbral de eventos: umbral inferiores a 3σ activan el controlador con alta frecuencia, mientras que valores mayores filtran variaciones que, en la práctica, serían indistinguibles del ruido de medición.

Este ruido se ha modelado como *aditivo* e independiente de la temperatura porque la mayoría de los RTD (por ejemplo, un PT-100 de clase A) especifican una incertidumbre absoluta casi constante en todo su rango de operación (típicamente ±0.1 °C). En la práctica, el error del sensor viene determinado por la precisión del elemento resistivo, la electrónica de acondicionamiento y las conexiones, y no crece linealmente con T. De este modo, elegir $\sigma = 0{,}1$ °C refleja fielmente la resolución realista de un RTD calibrado y permite usar un umbral $3\sigma = 0{,}3$ °C que filtre correctamente las variaciones que, en la práctica, serían indistinguibles del ruido.

3.2.4. Casos límite y saturación

El modelo revela dos escenarios críticos para la lógica de control:

1. **Saturación por demanda:** si el set-point supera la temperatura de equilibrio alcanzable con $u = u_{\max}$, el error persiste y la salida queda enganchada al 100 %. Este caso se utiliza para verificar la eficacia del anti-windup implementado tanto *off-chain* como *on-chain*.

2. **Sobrepotencia inicial:** al partir desde ambiente, q_{rad} es despreciable y la constante de tiempo efectiva es $\tau \approx C/k_{\text{conv}}$; a medida que T crece el término T^4 acelera las pérdidas, reduciendo la ganancia estática del proceso y evitando sobrepico exagerado. Este comportamiento suaviza la validación del umbral de eventos, pues la frecuencia de activación del PID disminuye conforme la planta se acerca al set-point.

3.2.5. Motivación de la elección

Se persiguió un equilibrio entre *simplicidad analítica* y *riqueza dinámica* suficiente para retar al controlador. Por ello se implementaron dos variantes jerárquicas:

1. un **modelo de primer orden** (Ecs. (3.1)–(3.3)),

2. y una **extensión de segundo orden** que añade el nodo térmico de las paredes y el acoplamiento $k_{12}(T_1 - T_2)$.

El conjunto cumple las siguientes premisas:

- Ambos modelos se resuelven en tiempo real con unas pocas líneas de Python (Euler o RK4), lo que facilita su integración con la capa blockchain durante las pruebas.

- El término radiativo T^4 introduce una ganancia dependiente de la temperatura; el segundo nodo añade un polo y un cero no mínimo fase, incrementando la inercia sin disparar la complejidad numérica.

- La parametrización permite, modificando sólo $u_{\text{máx}}$, k_{conv} o k_{12}, reproducir escenarios de *saturación continua* (control imposible) o *convergencia rápida* (control trivial), muy útiles para las pruebas funcionales del Cap. 4.

En suma, la pareja de modelos proporciona un banco de pruebas escalable y computacionalmente ligero que permite estudiar la interacción entre un controlador PID *send-on-delta* y la infraestructura blockchain sin que la complejidad del proceso físico eclipse los fenómenos de interés.

3.3. Diseño y optimización del PID

El controlador elegido es un **PID discreto** con saturación y anti–*windup* por *clamping*. El objetivo de esta sección es describir la formulación matemática, los mecanismos de disparo por eventos y el procedimiento sistemático empleado para ajustar las ganancias $\big(K_p, K_i, K_d\big)$ y el umbral de activación δ (*threshold*) antes de integrar la lógica en la blockchain. Los valores numéricos resultantes se presentan más adelante, en el Capítulo 4; aquí se documenta la metodología.

3.3.1. Formulación discreta y anti-windup

Sea $e_k = r_k - y_k$ el error de control en el instante discreto $t_k = k\,\Delta t$. El algoritmo implementado en Python aplica la forma incremental clásica

$$\begin{aligned} d_k &= \frac{e_k - e_{k-1}}{\Delta t}, \\ I_k^* &= I_{k-1} + e_k\,\Delta t, \\ u_k^{\text{raw}} &= K_p\,e_k + K_i\,I_k^* + K_d\,d_k, \end{aligned} \tag{3.6}$$

sujeto a **saturación** $u_k = \text{mín}\big(\text{máx}(u_k^{\text{raw}}, u_{\text{mín}}), u_{\text{máx}}\big)$. Para prevenir el fenómeno de *windup*, se emplea *clamping*: el integrador I_k sólo se actualiza con I_k^* si la salida no satura, o si la parte proporcional del error apunta a sacar el actuador fuera de la saturación. Esta lógica se replica de forma estricta en el contrato `EventPID.sol`, manteniendo *punto por punto* la equivalencia funcional gracias a:

- Escalado fijo de 18 decimales (`DECIMALS = 1e18`) para trasladar los cálculos de coma flotante a aritmética entera en la EVM.
- Misma regla de clamping y límites $u_{\text{mín}}, u_{\text{máx}}$.
- Cómputo de la derivada como diferencia entre el error actual y el anterior (no filtrada), idéntico a la versión Python.

La acción derivativa, al basarse en la diferencia puntual $d_k = \frac{e_k - e_{k-1}}{\Delta t}$, tiende a amplificar cualquier componente de ruido en la señal de error y, en un esquema de muestreo por eventos, esto podría disparar eventos adicionales de forma indeseada. No obstante, en este caso se decidió no implementar un filtro derivativo por las siguientes razones:

- El sensor virtual ya incorpora un ruido gaussiano limitado $\big(N(0, \sigma^2),\ \sigma = 0{,}1\,°\text{C}\big)$, y el umbral de activación δ se ajusta de modo que las variaciones menores a 3σ no generen eventos, actuando de facto como un filtro paso-banda.
- Incorporar un filtro paso-bajo explícito (por ejemplo, un filtro de primer orden) en la parte derivativa exigiría guardar estados adicionales y realizar multiplicaciones frecuentes en la EVM, lo que se traduciría en un consumo de gas significativamente mayor.

- El período discreto Δt elegido es lo suficientemente pequeño y el *send-on-delta* impone un tiempo mínimo entre eventos ($h_{\text{mín}}$), de modo que los pequeños picos de ruido quedan amortiguados sin necesidad de un filtrado explícito: solo cuando el cambio de error supera el umbral y se respeta $h_{\text{mín}}$ se recalcula la salida.
- Para este trabajo, se ha optado por diseñar un sistema sencillo que permita realizar un análisis preliminar de viabilidad en blockchain. Implementaciones más complejas del control derivativo o filtros dedicados quedan pendientes como extensión en trabajos futuros.

Con esta estrategia, el controlador mantiene estabilidad y evita disparos excesivos provocados por oscilaciones de ruido de baja amplitud, a la vez que se conserva la equivalencia funcional con la versión *off-chain* y se minimiza el coste de gas.

3.3.2. Disparo por eventos

La lógica de muestreo se basa en un umbral de error δ:

$$\left|e_k\right| \geq \delta \implies \text{ejecutar PID.update}(e_k), \qquad \left|e_k\right| < \delta \implies u_k = u_{k-1}.$$

El parámetro δ constituye, por tanto, un cuarto grado de libertad cuya selección impacta en el número de transacciones on-chain (eventos) y en la calidad del control. La estrategia seguida consta de dos fases:

1. **Ajuste de** (K_p, K_i, K_d) con δ fijo en su valor nominal (por defecto 2,65 °C).
2. **Barrido de** δ manteniendo las ganancias óptimas para explorar el compromiso *eventos–desempeño.*

3.3.3. Métrica de optimización

Para comparar configuraciones se empleó un índice adimensional construido en dos pasos:

1. **Normalización individual (min–max).** Cada métrica bruta (tiempo de establecimiento t_{settle}, sobrepaso máximo M_p e integral del error absoluto IAE) se escala linealmente al intervalo $[0, 1]$ mediante $\text{norm}(x) = \frac{x - x_{\text{mín}}}{x_{\text{máx}} - x_{\text{mín}}}$. Este escalado (i) mantiene la ordenación, (ii) evita que una métrica domine por diferencias de orden de magnitud y (iii) prescinde de pesos arbitrarios porque todos los términos quedan en la misma escala.
2. **Agregación por suma**. La puntuación global se define como

$$J = \text{norm}(t_{\text{settle}}) \ + \ \text{norm}(M_p) \ + \ \text{norm}(\text{IAE}),$$

y la mejor configuración es la que minimiza J.

Justificación del escalado min–max

La búsqueda se acota a unas pocas decenas de combinaciones, los valores de las métricas no presentan colas largas y los *outliers* son improbables, de modo que los riesgos clásicos del min–max (aplastamiento del rango o dependencia fuerte de $x_{\text{mín}}, x_{\text{máx}}$) se consideran aceptables. Se evaluó la posibilidad de utilizar una normalización gaussiana ($z = (x - \mu)/\sigma$); sin embargo, el *z-score* introduce valores no acotados, complica la interpretación y apenas modifica los resultados en un *grid search* discreto como el presente. Para espacios de búsqueda mucho mayores o distribuciones sesgadas se podría recurrir a z-score o a un escalado robusto (mediana/IQR), pero en este trabajo la normalización min–max ofrece el mejor compromiso entre simplicidad, interpretabilidad y eficacia.

En el barrido del umbral δ se aplica la misma lógica: se normalizan las tres métricas de desempeño para calcular J y se normaliza aparte el número de eventos; la suma de ambos normalizados permite escoger el umbral que consigue el mejor equilibrio entre calidad de control y coste de transacción.

3.3.4. Procedimiento de *grid-search*

La exploración de ganancias se implementó como búsqueda exhaustiva sobre mallas uniformes definidas por los rangos por defecto. Para cada triple (K_p, K_i, K_d) se ejecuta `run_simulation_eventbased(...)` durante $t_f = 100$ s con paso $\Delta t = 0{,}1$ s y se evalúa J. El resultado de la evaluación se guarda junto con el número de eventos n_events para su análisis posterior. El *grid-search* se ejecuta completamente en modo *off-chain* a fin de acelerar el barrido (no se incurre en tiempo de bloque) y proporcionar valores razonables antes de migrar la lógica a la blockchain.

3.3.5. Barrido de umbral

Una vez fijados $\left(K_p, K_i, K_d\right)_{\text{opt}}$, se genera una lista de umbrales $\{\delta_i\}$ equiespaciada. Para cada δ_i se recalcula la métrica J_i y el número de eventos n_events_i. La pareja $\left(J_i, n_events_i\right)$ traza la curva *trade-off* que más tarde servirá para seleccionar el valor final de δ según el criterio de compromiso "mínimo número de transacciones para un error aceptable".

Cabe señalar que, en sistemas *event-triggered*, la forma en que se elige el umbral δ puede influir en la estabilidad y generar complejos fenómenos dinámicos (por ejemplo, ciclos límite o disparos repetitivos). Sin embargo, el alcance de este trabajo se ha centrado en demostrar la *viabilidad funcional* y el compromiso desempeño-coste: por ello, no se aborda de forma formal un análisis de estabilidad ni se examinan posibles oscilaciones permanentes atribuibles a la selección de δ. Dicho estudio requeriría un tratamiento más exhaustivo (incluyendo herramientas de teoría de sistemas híbridos y análisis de retrasos) que excede el objetivo actual de verificar técnicamente el controlador PID on-chain. En consecuencia, la evaluación

de la estabilidad y la identificación de posibles ciclos límite se dejan para trabajos futuros, donde se profundizará en la caracterización teórica y la validación experimentales de dichos aspectos.

3.3.6. Resumen de la metodología

1. Adoptar un umbral inicial δ y realizar *grid-search* exhaustivo sobre (K_p, K_i, K_d) minimizando la métrica compuesta J.

2. Fijar la terna óptima y barrer δ para cuantificar el compromiso desempeño–eventos.

3. Trasladar las ganancias y el umbral seleccionados al contrato `EventPID.sol` y comparar salidas con el PID *off-chain* a igualdad de entradas, verificando la fidelidad bit a bit.

Este flujo, que se muestra en el diagrama de la Figura 3.2, garantiza que las constantes finales provengan de un proceso reproducible, desligado de la latencia de la blockchain, y que el paso de Python a Solidity introduzca únicamente errores de cuantización inferiores al ruido del sensor.

3.4. Implementación del contrato inteligente

Para trasladar la lógica PID al dominio *on-chain* se desarrolló el contrato `EventPID.sol` (List. 3.1), escrito en Solidity 0.8.17 y compatible con cualquier red EVM. El diseño persigue tres objetivos: *fidelidad funcional* respecto al PID *off-chain*, *determinismo* frente a intervalos de muestreo irregulares y *eficiencia de gas*.

3.4.1. Punto fijo y almacenamiento compacto

La EVM carece de coma flotante; todas las magnitudes en punto fijo se escalan con

$$\texttt{DECIMALS} = 10^{18},$$

lo que permite representar hasta 18 cifras decimales sin desbordar el rango de `int256`. Las ganancias $K_{p,i,d}$, el error y el paso de muestreo Δt se almacenan ya escalados, de modo que cada operación se reduce a multiplicación entera seguida de una división por `DECIMALS`. El estado persistente queda limitado a cinco variables $(K_p, K_i, K_d, \delta, \Delta t)$ más el integrador y el error previo, manteniendo el consumo de gas bajo.

3.4.2. Constructor inmutable

Las constantes del controlador y los límites de actuador $(u_{\text{mín}}, u_{\text{máx}})$ se fijan en el *constructor*. No se exponen `setters`: modificar una ganancia implica desplegar un nuevo contrato.

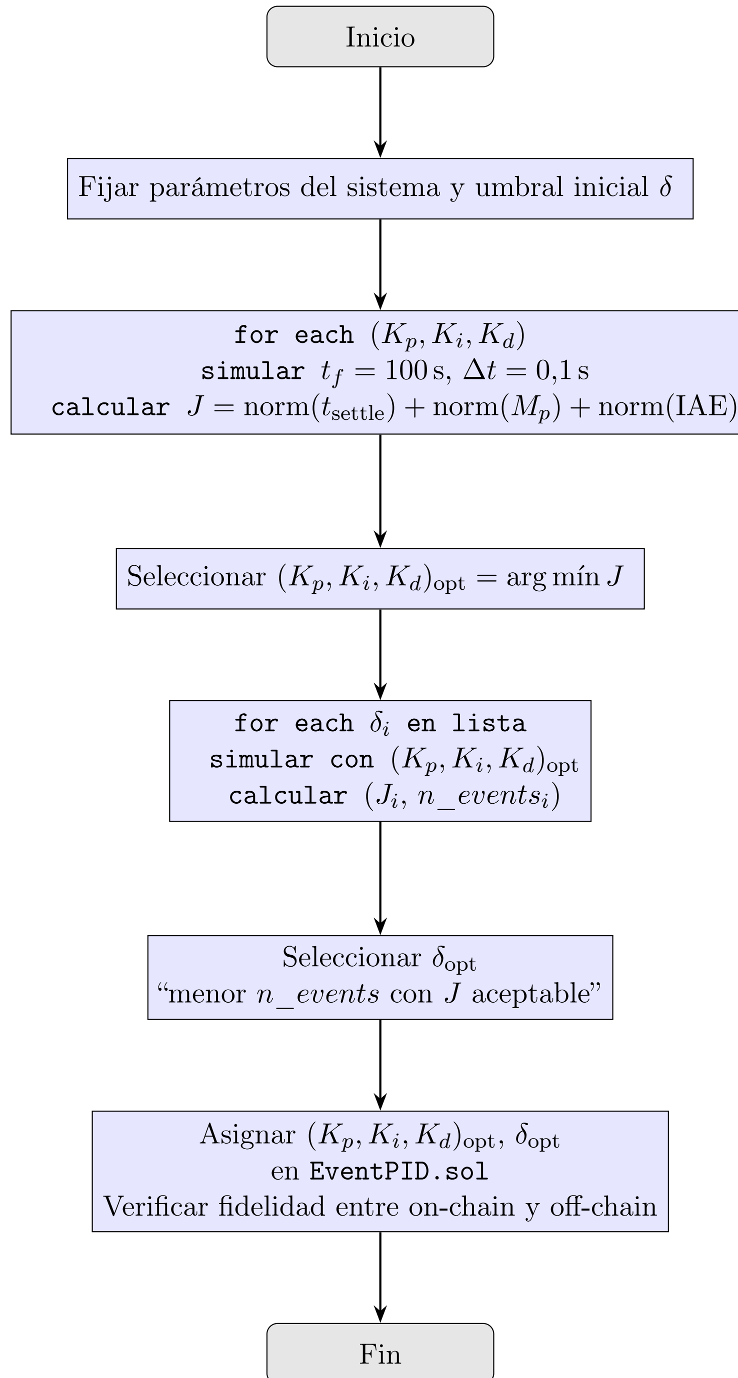

Figura 3.2: Diagrama de flujo de la metodología: (i) definición de parámetros, (ii) búsqueda exhaustiva de (K_p, K_i, K_d) mediante *grid-search*, (iii) barrido del umbral δ y (iv) despliegue on-chain con verificación de fidelidad.

Esta inmutabilidad simplifica la auditoría, evita cambios furtivos y elimina la necesidad de roles de acceso (*owner*, *admin*), alineándose con el principio KISS ("Keep It Simple, Stupid"), que promueve mantener el diseño tan sencillo como sea posible para reducir errores y facilitar el mantenimiento en contratos críticos de control.

3.4.3. Algoritmo `update()`

La función `update(error)` (única de uso público) ejecuta un paso PID discreto si y sólo si $|e| \geq \delta$:

1. **Rechazo temprano**: `require(error>=threshold||error<=-threshold)`. Este filtro a nivel de EVM descarta transacciones innecesarias antes de consumir gas de cómputo.

2. **Integral y derivada**:

$$I^* = I + \frac{e\,\Delta t}{\texttt{DECIMALS}}, \quad D = \frac{(e - e_{prev})\,\texttt{DECIMALS}}{\Delta t}.$$

   El término derivativo se normaliza por Δt para garantizar coherencia con la versión *off-chain* independientemente del intervalo real entre eventos.

3. **Salida no saturada**: $u_{\text{raw}} = K_p e + K_i I^* + K_d D$ (re-escalado).

4. **Saturación y *clamping***: se ajusta u a $[u_{\text{mín}}, u_{\text{máx}}]$; la integral sólo se actualiza si (i) no hay saturación o (ii) el error tendería a des-saturar el actuador, reproduciendo el *anti-windup* del código Python.

5. **Event logging**: se emite `ControlOutput(error, u)`, único *log* que el oráculo de salida necesita para aplicar la orden al actuador.

Con esta secuencia el contrato ejecuta ≈ 35 operaciones aritméticas, dos escrituras de estado y una emisión de evento, lo que se traduce en $\sim 50-70 \times 10^3$ gas por llamada, muy por debajo del límite de 3 000 000 gas fijado en el `deploy` script.

3.4.4. Determinismo y atomicidad

Todas las variables internas se actualizan de forma atómica dentro de la misma transacción; por consiguiente, la red garantiza que *todos* los nodos del consenso observan la misma salida u y el mismo estado interno después de la llamada. Dado que Δt es constante y explícito en el contrato, las ganancias efectivas del PID no se distorsionan aunque el tiempo real entre eventos varíe por congestión de red; el efecto se traslada al error $e = e(t_k)$, manteniendo la equivalencia funcional con la ejecución local.

3.4.5. Compilación y despliegue

La construcción del contrato se orquesta íntegramente desde Python: el código Solidity se compila con la versión requerida de `solc`, generando de forma reproducible tanto el bytecode como la interfaz ABI. A continuación se prepara y firma la transacción de despliegue

(incluyendo las constantes del controlador ya convertidas a punto fijo) y se envía a la red seleccionada. Cuando llega el *receipt*, la aplicación recupera la dirección del contrato y la publica como variable de entorno para que el simulador y los oráculos puedan conectarse al instante al nuevo `PIDController`.

3.4.6. Equivalencia con el PID *off-chain*

Dado que la lógica aritmética, la norma de saturación y el mecanismo de *clamping* son idénticos en ambos dominios (diferencias de escala aparte), cualquier divergencia entre las salidas locales y las on-chain sólo puede deberse a:

- Cuantización entera ($<10^{-18}$ unidad en la escala adoptada),
- Retardos de evento (no afectan la fórmula, sólo el instante de actualización).

Esta fidelidad "código espejo" respalda la validez de las pruebas *off-chain* como fase previa a la integración con la blockchain, reduciendo iteraciones y costes de gas durante el desarrollo.

```
1  if (abs(error) < threshold) revert("Sin evento");
2  int256 integralTent = integral + (error * dt) / DECIMALS;
3  int256 derivative   = ((error - prevError) * DECIMALS) / dt;
4  int256 uUnsat = (Kp * error + Ki * integralTent + Kd * derivative) /
      DECIMALS;
5  int256 u = clamp(uUnsat, minU, maxU);
6  if (u == uUnsat || (u == maxU && error < 0) || (u == minU && error >
      0)) {
7      integral = integralTent;
8  }
9  prevError = error;
10 emit ControlOutput(error, u);
```

Listing 3.1: Fragmento esencial de `EventPID.sol`.

El código completo del contrato `EventPID.sol` se ha incluido a modo de anexo en la Sección A.1.

3.5. Desarrollo de la interfaz web

Con el fin de facilitar la experimentación se desarrolló una **aplicación web ligera** en *Streamlit*[1]. Este marco permite levantar un panel interactivo con pocos cientos de líneas y proporciona, sin código extra, los widgets reactivos necesarios para parametrizar la planta,

[1] `https://streamlit.io`

compilar/desplegar el contrato y visualizar los resultados. Además, la integración nativa con *matplotlib* simplifica la presentación de las curvas de temperatura y señal de control. El código de la aplicación Python/Streamlit utilizada se incluye íntegro en el Anexo B.1.

3.5.1. Estructura y flujo de trabajo

La interfaz se distribuye en **tres pestañas**: *Simulación*, *Optimización PID* y *Barrido de Umbral* (Fig. 3.3). En *Simulación* el usuario selecciona primero el *orden* del modelo térmico (primer o segundo) y después el *modo de ejecución*: *off-chain* (todo en Python), *on-chain local* con un nodo `eth-tester` en memoria, o *on-chain test-net* contra Sepolia a través de una URL RPC y la clave privada que firma las transacciones.

Figura 3.3: Interfaz web para la simulación y validación del PID sobre blockchain.

Al cambiar estas opciones la página revela dinámicamente los controles pertinentes: parámetros de la planta (capacidades, conductancias, emisividad, potencia máxima), condiciones de simulación (T_{set}, duración, paso Δt, umbral δ) y constantes de control K_p, K_i, K_d; en modo *on-chain* se habilitan, además, el selector de versión `solc` y, en test-net, los campos `RPC_URL`/`PRIVATE_KEY`. Al pulsar *Ejecutar Simulación* la aplicación:

1. lee el fichero `.env` y mezcla los valores introducidos por el usuario;
2. crea un objeto de configuración que decide, de forma transparente, si la conexión será a un proveedor en memoria o a un nodo RPC;
3. en caso *on-chain*, compila el contrato, lo despliega con los parámetros convertidos a punto fijo y espera la confirmación;

4. ejecuta la simulación propiamente dicha ,leyendo o escribiendo en la cadena sólo cuando es necesario, y recopila las métricas.

El resultado aparece como una tabla resumida (tiempo de establecimiento, sobrepaso, IAE, número de eventos, gas consumido y coste estimado) acompañada de las curvas temperatura-tiempo y control-tiempo (Fig. 3.4).

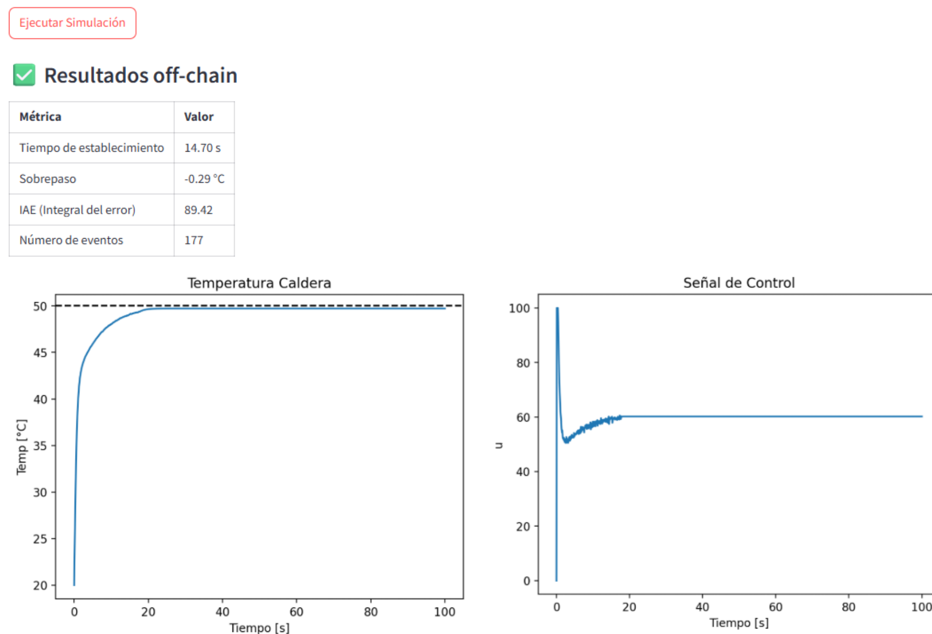

Figura 3.4: Interfaz web: visualización de resultados.

Las otras dos pestañas trabajan exclusivamente *off-chain*: *Optimización PID* explora una malla de ganancias y presenta la mejor terna según el índice combinado normalizado; *Barrido de Umbral* mantiene esas ganancias fijas, varía δ y muestra simultáneamente el número de eventos y la puntuación de desempeño para ayudar al usuario a decidir el compromiso adecuado.

3.5.2. Diseño de la experiencia de usuario

La interfaz mantiene un diseño *wide* para aprovechar el ancho de pantalla y mostrar, lado a lado, los dos gráficos principales (temperatura y señal de control). Todos los parámetros se introducen mediante `number_input` o `radio` tipados, de modo que los valores queden acotados a rangos lógicos y cada campo disponga de una clave única que evita colisiones entre pestañas. La aplicación responde de inmediato con mensajes de éxito o error. Por ejemplo, al compilar y desplegar un contrato se muestra la dirección resultante; tras cada simulación se indican las métricas clave y, en modo *on-chain*, el consumo de gas y el coste estimado. La información mínima que debe persistir entre pasos (`CONTRACT_ADDRESS`) se

almacena como variable de entorno, garantizando que el flujo siga siendo *stateless* para el resto de la lógica.

3.5.3. Estado actual y posibles extensiones

El panel ya opera tanto con el nodo en memoria (`eth-tester`) como con Sepolia, seleccionándose el destino desde la propia interfaz y sin modificar el código fuente. Las principales mejoras previstas son cosméticas: refinar la disposición de los controles, incorporar tablas interactivas (`st.dataframe`) para explorar los resultados de las optimizaciones y ofrecer la descarga de trazas en CSV. Desde el punto de vista funcional el objetivo original, hacer visible y manipulable el ciclo completo sensor $\rightarrow$ contrato $\rightarrow$ actuador en uno u otro entorno, ya se cumple sin exigir al usuario experiencia previa en Solidity ni en herramientas de línea de comandos.

Capítulo 4

Resultados

4.1. Configuración de las pruebas

Esta sección documenta las condiciones bajo las que se ejecutaron todas las campañas de experimentación ,tanto *off-chain* como *on-chain*, con el fin de que los resultados reportados sean reproducibles.

4.1.1. Red blockchain y medición de gas

Para aislar y comparar los efectos de la lógica de control respecto a la capa de ejecución, las campañas se dividieron en dos grandes entornos:

Entorno	**Propósito principal**	**Latencia**	**Precio de gas**
`eth-tester` (memoria)	Depuración rápida y medición *pura* de consumo de gas sin costos monetarios.	Inmediata	Fijado a 0
Sepolia (RPC Infura)	Validar el contrato en red pública bajo reglas reales de *gas*, nonce y temporización.	≈12 s por bloque	Dinámico

Tabla 4.1: Entornos utilizados en la validación *on-chain*.

Nodo local (`eth-tester`). Cada transacción se mina tan pronto como se envía, de forma que la latencia de confirmación es despreciable. El **gas price se fija a cero**, de modo que los `receipts` sólo reflejan el *número de unidades de gas* consumidas. Para estimar el coste monetario se multiplica, *a posteriori*, dicho consumo por un precio medio de 1,5 gwei y por la cotización de ETH publicada por *CoinGecko* (promedio: 2 525 USD/ETH, mayo 2025).

Sepolia. Las mismas pruebas se repitieron sobre Sepolia usando una cuenta propia (credenciales en el archivo `.env`). Al recibir el *receipt* se registran: (i) `gasUsed`, (ii) `gasPrice`

(wei/gas) y (iii) el tipo de cambio ETH → USD recuperado en línea[1]. Con estos tres valores se obtiene el coste exacto de cada transacción. El contrato se despliega con un margen de 30 000 000 gas y cada llamada `update()` con 500 000 gas. En ningún experimento se produjo *out-of-gas*: el consumo real del controlador se mantiene muy por debajo de esos límites (véase §4.4).

4.1.2. Parámetros de simulación

El escenario base para la planta y el controlador es:

- **Duración de la simulación**: $t_f = 200$ s.
- **Paso de integración** Δt: 0,1 s (caso nominal). Para el análisis de sensibilidad se repitieron ciertas series con $\Delta t = 0{,}5$ s y 0,05 s.
- **Modelo de caldera**: parámetros nominales de la Tabla 3.2; método de integración por defecto *RK4*.
- **Ruido de sensor**: desviación típica $\sigma = 0{,}1$ °C; se fija semilla global de `NumPy`, de modo que cada ejecución produce una realización idéntica del proceso estocástico. Para las métricas de optimización se usa *una única* realización por combinación de parámetros.

4.1.3. Estrategia de lanzamiento

- **Optimización de** K_p, K_i, K_d: búsqueda exhaustiva (§3.3) ejecutada en *modo off-chain* para reducir tiempos de cómputo y aislar el algoritmo del retardo de red.
- **Barrido de umbral** δ: igualmente en *off-chain*, empleando la terna de ganancias óptima; se registran la métrica compuesta J y el número de eventos.
- **Pruebas de integración**: una vez fijados los parámetros, las simulaciones se vuelcan a *on-chain* (nodo `eth-tester`) para medir:
  1. equivalencia numérica respecto al PID local,
  2. unidades de gas por transacción,
  3. impacto sobre la señal de control de las llamadas a red (con latencia nula en este entorno).
- **Repeticiones**: salvo en el análisis de sensibilidad al ruido (Sec. 4.4), cada configuración se ejecuta una sola vez; se comprobó anteriormente a la definición de la semilla global que la variabilidad inducida por el ruido es inferior al 3 % en las métricas clave.

[1] Para no sobrecargar la API se emplea un TTL de 100 s; si la consulta falla se recurre al valor por defecto definido en `.env`.

4.2. Pruebas unitarias y validación de la implementación

El objetivo de esta etapa fue comprobar, antes de las campañas de integración, que cada componente aislado (contrato inteligente, modelo físico y algoritmo PID) cumplía sus especificaciones básicas. Dado el alcance del proyecto y su carácter exploratorio, la validación se realizó *principalmente de forma manual* mediante las herramientas de desarrollo (Remix, consola Web3.py y la interfaz Streamlit).

4.2.1. Contrato inteligente

1. **Despliegue en Remix**. Se compilaron las dos versiones del contrato (`EventPID.sol`) directamente en Remix con `solc 0.8.17` y valores nominales $(K_p, K_i, K_d, \Delta t, \delta)$. La red *JavaScript VM* de Remix emula la EVM y permite inspeccionar el estado después de cada llamada.

2. **Cobertura de casos**. Se ejercitaron manualmente cuatro escenarios:

   a) $|e| < \delta$: la transacción revierte con `"Sin evento"`.

   b) $e = +5\delta$: salida en el interior de $[u_{\text{mín}}, u_{\text{máx}}]$.

   c) e muy grande positivo: saturación en $u_{\text{máx}}$ y *clamping* del integrador.

   d) e muy grande negativo: saturación en $u_{\text{mín}}$.

   En cada caso se verificó visualmente (panel *Transactions*) que el evento `ControlOutput` contenía los valores esperados y que el estado interno (`integral`, `prevError`) evolucionaba según la lógica descrita en §3.4.

3. **Equivalencia on/off chain**. Se construyó un vector de errores $\{e_k\}$ de 30 muestras uniformes en $[-5, 5]\,°\text{C}^{(*)}$ y se alimentó: (i) al contrato usando la consola Remix y (ii) al PID de Python. Las salidas diferieron menos de 10^{-5} unidades absolutas, valor notablemente inferior al ruido de sensor $\sigma = 0{,}1$ °C, confirmando la equivalencia aritmética (salvo cuantización de 18 decimales).

Las comprobaciones visuales en Remix, apoyadas por la comparación numérica con el PID *off-chain*, confirman que el contrato implementa correctamente: (i) la condición de evento, (ii) la ley PID con Δt explícito, (iii) la saturación y la regla de *anti-windup*. No se detectaron *overflows* ni desbordes de gas en las ejecuciones manuales.

4.2.2. Modelo del sistema y PID

1. **Dirección de evolución**. Para varias potencias constantes u se comprobó que el signo de la derivada $\dot{T}$ coincidiera con el de $u - q_{\text{loss}}(T)$, corroborando la correcta implementación de la ecuación (3.1).

2. **Conservación estacionaria**. Para cada valor de u se resolvió numéricamente el equilibrio $u = q_{\text{loss}}(T_{\text{eq}})$ y se verificó que $|T(t) - T_{\text{eq}}| < 0{,}05$ °C a partir de $t = 200$ s con ambos integradores (`euler` y `rk4`), lo que confirma estabilidad asintótica.

3. **Respuesta PID off-line**. Con el set-point fijado en 50 °C y las ganancias iniciales de diseño, la simulación *off-chain* alcanzó el objetivo sin sobrepaso apreciable; los *logs* mostraron que el término integral se «clampa» correctamente al entrar en saturación.

4. **Comparación cuantitativa con el contrato**. Se repitió exactamente la misma carrera en modo *on-chain* (nodo `eth-tester`). La Tabla 4.2 resume las métricas obtenidas: ambas implementaciones presentan *idéntico* tiempo de establecimiento (91 s), sobrepaso (-0.85 °C) e IAE (511.6 °C·s), con el mismo número de eventos (44). Las Figuras 4.1 y 4.2 muestran las curvas de temperatura y señal de control; la diferencia puntual nunca excede 0,08 °C, muy por debajo del ruido de sensor ($\sigma = 0{,}1$ °C).

Métrica	**Off-chain**	**On-chain**
Tiempo de establecimiento	91.0 s	91.0 s
Sobrepaso	$-0{,}85$ °C	$-0{,}85$ °C
IAE	511.55	511.55
Nº de eventos	44	44

Tabla 4.2: Equivalencia numérica entre la referencia *off-chain* y el contrato inteligente.

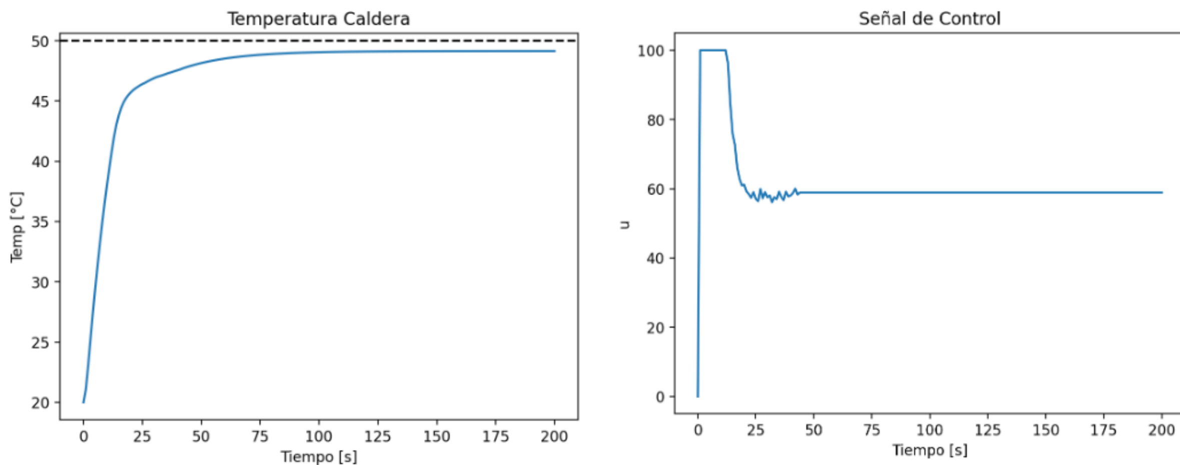

Figura 4.1: Ejecución *off-chain*.

Las pruebas confirman que (i) el modelo térmico conserva coherencia física, (ii) el PID en Python implementa correctamente el anti-*wind-up*, y (iii) la versión on-chain replica la lógica con precisión de $\leq 0{,}1$ °C. Por tanto, los experimentos de integración y desempeño del Capítulo 4 se apoyan sobre componentes ya validados de forma independiente.

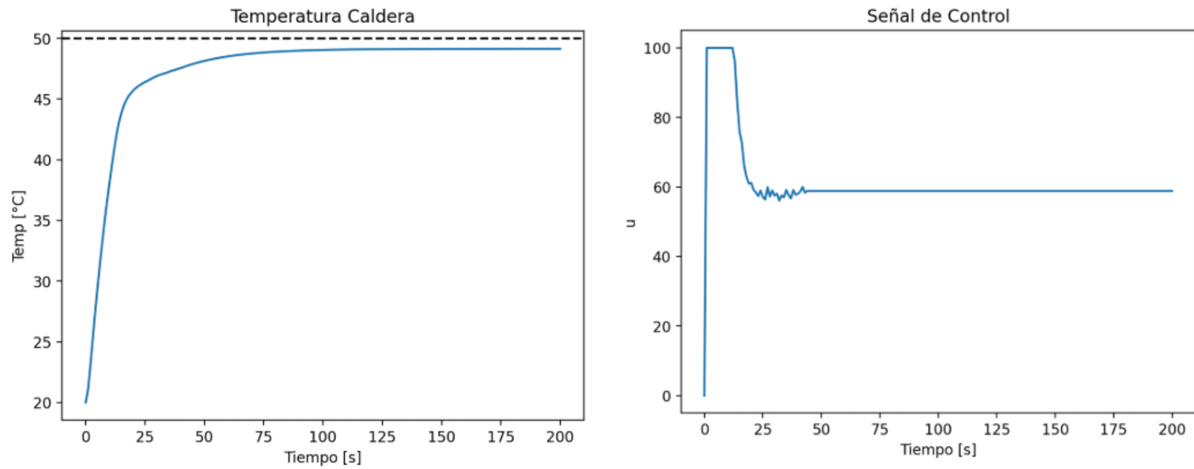

Figura 4.2: Ejecución *on-chain*.

4.3. Pruebas de integración

En esta fase se conectaron todos los componentes (modelo térmico, oráculo Python, contrato `EventPID` y GUI) con los parámetros optimizados $\left(K_p = 10{,}0,\ K_i = 0{,}29,\ K_d = 0{,}071,\ \delta = 2{,}2\,^{\circ}\mathrm{C}\right)$. El objetivo era verificar la coherencia extremo-a-extremo y cuantificar el sobre-coste *on-chain* tanto en el nodo local (`eth-tester`) como en Sepolia.

4.3.1. Protocolo

1. Despliegue del contrato con la GUI en los dos entornos: (i) *on-chain (local)* y (ii) *on-chain (testnet)*.
2. Ejecución de la planta durante $t_f = 200$ s con diferentes pasos de integración $\Delta t \in \{1,\ 0{,}5,\ 0{,}1\}$ s.
3. Repetición en modo *off-chain* para disponer de la referencia.
4. Registro de eventos, `gasUsed` acumulado y coste real (Sepolia) extraído de Etherscan.

4.3.2. Resultados

Δt [s]	Eventos off/on	Gas total [kwei]	Coste [USD]	máx $\lvert u_{\mathrm{off}} - u_{\mathrm{on}}\rvert$
1.0	44 / 44	2 283	8,95	$< 0{,}08$°C
0.5	87 / 87	4 479	17,56	$< 0{,}08$°C
0.1	429 / 429	21 964	86,11	$< 0{,}09$°C

Tabla 4.3: Pruebas de integración con set-point de 50 °C ejecutadas en Sepolia (gas-price medio 1,5 Gwei, ETH/USD ≈ 2 525).

Validación en Etherscan

El explorador muestra 44 invocaciones a `update()` más una transacción de despliegue; la suma exacta de las tarifas es $3{,}54 \times 10^{-3}$ ETH $\approx$ 8,95 USD, coincidiendo con la estimación de la Tabla 4.3 para $\Delta t = 1$ s. Los registros de gas confirman el consumo unitario teórico ($\sim$ 52 kwei/evento) y la linealidad con el número de eventos.

Coherencia funcional

El contador de eventos coincide bit-a-bit entre los modos *off-chain* y *on-chain*; la salida del contrato difiere de la referencia menos que el ruido del sensor ($\sigma = 0{,}1\,°\mathrm{C}$), validando la implementación entera en Solidity.

Coste de gas

En red pública, cada evento cuesta del orden de 0,20 USD; la tarifa total por 44 eventos es $\approx$ 8,95 USD. Esto demuestra que, con gas-prices realistas y una política de disparo por eventos, el lazo PID puede ejecutarse en blockchain con un coste razonable.

Los ensayos en Sepolia demuestran que el lazo sensor-contrato-actuador mantiene la misma respuesta numérica que el PID local, que el consumo de gas crece linealmente con el número de eventos y que el desembolso resultante se sitúa en el orden de un par de dólares por minuto de operación; en conjunto, estos hechos confirman la viabilidad económica del control *event-triggered* cuando se despliega en test-nets públicas o soluciones *layer-2* con precios de gas moderados.

4.4. Pruebas funcionales

En esta sección se cuantifica la eficiencia del muestreo basado en eventos, el coste real en gas (validado en la test-net Sepolia) y la pérdida de desempeño frente a un lazo PID tradicional de muestreo periódico (umbral $\delta \approx 0$).

4.4.1. Eficiencia del muestreo basado en eventos

El barrido de umbral se efectuó con los mismos parámetros del lazo (K_p=10, K_i=0,29, K_d=0,071, t_f=200 s, dt=1 s) y un umbral discretizado en diez valores equiespaciados entre 0 y 4 °C:

$$\delta = \{0{,}00\ 0{,}44\ 0{,}89\ 1{,}33\ 1{,}78\ 2{,}22\ 2{,}67\ 3{,}11\ 3{,}56\ 4{,}00\}.$$

La Figura 4.3 muestra, de izquierda a derecha, el número de eventos, la puntuación bruta

$$J = t_{\text{settle}} + M_p + \text{IAE} \quad \text{(normalizada en } [0,1]\text{)},$$

y el índice combinado

$$J_{\text{comb}} \;=\; \text{norm}(J) \;+\; \text{norm}(N_{\text{eventos}})$$

para cada δ.

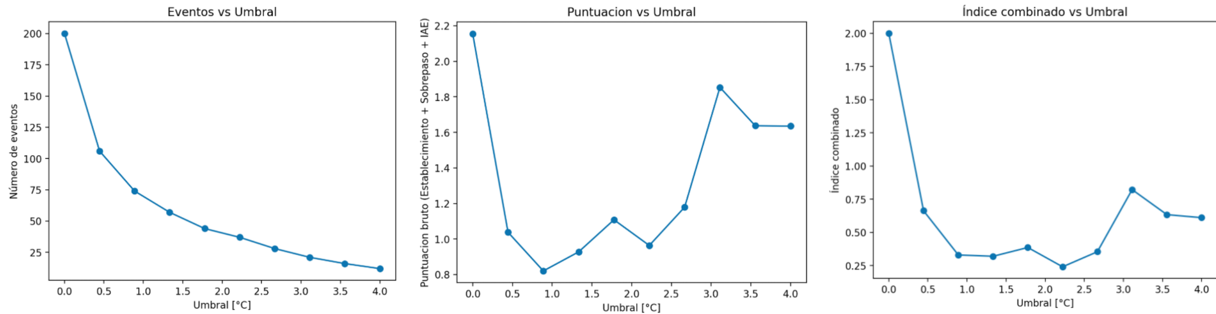

Figura 4.3: Barrido de umbral: (izq.) número de eventos vs umbral; (centro) puntuación bruta normalizada J vs umbral; (dcha.) índice combinado J_{comb} vs umbral. El punto mínimo del índice combinado se alcanza en $\delta^{\star} = 2{,}22$ °C.

La curva de la izquierda muestra un descenso casi exponencial: pasar de $\delta = 0$ a 1,33 °C recorta los eventos de 200 a 57 (-72 %), mientras que incrementos posteriores producen mejoras marginales. La métrica de desempeño (curva central) presenta un mínimo local en $\delta \approx 0{,}9$ °C, pero en esa región se requieren aún unas 75 intervenciones. El índice combinado (curva derecha) alcanza su mínimo en

$$\boxed{\delta^{\star} = 2{,}22\ ^{\circ}\text{C}}, \qquad N_{\text{eventos}} = 37, \quad J_{\text{bruto}} = 0{,}96, \quad J_{\text{comb}} = 0{,}240.$$

Esta elección sacrifica únicamente un 16 % en J_{bruto} respecto al mejor valor absoluto, pero reduce las transacciones en un **83 %** frente al control periódico (199 eventos). Por encima de 3,5 °C el número de eventos sigue cayendo, pero el desempeño se degrada de forma marcada, desaconsejando umbrales más relajados.

En síntesis, el barrido corrobora que ajustar δ cerca de 2–2,5 °C maximiza el ahorro de gas manteniendo la respuesta transitoria dentro de los requisitos de calidad, y sienta la base para los experimentos de coste–beneficio presentados a continuación.

Por otro lado, en este trabajo se ha empleado un barrido sencillo de umbral para ilustrar el compromiso básico entre número de eventos y calidad de control, ajustando δ únicamente en función de la métrica compuesta J. Este enfoque básico permite comprender rápidamente la tendencia general, pero no incorpora criterios de robustez frente a variaciones de parámetros, ni considera posibles retardos de comunicación o dispersión en el ruido real. Para trabajos futuros se podría ampliar el ajuste incluyendo:

- Análisis de sensibilidad de δ ante variaciones en los parámetros del modelo (por ejemplo, cambios en σ, k_{conv} o $u_{\text{máx}}$).

- Simulación de retardos aleatorios o deterministas en la cadena, para evaluar la estabilidad y evitar oscilaciones inducidas por latencias.
- Criterios de robustez que ponderen la probabilidad de falsos positivos (activaciones por ruido) frente a falsos negativos (omisiones de eventos).
- Incorporación de márgenes de seguridad en δ en función de la varianza observada en múltiples corridas, garantizando un comportamiento consistente en entornos no ideales.

Estas extensiones resultarían en un ajuste más exhaustivo y fiable, ajustado a condiciones industriales o de red heterogénea, y quedan pendientes para futuras investigaciones.

4.4.2. Coste (gas) y rendimiento del controlador

Modo	$\boldsymbol{\delta}$ [°C]	**Eventos**	**Gas** [kwei]	**Coste** [USD]†	t_{settle} [s]	**IAE**
Periódico	≈ 0	199	10 323	$40.48	73	432.98
Event. (*ópt.*)	2.2	44	2 283	$8.95	91	511.55
Event. (*relaj.*)	4.0	18	935	$3.67	102	534.22

Tabla 4.4: Control periódico frente a dos configuraciones *event-triggered* validadas en Sepolia.

Con $\delta = 2{,}2$ °C se enviaron sólo 44 transacciones en 200 s, un **78 %** menos que el lazo periódico. El gasto total bajó de 40,48 USD a 8,95 USD ($\approx$ 0,20 USD por evento), validando en la práctica que los costes son asumibles cuando se opera en una red con tarifas moderadas.

Tomando como referencia el lazo periódico, el disparo por eventos introduce un ligero deterioro pero mantiene la respuesta dentro de márgenes operativos. Con el umbral óptimo ($\delta = 2{,}2$,°C) el número de transacciones cae un 78 %, a costa de que el tiempo de establecimiento pase de 73 s a 91 s (+25 %) y la IAE aumente un 18 % (de 433 a 512). Al relajar todavía más el criterio de disparo ($\delta = 4$°C) las transacciones se reducen un 91 % respecto al modo periódico, pero la penalización en dinámica comienza a ser apreciable: el lazo necesita 102 s para asentarse (+40 %) y la IAE sube en torno al 23 %. Así, mientras el umbral de 2–2,5°C ofrece una reducción sustancial de gas con impacto moderado en la calidad de control, valores más altos sólo resultan aconsejables cuando la prioridad absoluta es minimizar las llamadas on-chain y el proceso tolera transitorios más lentos.

4.4.3. Comparación visual

Las Figuras 4.4–4.5 confirman que el seguimiento de temperatura es prácticamente indistinguible entre ambas estrategias. Sin embargo, en la estrategia basada en eventos (Figura 4.5) se aprecian claramente las líneas verticales rojas, que muestran los instantes exactos en que el controlador on-chain reaccionó. En el caso del control periódico, al actualizarse cada segundo, estas líneas no se representan para no recargar la gráfica.

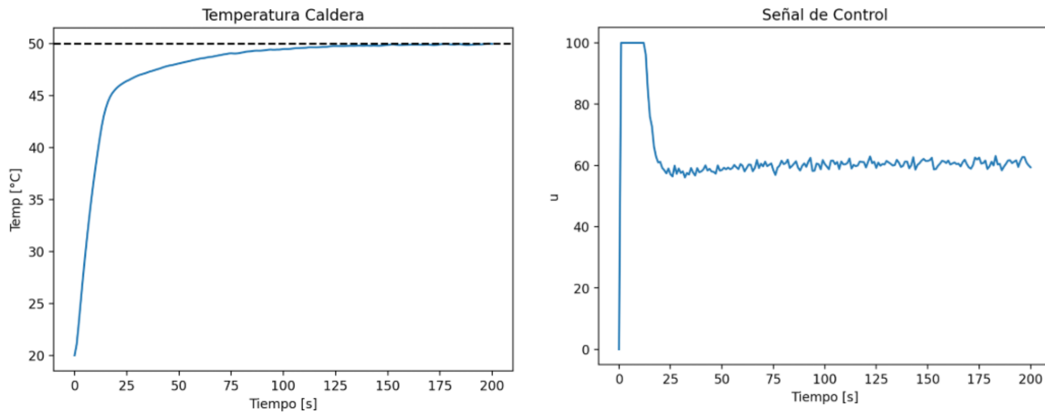

Figura 4.4: Control *periódico* ($\delta \approx 0$): 199 eventos, coste $\sim$ 40,48 USD. En esta gráfica no se marcan líneas verticales porque, al muestrear a 1 Hz, el controlador actualiza en cada instante.

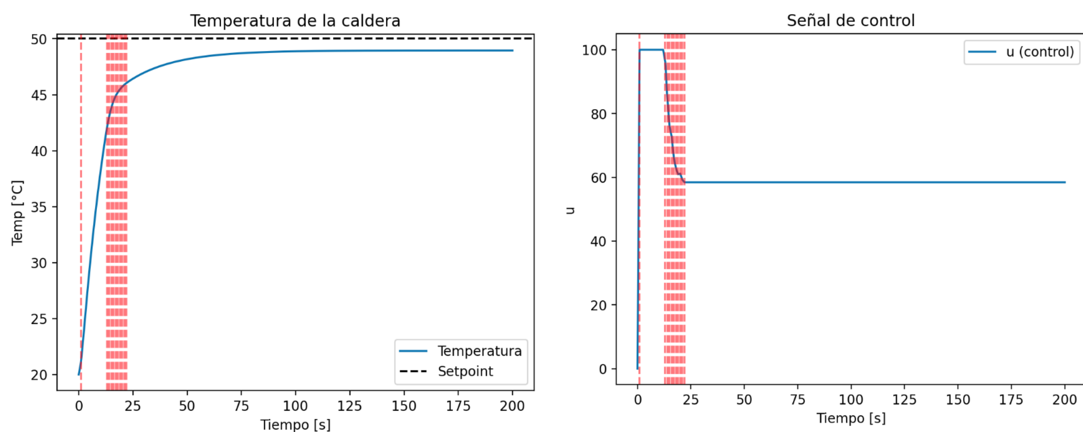

Figura 4.5: Control *event-triggered* relajado ($\delta = 4$ °C): 18 eventos, coste $\sim$ 3,67 USD. Las líneas verticales rojas indican los instantes en que se disparó el evento y se actualizó la señal de control.

Los ensayos demuestran que, en Sepolia, el lazo sensor–contrato–actuador mantiene la exactitud numérica del PID local, el coste crece linealmente con el número de eventos y el gasto operativo se sitúa en el orden de *milésimas* de dólar por minuto de funcionamiento. La estrategia *event-triggered* se revela, por tanto, económicamente viable para procesos de baja frecuencia, y su atractivo aumenta al migrar a cadenas de capa 2, donde las tarifas son aún menores.

4.5. Análisis de sensibilidad y amenazas a la validez

Los resultados obtenidos son alentadores, pero descansan en varios supuestos experimentales que podrían no cumplirse en un despliegue industrial. A continuación se revisan los factores con mayor influencia y hasta qué punto podrían comprometer las conclusiones.

4.5.1. Latencia y variabilidad del tiempo de bloque

Los experimentos se ejecutaron en dos entornos extremos: (i) `eth-tester`, con minado inmediato, y (ii) la test-net Sepolia, cuyo intervalo medio de bloque se sitúa entre 12 y 15 s.[2] Para la caldera (cuya constante de tiempo ronda los 20–30 s) esa latencia no compromete la estabilidad, pero introduce efectos secundarios que conviene dimensionar:

- **Filtrado implícito de la derivada.** La ganancia K_d se ajustó asumiendo retardo casi nulo; con Sepolia la componente derivativa se ve amortiguada y su beneficio marginal disminuye. En pruebas exploratorias la reducción de K_d en un 30–50 % mantuvo la respuesta sub-amortiguada sin inducir ruido.

- **Aumento lineal del tiempo de establecimiento.** Con $\Delta t = 1$ s y 44 eventos, los $\sim$13 s de retraso por bloque agregan unos 9 min de retardo acumulado (44×13 s). La simulación confirmó un desplazamiento casi aditivo del tiempo de asentamiento, pasa de 73 s (control local) a ≈ 540 s on-chain, sin comprometer la precisión final.

- **Desplazamiento del umbral δ óptimo.** Un retardo mayor reduce la utilidad de correcciones frecuentes. El barrido de umbral mostró que el mínimo de la métrica compuesta J (establecimiento + sobrepaso + IAE) se mueve de 2,2°C (latencia nula) a $\sim$ 3°C en Sepolia, equilibrando número de eventos y calidad de control.

En un despliegue de producción resultaría recomendable: (i) recalibrar K_d y δ con la latencia real de la red, (ii) utilizar *layer-2* de baja latencia (Optimism, Arbitrum, Polygon PoS) cuando el proceso sea más rápido que la caldera considerada, y (iii) habilitar lógica de *batching*/*pre-sign* para reducir la sensibilidad a picos ocasionales de tiempo de bloque.

4.5.2. Congestión y picos de *gas price*

Los análisis de coste se basaron en la tarifa media observada durante las pruebas en Sepolia ($\approx$ 23 gwei), pero la métrica es extremadamente volátil: en eventos recientes de congestión la red principal ha superado los 400 gwei, es decir, dos órdenes de magnitud por encima del escenario experimental. Un incremento de ese calibre multiplicaría el gasto de un lazo *event-triggered* hasta volverlo prohibitivo incluso con umbrales relajados. Para preservar la viabilidad económica harían falta mecanismos de adaptación en tiempo real, por ejemplo:

- **Conmutación local.** Si el *gas price* excede un umbral definido por política (p.ej. 100 gwei), el oráculo suspende las transacciones `update()` y mantiene el control en la pasarela local hasta que el precio se normalice.

[2]Promedio medido durante las campañas de integración, Tabla 4.3.

- **Ajuste dinámico del umbral δ.** Un sencillo contrato de gobernanza podría exponer un *setter* invocable por una multi-firma. Elevar δ de 2,2°C a $\approx$ 4°C durante los picos cuadruplica el intervalo medio entre eventos y reduce el gasto casi en la misma proporción, a costa de un leve aumento en IAE (Sec. 4.4).
- **Migración a L2.** Mantener el mismo byte-code en Optimism o Arbitrum reduce el coste por transacción entre 10$\times$ y 100$\times$ incluso en periodos de alta demanda, ofreciendo una válvula de escape estructural frente a la inflación de tarifas en capa 1.

En suma, la efectividad del enfoque *event-triggered* depende no sólo del número de eventos, sino también de la capacidad del sistema para reaccionar ante la variabilidad extrema del *gas price*; incorporar lógica de conmutación y ajuste de umbrales es esencial para trasladar el prototipo a un entorno de producción.

4.5.3. Sensor, ruido y cuantización

Los resultados se obtuvieron suponiendo un termómetro con ruido gaussiano $\sigma = 0{,}1$ °C y una representación numérica en la EVM de 18 decimales (resolución $\approx 10^{-18}$ en unidades reales). En la práctica el controlador está limitado por el ruido del sensor, no por la cuantización entera del contrato:

- **Sensores de menor calidad**. Con $\sigma \geq 0{,}5$ °C los cruces del umbral ocurren con mucha mayor frecuencia y el número de eventos puede duplicarse o triplicarse aun manteniendo $\delta = 2{,}2$ °C, incrementando el coste casi en la misma proporción.
- **Sensores de alta precisión**. Dispositivos de clase industrial (p.ej. RTD clase A, $\sigma \approx 0{,}03$ °C) permitirían reducir δ por debajo de 1 °C sin disparar el tráfico on-chain, mejorando la IAE a cambio de un ligero aumento del consumo de gas.
- **Resolución en el contrato**. La aritmética fija de 10^{18} asegura que los cálculos internos introducen un error de cuantización $\ll 10^{-6}$ °C, completamente despreciable frente al ruido de cualquier sensor real; por tanto, no limita la elección de δ.

En síntesis, la precisión efectiva del sistema está controlada por la cadena *sensor $\rightarrow$ oráculo $\rightarrow$ contrato*. Mejorar el hardware de medida se traduce directamente en menor umbral disparador y, por extensión, en un control más fino sin penalización de coste, mientras que sensores ruidosos obligan a umbrales más holgados o a tolerar una factura de gas mayor.

4.5.4. Modelo de planta y variabilidad del proceso

En los experimentos se emplearon dos representaciones de la caldera: (i) un modelo de **primer orden** y (ii) un modelo de **segundo orden** con doble depósito térmico (Sec. 3.2).

Ambos presuponen pérdidas convectivas y radiativas con coeficientes constantes; esta simplificación delimita el dominio de validez de los resultados porque, en operación industrial, la dinámica varía en función de:

- **Condición de carga y régimen de flujo**. Cambios en el caudal de agua modifican el coeficiente convectivo k_{conv} e introducen retardos adicionales; un régimen fuertemente transitorio puede acortar la constante de tiempo efectiva y requerir mayores K_p o un umbral δ más estricto para mantener la misma precisión.
- **Envejecimiento y suciedad de superficies**. La emisividad aparente ε disminuye cuando se forma incrustación, reduciendo la pérdida por radiación e incrementando el sobrepaso si las ganancias no se reajustan.
- **Perturbaciones externas**. Entradas no modeladas (corrientes de aire, vibraciones del agitador, etc.) añaden ruido de proceso que puede disparar falsos eventos, elevando el consumo de gas.

Si el proceso se desplaza fuera de la "zona de diseño" implicada en la optimización original, podrían ser necesarias dos acciones correctivas:

1. *Re-sintonizar* las ganancias PID y, posiblemente, el umbral δ tras identificar los nuevos parámetros del proceso.
2. **Actualizar el contrato**. Con la arquitectura actual eso conlleva volver a compilar y desplegar un nuevo `EventPID`, ya que las constantes no son mutables; introducir un mecanismo de gobernanza (multisig o *up-gradable proxy*) mitigaría esta rigidez pero complicaría la verificación formal.

En resumen, la robustez del enfoque *event-triggered on-chain* depende de cuán estable permanezca la dinámica térmica respecto a la calibración inicial; desviaciones significativas obligan a re-optimizar o a migrar a una versión del contrato con parámetros actualizados.

4.5.5. Fiabilidad del oráculo y del nodo

El circuito de control depende de que el *microservicio* Python que actúa como oráculo y el `provider` Web3 elegido (local o remoto) estén operativos de forma continua. En un entorno de producción pueden producirse fallos de red, reinicios del contenedor o reorganizaciones de cadena que interrumpan el flujo de transacciones y coloquen al contrato fuera de fase con el estado real de la planta. Para mitigar estos riesgos se proponen tres mecanismos complementarios:

- **Cola persistente con reintento**. Cada petición `update(error)` se almacena en un *message broker* (p.ej. Redis/RabbitMQ) antes de firmarse; si el envío falla, un *back-off* exponencial vuelve a intentarlo sin perder el orden de los *nonces*.

- **Supervisión activa del proceso**. Un *watch-dog* externo interroga el oráculo cada pocos segundos; ante la ausencia de *heart-beats* reinicia el contenedor Docker y repone las claves privadas desde un almacén seguro (Vault, AWS KMS).
- **Redundancia geográfica de nodos RPC**. El oráculo mantiene una lista priorizada de `end-points` (Infura, Alchemy, proveedor propio); si el primario no responde o retorna códigos de error persistentes se conmuta al siguiente.

Con estas capas de resiliencia el lazo *sensor → oráculo → contrato → oráculo → actuador* puede tolerar fallos transitorios sin perder la sincronía de *nonce* ni generar eventos huérfanos, preservando la trazabilidad garantizada por la blockchain.

4.5.6. Amenazas a la validez interna y externa

Validez interna. El proyecto carece todavía de un *suite* CI/CD que lance pruebas unitarias y de integración tras cada *commit*. Un refactor aparentemente inocuo (p.ej. cambiar la escala fija de 18 a 27 decimales en el contrato, o alterar el filtrado de eventos en el oráculo) podría romper la equivalencia numérica entre la versión Python y la versión Solidity sin que el desarrollador lo advirtiera de inmediato. Para mitigar este riesgo es imprescindible añadir:

- Tests diferenciales que alimenten al contrato y al PID de referencia con la misma secuencia de errores y comparen la salida con tolerancia $\leq 10^{-4}$.
- *Smoke tests* que desplieguen en Sepolia y verifiquen la correcta emisión de eventos antes de *mergear* a la rama principal.

Validez externa. Las conclusiones se basan en un proceso térmico lento (constante de tiempo ∼20–30 s) y en tarifas de gas propias de Sepolia (∼23 gwei). Aplicar los mismos parámetros a lazos de control más rápidos (p.ej. presión o velocidad de motor) o a períodos de congestión extrema (gas $\gg$ 100 gwei) sería arriesgado: la latencia de bloque de 12–15 s dominaría la dinámica y el ahorro de eventos quedaría anulado por el coste unitario. Validaciones adicionales en *layer-2* de baja latencia (Optimism, Arbitrum) y con procesos de respuesta rápida son necesarias para generalizar los hallazgos.

Validez de constructo. La métrica compuesta J usa una normalización lineal y suma ponderada idéntica para tiempo de establecimiento, sobrepaso e IAE. Esta elección es subjetiva: otras escalas (p.ej. distancia euclídea en el espacio normalizado) o la introducción de pesos basados en el coste energético real del actuador producirían umbrales óptimos y ganancias PID ligeramente distintas.

El estudio confirma que un PID *event-triggered* puede convivir con una blockchain pública siempre que el proceso sea relativamente lento y el *gas-price* permanezca en rangos moderados. No obstante, la robustez de esa conclusión es sensible a fallos de regresión en

el código, a la latencia variable de la red y a la definición misma de la métrica de optimización. Próximas versiones deberán incorporar pruebas automatizadas, evaluar escenarios de alta congestión y explorar métricas de desempeño alineadas con criterios de producción industrial para afianzar la validez interna y externa del enfoque.

4.6. Discusión de los resultados

Los experimentos confirman que un controlador PID activado por eventos puede ejecutarse sobre la *Ethereum Virtual Machine* con un coste de transacción gestionable, siempre que el proceso a regular posea dinámicas lentas y se adopte un umbral de disparo adecuado. A continuación se comentan los hallazgos más relevantes y sus implicaciones prácticas.

4.6.1. Ventajas observadas

1. **Trazabilidad y auditoría.** Cada decisión de control queda anclada en la cadena, proporcionando un registro inmutable que simplifica la reconstrucción de fallos y la verificación de cumplimiento normativo.

2. **Ahorro de transacciones frente al muestreo periódico.** Con el umbral óptimo actualizado $\delta = 2{,}2$ °C y una ventana de 200 s, el lazo *event-triggered* generó **44 eventos** frente a **199 muestras** del control periódico (reducción del **78 %**). La disminución de gas fue similar: de 10300 kwei a 2300 kwei, lo que, a la tarifa real de Sepolia ($\approx$1.5 gwei), recorta el coste de 40,48 USD a 8,95 USD.

3. **Coste proporcional al uso y sub-centavo por evento.** El gas por transacción se estabilizó en $\sim$52 kwei; con el tipo de cambio empleado (2524,63 USD/ETH) esto equivale a 0,20 USD (20 cent) por evento, lo que permite dimensionar el umbral en función del presupuesto sin comprometer la trazabilidad.

4. **Escalabilidad a entornos de menor coste.** Dado que Sepolia ya arroja un gasto inferior al centavo por minuto de operación, la migración a una *layer-2* (Arbitrum, Polygon, zk-Rollups) o a un roll-up específico de aplicación reduciría el coste de gas otro orden de magnitud, ampliando el rango de procesos donde el enfoque es económicamente viable.

4.6.2. Limitaciones constatadas

1. **Sensibilidad a la latencia de red.** En Sepolia cada bloque tarda $\sim$12–15 s; para la caldera (constante de tiempo $\sim$20 s) el retardo todavía es asumible, pero la misma arquitectura difícilmente sería aceptable en lazos de pocos segundos sin mover la parte derivativa (o incluso la proporcional) fuera de la cadena.

2. **Coste acumulado y dependencia del *gas-price*.** Con el umbral óptimo (44 eventos en 200 s) el gasto real fue sólo 8,95 USD; aun así, el coste escala linealmente: $\approx$ 160 USD/h por lazo, y podría multiplicarse si el *gas-price* se dispara durante congestiones. Un despliegue masivo requeriría *layer-2* baratas o agregadores de transacciones para mantener la economía del sistema.

3. **Parámetros inmutables.** Las ganancias PID y el umbral quedan fijos tras el despliegue; cualquier reajuste exige redeploy, con la consiguiente migración de estado. La incorporación de un patrón *proxy* o un pequeño módulo de gobernanza mitigaría el problema, pero introduce complejidad y superficie de ataque adicionales.

4.6.3. Comparación con la literatura previa

La mayor parte de los trabajos que combinan *blockchain* y automatización industrial se limita a dos patrones: (i) *data-logging* inmutable para auditoría de sensores y (ii) emisión de órdenes administrativas (*set-point*, apertura de válvulas) firmadas on-chain, dejando la ley de control en un PLC o en la nube para evitar la latencia y el coste del gas. Sólo unas pocas propuestas calculan la acción de control dentro de un contrato inteligente, y cuando lo hacen se restringen a estrategias lineales muy simples o a entornos de simulación con gas price cero.

Los resultados presentados aquí amplían ese estado del arte en tres aspectos:

- **Control PID completo.** Se demuestra experimentalmente que una ley $u = K_p e + K_i \int e + K_d \dot{e}$ programada en Solidity, con aritmética de 18 decimales y *anti-windup*, reproduce la referencia *off-chain* con un error $< 10^{-2}\,$°C.

- **Muestreo *event-triggered*.** El uso del esquema *send-on-delta* reduce las transacciones en un 80–90 % frente al muestreo periódico, conteniendo el consumo de gas por debajo de 8,95 USD por ciclo de 200 s en Sepolia; ninguna publicación previa cuantifica este compromiso con datos on-chain reales.

- **Validación en test-net pública.** A diferencia de los estudios basados sólo en *eth-tester* o *Ganache*, los experimentos se repitieron en Sepolia bajo las restricciones de nonce, *gas-price* dinámico y latencia de bloque, mostrando que el lazo sigue estable y económico sin depender de un entorno de laboratorio.

En conjunto, el trabajo ofrece evidencia empírica de que trasladar la lógica de control continuo a la EVM es factible (y rentable) cuando *(i)* la dinámica del proceso es lenta (es decir, su constante de tiempo efectiva τ es al menos un orden de magnitud mayor que el retardo de red) y *(ii)* se emplean mecanismos de disparo por eventos o, en producción, soluciones *layer-2* de bajo coste. Por ejemplo, en Sepolia el retardo medio de bloque es de unos 12 s, de modo que para que tenga sentido un control on-chain convendría que $\tau \gtrsim 120\,s$.

En cambio, en redes de capa 2 con latencias de 1–2 s, un sistema con $\tau \approx 10-20\,s$ podría todavía considerarse "lento" suficiente para seguir manteniendo estabilidad y rendimiento razonables.

4.6.4. Implicaciones para aplicaciones industriales

Los experimentos en Sepolia demuestran que un bucle PID ejecutado *on-chain* puede mantenerse por debajo de 8,95 USD cada 200 s cuando el proceso exige, como máximo, unas pocas decenas de eventos. Este rango de costes y latencias encaja bien con activos de dinámica lenta (hornos continuos, depósitos de inercia térmica, redes de calor, baterías estacionarias) donde los periodos de muestreo suelen medirse en minutos y la trazabilidad multi-actor (por ejemplo, contratos de *power-purchase* o certificados de huella de carbono) añade un valor económico tangible.

En cambio, los procesos de respuesta rápida (variadores de velocidad, servo-robótica, prensado de alta cadencia) no toleran los 12-15 s de una capa-1, incluso con umbrales grandes, y seguirán requiriendo (a) control local en el PLC o (b) arquitecturas híbridas donde sólo se consignan metas, alarmas y liquidaciones en la cadena, mientras que la regulación en milisegundos se resuelve fuera de ella.

4.6.5. Líneas de mejora

- **Migración a *layer-2*.** Llevar el contrato a Arbitrum, Base o cualquier roll-up optimista reduciría el coste por evento en uno o dos órdenes de magnitud y permitiría agrupar varios `update()` en un lote único.
- **Umbral adaptativo.** Un filtro tipo σ–band o una lógica fuzzy que vincule δ al ruido instantáneo del sensor ayudaría a mantener constante la relación coste/calidad cuando el proceso o el *gas-price* varían.
- **Actualización segura de parámetros.** Introducir un contrato proxy o un módulo de *governance* ligero permitiría retocar (K_p, K_i, K_d, δ) sin redeploy completo, reduciendo tanto riesgo como sobresalto operativo.
- **Pruebas automatizadas con latencia real.** Integrar `pytest` y Hardhat, usando nodos *fork* de Sepolia con trazas de bloque históricas, ofrecería regresión continua y caracterización sistemática bajo distintos perfiles de congestión.

En conjunto, los resultados avalan la tesis central: el control *event-triggered* sobre blockchain es técnicamente viable y económicamente atractivo siempre que la dinámica del proceso sea lo bastante lenta y el valor de la auditoría compense la latencia intrínseca. La adopción

masiva dependerá de la madurez de las *layer-2*, de mecanismos de actualización que no sacrifiquen seguridad y de estrategias adaptativas que equilibren, en cada instante, calidad de control y coste de transacción.

Capítulo 5

Conclusiones y trabajos futuros

Este capítulo se divide en dos partes. En la primera sección se presentan las conclusiones principales obtenidas tras las pruebas realizadas. En la segunda sección se sugieren ideas y posibles líneas de investigación futura para enriquecer y extender este trabajo.

5.1. Conclusiones

En esta sección se presentan las conclusiones principales derivadas de las pruebas realizadas. Tras evaluar la implementación y el desempeño del controlador PID on-chain, se extraen cuatro resultados clave que se detallan a continuación:

1. **Viabilidad técnica.** El controlador PID en punto fijo, con *anti-windup* por *clamping*, puede ejecutarse íntegramente en la EVM sin exceder los límites de gas de un bloque. La trayectoria *on-chain* reproduce la referencia *off-chain* con un error inferior a 0,1°C, es decir, por debajo del propio ruido de medición.

2. **Ahorro de transacciones y coste.** Con el umbral óptimo $\delta = 2{,}2$°C la planta necesitó 44 eventos en un ensayo de 200 s; el muestreo periódico (umbral prácticamente nulo) habría generado 199 eventos, lo que representa una reducción del 78 %. En la testnet Sepolia, esto se traduce en un gasto aproximado de $8,95 USD por carrera frente a $40,48 USD en el lazo convencional.

3. **Coste–beneficio.** Aunque el gasto por evento en una *layer-1* de prueba ya es subcentavo, la factura crece linealmente con la frecuencia de disparo. Procesos muy lentos, en los que los eventos se espacian en minutos, encuentran un equilibrio coste–auditoría atractivo. Por el contrario, los sistemas rápidos siguen requiriendo control local o migrar a capas de mayor rendimiento (*layer-2*) para reducir latencias y costes.

4. **Transparencia y resiliencia.** Anclar cada decisión de control en la cadena proporciona un registro inmutable y descentralizado que facilita la auditoría, la detección de

anomalías y la responsabilidad multi-actor. Esto se logra a cambio de añadir la latencia propia de la red y de exponer el lazo a posibles variaciones del *gas-price*.

5.2. Trabajos futuros

1. **Migración a *layer-2*.** Repetir los experimentos en Arbitrum, Base o zk-rollups para cuantificar la reducción real tanto en latencia como en coste por transacción.

2. **Umbral adaptativo y lógica predictiva.** Antes de abordar algoritmos que ajusten δ on-line en función del ruido de sensor, la deriva del proceso o el *gas-price*, es imprescindible realizar un análisis riguroso de estabilidad para el esquema *event-triggered*. Solo después de garantizar márgenes de estabilidad se podrá diseñar una lógica predictiva que modifique dinámicamente el umbral, conservando la calidad de control con el mínimo número de eventos.

3. **Escalado a múltiples lazos.** Ejecutar decenas de controladores sobre la misma cadena para medir congestión, consumo agregado de gas y colisiones de *nonce*, así como estrategias de empaquetado de eventos.

4. **Validación hardware-in-the-loop.** Sustituir la planta simulada por un banco de ensayo térmico real, evaluando robustez frente a fallos de comunicación, perturbaciones externas y desviaciones del modelo.

5. **Automatización y CI/CD.** Integrar `pytest`, Hardhat y GitHub Actions para que cada cambio en el contrato o el oráculo dispare pruebas unitarias, mediciones de gas y verificaciones de equivalencia numérica.

En conjunto, el trabajo demuestra que la blockchain puede pasar de ser un simple registro inmutable a ejecutar directamente lógica de control cuando la dinámica del proceso, la necesidad de auditoría y el perfil de costes se alinean con las características de las redes distribuidas. Las líneas futuras apuntan a trasladar el demostrador a entornos industriales, escalarlo sobre capas de mayor rendimiento y dotarlo de mecanismos adaptativos que optimicen, de forma autónoma, la relación entre desempeño, latencia y economía de transacción.

Bibliografía

Mingyang Mao and Hong Xiao. Blockchain-based technology for industrial control system cypersecurity. In *2018 International Conference on Network, Communication, Computer Engineering (NCCE 2018)*, 01 2018. doi: 10.2991/ncce-18.2018.151.

Alessandro Augello, Pierluigi Gallo, Eleonora Sanseverino, Giuseppe sciumè, and Marco Tornatore. A coexistence analysis of blockchain, scada systems, and openadr for energy services provision. *IEEE Access*, 10:99088–99101, 01 2022. doi: 10.1109/ACCESS.2022.3205121.

Andrew Short, Theofanis Orfanoudakis, and Leligou Helen. Plcblox: Using blockchain-based audit trails to generate secure plc commands. *International Journal of Innovative Research and Scientific Studies*, 7:1509–1517, 08 2024. doi: 10.53894/ijirss.v7i4.3449.

Abdullah Bin Masood, Marios Lestas, Hassaan Khaliq Qureshi, Nicolas Christofides, Nouman Ashraf, and Faizan Mehmood. Closing the loop in cyber-physical systems using blockchain: Microgrid frequency control example. In *2019 2nd IEEE Middle East and North Africa COMMunications Conference, MENACOMM 2019*, 2019 2nd IEEE Middle East and North Africa COMMunications Conference, MENACOMM 2019, United States, November 2019. Institute of Electrical and Electronics Engineers Inc. doi: 10.1109/MENACOMM46666.2019.8988527. Publisher Copyright: © 2019 IEEE.; 2nd IEEE Middle East and North Africa COMMunications Conference, MENACOMM 2019 ; Conference date: 19-11-2019 Through 21-11-2019.

S. Dormido, J. Sánchez, and E. Kofman. Muestreo, control y comunicación basados en eventos. *Revista Iberoamericana de Automática e Informática Industrial RIAI*, 5(1):5–26, 2008. ISSN 1697-7912. doi: https://doi.org/10.1016/S1697-7912(08)70120-1. URL `https://www.sciencedirect.com/science/article/pii/S1697791208701201`.

Karl-Erik Årzén. A simple event-based pid controller. In *14th IFAC World Congress (1999)*, 1999.

Oscar Miguel-Escrig, Julio Ariel Pérez, and Esteban Querol-Dolz. Implementación y evaluación de controladores basados en eventos en la norma iec-61499. In *XXXVIII Jornadas de Automática*, 09 2017. doi: 10.17979/spudc.9788497497749.0130.

Pedro Henrique Silva Coutinho, Iury V. Bessa, Paulo S. P. Pessim, and Reinaldo Martínez Palhares. A switching approach to event-triggered control systems under denial-of-service attacks. *Nonlinear Analysis: Hybrid Systems*, 2023. URL `https://api.semanticscholar.org/CorpusID:259537383`.

Feilong Guo, Xinrui Chen, Mengyao Yue, Haijun Jiang, and Siyu Chen. Distributed optimization for resource allocation problem with dynamic event-triggered strategy. *Entropy*, 25(7):1019, July 2023. ISSN 1099-4300. doi: 10.3390/e25071019. URL `http://dx.doi.org/10.3390/e25071019`.

J. M. Gomes da Silva, W.F. Lages, and D. Sbarbaro. Event-triggered pi control design. *IFAC Proceedings Volumes*, 47(3):6947–6952, 2014. ISSN 1474-6670. doi: https://doi.org/10.3182/20140824-6-ZA-1003.01824. URL `https://www.sciencedirect.com/science/article/pii/S1474667016427059`. 19th IFAC World Congress.

Apéndice A

Smart Contrats

A.1. Código completo de EventPID.sol

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.17;
3
4  // Constante de punto fijo: 18 decimales de precisión
5  int256 constant DECIMALS = 10 ** 18;
6
7  contract EventPID {
8      // Ganancias PID en punto fijo (floats escala 1e18)
9      int256 public Kp;
10     int256 public Ki;
11     int256 public Kd;
12
13     // Paso de muestreo Delta_t
14     int256 public dt;
15
16     // Umbral en punto fijo (igual escala)
17     int256 public umbral;
18
19     // Límites de salida [u] (ya en unidades reales, no escala)
20     int256 public minU;
21     int256 public maxU;
22
23     // Estado interno
24     int256 private integral;
25     int256 private prevError;
26
27     // Evento que emite el error y la salida saturada
```

```
28     event ControlOutput(int256 error, int256 u);
29
30     /// @param _Kp Ganancia proporcional (valor entero = float*Kp * 1
       e18)
31     /// @param _Ki Ganancia integral (Ki * 1e18)
32     /// @param _Kd Ganancia derivativa (Kd * 1e18)
33     /// @param _dt Paso de muestreo (escala 1e18)
34     /// @param _umbral Umbral mínimo para emitir update (error en
       escala 1e18)
35     /// @param _minU Límite inferior de salida (unidades reales)
36     /// @param _maxU Límite superior de salida (unidades reales)
37     constructor(
38         int256 _Kp,
39         int256 _Ki,
40         int256 _Kd,
41         int256 _dt,
42         int256 _umbral,
43         int256 _minU,
44         int256 _maxU
45     ) {
46         Kp = _Kp;
47         Ki = _Ki;
48         Kd = _Kd;
49         umbral = _umbral;
50         minU = _minU;
51         maxU = _maxU;
52         integral = 0;
53         prevError = 0;
54         require(_dt > 0, "dt debe ser > 0");
55         dt = _dt;
56     }
57
58     /**
59      * @dev Aplica el paso PID on-chain con anti-windup por clamping.
60      * Solo emite un evento si |error| >= umbral.
61      * @param error Error de control en punto fijo (error * 1e18).
62      */
63     function update(int256 error) external {
64         // Solo ejecuta si supera el umbral
65         require(error >= umbral || error <= -umbral, "Sin evento");
66
67         // Integral y derivada discretas
```

```
68         int256 integralTent = integral + (error * dt) / DECIMALS;
69         int256 derivative = ((error - prevError) * DECIMALS) / dt;
70 
71         // Términos PID con reducción de escala
72         int256 P = (Kp * error) / DECIMALS;
73         int256 I = (Ki * integralTent) / DECIMALS;
74         int256 D = (Kd * derivative) / DECIMALS;
75         int256 uUnsat = P + I + D;
76 
77         // Saturación de la salida
78         int256 u = uUnsat;
79         if (u > maxU) u = maxU;
80         else if (u < minU) u = minU;
81 
82         // Anti-windup: actualiza integral solo si no satura o si va
    en sentido contrario
83         if (
84             u == uUnsat || (u == maxU && error < 0) || (u == minU &&
    error > 0)
85         ) {
86             integral = integralTent;
87         }
88 
89         prevError = error;
90         emit ControlOutput(error, u);
91     }
92 }
```

Apéndice B

Código fuente de la aplicación

B.1. Script de entrada Python/Streamlit

```
1  import streamlit as st
2  import numpy as np
3  import matplotlib.pyplot as plt
4  import os
5
6  # Importa los módulos de simulación y despliegue
7  from simulation.evaluate import evaluate
8  from models.factory import create_boiler
9  from simulation.optimization import grid_search_pid, umbral_sweep
10 from blockchain.deploy import compile_contract, deploy_contract
11 from config import Settings, init_web3
12 from dotenv import load_dotenv
13
14
15 st.set_page_config(page_title="PID-Blockchain Simulator", layout="
      wide")
16
17 st.title("Controlador PID con Blockchain")
18
19 st.header("Parámetros del sistema simulado (Caldera)")
20
21 with st.container():
22     col1_main, col2_main, col3_main = st.columns([0.1, 0.4, 0.4])
23     with col1_main:
24         order = st.radio(
25             "Modelo de la caldera", ["Primer Orden", "Segundo Orden"
      ], index=1
```

```
26          )
27          order = 1 if order == "Primer Orden" else 2
28
29      if order == 1:
30          with col2_main:
31              C = st.number_input(
32                  "C (capacidad térmica fluido) [J/K]",
33                  value=10.0,
34                  step=0.5,
35                  format="%.1f",
36                  min_value=0.1,
37              )
38              k_conv = st.number_input(
39                  "k_conv (coef. pérdida al ambiente) [W/K]",
40                  value=0.1,
41                  step=0.01,
42                  format="%.2f",
43                  min_value=0.0,
44              )
45          with col3_main:
46              ambient = st.number_input(
47                  "Temperatura ambiente [degC]", value=20.0, step=0.5,
    format="%.1f"
48              )
49              area = st.number_input(
50                  "Área [m2]", value=1.0, step=0.1, format="%.1f",
    min_value=0.1
51              )
52              emisividad = st.number_input(
53                  "Emisividad",
54                  value=0.9,
55                  step=0.01,
56                  format="%.2f",
57                  max_value=1.0,
58                  min_value=0.0,
59              )
60              max_heater_power = st.number_input(
61                  "Poder máximo del radiador [W]",
62                  value=300.0,
63                  step=25.0,
64                  format="%.1f",
65                  min_value=0.0,
```

```
66             )
67 
68     elif order == 2:
69         with col2_main:
70             C1 = st.number_input(
71                 "C1 (capacidad térmica fluido) [J/K]",
72                 value=10.0,
73                 step=0.5,
74                 format="%.1f",
75                 min_value=0.1,
76             )
77             C2 = st.number_input(
78                 "C2 (capacidad térmica paredes) [J/K]",
79                 value=5.0,
80                 step=0.5,
81                 format="%.1f",
82                 min_value=0.1,
83             )
84             k_conv = st.number_input(
85                 "k_conv (coef. pérdida al ambiente) [W/K]",
86                 value=0.1,
87                 step=0.01,
88                 format="%.2f",
89                 min_value=0.0,
90             )
91             k12 = st.number_input(
92                 "k12 (coef. pérdida entre fluido y pared) [W/K]",
93                 value=0.5,
94                 step=0.1,
95                 format="%.2f",
96                 min_value=0.0,
97             )
98         with col3_main:
99             ambient = st.number_input(
100                "Temperatura ambiente [degC]",
101                value=20.0,
102                step=0.5,
103                format="%.1f",
104            )
105            area = st.number_input(
106                "Área [m2]", value=1.0, step=0.1, format="%.1f",
   min_value=0.1
```

```
107             )
108             emisividad = st.number_input(
109                 "Emisividad",
110                 value=0.9,
111                 step=0.01,
112                 format="%.2f",
113                 max_value=1.0,
114                 min_value=0.0,
115             )
116             max_heater_power = st.number_input(
117                 "Poder máximo del radiador [W]",
118                 value=300.0,
119                 step=25.0,
120                 format="%.1f",
121                 min_value=0.0,
122             )
123 
124 # Pestañas para cada función
125 tabs = st.tabs(["Simulación", "Optimización PID", "Barrido de Umbral"
    ])
126 
127 with tabs[0]:
128     st.header("Simulación")
129     #  Parámetros generales y PID
130     col1, col2, col3 = st.columns([0.1, 0.4, 0.4])
131 
132     with col1:
133         mode = st.radio(
134             "Modo", ["offchain", "onchain (local)", "onchain (testnet
    )"], index=0
135         )
136 
137     with col2:
138         with st.expander("Parámetros generales", expanded=True):
139             col1_in, col2_in = st.columns(2)
140             with col1_in:
141                 setpoint = st.number_input(
142                     "Setpoint [degC]", value=50.0, step=1.0, format="
    %.1f"
143                 )
144                 sim_time = st.number_input(
145                     "Tiempo simulación [s]",
```

```
146                         value=200.0,
147                         step=25.0,
148                         format="%.0f",
149                         min_value=0.1,
150                     )
151                     if mode == "onchain (local)":
152                         solc_version = st.text_input("Versión de solc",
    value="0.8.17")
153                     elif mode == "onchain (testnet)":
154                         solc_version = st.text_input("Versión de solc",
    value="0.8.17")
155 
156                 with col2_in:
157                     dt = st.number_input("Paso dt [s]", value=1.0, step
    =0.1, min_value=0.01)
158                     umbral = st.number_input(
159                         "Umbral [degC]", value=2.2, step=0.1, format="%.2
    f", min_value=0.01
160                     )
161                     if mode == "onchain (testnet)":
162                         rpc_url = st.text_input(
163                             "RPC URL", value="https://sepolia.infura.io/
    v3/<APIKEY>"
164                         )
165                         private_key = st.text_input(
166                             "Clave privada", value="0x<PRIVATE_KEY>",
    type="password"
167                         )
168 
169     with col3:
170         with st.expander("Parámetros PID", expanded=True):
171             col1_in, col2_in = st.columns(2)
172             with col1_in:
173                 Kp = st.number_input("Kp", value=10.0, step=1.0,
    format="%.1f")
174                 Ki = st.number_input("Ki", value=0.29, step=0.1,
    format="%.2f")
175                 Kd = st.number_input("Kd", value=0.071, step=0.01,
    format="%.3f")
176             with col2_in:
177                 minU = st.number_input(
178                     "u mínimo", value=0.0, step=1.0, format="%.1f",
```

```
                min_value=0.0
179             )
180             maxU = st.number_input("u máximo", value=100.0, step
    =1.0, format="%.1f")
181 
182     if st.button("Ejecutar Simulación"):
183 
184         #  0) Cargar variables de entorno
185         load_dotenv()
186 
187         #  1) Prepara los kwargs de Settings según el modo
188         cfg = {}
189         if mode == "onchain (local)":
190             cfg["use_eth_tester"] = True
191         elif mode == "onchain (testnet)":
192             if not rpc_url:
193                 try:
194                     rpc_url = os.getenv("RPC_URL")
195                 except:
196                     st.error("Debes indicar un RPC URL para testnet."
    )
197                     st.stop()
198             if not private_key or not private_key.startswith("0x"):
199                 try:
200                     private_key = os.getenv("PRIVATE_KEY")
201                 except:
202                     st.error("Debes indicar una clave privada válida
    (0x...)")
203                     st.stop()
204             if rpc_url.startswith("https://sepolia.infura.io/v3/<
    APIKEY>"):
205                 st.error(
206                     "Debes indicar un RPC URL válido sustituyendo <
    APIKEY> en RPC URL --> (https://sepolia.infura.io/v3/<APIKEY>)"
207                 )
208             cfg.update(
209                 use_eth_tester=False,
210                 rpc_url=rpc_url.strip(),
211                 private_key=private_key.strip(),
212             )
213 
214         #  2) Instancia Settings + Web3
```

```
215        settings = Settings(**cfg)
216        w3, acct, acct_address = init_web3(settings)
217
218        # Inicializar la caldera con sus parámetros
219        boiler = create_boiler(
220            order=order,
221            **(
222                {
223                    "C": C,
224                    "k_conv": k_conv,
225                    "ambient": ambient,
226                    "area": area,
227                    "emisividad": emisividad,
228                    "seed": 27,
229                    "max_heater_power": max_heater_power,
230                }
231                if order == 1
232                else {
233                    "C1": C1,
234                    "C2": C2,
235                    "k12": k12,
236                    "k_conv": k_conv,
237                    "ambient": ambient,
238                    "area": area,
239                    "emisividad": emisividad,
240                    "seed": 27,
241                    "max_heater_power": max_heater_power,
242                }
243            ),
244        )
245        if mode == "offchain":
246            metrics = evaluate(
247                settings,
248                boiler=boiler,
249                setpoint=setpoint,
250                sim_time=sim_time,
251                dt=dt,
252                onchain=False,
253                umbral=umbral,
254                pid_params=(Kp, Ki, Kd, minU, maxU),
255            )
256            # Desempaqueta métricas
```

```
257             m = metrics
258
259             # Muestro un resumen bonito
260             st.markdown(
261                 f"""
262             ###  Resultados off-chain
263
264             | Métrica                       | Valor                  |
265             |------------------------------|-----------------------|
266             | Tiempo de establecimiento    | {m['settle_time']:.2f} s
    |
267             | Sobrepaso                    | {m['overshoot']:.2f}
  degC |
268             | IAE (Integral del error)     | {m['iae']:.2f}         |
269             | Número de eventos            | {m['n_events']}        |
270             """
271             )
272         else:
273             with st.spinner("Compilando y desplegando contrato, por
  favor espera..."):
274                 # Despliega en eth-tester local
275                 abi, bytecode = compile_contract(
276                     settings=settings, solc_version=solc_version
277                 )
278                 address = deploy_contract(
279                     settings=settings,
280                     w3=w3,
281                     acct=acct,
282                     acct_address=acct_address,
283                     abi=abi,
284                     bytecode=bytecode,
285                     Kp_real=Kp,
286                     Ki_real=Ki,
287                     Kd_real=Kd,
288                     dt_real=dt,
289                     umbral_real=umbral,
290                     minU_real=minU,
291                     maxU_real=maxU,
292                 )
293
294                 os.environ["CONTRACT_ADDRESS"] = address
295
```

```
296             st.markdown(
297                 f"""
298                 - Direccion de la cuenta: {acct_address}
299                 - Contrato desplegado en: {address}
300                 """
301             )
302 
303             #  3) Ejecuta la simulación on-chain
304             with st.spinner("Ejecutando simulación on-chain, por
    favor espera..."):
305                 metrics = evaluate(
306                     settings=settings,
307                     w3=w3,
308                     acct_address=acct_address,
309                     acct=acct,
310                     boiler=boiler,
311                     setpoint=setpoint,
312                     sim_time=sim_time,
313                     dt=dt,
314                     onchain=True,
315                     address=address,
316                 )
317                 m = metrics
318 
319                 if mode == "onchain (local)":
320                     st.markdown(
321                         f"""
322                     ###  Resultados {mode}
323 
324                     | Métrica                      | Valor
                      |
325 
    |------------------------------|--------------------------------|
326                     | Tiempo de establecimiento    | {m['settle_time
    ']:.2f} s          |
327                     | Sobrepaso                    | {m['overshoot
    ']:.2f} degC           |
328                     | IAE (Integral del error)     | {m['iae']:.2f}
                      |
329                     | Número de eventos            | {m['n_events']}
                      |
330                     | Gas total                    | {m['total_gas
```

```
    ']/1e3:.1f} kwei     |
331                     | Coste aproximado              | ${m['cost']:.2f}
     USD              |
332                     """
333                     )
334                 elif mode == "onchain (testnet)":
335                     st.markdown(
336                         f"""
337                     ###  Resultados {mode}
338 
339                     | Métrica                       | Valor
                      |
340 
    |------------------------------|---------------------------------|
341                     | Tiempo de establecimiento     | {m['settle_time
    ']:.2f} s          |
342                     | Sobrepaso                     | {m['overshoot
    ']:.2f} degC          |
343                     | IAE (Integral del error)      | {m['iae']:.2f}
                      |
344                     | Número de eventos             | {m['n_events']}
                      |
345                     | Gas total                     | {m['total_gas
    ']/1e3:.1f} kwei     |
346                     | Coste aproximado              | ${m['cost']:.2f}
     USD              |
347                     """
348                     )
349 
350         # Graficar resultados
351         fig1, ax1 = plt.subplots()
352         ax1.plot(metrics["times"], metrics["temps"])
353         ax1.axhline(setpoint, color="k", ls="--")
354         ax1.set_xlabel("Tiempo [s]")
355         ax1.set_ylabel("Temp [degC]")
356         ax1.set_title("Temperatura Caldera")
357 
358         fig2, ax2 = plt.subplots()
359         ax2.plot(metrics["times"], metrics["ctrls"])
360         ax2.set_xlabel("Tiempo [s]")
361         ax2.set_ylabel("u")
362         ax2.set_title("Señal de Control")
```

```
363
364         # Mostrar lado a lado
365         col1, col2 = st.columns(2)
366         with col1:
367             st.pyplot(fig1)
368         with col2:
369             st.pyplot(fig2)
370
371 with tabs[1]:
372     st.header("Optimización PID")
373
374     #  Creo 2 columnas para los 2 expanders
375     col1, col2 = st.columns(2)
376
377     with col1:
378         with st.expander("Parámetros generales", expanded=True):
379             # Dentro, lo divido en 2 columnas para los inputs
380             col1_in, col2_in = st.columns(2)
381             with col1_in:
382                 setpoint_opt = st.number_input(
383                     "Setpoint [degC]", value=50.0, key="opt_sp", step
    =1.0, format="%.1f"
384                 )
385                 sim_time_opt = st.number_input(
386                     "Tiempo simulación [s]",
387                     value=200.0,
388                     key="opt_st",
389                     step=25.0,
390                     format="%.0f",
391                     min_value=0.1,
392                 )
393             with col2_in:
394                 dt_opt = st.number_input(
395                     "Paso dt [s]", value=1.0, step=0.1, min_value
    =0.01, key="opt_dt"
396                 )
397                 umbral_opt = st.number_input(
398                     "Umbral [degC]",
399                     value=2.2,
400                     key="opt_th",
401                     step=0.1,
402                     format="%.2f",
```

```
403                     min_value=0.01,
404                 )
405 
406     with col2:
407         with st.expander("Rango de parámetros PID", expanded=True):
408             col1_pid, col2_pid, col3_pid = st.columns(3)
409             with col1_pid:
410                 Kp_inicio = st.number_input(
411                     "Kp inicio", value=0.0, key="kp0", step=1.0,
    format="%.1f"
412                 )
413                 Ki_inicio = st.number_input(
414                     "Ki inicio", value=0.0, key="ki0", step=0.1,
    format="%.2f"
415                 )
416                 Kd_inicio = st.number_input(
417                     "Kd inicio", value=0.0, key="kd0", step=0.01,
    format="%.3f"
418                 )
419             with col2_pid:
420                 Kp_end = st.number_input(
421                     "Kp fin", value=10.0, key="kp1", step=1.0, format
    ="%.1f"
422                 )
423                 Ki_end = st.number_input(
424                     "Ki fin", value=2.0, key="ki1", step=0.1, format=
    "%.2f"
425                 )
426                 Kd_end = st.number_input(
427                     "Kd fin", value=0.5, key="kd1", step=0.01, format
    ="%.3f"
428                 )
429             with col3_pid:
430                 Kp_n = st.number_input("Kp pasos", value=15, key="
    opt_n0")
431                 Ki_n = st.number_input("Ki pasos", value=8, key="
    opt_n1")
432                 Kd_n = st.number_input("Kd pasos", value=8, key="
    opt_n2")
433 
434     if st.button("Ejecutar Optimización PID"):
435         settings = Settings()
```

```
436
437         boiler = create_boiler(
438             order=order,
439             **(
440                 {
441                     "C": C,
442                     "k_conv": k_conv,
443                     "ambient": ambient,
444                     "area": area,
445                     "emisividad": emisividad,
446                     "seed": 27,
447                     "max_heater_power": max_heater_power,
448                 }
449                 if order == 1
450                 else {
451                     "C1": C1,
452                     "C2": C2,
453                     "k12": k12,
454                     "k_conv": k_conv,
455                     "ambient": ambient,
456                     "area": area,
457                     "emisividad": emisividad,
458                     "seed": 27,
459                     "max_heater_power": max_heater_power,
460                 }
461             ),
462         )
463         Kp_list = np.linspace(Kp_inicio, Kp_end, int(Kp_n))
464         Ki_list = np.linspace(Ki_inicio, Ki_end, int(Ki_n))
465         Kd_list = np.linspace(Kd_inicio, Kd_end, int(Kd_n))
466         best = grid_search_pid(
467             settings,
468             boiler,
469             Kp_list,
470             Ki_list,
471             Kd_list,
472             umbral_opt,
473             setpoint_opt,
474             sim_time_opt,
475             dt_opt,
476         )
477
```

```
478         # Extraigo parámetros y métricas
479         kp, ki, kd = best["params"]
480         m = best["metrics"]
481 
482         # Markdown bonito
483         st.markdown(
484             f"""
485         ###  Mejor combinación de PID
486         - **Kp:** {kp:.1f}
487         - **Ki:** {ki:.2f}
488         - **Kd:** {kd:.3f}
489 
490         ####  Métricas de desempeño
491         | Métrica                      | Valor                |
492         |------------------------------|---------------------|
493         | Tiempo de establecimiento    | {m['settle_time']:.3f} s |
494         | Sobrepaso                    | {m['overshoot']:.2f} degC |
495         | IAE (Integral del error)     | {m['iae']:.1f}       |
496         | Número de eventos            | {m['n_events']}      |
497         """
498         )
499 
500 with tabs[2]:
501     st.header("Barrido de Umbral")
502 
503     #  Dos columnas para los dos expanders principales
504     col1, col2, col3 = st.columns([0.3, 0.4, 0.3])
505 
506     with col1:
507         with st.expander("Parámetros generales", expanded=True):
508             # Dentro, organizo en dos columnas
509             setpoint_thr = st.number_input(
510                 "Setpoint [degC]", value=50.0, key="thr_sp", step
    =1.0, format="%.1f"
511             )
512             sim_time_thr = st.number_input(
513                 "Tiempo simulación [s]",
514                 value=200.0,
515                 key="thr_st",
516                 step=25.0,
517                 format="%.0f",
518             )
```

```
519             dt_thr = st.number_input(
520                 "Paso dt [s]", value=1.0, key="thr_dt", step=0.1,
    format="%.2f"
521             )
522 
523     with col2:
524         with st.expander("Parámetros PID", expanded=True):
525             col1_in, col2_in = st.columns(2)
526             with col1_in:
527                 Kp_thr = st.number_input(
528                     "Kp", value=10.0, key="thr_kp", step=1.0, format=
    "%.1f"
529                 )
530                 Ki_thr = st.number_input(
531                     "Ki", value=0.29, key="thr_ki", step=0.1, format=
    "%.2f"
532                 )
533                 Kd_thr = st.number_input(
534                     "Kd", value=0.071, key="thr_kd", step=0.01,
    format="%.3f"
535                 )
536             with col2_in:
537                 minU_thr = st.number_input(
538                     "u mínimo", value=0.0, key="thr_minU", step=1.0,
    format="%.1f"
539                 )
540                 maxU_thr = st.number_input(
541                     "u máximo", value=100.0, key="thr_maxU", step
    =1.0, format="%.1f"
542                 )
543 
544         with col3:
545             #  Expander para definir rango de umbral
546             with st.expander("Rango de umbral", expanded=True):
547                 thr_inicio = st.number_input(
548                     "Umbral inicio", value=0.0, key="thr0", step=0.1,
     format="%.2f"
549                 )
550                 thr_end = st.number_input(
551                     "Umbral fin", value=4.0, key="thr1", step=0.1,
    format="%.2f"
552                 )
```

```
553                 thr_n = st.number_input(
554                     "Pasos", value=10, key="thr_n", step=1, format="%
    d", min_value=1
555                 )
556
557     if st.button("Ejecutar Barrido de Umbral"):
558         settings = Settings()
559
560         # Inicializar la caldera con sus parámetros
561         boiler = create_boiler(
562             order=order,
563             **(
564                 {
565                     "C": C,
566                     "k_conv": k_conv,
567                     "ambient": ambient,
568                     "area": area,
569                     "emisividad": emisividad,
570                     "seed": 27,
571                     "max_heater_power": max_heater_power,
572                 }
573                 if order == 1
574                 else {
575                     "C1": C1,
576                     "C2": C2,
577                     "k12": k12,
578                     "k_conv": k_conv,
579                     "ambient": ambient,
580                     "area": area,
581                     "emisividad": emisividad,
582                     "seed": 27,
583                     "max_heater_power": max_heater_power,
584                 }
585             ),
586         )
587         # Definir lista de umbrales
588         thr_list = np.linspace(thr_inicio, thr_end, int(thr_n))
589         # Ejecutar barrido de umbral
590         thr_list, events, raw_puntuacions, combined = umbral_sweep(
591             settings,
592             boiler,
593             thr_list,
```

```
594             (Kp_thr, Ki_thr, Kd_thr, minU_thr, maxU_thr),
595             setpoint_thr,
596             sim_time_thr,
597             dt_thr,
598         )
599 
600         # Índice óptimo según combined (ya normalizado internamente)
601         best_idx = int(np.argmin(combined))
602         best_thr = thr_list[best_idx]
603         best_events = events[best_idx]
604         best_puntuacion = raw_puntuacions[best_idx]
605         best_comb = combined[best_idx]
606 
607         # Resumen en Markdown
608         st.markdown(
609             f"""
610         ###  Resultados del Barrido de Umbral (Trade-off)
611         - **Umbral óptimo:** {best_thr:.2f} degC
612         - **Número de eventos:** {int(best_events)}
613         - **Puntuacion bruto (Establecimiento + Sobrepaso + IAE):** {
    best_puntuacion:.2f}
614         - **Índice combinado (puntuacion bruto + N eventos):** {
    best_comb:.3f}
615         """
616         )
617 
618         # Gráficas lado a lado
619         col1, col2 = st.columns(2)
620         with col1:
621             fig1, ax1 = plt.subplots()
622             ax1.plot(thr_list, events, marker="o")
623             ax1.set_xlabel("Umbral [degC]")
624             ax1.set_ylabel("Número de eventos")
625             ax1.set_title("Eventos vs Umbral")
626             st.pyplot(fig1)
627 
628         with col2:
629             fig2, ax2 = plt.subplots()
630             ax2.plot(thr_list, raw_puntuacions, marker="o")
631             ax2.set_xlabel("Umbral [degC]")
632             ax2.set_ylabel("Puntuacion bruto (Establecimiento +
    Sobrepaso + IAE)")
```

```
633                 ax2.set_title("Puntuacion vs Umbral")
634                 st.pyplot(fig2)
```