



Universidad Autónoma de Ciudad Juárez

Instituto de Ingeniería y Tecnología

Departamento de Eléctrica y Computación

Maestría en Cómputo Aplicado

“Sistema de Detección y Ejecución Automatizada de Arbitraje Financiero mediante Préstamos *Flash* y Protección contra *MEV* en la *Blockchain* de *Ethereum*”

Reporte Técnico para obtener el grado de

Maestro en Cómputo Aplicado

Lic. César Cárdenas Sánchez

“Becado por el Consejo Nacional de Ciencia y Tecnología”

Bajo la Dirección de: Dr. Victor Manuel Morales Rocha

Ciudad Juárez, Chihuahua, marzo de 2026

Agradecimientos

A mi madre, cuyo apoyo constante —emocional y práctico— hizo posible que este trabajo llegara a término. Su ejemplo de disciplina y compromiso ha sido mi guía.

A mi novia, por comprender mis ausencias, celebrar los avances y recordarme el propósito cuando las fuerzas faltaban. Este logro también es tuyo.

A mis docentes por la enseñanza recibida dentro y fuera del aula. Su exigencia, retroalimentación y disposición para orientar me ayudaron a convertir ideas en resultados. Extiendo este agradecimiento a mi tutor de tesis por sus observaciones puntuales y su acompañamiento académico.

A mis compañeros y colegas que compartieron dudas, lecturas y pruebas; las conversaciones técnicas y el trabajo en equipo fortalecieron varios apartados de la investigación.

Financiamiento. Este trabajo fue posible gracias al apoyo de una beca del Consejo Nacional de Humanidades, Ciencias y Tecnologías (CONAHCYT), cuyo respaldo permitió dedicar tiempo y esfuerzo a la investigación y a la redacción de este documento. Las opiniones y conclusiones expresadas son responsabilidad exclusiva del autor.

Finalmente, agradezco a todas las personas que, de forma directa o indirecta, contribuyeron a que esta tesis se concretara: desde una referencia oportuna hasta una palabra de aliento, cada gesto dejó huella en el resultado final.

Dedicatoria

A mi madre, por su amor, su fortaleza y por enseñarme a mantener la calma y el rumbo cuando nada era sencillo.

A mi pareja, por su compañía, comprensión y confianza en mí incluso en los días difíciles.

ÍNDICE GENERAL

CAPÍTULO 1: PLANTEAMIENTO DEL PROBLEMA	15
1.1 Contexto General	15
1.2 Antecedentes (Dominio de Aplicación)	15
1.3 Identificación del Problema o Necesidad.....	16
1.4 OBJETIVOS	17
1.4.1 Objetivo General	17
1.4.2 Objetivos Específicos	17
1.5 JUSTIFICACIÓN	18
1.6 METODOLOGÍA	19
1.6.1 Tipo y enfoque de la investigación.....	19
1.6.2 Modelo de procesos.....	19
1.6.3 Estrategia de validación	20
1.6.4 Evaluación de madurez tecnológica	21
1.6.5 Reproducibilidad	21
CAPÍTULO 2: MARCO TEÓRICO	22
2.1 Fundamentos de Criptomonedas y <i>blockchain</i>	22
2.1.1 Conceptos básicos de las criptomonedas y <i>blockchain</i>	22
2.1.2 Tecnologías y protocolos blockchain	23
2.1.3 Ethereum Virtual Machine (EVM).....	26
2.1.4 Contratos Inteligentes (Smart Contracts): Fundamentos y Principios de Diseño	28
2.1.5 Protocolos de Interoperabilidad y Estandarización de Activos en <i>Blockchain</i>	29
2.1.6 <i>Gas</i> y costos de ejecución	30
2.1.7 Implementación del EIP-1559.....	30

2.1.8 Interoperabilidad <i>Multi-Chain</i> y Protocolos <i>Cross-Chain</i>	31
2.2 Introducción a Finanzas Descentralizadas (<i>DeFi</i>).....	33
2.2.1 Diferencias y ventajas sobre las Finanzas Centralizadas.	34
2.2.2 Exchanges Descentralizados (DEXes) vs Exchanges Centralizados (CEXes)	35
2.2.3 Mecánica de los DEXes	35
2.2.4 Protocolos de Intercambio Descentralizado y Automated Market Makers (AMMs).....	36
2.2.5 Impacto de Precio (Price Impact) y Dinámica Asintótica del Precio	37
2.2.6 Deslizamiento (Slippage) como Consecuencia Matemática	37
2.2.7 Liquidez Concentrada como Evolución del Modelo CPMM	38
2.3 Teoría del Arbitraje Financiero	40
2.3.1 Hipótesis del Mercado Eficiente	40
2.3.2 Tipos de Arbitraje en el Mercado de Criptomonedas.....	40
2.4 Automatización y Sistemas Algorítmicos <i>en DeFi</i>	43
2.4.1 Trading algorítmico de alta frecuencia en mercados descentralizados	43
2.4.2 Arquitectura general de los <i>bots</i> de arbitraje.....	44
2.4.3 Teoría y Diseño de Algoritmos para la Detección de Oportunidades	45
2.5 Introducción a los préstamos relámpago (<i>Flash Loans</i>)	46
2.5.1 Atomicidad y eliminación del riesgo de incumplimiento.....	47
2.5.2 Funcionamiento de los protocolos de préstamo	47
2.5.3 Aave V3 como protocolo de préstamo	48
2.5.4 <i>Flash loans</i> como herramienta de arbitraje	49
2.6 Riesgos en las operaciones de arbitraje con <i>Flash Loans</i>	50
2.6.1 Riesgo de ejecución y variación de precio	50
2.6.2 Riesgo de liquidez insuficiente.....	52

2.6.3 Riesgo de fallos en contratos inteligentes y dependencias externas.....	52
2.6.4 Vulnerabilidades comunes: Ataques de reentrada (<i>reentrancy</i>).....	53
2.6.5 MEV, front-running y competencia estratégica	54
2.6.6 <i>Flashbots</i> y rutas privadas de transacción	55
2.6.7 Riesgos de Infraestructura Multi-Cadena y Vulnerabilidad en Puentes.....	56
2.7 Oráculos de precios	57
2.7.1 Tipos de oráculos: centralizados, federados y descentralizados	57
2.7.2 Manipulación de oráculos, préstamos <i>flash</i> y arbitraje	59
2.7.3 El oráculo como mecanismo de referencia para el cálculo de rentabilidad.....	60
CAPÍTULO 3: DESARROLLO DEL PROYECTO	62
3.1 Introducción	62
3.2 Entorno de Desarrollo y Estructura del Proyecto	63
3.2.1 Lenguajes Utilizados.....	63
3.2.2 Fundamentos del Framework Hardhat	64
3.2.3 Plataformas, Herramientas y <i>Frameworks</i>	65
3.2.4 Estructura del Proyecto	66
3.2.5 Interfaces Utilizadas	71
3.2.6 Librerías utilizadas	74
3.3 Arquitectura <i>On-Chain</i> : Contratos Inteligentes	75
3.3.1 Contratos Inteligentes y Arquitectura de Smart Contracts	76
3.3.2 Funciones Clave del Contrato Principal.....	80
3.3.3 Flujo de Ejecución Atómica.....	82
3.3.4 Roles, Permisos y Seguridad.....	83
3.3.5 Modularidad, Escalabilidad y Extensión.....	85

3.3.6 Optimización de Comisiones por Transacción.....	87
3.3.7 Gestión y Evaluación de Liquidez en Pools.....	88
3.3.8 Control del <i>Slippage</i> en Ejecución.....	90
3.3.9 Algoritmo de Detección y Evaluación de Oportunidades.....	90
3.3.10 Eficiencia Computacional.....	92
3.4 Arquitectura del sistema Off Chain: Automatización y Scripts.....	93
3.4.1 Arquitectura de Scripts <i>Off-Chain</i> : El Cerebro Operativo.....	93
3.4.2 Script de Despliegue Automatizado.....	97
3.4.3 Recolección de Datos de Liquidez.....	100
3.4.4 Gestión de Tarifas de Gas.....	103
3.4.5 Utilidad de Seguridad.....	105
3.4.6 Configuración Centralizada.....	107
3.5 Flujo de Trabajo Operativo y Ciclo de Vida del Sistema.....	109
3.5.1 Configuración Inicial del Entorno.....	109
3.5.2 Recopilación de Datos.....	110
3.5.3 Monitoreo y Ejecución del Arbitraje.....	111
3.6 Seguridad y Mitigación de Amenazas.....	112
3.6.1 Principales riesgos en Contratos Inteligentes <i>DeFi</i>	112
3.6.2 Prevención de Ataques de Reentrada.....	113
3.6.3 Control de Accesos y Permisos.....	114
3.6.4 Simulación de Ataques Comunes.....	116
3.6.5 Ataques de Reordenamiento de Transacciones (<i>Front-running</i> y <i>MEV</i>).....	117
3.6.6 Protección contra Frontrunning y <i>Bots MEV</i>	117
3.6.7 Estrategia de Seguridad Operacional y Validación ante Default.....	118

3.7 Pruebas, simulación e integración.....	118
3.7.1 Pruebas Unitarias.....	118
3.7.2 Simulación del Contrato en Red Local y con Forks.....	119
3.7.3 Simulaciones previas a la Ejecución	121
3.7.4 Pruebas de Integración y Flujo Simulado.....	121
3.7.5 Despliegue y Verificación en Entornos de Desarrollo y Producción	123
3.7.6 Herramientas para simulación y Pruebas	124
3.7.7 Manejo de Logs y Monitoreo	124
CAPÍTULO 4: RESULTADOS	126
4.1 Resultado 1: Ejecución Exitosa.....	126
4.2 Resultado 2: Oportunidad Abortada por Cambio de Precio	126
4.3 Resultado 3: Error de Transacción Inesperado.....	127
4.4 Observaciones Finales.....	128
CAPÍTULO 5: CONCLUSIONES	130
5.1 Cumplimiento de Objetivos	130
5.2 Decisiones técnicas y de diseño	132
5.3 Limitaciones identificadas.....	132
5.4 Proyección y trabajos futuros	133
CAPÍTULO 6: GLOSARIO	134
CAPÍTULO 7: REFERENCIAS.....	142

ÍNDICE DE RÓTULOS

FIGURAS

Figura 2.1: Componentes de la EVM.....	27
Figura 3.1: Arquitectura general del sistema de arbitraje.	70

CÓDIGOS

Código 3.1 Importaciones principales del archivo de configuración <code>hardhat.config.ts</code> escrito en lenguaje typescript.....	65
Código 3.2: Estructura de enumeración <code>DexType</code> en lenguaje Solidity para la clasificación e indexación de interfaces de protocolos de intercambio descentralizado (<i>DEXes</i>) implementada en el contrato inteligente <code>DexAggregator.sol</code>	72
Código 3.3: Declaración principal de herencia, dependencias de seguridad de OpenZeppelin e interfaces de Aave implementada en el contrato inteligente <code>FlashLoanArbitrage.sol</code>	78
Código 3.4: Estructura general, variables de estado y firma de la función <code>executeSwap()</code> implementadas en el contrato inteligente <code>DexAggregator.sol</code> con lenguaje solidity.....	79
Código 3.5: Función <code>executeFlashLoanSimple()</code> en lenguaje Solidity para solicitar préstamo flash simple de Aave V3 implementada en el contrato inteligente <code>FlashLoanArbitrage.sol</code> ..	80
Código 3.6: Función de <i>callback</i> <code>executeOperation()</code> en lenguaje Solidity exigida por Aave para validar la recepción de fondos y ejecutar la lógica de arbitraje implementada en el contrato inteligente <code>FlashLoanArbitrage.sol</code>	81
Código 3.7: Función <code>_executeArbitrage()</code> en lenguaje Solidity para evaluar cotizaciones en tokens intermediarios y ejecutar los intercambios atómicos implementada en el contrato inteligente <code>FlashLoanArbitrage.sol</code>	82

Código 3.8: Modificadores de seguridad <code>onlyOwner()</code> y <code>nonReentrant()</code> de OpenZeppelin para control de acceso y prevención de ataques de reentrada implementados con lenguaje Solidity en los contratos inteligentes.	84
Código 3.9: Variable <code>_flashLoanReentrancyGuard</code> para crear un mecanismo de bloqueo específico para operaciones flash implementada en lenguaje solidity en el contrato inteligente <code>FlashLoanArbitrage.sol</code>	84
Código 3.10: Funciones administrativas de emergencia <code>setTestMode()</code> , <code>rescueTokens()</code> y <code>rescueETH()</code> para recuperación de fondos implementadas con lenguaje Solidity en el contrato inteligente <code>FlashLoanArbitrage.sol</code>	85
Código 3.11: Configuración del arreglo <code>intermediaryTokens</code> para definir los 39 activos de alta liquidez como puentes de enrutamiento implementada con lenguaje solidity en el contrato inteligente <code>FlashLoanArbitrage.sol</code>	86
Código 3.12: Consulta del precio de <i>gas</i> actual mediante el proveedor <i>RPC</i> escrito en lenguaje typescript en el script <code>gasFee.ts</code>	87
Código 3.13: Estructura del archivo <code>gasFee.json</code> con datos de tarifas de gas actuales e históricas.	87
Código 3.14: Construcción de parámetros de transacción compatibles con EIP-1559 mediante <code>getFeeData()</code> escrito en lenguaje typescript.....	88
Código 3.15: Consulta de reservas de un pool de liquidez mediante la función <code>getReserves()</code> escrito en lenguaje typescript.	89
Código 3.16: Filtrado de pools con liquidez inferior a \$100,000 USD en el script <code>arbitrage.ts</code>	89
Código 3.17: Validación de disponibilidad de un token para préstamos relámpago en las reservas de Aave escrito en lenguaje typescript.....	90

Código 3.18: Validación <i>on-chain</i> de <i>slippage</i> máximo permitido en la ejecución de intercambios escrito en lenguaje solidity.....	90
Código 3.19: Cálculo off-chain del monto mínimo aceptable con tolerancia de slippage del 0.5% escrito con typescript.....	90
Código 3.20: Mapeo de nombres de <i>DEXes</i> a índices del contrato y secuencia de inicialización en <code>arbitrage.ts</code> escrito con lenguaje typescript.....	94
Código 3.21: Cálculo de rentabilidad neta descontando costos de gas y prima del préstamo en <code>arbitrage.ts</code>	95
Código 3.22: Estrategia de envío dual: transacción directa con <i>gas</i> agresivo y bundle de <i>Flashbots</i> escrito en lenguaje typescript.....	96
Código 3.23: Función principal del script de despliegue <code>deploy.ts</code> con obtención del <i>deployer</i>	97
Código 3.24: Validación de fondos y configuración dinámica de <i>gas</i> previo al despliegue de contratos escrito con lenguaje typescript.....	98
Código 3.25: Despliegue secuencial de <code>DexAggregator.sol</code> y <code>FlashLoanArbitrage.sol</code> mediante Hardhat.....	98
Código 3.26: Transferencia de propiedad del <code>DexAggregator.sol</code> al contrato <code>FlashLoanArbitrage.sol</code>	99
Código 3.27: Verificación de permisos e inicialización de los 27 <i>DEXes</i> mediante <code>setupAllDexes()</code>	99
Código 3.28: Actualización automática de <code>addresses.ts</code> con las direcciones de los contratos desplegados.....	100
Código 3.29: Lista priorizada de <i>DEXes</i> seleccionados por liquidez histórica en <code>liquidity.ts</code>	100
Código 3.30: Filtrado de <i>DEXes</i> disponibles en la red antes de la consulta de datos.....	101

Código 3.31: Función <code>fetchTopPools()</code> para obtener <i>pools</i> de liquidez desde la <i>API</i> de GeckoTerminal.....	101
Código 3.32: Ordenamiento de pools por liquidez total y persistencia de metadatos completos escrito con typescript.....	102
Código 3.33: Generación de resumen de liquidez total por <i>DEX</i> y escritura del archivo <code>dexespools.json</code>	103
Código 3.34: Función <code>getGasFeeData()</code> para consultar y <i>formatear</i> datos de <i>gas</i> en tiempo real.	104
Código 3.35: Estructura del archivo <code>gasFee.json</code> con lectura actual e historial de tarifas de <i>gas</i>	105
Código 3.36: Generación de wallet dedicada para firma de <i>bundles</i> de <i>Flashbots</i> y actualización del archivo <code>.env</code>	107
Código 3.37: Extracto del archivo <code>addresses.ts</code> con las direcciones centralizadas de <i>DEXes</i> , contratos desplegados, Aave V3 y oráculos de Chainlink.	108
Código 3.38: Aplicación del modificador <code>nonReentrant</code> de OpenZeppelin a la función <code>executeOperation()</code>	113
Código 3.39: Implementación del <i>guard</i> personalizado <code>_flashLoanReentrancyGuard</code> dentro de <code>executeOperation()</code>	114
Código 40: Modificador <code>onlyOwner()</code> para restringir funciones administrativas al propietario del contrato.....	114
Código 3.41: Validaciones de origen en <code>executeOperation()</code> : verificación del <i>pool</i> de Aave y del iniciador de la transacción.	115
Código 3.42: Función <code>getEthPriceFromChainlink()</code> para obtener el precio de ETH/USD desde el oráculo de Chainlink.	116

Código 3.43: Validación de slippage en el contrato <code>DexAggregator.sol</code> durante la ejecución de cada intercambio.	116
Código 3.44: Verificación de rentabilidad mínima y repago del préstamo al pool de Aave.....	118
Código 3.45: Prueba unitaria con Hardhat para verificar la emisión del evento <code>FlashLoanExecuted</code> tras un préstamo relámpago.....	119
Código 3.46: Configuración de <code>hardhat.config.ts</code> para bifurcación (<i>fork</i>) de Ethereum Mainnet en pruebas locales.	120
Código 3.47: Simulación pre-ejecución mediante <code>callStatic</code> para verificar la viabilidad de la transacción sin consumir <i>gas</i>	121
Código 3.48: Ejemplo de salida del sistema de logging mostrando una oportunidad de arbitraje con costos y beneficio neto.	124

CAPÍTULO 1: PLANTEAMIENTO DEL PROBLEMA

1.1 Contexto General

El mercado de criptomonedas ha experimentado un crecimiento exponencial desde su creación, atrayendo tanto a inversores individuales como a instituciones financieras. Caracterizado por su alta volatilidad y descentralización, este mercado ofrece oportunidades únicas de inversión y especulación. De acuerdo con [1], la emergencia de las plataformas de finanzas descentralizadas (*DeFi*) ha añadido una capa adicional de complejidad y oportunidad, destacando la necesidad de nuevas estrategias de inversión y herramientas tecnológicas.

En este contexto, el arbitraje de criptomonedas ha surgido como una técnica popular para capitalizar las diferencias de precio entre distintos mercados o plataformas. Sin embargo, la naturaleza dinámica y a menudo no regulada de estas plataformas presenta tanto oportunidades como desafíos únicos para los inversores.

1.2 Antecedentes (Dominio de Aplicación)

Desde el punto de vista tecnológico, el arbitraje en el mercado de criptomonedas se ha desarrollado desde métodos manuales hasta sistemas automatizados que utilizan *bots* y algoritmos sofisticados. Según [2] estos sistemas son capaces de identificar y actuar sobre diferencias de precio en fracciones de segundo, una capacidad crítica dada la rapidez con que cambian los precios en este mercado.

Sin embargo, tal como se menciona en [3], el arbitraje en criptomonedas también enfrenta desafíos significativos. Los costos de transacción y la velocidad de ejecución varían entre plataformas, lo que puede afectar la rentabilidad de las operaciones de arbitraje. Además, las restricciones locales en la transferencia de monedas fiduciarias y las diferencias en la regulación pueden complicar el acceso a ciertos mercados.

Otros problemas importantes señalados en [4] son la alta volatilidad de los precios de las criptomonedas, que puede conducir a rápidos cambios en las oportunidades de arbitraje, así como la falta de liquidez en ciertos *exchanges*, que además pueden dar lugar a disparidades de precios, por lo que aprovechar estas oportunidades puede ser difícil debido a los retrasos en la ejecución de las órdenes.

El enfoque tradicional de arbitraje, que implica comprar en un mercado y vender en otro, se ve desafiado por la necesidad de gestionar fondos y criptomonedas en múltiples plataformas simultáneamente. Esta complejidad logística y financiera ha impulsado el desarrollo de soluciones

como los *flash loans*, que según [5], permiten operaciones de arbitraje sin la necesidad de capital inicial significativo. Aunque estos ofrecen ventajas, también conllevan riesgos inherentes y requieren un entendimiento profundo de los mecanismos de préstamo y devolución en el mismo bloque de transacción, de acuerdo con lo expuesto por [6].

En el ámbito de las finanzas descentralizadas (*DeFi*), varios estudios han explorado la automatización del arbitraje mediante mecanismos avanzados como los *flash loans*. Por ejemplo, en [7] se documenta cómo estos préstamos instantáneos permiten ejecutar estrategias de arbitraje sin la necesidad de capital inicial, aprovechando la atomicidad de las transacciones en *blockchain*. De manera complementaria, [8] analiza el uso de *flash loans* en diversos protocolos *DeFi*, destacando sus ventajas operativas, pero también señalando limitaciones asociadas a la liquidez, la congestión de la red y el riesgo de fallas en contratos inteligentes.

A pesar de estos avances, investigaciones como las [9] muestran que el entorno *DeFi* continúa enfrentando importantes desafíos estructurales, incluyendo la competencia por la prioridad en la ejecución de transacciones (*MEV*), altos costos de gas y variaciones en la eficiencia entre *exchanges* descentralizados. Estas limitaciones afectan directamente la viabilidad del arbitraje en escenarios de alta volatilidad o congestión. Asimismo, [10] evidencia que los *bots* de arbitraje basados en *flash loans* presentan restricciones de escalabilidad y dependencia de la infraestructura de cada red, lo que dificulta su uso en entornos *multi-chain*.

En conjunto, la literatura indica que, aunque las herramientas actuales permiten arbitraje automatizado de manera innovadora, aún existen oportunidades para mejorar la eficiencia, reducir costos operativos y aumentar la adaptabilidad del sistema. Estas limitaciones motivan el desarrollo del presente proyecto

1.3 Identificación del Problema o Necesidad

A pesar del desarrollo continuo de estrategias de arbitraje en mercados de criptomonedas, los sistemas actuales enfrentan múltiples desafíos estructurales y operativos que limitan su accesibilidad, eficiencia y escalabilidad. Tal como lo plantea [9], la ejecución de arbitraje en plataformas *DeFi* está dominada por *bots* de alta frecuencia que dependen de complejos mecanismos como las *Priority Gas Auctions (PGAs)*, lo que genera una competencia intensiva por el orden de ejecución y eleva significativamente los costos transaccionales. Esta dinámica no solo reduce la rentabilidad potencial, sino que también introduce riesgos para la estabilidad del consenso en la *blockchain*.

Adicionalmente, [7] demuestra que aunque los *flash loans* han revolucionado el acceso al capital instantáneo para ejecutar arbitraje sin garantía inicial, su implementación actual ha sido

mayoritariamente utilizada en ataques y manipulaciones de precios, evidenciando una carencia de estructuras de seguridad sólidas. Por otro lado, la mayoría de los sistemas existentes están diseñados para operar en una única cadena de bloques, careciendo de interoperabilidad entre distintas plataformas, lo que limita su aplicabilidad en un ecosistema *DeFi* cada vez más fragmentado y *multi-chain*.

Ante este contexto, se identifica la necesidad concreta del desarrollo de un sistema de arbitraje automatizado que sea seguro, escalable, interoperable y optimizado para operar en múltiples plataformas *DeFi*. Este sistema debe minimizar los costos transaccionales, mitigar los riesgos asociados con la ejecución automatizada y ser accesible para usuarios e investigadores que no cuenten con infraestructura técnica avanzada.

1.4 OBJETIVOS

1.4.1 Objetivo General

Desarrollar un sistema integral de arbitraje de criptomonedas que utilice algoritmos de contratos inteligentes en protocolos de *blockchain* con *flash loans* para optimizar las ganancias, manteniendo un perfil de riesgo bajo.

1.4.2 Objetivos Específicos

- Diseñar un algoritmo eficiente para la detección en tiempo real de diferencias de precios entre múltiples ecosistemas *blockchain*, así como en diferentes plataformas descentralizadas de intercambio de criptomonedas, permitiendo la identificación rápida de oportunidades de arbitraje.
- Implementar un sistema de evaluación y selección que determine cada oportunidad de arbitraje, basándose en criterios de costo, riesgo y tiempo de ejecución.
- Desarrollar un algoritmo de ejecución automatizado que, basado en la identificación en tiempo real de oportunidades de arbitraje y la selección del proveedor de *flash loan* óptimo, ejecute operaciones de arbitraje de manera eficiente y segura en el mercado de criptomonedas.
- Desarrollar un módulo de simulación y prueba que valide la efectividad y rentabilidad de las estrategias de arbitraje propuestas, incluyendo análisis de escenarios adversos para evaluar la resiliencia del sistema.

1.5 JUSTIFICACIÓN

El desarrollo de un sistema automatizado de arbitraje en plataformas *DeFi* representa una contribución significativa tanto al ámbito tecnológico como al financiero, en especial en contextos emergentes como el de México. Como se mencionó anteriormente, los modelos actuales de arbitraje sufren de problemas estructurales como altos costos de transacción, vulnerabilidades de seguridad y una limitada interoperabilidad entre *blockchains*. Superar estas limitaciones no solo optimiza la eficiencia operativa del ecosistema *DeFi*, sino que también abre la puerta a su adopción institucional. En este contexto, una de las aplicaciones más prometedoras de esta tecnología recae en el uso institucional por parte de las Administradoras de Fondos para el Retiro (AFORES). Las AFORES, responsables de gestionar los fondos de pensión de millones de mexicanos, enfrentan desafíos importantes para garantizar rendimientos competitivos en un entorno de baja rentabilidad en instrumentos tradicionales. Según datos de la CONSAR (2025) documentados en [11], aproximadamente el 50.6% de los activos del Sistema de Ahorro para el Retiro (SAR) se encontraba concentrado en instrumentos de deuda gubernamental, mientras que la volatilidad macroeconómica derivada de la incertidumbre arancelaria internacional y la depreciación cambiaria generó minusvalías por más de \$31,674 MDP en el primer trimestre de 2025. Esta combinación de baja diversificación y presión sobre los rendimientos netos limita la capacidad del sistema pensionario para incrementar el patrimonio de los afiliados. El presente proyecto propone un sistema automatizado de arbitraje en finanzas descentralizadas (*DeFi*) con el fin de superar las limitaciones técnicas actuales, como altos costos de transacción, liquidez variable, congestión de red y riesgos asociados a ejecución, documentadas en la literatura especializada. Gracias al uso de mecanismos como *flash loans* y arquitecturas modulares, el sistema busca optimizar eficiencia, reducir costos operativos y mejorar la seguridad.

Este enfoque no solo tiene relevancia técnica, sino también potencial social y económico: mediante su adopción por AFORES, podría ofrecer una vía innovadora de diversificación de inversiones para los fondos de pensiones, con el propósito de generar rendimientos más competitivos que incrementen los recursos disponibles para jubilación en México. En consecuencia, el proyecto representa una contribución tanto al desarrollo de infraestructura *DeFi* como a la mejora del bienestar de los pensionados.

1.6 METODOLOGÍA

1.6.1 Tipo y enfoque de la investigación

El presente trabajo corresponde a una investigación aplicada de tipo tecnológico, con enfoque constructivo, conducida bajo el paradigma de la Ciencia del Diseño (*Design Science Research, DSR*). Este paradigma es pertinente porque el resultado central de la investigación no es una teoría, sino un artefacto funcional: un sistema integrado por contratos inteligentes y *scripts* de automatización *off-chain* orientado a la detección y ejecución de arbitraje financiero mediante préstamos *flash*. Conforme a [12], en la Ciencia del Diseño el conocimiento sobre un problema y su solución se adquiere a través de la construcción y aplicación del artefacto diseñado, en un ciclo iterativo de diseño, prueba y refinamiento. Este ciclo describe con fidelidad el proceso seguido en el proyecto, en el que cada versión del sistema fue sometida a prueba, sus fallas fueron analizadas y el diseño fue corregido antes de una nueva evaluación.

1.6.2 Modelo de procesos

La conducción del proyecto se estructuró siguiendo las seis actividades del modelo de proceso para investigación en Ciencia del Diseño propuesto por [13], cuyas etapas se corresponden con la estructura de este documento:

1. Identificación del problema y motivación. Se caracteriza la fragmentación de la liquidez en el ecosistema *DeFi* y los obstáculos que enfrenta el arbitraje automatizado, a saber, la necesidad de capital inicial, la volatilidad del costo de gas y la exposición al *MEV* (Secciones 1.1 a 1.3).
2. Definición de los objetivos de la solución. Se establecen el objetivo general y los objetivos específicos que el artefacto debe satisfacer (Sección 1.4).
3. Diseño y desarrollo. Se construyen los contratos `FlashLoanArbitrage.sol` y `DexAggregator.sol`, así como los *scripts* de monitoreo, evaluación de rentabilidad y ejecución (Capítulo 3). Es en esta etapa donde opera el ciclo iterativo de diseño y corrección característico de la Ciencia del Diseño.
4. Demostración. Se ejecuta el sistema sobre el entorno de validación descrito en la Sección 1.6.3, documentando su comportamiento ante oportunidades de arbitraje reales (Capítulo 4).

5. Evaluación. Se valoran la rentabilidad neta, la capacidad de detección y la respuesta del sistema ante condiciones adversas, y se contrastan los resultados obtenidos frente a los objetivos planteados (Capítulo 4 y Secciones 5.1 y 5.3).

6. Comunicación. Los resultados, las decisiones de diseño y las limitaciones identificadas se documentan en el Capítulo 5 y en la defensa correspondiente.

1.6.3 Estrategia de validación

La validación del artefacto se organizó en tres niveles de rigor creciente, descritos en detalle en la Sección 3.7:

- a) Pruebas unitarias: Se verificó de forma aislada la lógica de las funciones críticas de ambos contratos, incluyendo control de acceso, validación de parámetros y condiciones de éxito y fallo, así como casos extremos orientados a comprobar que las validaciones abortaran la ejecución de manera segura (Sección 3.7.1).
- b) Pruebas de integración sobre bifurcación de *Ethereum Mainnet*: Se empleó la funcionalidad de *fork* de Hardhat, apuntando a la red principal a través de un nodo de Alchemy, para replicar localmente el estado real de la red, incluyendo balances, *pools* de liquidez y contratos desplegados de protocolos como Aave V3, Uniswap y Chainlink. Este nivel permitió someter el sistema a datos de mercado reales sin comprometer capital, y constituye el instrumento principal de validación del prototipo (Secciones 3.7.2 y 3.7.4). Como salvaguarda adicional previa a cada operación, se empleó `callStatic` para anticipar el resultado de la transacción sin modificar el estado ni consumir gas (Sección 3.7.3).
- c) Pruebas en red de prueba pública: Se realizaron ejecuciones en la *testnet* Sepolia para evaluar la estabilidad del sistema en condiciones de red reales con menor exposición de riesgo y costo, permitiendo validaciones cruzadas entre los escenarios simulados y el comportamiento observado (Sección 3.7.6).

Las métricas empleadas para la evaluación fueron: la rentabilidad neta en dólares una vez descontados el costo de gas, la prima del préstamo *flash* y el deslizamiento; la capacidad de detección de oportunidades; el comportamiento de aborto seguro ante la desaparición de una oportunidad antes de su ejecución; y el consumo de *gas*.

La recolección de datos se efectuó mediante una arquitectura de registro estructurado que documenta, en cada ejecución, los datos de transacción, las rutas de intercambio, los detalles financieros y los indicadores de deslizamiento y liquidez, almacenados en formatos *.log* y *.json* conforme se detalla en la Sección 3.7.7. A partir de estos registros se identificaron y documentaron tres escenarios recurrentes que representan el espectro de resultados observados durante las pruebas de ejecución, los cuales se analizan en el Capítulo 4.

1.6.4 Evaluación de madurez tecnológica

El grado de avance alcanzado por el prototipo se valoró con la escala de Niveles de Madurez Tecnológica (TRL) [14], empleada como instrumento para situar el desarrollo dentro de un marco reconocido de progresión tecnológica. El nivel alcanzado y su justificación se discuten en el Capítulo 4 y en la Sección 5.2.

1.6.5 Reproducibilidad

Con el fin de favorecer la reproducibilidad, se documentan las versiones de las herramientas empleadas (Solidity 0.8.20, Hardhat, Ethers.js v5, OpenZeppelin 4.9 y Aave V3), así como la configuración del entorno de pruebas descrita en el Capítulo 3.

CAPÍTULO 2: MARCO TEÓRICO

2.1 Fundamentos de Criptomonedas y *blockchain*

Las criptomonedas y la tecnología *blockchain*¹ surgieron como respuesta a problemas concretos del sistema financiero tradicional: la dependencia de intermediarios, la opacidad en las transacciones y la vulnerabilidad ante puntos únicos de fallo. Desde la publicación del whitepaper de Bitcoin en 2008, el ecosistema ha crecido hasta superar los 20,000 activos digitales registrados, con una capitalización que alcanzó los 2.5 billones de dólares en 2024. La *blockchain*, como infraestructura subyacente, elimina la necesidad de una autoridad central al distribuir la verificación entre los nodos de la red, lo que dificulta la alteración retroactiva de los registros.

A partir de 2015, con el lanzamiento de Ethereum, la tecnología dejó de limitarse al registro de transferencias monetarias. Los contratos inteligentes (*smart contracts*²) permitieron codificar reglas de negocio directamente en la cadena, dando origen a las aplicaciones descentralizadas (*DApps*). Esta capacidad programable es la que habilita los mecanismos financieros que se analizan en el presente trabajo: préstamos sin colateral, intercambios automatizados y estrategias de arbitraje ejecutadas en una sola transacción.

2.1.1 Conceptos básicos de las criptomonedas y *blockchain*

Conforme a [15], la *blockchain*, o cadena de bloques, es una tecnología de registro distribuido que permite almacenar información digital de forma segura, transparente e inmutable. Consiste en una estructura de datos formada por bloques que contienen transacciones u otros datos, los cuales están criptográficamente enlazados entre sí y organizados cronológicamente. Cada bloque está vinculado al anterior mediante un hash criptográfico, lo que garantiza que, una vez añadida la información, esta no pueda ser alterada sin invalidar toda la cadena. Esta tecnología elimina la necesidad de intermediarios confiables, ya que todas las transacciones son verificadas colectivamente

¹ **Blockchain:** es una tecnología de registro distribuido que permite la creación de una base de datos segura, inmutable y descentralizada mediante la concatenación de bloques de datos criptográficamente protegidos. Cada bloque contiene un registro de transacciones verificadas, que son validadas por una red de nodos independientes, asegurando la integridad y transparencia del sistema sin necesidad de intermediarios centralizados.

² **Smart contracts:** son programas autoejecutables que se ejecutan en una *blockchain* y cuyo comportamiento está determinado por un conjunto de condiciones predefinidas. Estos contratos digitales permiten la automatización y verificación de acuerdos entre partes sin la necesidad de intermediarios, garantizando que las condiciones establecidas se cumplan de manera transparente y segura.

por una red descentralizada de nodos mediante algoritmos de consenso, como la prueba de trabajo (*proof-of-work*³) o la prueba de participación (*proof-of-stake*⁴).

Por otro lado, según lo expuesto por [16] las criptomonedas son activos digitales protegidos por criptografía diseñados para funcionar como medios de intercambio o reserva de valor, sin depender de una autoridad central, mediante transacciones electrónicas *peer-to-peer*⁵, las cuales son registradas en un libro mayor inmutable compartido y gestionado por los nodos de la red. Basado en [17], la tecnología *blockchain* actúa como la columna vertebral de las criptomonedas, proporcionando un sistema descentralizado y distribuido que registra todas las transacciones en múltiples computadoras (nodos) de manera que el registro no puede ser alterado retroactivamente sin la alteración de todos los bloques subsiguientes y el consenso de la red. Como ha sido documentado en [18], ésta tecnología no solo soporta el aspecto financiero de las criptomonedas, sino que también ha abierto el camino para aplicaciones inteligentes que pueden ejecutarse en la red, conocidas como *smart contracts* o contratos inteligentes. De acuerdo con [19] estos contratos son programas autoejecutables que llevan a cabo acciones automáticamente cuando se cumplen condiciones predeterminadas, permitiendo una diversidad de aplicaciones más allá de las transacciones financieras.

2.1.2 Tecnologías y protocolos blockchain

La tecnología *blockchain*, o cadena de bloques, constituye una innovación fundamental en el ámbito de los sistemas distribuidos y criptográficos. Su objetivo principal es permitir el registro descentralizado, transparente e inmutable de transacciones digitales, eliminando la necesidad de intermediarios confiables. Este concepto fue introducido formalmente con *Bitcoin*⁶ por [17], donde

³ **Proof of Work (PoW):** es un mecanismo de consenso en *blockchain* donde los participantes (llamados “mineros”) resuelven complejos problemas matemáticos utilizando hardware potente para validar transacciones y agregar nuevos bloques. Este proceso asegura la red mediante el uso intensivo de poder computacional, garantizando la integridad y descentralización de la *blockchain*.

⁴ **Proof of Stake (PoS)** es un mecanismo de consenso de *blockchain* alternativo a *PoW*. Este difiere de tal forma que, en lugar de utilizar potencia informática para validar transacciones, los validadores deben bloquear una cierta cantidad de criptomonedas para ayudar a proteger y respaldar las operaciones de una red *blockchain* a cambio de ser recompensados con criptomonedas adicionales (“*staking*”). Este hecho reduce drásticamente el consumo de energía necesario, mejorando la descentralización, seguridad y escalabilidad.

⁵ **Peer-to-peer:** Modelo de red en el que cada participante (nodo) actúa simultáneamente como cliente y servidor, permitiendo la transferencia directa de datos o activos entre usuarios sin necesidad de una autoridad central o intermediario.

⁶ **Bitcoin:** es la primera moneda digital descentralizada creada en 2008 por Satoshi Nakamoto, que permite realizar transacciones electrónicas sin necesidad de intermediarios financieros, mediante el uso de una red distribuida y segura basada en tecnología *blockchain*.

propuso un sistema de efectivo electrónico *peer-to-peer* basado en una red descentralizada, capaz de resolver el problema del doble gasto mediante una cadena de bloques protegida por criptografía y consenso distribuido.

Tal y como lo ha reportado en [19], a partir de ese diseño inicial, se han desarrollado nuevos modelos de *blockchain* que permiten aplicaciones más allá del dinero digital, incorporando contratos inteligentes, mayor complejidad computacional y casos de uso en diversas industrias. En términos generales según [20], se trata de una base de datos distribuida compuesta por bloques enlazados criptográficamente, que registra transacciones de forma cronológica y que es mantenida colectivamente por nodos en una red descentralizada. Estos bloques contienen datos validados mediante algoritmos de consenso y son resistentes a la alteración una vez registrados, garantizando la integridad del sistema.

Uno de los desarrollos más significativos en esta evolución es la aparición de *Ethereum*⁷, que amplía el modelo de *Bitcoin* al permitir la ejecución de contratos inteligentes (*smart contracts*). Estos son programas autoejecutables almacenados en la *blockchain* que automatizan acuerdos entre partes sin necesidad de intermediarios. Tal y como lo menciona [19], un contrato inteligente “puede utilizarse dentro de un ecosistema *blockchain* para negociar, ejecutar y hacer cumplir automáticamente las condiciones de un acuerdo legalmente vinculante” (p. 2, traducción propia). *Ethereum* logra esto mediante su *Ethereum Virtual Machine (EVM)*⁸, una máquina virtual distribuida que permite ejecutar cualquier contrato inteligente en todos los nodos de la red de forma consistente, creando así un entorno de computación descentralizada. De este modo, la *blockchain* no solo sirve como un libro contable como en el caso de *Bitcoin*, sino también como infraestructura para aplicaciones descentralizadas (*dApps*) que operan sobre reglas codificadas, inmutables y transparentes.

La importancia de esta evolución se refleja también en los estudios que exploran el uso de *blockchain* más allá del ámbito financiero. Por ejemplo, en [21] realizaron una revisión en la que concluyen que la *blockchain* ha sido adoptada en sectores como salud, educación, cadena de suministro, energía y administración pública, aprovechando sus propiedades de seguridad, trazabilidad y descentralización. En el artículo [22] refuerzan esta perspectiva al demostrar que los contratos inteligentes permiten

⁷ ***Ethereum***: es una plataforma *blockchain* descentralizada que permite la creación y ejecución de contratos inteligentes (*smart contracts*) y aplicaciones descentralizadas (*dApps*), mediante una máquina virtual integrada (*EVM*) que ejecuta código de forma distribuida en todos los nodos de la red.

⁸ ***Ethereum Virtual Machine (EVM)***: es el entorno de ejecución descentralizado de *Ethereum*, responsable de procesar y ejecutar contratos inteligentes en todos los nodos de la red de manera uniforme. Funciona como una máquina virtual Turing completa que garantiza que todos los participantes lleguen al mismo resultado computacional, asegurando la coherencia y seguridad del sistema.

automatizar procesos en múltiples sectores, mejorando la eficiencia operativa y reduciendo riesgos humanos.

Además de sus múltiples aplicaciones en distintos sectores, la estructura tecnológica de *blockchain* ha sido conceptualizada como un conjunto de “capas” que facilitan diferentes funciones dentro del ecosistema tecnológico. En este marco, las *blockchains* de primera capa (*Layer 1* o *L1*) representan las cadenas base que gestionan y aseguran de manera independiente transacciones y consenso, como Bitcoin y *Ethereum*, ya mencionados con anterioridad. Dado que las *L1* aún enfrentan la cuestión de escalabilidad y eficiencia, aparecieron soluciones de la segunda capa (*Layer 2* o *L2*) que están construidas sobre las primeras para descentralizar la carga transaccional y mejorar el rendimiento sin comprometer la seguridad proporcionada por el nivel base, de acuerdo con [23]. Algunos de los protocolos *L2* más populares, según [24], son *rollups*⁹, como *Optimistic*¹⁰ y *zk-rollups*¹¹ (en *Ethereum*), y *Lightning Network* (para Bitcoin), que agregan transacciones *off-chain*¹² y periódicamente consolidan los resultados de los canales a la base. Una indicación satisfactoria del problema y la solución puede observarse en la sistematización de tales protocolos en estudios académicos. Los protocolos *L2* muestran cómo optimizar los tiempos de procesamiento y disminuir los costes de transacción al descargar del trabajo en la *blockchain* principal una parte de éste.

Por otro lado, aparte de las *L1* y *L2*, los pasados años vieron aparecer también capas adicionales o *L3*, que proporcionan interoperabilidad, una funcionalidad específica o aplicaciones de alto nivel, como *multichain frameworks* o protocolos de comunicación *cross-chain*¹³, que mejoran la conectividad entre las redes y capacitan el tipo más avanzado de la *DeFi* o *dApps*. Esas capas superiores consolidan

⁹ **Rollup:** es una solución de escalabilidad de segunda capa (*Layer 2*) en *blockchains* como *Ethereum*, que permite procesar múltiples transacciones fuera de la cadena principal (*off-chain*), para posteriormente enviar una única transacción resumida a la cadena base (*Layer 1*). Esto reduce costos y congestión, manteniendo al mismo tiempo la seguridad y descentralización de la *blockchain* principal.

¹⁰ **Optimistic Rollups:** soluciones de escalabilidad de segunda capa (*L2*) que asumen que las transacciones son válidas por defecto (*optimistically*), y solo se ejecutan pruebas de fraude si se presenta una objeción durante un periodo de disputa. Esto reduce la carga computacional pero requiere ventanas de espera para garantizar la validez.

¹¹ **zk-Rollups (Zero-Knowledge Rollups):** Utilizan pruebas criptográficas llamadas *zero-knowledge proofs* para demostrar que un conjunto de transacciones es válido, sin revelar toda la información. Estas pruebas son verificadas por la cadena principal (*L1*), lo que permite una validación más rápida y segura, sin necesidad de periodo de disputa.

¹² **Off-Chain:** operaciones o transacciones que ocurren fuera de la cadena de bloques principal (*on-chain*). Estas interacciones pueden registrarse posteriormente en la *blockchain* o mantenerse fuera de ella para reducir costos, mejorar la velocidad o preservar privacidad.

¹³ **Cross-chain:** Mecanismo de interoperabilidad que facilita la comunicación, la lectura de estados y la transferencia de activos o datos entre dos o más redes *blockchain* independientes. Estas soluciones operan mediante protocolos de enlace (como puentes criptográficos) que superan el aislamiento natural de cada cadena, permitiendo que ecosistemas subyacentes distintos interactúen de forma segura.

una arquitectura más modular y escalable que permite aprovechar las ventajas de cada protocolo para un caso único de uso.

2.1.3 Ethereum Virtual Machine (EVM)

El cofundador de *Ethereum* Gavin Wood describe en [25] a la *EVM* como el entorno de ejecución que permite la implementación y ejecución de contratos inteligentes en la red *Ethereum*. Funciona como una máquina *cuasi-Turing*¹⁴ completa, lo que significa que puede ejecutar cualquier cálculo computacional dado suficiente tiempo y recursos, aunque con ciertas restricciones deliberadas impuestas para evitar ciclos infinitos, como el uso del *gas*¹⁵. Cada nodo de la red ejecuta la *EVM* de forma sincronizada, asegurando que los resultados de la ejecución sean deterministas y reproducibles en toda la red. El estado global de *Ethereum* se actualiza bloque a bloque mediante transacciones que modifican las cuentas, balances, almacenamiento y código asociado a cada dirección. Esta ejecución uniforme es clave para mantener la integridad y consenso en un sistema descentralizado.

Para comprender el funcionamiento interno de la *EVM*, es útil desglosar sus principales componentes (**¡Error! No se encuentra el origen de la referencia.**). La máquina se divide conceptualmente en estado del mundo (*world state*), que es persistente y contiene el almacenamiento asociado a las cuentas, y en el estado de máquina (*machine state*), que incluye tanto elementos persistentes como volátiles. Entre estos últimos se encuentran el contador de programa (*program counter PC*), que indica la siguiente instrucción a ejecutar; el *gas* disponible, que limita la ejecución de instrucciones; la pila (*stack*) para operaciones intermedias; la memoria temporal, y el almacenamiento transitorio que se utiliza durante la ejecución de cada llamada. Además, el código del contrato se almacena en una ROM virtual inmutable. Esta arquitectura modular permite que la *EVM* mantenga una ejecución segura y estandarizada entre todos los nodos de la red.

¹⁴ **Máquina Virtual cuasi-Turing:** Término que describe sistemas computacionales que poseen las capacidades de una máquina de *Turing* pero con limitaciones prácticas impuestas, como restricciones de tiempo o recursos. En el contexto de *Ethereum*, la *EVM* es considerada cuasi-Turing completa porque impone límites mediante el uso de “*gas*” para evitar ejecuciones infinitas.

¹⁵ **Gas:** Unidad de medida utilizada en *Ethereum* para cuantificar el costo computacional de ejecutar operaciones, como transacciones o llamadas a contratos inteligentes. Cada instrucción en la *EVM* tiene un precio en *gas*, y los usuarios deben pagar este costo en *ether (ETH)* para incentivar la ejecución y prevenir abusos como bucles infinitos o ataques de denegación de servicio

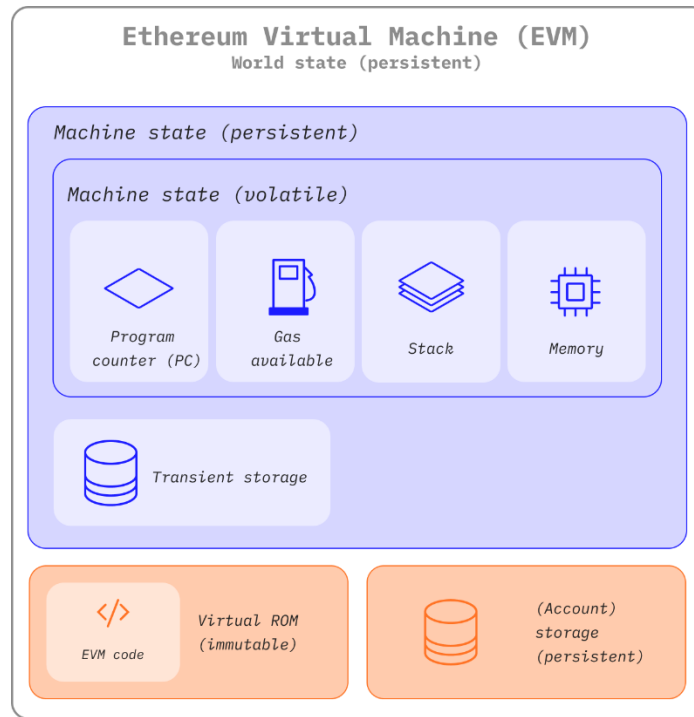


Figura 2.1: Componentes de la EVM

El *world state* puede entenderse como una función que asigna a cada dirección de la red una cuenta con atributos específicos, de acuerdo con [25]. Cada cuenta en *Ethereum* contiene cuatro componentes fundamentales: *nonce*, *balance*, *storage root* y *code hash*. El *nonce* es un contador que registra el número de transacciones enviadas desde una cuenta externa o el número de contratos creados por una cuenta de contrato; su función principal es prevenir ataques de repetición. El *balance* representa la cantidad de Ether asociada a la dirección. El *storage root* apunta al almacenamiento persistente del contrato, donde se guardan variables de estado definidas en el código. Finalmente, el *code hash* identifica el código ejecutable asociado a la cuenta.

El estado global puede representarse como:

$$\sigma : \text{Address} \rightarrow (\text{nonce}, \text{balance}, \text{storage}, \text{code})$$

Cada transacción aplicada a la red transforma este estado global en un nuevo estado mediante una función de transición determinista. Si una transacción se ejecuta correctamente, el estado cambia; si falla, el estado previo se conserva sin modificación. Esta propiedad de transición atómica del estado será especialmente relevante al analizar mecanismos como los préstamos *flash* en capítulos

posteriores, donde la validez de una operación depende de que el estado final cumpla ciertas condiciones.

2.1.4 Contratos Inteligentes (Smart Contracts): Fundamentos y Principios de Diseño

Los contratos inteligentes se definen en como programas o algoritmos inmutables, autoejecutables y autoverificables que codifican y aplican los términos de un acuerdo directamente en la infraestructura de la cadena de bloques. Según [19], un contrato inteligente es una colección de algoritmos resistentes a la manipulación, autoejecutables y autoverificables, cuya lógica se despliega y ejecuta de forma automática cuando se cumplen condiciones predefinidas en la red *blockchain*, sin requerir intervención humana para su ejecución ni intermediarios confiables.

La arquitectura subyacente de estos contratos opera sobre una máquina virtual descentralizada (como la *Ethereum Virtual Machine*), que aísla la ejecución del código del resto de la red para mantener la seguridad. De acuerdo con [26], el ciclo de vida de un contrato inteligente abarca desde la negociación y el desarrollo del código hasta su despliegue y ejecución final. Una característica crítica es que, al estar basados en *blockchain*, los datos y el estado del contrato son transparentes y verificables por todos los participantes de la red, lo que mitiga los riesgos de manipulación y fraude inherentes a los sistemas centralizados tradicionales.

Sin embargo, la transición de la teoría a la implementación práctica presenta desafíos significativos en la ingeniería de software. A diferencia del software convencional, los contratos inteligentes, una vez desplegados, son extremadamente difíciles de modificar debido a la propiedad de inmutabilidad de la cadena de bloques. Conforme a lo destacado en [27] esta rigidez convierte a los errores de programación en vulnerabilidades críticas y permanentes, a menudo explotadas por actores maliciosos para drenar activos financieros. Los desarrolladores enfrentan obstáculos únicos, como la optimización del costo de ejecución ("gas"), la gestión de la seguridad ante ataques de reentrada y la falta de herramientas de depuración maduras en comparación con otros entornos de desarrollo.

Más allá de la implementación técnica, existe una dicotomía compleja entre el código y la legalidad. En [28] argumentan que un contrato inteligente no es necesariamente un contrato legal en el sentido jurídico tradicional, sino más bien un mecanismo de ejecución automatizada. La investigación sugiere el uso de plantillas de contratos inteligentes para cerrar la brecha entre los términos legales en lenguaje natural y el código ejecutable. Este enfoque busca crear un vínculo estandarizado donde los parámetros del código se deriven directamente de acuerdos legales validados, asegurando que la automatización tecnológica cumpla con las normativas vigentes y reduzca la ambigüedad en la interpretación de los acuerdos.

2.1.5 Protocolos de Interoperabilidad y Estandarización de Activos en *Blockchain*

La estandarización de activos fungibles dentro de las redes *blockchain*, específicamente en ecosistemas de contratos inteligentes como *Ethereum*, representa uno de los pilares fundamentales para la interoperabilidad financiera descentralizada. Un activo fungible se define por su propiedad de intercambiabilidad idéntica y divisibilidad uniforme. Para lograr esto técnicamente, la industria ha convertido en protocolos de interfaz, siendo el *ERC-20*¹⁶ (*Ethereum Request for Comments 20*) el marco de referencia hegemónico.

Desde una perspectiva técnica, la implementación de un activo fungible no es meramente una entrada en un libro mayor, sino la adhesión estricta a una interfaz de programación de aplicaciones (*API*) dentro de un contrato inteligente. Como se establece en [29], el estándar *ERC-20* dicta seis funciones obligatorias:

1. **totalSupply()**: Es una función de lectura que devuelve el número total de *tokens* que existen en circulación dentro de ese contrato.
2. **balanceOf(account)**: Es una función de lectura que permite consultar cuántos tokens posee una dirección (*account*) específica en el libro mayor del contrato.
3. **transfer(recipient, amount)**: Es la función de escritura básica para transacciones directas. Mueve una cantidad (*amount*) de tokens desde la dirección de quien ejecuta la función (*msg.sender*) hacia la dirección del destinatario (*recipient*).
4. **approve(spender, amount)**: Es el mecanismo de delegación. Permite al dueño de los tokens autorizar a otra dirección (generalmente un contrato inteligente o *exchange*) a gastar hasta una cantidad específica (*amount*) de sus fondos en el futuro.
5. **transferFrom(sender, recipient, amount)**: Es la función de cobro indirecto. Permite que un tercero autorizado (el *spender*) mueva tokens desde la cuenta del dueño original (*sender*) hacia un destino (*recipient*), siempre que no exceda lo aprobado previamente.
6. **allowance(owner, spender)**: Es una función de lectura para **auditar permisos**. Devuelve la cantidad restante de tokens que el *spender* todavía tiene permiso para retirar de la cuenta del *owner*.

¹⁶ **ERC-20** (*Ethereum Request for Comments 20*): Es el estándar técnico o "manual de reglas" universal utilizado en la red *Ethereum* para crear tokens digitales. Su función principal es garantizar que todos los tokens sean **fungibles** (es decir, idénticos e intercambiables entre sí, como las monedas tradicionales) y compatibles con cualquier billetera o aplicación de la red.

Estas funciones permiten que aplicaciones externas interactúen con cualquier token de manera predecible sin necesidad de código personalizado para cada activo. Además, el estándar prescribe dos eventos críticos (*transfer* y *approval*), que facilitan la trazabilidad *off-chain* de las transacciones. Esta estandarización es crítica porque reduce la complejidad del desarrollo; cualquier billetera o *exchange* que integre el estándar puede interactuar automáticamente con miles de activos diferentes.

2.1.6 Gas y costos de ejecución

El concepto de "gas" en la tecnología *blockchain* actúa como una unidad de medida fundamental para cuantificar el esfuerzo computacional requerido para ejecutar operaciones en una red descentralizada. Como se establece en [30], el gas no es simplemente una tarifa monetaria, sino un mecanismo de medición de recursos que protege a la red contra bucles infinitos y ataques de denegación de servicio, asignando un costo específico a cada instrucción de código.

La determinación del costo total de ejecución ha evolucionado significativamente. En el modelo estándar de *Ethereum*, tras la implementación de la propuesta *EIP-1559*, la tarifa se compone de una "tarifa base" (*baseFee*) determinada algorítmicamente por la congestión del bloque anterior y una "propina" (*priorityFee*) de prioridad opcional, como lo sugiere [31]. Además, de acuerdo con [32], en entornos de Finanzas Descentralizadas (*DeFi*), se ha introducido el concepto de "Superficie de Costo de Gas" (Gas Cost Surface), un modelo que permite a los usuarios identificar momentos óptimos para transaccionar basándose en la volatilidad del mercado y el volumen de la transacción. Dado lo anterior, optimización del código de los contratos inteligentes es crucial para la reducción de costos. El análisis estático del código fuente en lenguajes como Solidity permite detectar patrones ineficientes, como el uso redundante de almacenamiento o bucles mal estructurados. De acuerdo con [31], la aplicación de optimizadores estáticos ha demostrado reducciones significativas en el consumo de gas tanto en el despliegue como en la invocación de contratos, sin comprometer su seguridad.

2.1.7 Implementación del EIP-1559

La propuesta de mejora *EIP-1559*, formalizada en [33] e implementada en la red *Ethereum* mediante la actualización London en agosto de 2021, transformó radicalmente el modelo de comisiones. Anteriormente, los usuarios competían en una "subasta a ciegas" (subasta de primer precio) que provocaba sobrepagos innecesarios. El nuevo modelo reemplazó este sistema por un mecanismo predecible compuesto por dos tarifas separadas: una tarifa base (*baseFee*) y una propina de prioridad (*priorityFee* o *tip*).

La **baseFee** es una tarifa obligatoria fijada algorítmicamente por el protocolo en función de qué tan congestionada esté la red. Para lograr esto, el sistema establece un tamaño de bloque "ideal" de 15 millones de unidades de *gas*. Si la demanda sube, el bloque puede expandirse temporalmente hasta un límite máximo de 30 millones. Cuando el bloque anterior supera la capacidad ideal, la tarifa del siguiente bloque aumenta según la siguiente regla de ajuste:

$$baseFee_{n+1} = baseFee_n \left(1 + \frac{gas_used_n - gas_target}{gas_target \times 8} \right) \quad (2.1)$$

El denominador de la formula asegura que el cambio de la tarifa nunca sea mayor al 12.5 % entre un bloque y otro, evitando saltos bruscos de precio. Un aspecto revolucionario de este mecanismo es que la **baseFee** se quema (*burn*¹⁷) en su totalidad, retirándose permanentemente de circulación. Por lo tanto, el único ingreso directo que reciben los validadores es la **priorityFee**, la cual es ofrecida voluntariamente por el usuario para incentivar que su transacción sea procesada más rápido. Como demuestra el estudio [34], este diseño alinea correctamente los incentivos económicos, ya que los usuarios no necesitan adivinar cuánto pagar y los validadores procesan las transacciones de manera más justa.

Para el diseño de un sistema de arbitraje automatizado, esta arquitectura facilita enormemente la estimación del costo de ejecución (C_{gas}). Al ser predecible, el algoritmo puede consultar la **baseFee** actual antes de enviar la transacción para conocer el costo mínimo requerido. De esta forma, la **priorityFee** se convierte en la herramienta estratégica principal del *bot* en la que el sistema puede calcular algorítmicamente cuánta propina añadir para acelerar su inclusión en el bloque y asegurar la oportunidad de arbitraje antes que sus competidores.

2.1.8 Interoperabilidad *Multi-Chain* y Protocolos *Cross-Chain*

El ecosistema *DeFi* se caracteriza por una constante y acelerada evolución tecnológica, donde la fragmentación de la liquidez se ha convertido en uno de los retos estructurales más significativos. Cuando un activo financiero cotiza de manera simultánea en redes heterogéneas y aisladas entre sí, la infraestructura local de cada cadena pierde la visión global del mercado. Esta separación crea una

¹⁷ **Burn (Quema):** Proceso mediante el cual se retiran criptomonedas de circulación de forma permanente e irreversible. En el modelo EIP-1559, la destrucción algorítmica de la tarifa base (**baseFee**) funciona como un mecanismo deflacionario que contrarresta la emisión de nuevas monedas en la red.

necesidad técnica imperativa por conectar estas "islas de información" para unificar el ecosistema. En este escenario, la tecnología *cross-chain* surge como la arquitectura de interoperabilidad indispensable para superar la fragmentación del entorno *multi-chain*¹⁸, permitiendo sincronizar el estado y establecer comunicación directa entre múltiples cadenas de bloques independientes. Esta tecnología no solo facilita la transferencia de valor, sino que permite a los contratos inteligentes operar bajo un consenso de precios unificado, logrando transmitir mensajes criptográficos entre máquinas virtuales que, por su propio diseño informático, operan con distintas latencias y tiempos de finalidad (*finality*).

Para que esta interoperabilidad funcione de manera fluida, segura y sin depender de entidades centralizadas que actúen como intermediarios, el desarrollo de protocolos *cross-chain* se ha apoyado fuertemente en redes de oráculos especializados. De acuerdo con lo expuesto en [35], estos oráculos *cross-chain* representan un componente crítico y fundamental para asegurar la integridad de los datos, especialmente en entornos que sufren de alta latencia de coordinación. Al actuar como retransmisores (*relays*) verificables, estos protocolos avanzados logran mitigar los riesgos estructurales que tradicionalmente han afectado a los puentes (*bridges*¹⁹) de primera generación. De esta forma, garantizan que la información que fluye de una red a otra sea exacta, inmutable y resistente a manipulaciones externas, lo cual es vital para el cálculo preciso de rentabilidades en el arbitraje automatizado.

A pesar de los grandes avances en esta infraestructura interconectada, la comunicación entre diferentes cadenas introduce nuevas e importantes vulnerabilidades que deben ser analizadas y mitigadas cuidadosamente al diseñar sistemas algorítmicos. Tal como ha sido documentado en [36], la falta de una sincronización global rigurosa y la asincronía natural que existe entre las distintas redes exponen a los protocolos a vectores de ataque temporal altamente sofisticados. En este contexto de vulnerabilidad, los actores maliciosos pueden observar las intenciones de transacción de un usuario

¹⁸ **Multi-chain (Multicadena):** Arquitectura en la que un protocolo o aplicación descentralizada se despliega simultáneamente en múltiples redes *blockchain*. Cada instancia opera de manera autónoma y paralela dentro de su respectivo entorno, gestionando sus propios contratos inteligentes y liquidez, sin mantener un estado compartido ni comunicación nativa con sus equivalentes en otras redes.

¹⁹ **Bridges:** Protocolos de infraestructura que facilitan la interoperabilidad entre dos redes *blockchain* independientes. Funciona como un enlace tecnológico que permite la transferencia de activos digitales, datos y mensajes de estado entre ecosistemas aislados, habitualmente bloqueando la liquidez en la red de origen y acuñando (*minting*) una representación sintética equivalente en la red de destino.

en una cadena de origen y explotar los tiempos de espera (latencia) para ejecutar ataques de sándwich (*sandwich attacks*) o extraer valor mediante estrategias anticipatorias en la cadena de destino. Por lo tanto, para garantizar la viabilidad de un sistema de arbitraje, comprender y protegerse contra estas discrepancias temporales es un requisito absoluto; de lo contrario, el valor extraído por terceros podría anular cualquier margen de ganancia una vez que la operación atraviesa las fronteras de una única cadena de bloques.

2.2 Introducción a Finanzas Descentralizadas (*DeFi*)

Las Finanzas Descentralizadas (*DeFi*²⁰), según [37], constituyen un nuevo paradigma financiero que emerge sobre la base de tecnologías *blockchain* y contratos inteligentes, proveyendo servicios financieros sin intermediarios tradicionales como bancos, casas de bolsa o cámaras de compensación. Desde una perspectiva académica, *DeFi* se define como un ecosistema de protocolos y aplicaciones sobre *blockchains* públicas, principalmente *Ethereum*, que busca replicar y mejorar funciones de los mercados financieros convencionales mediante software de código abierto y protocolos interoperables. Las Finanzas Descentralizadas se están posicionando como un cambio paradigmático en el panorama financiero global, proporcionando un acceso sin precedentes a servicios financieros para personas de todo el mundo. A través de la eliminación de intermediarios tradicionales y la promoción de un ecosistema financiero basado en tecnología *blockchain*, *DeFi* ofrece un modelo más inclusivo.

De acuerdo con [38], *DeFi* es un sistema financiero alternativo que habilita actividades como intercambio de activos, préstamos, depósitos, pagos, gestión de activos, mercados de derivados y seguros, operando en un entorno descentralizado y accesible globalmente. La infraestructura subyacente se basa en *smart contracts* autoejecutables que codifican reglas y acuerdos financieros que se ejecutan de forma determinista en la *blockchain*, reduciendo la necesidad de intermediación y aumentando la transparencia, composabilidad y disponibilidad de servicios financieros.

Aunque el ecosistema *DeFi* intenta emular servicios tradicionales, su diseño arquitectónico introduce características únicas como interoperabilidad entre protocolos, transparencia total de transacciones y gobernanza descentralizada mediante tokens. Dichos mecanismos permiten que múltiples aplicaciones descentralizadas (*dApps*) se construyan unas sobre otras, generando un ecosistema

²⁰ **DeFi:** Ecosistema financiero que opera sin entidades centralizadas, utilizando en su lugar contratos inteligentes en *blockchains*.

“componible” el cual [38] lo describe como el “dinero lego”, es decir un conjunto de piezas financieras que pueden combinarse para crear servicios más complejos.

A pesar del potencial de *DeFi* para aumentar la inclusión financiera, reducir costos de transacción y permitir la creación de mercados abiertos, los estudios de [39] también resaltan que la adopción generalizada enfrenta desafíos técnicos, de seguridad y regulatorios debido a la complejidad inherente de sus protocolos y la ausencia de supervisión centralizada.

2.2.1 Diferencias y ventajas sobre las Finanzas Centralizadas.

Las Finanzas Descentralizadas (*DeFi*) y las Finanzas Tradicionales, también referidas como *Centralized Finance* (*CeFi*), representan dos paradigmas financieros con fundamentos, estructuras operativas y mecanismos de confianza profundamente distintos. Mientras que el sistema financiero tradicional opera mediante intermediarios centralizados como bancos, cámaras de compensación y compañías de seguros que actúan como custodios del capital, gestores del riesgo y agentes de cumplimiento regulatorio, los protocolos *DeFi* eliminan o reducen drásticamente la necesidad de estos intermediarios, delegando la custodia de activos y la ejecución de servicios financieros a contratos inteligentes autoejecutables que corren sobre *blockchains* públicas. De acuerdo con [40], esta diferencia sitúa a *DeFi* como un ecosistema financiero que permite interacciones peer-to-peer directas, transparencia en las reglas de operación y un acceso global sin barreras de entrada tradicionales.

Desde la perspectiva de [40], se analiza de forma sistemática las diferencias entre finanzas centralizadas (*CeFi*) y finanzas descentralizadas (*DeFi*) en dimensiones legales, económicas, de seguridad y privacidad, destacando que *DeFi* tiende a ofrecer mayor transparencia y control al usuario, al tiempo que desafía conceptos de intermediación financiera clásica. Este estudio propone una metodología para distinguir entre servicios estrictamente *CeFi* y aquellos que pueden clasificarse como *DeFi*, subrayando que no todos los activos o servicios comúnmente asociados a *DeFi* son realmente descentralizados si dependen de entidades centralizadas para su gobernanza o custodio.

En [38] se señala que *DeFi* emula y extiende las funciones del sistema financiero tradicional, pero lo hace bajo principios de descentralización, interoperabilidad y ausencia de una autoridad central de control. Los autores subrayan que, aunque *DeFi* puede replicar servicios como préstamos, intercambios o gestión de activos, su arquitectura se basa en protocolos programables y contratos inteligentes que redefinen los mecanismos de confianza y reducción de contraparte.

No obstante, [38] sugiere que estas diferencias no implican una sustitución unilateral de los sistemas tradicionales por *DeFi*, sino más bien la coexistencia y, en algunos casos, una posible convergencia

híbrida. Estudios comparativos recientes muestran que ciertas operaciones financieras pueden requerir marcos regulatorios o estructuras institucionales para garantizar estabilidad, protección al consumidor y cumplimiento, aspectos que en *CeFi* están integrados por diseño pero que en *DeFi* aún enfrentan retos metodológicos y normativos.

2.2.2 Exchanges Descentralizados (DEXes) vs Exchanges Centralizados (CEXes)

Un *CEX* opera con un motor de *matching* que cruza órdenes en un libro límite central (*Central Limit Order Book*). Los usuarios depositan custodia (o al menos garantías) en el *exchange*; el *CEX* administra el libro, el enrutamiento, la prioridad (precio-tiempo), y la liquidación interna.

Un *DEX* permite intercambios *on-chain* mediante *smart contracts*. En muchos *DEX* modernos, el precio no surge de emparejar compradores y vendedores en un libro, sino de una función de intercambio (*conservation / trading*²¹ *function*) y reservas en *pools* de liquidez²². Esto reemplaza el “*peer-to-peer matching*” por un esquema *peer-to-pool*.

2.2.3 Mecánica de los DEXes

Los *exchanges* descentralizados (*DEXes*) surgen como una alternativa a los *exchanges* centralizados (*CEXes*), eliminando la custodia de activos y el uso de intermediarios confiables. En un *CEX*, la formación de precios se basa en un libro de órdenes centralizado (*Central Limit Order Book*) administrado por una entidad que empareja órdenes de compra y venta según reglas de prioridad de precio y tiempo. En contraste, los *DEXes* ejecutan intercambios directamente sobre una *blockchain* mediante contratos inteligentes, donde la liquidación ocurre de forma determinista y sin custodia centralizada. Esta diferencia estructural implica que, mientras los *CEX* dependen de participantes activos que proveen liquidez mediante órdenes limitadas, los *DEX* basados en *AMMs* reemplazan el proceso de emparejamiento por reglas algorítmicas de intercambio contra *pools* de liquidez.

²¹ **Trading (negociación o comercio):** En el ámbito financiero, se refiere a la compra y venta de activos (como acciones, divisas, materias primas, criptomonedas, etc.) en mercados regulados o no regulados, generalmente a corto o mediano plazo. Su objetivo principal es generar rentabilidad capitalizando las fluctuaciones constantes de los precios.

²² **Pools de Liquidez:** Conjunto de activos bloqueados en un contrato inteligente que actúa como contraparte algorítmica para las operaciones de intercambio en un *DEX*. Los precios y las cantidades intercambiables se determinan mediante una función matemática invariante que gobierna las reservas del *pool*, permitiendo el comercio continuo sin un libro de órdenes tradicional. Los participantes que aportan activos al *pool*, llamados *liquidity providers*, reciben comisiones proporcionales al uso del *pool* como compensación por proveer liquidez y asumir riesgo.

Se describe formalmente a los *DEX* modernos, de acuerdo con [41], como sistemas *peer-to-pool*, en contraposición al modelo *peer-to-peer* de los mercados tradicionales, destacando que la formación de precios en *AMMs* no depende de expectativas humanas directas sino de funciones matemáticas de conservación que gobiernan las reservas del *pool*.

2.2.4 Protocolos de Intercambio Descentralizado y Automated Market Makers (AMMs)

Los *Automated Market Makers (AMMs)* son definidos por [42] como mecanismos algorítmicos que sustituyen el descubrimiento de precios basado en órdenes por reglas determinísticas embebidas en contratos inteligentes. Estos constituyen el mecanismo predominante en los *DEX* actuales para la provisión de liquidez y la formación de precios. En lugar de un libro de órdenes, los *AMMs* utilizan pools de liquidez, donde pares de activos son depositados por proveedores de liquidez (*Liquidity Providers*). El intercambio se realiza directamente contra el *pool*, y el precio resulta de una función matemática invariante que define cómo deben ajustarse las reservas tras cada operación. Esta arquitectura permite la negociación continua incluso en ausencia de contrapartes directas, a costa de introducir una relación no lineal entre tamaño de la orden y precio ejecutado.

Uno de los modelos de *AMM* más influyentes es el *Constant Product Market Maker (CPMM)*. En este modelo, el estado del *pool* está definido por dos reservas correspondientes a dos criptoactivos, cuyo producto debe permanecer constante, es decir:

$$x \cdot y = k \quad (2.2)$$

Esta ecuación define el conjunto de estados válidos del *pool*. Cualquier operación de intercambio debe mover el sistema a lo largo de esta hipérbola, alterando las reservas relativas de los activos. Como consecuencia, el precio implícito de un token en términos del otro puede aproximarse por la razón $\frac{y}{x}$, y cambia automáticamente tras cada trade, sin necesidad de órdenes explícitas.

Las propiedades económicas de los *CPMMs* fueron analizadas por [43] demostrando que la convexidad de la función impone costos de ejecución crecientes conforme el tamaño relativo de la orden aumenta. Este resultado fundamenta matemáticamente por qué los precios en *AMMs* divergen de los precios externos cuando se ejecutan órdenes grandes.

2.2.5 Impacto de Precio (Price Impact) y Dinámica Asintótica del Precio

Cuando un usuario compra uno de los activos del *pool*, la cantidad disponible de dicho activo disminuye, mientras que el otro se vuelve relativamente más abundante. Debido a la restricción, esta modificación fuerza al precio implícito a moverse de forma no lineal y asintótica. A medida que una reserva se aproxima a cero, el costo marginal de adquirir unidades adicionales tiende al infinito, lo que impide el vaciamiento total del *pool*.

Este fenómeno desde la perspectiva de [44], muestra que el *impacto* de precio en *AMMs* es una consecuencia directa de la forma funcional de la invariante, y no de fricciones externas. Se demuestra que esta característica genera oportunidades sistemáticas de arbitraje entre *AMMs* y otros mercados cuando existen discrepancias temporales de precios.

2.2.6 Deslizamiento (Slippage) como Consecuencia Matemática

El concepto de *slippage* es descrito por [41], como la diferencia entre el precio esperado de una operación y el precio promedio efectivamente ejecutado. En el contexto de *AMMs*, el *slippage* no es un error operativo, sino una consecuencia estructural de la curva de intercambio. Matemáticamente, puede expresarse como una función creciente del cociente entre el tamaño de la orden y la liquidez disponible en el *pool*. Cuanto mayor es la proporción de la reserva que se intercambia, mayor es la desviación del precio respecto al spot inicial.

$$Slippage = \frac{P_{exec} - P_{spot}}{P_{spot}} \quad (2.3)$$

donde:

- P_{spot} es el precio marginal antes del swap, es decir el precio spot del *AMM* inmediatamente antes de la operación, definido como:

$$P_{spot} = \frac{y}{x} \quad (2.4)$$

donde:

- x es la cantidad del activo X almacenada en el *pool*.
- y es la cantidad del activo Y almacenada en el *pool*.
- P_{exec} es el precio de ejecución promedio, definido como:

$$P_{exec} = \frac{\Delta x}{\Delta y} \quad (2.5)$$

donde:

- Δx es la cantidad total del activo de entrada (*input amount*),
- Δy es la cantidad total del activo de salida (*output amount*).

Esta definición expresa el precio efectivo pagado por unidad del activo recibido, y es consistente con la convención habitual en microestructura de mercados y en el análisis académico de *AMMs*.

Esta formulación se enmarca dentro del análisis donde se establece una distinción explícita entre: *spot price* (precio marginal previo a la operación), *execution price* (precio de ejecución promedio efectivamente realizado), y *slippage* como la desviación relativa entre ambos precios.

Se considera en [41] que el *slippage* es una propiedad inherente a los *AMMs* basados en funciones invariantes, y que su comportamiento puede derivarse analíticamente a partir de la función de intercambio del protocolo.

2.2.7 Liquidez Concentrada como Evolución del Modelo CPMM

El modelo tradicional de producto constante (*CPMM*) tiene una característica particular: distribuye la liquidez de manera uniforme a lo largo de toda la curva de precios, desde cero hasta el infinito. Aunque esto garantiza que siempre habrá liquidez disponible sin importar el nivel de precio, en la práctica resulta sumamente ineficiente, ya que la mayor parte del capital depositado se queda estancado e inactivo en rangos donde simplemente no hay actividad comercial.

Para resolver esta ineficiencia, el protocolo Uniswap V3, formalizado en [45], introduce el concepto de “liquidez concentrada” (*concentrated liquidity*), que permite a los proveedores de liquidez elegir exactamente en qué rango de precios $[p_a, p_b]$ desean colocar su capital. Dentro de ese intervalo específico, la posición funciona de manera idéntica a un *pool* tradicional, pero con reservas virtuales amplificadas, lo cual se representa matemáticamente con la fórmula:

$$\left(x + \frac{L}{\sqrt{p_b}}\right)(y + L\sqrt{p_a}) = L^2 \quad (2.6)$$

donde:

- L : representa la cantidad de liquidez.
- x : es la cantidad real del activo base (ej. el Token X) que existe físicamente dentro de esa posición de liquidez en ese momento.

- y : es la cantidad real del activo de cotización (ej. el Token Y) que existe físicamente en esa posición.
- p_a : es el límite inferior (precio mínimo) del rango que el proveedor eligió para su capital.
- p_b : es el límite superior (precio máximo) de ese mismo rango.

Para lograr esta precisión técnica, el espacio de precios se divide en segmentos minúsculos llamados *ticks*²³, donde cada *tick* i equivale a un incremento de un punto base, es decir, $p(i) = 1.0001^i$. Así, conforme el precio de mercado se mueve y cruza un *tick* activo, la liquidez asociada a ese punto se inyecta o se retira automáticamente de la liquidez global disponible.

Este cambio de arquitectura transforma por completo la dinámica del *slippage* y el impacto de precio. De acuerdo con lo analizado en [46], concentrar posiciones en rangos muy cercanos al precio actual de mercado hace que la curva de reservas sea mucho más plana comparada con la versión anterior (V2). Para el usuario, esto se traduce en una reducción drástica del deslizamiento al operar dentro de ese rango. De hecho, los autores demuestran cuantitativamente que estrategias sencillas de concentración pueden llegar a generar hasta 200 veces más comisiones que una aportación idéntica en Uniswap V2, confirmando su inmensa eficiencia de capital.

No obstante, este beneficio tiene un costo: la concentración amplifica considerablemente el riesgo de pérdida impermanente (*impermanent loss*) si el precio sale del rango elegido. Esto obliga al proveedor de liquidez a buscar un equilibrio constante entre rentabilidad y riesgo, un dilema que el modelo uniforme anterior no presentaba.

Finalmente, desde la perspectiva del diseño de un sistema de arbitraje automatizado, esta distribución desigual de la liquidez cambia por completo las reglas del juego. El impacto de precio de una operación ya no se puede estimar simplemente tomando la proporción global del *pool* ($\frac{y}{x}$). Ahora, el algoritmo debe leer e interpretar la configuración exacta de las posiciones en cada *tick* individual, lo que eleva significativamente la complejidad computacional necesaria para calcular la rentabilidad real de la operación antes de ejecutarla.

²³ **Tick:** Intervalo discreto que particiona la curva continua de precios en un creador de mercado automatizado (AMM). Equivale a una variación de un punto base (0.01 %) y funciona como la coordenada exacta para inyectar o retirar liquidez en un rango específico.

2.3 Teoría del Arbitraje Financiero

En la literatura de economía financiera, el arbitraje es definido, de acuerdo con [47] como una estrategia que explota diferencias de precios de activos equivalentes en dos (o más) mercados para obtener un beneficio sin riesgo neto y sin inversión inicial negativa. Este principio es fundamental tanto en mercados tradicionales como en mercados de criptomonedas, pues representa la ausencia de oportunidades de “ganancia libre de riesgo”, condición requerida para equilibrio de precios en modelos financieros. Esta definición es central en teoría de precios y modelos económicos que excluyen la posibilidad de beneficios arbitrarios prolongados.

2.3.1 Hipótesis del Mercado Eficiente

Tal como se menciona en [48], los mercados que reflejan toda la información disponible tienden a eliminar rápidamente oportunidades de arbitraje. La existencia recurrente de ineficiencias sugiere grados de eficiencia imperfecta, especialmente en mercados emergentes como los de criptomonedas. Estudios sobre la eficiencia dinámica del mercado Bitcoin, mencionados en [3] muestran cómo las oportunidades de arbitraje se reducen cuando mejora la eficiencia de precios, confirmando que la actividad arbitraria tiende a corregir brechas de información y valores relativos.

2.3.2 Tipos de Arbitraje en el Mercado de Criptomonedas

En los mercados de criptomonedas existen varios mecanismos de arbitraje que se alinean con las definiciones teóricas de explotación de diferencias de precios, adaptadas a la microestructura particular de *exchanges* centralizados y descentralizados. Dicha microestructura genera oportunidades que los agentes de arbitraje buscan explotar caracterizadas por fragmentación, heterogeneidad de liquidez y diferencias en velocidad de ejecución.

2.3.2.1 Arbitraje Espacial

También conocido como “Cross-Market-Arbitrage”, este tipo de arbitraje implica comprar un activo en un mercado donde cotiza a un precio más bajo y vender inmediatamente en otro donde su precio es más alto, explotando así las diferencias de precios de un mismo activo. En mercados eficientes, estas discrepancias deberían desaparecer rápidamente; sin embargo, debido a variaciones en liquidez, tiempo de actualización de precios, costos de transacción y estructura regulatoria, las

diferencias de precios pueden persistir temporalmente, dando lugar a oportunidades reales de arbitraje.

La ecuación de la brecha de precios (gap) entre dos mercados se modela formalmente en [49] como:

$$Arbitrage\ spread = \frac{P_B - P_A}{P_A} \quad (2.7)$$

donde:

- p_A = precio del activo en el mercado A.
- p_B = precio del mismo activo en el mercado B.

Estudios empíricos como los de [3] han demostrado que los mercados de criptomonedas exhiben desviaciones persistentes de precios entre *exchanges* que son aprovechables por arbitrajistas, aunque estas oportunidades tienden a desaparecer con el tiempo a medida que los precios convergen.

2.3.2.2 Arbitraje Triangular

Este tipo de arbitraje consiste en explotar discrepancias de precios entre 3 activos o pares de trading dentro de un mismo mercado (o *exchange*). El concepto básico es que existe un ciclo cerrado en el que un activo A se intercambia por un activo B, luego B por C, y finalmente C por A, con un resultado neto que debería ser neutral si los precios son consistentes. Donde existen micro ineficiencias en la cotización relativa de pares, este ciclo puede resultar en una ganancia neta sin exposición al mercado. Sin embargo, investigaciones como la de [50], demuestra que estas oportunidades no siempre son rentables después de considerar costos de transacción, liquidez y volumen de mercado.

La expresión matemática del arbitraje triangular se expresa comúnmente en literatura financiera como:

$$\frac{E_{AB} \cdot E_{BC}}{E_{AC}} \neq 1 \quad (2.8)$$

donde:

- E_{AB} es el tipo de cambio entre los activos A y B.
- E_{BC} es el tipo de cambio entre los activos B y C.
- E_{AC} es el tipo de cambio directamente entre los activos A y C.

Si ese cociente es distinto de 1 después de costos de transacción, existe una desviación arbitraria que puede convertirse en una potencial operación de arbitraje.

2.3.2.3 Arbitraje Intertemporal

El arbitraje intertemporal se refiere a la estrategia que busca explotar diferencias de precios de un mismo activo en diferentes momentos del tiempo, bajo la expectativa de que su valor cambiará por razones conocidas o anticipadas. Este tipo de arbitraje es común en mercados con alta volatilidad o cuando la información relevante aún no ha sido completamente incorporada en los precios de los activos.

La lógica detrás de esta estrategia se basa en el principio de expectativas racionales mencionada en [51], en la que si se espera que el precio de un activo aumente en el futuro, pero actualmente está subvalorado, un agente puede comprar hoy y beneficiarse de su revalorización, siempre que este diferencial no esté completamente ajustado en el precio actual. Es especialmente aplicable en mercados emergentes o en instrumentos con baja liquidez, donde la incorporación de la información es más lenta.

La condición general de arbitraje intertemporal, de acuerdo con [51], se expresa como:

$$P(t) < \mathbb{E}[P(t + \Delta t)] - C_t \quad (2.9)$$

donde:

- $P(t)$: precio actual del activo en el instante t .
- $\mathbb{E}_t[P(t + \Delta t)]$: valor esperado del precio en el futuro ($t + \Delta t$), condicionado a la información disponible en t .
- $C(t)$: costos asociados a mantener la posición hasta el tiempo $t + \Delta t$ (incluyendo financiamiento, almacenamiento, riesgo, etc.).

Esta condición indica que, si el precio esperado supera al precio actual más los costos de mantenimiento, existe una ganancia neta esperada que puede generar una oportunidad rentable de arbitraje.

2.3.2.4 Arbitraje Estadístico

El arbitraje estadístico es una estrategia basada en modelos cuantitativos que identifican relaciones históricas entre activos, como la cointegración o la alta correlación en series temporales. Cuando estas relaciones se desvían, se espera una convergencia futura que puede aprovecharse mediante posiciones de compra o venta.

Una formulación típica, de acuerdo con [52], supone que el spread entre dos activos sigue una distribución normal:

$$x_t - y_t \sim \mathcal{N}(\mu, \sigma^2) \quad (2.10)$$

donde:

- x_t : precio del activo A en el instante t .
- y_t : precio del activo B en el instante t .
- μ : media histórica del spread $x_t - y_t$.
- σ^2 : varianza del spread, medida de su dispersión.

Si en un momento dado el valor del spread se aleja de forma significativa de μ , se puede abrir una posición de arbitraje basada en la expectativa de que el diferencial se revertirá hacia su media. Esto se implementa, por ejemplo, comprando el activo infravalorado y vendiendo el sobrevalorado, esperando su convergencia.

2.4 Automatización y Sistemas Algorítmicos en DeFi

En los mercados financieros tradicionales, las oportunidades de arbitraje suelen desaparecer en cuestión de segundos, en algunos casos un operador humano con experiencia podría detectar y ejecutar manualmente la operación. En el ecosistema de las Finanzas Descentralizadas (*DeFi*), esta dinámica se intensifica, ya que los precios cambian en fracciones de segundo. La velocidad de reacción necesaria para aprovechar las diferencias de precios supera por completo la capacidad humana. Por esta razón, el arbitraje en la *blockchain* no puede realizarse de forma manual; requiere la construcción de sistemas algorítmicos autónomos, comúnmente llamados *bots*, que trabajan sin intervención manual.

2.4.1 Trading algorítmico de alta frecuencia en mercados descentralizados

El *High-Frequency Trading (HFT)* es una modalidad de trading algorítmico caracterizada por la ejecución de un gran número de órdenes en fracciones de segundo. adquiere en *DeFi* una forma distinta a la observada en bolsas tradicionales. En los mercados financieros tradicionales, la ventaja competitiva del *HFT* radica en la proximidad física de los servidores a las bolsas de valores para reducir la latencia de red. No obstante, en la cadena de bloques, el terreno de juego cambia por

completo. El mercado no es continuo, sino que se actualiza por bloques de información cada cierta cantidad de segundos.

En *DeFi* la competencia no es física, sino computacional, el mercado no opera de forma continua, sino en lotes discretos determinados por el tiempo de generación de cada bloque. Tal como lo demuestra [53], el HFT descentralizado no compite por proximidad física, sino por la capacidad de procesar información asíncrona de la red P2P (*mempool*) y optimizar el costo de ejecución (*Gas*). Un sistema algorítmico competitivo debe ser capaz de detectar transacciones pendientes, simular su impacto en las reservas de los *AMMs* y enviar una orden pre-calculada antes de que la red alcance un nuevo consenso. Esto exige una arquitectura computacional altamente reactiva, capaz de procesar flujos de datos en tiempo real y ejecutar lógica de decisión estricta.

2.4.2 Arquitectura general de los *bots* de arbitraje

Para lograr la autonomía operativa y la velocidad requerida en el HFT descentralizado, el *bot* debe dividirse en componentes modulares que separen el procesamiento intensivo de la ejecución determinista. Según [54], la arquitectura estándar de un sistema de arbitraje se fundamenta en un paradigma de diseño híbrido *off-chain/on-chain*, compuesto por tres capas teóricas principales:

1. **Capa de Ingesta y Sincronización de Estado (Indexador):** Es el módulo *off-chain* encargado de mantener una réplica exacta y en tiempo real del estado de los contratos inteligentes en la *blockchain*. Utiliza conexiones persistentes (como *WebSockets*²⁴) a través de proveedores de nodos *RPC*²⁵ para escuchar eventos de actualización de reservas. Su función es alimentar al sistema con los precios exactos al milisegundo sin tener que consultar la *blockchain* reiteradamente, lo cual introduciría latencia al modelo.
2. **Motor de Decisión Computacional (“El Cerebro”):** Es el núcleo algorítmico del sistema. Recibe el estado de los *pools* desde el Indexador y ejecuta modelos matemáticos para buscar discrepancias de precios. Este módulo debe ser programado en lenguajes de alto rendimiento o

²⁴ **Websockets:** Protocolo de comunicación informática que establece una conexión bidireccional y continua entre un cliente y un servidor. A diferencia de las consultas web tradicionales (HTTP), permite la transmisión de datos en tiempo real de forma ininterrumpida, siendo el estándar para aplicaciones que requieren recibir actualizaciones instantáneas sin necesidad de enviar solicitudes repetitivas al servidor

²⁵ **RPC (Remote Procedure Call):** Protocolo de comunicación en red que permite a un programa de computadora solicitar la ejecución de una función o servicio a otro programa ubicado en un ordenador diferente (servidor). Facilita la interacción en sistemas distribuidos al permitir que el desarrollador invoque procesos remotos como si fueran locales, abstrayendo la complejidad de la red subyacente.

tipado estricto para calcular el tamaño óptimo de la operación, restar la tarifa dinámica de la red (*Gas*) y el impacto en el precio (*slippage*). Solo si la función de rentabilidad arroja un valor estrictamente mayor a cero, este módulo genera la orden de ejecución.

3. **Capa de Ejecución Transaccional (*Smart Contracts*):** Es el componente *on-chain* que reside permanentemente en la *blockchain*. Recibe la instrucción criptográficamente firmada por el Motor de Decisión e interactúa directamente con los protocolos *DeFi* (como solicitar el *Flash Loan* y realizar los *swaps*). Su diseño debe ser atómico: si por alguna alteración del mercado en el último segundo la operación deja de ser rentable, la ejecución revierte algorítmicamente todo el estado para proteger el capital.

2.4.3 Teoría y Diseño de Algoritmos para la Detección de Oportunidades

Una vez que la arquitectura garantiza la velocidad de reacción, el desafío computacional central es cómo el motor de decisión identifica la ruta exacta de *tokens* que genera ganancias. Para resolver esto, la topología del ecosistema *DeFi* se modela matemáticamente utilizando la Teoría de Grafos²⁶.

En este modelo, el mercado se representa como un grafo dirigido ponderado:

$$G = (V, E, W) \quad (2.11)$$

donde:

- **Vértices o Nodos (*V*):** Representan los distintos criptoactivos disponibles.
- **Aristas (*E*):** Representan los *pools* de liquidez que conectan dos *tokens*.
- **Peso de la Arista (*W*):** Es el tipo de cambio efectivo entre los dos *tokens*, descontando matemáticamente la comisión del protocolo y el costo del *gas* necesario para realizar el salto.

En [55] se aborda este problema de detección algorítmica, demostrando que la búsqueda de ganancias equivale a detectar rutas y ciclos matemáticos rentables dentro del grafo de *tokens*. Para resolver este problema, el estándar de las ciencias computacionales es adaptar variantes del algoritmo de *Moore-Bellman-Ford*²⁷ (utilizado para encontrar caminos óptimos). El algoritmo revisa miles de rutas en

²⁶ **Teoría de Grafos:** es una disciplina matemática que modela las relaciones entre elementos mediante redes compuestas por "nodos" (las entidades individuales) y "aristas" (las líneas que las conectan). En la práctica, es la base lógica que permite analizar y optimizar sistemas interconectados, como rutas logísticas, arquitecturas de datos o mapas de navegación.

²⁷ **Moore-Bellman-Ford:** El algoritmo de Moore-Bellman-Ford es un método matemático de teoría de grafos que encuentra el camino óptimo en redes que tienen valores negativos. En este proyecto se utiliza porque es

milisegundos buscando un ciclo que inicie con una moneda y regrese a la misma con una cantidad matemáticamente superior, transformando el arbitraje financiero en un problema puramente computacional y verificable.

2.5 Introducción a los préstamos relámpago (*Flash Loans*)

El concepto de *flash loan* constituye una innovación financiera posibilitada por la arquitectura transaccional de las *blockchains* programables mediante *smart contracts*. Se trata de un mecanismo de crédito sin colateral cuya validez depende exclusivamente de la ejecución atómica²⁸ de una transacción dentro de un sistema distribuido. A diferencia del crédito tradicional, en el que la garantía mitiga el riesgo de incumplimiento, en un préstamo *flash* el riesgo se elimina mediante lógica computacional: el capital debe solicitarse, utilizarse y devolverse junto con una comisión dentro de una única transacción que se ejecuta en un mismo bloque. Si la devolución no ocurre, la transacción completa se revierte como si nunca se hubiera llevado a cabo. Esta posibilidad emerge del modelo de ejecución determinista introducido en [17] y ha sido analizada desde el punto de vista de su semántica formal y propiedades de seguridad en [56].

Desde la perspectiva de las finanzas descentralizadas (*DeFi*), los préstamos *flash* representan una primitiva que solo puede existir gracias a la atomicidad y a la computabilidad de los contratos inteligentes. La literatura sistemática sobre *DeFi* identifica a los protocolos de préstamo como una de las infraestructuras fundamentales del ecosistema, junto con los intercambios descentralizados (*DEX*) y los oráculos de precios. En este contexto, de acuerdo con [57], los *flash loans* surgen como una extensión particular de los modelos de préstamo basados en *liquidity pools*, permitiendo que la liquidez depositada por proveedores sea utilizada de manera temporal dentro de una única transacción.

capaz de detectar matemáticamente 'ciclos de peso negativo', lo que en finanzas descentralizadas equivale a encontrar una ruta de arbitraje circular exacta donde el balance final supera al capital inicial.

²⁸ **Ejecución Atómica:** Es una propiedad de una transacción según la cual todas sus operaciones se ejecutan completamente o no se ejecuta ninguna. Si ocurre un error durante el proceso, el sistema revierte automáticamente todos los cambios y el estado vuelve al punto anterior, como si la transacción nunca hubiera sucedido.

2.5.1 Atomicidad y eliminación del riesgo de incumplimiento

La atomicidad es el principio técnico que sustenta la viabilidad de los préstamos *flash*. En términos formales, una transacción en *Ethereum* se ejecuta bajo una semántica de “todo o nada”: si alguna instrucción produce un error o no se satisface una condición requerida, el estado global del sistema se revierte al punto previo a la ejecución. Tal como se demuestra en [56], la ejecución de contratos inteligentes puede modelarse como una transición determinista de estados donde los efectos parciales no persisten ante fallos.

En el caso del préstamo *flash*, esta propiedad se traduce en una garantía lógica para el prestamista: el capital solo sale del *pool* si regresa dentro de la misma transacción junto con la comisión correspondiente. Si la estrategia ejecutada por el prestatario, por ejemplo, una operación de arbitraje no genera el rendimiento esperado, la función de repago falla y la transacción completa se revierte. De esta manera, el riesgo de insolvencia individual se elimina a nivel técnico, sustituyendo la garantía económica tradicional por una garantía computacional.

2.5.2 Funcionamiento de los protocolos de préstamo

Los préstamos *flash* se implementan dentro de protocolos de préstamo que operan mediante *pools de liquidez*. En estos sistemas, los usuarios depositan activos digitales en contratos inteligentes que actúan como reservas colectivas. Dichos fondos pueden ser prestados bajo diferentes modalidades como los préstamos sobrecolateralizados tradicionales o préstamos *flash* sin garantía. La diferencia esencial entre ambos radica en la duración del préstamo y en la condición de devolución. Mientras que en el modelo tradicional el prestatario debe aportar colateral superior al valor solicitado, en el préstamo *flash* la garantía se sustituye por una restricción computacional: la obligación de devolver el capital dentro de la misma transacción en la que fue emitido.

En [8] se realizan uno de los primeros estudios empíricos sistemáticos sobre el uso de *flash loans* en el ecosistema *DeFi* y describen su funcionamiento como una secuencia compuesta de tres fases dentro de una sola transacción: 1) solicitud del préstamo, 2) ejecución de una estrategia, y 3) devolución del principal más la tarifa exigida por el protocolo.

Esta secuencia puede representarse como una transición de estados dentro de la Máquina Virtual de *Ethereum* (*EVM*). En este contexto, el “estado” se entiende como el conjunto de balances, reservas y variables almacenadas en los contratos inteligentes en un momento determinado.

2.5.3 Aave V3 como protocolo de préstamo

Entre los protocolos de préstamo que implementan esta primitiva, *Aave* se ha consolidado como la infraestructura dominante para préstamos relámpago en el ecosistema *Ethereum*. En su tercera versión, descrita en [58], la arquitectura del protocolo se organiza en torno a un contrato central denominado *Pool*, que actúa como punto de entrada para todas las operaciones de préstamo, depósito y liquidación, y cuya dirección se obtiene mediante un registro centralizado llamado `PoolAddressesProvider`. Para solicitar un préstamo relámpago de un solo activo, el contrato del usuario invoca la función `flashLoanSimple()` sobre el *Pool*, especificando el token solicitado y el monto deseado. Acto seguido, el *Pool* transfiere los fondos al contrato receptor e invoca automáticamente la función de `callback` `executeOperation()`, definida por la interfaz `IFlashLoanSimpleReceiver`, cediendo temporalmente el control de ejecución al prestatario para que implemente su estrategia (por ejemplo, una secuencia de intercambios de arbitraje). Una vez que `executeOperation()` retorna el control, el *Pool* verifica que el saldo devuelto sea igual al capital prestado más una prima del 0.05% (cinco puntos base), almacenada en la variable `FLASHLOAN_PREMIUM_TOTAL` y configurable mediante gobernanza del protocolo. Si esta condición no se cumple, la propiedad de atomicidad de la *EVM* garantiza la reversión completa de la transacción. Como ha sido formalizado en [59], este patrón de solicitud-ejecución-devolución puede modelarse como una transición determinista de estados dentro de un *pool* de liquidez, donde la operación de préstamo relámpago constituye una composición atómica de acciones que preserva los invariantes del sistema.

Sea S_0 el estado del sistema inmediatamente antes de ejecutar la transacción del préstamo *flash*. Este estado incluye, por ejemplo, las reservas del *pool* de liquidez, los balances del prestatario y las variables internas del contrato. Sea S_f el estado del sistema tras intentar ejecutar completamente la secuencia préstamo–estrategia–devolución.

Sea L el monto solicitado y α la comisión proporcional del protocolo. La transición de S_0 a S_f solo será válida si se cumple la condición:

$$f(L) \geq L(1 + \alpha) \quad (2.12)$$

donde:

- $f(L)$ representa el valor total disponible tras ejecutar la estrategia utilizando el capital prestado.
- L es el capital prestado.

- α representa la comisión del protocolo en términos porcentuales.
- $L(1 + \alpha)$ es el monto total que debe ser devuelto.

Si esta desigualdad no se satisface, la propiedad de atomicidad implica que la ejecución se revierte completamente, lo que puede expresarse como:

$$S_f = S_0 \quad (2.13)$$

Esto significa que el estado global, incluyendo reservas, balances y variables internas, permanece exactamente igual que antes de iniciar la transacción.

Desde el punto de vista económico, el beneficio neto de la operación puede representarse como:

$$\Pi = V_{out} - L - \alpha L - C_{gas} \quad (2.14)$$

donde:

- V_{out} es el valor obtenido tras ejecutar la estrategia.
- L es el capital que debe devolverse
- αL es la comisión pagada al protocolo.
- C_{gas} representa los costos transaccionales.
- Π denota la ganancia de la operación.

La operación solo se materializa si $\Pi > 0$; de lo contrario, la transacción se revierte automáticamente. El modelo económico del protocolo se sostiene gracias a la comisión αL , que compensa a los proveedores de liquidez por permitir el uso temporal de sus fondos. A diferencia de los préstamos tradicionales, donde la tasa de interés remunera el riesgo crediticio y el paso del tiempo, en el préstamo *flash* la comisión remunera exclusivamente el acceso instantáneo a liquidez y el uso de infraestructura computacional segura. Esta característica convierte al protocolo en un intermediario algorítmico cuya función principal no es evaluar solvencia, sino garantizar la correcta ejecución de la restricción atómica.

2.5.4 *Flash loans* como herramienta de arbitraje

Uno de los usos más relevantes y teóricamente justificables de los préstamos *flash* es el arbitraje entre intercambios descentralizados. En mercados automatizados basados en *Automated Market Makers (AMM)*, los precios pueden divergir temporalmente entre diferentes plataformas debido a variaciones en liquidez, volumen o retrasos en el ajuste del estado interno del contrato. En el contexto

de *DeFi*, de acuerdo con [57], el arbitraje se identifica como un mecanismo corrector que contribuye a la convergencia de precios entre mercados.

El préstamo *flash* reduce la barrera de entrada al arbitraje al eliminar la necesidad de capital inicial significativo. Un agente puede solicitar liquidez instantánea, comprar un activo en un *DEX* donde se encuentra subvaluado, venderlo en otro donde su precio es mayor y devolver el préstamo dentro de la misma transacción, capturando el diferencial neto tras comisiones y costos de gas. En [8] documentan que el arbitraje constituye uno de los principales casos de uso observados empíricamente en transacciones de *flash loans*.

No obstante, la misma capacidad de movilizar grandes volúmenes de liquidez de manera instantánea ha sido utilizada en ataques que combinan manipulación temporal de precios y arbitraje adversarial. De acuerdo con el estudio [60] sobre eventos de ataques con *flash loans*, se muestra cómo la ejecución atómica permite encadenar manipulación de mercado y extracción de valor dentro de un solo bloque. Esta evidencia empírica refuerza la necesidad de comprender los préstamos *flash* no solo como herramientas de eficiencia de mercado, sino también como instrumentos que amplifican vulnerabilidades de diseño cuando existen dependencias frágiles en mecanismos de fijación de precios.

2.6 Riesgos en las operaciones de arbitraje con *Flash Loans*

El uso de préstamos *flash* como herramienta de arbitraje introduce un conjunto particular de riesgos técnicos y económicos que deben analizarse desde una perspectiva sistémica. Aunque la atomicidad elimina el riesgo de impago dentro de la transacción, no elimina los riesgos asociados al entorno competitivo, a la dinámica de precios ni a la infraestructura de ejecución.

2.6.1 Riesgo de ejecución y variación de precio

El riesgo de ejecución surge del intervalo temporal entre el cálculo de la oportunidad de arbitraje y la inclusión efectiva de la transacción en un bloque. En redes como *Ethereum*, las transacciones primero ingresan en la *mempool*²⁹, donde permanecen visibles hasta que un validador las incorpora

²⁹ ***Mempool (memory pool)***: es el conjunto de transacciones pendientes que han sido difundidas en la red pero aún no han sido incluidas en un bloque. En *Ethereum*, estas transacciones son públicamente visibles antes de

en un bloque. Durante ese lapso, los precios en los *AMM* pueden cambiar debido a otras operaciones, lo que podría alterar la rentabilidad esperada.

Desde una perspectiva matemática, si el beneficio esperado al momento de detectar la oportunidad es:

$$\Pi_0 = V_{out}(L) - L(1 + \alpha) - C_{gas} \quad (2.15)$$

donde:

- $V_{out}(L)$ es el valor obtenido al ejecutar la operación de arbitraje, el cual depende del tamaño del préstamo debido al *impacto* de precio.
- L es el capital prestado.
- α es la comisión proporcional cobrada por el protocolo de préstamo.
- C_{gas} representa los costos de ejecución en la red.

El término $V_{out}(L)$ no crece linealmente con L , ya que operaciones de mayor tamaño generan mayor deslizamiento en el *AMM*, reduciendo progresivamente el margen de arbitraje. Por ello, existe típicamente un valor óptimo L^* que maximiza el beneficio antes de que el *impacto* de precio anule la oportunidad. Este análisis corresponde a una optimización estática bajo condiciones ideales de ejecución inmediata.

Sin embargo, en la práctica la ejecución no es instantánea. Si la oportunidad se detecta en el instante t , la inclusión efectiva puede ocurrir en $t + \Delta t$. El beneficio real pasa entonces a ser:

$$\Pi_{real} = V_{out}(t + \Delta t) - L(1 + \alpha) - C_{gas} \quad (2.16)$$

Aquí, $V_{out}(t + \Delta t)$ puede diferir del valor estimado inicialmente debido a operaciones de terceros, reordenamiento de transacciones o competencia por la misma oportunidad.

Si $\Pi_{real} \leq 0$, la transacción puede revertirse (si el contrato impone una condición mínima de retorno) o ejecutarse con pérdida. Este fenómeno está estrechamente relacionado con la competencia por

su confirmación, lo que permite su análisis y posible reordenamiento por parte de validadores o *bots* especializados

oportunidades de arbitraje y con la extracción de valor por reordenamiento de transacciones, analizado en [61].

2.6.2 Riesgo de liquidez insuficiente

El arbitraje financiado con *flash loans* depende de la profundidad de liquidez de los *pools* involucrados. En *AMM* del tipo producto constante $x \cdot y = k$, el *impacto* de precio (*slippage*) crece de manera no lineal con el tamaño de la operación. En [62], se demuestra que el precio marginal en estos mercados está dado por:

$$P = \frac{y}{x} \quad (2.17)$$

y que grandes variaciones en x inducen movimientos significativos en el precio implícito.

Por lo tanto, el tamaño óptimo del préstamo *flash* no es arbitrario, sino que debe maximizar:

$$\max_L \Pi(L) \quad (2.18)$$

sujeto a restricciones de liquidez del *pool* y costos de gas. Un tamaño excesivo puede eliminar la propia oportunidad de arbitraje debido al *slippage* inducido por la operación.

2.6.3 Riesgo de fallos en contratos inteligentes y dependencias externas

Los préstamos *flash* dependen de múltiples contratos interconectados: protocolo de préstamo, *DEX*, oráculos y eventualmente agregadores. Cualquier vulnerabilidad lógica puede amplificarse por la capacidad de movilizar grandes cantidades de capital en una sola transacción.

De acuerdo con [63] se han documentado múltiples ataques en los que los préstamos *flash* fueron utilizados para explotar vulnerabilidades en mecanismos de fijación de precios, oráculos y errores lógicos en contratos inteligentes. Estos estudios muestran que el riesgo no se origina en el préstamo *flash* como instrumento financiero en sí mismo, sino en la interacción basada en la composición de múltiples contratos inteligentes ejecutados de forma atómica. La capacidad de movilizar grandes volúmenes de capital dentro de una única transacción amplifica cualquier debilidad preexistente en el diseño del protocolo afectado.

2.6.4 Vulnerabilidades comunes: Ataques de reentrada (*reentrancy*)

La arquitectura de ejecución atómica y la interactividad inherente de los contratos inteligentes en la Máquina Virtual de Ethereum (*EVM*) introducen vectores de ataque críticos a nivel de código, siendo la reentrada (*reentrancy*) uno de los más documentados y destructivos. De acuerdo con lo expuesto por [64], un ataque de reentrada ocurre cuando un contrato vulnerable realiza una llamada externa a un contrato no confiable antes de actualizar su propio estado interno (como el balance de una cuenta). Esta secuencia lógica defectuosa permite que el contrato malicioso interrumpa el flujo de ejecución original y realice llamadas recurrentes (al contrato vulnerable) para extraer fondos repetidamente, aprovechando que el sistema aún no ha registrado la salida del capital inicial. Esta vulnerabilidad compromete la seguridad al interactuar entre protocolos, demostrando que el fallo no proviene de la red *blockchain*, sino de la lógica interna del propio contrato.

En el contexto específico de las Finanzas Descentralizadas (*DeFi*), la introducción de los préstamos relámpago ha amplificado exponencialmente la gravedad de los ataques de reentrada. Tradicionalmente, la capacidad de un atacante para drenar un protocolo vulnerable estaba limitada por su capital inicial. Sin embargo, tal como lo plantea [7], los préstamos *flash* eliminan esta barrera financiera, otorgando a los actores maliciosos acceso instantáneo a una liquidez masiva dentro de una sola transacción. Al inyectar este capital prestado en un contrato con vulnerabilidades de reentrada, el atacante puede multiplicar el impacto de la extracción de valor de forma ilimitada durante la fase de *callback*³⁰, antes de devolver el préstamo a la plataforma de origen. Esto convierte a los *flash loans* en un catalizador que transforma errores de código menores en colapsos financieros sistémicos. Para mitigar este riesgo sistémico en el diseño de protocolos automatizados de arbitraje, la ingeniería de software seguro en *blockchain* exige la implementación de arquitecturas defensivas estrictas. Como ha sido documentado por [27], la prevención algorítmica de la reentrada se fundamenta en la adopción del patrón de diseño *Checks-Effects-Interactions*³¹ (Validaciones-Efectos-Interacciones). Este estándar obliga al desarrollador a verificar primero todas las condiciones lógicas, actualizar

³⁰ **Callback (Llamada de retorno):** Función ejecutable que un sistema invoca automáticamente después de completar una acción previa. En el contexto de préstamos *flash*, es la función que el protocolo prestamista (ej. Aave) llama dentro del contrato del usuario para entregar temporalmente los fondos y ceder el control de la ejecución.

³¹ **Checks-Effects-Interactions (Validaciones-Efectos-Interacciones):** Patrón de diseño de seguridad en contratos inteligentes que establece un orden estricto de ejecución para evitar vulnerabilidades: primero se validan las condiciones (*Checks*), luego se actualiza el estado interno del contrato (*Effects*) y, únicamente como paso final, se interactúa con contratos externos (*Interactions*).

posteriormente el estado interno del contrato y, solo como paso final absoluto, transferir el capital o interactuar con contratos externos. Adicionalmente, se vuelve imperativo el uso de bloqueos de exclusión mutua (*mutex*³²), como los modificadores de estado `nonReentrant`³³, los cuales imponen una restricción a nivel de compilación que impide matemáticamente que una función crítica vuelva a ser invocada mientras su ejecución original siga en curso.

2.6.5 MEV, front-running y competencia estratégica

Uno de los riesgos más relevantes en estrategias de arbitraje es el fenómeno conocido como *Maximal Extractable Value (MEV)*, definido como el valor adicional que un minero o validador puede extraer al reordenar, insertar o censurar transacciones dentro de un bloque. Este concepto fue formalizado por [61] y demuestran que la estructura pública de la *mempool* y la discrecionalidad en el orden de inclusión generan incentivos económicos para capturar oportunidades detectadas por otros participantes.

En este contexto, el arbitraje deja de ser únicamente un problema de identificación de ineficiencias de precio y se convierte también en un problema de competencia estratégica por prioridad de ejecución. La visibilidad pública de las transacciones pendientes permite que terceros reproduzcan o modifiquen estrategias detectadas antes de que sean confirmadas en la cadena.

De esta dinámica emergen prácticas específicas de extracción de valor:

- **Front-running:** ocurre cuando un agente replica una transacción detectada en la *mempool* ofreciendo una comisión mayor para que sea incluida primero en el bloque, apropiándose así de la oportunidad de arbitraje.
- **Back-running:** consiste en posicionar una transacción inmediatamente después de otra para capturar el efecto de precio que esta genera en un *AMM*.

³² **Mutex (Exclusión mutua):** Mecanismo de control de concurrencia utilizado en ciencias computacionales para evitar que múltiples procesos, o llamadas recursivas, en el caso de *blockchain*, accedan simultáneamente a una sección crítica del código o modifiquen un estado compartido al mismo tiempo.

³³ **nonReentrant:** Modificador de código específico del lenguaje Solidity (estandarizado por la librería OpenZeppelin) que implementa un *mutex* a nivel de contrato. Funciona como un candado criptográfico que impide que una función sea invocada nuevamente antes de que su primera ejecución haya concluido por completo.

- ***Sandwich attacks***: combinan ambas estrategias: el atacante ejecuta una operación antes y otra después de la transacción de la víctima, alterando temporalmente el precio para extraer beneficio del deslizamiento inducido.

Estos mecanismos no solo reducen el beneficio esperado de una estrategia de arbitraje, sino que introducen una dimensión adicional de incertidumbre. El rendimiento ya no depende exclusivamente de la discrepancia de precios entre mercados, sino también de la probabilidad de mantener prioridad en la inclusión del bloque. En consecuencia, el beneficio esperado debe considerarse ajustado por riesgo de competencia estratégica, lo que transforma el arbitraje *on-chain* en un entorno cercano a una subasta por ordenamiento de transacciones.

2.6.6 *Flashbots* y rutas privadas de transacción

Como respuesta al problema del *MEV* en entornos de *mempool* pública, se han desarrollado mecanismos de envío privado de transacciones, entre los cuales destaca *Flashbots*. Este sistema permite a los participantes enviar *bundles*³⁴ (paquetes agrupados de transacciones) directamente a validadores, evitando su exposición previa en la *mempool* pública y reduciendo así la probabilidad de que terceros repliquen o reordenen la operación mediante estrategias de *front-running*.

En lugar de difundir una transacción individual al conjunto de nodos de la red, *Flashbots* introduce una arquitectura en la que los denominados *searchers* envían paquetes cerrados que contienen una secuencia ordenada de transacciones. Estos paquetes se ejecutan de forma atómica únicamente si cumplen condiciones de rentabilidad para el validador que construye el bloque. En este esquema, la prioridad ya no depende exclusivamente del gas ofrecido en una *mempool* abierta, sino de acuerdos directos y de la rentabilidad esperada del *bundle* completo.

Desde una perspectiva teórica, este mecanismo no elimina el *MEV*, sino que transforma su estructura de mercado. El entorno competitivo deja de ser una subasta abierta basada en visibilidad pública y pasa a configurarse como un mercado de subastas privadas por oportunidades de extracción de valor. La incertidumbre asociada al reordenamiento abierto disminuye, pero se sustituye por un modelo donde la competencia ocurre a través de ofertas privadas dirigidas a validadores.

³⁴ ***Bundle (Paquete de transacciones)***: un *bundle* es una agrupación secuencial de transacciones enviada directamente a los constructores de bloques, eludiendo la *mempool* pública. Este mecanismo garantiza que el grupo de transacciones se ejecute de forma completamente atómica y en un orden criptográficamente estricto, protegiendo las operaciones de arbitraje contra ataques de *front-running* o *sandwich*.

En [61] analizan empíricamente la dinámica del *MEV* y muestran cómo la visibilidad de la *mempool* genera incentivos para el reordenamiento estratégico de transacciones. La aparición de mecanismos como *Flashbots* puede interpretarse como una respuesta institucional emergente a estas ineficiencias, al buscar canalizar la extracción de valor hacia un entorno más predecible y menos disruptivo para los participantes minoristas.

2.6.7 Riesgos de Infraestructura Multi-Cadena y Vulnerabilidad en Puentes

La expansión de las estrategias de arbitraje hacia un ecosistema multi-cadena exige la utilización de puentes (*bridges*) para la transferencia de liquidez y datos entre redes heterogéneas. Sin embargo, esta infraestructura introduce vectores de ataque críticos que rompen con los paradigmas de seguridad de un entorno aislado. De acuerdo con lo expuesto por [65], los contratos inteligentes de los puentes *cross-chain* sufren frecuentemente de vulnerabilidades semánticas e inconsistencias lógicas en el control de acceso a los activos, denominadas “Vulnerabilidades *Cross-Chain (CCVs)*”. La conexión de blockchains independientes carece de un registro de emparejamiento unificado, lo que dificulta el rastreo riguroso de las transacciones y convierte a los puentes en el eslabón más susceptible de la arquitectura. Estas vulnerabilidades han provocado pérdidas financieras masivas en ecosistemas descentralizados al no poder garantizar la equivalencia de estados entre las dos redes conectadas.

Más allá de las fallas de código a nivel de contrato inteligente, el arbitraje en plataformas multi-cadena se enfrenta al grave problema estructural de la ejecución no atómica. A diferencia de las transacciones dentro de una sola cadena, donde el préstamo *flash* y los múltiples intercambios (*swaps*) ocurren en un mismo bloque bajo una estricta garantía computacional de "todo o nada", la interoperabilidad exige confiar en el tiempo de procesamiento de validadores externos. Tal como lo plantea [66], la naturaleza no atómica de estas operaciones complica enormemente tanto su ejecución como su viabilidad económica; los tiempos de bloque asíncronos y la exposición al tiempo de transferencia del puente (*bridging time*) son factores determinantes. Esta latencia estructural expone el capital del arbitrajista a fluctuaciones naturales del mercado, depreciación de los tokens durante el tránsito y demoras de validación, riesgos que pueden anular o volver negativa la rentabilidad proyectada al llegar a la cadena de destino.

Finalmente, esta asincronía temporal y la exposición pública de los mensajes de transferencia entre cadenas dan origen a una de las amenazas competitivas más sofisticadas de las Finanzas Descentralizadas: el *Maximal Extractable Value (MEV)* en dominios cruzados. Cuando un protocolo emite información sobre un intercambio intercadena inminente, los algoritmos adversarios

monitorean activamente estos eventos desde la *mempool* de la red de origen. En [36], se evidencia que la transparencia de estos mensajes crea una asimetría de información que permite a los atacantes colocar transacciones anticipadas (*front-running*) y de cierre (*back-running*) directamente en la red de destino. Al ejecutar estos ataques de sándwich *cross-chain* antes de que la transacción legítima de arbitraje logre confirmarse, los adversarios se apropian del valor generado por la discrepancia de precios, destruyendo la rentabilidad del operador original e ilustrando el profundo riesgo sistémico de operar algorítmicamente a través de redes fragmentadas.

2.7 Oráculos de precios

Los contratos inteligentes no pueden acceder directamente a información externa a la *blockchain*. Esta limitación, conocida como el problema del “*oracle*”, surge porque la ejecución debe ser determinista y verificable por todos los nodos de la red. Como consecuencia, cualquier dato externo (*off-chain*), como por ejemplo el precio de un activo en mercados globales, debe introducirse mediante mecanismos especializados denominados oráculos. Estos sistemas actúan como intermediarios que transmiten información del mundo real hacia la *blockchain*, permitiendo que los contratos inteligentes operen con datos actualizados y verificables.

La necesidad de oráculos es particularmente crítica en aplicaciones *DeFi*, donde la correcta determinación del precio de los activos condiciona decisiones de liquidación, colateralización, arbitraje y cálculo de rentabilidad. Como ha sido analizado en [61], sin un mecanismo confiable de referencia de precios, los protocolos quedan expuestos al riesgo de manipulaciones que pueden comprometer su estabilidad económica.

2.7.1 Tipos de oráculos: centralizados, federados y descentralizados

Los oráculos pueden clasificarse según quién controla la fuente de datos y cómo se valida la información antes de incorporarla a la *blockchain*. Esta diferencia no es solo organizativa; *impacta* directamente en la seguridad económica del protocolo que depende de esos precios.

2.7.1.1 Oráculos centralizados

Son operados por una única entidad que introduce datos externos en la red. En este modelo, el sistema confía en que ese proveedor actuará correctamente y no será comprometido. Aunque esta arquitectura puede ser más simple y rápida, introduce un punto único de riesgo. Si la fuente falla, es

manipulada o actúa de manera maliciosa, el protocolo completo puede verse afectado. Tal como lo plantea [67], los sistemas descentralizados buscan precisamente eliminar la dependencia de terceros confiables. Cuando un sistema financiero descentralizado depende de un solo proveedor de precios, se reintroduce precisamente ese tipo de dependencia.

2.7.1.2 Oráculos federados

Representan un punto intermedio. En este esquema, varias entidades autorizadas reportan el precio de un activo, y el sistema utiliza un mecanismo de agregación (por ejemplo, la mediana) para reducir el *impacto* de valores atípicos o intentos individuales de manipulación. Según lo expuesto en [68], los sistemas distribuidos que intercambian información deben incorporar mecanismos de verificación cruzada para preservar la integridad de los datos. En este sentido, el modelo federado mejora la resiliencia frente a errores individuales, aunque sigue dependiendo de un conjunto limitado de actores.

2.7.1.3 Oráculos descentralizados

Amplían aún más la distribución de la información. En este modelo, múltiples participantes independientes reportan datos externos y son incentivados económicamente para actuar de manera honesta. El sistema combina esas múltiples fuentes mediante métodos estadísticos, lo que dificulta que un solo actor pueda alterar el resultado final. De acuerdo con lo expuesto por [69], la descentralización en los mecanismos de precios reduce el riesgo de que un error o manipulación individual comprometa la estabilidad del protocolo.

2.7.1.4 Oráculos Cross-Chain (Sincronización global)

Si bien los oráculos descentralizados resuelven el problema de la confianza dentro de una red aislada, la fragmentación de la liquidez en el ecosistema moderno introduce un nuevo desafío computacional: la interoperabilidad y la sincronización de estado entre múltiples cadenas independientes. Cuando un activo financiero cotiza simultáneamente en redes heterogéneas, un oráculo local carece de la visión global necesaria, lo que da origen a la necesidad teórica de los oráculos *cross-chain* (inter-cadenas). Este modelo representa el nivel más complejo de infraestructura de datos, ya que su función no se limita a inyectar información externa, sino a lograr un consenso de precios transmitiendo mensajes criptográficos entre máquinas virtuales que operan con distintas

latencias y tiempos de finalidad (*finality*³⁵). Tal como lo formalizan [7], los oráculos *cross-chain* son un componente crítico para asegurar la integridad de los datos en entornos con alta latencia de coordinación, mitigando los riesgos inherentes de los puentes tradicionales. Asimismo, la falta de una sincronización global rigurosa expone a los protocolos a vectores de ataque temporal, permitiendo que actores maliciosos exploten la asincronía de la red para extraer valor mediante arbitraje de latencia en transacciones multi-cadena, un riesgo sistémico documentado por [36]. Por lo tanto, para un sistema algorítmico concebido bajo una arquitectura de múltiples cadenas, la integración de oráculos *cross-chain* se convierte en el requisito computacional indispensable para calcular la rentabilidad neta global antes de comprometer capital a través de puentes interoperables

2.7.2 Manipulación de oráculos, préstamos *flash* y arbitraje

Uno de los riesgos más relevantes en *DeFi* es la posibilidad de manipular temporalmente el precio de un activo para explotar protocolos que dependen de ese valor como referencia. Este fenómeno se ha vuelto especialmente significativo con la aparición de los préstamos *flash*. En *DeFi*, un precio no es solo un número informativo: es una señal que activa decisiones automáticas (por ejemplo, cuánto colateral se exige, cuándo se liquida una posición o si una operación es rentable). Por eso, cuando el “precio de referencia” puede ser alterado aunque sea por instantes, el protocolo queda expuesto a que alguien fuerce decisiones equivocadas y extraiga valor. Como se ha demostrado en [61], la visibilidad pública de transacciones y el reordenamiento estratégico dentro de un bloque permiten capturar oportunidades económicas a partir de estados temporales del mercado.

Los préstamos *flash*, al permitir acceder a grandes cantidades de capital sin necesidad de colateral por un momento dentro de una transacción atómica, amplifican ataques donde un actor altera momentáneamente el precio de un activo en un *exchange* descentralizado, ejecuta operaciones que dependen de ese precio inflado o reducido, y restablece el equilibrio antes de finalizar el bloque.

Un punto importante que se evidencia en [70] es que muchos ataques no implican vulnerar la criptografía ni el mecanismo de consenso de la red. En lugar de ello, explotan la lógica económica del protocolo. Basta con alterar temporalmente un precio *on-chain*. Por ejemplo la alteración de

³⁵ **Finality (Tiempo de finalidad):** Garantía criptográfica que asegura la irreversibilidad absoluta de una transacción confirmada en una *blockchain*, siendo una condición técnica indispensable en arquitecturas *cross-chain* para que un protocolo valide y transmita un estado hacia otra red sin riesgo de que la operación original sea revertida.

precios en un *AMM* con baja liquidez para que otro contrato que utiliza ese valor como referencia tome decisiones incorrectas, como liquidar posiciones, emitir deuda o valorar colateral de forma errónea. El atacante puede completar el ciclo dentro del mismo bloque: obtiene un préstamo *flash*, desplaza el precio, ejecuta la operación ventajosa y repaga el préstamo, capturando la diferencia.

2.7.3 El oráculo como mecanismo de referencia para el cálculo de rentabilidad

En un sistema automatizado de arbitraje financiado mediante préstamos *flash*, el cálculo previo de rentabilidad constituye el núcleo de la decisión de ejecución. Antes de solicitar el capital y enviar la transacción a la red, el sistema debe estimar si la discrepancia de precios observada es suficiente para cubrir comisiones, costos de gas y posibles variaciones en el estado del mercado. Este cálculo depende directamente de la calidad del precio utilizado como referencia.

En entornos *DeFi* abiertos y competitivos, el precio no es una variable estática ni necesariamente neutral. Como se ha mencionado en [61], la visibilidad pública de las transacciones y la posibilidad de reordenamiento generan incentivos para explotar estados transitorios del mercado. En este contexto, el precio observado en un instante determinado puede reflejar no solo condiciones genuinas de oferta y demanda, sino también estrategias oportunistas de otros participantes.

Para un sistema automatizado, esto implica un riesgo concreto: si el algoritmo identifica una oportunidad basándose en un precio momentáneamente distorsionado, puede ejecutar una operación que, una vez incluida en bloque, resulte en una operación con pérdidas. Cuando el precio dispara la señal, se convierte en un punto crítico de vulnerabilidad. Tal como se ha documentado en [7], las variaciones temporales inducidas estratégicamente pueden activar decisiones automatizadas en otros protocolos que confían en ese valor como referencia.

En este entorno competitivo, el oráculo cumple una función fundamental: actuar como mecanismo de validación externa del precio antes de la ejecución. No se trata únicamente de obtener un número actualizado, sino de contar con una referencia que reduzca la probabilidad de basarse en señales transitorias o manipulables. Una referencia robusta, por ejemplo, aquella que agrega múltiples fuentes o considera información a lo largo de un intervalo temporal, introduce mayor estabilidad en el cálculo de rentabilidad.

Desde la perspectiva del sistema propuesto, la importancia de esta validación es directa. El préstamo *flash* implica movilizar capital significativo dentro de una sola transacción. Si el cálculo de beneficio esperado se fundamenta en una señal inestable, el riesgo de ejecutar oportunidades falsas aumenta considerablemente. En cambio, cuando el precio utilizado como referencia ha sido previamente

validado mediante un mecanismo más resistente a fluctuaciones momentáneas, la probabilidad de errores inducidos por competencia estratégica disminuye.

En consecuencia, el oráculo no debe entenderse como un componente accesorio de la infraestructura, sino como un elemento estructural en la confiabilidad del proceso de detección. En un entorno donde el *MEV*, el *front-running* y la competencia por prioridad de inclusión forman parte de la dinámica normal del mercado, la calidad del precio de referencia condiciona directamente la robustez del sistema automatizado de arbitraje. Una referencia de precios sólida no elimina el riesgo, pero reduce significativamente la exposición a decisiones basadas en distorsiones temporales, fortaleciendo así la coherencia entre la detección teórica de la oportunidad y su ejecución efectiva en la cadena.

CAPÍTULO 3: DESARROLLO DEL PROYECTO

3.1 Introducción

El presente capítulo documenta el desarrollo completo de un sistema de arbitraje automatizado sobre la red *Ethereum*, que aprovecha los préstamos relámpago (*flash loans*) provistos por el protocolo Aave V3 para ejecutar operaciones de compraventa entre múltiples intermediarios descentralizados (*DEXes*) dentro de una sola transacción atómica. El propósito fundamental del sistema es identificar, simular y ejecutar oportunidades de arbitraje que sean rentables una vez descontadas las comisiones por transacción (gas fees), primas del préstamo y posibles desviaciones de precios (*slippage*).

La arquitectura está diseñada para operar de forma modular y segura, integrando contratos inteligentes escritos en lenguaje Solidity y scripts de automatización fuera de la cadena de bloques (*off-chain*) implementados en *TypeScript*. Estos últimos se encargan de monitorear el mercado, calcular rutas óptimas, validar condiciones del entorno y ejecutar operaciones en tiempo real.

El ecosistema de las finanzas descentralizadas (*DeFi*) se caracteriza por su dinamismo y fragmentación de la liquidez. Esta fragmentación, distribuida entre decenas de *DEXes* con diferentes modelos de precios, crea constantes e inevitables ineficiencias de mercado. Estas discrepancias de precios para un mismo activo, aunque a menudo efímeras, presentan oportunidades de arbitraje. Sin embargo, capitalizarlas enfrenta tres desafíos principales: la necesidad de un capital inicial significativo, la volatilidad extrema de las comisiones de la red (*gas*) que puede anular las ganancias, y el riesgo de ser víctima de estrategias de valor máximo extraíble (*MEV*) como el "*front-running*". Este proyecto aborda directamente estos desafíos mediante una solución integral: el uso de *flashloans* para eliminar la barrera del capital, un monitoreo constante de las tarifas de gas para asegurar la rentabilidad, y la integración con *Flashbots* para la ejecución privada y eficiente de las transacciones. La columna vertebral del sistema reside en dos contratos inteligentes principales. El primero, `FlashLoanArbitrage.sol`, actúa como el orquestador central. Es responsable de solicitar el *flashloan* a AAVE, recibir los fondos, y ejecutar la secuencia de intercambios. Contiene la lógica para verificar la rentabilidad final y devolver el préstamo más la prima correspondiente, todo dentro de una única transacción. El segundo contrato, `DexAggregator.sol`, funciona como un enrutador universal. Abstrae la complejidad de interactuar con más de 27 *DEXes* diferentes, cada uno con su propia interfaz y estructura. Este agregador expone una interfaz unificada que permite al contrato

principal solicitar cotizaciones y ejecutar intercambios a través de la ruta más favorable, maximizando así el potencial de ganancia en cada operación.

Complementando la lógica en la cadena (*on-chain*), un conjunto de scripts en *TypeScript* gestiona las operaciones fuera de la cadena (*off-chain*). El script `arbitrage.ts` es el cerebro del sistema, encargado de consultar datos de mercado en tiempo real a través de la *API* de *GeckoTerminal*, identificar pares de tokens y rutas de intercambio prometedoras, y simular la rentabilidad de las operaciones. Para ello, se apoya en otros scripts como `liquidity.ts` y `gasFee.ts`, que recopilan y almacenan datos actualizados sobre la liquidez de los pools y las tarifas de gas de la red. Una vez que se detecta una oportunidad viable, el script construye y envía la transacción al contrato inteligente, utilizando el proveedor de *Flashbots* para protegerla de la vista pública y optimizar su inclusión en un bloque.

Finalmente, el proyecto se estructura sobre un entorno de desarrollo robusto basado en *Hardhat*, que facilita la compilación, prueba y despliegue de los contratos inteligentes. La configuración `hardhat.config.ts` está preparada para operar en una bifurcación (*fork*) de la red principal de *Ethereum*, permitiendo realizar pruebas realistas sin incurrir en costos reales. La gestión de datos sensibles, como claves privadas y de *API*, se maneja de forma segura a través de variables de entorno. En conjunto, estos componentes conforman un sistema completo y sofisticado, diseñado no solo para ejecutar arbitraje de manera efectiva, sino también para servir como un caso de estudio avanzado sobre la interacción entre sistemas *on-chain* y *off-chain* en el competitivo entorno de *DeFi*.

3.2 Entorno de Desarrollo y Estructura del Proyecto

3.2.1 Lenguajes Utilizados

La elección de lenguajes de programación en este proyecto se basó en criterios de compatibilidad, eficiencia y escalabilidad:

- **Solidity:** fue utilizado para la programación de los contratos inteligentes. Este lenguaje es el estándar para el desarrollo sobre la *Ethereum Virtual Machine (EVM)* y permite un control detallado del flujo de ejecución, acceso a funciones del sistema, protección contra vulnerabilidades y uso de librerías auditadas. Gracias a la comunidad de *OpenZeppelin* y su extenso conjunto de contratos seguros, Solidity ofreció un entorno robusto para codificar la lógica crítica del sistema, incluyendo la gestión de *flash loans*, control de *swaps*, validaciones y control de acceso.

- **TypeScript:** por su parte, fue la tecnología elegida para la parte *off-chain*, es decir, para todos los scripts encargados de desplegar contratos, simular operaciones, ejecutar swaps, estimar rentabilidad y validar condiciones previas a la transacción. A diferencia de *JavaScript*, este lenguaje proporciona tipado estático, verificación en tiempo de desarrollo y una integración fluida con *frameworks* modernos como *Hardhat*, *Ethers.js*, *dotenv*. Esta elección facilita el desarrollo colaborativo, reduce errores lógicos y asegura consistencia en la ejecución de scripts automatizados.

En conjunto, la sinergia entre Solidity (*on-chain*) y TypeScript (*off-chain*) permitió un desarrollo eficiente, estructurado y preparado para escalar tanto en funcionalidades como en entornos de ejecución.

3.2.2 Fundamentos del Framework Hardhat

El sistema fue construido y desplegado utilizando el entorno de desarrollo **Hardhat**, una de las herramientas más robustas en el ecosistema *Ethereum* para compilar, probar, y desplegar contratos inteligentes.

3.2.2.1 Framework utilizado

- **hardhat** es un *framework* de desarrollo completo para *Ethereum*. Aunque se usa principalmente durante el desarrollo, sus *plugins* y su integración con *ethers.js* son esenciales para los scripts:
 - **@nomiclabs/hardhat-ethers:** Integra *ethers.js* con Hardhat, facilitando la obtención de cuentas y la conexión a la red configurada.
 - **@typechain/hardhat:** Genera automáticamente tipos de *TypeScript* a partir de los contratos Solidity, lo que ayuda a evitar errores al interactuar con los contratos desde los scripts.

Hardhat actúa como el "sistema operativo" del proyecto. Compila los contratos Solidity (versiones 0.5.5 hasta 0.8.28), genera un *fork* local de Ethereum Mainnet para pruebas sin recurrir a *gas* real, y gestiona el despliegue a Sepolia testnet y Mainnet. Todos los scripts corren dentro del entorno de Hardhat.

- **Configuración de Hardhat**

La configuración fue realizada en el script `hardhat.config.ts`

```
import { HardhatUserConfig } from "hardhat/config";
import "@nomiclabs/hardhat-ethers";
import "@typechain/hardhat";
// Otras importaciones...
```

Código 3.1 Importaciones principales del archivo de configuración `hardhat.config.ts` escrito en lenguaje typescript.

El *framework* de Hardhat permite:

- Simular *Mainnet* con mainnet forking
- Ejecutar scripts de despliegue mediante *run* y *deploy*
- Interactuar con contratos ya desplegados desde scripts *TypeScript*
- Usar hardhat-ethers y nomicfoundations/hardat-chai-matchers para pruebas realistas

La elección de Hardhat fue estratégica debido a su compatibilidad con herramientas modernas como **Ethers.js**, su capacidad de integrar oráculos y *flash loans*, y su estructura modular.

3.2.3 Plataformas, Herramientas y *Frameworks*

Para la implementación del sistema de arbitraje automatizado, fue indispensable seleccionar un conjunto de tecnologías que permitiera construir de manera sincronizada tanto la lógica central dentro de la cadena de bloques (*on-chain*) como los procesos externos de control, análisis y monitoreo (*off-chain*). La elección de cada plataforma, herramienta y entorno de trabajo (*framework*) se llevó a cabo evaluando estrictamente su compatibilidad con la Máquina Virtual de Ethereum (*EVM*), su robustez en materia de seguridad y su capacidad para interactuar con datos en tiempo real dentro del ecosistema de las Finanzas Descentralizadas (*DeFi*).

A continuación, en la Tabla 3.1 se presenta el listado detallado de las principales tecnologías integradas durante el desarrollo del proyecto. El propósito de esta tabla es exponer de forma clara la función específica que cada herramienta desempeña dentro de la arquitectura general del sistema. Este ecosistema tecnológico abarca desde los entornos de prueba hasta los proveedores de infraestructura, protocolos de liquidez y oráculos descentralizados necesarios para ejecutar transacciones atómicas y mitigar vulnerabilidades críticas.

Tabla 3.1: Plataformas y Herramientas Utilizadas

Herramienta / Plataforma	Descripción
Hardhat	Entorno de desarrollo y pruebas basado en EVM, configurado con fork de mainnet y compiladores múltiples.
Node.js	Motor de ejecución con tipado estático para mayor seguridad en el desarrollo.
Ethers.js v5	Biblioteca para interactuar con la red <i>Ethereum</i> . Permite conectar a nodos, leer datos de la cadena, enviar transacciones y trabajar con contratos inteligentes.
OpenZeppelin Contracts v4.9	Biblioteca de contratos inteligentes auditados, principalmente de seguridad.
AAVE V3 Protocol	Protocolo de préstamos relámpago.
Chainlink Price Feeds	Oráculos descentralizados para obtener precios en tiempo real
GeckoTerminal API	API externa para obtener datos de liquidez y precios de múltiples DEXes.
Alchemy Provider	Proveedor RPC principal para conexión con <i>Ethereum</i> Mainnet y obtención de datos de gas optimizados.
Flashbots Bundle Provider	Infraestructura <i>MEV</i> para envío de transacciones de alta prioridad a través de <i>bundles</i> privados.
TypeChain	Generador automático de tipos <i>TypeScript</i> para contratos, configurado en el script de configuración de Hardat.
dotenv	Gestión segura de variables de entorno desde .env incluyendo claves privadas y API keys.
Axios	Cliente HTTP para realizar llamadas a APIs externas como GeckoTerminal y CoinGecko.
Etherscan API	Verificación de contratos y descarga automática de ABIs a través de download-abis.ts.

3.2.4 Estructura del Proyecto

El sistema se organizó bajo una arquitectura modular que separa claramente las responsabilidades entre contratos inteligentes, lógica *off-chain*, datos de configuración y herramientas de desarrollo. La estructura del proyecto, almacenado en un repositorio, se representa con el siguiente árbol de

```

— .env          # Claves privadas, API Keys y configuración sensible.
— config/
  — addresses.ts      #Direcciones de contratos, DEXes y tokens de Aave.
— contracts/
  — DexAggregator.sol    #Enrutador de swaps para los 27 DEXes.
```

```

├── FlashLoanArbitrage.sol #Contrato principal de arbitraje con flashloans.
├── data/
│   ├── dexespools.json # Datos de pools y precios por DEX.
│   ├── aave-reserves.json # Datos de tokens de las reservas de Aave V3.
│   └── gasFee.json # Registro histórico de las tarifas gas.
├── external/
│   └── abis/
│       └── mainnet/ #ABIs de contratos de cada DEX.
│           ├── ...
│           ├── PancakeSwap/
│           │   ├── factory.json
│           │   ├── poolmanager.json
│           │   ├── quoter.json
│           │   └── router.json
│           ├── PancakeSwapV2/
│           │   ├── factory.json
│           │   └── router.json
│           ├── SushiSwapV2/
│           │   ├── factory.json
│           │   └── router.json
│           ├── SushiSwapV3/
│           │   ├── factory.json
│           │   └── router.json
│           ├── UniswapV2/
│           │   ├── factory.json
│           │   └── router.json
│           ├── UniswapV3/
│           │   ├── factory.json
│           │   ├── quoter.json
│           │   └── router.json
│           └── # Otros DEXes soportados
├── scripts/
│   ├── arbitrage.ts # Script principal que detecta y ejecuta arbitraje.
│   ├── deploy.ts # Script automatizado para desplegar los contratos.
│   ├── gasFee.ts # Consulta y registro de tarifas de gas.
│   ├── generate-flashbots-key.ts # Generador de claves para Flashbots.
│   ├── get-aave-reserves.ts # Consulta reservas de Aave V3 on-chain.
│   ├── liquidity.ts # Descarga datos de liquidez vía GeckoTerminal.
│   └── setup-dexes.ts # Registra los 27 DEXes en el contrato.
├── download-abis.ts # Descarga ABIs de contratos desde Etherscan.
├── hardhat.config.ts # Configuración de red, compiladores y plugins.
└── package.json # Dependencias y scripts del proyecto Node.js.

```

Gracias a este diseño modular, el sistema puede ser mantenido, probado y extendido con facilidad. Además, se pueden integrar nuevas estrategias de arbitraje o conectar a más *DEXes* simplemente agregando nuevas funciones al contrato inteligente `DexAggregator.sol` y actualizando `dexespools.json` periódicamente.

La organización del proyecto en carpetas clave facilitó la automatización y reutilización de scripts. El proyecto está estructurado en torno a *npm* y *TypeScript*, utilizando Hardhat para el desarrollo en *Ethereum*. La arquitectura se centra en los siguientes componentes principales:

- **Contratos Inteligentes (*Smart contracts*)**

- `FlashLoanArbitrage.sol`: Orquestar la lógica de arbitraje, solicitar un préstamo *flash* a Aave, ejecutar las operaciones y devolver el préstamo en una única transacción atómica.
- `DexAggregator.sol`: Actuar como un enrutador universal para interactuar con múltiples *DEXes*, obteniendo cotizaciones y ejecutando intercambios de manera eficiente.

- **Scripts de Automatización**

- `deploy.ts`: Desplegar los contratos inteligentes en la red *Ethereum*.
- `setup-dexes.ts`: Configurar y registrar las direcciones de los *DEXes* soportados dentro del contrato `DexAggregator.sol`.
- `arbitrage.ts`: Detectar oportunidades de arbitraje, construir y enviar las transacciones (potencialmente a través de *Flashbots*) para su ejecución.
- `liquidity.ts`: Recopilar datos de liquidez de los pools de los *DEXes* a través de la API de GeckoTerminal.
- `gasFee.ts`: Monitorear y almacenar las tarifas de gas actuales e históricas de la red.
- `generate-flashbots-key.ts`: Crear una clave privada segura para firmar *bundles* de *Flashbots*.

Configuración (/config):

- `hardhat.config.ts`: Define la configuración central del entorno de desarrollo, incluyendo redes, versiones del compilador y optimizador.
- `addresses.ts`: Mantiene un registro centralizado de direcciones importantes como las de los contratos desplegados, tokens y oráculos.
- `.env`: Almacena de forma segura información sensible como claves privadas y de API.

Datos (/data)

- `dexespools.json`: Guardar la información detallada sobre los pools de liquidez obtenida por el script `liquidity.ts`.

- `gasFee.json`: Almacenar los datos actuales e históricos de las tarifas de gas para optimizar los costos de transacción.

La estructura del proyecto está diseñada para ser modular y automatizada. Los scripts de *TypeScript* gestionan el ciclo de vida completo: desde la recopilación de datos y el despliegue de contratos hasta la ejecución de estrategias de arbitraje. Los contratos de Solidity contienen la lógica *on-chain*, mientras que los archivos de configuración y datos proporcionan la flexibilidad y la información necesarias para operar en el dinámico ecosistema *DeFi*.

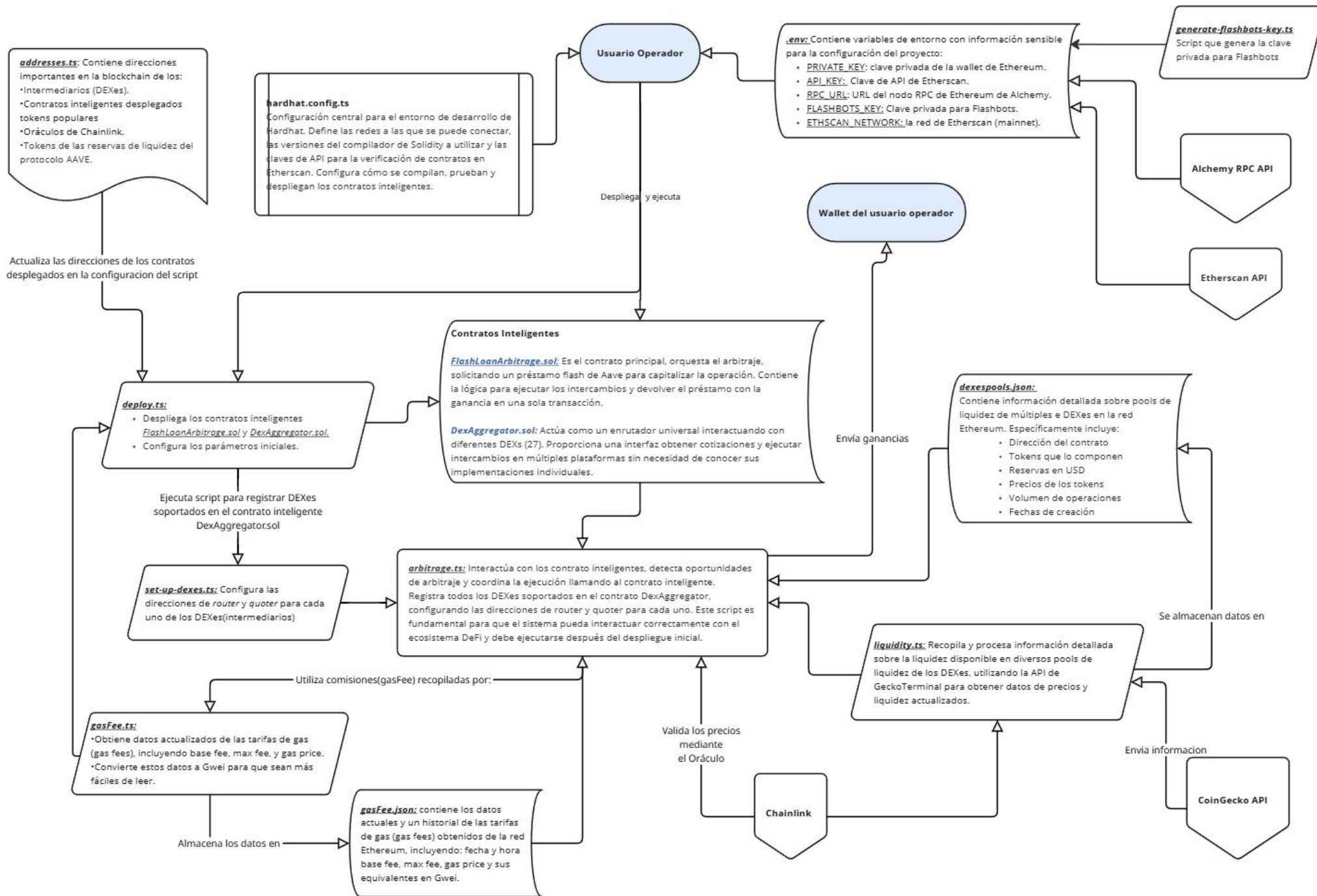


Figura 3.1: Arquitectura general del sistema de arbitraje.

3.2.5 Interfaces Utilizadas

Para interactuar con distintos servicios y *DEXes* del ecosistema *Ethereum*, el sistema implementó interfaces estandarizadas que permiten desacoplar la lógica principal del contrato de los detalles de cada protocolo.

3.2.5.1 Interfaces de *DEXes*

- **Ecosistema Uniswap:**

- **UniswapV2Router02:** Para realizar intercambios de tokens en *DEXes* compatibles con Uniswap V2, calculando previamente la cantidad de salida.
- **ISwapRouter:** Para ejecutar intercambios de un solo paso (*token A* por *token B*) en *DEXes* compatibles con Uniswap V3.
- **IUniswapV4Router:** Para ejecutar intercambios en Uniswap V4, permitiendo lógica personalizada.
- **IQuoterV2** y **IUniswapV4Quoter:** Sirven para obtener cotizaciones de precios sin ejecutar un intercambio real, para estimar costos y resultados.

- **Ecosistema SushiSwap:**

- **IUniswapV2Router02:** Su función es realizar intercambios en SushiSwap V2, ya que comparte la misma arquitectura que Uniswap V2.
- **ISwapRouter (para V3):** Para ejecutar intercambios de un solo paso en SushiSwap V3, que sigue el modelo de Uniswap V3.

- **Ecosistema PancakeSwap:**

- **IPancakeRouter:** Para ejecutar intercambios de un solo paso en PancakeSwap V3, utilizando una estructura similar a la de Uniswap V3.
- **IPancakeQuoter:** Para obtener cotizaciones de precios en PancakeSwap V3, que puede incluir una estimación del gas necesario.

- **Protocolos *DeFi* Avanzados:**

- **IBalancerVault:** Sirve para interactuar con la bóveda central de Balancer para realizar intercambios, que pueden involucrar múltiples tokens en una sola operación.
- **ICurveRouter:** Para encontrar el mejor tipo de cambio y ejecutar intercambios en Curve, que está optimizado para activos con precios similares como las *stablecoins*.
- **ISolidlyRouter:** Para realizar intercambios en *DEXes* que utilizan el modelo de Solidly, obteniendo primero la cantidad de salida esperada.

- **DEXes Especializados:**

- **IKyberClassicRouter:** Para obtener una tasa de cambio esperada y ejecutar un intercambio basado en esa tasa, común en versiones anteriores de Kyber.
- **IKyberElasticRouter:** Para ejecutar intercambios de un solo paso en Kyber Elastic, que funciona de manera similar a Uniswap V3.
- **IMaverickRouter:** Para ejecutar intercambios de un solo paso en Maverick, requiriendo especificar la dirección del pool exacto para la operación.

3.2.5.2 Interfaces de Protocolos

- **AAVE V3**

- **IFlashLoanSimpleReceiver:** Define el estándar que un contrato debe seguir para poder recibir un *flash loan* de Aave, implementando la lógica en la función `executeOperation()`.
- **IPool:** Interactúa con el contrato principal de Aave para solicitar un *flash loan* a través de la función `flashLoanSimple()`, siendo más eficiente en *gas* que `flashLoan()` para un solo activo.
- **IPoolAddressesProvider:** Funciona como un registro para encontrar las direcciones actualizadas de los contratos clave del ecosistema Aave.

- **Chainlink**

- **AggregatorV3Interface:** Obtiene datos de precios fiables y descentralizados desde fuera de la *blockchain*, como el precio de ETH en USD.

3.2.5.3 Arquitectura de interfaces

Todas las interfaces de los distintos *DEXes* están unificadas dentro del contrato inteligente `DexAggregator.sol`. Este contrato define un enumerador llamado `enum DexType` que identifica hasta 27 tipos de *DEXes* diferentes, cada uno asociado a un índice único (del 0 al 26).

```
enum DexType {
    UniswapV2,    // 0
    UniswapV3,    // 1
    UniswapV4,    // 2
    SushiSwapV2,  // 3
    // ... hasta (26)
}
```

Código 3.2: Estructura de enumeración `DexType` en lenguaje Solidity para la clasificación e indexación de interfaces de protocolos de intercambio descentralizado (*DEXes*) implementada en el contrato inteligente `DexAggregator.sol`.

El enumerador organiza el código de tal manera que actúa como un puente entre la lógica *off-chain* (en `arbitrage.ts`) y el contrato *on-chain*. El script de *TypeScript* puede usar estos datos para encontrar el índice correcto y utilizarlos en el contrato.

- **Configuración Dinámica**

El sistema utiliza dos mapas `isV2Like` y `isV3Like` para clasificar automáticamente cada *DEX* según su tipo de interfaz (similar a Uniswap V2 o V3). Esto permite que, en tiempo de ejecución, el contrato sepa qué funciones debe usar para interactuar con cada *DEX*. La función `setupAllDexes()` automatiza la configuración, añadiendo los 27 *DEXes* con sus respectivos *routers* y *quoters* correspondientes.

- **Integración de APIs Externas**

- **GeckoTerminal API:** Utilizada en el script `liquidity.ts` para obtener datos de liquidez en tiempo real de los *DEXes*.
- **Alchemy Provider:** Configurado en `hardhat.config.ts` para conectar de manera eficiente con la red principal de *Ethereum*.
- **Flashbots:** Integrado mediante `@flashbots/ethers-provider-bundle` para enviar transacciones protegidas contra *MEV* (arbitraje de valor extraíble por mineros).

Esta arquitectura facilita la incorporación de nuevos *DEXes*. Solo es necesario agregar un nuevo valor al `enum DexType` y definir la lógica específica en las funciones internas del agregador, sin modificar la estructura general del sistema.

3.2.5.4 Integración Multi-DEX

El contrato inteligente desarrollado `DexAggregator.sol` está diseñado para enrutar swaps entre 27 *DEXes* diferentes de forma independiente a través de `enum DexType`.

```
enum DexType {
  UniswapV2,      // 0 - AMM V2 estándar
  UniswapV3,      // 1 - Concentrated liquidity
  UniswapV4,      // 2 - Hooks y nueva arquitectura
  SushiSwapV2,    // 3 - Fork de Uniswap V2
  SushiSwapV3,    // 4 - V3 con mejoras
  PancakeSwapV2,  // 5 - Multi-chain AMM
  PancakeSwapV3,  // 6 - V3 con características únicas
  Balancer,       // 7 - Pools balanceados
  Curve,          // 8 - Optimizado para stablecoins
  MaverickV2,     // 9 - Concentrated liquidity avanzada
  // DEXes adicionales (10-26)
  Antfarm, Apeswap, Defiswap, Elkfinance, Ethervista,
```

```
Fraxswap, Hopeswap, KyberClassic, KyberElastic,
Radioshack, Saitaswap, Sakeswap, Shibaswap,
Swardex, Solidly, Swapr, Verse
}
```

- **DEXes V2-like soportados:**
 - UniswapV2, SushiSwapV2, PancakeSwapV2
 - Antfarm, Apeswap, Defiswap, Elkfinance
 - Ethervista, Fraxswap, Hopeswap, Radioshack
 - Saitaswap, Sakeswap, Swardex, Solidly, Swapr, Verse
- **DEXes V3-like soportados**
 - UniswapV3, UniswapV4, SushiSwapV3, PancakeSwapV3
 - KyberElastic, Shibaswap
- **Protocolos especializados:**
 - Balancer (*AMM* con pools balanceados)
 - Curve (optimizado para *stablecoins*)
 - MaverickV2 (*concentrated liquidity*)
 - KyberClassic (reserva dinámica)
- **Capacidades del sistema:**
 - Evaluación simultanea de múltiples rutas en los 27 *DEXes* a través de la función `findArbitrageOpportunities()`.
 - Priorización inteligente basada en gas, *slippage* y rentabilidad neta.
 - Actualización dinámica de datos de liquidez vía *GeckoTerminal API*.
 - Configuración automática mediante `setupAllDexes()` en `FlashLoanArbitrage.sol`.

3.2.6 Librerías utilizadas

El directorio `node_modules` es el núcleo del entorno de desarrollo y ejecución fuera de la *blockchain* (*off-chain*) del proyecto. Administrado por `npm`, contiene todas las librerías y herramientas de *TypeScript* necesarias para compilar y desplegar los contratos inteligentes, así como para ejecutar los scripts de arbitraje en tiempo real. La selección de estas dependencias fue cuidadosa, priorizando la robustez, seguridad y eficiencia.

Estas librerías permiten que los scripts escritos en *TypeScript* se comuniquen con la *blockchain* de *Ethereum*:

- **ethers.js** es la librería principal para interactuar con *Ethereum*. Se utiliza para

- Crear instancias de contratos inteligentes.
 - Conectarse a un nodo *Ethereum* mediante un proveedor RPC.
 - Firmar y enviar transacciones usando una clave privada.
 - Convertir valores entre diferentes unidades (por ejemplo, de Wei a Ether).
 - Consultar información de la *blockchain*, como balances y precios de gas.
- **OpenZeppelin**
 - **SafeERC20**: Para realizar operaciones con *tokens ERC20* de forma segura, asegurando que las transacciones fallen si el *token* no se transfiere correctamente. Utiliza funciones como `safeTransfer()`, `safeApprove()`, y `_callOptionalReturn()`.

3.2.6.1 Librerías de Scripts de Automatización y Utilidades

Estas herramientas apoyan la lógica de los scripts que se ejecutan fuera de la *blockchain*:

1. **axios**: Cliente HTTP basado en promesas, usado para consultar APIs externas como GeckoTerminal y obtener datos de pools de liquidez.
2. **dotenv**: Permite cargar variables de entorno desde un archivo `.env`, manteniendo claves privadas y configuraciones sensibles fuera del código fuente.
3. **@flashbots/ethers-provider-bundle**: Librería especializada para enviar transacciones directamente a los mineros/validadores a través de *Flashbots*, lo que ofrece:
 - **Privacidad**: Las transacciones no se publican en el *mempool* público, evitando que otros *bots* las copien o adelanten.
 - **Pago condicional**: Solo se paga la tarifa de gas si la transacción se incluye en un bloque, evitando costos por transacciones fallidas.

En resumen, la arquitectura del sistema no solo depende de los contratos inteligentes, sino también de un conjunto de herramientas y librerías cuidadosamente seleccionadas que trabajan juntas para crear una aplicación *DeFi* robusta, segura y eficiente.

3.3 Arquitectura *On-Chain*: Contratos Inteligentes

La arquitectura *on-chain* es el núcleo del sistema, diseñada para ser segura, modular y eficiente en términos de gas. Se compone de dos contratos inteligentes principales que trabajan en

conjunto: `FlashLoanArbitrage.sol` y `DexAggregator.sol`. Este diseño sigue el principio de separación de responsabilidades, donde cada contrato tiene un propósito bien definido.

3.3.1 Contratos Inteligentes y Arquitectura de Smart Contracts

El sistema propuesto se compone por una arquitectura distribuida de contratos inteligentes, cada uno con una responsabilidad específica que, en conjunto, permiten ejecutar el arbitraje de manera atómica, segura y modular.

- **Contratos Principales Desarrollados:**

1. `FlashLoanArbitrage.sol`: Es el contrato principal del proyecto, el cual implementa la interfaz `IFlashLoanSimpleReceiver` del protocolo AAVE V3. Sus características principales incluyen:
 - Implementar la función `executeFlashLoanSimple()` para solicitar préstamos *flash* optimizados de un solo token.
 - Contiene la función `executeOperation()` requerida por AAVE para procesar el préstamo recibido.
 - Mantiene una lista de 39 *tokens* intermediarios configurables para optimizar rutas de arbitraje
 - Configura automáticamente 27 *DEXes* diferentes mediante `setupAllDexes()`.
 - Incluye protección contra reentrada específica para *flash loans* con `flashLoanReentrancyGuard`.
 - Implementa eventos de *debug* detallados para monitoreo y análisis de transacciones
 - Cachea el premium de *flash loan* para optimización de *gas*.
2. `DexAggregator.sol`: Contrato especializado que abstrae la lógica de ejecución entre múltiples *DEXes*:
 - Define un `enum DexType` con 27 tipos diferentes de *DEXes* (índices 0-26) y clasificar cada protocolo.
 - Implementa interfaces específicas para cada protocolo con estructuras de parámetros optimizadas
 - Proporciona funciones `getBestDexQuote()` y `swapOnDex()` para cotización y ejecución.
 - Utiliza *mappings* `isV2Like` y `isV3Like` para categorización automática de protocolos
 - Soporta diferentes niveles de comisiones para Uniswap V3.

- **Contratos Inteligentes de Terceros Utilizados**

- **Ownable.sol**: Implementa un control de acceso simple permitiendo restringir funciones administrativas donde solo un solo *address* (el "propietario") puede ejecutar funciones críticas.
- **ReentrancyGuard.sol**: Protege funciones contra ataques de reentrada, especialmente importante en funciones que manejan fondos como `executeOperation()`.

3.3.1.1 El Orquestador Central

El contrato `FlashLoanArbitrage.sol` es el cerebro de la operación de arbitraje. Su función no es ejecutar los swaps directamente, sino orquestar todo el proceso, desde la solicitud del préstamo *flash* hasta la gestión de las ganancias.

- **Responsabilidades Clave:**

- **Integración con Aave V3**: Interactúa directamente con el *pool* de Aave V3 para solicitar préstamos *flash*, implementando la interfaz `IFlashLoanSimpleReceiver.sol`.
- **Punto de Entrada**: Expone la función `executeFlashLoanSimple()`, que es llamada por el script *off-chain* (`arbitrage.ts`) para iniciar una oportunidad de arbitraje.
- **Lógica de Ejecución**: La función de callback `executeOperation()` contiene la lógica principal que se ejecuta una vez que se reciben los fondos del préstamo. Esta función es la que invoca al “*DexAggregator*” para realizar los intercambios
- **Seguridad**: Implementa `ReentrancyGuard` de OpenZeppelin en su función de callback para prevenir ataques de reentrada, un vector de ataque común en protocolos que manejan *flash loans*. También utiliza “*Ownable*” para restringir funciones administrativas críticas
- **Gestión de Fondos**: Se encarga de calcular la prima del préstamo y de asegurar que el monto total (préstamo + prima) sea devuelto al *pool* de Aave al final de la transacción.

```
// ...
contract FlashLoanArbitrage is Ownable, ReentrancyGuard, IFlashLoanSimpleReceiver {
    using SafeERC20 for IERC20;
    IPoolAddressesProvider public override ADDRESSES_PROVIDER;
    IPool public immutable POOL;
    DexAggregator public dexAggregator;
    // ...código existente...
    function executeOperation(
        address asset,
        uint256 amount,
```

```

        uint256 premium,
        address initiator,
        bytes calldata params
    ) external override nonReentrant returns (bool) {
        // ... Lógica de validación ...
        uint256 amountOwing = amount + premium;
        // ... Lógica de arbitraje llamando a _executeArbitrage ...
        _executeArbitrage(asset, amount, amountOwing);
        // ... Lógica de repago del préstamo ...
        IERC20(asset).safeApprove(address(PPOOL), amountOwing);
        return true;
    }
    // ...código existente...
}

```

Código 3.3: Declaración principal de herencia, dependencias de seguridad de OpenZeppelin e interfaces de Aave implementada en el contrato inteligente `FlashLoanArbitrage.sol`.

3.3.1.2 El Agregador de Liquidez

El contrato `DexAggregator.sol` actúa como una capa de abstracción o un "router universal" para interactuar con los diferentes intermediarios descentralizados (*DEXes*). En lugar de que `FlashLoanArbitrage.sol` contenga la lógica específica para cada *DEX*, delega esta tarea al `DexAggregator.sol`.

Responsabilidades Clave:

- **Interfaz Unificada:** Proporciona una única función, `executeSwap()`, que el contrato principal puede llamar sin necesidad de conocer los detalles de implementación del *DEX* subyacente.
- **Registro de DEXes:** Mantiene un registro de los *DEXes* soportados, incluyendo sus direcciones de router/*pool* y el tipo de *DEX* que son (ej. Uniswap V2, Uniswap V3, Curve, etc.).
- **Modularidad y Extensibilidad:** Este es el mayor beneficio de su diseño. Para agregar soporte para un nuevo *DEX*, solo es necesario actualizar la lógica dentro de `DexAggregator.sol` y registrar el nuevo *DEX* a través de una función administrativa, sin tener que modificar o redesplegar el contrato `FlashLoanArbitrage.sol`.
- **Lógica de Swap Específica:** Contiene el código de bajo nivel para construir las llamadas a las funciones de swap de cada protocolo, manejando las diferencias entre ellos (por ejemplo, rutas de tokens para V2 vs. pools con fees para V3).

```

// ...código existente...
contract DexAggregator is Ownable {
    // Enum para identificar el tipo de DEX

```

```

enum DexType {
    UniswapV2,
    UniswapV3,
    SushiSwap,
    // ... otros tipos
}

// Estructura para almacenar la información de cada DEX
struct DexInfo {
    string name;
    DexType dexType;
    address router; // Router para V2-like
    address quoter; // Quoter para V3-like
    bool active;
}

DexInfo[] public dexes;

function executeSwap(
    uint256 dexIndex,
    address tokenIn,
    address tokenOut,
    uint256 amountIn
) external returns (uint256 amountOut) {
    // ... Lógica para despachar el swap al DEX correcto ...
}
// ...código existente...
}

```

Código 3.4: Estructura general, variables de estado y firma de la función `executeSwap()` implementadas en el contrato inteligente `DexAggregator.sol` con lenguaje solidity.

- **Flujo de Ejecución *On-Chain***

1. **Inicio:** El script `arbitrage.ts` identifica una oportunidad y llama a `executeFlashLoanSimple` en `FlashLoanArbitrage.sol`.
2. **Préstamo:** El contrato solicita un préstamo relámpago al *POOL* de Aave.
3. **Callback:** Aave transfiere los fondos al contrato y llama a `executeOperation`.
4. **Ejecución de Arbitraje:** Dentro de `executeOperation`, el contrato invoca a `dexAggregator.executeSwap` para realizar el primer intercambio (ej. comprar barato).
5. **Segundo Intercambio:** El contrato vuelve a llamar a `dexAggregator.executeSwap` para realizar el segundo intercambio (ej. vender caro).
6. **Repago:** El contrato aprueba al *POOL* de Aave para retirar el monto del préstamo más la prima.

7. **Finalización:** La función `executeOperation` termina. Aave verifica su balance y retira los fondos. El beneficio neto permanece en el contrato `FlashLoanArbitrage.sol`, listo para ser retirado por el propietario.

3.3.2 Funciones Clave del Contrato Principal

A continuación, se detallan las dos funciones centrales del contrato `FlashLoanArbitrage.sol`, las cuales son las encargadas de solicitar y gestionar un *flashloan*:

- **Función de Solicitud de Préstamo *Flash***

```
function executeFlashLoanSimple(address asset, uint256 amount) external {
    emit DebugArbitrageStep("Starting flash loan request", asset, amount);

    try POOL.flashLoanSimple(address(this), address(asset), amount, "", 0) {
        emit DebugArbitrageStep("Flash loan initiated successfully", asset,
amount);
    } catch Error(string memory reason) {
        emit DebugMessage(string(abi.encodePacked("Flash loan failed: ",
reason)));
        revert(reason);
    }
}
```

Código 3.5: Función `executeFlashLoanSimple()` en lenguaje Solidity para solicitar préstamo flash simple de Aave V3 implementada en el contrato inteligente `FlashLoanArbitrage.sol`.

- **Función Callback de AAVE**

```
function executeOperation(
    address asset,
    uint256 amount,
    uint256 premium,
    address initiator,
    bytes calldata params
) external override returns (bool) {
    emit DebugArbitrageStep("executeOperation called", asset, amount);

    require(msg.sender == address(POOL), "Caller must be Aave Pool");
    require(initiator == address(this), "Initiator must be this contract");
    require(!_flashLoanReentrancyGuard, "Reentrant call");

    _flashLoanReentrancyGuard = true;

    uint256 amountOwing = amount + premium;

    // Ejecutar la lógica de arbitraje
```

```

(uint256 received, uint256 operationProfit) = _executeArbitrage(
    asset,
    amount,
    amountOwing
);

// Aprobar el repago del préstamo flash
IERC20(asset).safeApprove(address(P00L), 0);
IERC20(asset).safeApprove(address(P00L), amountOwing);

_flashLoanReentrancyGuard = false;
return true;
}

```

Código 3.6: Función de *callback* `executeOperation()` en lenguaje Solidity exigida por Aave para validar la recepción de fondos y ejecutar la lógica de arbitraje implementada en el contrato inteligente `FlashLoanArbitrage.sol`.

Características:

- Verifica que el llamador sea el *pool* de AAVE y que el iniciador sea el propio contrato, evitando ataques de reentrada.
- Calcula automáticamente el monto total a devolver (préstamo + prima).
- Ejecuta la función interna `_executeArbitrage()` para determinar y ejecutar la mejor ruta de swaps.
- Finalmente, aprueba y realiza el repago del préstamo.

• Función de Arbitraje Interna:

```

function _executeArbitrage(
    address tokenIn,
    uint256 amountIn,
    uint256 amountOwing
) internal returns (uint256 receivedAmount, uint256 profit) {
    // Buscar la mejor ruta entre 39 tokens intermediarios
    for (uint256 i = 0; i < intermediaryTokens.length; i++) {
        address intermediaryToken = intermediaryTokens[i];
        // Obtener cotización para el primer swap
        (uint256 midAmountOut, uint256 firstDexIndex) = dexAggregator
            .getBestDexQuote(tokenIn, intermediaryToken, amountIn);
        if (midAmountOut == 0) continue;
        // Obtener cotización para el segundo swap
        (uint256 finalAmountOut, uint256 secondDexIndex) = dexAggregator
            .getBestDexQuote(intermediaryToken, tokenIn, midAmountOut);
        // Actualizar la mejor ruta si se obtiene mayor cantidad
        if (finalAmountOut > bestReceivedAmount) {
            bestReceivedAmount = finalAmountOut;
            bestIntermediaryToken = intermediaryToken;
            bestFirstDexIndex = firstDexIndex;
            bestSecondDexIndex = secondDexIndex;
        }
    }
}

```

```

        bestMidAmountOut = midAmountOut;
    }
}
// Ejecutar los swaps solo si la ruta es rentable
if (bestReceivedAmount > amountIn) {
    // Primer swap: tokenIn → token intermedio
    IERC20(tokenIn).safeApprove(address(dexAggregator), amountIn);
    uint256 midAmount = dexAggregator.swapOnDex(
        bestFirstDexIndex,
        tokenIn,
        bestIntermediaryToken,
        amountIn,
        (bestMidAmountOut * 995) / 1000 // Tolerancia de deslizamiento del
0.5%
    );
    // Segundo swap: token intermedio → tokenIn
    IERC20(bestIntermediaryToken).safeApprove(address(dexAggregator),
midAmount);
    receivedAmount = dexAggregator.swapOnDex(
        bestSecondDexIndex,
        bestIntermediaryToken,
        tokenIn,
        midAmount,
        (bestReceivedAmount * 995) / 1000 // Tolerancia de deslizamiento
(slippage) del 0.5%
    );
}
profit = receivedAmount > amountOwing ? receivedAmount - amountOwing : 0;
return (receivedAmount, profit);
}

```

Código 3.7: Función `_executeArbitrage()` en lenguaje Solidity para evaluar cotizaciones en tokens intermediarios y ejecutar los intercambios atómicos implementada en el contrato inteligente `FlashLoanArbitrage.sol`.

Características:

- Evalúa 39 tokens intermediarios para encontrar la ruta óptima entre los 27 DEXes configurados en el *Smart contract* `DexAggregator.sol`.
- Realiza dos swaps atómicos: uno del token prestado al token intermedio y otro del token intermedio de vuelta al token original, siempre que la operación sea rentable.
- Calcula el beneficio neto restando el monto adeudado (préstamo + prima) del monto final recibido.

3.3.3 Flujo de Ejecución Atómica

La operación completa se realiza en una única transacción, garantizando la atomicidad del proceso de arbitraje. La secuencia es la siguiente:

1. **Iniciación:** El script `arbitrage.ts` detecta una oportunidad y llama a la función `executeFlashLoanSimple()` especificando el token y la cantidad calculada.
2. **Préstamo *Flash*:** AAVE transfiere instantáneamente los fondos prestados al contrato utilizando `flashLoanSimple()`.
3. **Callback Automático:** El *pool* de AAVE invoca `executeOperation()` del contrato, pasando parámetros como el token, cantidad, premium e iniciador.
4. **Validaciones de Seguridad:** Se verifican condiciones críticas (correcta procedencia del llamador, ausencia de reentrada y validez del iniciador).
5. **Búsqueda de Ruta Óptima:** La función interna `_executeArbitrage()` explora 39 tokens intermediarios para determinar la ruta más rentable, utilizando información de cotizaciones de 27 *DEXes*.
6. **Ejecución de Swaps:** Se realizan dos swaps atómicos para convertir el token prestado en una cantidad superior del mismo token.
 - **Primer swap:** del token prestado a un token intermedio.
 - **Segundo swap:** del token intermedio nuevamente al token prestado.
7. **Cálculo de Beneficio:** Se determina el beneficio neto deduciendo el premium del *flash loan* al monto final obtenido.
8. **Repago Automático:** Se aprueban y ejecutan las transferencias necesarias para reembolsar el préstamo junto con el premium a AAVE.
9. **Finalización:** Si todos los pasos se completan exitosamente, la transacción se consolida; de lo contrario, se revierte por completo (garantizando la atomicidad).

3.3.4 Roles, Permisos y Seguridad

La protección de funciones utiliza múltiples capas de seguridad:

- **Protección Específica para *Flash Loans*:**
 - Uso de `_flashLoanReentrancyGuard` que se activa antes de iniciar la lógica de arbitraje y se desactiva al finalizar, evitando llamadas reentrantes durante operaciones *flash*.
- **Funciones de Emergencia:**
 - `setTestMode(bool _testMode)`: Permite activar o desactivar el modo prueba.
 - `rescueTokens(address token, address to, uint256 amount)`: Permiten retirar tokens en casos de emergencia.

- `rescueETH(address payable to, uint256 amount)`: Permiten retirar ETH del contrato cuando sea necesario.

- **Modificadores de Seguridad:**

```
modifier onlyOwner() {
    _checkOwner();
    _;
}
modifier nonReentrant() {
    _nonReentrantBefore();
    _;
    _nonReentrantAfter();
}
```

Código 3.8: Modificadores de seguridad `onlyOwner()` y `nonReentrant()` de OpenZeppelin para control de acceso y prevención de ataques de reentrada implementados con lenguaje Solidity en los contratos inteligentes.

El modificador `onlyOwner()` protege las siguientes funciones administrativas:

- `setupAllDexes()`: Configuración inicial de 27 *DEXes*.
- `setupIntermediaryTokens()`: Configuración de 39 tokens intermediarios.
- `setDexAggregator()`: Cambio del agregador de *DEXes*.
- `rescueTokens()` y `rescueETH()`: Recuperación de fondos en emergencias.

El modificador `nonReentrant()` aplicado a funciones que manejan transferencias de tokens para prevenir ataques de reentrada.

- **Protección Específica para *Flash Loans***

El fragmento con `_flashLoanReentrancyGuard` implementa una bandera interna que previene ataques de reentrada específicamente durante la ejecución de operaciones de *flash loans*.

```
bool private _flashLoanReentrancyGuard;

require(!_flashLoanReentrancyGuard, "Reentrant call");
_flashLoanReentrancyGuard = true;
// ... lógica de arbitraje ...
_flashLoanReentrancyGuard
```

Código 3.9: Variable `_flashLoanReentrancyGuard` para crear un mecanismo de bloqueo específico para operaciones flash implementada en lenguaje solidity en el contrato inteligente `FlashLoanArbitrage.sol`.

Esto asegura que la lógica de arbitraje dentro del préstamo *flash* no pueda ser ejecutada más de una vez en la misma transacción, reforzando la seguridad del contrato.

- **Funciones de Emergencia**

Las funciones `setTestMode`, `rescueTokens` y `rescueETH` permiten al propietario del contrato activar un modo de prueba y recuperar tanto tokens ERC20 como ETH que hayan quedado

```
function setTestMode(bool _testMode) external onlyOwner {
    testMode = _testMode;
}

function rescueTokens(address token, address to, uint256 amount) external
onlyOwner {
    IERC20(token).safeTransfer(to, amount);
}

function rescueETH(address payable to, uint256 amount) external onlyOwner {
    to.transfer(amount);
}
```

Código 3.10: Funciones de emergencia `setTestMode()`, `rescueTokens()` y `rescueETH()` para recuperación de fondos implementadas con lenguaje Solidity en el contrato inteligente `FlashLoanArbitrage.sol`.

Estas funciones son herramientas administrativas para manejar situaciones imprevistas y proteger los fondos del contrato.

3.3.5 Modularidad, Escalabilidad y Extensión

La arquitectura basada en dos contratos principales permite escalabilidad y mantenimiento eficientes:

- **Extensión de *DEXes***

- Pasos para agregar un nuevo *DEX*:
 1. Extender el `enum DexType` en `DexAggregator.sol`.
 2. Añadir la interfaz específica del *DEX* (ej: `INewDexRouter`).
 3. Implementar la lógica en `swapOnDex()` para el nuevo tipo.
 4. Actualizar los mappings `isV2Like` o `isV3Like` según corresponda.
 5. Agregar la llamada en `setupAllDexes()` del contrato principal.

- **Configuración Dinámica**

Tokens Intermediarios: El sistema utiliza 39 tokens ordenados por liquidez:

```
// Tokens principales (mayor liquidez)
intermediaryTokens.push(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2); // WETH
intermediaryTokens.push(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48); // USDC
intermediaryTokens.push(0x6B175474E89094C44Da98b954EedeAC495271d0F); // DAI
```

```
// ... hasta completar 39 diferentes tokens
```

Código 3.11: Configuración del arreglo `intermediaryTokens` para definir los 39 activos de alta liquidez como puentes de enrutamiento implementada con lenguaje solidity en el contrato inteligente `FlashLoanArbitrage.sol`.

DEXes Soportados: El sistema configura automáticamente 27 DEXes:

- 16 *DEXes* V2-like (Uniswap V2, SushiSwap V2, PancakeSwap V2, etc.)
 - 7 *DEXes* V3-like (Uniswap V3, SushiSwap V3, PancakeSwap V3, etc.)
 - 4 *DEXes* especializados (Balancer, Curve, KyberClassic, MaverickV2)
- **Optimizaciones Implementadas**
 1. **Gas Optimization:**
 - Uso de `flashLoanSimple()` vs `flashLoan()` ahorra comisiones de gas por operación.
 - Cache del premium de *flash loan* evita consultas repetidas.
 - Tokens intermediarios ordenados por liquidez para encontrar rutas óptimas.
 2. **MEV Protection:**
 - Integración con “Flashbots Bundle Provider” para transacciones privadas.
 - Gas agresivo configurable para competir en el *mempool*.
 - Validación de rentabilidad antes de ejecutar para evitar transacciones no rentables.
 - **Separación de Responsabilidades**
 1. **Capa On-Chain:** Los contratos se encargan únicamente de:
 - Ejecutar swaps validados
 - Gestionar seguridad y permisos
 - Optimizar rutas de arbitraje
 2. **Capa Off-Chain:** Los scripts se encargan de:
 - Detectar oportunidades mediante `arbitrage.ts`.
 - Actualización de datos de liquidez via `liquidity.ts`.
 - Análisis de costos de gas con `gasFee.ts`.

Esta separación permite que los contratos sean simples, seguros y gas-eficientes, mientras que la lógica compleja de detección y análisis se ejecuta *off-chain* donde es más económica y flexible.

3.3.6 Optimización de Comisiones por Transacción

Una de las variables más críticas al momento de evaluar la rentabilidad real de una operación de arbitraje es el costo total en gas. En *Ethereum*, el gas es la unidad de medida del cómputo requerido para ejecutar una transacción o interacción con un contrato inteligente, y su precio varía constantemente en función de la demanda en la red. Aunque una oportunidad de arbitraje pueda ser rentable en términos de precios “*tokenizados*”, si el costo de gas supera la ganancia bruta estimada, la operación se vuelve inviable o incluso puede generar pérdidas.

3.3.6.1 Consulta del precio de gas

Para abordar esta problemática, se diseñó e implementó el módulo `gasFee.ts`, el cual se ejecuta automáticamente en intervalos predefinidos o puede invocarse *on-demand*. Sus principales funciones son:

```
const gasPrice = await provider.getGasPrice();
console.log("Gas Price actual:", ethers.utils.formatUnits(gasPrice, "gwei"),
"gwei");
```

Código 3.12: Consulta del precio de *gas* actual mediante el proveedor *RPC* escrito en lenguaje typescript en el script `gasFee.ts`.

Este valor se transforma de *wei* a Gwei utilizando `ethers.utils.formatUnits`, facilitando la interpretación y el ajuste dinámico de los parámetros de ejecución *off-chain*.

3.3.6.2 Almacenamiento de Datos en Tiempo Real

El resultado de estas consultas se almacena en un archivo `gasFee.json` con el siguiente formato:

```
{
  "timestamp": 1717882425,
  "baseFeePerGasGwei": 34.25,
  "maxFeePerGasGwei": 39.88,
  "maxPriorityFeePerGasGwei": 2.5,
  "gasPrice": "0x0de0b6b3a7640000"
  ...
}
```

Código 3.13: Estructura del archivo `gasFee.json` con datos de tarifas de gas actuales e históricas.

Estos datos son consumidos por los scripts de ejecución de arbitraje `arbitrage.ts` y de consulta de liquidez `liquidity.ts` para evaluar si la operación es rentable bajo las condiciones actuales del gas. Se definen umbrales de rentabilidad, y si el gas fee supera el límite tolerado, la operación es abortada.

3.3.6.3 Adopción de EIP-1559

Se integró compatibilidad con el modelo *EIP-1559* introducido en *Londres Hardfork* de *Ethereum*, mediante el uso de `maxFeePerGas` y `maxPriorityFeePerGas`. Este modelo mejora la previsibilidad del costo de las transacciones y es fundamental para competir por inclusión en bloques congestionados, crucial para competir en una red congestionada, minimizando así el riesgo de exclusión o *front-running*.

```
const feeData = await provider.getFeeData();
const tx = {
  maxPriorityFeePerGas: feeData.maxPriorityFeePerGas,
  maxFeePerGas: feeData.maxFeePerGas,
  gasLimit: 2500000,
};
```

Código 3.14: Construcción de parámetros de transacción compatibles con EIP-1559 mediante `getFeeData()` escrito en lenguaje typescript.

En resumen, este mecanismo recolecta información en tiempo real y condiciona inteligentemente la ejecución del arbitraje sobre la base de un contexto económico favorable. De este modo se optimiza la probabilidad de éxito de la operación.

3.3.7 Gestión y Evaluación de Liquidez en Pools

El arbitraje no puede ejecutarse de manera efectiva si uno o más de los pools involucrados carecen de suficiente liquidez. Un bajo nivel de liquidez puede provocar ejecuciones con precios adversos (*slippage*), lo que significa que el precio real de ejecución varía negativamente respecto al esperado, disminuyendo o anulando la rentabilidad esperada. Para garantizar la viabilidad de las operaciones, el sistema implementa un proceso de consulta, filtrado y validación

3.3.7.1 Consulta de liquidez

Para mitigar este problema, se implementó el script `liquidity.ts`, encargado de consultar directamente los contratos de pools de los *DEXes*, utilizando funciones como `getReserves()` o `getPoolTokens()`.

```
const reserves = await pair.getReserves();
console.log("Reserves:", reserves[0].toString(), reserves[1].toString());
```

Código 3.15: Consulta de reservas de un pool de liquidez mediante la función `getReserves()` escrito en lenguaje typescript.

Los resultados son almacenados en el archivo `dexspools.json` para posteriormente ser analizado por los scripts creados. El sistema prioriza aquellas rutas de swap con mayor liquidez y evita rutas con reservas insuficientes o desbalanceadas.

3.3.7.2 Filtrado por liquidez mínima

Una vez obtenidos los datos, el sistema implementa filtros para prevenir operaciones y manipulaciones en pools de bajo volumen. Solo se consideran pools que cuenten con una liquidez superior a \$100,000 USD:

```
const MIN_LIQUIDITY_USD = 100000; // $100K mínimo

const filteredPools = dexData.data.filter(pool =>
  parseFloat(pool.attributes.reserve_in_usd) >= MIN_LIQUIDITY_USD
);
```

Código 3.16: Filtrado de pools con liquidez inferior a \$100,000 USD en el script `arbitrage.ts`.

Al establecer este umbral mínimo, se descartan los pools susceptibles a manipulación o que no soportarían el volumen del préstamo *flash* sin generar un impacto negativo en el precio.

3.3.7.3 Validación de disponibilidad en Aave

Como último paso de seguridad antes de intentar ejecutar cualquier operación de arbitraje en los pools filtrados, el sistema verifica que el token base solicitado esté disponible para préstamos flash en el protocolo de Aave:

```

async function isTokenAvailableForFlashloan(tokenAddress: string):
Promise<boolean> {
    return AAVE_TOKENS.includes(tokenAddress.toLowerCase());
}

```

Código 3.17: Validación de disponibilidad de un token para préstamos relámpago en las reservas de Aave escrito en lenguaje typescript.

Esta validación previene intentos de arbitraje con tokens no soportados por el proveedor de liquidez, evitando fallos en la transacción y el consecuente gasto innecesario de comisiones de *gas*.

3.3.8 Control del *Slippage* en Ejecución

El *slippage* es una métrica clave para la toma de decisiones dentro del contrato. Se define como la variación adversa entre el precio proyectado y el obtenido. El contrato incluye validaciones explícitas que abortan la operación si el valor de `amountOut` cae por debajo del umbral mínimo: `minAmountOut` preconfigurado según la tolerancia aceptada (de 0.3% a 1%).

```

require(amountOut >= minAmountOut, "Slippage too high");

```

Código 3.18: Validación *on-chain* de *slippage* máximo permitido en la ejecución de intercambios escrito en lenguaje solidity.

- **Cálculo *off-chain*:**

```

const minAmountOut = expectedAmountOut.mul(995).div(1000); // Permite un
slippage de 0.5%

```

Código 3.19: Cálculo *off-chain* del monto mínimo aceptable con tolerancia de *slippage* del 0.5% escrito con typescript.

Esta condición actúa como una protección crítica ante cambios súbitos de mercado entre la detección y ejecución de la oportunidad.

3.3.9 Algoritmo de Detección y Evaluación de Oportunidades

La rentabilidad de un arbitraje no solo depende de los precios entre dos *DEXes*, sino de la capacidad del sistema para identificar, evaluar y filtrar eficientemente estas oportunidades en tiempo real.

El proyecto implementa esta lógica de manera integrada dentro de la función `findArbitrageOpportunities()` en el script `arbitrage.ts`. Este enfoque prioriza la

velocidad y la precisión en la evaluación de oportunidades de arbitraje simples (comprar en un *DEX*, vender en otro).

El algoritmo opera a través de una secuencia de pasos optimizados:

1. **Agrupación por Pares de Tokens:** Para evitar comparaciones ineficientes, todos los puntos de precio obtenidos de los diversos *DEXes* se agrupan en un mapa (*Record<string, TokenPrice[]>*). La clave de este mapa es una representación normalizada del par de tokens (ej. WETH/USDC), permitiendo un acceso casi instantáneo a todos los precios disponibles para un mismo par en diferentes mercados.
2. **Clasificación de Precios:** Dentro de cada grupo, los precios se clasifican en dos listas distintas:
 - **buyPrices:** Ordenada de forma ascendente, presentando los precios de compra más bajos primero.
 - **sellPrices:** Ordenada de forma descendente, presentando los precios de venta más altos primero.
3. **Análisis Heurístico de Combinaciones:** Para reducir drásticamente el espacio de búsqueda y el tiempo de cómputo, el algoritmo no evalúa cada posible combinación de compra y venta. En su lugar, aplica una heurística que se enfoca únicamente en las oportunidades más prometedoras. Típicamente, compara las 3 mejores ofertas de compra con las 3 mejores ofertas de venta asumiendo que las oportunidades más rentables se encontrarán en estos extremos.
4. **Evaluación Detallada de Rentabilidad:** Para cada combinación prometedora, se realiza un análisis exhaustivo que considera múltiples factores críticos para determinar la viabilidad real de la operación:
 - **Margen Bruto y Slippage:** Se calcula el porcentaje de ganancia teórica. Este valor se ajusta inmediatamente para tener en cuenta el peor escenario de *slippage*, utilizando una constante predefinida (**MAX_SLIPPAGE_PERCENT**) para derivar un precio de compra efectivo `effectiveBuyPrice` y un precio de venta efectivo `effectiveSellPrice`.
 - **Estimación del Volumen de Operación:** El tamaño máximo de la operación `maxTradeSize` se estima como un pequeño porcentaje (ej. 0.3%) de la liquidez del *pool* más pequeño de los dos involucrados, para minimizar el *impacto* en el precio.
 - **Cálculo de Costos:** Se cuantifican los costos operativos. El costo del gas se calcula multiplicando un límite de gas conservador (**GAS_LIMIT_ARBITRAGE**) por el

precio del gas en tiempo real. La comisión del préstamo *flash* de Aave (**FLASH_LOAN_FEE**) se calcula sobre el tamaño de la operación.

- **Beneficio Neto (*Net Profit*):** El beneficio neto en USD (**netProfitUSD**) se calcula restando los costos totales (*gas* + comisión del *flash loan*) del beneficio bruto estimado.

5. **Filtrado y Selección:** Finalmente, solo las oportunidades cuyo beneficio neto supera un umbral mínimo predefinido (**MIN_PROFIT_USD**) y cuyo token requerido para el préstamo *flash* está disponible en las reservas de Aave V3 son consideradas válidas. Las oportunidades resultantes se ordenan por el mayor beneficio neto, y la más rentable se pasa a la siguiente fase para su posible ejecución.

Este diseño metodológico asegura que el sistema no solo encuentre discrepancias de precios, sino que también valide rigurosamente su rentabilidad económica en el contexto de las condiciones actuales de la red y la liquidez del mercado.

3.3.10 Eficiencia Computacional

Dado que las oportunidades de arbitraje son extremadamente volátiles y de corta duración (a menudo existiendo por pocos segundos o bloques), la eficiencia computacional de los scripts de análisis es un factor determinante para el éxito. Se implementaron varias optimizaciones clave para asegurar que la evaluación se complete en el menor tiempo posible.

- **Uso de Estructuras de Datos Eficientes:** La agrupación de precios por par de tokens se realiza utilizando un objeto tipo “Record” en *Typescript*, que funciona como un mapa hash. Esta estructura permite agrupar y acceder a los precios asociados a cada par de tokens de forma casi inmediata, sin tener que recorrer listas completas cada vez que se requiere una comparación. Esto se traduce en una gran reducción del tiempo de búsqueda y procesamiento.
- **Reducción del Espacio de Búsqueda:** Se aplican dos filtros proactivos para reducir drásticamente la cantidad de datos a procesar:
 1. **Filtrado por Liquidez:** Durante la fase de carga de datos (mediante la función `loadTopTokens`), se descartan los pools cuyo nivel de liquidez sea inferior a un umbral mínimo (**MIN_LIQUIDITY_USD**). Así, el sistema solo analiza mercados con suficiente profundidad, lo cual es fundamental para operar sin incurrir en un deslizamiento (*slippage*) excesivo.

2. **Heurística de Combinaciones:** En vez de explorar todas las combinaciones posibles para transacciones de compra y venta, se limita el análisis a un subconjunto reducido de las mejores opciones. Esta heurística evita una explosión combinatoria en las evaluaciones, reduciendo de manera significativa el número de cálculos necesarios.

- **Paralelización de Consultas de Red:** La recolección de datos de liquidez de diferentes *DEXes* se realiza de forma concurrente utilizando la función `Promise.all` en el script `liquidity.ts`. Al ejecutar múltiples solicitudes simultáneamente, en lugar de hacerlo de manera secuencial, se reduce considerablemente la latencia total de la fase de recolección. Esto resulta esencial para obtener datos actualizados en tiempo real y evaluar la rentabilidad antes de que la oportunidad desaparezca o sea captada por otros *bots*.

Estas medidas combinadas reducen el tiempo promedio de evaluación por ciclo de manera sustancial, lo cual es crucial para competir con otros *bots* y validar la rentabilidad de una oportunidad antes de que sea explotada por otro actor del mercado o desaparezca debido a las fluctuaciones naturales del mismo.

3.4 Arquitectura del sistema Off Chain: Automatización y Scripts

3.4.1 Arquitectura de Scripts *Off-Chain*: El Cerebro Operativo

La lógica *off-chain*, implementada en *TypeScript* y ejecutada con Node.js a través de Hardhat, es el componente que dirige y automatiza todo el sistema. Mientras los contratos inteligentes proporcionan la capacidad de ejecución atómica *on-chain*, los scripts *off-chain* son responsables de la inteligencia: detectar oportunidades, calcular la rentabilidad y decidir cuándo y cómo actuar. El sistema se basa en dos scripts principales: `arbitrage.ts`.

3.4.1.1 El Orquestador de Arbitraje Avanzado

El script `arbitrage.ts` es el núcleo de la operación *off-chain*. Su diseño está optimizado para la velocidad, la precisión y la resiliencia, reconociendo que las oportunidades de arbitraje son efímeras y altamente competidas. Su flujo de trabajo se puede dividir en varias fases críticas.

Fase 1: Inicialización y Carga de Datos en Memoria

Para minimizar la latencia durante la detección de oportunidades, el script no consulta APIs externas en su bucle principal. En su lugar, carga en memoria los datos recopilados previamente por otros procesos, como `liquidity.ts`.

1. **Carga de Datos de Pools:** Lee el archivo `dexespools.json` para obtener una instantánea completa de los pools de liquidez, sus reservas, precios y tokens asociados en docenas de *DEXs*.
2. **Configuración Dinámica de DEXes:** A diferencia de las listas estáticas, el script construye su conocimiento de los *DEXes* disponibles directamente desde los datos cargados. Utiliza un mapeo (***DEX_NAME_TO_INDEX***) para traducir los nombres de los *DEXes* (ej. 'uniswap_v3') a los índices numéricos que espera el contrato `DexAggregator.sol`, garantizando la extensibilidad del sistema.

```
// ...
const DEX_NAME_TO_INDEX = {
  // Main DEXes
  'uniswap_v2': 0,           // UniswapV2
  'uniswap_v3': 1,           // UniswapV3
  // ...
  'verse': 26,              // Verse
};

// ...
async function initialize() {
  try {
    await initWallet();
    await setupFlashbots();
    currentGasData = await getOptimizedGasData();
    await loadAaveReservesData(); // Carga datos de Aave
    const poolData = await loadPoolData(); // Carga datos de DEXs
    await updateDexInfo(poolData); // Configura los DEXs dinámicamente
    await setupContracts();
    await monitor();
  }
  // ...
}
```

Código 3.20: Mapeo de nombres de *DEXes* a índices del contrato y secuencia de inicialización en `arbitrage.ts` escrito con lenguaje typescript.

Fase 2: Detección y Cálculo de Rentabilidad

Esta es la fase computacionalmente más intensiva, donde el script analiza miles de puntos de precio para encontrar discrepancias rentables.

1. **Extracción de Precios:** El script procesa los datos de los pools para crear una lista unificada de precios `TokenPrice[]`, incluyendo tanto el precio directo (**BASE/QUOTE**) como el inverso (**QUOTE/BASE**) para cada *pool*.
2. **Agrupación por Pares:** Para una comparación eficiente, los precios se agrupan por par de tokens (ej. WETH/USDC).
3. **Cálculo de Costos Reales:** Antes de buscar oportunidades, el script obtiene el precio actual del gas y el precio de ETH/USD (usando **Chainlink** como fuente primaria) para calcular el costo estimado de una transacción de arbitraje en USD. Este es un paso fundamental para determinar la rentabilidad neta.
4. **Búsqueda de Oportunidades:** El script itera sobre cada grupo de pares, comparando los precios de compra más bajos con los precios de venta más altos entre diferentes *DEXs*. Para cada posible arbitraje, calcula:
 - **Beneficio Bruto:** La diferencia porcentual entre el precio de venta y el de compra.
 - **Impacto de Slippage:** Ajusta los precios efectivos para tener en cuenta el deslizamiento máximo configurado.
 - **Tamaño Óptimo de la Operación:** Estima un tamaño de operación basado en un pequeño porcentaje de la liquidez disponible en los pools involucrados para minimizar el *impacto* en el precio.
 - **Costos Totales:** Suma el costo del gas en USD y la prima del préstamo relámpago de Aave (0.05% del monto del préstamo).
 - **Beneficio Neto en USD:** El beneficio final después de restar todos los costos.

```
// ...
// Calcular prima de préstamo flash
const flashLoanFeeUSD = maxTradeSize * FLASH_LOAN_FEE;

// Calcular beneficio neto
const netProfitUSD = tradingProfitUSD - gasCostUSD - flashLoanFeeUSD;

// Determinar token de préstamo flash (preferir stablecoins)
let flashLoanToken, flashLoanAmount;
// ...
```

Código 3.21: Cálculo de rentabilidad neta descontando costos de gas y prima del préstamo en `arbitrage.ts`.

Fase 3: Ejecución Atómica y Estrategia Anti-Front-Running (MEV)

Una vez que se identifica una oportunidad con un beneficio neto positivo que supera el umbral mínimo (`MIN_PROFIT_USD`), el script pasa a la fase de ejecución.

1. **Validación Final:** Realiza una última comprobación para asegurar que el token requerido para el *flashloan* está efectivamente disponible en Aave.
2. **Simulación (`callStatic`):** Antes de enviar la transacción real, invoca la función en modo de simulación (`callStatic`). Esto permite verificar si la transacción probablemente tendría éxito sin consumir gas, previniendo fallos costosos debido a cambios repentinos en las condiciones del mercado.
3. **Estrategia de Envío Dual (Directa + *Flashbots*):** Para maximizar la probabilidad de ejecución y protegerse contra el *front-running* (una forma de *MEV*), el script emplea una estrategia de envío dual:
 - **Transacción Directa Agresiva:** Envía la transacción directamente a la *mempool* pública con una tarifa de gas muy alta (`maxFeePerGas` y `maxPriorityFeePerGas`) para incentivar a los mineros a incluirla rápidamente.
 - **Bundle de *Flashbots*:** Simultáneamente, empaqueta la misma transacción en un "bundle" y la envía al relay de *Flashbots*. Los *Flashbots* permiten a los "searchers" comunicarse directamente con los mineros, evitando la *mempool* pública y protegiendo la transacción de ser vista y explotada por *bots* de *front-running*. El bundle se envía para los siguientes 3 bloques para aumentar las posibilidades de inclusión.

```
// ...código existente...
// ESTRATEGIA 1: Envío de transacción directa
console.log("🚀 Sending transaction with aggressive gas...");
const tx = await wallet.sendTransaction(unsignedTx);
console.log(`✅ Transaction sent: ${tx.hash}`);

// ESTRATEGIA 2: MEV-BUNDLE VIA FLASHBOTS
if (flashbotsProvider) {
  try {
    console.log("📦 Enviando bundle a Flashbots...");
  } catch (error) {
    console.error("Error sending bundle to Flashbots:", error);
  }
}
// ...código existente...
// Crear bundle para los siguientes 3 bloques
for (let i = 1; i <= 3; i++) {
  await flashbotsProvider.sendBundle(bundle, blockNumber + i);
}
// ...código existente...
```

Código 3.22: Estrategia de envío dual: transacción directa con *gas* agresivo y bundle de *Flashbots* escrito en lenguaje typescript.

Esta arquitectura robusta y multifacética convierte a `arbitrage.ts` en una herramienta de grado profesional para la ejecución de arbitraje, diseñada para competir en el entorno de alta frecuencia y adversario de las finanzas descentralizadas.

3.4.2 Script de Despliegue Automatizado

El script `deploy.ts` de despliegue es una pieza fundamental que orquesta todo el proceso de inicialización del sistema en la *blockchain*. Va más allá de simplemente desplegar contratos; establece las relaciones entre ellos, configura parámetros críticos y valida que todo esté listo para operación en producción.

3.4.2.1 Arquitectura del Proceso de Despliegue

El despliegue sigue una secuencia específica y crítica que debe respetarse para garantizar la correcta funcionalidad del sistema. Esta secuencia no es arbitraria, sino que está diseñada considerando las dependencias entre contratos y la optimización de *gas*.

```
async function main() {
  console.log("🌀 Iniciando despliegue de contratos de arbitraje...");

  // Configuración inicial del deployer
  const [deployer] = await ethers.getSigners();
  const initialBalance = await deployer.getBalance();
  console.log(`💰 Balance inicial: ${ethers.utils.formatEther(initialBalance)} ETH`
);
```

Código 3.23: Función principal del script de despliegue `deploy.ts` con obtención del *deployer*.

3.4.2.2 Validación del Entorno

Antes de proceder con cualquier despliegue, el script realiza validaciones críticas del entorno. Esta fase es crucial porque un despliegue fallido puede resultar en pérdida de *gas* y, en algunos casos, contratos inutilizables.

El sistema verifica la conectividad con la red *Ethereum*, la disponibilidad de fondos suficientes para el despliegue, y la existencia de todas las dependencias necesarias. También valida que las direcciones de los protocolos externos (como Aave V3) estén correctas para la red objetivo.

```
// Validar conexión y fondos
if (initialBalance.lt(ethers.utils.parseEther("0.1"))) {
  throw new Error("❌ Fondos insuficientes para despliegue");
}

// Optimización dinámica de gas
let deployOptions: any = { gasLimit: 6000000 };
try {
```

```

const gasFeeData = JSON.parse(fs.readFileSync('./data/gasFee.json', 'utf8'));
deployOptions.maxFeePerGas =
ethers.BigNumber.from(Math.ceil(gasFeeData.maxFeePerGasGwei * 1e9).toString());
deployOptions.maxPriorityFeePerGas =
ethers.BigNumber.from(Math.ceil(gasFeeData.maxPriorityFeePerGasGwei *
1e9).toString());
} catch {
  console.warn("⚠ No se pudo leer gasFee.json, usando valores por defecto");
}

```

Código 3.24: Validación de fondos y configuración dinámica de *gas* previo al despliegue de contratos escrito con lenguaje typescript.

3.4.2.3 Despliegue Secuencial de Contratos

La secuencia de despliegue es crítica debido a las dependencias entre contratos. *DexAggregator* debe desplegarse primero porque *FlashLoanArbitrage* necesita su dirección en su constructor:

```

// Desplegar contratos en orden de dependencia
const DexAggregator = await ethers.getContractFactory("DexAggregator", deployer);
const dexAggregator = await DexAggregator.deploy(DEX_ROUTERS.UNISWAP_V3_QUOTER,
deployOptions);
await dexAggregator.deployed();
console.log("✅ DexAggregator:", dexAggregator.address);

const FlashLoanArbitrage = await ethers.getContractFactory("FlashLoanArbitrage",
deployer);
const flashLoanArbitrage = await FlashLoanArbitrage.deploy(
  AAVE_V3.POOL_ADDRESSES_PROVIDER,
  dexAggregator.address,
  deployOptions
);
await flashLoanArbitrage.deployed();
console.log("✅ FlashLoanArbitrage:", flashLoanArbitrage.address);

```

Código 3.25: Despliegue secuencial de **DexAggregator.sol** y **FlashLoanArbitrage.sol** mediante Hardhat.

3.4.2.4 Configuración de *Ownership* y Permisos

Una vez desplegados ambos contratos, se establece la relación de “*ownership*” correcta. Esto es un paso de seguridad crítico: *DexAggregator* debe ser propiedad de *FlashLoanArbitrage* para que este último pueda administrar la configuración de *DEXes*.

```

// Transferir ownership y esperar confirmación
const tx = await dexAggregator.transferOwnership(flashLoanArbitrage.address);

```

```
await tx.wait(1); // Espera confirmación en blockchain
console.log("☑ DexAggregator ownership transferido a FlashLoanArbitrage");
```

Código 3.26: Transferencia de propiedad del `DexAggregator.sol` al contrato `FlashLoanArbitrage.sol`.

Esta transferencia de *ownership* es fundamental para la seguridad del sistema. Significa que solo el contrato *FlashLoanArbitrage* puede agregar, modificar o deshabilitar *DEXes* en el agregador, evitando manipulaciones externas.

3.4.2.5 Inicialización de *DEXes*

La inicialización de los *DEXes* es probablemente la fase más compleja del despliegue. El sistema debe registrar todos los *exchanges* descentralizados soportados con sus direcciones de *router*, *quoter* (cuando aplique), y tipo específico.

```
// Inicialización de los DEXes
const owner = await flashLoanArbitrage.owner();
console.log("👑 Owner del contrato:", owner);

if (owner.toLowerCase() === deployer.address.toLowerCase()) {
  const flashLoanArbitrageWithSigner = flashLoanArbitrage.connect(deployer);
  const tx2 = await flashLoanArbitrageWithSigner.setupAllDexes({
    gasLimit: 9_000_000,
    gasPrice: deployOptions.gasPrice
  });
  await tx2.wait(1);
  console.log("☑ DEXes inicializados correctamente");
}
```

Código 3.27: Verificación de permisos e inicialización de los 27 *DEXes* mediante `setupAllDexes()`.

La función `setupAllDexes()` es crítica porque configura todas las integraciones de *DEX* en una sola transacción. Esto incluye Uniswap V2/V3/V4, SushiSwap, PancakeSwap, Curve, Balancer, y muchos otros. El alto límite de gas (9M) es necesario debido a la cantidad de operaciones de escritura en *storage*.

3.4.2.6 Persistencia de Configuración

Una vez completado el despliegue exitoso, el script actualiza automáticamente el archivo de configuración `addresses.ts` con las nuevas direcciones de contrato. Esto es crucial para que los scripts de arbitraje puedan encontrar y conectarse a los contratos desplegados.

```
// Actualizar addresses.ts

const addressesFile = fs.readFileSync(ADDRESSES_PATH, "utf8");
const newContracts =
`export const DEPLOYED_CONTRACTS = {
  DEX_AGGREGATOR: "${dexAggregator.address}",
  FLASH_LOAN_ARBITRAGE: "${flashLoanArbitrage.address}"
}`;
fs.writeFileSync(ADDRESSES_PATH, addressesFile.replace(/export const
DEPLOYED_CONTRACTS = {[^}]+\}/m, newContracts), "utf8");
```

Código 3.28: Actualización automática de `addresses.ts` con las direcciones de los contratos desplegados.

3.4.3 Recolección de Datos de Liquidez

El script `liquidity.ts` representa el sistema nervioso del proyecto, encargado de recopilar, procesar y estructurar información crítica de liquidez de múltiples *exchanges* descentralizados. Este componente es fundamental porque la calidad de los datos de liquidez determina directamente la precisión de las oportunidades de arbitraje identificadas.

3.4.3.1 Arquitectura de Recolección de Datos

La recolección de datos sigue un enfoque estratificado que prioriza la calidad sobre la cantidad. En lugar de obtener datos indiscriminadamente de todos los *DEXs* disponibles, el sistema implementa una selección inteligente basada en volumen de transacciones, liquidez total disponible y estabilidad histórica de los protocolos.

```
const topDexes = [
  'uniswap_v3',      //
  'curve',           //
  'uniswap_v2',      //
  'balancer_ethereum', //
  'sushiswap',       //
  // ... más DEXs seleccionados estratégicamente
];
```

Código 3.29: Lista priorizada de *DEXes* seleccionados por liquidez histórica en `liquidity.ts`.

Esta selección no es arbitraria; está basada en análisis histórico de datos donde se identificaron los *DEXs* con mayor probabilidad de ofrecer oportunidades de arbitraje rentables. La jerarquización considera factores como la profundidad de liquidez, la frecuencia de actualizaciones de precios, y la estabilidad de los contratos inteligentes subyacentes.

3.4.3.2 Proceso de Filtrado y Priorización

El sistema implementa múltiples capas de filtrado para asegurar que solo se procesen datos de alta calidad. Primero, verifica la disponibilidad de cada *DEX* en la red objetivo, luego evalúa la liquidez mínima requerida para considerar un *pool* como viable para arbitraje.

```
// Filtrar a los DEXes con mayor liquidez conocida
const dexesToFetch = topDexes.filter(dex => availableDexes.includes(dex));
console.log(`🔍 Consultando datos para ${dexesToFetch.length} DEXes de mayor liquidez`, dexesToFetch);
```

Código 3.30: Filtrado de *DEXes* disponibles en la red antes de la consulta de datos.

La lógica de filtrado va más allá de simplemente verificar disponibilidad. Evalúa la calidad histórica de los datos de cada *DEX*, descartando aquellos que han mostrado inconsistencias o que tienen patrones de liquidez volátiles que podrían generar señales falsas de arbitraje.

3.4.3.3 Integración con GeckoTerminal API

La integración con GeckoTerminal API es sofisticada y está diseñada para manejar las limitaciones y peculiaridades de esta fuente de datos externa. El sistema implementa estrategias de retry, manejo de rate limiting, y validación exhaustiva de la estructura de datos recibida.

```
const fetchTopPools = async (dexId: string): Promise<DexPoolsResult> => {
  const url =
    `https://api.geckoterminal.com/api/v2/networks/eth/dexes/${dexId}/pools?page=1&include=tokens`;
  try {
    const response = await fetch(url);
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }

    const result = await response.json();
    return { dex: dexId, result };
  } catch (error) {
    console.warn(`⚠️ Error fetching data for ${dexId}:`, error);
    return { dex: dexId, result: null };
  }
};
```

Código 3.31: Función `fetchTopPools()` para obtener *pools* de liquidez desde la API de GeckoTerminal.

La función de *fetch* implementa manejo robusto de errores porque la disponibilidad de datos externos puede ser inconsistente. El sistema está diseñado para continuar operando incluso si algunos *DEXs* no responden, asegurando que una falla parcial no comprometa todo el proceso de recolección.

3.4.3.4 Procesamiento y Enriquecimiento de Datos

Una vez obtenidos los datos brutos, el sistema los procesa extensivamente para extraer la información más relevante. Este procesamiento incluye normalización de formatos, cálculo de métricas derivadas, y enriquecimiento con metadatos adicionales.

```
const sortedPools = [...result.result.data].sort((a, b) => {
  const reserveA = parseFloat(a.attributes?.reserve_in_usd || '0');
  const reserveB = parseFloat(b.attributes?.reserve_in_usd || '0');
  return reserveB - reserveA;
});

// Guardar toda la información, incluido metadata completo
acc[dex] = {
  data: sortedPools,
  included: result.result.included,
  meta: result.result.meta
};
```

Código 3.32: Ordenamiento de pools por liquidez total y persistencia de metadatos completos escrito con typescript.

El ordenamiento por liquidez es crítico porque permite al sistema de arbitraje priorizar pools con mayor profundidad, donde las transacciones grandes tienen menor *impacto* en el precio. La inclusión de metadatos completos asegura que scripts posteriores tengan acceso a información contextual que puede ser crucial para la toma de decisiones.

3.4.3.5 Persistencia y Estructura de Datos

El resultado final del procesamiento se guarda en un archivo JSON estructurado (`dexespools.json`) que sirve como fuente de datos principal para el sistema de arbitraje. Esta estructura está cuidadosamente diseñada para optimizar tanto el almacenamiento como el acceso posterior.

```
console.log(` 📄 Resumen de pools encontrados:`);
for (const dex of dexesToFetch) {
  const poolCount = output[dex]?.data?.length || 0;
  if (poolCount > 0) {
    const totalLiquidity = output[dex].data.reduce((sum, pool) => {
      return sum + parseFloat(pool.attributes?.reserve_in_usd || '0');
    }, 0);
```

```

    console.log(`💰 ${dex}: ${poolCount} pools - Liquidez total:
    ${totalLiquidity.toLocaleString()}`);
  }
}

```

Código 3.33: Generación de resumen de liquidez total por *DEX* y escritura del archivo `dexespools.json`.

3.4.4 Gestión de Tarifas de Gas

En el competitivo entorno de la *blockchain* de *Ethereum*, la gestión de las tarifas de gas (*gas fees*) no es una simple optimización, sino un componente crítico para la rentabilidad. Las oportunidades de arbitraje pueden desaparecer en segundos, y una transacción que llega demasiado tarde o que es demasiado costosa puede convertir una ganancia potencial en una pérdida. El sistema aborda este desafío con un subsistema dedicado compuesto por el script `gasFee.ts` y su archivo de datos de salida, `gasFee.json`.

3.4.4.1 El Monitor de Gas en Tiempo Real

El script `gasFee.ts` actúa como un servicio de inteligencia de gas para todo el sistema. Su única responsabilidad es consultar periódicamente la red *Ethereum* para obtener una instantánea precisa de las condiciones actuales del mercado de gas y persistir esta información.

Funcionamiento Clave:

1. **Consulta Directa al Proveedor:**

Utiliza `provider.getFeeData()` y `provider.getBlock("latest")` de Ethers.js para obtener los datos más recientes sobre `baseFeePerGas`, `maxFeePerGas` y `maxPriorityFeePerGas`. Estos valores son fundamentales para construir transacciones que cumplan con el estándar EIP-1559, asegurando una inclusión más predecible en los bloques.

2. **Conversión a Gwei:** Para facilitar la lectura y el cálculo, todos los valores de gas, que se obtienen en *Wei*, se convierten a *Gwei*, la unidad más comúnmente utilizada para discutir las tarifas de gas.

3. **Persistencia Histórica:** El script no solo guarda el estado actual. Mantiene un historial de las últimas lecturas de gas (configurable a través de `MAX_HISTORY_ENTRIES`). Esto permite un análisis de tendencias y ayuda a los scripts principales a tomar decisiones más informadas, como evitar la ejecución si el gas ha subido de forma anómala en los últimos minutos.

4. **Ejecución Independiente:** Está diseñado para ser ejecutado de forma independiente, ya sea como una tarea programada (*cron job*) o como el primer paso en un proceso más grande, como el despliegue `deploy.ts`, para asegurar que las operaciones posteriores utilicen los datos de gas más frescos posibles.

```
// ...código existente...
// Función para obtener datos de gas actuales
async function getGasFeeData(): Promise<GasFeeData> {
  try {
    // Obtener datos de comisiones
    const feeData = await provider.getFeeData();
    const block = await provider.getBlock("latest");

    // ...código existente...

    // Convertir a Gwei para mejor legibilidad
    const baseFeePerGasGwei =
      parseFloat(ethers.utils.formatUnits(block.baseFeePerGas || 0, "gwei"));
    const maxFeePerGasGwei =
      parseFloat(ethers.utils.formatUnits(feeData.maxFeePerGas || 0, "gwei"));
    // ...código existente...

    return {
      // ...
      baseFeePerGasGwei,
      maxFeePerGasGwei,
      maxPriorityFeePerGasGwei,
      gasPriceGwei
    };
  } catch (error) {
    // ...código existente...
  }
}
// ...código existente...
```

Código 3.34: Función `getGasFeeData()` para consultar y *formatear* datos de gas en tiempo real.

3.4.4.2 La Base de Datos de Gas

El archivo `gasFee.json` es el resultado tangible de la ejecución de `gasFee.ts`. Actúa como una caché o una base de datos simple que desacopla la necesidad de consultar el gas de la lógica de negocio principal. Scripts como `arbitrage.ts` pueden simplemente leer este archivo localmente, una operación casi instantánea, en lugar de realizar una llamada de red que podría añadir una latencia fatal en el momento de ejecutar un arbitraje.

Estructura del Archivo:

- **lastUpdate:** Un timestamp Unix que indica cuándo se actualizó el archivo por última vez.

- **current:** Un objeto que contiene la lectura de gas más reciente, con valores tanto en Wei como en *Gwei*.
- **history:** Un array de objetos, donde cada uno representa una lectura de gas pasada, permitiendo el análisis histórico

```
{
  "lastUpdate": 1750292726,
  "lastUpdateDate": "2025-06-19T00:25:26.797Z",
  "current": {
    "timestamp": 1750292726,
    "date": "2025-06-19T00:25:26.797Z",
    "baseFeePerGas": "429166475",
    "maxFeePerGas": "2326868348",
    "maxPriorityFeePerGas": "1500000000",
    "gasPrice": "429166477",
    "baseFeePerGasGwei": 0.429166475,
    "maxFeePerGasGwei": 2.326868348,
    "maxPriorityFeePerGasGwei": 1.5,
    "gasPriceGwei": 0.429166477
  },
  "history": [
    {
      "timestamp": 1750292726,
      "date": "2025-06-19T00:25:26.797Z",
      "baseFeePerGasGwei": 0.429166475,
      "maxFeePerGasGwei": 2.326868348,
      "maxPriorityFeePerGasGwei": 1.5,
      "gasPriceGwei": 0.429166477
    }
  ]
}
```

Código 3.35: Estructura del archivo `gasFee.json` con lectura actual e historial de tarifas de *gas*.

Esta arquitectura de gestión de *gas* es un ejemplo claro de cómo el sistema está diseñado para la eficiencia y la velocidad, separando las preocupaciones y utilizando datos en caché para minimizar la latencia en las operaciones más críticas.

3.4.5 Utilidad de Seguridad

La seguridad es un pilar fundamental en el diseño de este sistema, especialmente cuando se interactúa con mecanismos avanzados como *Flashbots* para la protección contra el Valor Máximo Extraíble “MEV” (*Maximal Extractable Value* por sus siglas en inglés). El script `generate-flashbots-key.ts` es una utilidad simple pero crucial que refuerza la postura de seguridad del sistema.

Propósito y Funcionamiento:

El principio de mínima exposición dicta que una clave privada con acceso a fondos (**PRIVATE_KEY** en `.env`) nunca debe ser utilizada para más propósitos de los estrictamente necesarios. La firma de "*bundles*" de *Flashbots* no requiere que la *wallet* tenga fondos, solo que pueda firmar mensajes.

Este script aborda esa necesidad de forma segura y automatizada:

1. **Generación Segura:** Utiliza `ethers.Wallet.createRandom()` para generar una *wallet* de *Ethereum* completamente nueva y criptográficamente segura. Esta *wallet* no tiene historial y, lo que es más importante, no contendrá fondos.
2. **Aislamiento de Claves:** La clave privada de esta nueva *wallet* se destina exclusivamente para ser la **FLASHBOTS_KEY**. Esto significa que, incluso si esta clave se viera comprometida, el atacante no tendría acceso a los fondos controlados por la **PRIVATE_KEY** principal.
3. **Actualización Automática de `.env`:** Para facilitar el uso y reducir el error humano, el script lee el archivo `.env`, busca la variable **FLASHBOTS_KEY**, y la actualiza con la clave recién generada. Si la variable no existe, la añade.

```
const { ethers } = require("ethers");
const fs = require("fs");
const path = require("path");

// Generar una wallet nueva
const wallet = ethers.Wallet.createRandom();

console.log("\nNueva clave privada generada para Flashbots:");
console.log(wallet.privateKey);
console.log("\nDirección asociada:");
console.log(wallet.address);

// Actualizar automáticamente el archivo .env
const envPath = path.join(__dirname, "../.env");
let envContent = fs.readFileSync(envPath, "utf8");

// Reemplazar la línea existente o añadir una nueva
if (envContent.includes("FLASHBOTS_KEY=")) {
  envContent = envContent.replace(
    /FLASHBOTS_KEY=.*$/,
    `FLASHBOTS_KEY=${wallet.privateKey}`
  );
} else {
  envContent += `\nFLASHBOTS_KEY=${wallet.privateKey}`;
}
```

```
fs.writeFileSync(envPath, envContent);
console.log("\n☑ Clave privada guardada en .env");
```

Código 3.36: Generación de wallet dedicada para firma de *bundles* de *Flashbots* y actualización del archivo `.env`

Al ejecutar en la terminal `node scripts/generate-flashbots-key.ts`, el operador del *bot* puede establecer una clave de firma segura para *Flashbots* con un solo comando, adhiriéndose a las mejores prácticas de seguridad sin un proceso manual propenso a errores

3.4.6 Configuración Centralizada.

La mantenibilidad y la capacidad de adaptación son esenciales para la longevidad de un sistema de software complejo. El archivo `addresses.ts` es el "directorío telefónico" del proyecto, un único lugar que centraliza todas las direcciones de contratos y puntos de datos *on-chain* con los que el sistema necesita interactuar.

Esta centralización es una decisión de diseño estratégica que ofrece múltiples beneficios:

- **Fuente Única de Verdad:** Evita la dispersión de direcciones codificadas a lo largo de múltiples scripts y contratos, lo que haría que las actualizaciones fueran una pesadilla.
- **Facilidad de Mantenimiento:** Si un protocolo actualiza su *router* o si se quiere añadir un nuevo *DEX*, solo es necesario modificar este archivo.
- **Adaptabilidad entre Redes:** Aunque este proyecto se centra en Ethereum Mainnet, esta estructura facilita enormemente la adaptación a otras redes (como Base, Polygon, Arbitrum, o redes de prueba) simplemente cambiando las direcciones en este archivo.

3.4.6.1 Estructura y Componentes:

El archivo está organizado en secciones lógicas y exporta constantes que son importadas por el resto del sistema.

- **FACTORIES y DEX_ROUTERS:** Contienen las direcciones de los contratos de fábrica y de *router* para una vasta gama de *DEXes*. Esta lista exhaustiva es el resultado de una investigación profunda del ecosistema *DeFi* y es lo que permite al `DexAggregator.sol` ser tan versátil.
- **DEPLOYED_CONTRACTS:** Esta es una sección dinámica y de vital importancia. Está diseñada para ser actualizada automáticamente por el script de despliegue (`deploy.ts`). Después de que `FlashLoanArbitrage.sol` y `DexAggregator.sol` son desplegados, sus nuevas direcciones se escriben aquí. Esto cierra el ciclo de automatización, permitiendo que

los scripts de arbitraje sepan inmediatamente a qué contratos deben conectarse sin intervención manual.

- **AAVE_V3 y AAVE_TOKENS:** Define las direcciones del *pool* de Aave V3 y una lista curada de tokens que están disponibles para préstamos *flash*, convenientemente agrupados por su número de decimales para facilitar los cálculos de precisión.
- **CHAINLINK_FEEDS:** Proporciona las direcciones de los oráculos de precios de Chainlink, utilizados por el script de arbitraje para obtener un precio fiable de ETH/USD, crucial para calcular la rentabilidad neta de las operaciones.

```
//-----  
// DEX ROUTER ADDRESSES  
// -----  
export const DEX_ROUTERS = {  
  // Uniswap  
  UNISWAP_V2: "0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D",  
  UNISWAP_V3: "0xE592427A0AEce92De3Edee1F18E0157C05861564",  
  // ...código existente...  
};  
  
// Deployed Contracts  
// ESTA SECCIÓN ES ACTUALIZADA AUTOMÁTICAMENTE POR deploy.ts  
export const DEPLOYED_CONTRACTS = {  
  DEX_AGGREGATOR: "0x0Ee29547347E41fD7Bf14a0d064421275C2ae0bA",  
  FLASH_LOAN_ARBITRAGE: "0xCC4699e54c5D98b82702305e63835B50e6B5Bfb5"  
};  
  
// Aave V3 System Contracts  
export const AAVE_V3 = {  
  POOL_ADDRESSES_PROVIDER: "0x2f39d218133AFaB8F2B819B1066c7E434Ad94E9e",  
  POOL: "0x794a61358D6845594F94dc1DB02A252b5b4814aD"  
};  
  
//Chainlink Feed Addresses  
export const CHAINLINK_FEEDS = {  
  ETH_USD: "0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419",  
};  
// ...código existente...
```

Código 3.37: Extracto del archivo `addresses.ts` con las direcciones centralizadas de *DEXes*, contratos desplegados, Aave V3 y oráculos de Chainlink.

En resumen, `addresses.ts` es el mapa que guía a todo el sistema a través del complejo paisaje de la *blockchain*, asegurando que cada componente se comuniquen con el destino correcto de manera consistente y mantenible.

3.5 Flujo de Trabajo Operativo y Ciclo de Vida del Sistema

Esta sección detalla el ciclo de vida completo del sistema de arbitraje, desde la configuración inicial del entorno hasta la ejecución y el monitoreo de operaciones. El flujo está diseñado para ser secuencial y robusto, asegurando que cada componente esté correctamente preparado antes de pasar a la siguiente fase.

3.5.1 Configuración Inicial del Entorno

Estos pasos preparan el entorno de desarrollo y las credenciales necesarias para interactuar con la red *Ethereum* y servicios de terceros.

3.5.1.1 Configuración del Entorno

Crear y configurar el archivo `.env` en la raíz del proyecto:

- **Descripción:** Este archivo contiene claves y configuraciones sensibles que no deben ser públicas:
 - **PRIVATE_KEY:** La clave privada de la *wallet*, usada para firmar y enviar transacciones (despliegue, ejecución de arbitraje).
 - **RPC_URL:** La URL del nodo de *Ethereum* (ej. Alchemy o Infura), necesaria para comunicarse con la *blockchain*.
 - **API_KEY:** Tu clave de API para Etherscan, usada para verificar los contratos desplegados.
 - **FLASHBOTS_KEY:** Clave privada dedicada para firmar "*bundles*" de *Flashbots*.

3.5.1.2 Generación de Clave para *Flashbots*

- **Comando:** `node scripts/generate-flashbots-key.ts`
- **Descripción:** Este script crea una nueva *wallet* de *Ethereum* de forma aleatoria y guarda su clave privada en el archivo `.env` como "**FLASHBOTS_KEY**". *Flashbots* es un servicio que permite enviar transacciones directamente a los mineros, evitando el "*mempool*" público. Esto es crucial para proteger las operaciones de arbitraje de *bots* maliciosos (*front-running*).

3.5.1.3 Despliegue de Contratos Inteligentes

- **Comando:** `npx hardhat run deploy.ts --network localhost`

- **Descripción:** Este es el script principal para poner en marcha la infraestructura *on-chain*. Realiza varias tareas en secuencia:
 1. **Ejecuta `gasFee.ts`:** Primero, obtiene las tarifas de gas actuales para asegurar que el despliegue se realice con un costo óptimo.
 2. **Despliega `DexAggregator.sol`:** Este contrato actúa como una interfaz unificada para interactuar con los diferentes tipos de *DEXes* (Uniswap V2, V3, etc.).
 3. **Despliega `FlashLoanArbitrage.sol`:** Este es el cerebro de la operación. Contiene la lógica para:
 - Solicitar un préstamo *flash* a Aave.
 - Ejecutar los intercambios (swaps) en los *DEXes* a través del *DexAggregator*.
 - Repagar el préstamo *flash* más la comisión, todo dentro de una misma transacción atómica.
 4. **Transfiere la Propiedad:** El script transfiere la propiedad (*ownership*) del *DexAggregator* al contrato *FlashLoanArbitrage*. Esto es necesario para que el contrato de arbitraje pueda llamar a las funciones del agregador.
 5. **Actualiza Direcciones:** Guarda las direcciones de los contratos recién desplegados en `addresses.ts` para que otros scripts puedan localizarlos fácilmente.
 6. **Inicializa los DEXes:** Al final, ejecuta automáticamente el script `setup-dexes.ts`.

3.5.1.4 Configuración de los DEXes en el Contrato

- **Comando:** `npx hardhat run setup-dexes.ts --network localhost`
- **Descripción:** Este script llama a la función `setupAllDexes` en el contrato *FlashLoanArbitrage*. Esta función registra las direcciones de los *routers* de cada *DEX* (Uniswap, Sushiswap, etc.) en el contrato, permitiéndole saber a dónde dirigir las operaciones de intercambio.

3.5.2 Recopilación de Datos

Para encontrar oportunidades, el *bot* necesita datos actualizados sobre precios, liquidez y disponibilidad de préstamos. Estos scripts se encargan de obtener y almacenar esa información.

1. Obtener Datos de Pools y Liquidez

- **Comando:** `npx hardhat run scripts/liquidity.ts`

- **Descripción:** Este script se conecta a una API externa (GeckoTerminal) para obtener una lista completa de los pools de liquidez de los principales *DEXes* en *Ethereum*. Para cada *pool*, extrae información vital como los tokens que lo componen, sus precios y la liquidez total en USD. El resultado se guarda en el archivo `dexespools.json`. Este archivo es la fuente principal de datos para encontrar diferencias de precios.

3.5.3 Monitoreo y Ejecución del Arbitraje

Esta es la fase operativa donde el bot se ejecuta, analiza los datos y, si encuentra una oportunidad rentable, la ejecuta.

1. Ejecución del Bot de Arbitraje

- **Comando:** `npx hardhat run arbitrage.ts --network localhost`

Este es el script principal que orquesta todo el proceso de arbitraje. Su flujo de ejecución es el siguiente:

- **Inicialización (`initialize`):**
 1. Configura la *wallet* del usuario y los proveedores de conexión a la *blockchain* (Alchemy) y a *Flashbots*.
 2. Carga en memoria los datos previamente recopilados de los archivos `.json` (`dexespools.json`).
 3. Obtiene las tarifas de gas en tiempo real.
- **Monitoreo (`monitor`):**
 1. **Extraer Precios (`extractPrices`):** Procesa los datos de `dexespools.json` para crear una lista estructurada de precios de cada par de tokens en cada *DEX*.
 2. **Buscar Oportunidades (`findArbitrageOpportunities`):** Itera sobre la lista de precios, comparando el precio de un mismo par de tokens entre diferentes *DEXes*. Si ($PrecioVenta > PrecioCompra$), lo identifica como una oportunidad potencial.
 3. **Analizar Rentabilidad (`processOpportunities`):** Para cada oportunidad, calcula el beneficio neto restando los costos estimados:
 - **Costo de Gas:** La tarifa de la transacción en la red.
 - **Comisión del Préstamo *Flash*:** Aave cobra una pequeña comisión (0.05%).
 - Filtra las oportunidades que no superan el umbral de beneficio mínimo (`MIN_PROFIT_USD`).
 - Verifica que el token necesario esté disponible para un préstamo *flash* según los datos de Aave.

- **Ejecución de la Operación (`executeFlashLoan`):**
 1. Si se encuentra una oportunidad rentable y la ejecución está habilitada (`IS_EXECUTION_ENABLED = true`), procede a ejecutarla.
 2. **Simulación (Opcional):** Primero realiza una llamada estática (`callStatic`) para simular la transacción y asegurarse de que no fallará.
 3. **Construcción del *Bundle de Flashbots*:** Prepara la transacción que llamará a la función `executeFlashLoanSimple` de tu contrato. En lugar de enviarla al *mempool* público, la empaqueta en un "*bundle*" de *Flashbots*.
 4. **Envío Directo a Mineros:** Envía el *bundle* a través del *relay* de *Flashbots*. Esto garantiza que la transacción sea privada y se ejecute sin riesgo de ser interceptada. Si *Flashbots* falla, el script tiene un mecanismo de respaldo para enviar una transacción normal con alta prioridad.
- **Confirmación y Registro:** Espera a que la transacción se confirme en la *blockchain* y registra el resultado, mostrando el cambio final en el balance de la *wallet*.

Esta sección describe el ciclo de vida completo para la configuración, despliegue, ejecución y mantenimiento del sistema de arbitraje. El flujo de trabajo está diseñado para ser metódico, asegurando que todos los componentes estén correctamente inicializados antes de iniciar las operaciones en vivo o simuladas.

3.6 Seguridad y Mitigación de Amenazas

3.6.1 Principales riesgos en Contratos Inteligentes *DeFi*

Los contratos inteligentes desplegados en entornos *DeFi* están expuestos a múltiples vectores de ataque que pueden comprometer tanto la funcionalidad como la integridad económica del sistema. En este proyecto se identificaron y abordaron los siguientes riesgos principales:

1. **Ataques de *Reentrancy*:** Aprovechamiento de llamadas externas no protegidas para reingresar a funciones críticas
2. **Manipulación de precios:** Alteración temporal de precios en pools de baja liquidez
3. ***Slippage* excesivo:** Pérdidas por diferencias entre precios esperados y ejecutados
4. **Control de acceso deficiente:** Ejecución no autorizada de funciones administrativas
5. **Ataques de *gas griefing*:** Consumo excesivo de gas para generar fallos en transacciones
6. ***Frontrunning* y *sandwich attacks*:** Explotación del orden de ejecución de transacciones

El diseño del sistema consideró estos aspectos de seguridad desde la fase inicial de arquitectura, integrando patrones de defensa reconocidos por la comunidad *Ethereum* y herramientas de auditoría para su mitigación efectiva.

3.6.2 Prevención de Ataques de Reentrada

La principal vulnerabilidad del contrato inteligente `FlashLoanArbitrage.sol` es un ataque de reentrada, donde un atacante puede explotar llamadas externas para reingresar a funciones críticas antes de que se complete su ejecución. El sistema implementa múltiples capas de protección.

3.6.2.1 Implementación del Patrón *ReentrancyGuard*

El contrato `FlashLoanArbitrage.sol` fue protegido contra ataques de reentrada mediante múltiples capas de seguridad. La primera línea de defensa utiliza el modificador `nonReentrant` del módulo `ReentrancyGuard` de OpenZeppelin, asegurando que las funciones críticas no puedan ser reingresadas antes de completar su ejecución.

```
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract FlashLoanArbitrage is Ownable, ReentrancyGuard,
IFlashLoanSimpleReceiver {
    function executeOperation(
        address asset,
        uint256 amount,
        uint256 premium,
        address initiator,
        bytes calldata params
    ) external override nonReentrant returns (bool) {
        // lógica del arbitraje protegida
    }
}
```

Código 3.38: Aplicación del modificador `nonReentrant` de OpenZeppelin a la función `executeOperation()`.

Este patrón evita que un atacante, aprovechando una llamada externa mal protegida, pueda reingresar al contrato antes de que las variables de estado hayan sido actualizadas, evitando “*exploits*”.

Este fragmento de código muestra cómo el contrato hereda de `ReentrancyGuard` y aplica el modificador `nonReentrant` a la función crítica `executeOperation()`. El modificador establece un *flag* interno que previene llamadas recursivas, revirtiendo cualquier intento de reentrada antes de que la función original complete su ejecución.

3.6.2.2 Guard Personalizado para *Flash Loans*

Adicionalmente, se implementó un mecanismo de protección específico para operaciones de *flash loan* mediante la variable booleana `_flashLoanReentrancyGuard`:

```
bool private _flashLoanReentrancyGuard;

function executeOperation(...) external override returns (bool) {
    require(!_flashLoanReentrancyGuard, "Reentrant call");
    _flashLoanReentrancyGuard = true;

    // Lógica de arbitraje

    _flashLoanReentrancyGuard = false;
    return true;
}
```

Código 3.39: Implementación del *guard* personalizado `_flashLoanReentrancyGuard` dentro de `executeOperation()`.

Esta implementación dual asegura que tanto las funciones públicas como las operaciones específicas de *flash loan* estén protegidas contra ataques de *reentrancy*, proporcionando una defensa robusta.

3.6.3 Control de Accesos y Permisos

3.6.3.1 Control de Accesos y Gestión de Permisos

El sistema implementa un control de acceso estricto basado en el patrón Ownable de OpenZeppelin, restringiendo las operaciones administrativas críticas únicamente al propietario del contrato.

```
modifier onlyOwner() {
    require(msg.sender == owner(), "Not owner");
    _;
}
```

Código 40: Modificador `onlyOwner()` para restringir funciones administrativas al propietario del contrato.

Todos los contratos del sistema utilizan el módulo Ownable de OpenZeppelin para restringir las funciones sensibles únicamente al operador del sistema. Este patrón fue utilizado para operaciones como:

- Actualización de direcciones críticas
- Activación o pausa del contrato
- Cambio de parámetros de ejecución como *slippage*, beneficio mínimo, etc.

3.6.3.2 Validación de Origen de Llamadas

El sistema valida que las llamadas a `executeOperation()` provengan exclusivamente del *pool* de Aave autorizado:

```
require(msg.sender == address(POOL), "Caller must be Aave Pool");
require(initiator == address(this), "Initiator must be this contract");
// Inicialización de Dexes
```

Código 3.41: Validaciones de origen en `executeOperation()`: verificación del *pool* de Aave y del iniciador de la transacción.

Esta validación asegura que solo el protocolo Aave pueda invocar la lógica de arbitraje, previniendo ataques externos que intenten explotar la función directamente.

3.6.3.3 Protección contra *Slippage* Programado por Atacantes

Una amenaza más sofisticada es cuando un atacante proporciona liquidez falsa para simular oportunidades rentables. Esto puede suceder al inflar momentáneamente las reservas de un *pool*. Para evitarlo:

- Se consulta el histórico reciente de volumen en el par y se ignoran tokens con movimientos anómalos.
- Se requiere que el precio entre dos bloques consecutivos sea estable antes de ejecutar.
- Se calcula un promedio móvil de precios, no solo el *snapshot* actual.

A diferencia de proyectos simples que solo enfocan la seguridad en el contrato inteligente, este sistema adopta una visión integral de amenazas, abordando tanto el código como el contexto operativo. La implementación de capas de defensa, desde validación de precios hasta redundancia en oráculos y simulaciones pre-ejecución, permite mitigar la mayoría de los riesgos críticos que amenazan la viabilidad y estabilidad de un sistema de arbitraje basado en *flash loans*.

3.6.3.4 Oráculos de Chainlink

El sistema utiliza oráculos de Chainlink para obtener precios de referencia y validar la consistencia de precios entre diferentes fuentes:

```
async function getEthPriceFromChainlink(): Promise<number> {
  const chainlinkContract = new ethers.Contract(
    ETH_USD_FEED_ADDRESS,
    chainlinkAggregatorABI,
```

```

        provider
    );
    const [, answer] = await chainlinkContract.latestRoundData();
    return parseFloat(ethers.utils.formatUnits(answer, 8));
}

```

Código 3.42: Función `getEthPriceFromChainlink()` para obtener el precio de ETH/USD desde el oráculo de Chainlink.

3.6.3.5 Validación de Montos y *Slippage* en Tiempo de Ejecución

El sistema implementa validaciones dinámicas de *slippage* y montos para proteger contra pérdidas imprevistas durante la ejecución de swaps. Estas validaciones se realizan tanto a nivel de contrato como en los scripts *off-chain*.

- **Cálculo dinámico de *Slippage***

En el contrato `DexAggregator.sol`, cada ruta de swap se valida en tiempo de ejecución mediante la estimación esperada de salida `amountOutMin` y la cantidad real obtenida del intercambio. Si el resultado está por debajo del mínimo esperado, la transacción se revierte automáticamente.

Esto evita que swaps poco eficientes, errores de cálculo o manipulaciones inesperadas consuman gas y fondos sin generar beneficio.

```

require(amountOut >= amountOutMin, "Slippage exceeded");

```

Código 3.43: Validación de *slippage* en el contrato `DexAggregator.sol` durante la ejecución de cada intercambio de tokens.

3.6.4 Simulación de Ataques Comunes

Durante el proceso de desarrollo, se simularon los siguientes ataques para asegurar la resistencia del sistema:

- Ataque de *reentrancy* en `executeOperation()`
- Manipulación de liquidez antes del swap
- Transacciones *frontrunning* antes de la ejecución del arbitraje
- *Gas griefing* con sobregasto para revertir operación

Cada simulación fue ejecutada en entorno *forkeado* con Hardhat + Mainnet, utilizando cuentas generadas y manipulando las condiciones de los pools. Las protecciones implementadas abortaron correctamente todas las operaciones anómalas.

3.6.5 Ataques de Reordenamiento de Transacciones (*Front-running* y *MEV*)

Los sistemas automatizados de arbitraje en *Ethereum* operan en entornos extremadamente competitivos, volátiles y plagados de amenazas tanto técnicas como estratégicas. Los principales vectores de ataque o sabotaje para este tipo de sistemas incluyen:

- **Ataques *MEV* (Maximum Extractable Value):** *Bots* especializados que monitorean el *mempool* para copiar transacciones rentables con mayor prioridad de gas
- **Sandwich Attacks:** Estrategia donde un atacante coloca transacciones antes y después de la operación objetivo para extraer valor.
- ***Front-running*:** Ejecución anticipada de transacciones similares con mayor comisión de gas
- **Manipulación temporal de precios:** Alteración artificial de precios en pools de baja liquidez.
- **Ataques de reentrada:** Explotación de *callbacks* durante la ejecución de *flash loan*.
- **Fallas de infraestructura:** Interrupciones en nodos RPC, oráculos de precios o congestión de red.

El sistema desarrollado fue diseñado con contramedidas explícitas para mitigar cada una de estas amenazas.

3.6.6 Protección contra Frontrunning y *Bots MEV*

En entornos donde los *bots* monitorean el *mempool* constantemente, cualquier transacción visible que implique una ganancia considerable puede ser interceptada o replicada con una mayor comisión de gas. Para mitigar esta amenaza se implementaron las siguientes estrategias:

- ***Bundle execution* en testeo con *Flashbots* RPC**
- **Limitación del tiempo de exposición de oportunidades,** ejecutando el arbitraje inmediatamente después del escaneo.
- **Filtro de oportunidad basado en margen ajustado,** que excluye oportunidades con margen bajo que serían fácilmente capturadas por *bots MEV*.

Adicionalmente, el sistema evalúa el costo de gas estimado contra el beneficio. Si la operación genera menos de 1.5x el gasto en gas, se descarta automáticamente, asumiendo que podría ser superada por un *bot* con prioridad de inclusión.

3.6.7 Estrategia de Seguridad Operacional y Validación ante Default

Más allá de los contratos, también se definió una política de seguridad para el entorno operativo:

- Llaves privadas cifradas localmente y nunca expuestas en el código fuente.
- Separación entre entorno de desarrollo (testnet) y entorno de producción (mainnet).
- Control de versiones del archivo `.env`, con doble autenticación para operadores del sistema.

Un aspecto crucial es asegurar que, pase lo que pase durante la operación, el préstamo *flash* se devuelva al final del bloque. Para esto:

- El contrato implementa una lógica condicional que verifica que la ganancia neta sea positivo antes del *transferBack* del préstamo.
- Si ocurre cualquier error en la cadena de swaps o no se cumple el beneficio mínimo, la transacción completa se revierte sin afectar la liquidez del *pool* de Aave.

Fragmento crítico:

```
require(profit > minProfit, "Not enough profit");
IERC20(asset).transfer(pool, amount + premium);
```

Código 3.44: Verificación de rentabilidad mínima y repago del préstamo al pool de Aave.

3.7 Pruebas, simulación e integración

3.7.1 Pruebas Unitarias

Las pruebas unitarias fueron diseñadas como la primera línea de verificación lógica del sistema de arbitraje. Estas pruebas permiten validar que cada función aislada de cada contrato inteligente se comporte exactamente como se espera bajo condiciones controladas. Para ello, se utilizó el framework Hardhat, junto con las bibliotecas ampliamente utilizadas en el entorno de desarrollo *Ethereum* por su capacidad para definir y estructurar pruebas automatizadas legibles y confiables.

El enfoque seguido fue el de “*test-driven development*” (TDD) parcial, asegurando que las funciones críticas como `executeFlashLoan()`, `executeOperation()` y `swap()` en el contrato `DexAggregator.sol`, tuvieran una cobertura mínima aceptable. La lógica de estas funciones involucra múltiples validaciones: control de acceso, verificación de parámetros de entrada, y condiciones de éxito/fallo, todo lo cual fue testado con casos reales y simulados.

Prueba unitaria con Hardhat:

```
it("should execute flashloan and return true", async () => {
```

```
const tx = await flashLoanArbitrage.executeFlashLoan(tokens, amounts, params);
await expect(tx).to.emit(flashLoanArbitrage, "FlashLoanExecuted");
});
```

Código 3.45: Prueba unitaria con Hardhat para verificar la emisión del evento `FlashLoanExecuted` tras un préstamo relámpago.

Esta prueba asegura que la ejecución del préstamo *flash* se emite correctamente y lanza un evento esperado, validando así tanto la ejecución como su rastreo en la *blockchain*.

Adicionalmente, se probaron casos extremos como entradas vacías, tokens no compatibles, y errores de lógica intencionales para verificar que los `require()` definidos abortaran adecuadamente la ejecución sin consumir gas innecesario. El objetivo era garantizar la robustez del sistema frente a llamadas inválidas, uso indebido o condiciones inesperadas durante la operación.

3.7.2 Simulación del Contrato en Red Local y con Forks

3.7.2.1 Ventajas del Enfoque de Forkeo

Para observar el comportamiento del sistema en condiciones cercanas al entorno real, se empleó la funcionalidad de “forkeo de red” de Hardhat, apuntando directamente a la red *Ethereum Mainnet* mediante una clave API de Alchemy.

Este enfoque permitió replicar el estado completo de la red principal en un entorno local, incluyendo balances de tokens, contratos existentes y pools de liquidez. De esta manera, se realizaron pruebas directamente sobre datos reales sobre las direcciones de los contratos

- Configuración del fork en Hardhat:

```
const config: HardhatUserConfig = {
  networks: {
    hardhat: {
      forking: {
        url: process.env.RPC_URL || "https://eth-mainnet.g.alchemy.com/v2/[API_KEY]",
        blockNumber: undefined // Utiliza el último bloque disponible
      },
      accounts: {
        count: 10,
        initialIndex: 0,
        accountsBalance: "100000000000000000000" // 10,000 ETH por cuenta
      }
    }
  }
}
```

```
}  
}  
};
```

Código 3.46: Configuración de `hardhat.config.ts` para bifurcación (*fork*) de Ethereum Mainnet en pruebas locales.

Se configura el archivo `hardhat.config.ts` para:

- Ejecutar pruebas con tokens como sin necesidad de usar la red real.
- Verificar la viabilidad de oportunidades reales detectadas por el sistema.
- Evaluar las rutas de arbitraje usando las reservas reales de los *DEXes*.
- Validar que el comportamiento de contratos como Aave o Uniswap se mantenía conforme a lo esperado.

Durante estas pruebas se registró, por ejemplo, que algunas oportunidades detectadas con base en precios teóricos se revertían cuando se ejecutaban, ya que el *slippage* o la falta de profundidad en los pools invalidaba el arbitraje. Esto resaltó la importancia del *forkeo* como mecanismo de validación final antes de cualquier despliegue en producción.

3.7.2.2 Ventajas del Enfoque de Forkeo

Esta metodología proporcionó beneficios significativos para la validación del sistema:

1. **Acceso a Datos Reales:** Utilización de balances actuales, precios de mercado y liquidez real de protocolos como Uniswap V2/V3, SushiSwap, Curve y Balancer.
2. **Validación de Integraciones:** Verificación de la compatibilidad con contratos desplegados en producción, incluyendo Aave V3 y oráculos de Chainlink.
3. **Evaluación de Costos Reales:** Análisis preciso de *gas fees* y *slippage* bajo condiciones de mercado actuales.
4. **Detección de Arbitrajes Viables:** Identificación de oportunidades reales utilizando datos de liquidez en tiempo real.

Además, se consideraron costos operativos, incluyendo tarifas de gas en bloques congestionados, y se ajustó la lógica del sistema para solo ejecutarse cuando los beneficios netos superaran los costos esperados.

3.7.3 Simulaciones previas a la Ejecución

Antes de enviar cualquier transacción a la red principal, el sistema realiza simulaciones completas utilizando el mecanismo `callStatic` de `ethers.js`:

```
try {
  console.log("🔧 Simulating transaction execution...");
  await flashLoanContract.callStatic.executeFlashLoanSimple(
    opportunity.flashLoanToken,
    flashLoanAmount
  );
  console.log("✅ Simulation successful!");
} catch (simError: any) {
  console.log(`❌ Simulation failed: ${simError.message}`);
  // Continuar con cautela o abortar según el error
}
```

Código 3.47: Simulación pre-ejecución mediante `callStatic` para verificar la viabilidad de la transacción sin consumir *gas*.

La función `callStatic` permite ejecutar la transacción sin modificar el estado de la *blockchain*, proporcionando una vista previa exacta de si la operación será exitosa. Esto permite detectar errores potenciales como falta de liquidez, problemas de *slippage* o fallos en los swaps antes de incurrir en costos de *gas*.

3.7.4 Pruebas de Integración y Flujo Simulado

Una vez validados los componentes individuales a través de pruebas unitarias, el siguiente paso crítico fue la realización de pruebas de integración. El propósito fundamental de esta fase fue verificar la correcta orquestación y comunicación entre los distintos módulos del sistema en un entorno que simula fielmente las condiciones de la red principal de *Ethereum*. A diferencia de las pruebas unitarias, que se centran en la lógica aislada de una función, las pruebas de integración se enfocan en el flujo completo de una operación de arbitraje.

El objetivo era confirmar que los scripts *off-chain*, los contratos inteligentes desplegados, los oráculos de precios, los proveedores de datos de *gas* y los protocolos externos como Aave V3 interactuaban de manera coherente y predecible. Específicamente, se buscó validar los siguientes puntos críticos:

- **Interacción con el Protocolo Aave:** La capacidad del contrato `FlashLoanArbitrage.sol` para solicitar un préstamo *flash*, recibir los fondos y ejecutar la lógica de *callback* (`executeOperation`) correctamente.

- **Agregación de DEXes:** El funcionamiento del *DexAggregator* para enrutar los swaps a través de los *exchanges* descentralizados correctos (Uniswap, SushiSwap, etc.) según la oportunidad detectada.
- **Validación de Rentabilidad *Off-chain*:** La precisión del script `arbitrage.ts` para calcular la rentabilidad, descontando la prima del préstamo *flash* y los costos de gas estimados, antes de iniciar cualquier transacción *on-chain*.
- **Ciclo Completo de la Transacción:** La ejecución atómica de la secuencia completa: *préstamo -> swap de compra -> swap de venta -> devolución del préstamo con prima -> transferencia del beneficio neto al monedero del operador*.

Para lograr un alto grado de realismo sin arriesgar capital real, todas las pruebas de integración se llevaron a cabo en un entorno de red local que era una bifurcación (*fork*) de la *blockchain* Ethereum Mainnet. Esta configuración, habilitada a través de Hardhat, permitió que nuestros contratos interactuaran con las versiones desplegadas y el estado actual de los contratos de Aave, Uniswap, Curve y otros protocolos *DeFi*.

Configuración del Entorno:

- **Red:** Hardhat con *forking* habilitado desde un nodo de Alchemy apuntando a *Ethereum* Mainnet, fijado en un bloque específico (**blockNumber: 19600000**) para garantizar la reproducibilidad de las pruebas.
- **Ejecución:** El script principal de monitoreo se lanzó mediante el siguiente comando, que instruye a Hardhat a utilizar la configuración de red local (el *fork*).

```
npx hardhat run scripts/arbitrage.ts --network localhost
```

- **Datos de Gas:** Se utilizó la función `getFeeData()` de **Ethers.js** para obtener precios de gas dinámicos y realistas, en lugar de valores fijos.
- **Oráculos:** Se integraron lecturas directas del oráculo de precios de Chainlink para ETH/USD como una fuente de verdad externa para la validación de costos.
- **Préstamo *Flash*:** Se configuró para solicitar préstamos principalmente en WETH, USDC y DAI desde el *pool* de Aave V3, utilizando la función `executeFlashLoanSimple` para optimizar el consumo de gas.

Secuencia del Flujo Probado: El script `arbitrage.ts` orquestó una secuencia automatizada de pasos:

1. **Recopilación de Datos:** El script primero cargó los datos de los pools de liquidez desde un archivo `dexespools.json` previamente generado, que contenía información de más de 30 DEXes.
2. **Análisis de Oportunidades:** Analizó más de 480 puntos de precio para identificar discrepancias entre los DEXes. La función `findArbitrageOpportunities` filtró estas oportunidades, conservando solo aquellas con una rentabilidad bruta superior al umbral configurado (ej. 0.5%).
3. **Validación de Rentabilidad Neta:** Para cada oportunidad potencial, se calculó el costo estimado de gas y la prima del préstamo *flash*. Solo las oportunidades con un beneficio neto positivo (`netProfitUSD > MIN_PROFIT_USD`) fueron consideradas para ejecución.
4. **Revalidación de Precios:** Justo antes de enviar la transacción, el sistema realizaba una última comprobación de precios para protegerse contra la volatilidad del mercado (*front-running* o cambios naturales). Si la rentabilidad había desaparecido, la operación se abortaba.
5. **Ejecución Atómica:** Si la oportunidad seguía siendo viable, se llamaba a la función `executeFlashLoan()` del contrato, que iniciaba la transacción atómica en la *blockchain*.
6. **Confirmación y Registro:** El script esperaba la confirmación de la transacción y registraba el resultado, ya fuera exitoso o fallido, junto con el hash de la transacción y el gas consumido.

3.7.5 Despliegue y Verificación en Entornos de Desarrollo y Producción

El despliegue de los contratos se realizó utilizando Hardhat y se integró el **plugin hardhat-etherscan** para verificar la correcta publicación de los contratos en Etherscan. Se configuró estrictamente la separación de claves privadas y *endpoints* en archivos `.env`, garantizando la seguridad de las credenciales. Entre las consideraciones adoptadas para el despliegue se incluyen:

Transferencia de Ownership y Roles Administrativos: Tras el despliegue de los contratos (por ejemplo, `FlashLoanArbitrage.sol` y `DexAggregator.sol`), se efectuó la transferencia del *ownership* al contrato de arbitraje, para centralizar el control de ejecución.

Inicialización de DEXes: Una vez desplegados, se ejecutó un script de configuración (`setup-dexes.ts`) que registra los diferentes DEXes utilizados, asignándoles índices específicos y realizando las aprobaciones necesarias para los swaps.

Integración con Flashbots: Para mitigar riesgos de *front-running* y la explotación por MEV, se preparó la infraestructura para integrar *bundles* de transacciones a través de *Flashbots*, utilizando claves privadas separadas para firmar estas operaciones.

3.7.6 Herramientas para simulación y Pruebas

Además del entorno *forkeado* en Hardhat, se realizaron pruebas en *testnets* como “Sepolia”, lo que permitió evaluar la estabilidad y rendimiento en condiciones de red en entornos reales pero con menores riesgos y costos de gas. Estas pruebas incluyeron:

- Ejecución de transacciones de arbitraje en condiciones de gasto real (aunque menores a las de Mainnet).
- Validaciones cruzadas entre escenarios simulados y en *testnets*, facilitando la identificación de discrepancias en la lógica de negocio.
- Uso de herramientas de cobertura de pruebas, como *solidity-coverage*, para asegurar que la totalidad del código crítico contara con una cobertura de *test* mínima aceptable.

3.7.7 Manejo de Logs y Monitoreo

La trazabilidad y auditoría son componentes fundamentales del sistema de arbitraje. Para ello, se diseñó una arquitectura de *logging* que, en cada ejecución *off-chain*, registra de forma estructurada y detallada la operación, incluyendo:

- **Datos de Transacción:** *Hash*, dirección del contrato ejecutado y bloque en que se confirmó la transacción.
- **Rutas de Swap:** Información sobre los *DEXes* involucrados, rutas utilizadas y direcciones de los tokens operados.
- **Detalles Financieros:** Cantidades de tokens intercambiados, precios de compra y venta, costos estimados de gas, primas de *flash loan (FL Fee)* y cálculo del beneficio bruto y neto.
- **Otros Indicadores:** *Slippage* reportado, datos en tiempo real de la liquidez y balances pre y post ejecución.

Ejemplo de salida de logs:

```
● MEJORES OPORTUNIDADES DE ARBITRAJE:  
[1] SHX/WETH: 0.78% beneficio  
Comprar en SushiSwap V3 a 0.000003479491083086546  
Vender en SushiSwap a 0.000003566699325653918  
Beneficio bruto: $2.99  
Costos: Gas $2.20, Prima FL $0.24  
Beneficio neto: $0.55
```

Código 3.48: Ejemplo de *output* del Sistema logging tras correr los scripts mostrando una oportunidad de arbitraje mostrando los costos y beneficio neto.

Se habilitó la posibilidad de registrar estos logs en archivos tanto en formato “.log” como en “.json”, permitiendo una auditoría exhaustiva y facilitando la depuración y análisis en fases posteriores del proyecto. Este sistema de monitoreo resulta esencial para la evaluación continua del rendimiento y la estabilidad del modelo de arbitraje, permitiendo iterar y optimizar la lógica de selección de rutas y la gestión de riesgos en función de datos reales extraídos tanto del entorno “*forkeado*” como de *testnets*.

CAPÍTULO 4: RESULTADOS

Se documentaron 3 escenarios recurrentes durante las pruebas de ejecución realizadas con Hardhat que representan el espectro completo de resultados posibles en un entorno de pruebas de arbitraje en la blockchain de Ethereum con los 27 DEXes utilizados.

4.1 Resultado 1: Ejecución Exitosa

Este escenario representa el caso ideal donde todo el sistema funciona como se esperaba. Se detectó una oportunidad, se validó como rentable y se ejecutó con éxito.

```
↑ MEJORES OPORTUNIDADES DE ARBITRAJE:
TOP
[1] SHX/WETH: 0.78% beneficio
    Comprar en SushiSwap V3 a 0.0000034794109830086546
    Vender en SushiSwap a 0.00000356699325653918
    Beneficio neto: $0.55
...
⚡ Ejecutando arbitraje de préstamo flash para la mejor oportunidad...
✅ Transacción enviada:
0x2835d38ddbfb25abaaf54a8897b4601167adec9716239d2013b5ce82d005d303
✅ Transacción confirmada en bloque 22584625
```

Nota: Validado en etherscan:

<https://etherscan.io/tx/0x2835d38ddbfb25abaaf54a8897b4601167adec9716239d2013b5ce82d005d303>

El sistema solicitó correctamente el préstamo *flash*, ejecutó los swaps a través del contrato inteligente desarrollado `DexAggregator.sol`, se devolvió el préstamo con su prima al protocolo Aave y aseguró una ganancia neta de \$0.55 USD. Este resultado validó la correcta implementación de la lógica central del proyecto.

4.2 Resultado 2: Oportunidad Abortada por Cambio de Precio

Este escenario demuestra la eficacia de los mecanismos de protección del sistema. Se detectó una oportunidad muy rentable, pero fue invalidada por la volatilidad del mercado antes de que pudiera ser ejecutada.

```
💡 Oportunidad detectada: 10.50% beneficio
    Comprar en solidlydex, vender en antfarm-ethereum...
🔄 Re-checking profitability before execution...
```

Actualizado: Profit: 0.00%

⚠ La oportunidad ya no es rentable. Abortando ejecución.

El sistema logró identificar correctamente una oportunidad de arbitraje con un margen de beneficio proyectado del 10.5%. Sin embargo, en los segundos que transcurrieron entre la detección y el momento de la ejecución, los precios en los *pools* de liquidez cambiaron, eliminando el margen de ganancia. El mecanismo de revalidación funcionó a la perfección, abortando la transacción y evitando así una operación que habría resultado en una pérdida.

4.3 Resultado 3: Error de Transacción Inesperado

Este escenario resalta la naturaleza impredecible de las interacciones en la *blockchain* y la importancia de un manejo de errores robusto.

```
✖ Error en arbitraje: Transaction reverted without a reason string
Buy DEX: sushiswap-v3-ethereum
Sell DEX: pancakeswap_ethereum
Token: WETH
Amount: 0.012075
Simulation successful!
```

A pesar de que la simulación *off-chain* (`callStatic`) fue exitosa y se validó la disponibilidad del token WETH en el pool de Aave, la transacción real fue revertida por la red sin un mensaje de error específico. Este tipo de error es particularmente difícil de depurar, ya que no corresponde a ninguno de los códigos de error estándar predefinidos en la librería `Errors.sol` de Aave V3.

Las causas probables residen en una condición externa al contrato de Aave, como una condición interna no cumplida (un `require`) en uno de los contratos de los *DEXes*, un problema de transferencia con un token no estándar, o una condición *race condition* en la red, es decir que otro actor en la red ejecutó una transacción similar milisegundos antes, alterando el mercado y anulando nuestra oportunidad.

El sistema identificó y preparó correctamente la oportunidad, pero fue frustrado por factores dinámicos e incontrolables del entorno de ejecución. Este resultado subraya la necesidad de implementar un registro de eventos *on-chain* aún más detallado (*DebugLog on-chain*) y diseñar planes de contingencia (estrategias *fallback*) para manejar fallos inesperados.

4.4 Observaciones Finales

Las pruebas de integración demostraron de manera concluyente que el sistema es funcional y robusto. La arquitectura diseñada es capaz de navegar el complejo ecosistema *DeFi*, protegerse contra pérdidas por volatilidad y ejecutar operaciones rentables. Los mecanismos de validación *off-chain* y la lógica defensiva en los contratos inteligentes son pilares fundamentales que garantizan su viabilidad.

Sin embargo, la aparición del error "*Transaction reverted without a reason string*" en el Resultado 3 es un hallazgo crítico. Este tipo de reversión sin un mensaje claro indica que el fallo no se origina en una de las validaciones explícitas del protocolo Aave. Para confirmar esta hipótesis, se analizó el contrato "`Errors.sol`" del repositorio oficial de Aave V3. Este archivo define una librería exhaustiva de más de 90 códigos de error estandarizados, tales como:

- **CALLER_MUST_BE_POOL** (Error '23'): Indica que la función `executeOperation` fue llamada por una dirección que no es el *Pool* de Aave.
- **RESERVE_FROZEN** (Error '28'): La reserva del activo está congelada.
- **BORROW_CAP_EXCEEDED** (Error '50'): Se ha superado el límite de endeudamiento para un activo.
- **INVALID_FLASHLOAN_EXECUTOR_RETURN** (Error '13'): El `callback` del préstamo *flash* no retornó "true".

El hecho de que la transacción se revierta sin ninguno de estos códigos sugiere que la ejecución supera con éxito todas las validaciones internas de Aave, pero falla en un punto posterior de la lógica de arbitraje, muy probablemente durante la interacción con los contratos de los *DEXes* externos.

Este análisis subraya la necesidad de instrumentar mejoras específicas para diagnosticar fallos externos:

- **Registro de Eventos *On-chain* Detallado:** Implementar la emisión de eventos (`emit DebugLog`) en más puntos críticos dentro de los contratos, especialmente antes y después de cada llamada a un *DEX*, para aislar el punto exacto del fallo.
- **Simulaciones Granulares:** Utilizar herramientas avanzadas como *Tenderly* para realizar simulaciones que ofrezcan un rastreo de pila (*stack trace*) completo de la ejecución, revelando la función y el contrato exacto que causa la reversión.
- **Análisis Profundo del Gas:** Investigar si las reversiones están relacionadas con estimaciones de gas insuficientes para rutas de swap complejas que involucran múltiples contratos.

Estas observaciones no invalidan el éxito del proyecto, sino que proporcionan una hoja de ruta clara para su mejora continua y su eventual transición a un entorno de producción completamente

automatizado, donde la capacidad de diagnosticar y recuperarse de errores imprevistos es fundamental.

En términos de madurez tecnológica, el sistema alcanzó el Nivel 7 de la escala TRL (Nivel de Madurez Tecnológica), evidenciado por la demostración funcional del prototipo completo en un entorno operativo real. La transacción confirmada en el bloque 22584625 de Ethereum Mainnet, con una ganancia neta de \$0.55 USD tras descontar costos de gas y prima del *flash loan*, constituye la prueba tangible de este nivel de madurez.

CAPÍTULO 5: CONCLUSIONES

5.1 Cumplimiento de Objetivos

Tras la finalización de las fases de diseño, desarrollo y experimentación, se puede afirmar que el proyecto cumplió con el objetivo general planteado. Se desarrolló un prototipo funcional de arbitraje de criptomonedas sobre la red de Ethereum compuesto por 2 contratos inteligentes (`FlashLoanArbitrage.sol` y `DexAggregator.sol`) capaces de interactuar con el protocolo de *flash loan* de Aave V3 para optimizar las ganancias, manteniendo un perfil de riesgo bajo. La transacción confirmada en el bloque 22584625, verificable públicamente en Etherscan, es la evidencia concreta de que el sistema desarrollado funciona en condiciones reales de mercado. La ganancia neta de \$0.55 USD obtenida en esa operación no es un número impresionante en términos absolutos, pero su importancia radica en lo que representa: una operación de arbitraje sin capital propio, ejecutada de forma completamente automatizada.

En cuanto a los objetivos específicos, el primero se cumplió parcialmente. El script `arbitrage.ts` es la representación del algoritmo del primer objetivo específico, el cual consulta y compara precios en tiempo real entre los 27 DEXes configurados, calcula rutas óptimas sobre 39 activos de alta liquidez y entrega resultados en segundos. El Resultado 2 del capítulo anterior ilustra con claridad la capacidad de detección del sistema identificando correctamente una oportunidad con un margen del 10.5%, sin embargo, los precios se movieron antes de que pudiera ejecutarse, y el sistema abortó la operación sin incurrir en pérdida alguna. Este resultado demuestra la eficacia del sistema al abortar operaciones cuando la volatilidad del mercado eliminaba el beneficio proyectado antes de la ejecución.

No obstante, el objetivo original contemplaba la detección de diferencias de precios entre múltiples ecosistemas blockchain, y este componente no se implementó. Si bien el diseño teórico consideró la interoperabilidad cross-chain, la complejidad técnica asociada a la integración con puentes entre redes y la heterogeneidad de los ecosistemas DeFi habría comprometido la viabilidad del proyecto dentro del plazo establecido. Por esta razón se priorizó concentrar el desarrollo en el ecosistema de Ethereum para alcanzar un nivel de madurez tecnológica (TRL) 7, favoreciendo la estabilidad y la seguridad sobre la expansión horizontal. Esta decisión se justifica con mayor detalle en la sección 5.2.

El segundo objetivo, correspondiente a la evaluación y selección de oportunidades con base en criterios de costo, riesgo y tiempo, también se cumplió. El sistema descarta automáticamente operaciones en las que el costo de *gas* supera el beneficio proyectado, filtra *pools* con liquidez inferior

a los \$100,000 USD y valida la disponibilidad del activo en las reservas de Aave antes de proceder. Estos criterios se aplican tanto a nivel *off-chain* en `arbitrage.ts` como a nivel *on-chain* dentro de `DexAggregator.sol`, que verifica la actividad y disponibilidad de cada *DEX* antes de solicitar una cotización.

El tercer objetivo, correspondiente al desarrollo de un algoritmo de ejecución automatizado, se cumplió. El contrato `FlashLoanArbitrage.sol` implementa la interfaz `IFlashLoanSimpleReceiver` de Aave V3 y encapsula la lógica completa de ejecución atómica: solicitud del préstamo flash, búsqueda de la mejor ruta de arbitraje a través de tokens intermediarios, ejecución de los intercambios mediante `DexAggregator.sol`, repago del préstamo con la prima correspondiente y transferencia de las ganancias al operador. La validación en producción real quedó documentada en la transacción del Resultado 1. mencionado en el capítulo anterior. En cuanto a la selección del proveedor de *flash loan* óptimo mencionada en el objetivo, el sistema utiliza exclusivamente el protocolo Aave V3. Esta decisión se fundamentó en que Aave V3 ofrece la función `flashLoanSimple` con la prima más competitiva del mercado (0.05%), soporta la mayor cantidad de activos en sus reservas y cuenta con la documentación más madura entre los proveedores disponibles. Se evaluaron de forma preliminar alternativas como dYdX y Balancer; sin embargo, dYdX opera bajo un modelo de préstamo sin prima pero con menor variedad de activos, y Balancer presentaba limitaciones de liquidez para los pares monitoreados. Por lo tanto, la integración de múltiples proveedores no se justificó técnicamente para el alcance actual del proyecto, aunque se identifica como una extensión viable en trabajos futuros.

El cuarto objetivo, referente al módulo de simulación y prueba, se cumplió con condicionantes. El módulo implementado incluye pruebas unitarias con Hardhat, simulaciones mediante *fork* de Mainnet y verificación previa a la ejecución con `callStatic`, lo cual demostró ser eficaz para anticipar fallas sin consumir gas. En relación con el análisis de escenarios adversos exigido por el objetivo, el resultado 2 documenta un escenario de volatilidad de precios en el que el sistema respondió correctamente al abortar la operación, y el resultado 3 documenta un escenario de fallo externo: una reversión sin mensaje de error originada en un contrato de un DEX externo, fuera del alcance directo del sistema. Este último hallazgo no invalida el objetivo, pero señala un área de mejora concreta: el diseño de una trazabilidad *on-chain* más específica que permita identificar el origen exacto de los fallos en contratos de terceros.

5.2 Decisiones técnicas y de diseño

La primera decisión de diseño que merece destacarse es la renuncia a la arquitectura *multi-chain* en favor de un sistema concentrado en la red de Ethereum. La integración con puentes *cross-chain* y la heterogeneidad de los ecosistemas *DeFi* en distintas redes implicaban una complejidad técnica que habría comprometido la viabilidad del proyecto dentro del plazo de la maestría. Concentrar el esfuerzo en una sola red permitió llevar el sistema a un nivel de madurez real y demostrable, en lugar de construir una arquitectura amplia pero superficial.

La segunda decisión relevante fue la adopción de Aave V3 como proveedor exclusivo de *flash loans*. Aunque el objetivo original planteaba la selección del proveedor óptimo, el análisis preliminar de las alternativas disponibles determinó que Aave V3 reunía las condiciones más favorables en términos de prima (0.05%), variedad de activos en reserva y estabilidad del protocolo. La integración de un solo proveedor redujo la superficie de ataque del contrato inteligente y simplificó la lógica de ejecución atómica.

Finalmente, la dependencia de la API de GeckoTerminal como fuente de datos de liquidez fue una decisión pragmática que permitió acceder rápidamente a información de pools en múltiples *DEXes* sin necesidad de mantener nodos propios de indexación.

5.3 Limitaciones identificadas

Las limitaciones del sistema son conocidas y están bien delimitadas. En periodos de alta congestión en Ethereum, los costos de *gas* pueden erosionar completamente los márgenes de ganancia disponibles, lo que hace que la escalabilidad económica del sistema dependa de condiciones de mercado favorables o de capital de préstamo significativamente mayor. Adicionalmente, la dependencia de la *API* de GeckoTerminal como fuente de datos de liquidez introduce un punto de falla externo que en un entorno de producción debería complementarse con lecturas directas *on-chain*. Finalmente, la reversión sin mensaje de error documentada en el Resultado 3 indica que algunos fallos ocurren dentro de contratos de *DEXes* externos, fuera del alcance directo del sistema. Diagnosticar estos errores requiere herramientas adicionales como Tenderly para obtener un rastreo de pila completo.

5.4 Proyección y trabajos futuros

Con la base construida, las extensiones más directas son ampliar el número de tokens monitoreados más allá de los 39 actualmente configurados, así como la implementación de un registro de eventos (*events*) detallado dentro de los contratos inteligentes para facilitar el depurado *on-chain*. Asimismo, la integración de lecturas directas de los pools de liquidez a través de nodos propios eliminaría la latencia de las *APIs* externas.

A mediano plazo, la expansión a redes de Capa 2 (*L2*) donde los costos de gas son menores, representaría una evolución del sistema. En el largo plazo, la sofisticación de estos algoritmos abre la puerta a su futura aplicación en estrategias de inversión institucional, siempre que el marco regulatorio evolucione para permitir una integración segura de activos digitales en carteras de inversión diversificadas.

CAPÍTULO 6: GLOSARIO

Término	Definición
AAVE / AAVE V3 Protocol	Protocolo <i>DeFi</i> de préstamos y créditos en <i>Ethereum</i> que permite operaciones como préstamos <i>flash</i> sin garantía, usando contratos inteligentes.
ABI (<i>Application Binary Interface</i>)	Especificación que define cómo interactuar externamente con un contrato inteligente en <i>Ethereum</i> (llamar funciones, leer datos, etc).
<i>Address</i> (Dirección)	Identificador único de cuentas y contratos en la <i>blockchain</i> .
AggregatorV3Interface	Interfaz de Chainlink para obtener datos de precios descentralizados.
AMM (Automated Market Maker)	Modelo que permite el intercambio de activos en <i>DEXes</i> usando fórmulas matemáticas y pools de liquidez en vez de libro de órdenes.
API (Application Programming Interface)	Conjunto de definiciones y protocolos para que aplicaciones interactúen entre sí, por ejemplo, para consultar precios.
Arbitraje (<i>Arbitrage</i>)	Estrategia que aprovecha diferencias de precio de un mismo activo en diferentes mercados para obtener una ganancia libre de riesgo.
Atomicidad (<i>Atomicity</i>)	Propiedad de las operaciones en <i>blockchain</i> por la que se ejecutan completamente o no se ejecuta nada.
Axios	Librería JavaScript para realizar solicitudes HTTP, útil para conectarse a <i>APIs</i> y recuperar datos externos desde Node.js o el navegador.
<i>Backend</i>	Parte lógica del sistema que se ejecuta en servidores, encargada de cálculos, procesos y conexión con bases de datos o <i>blockchains</i> .
Balancer	<i>DEX</i> basado en <i>AMM</i> con pools de múltiples tokens y pesos personalizados.
BaseFeePerGas	Valor base del gas determinado por el protocolo Ethereum para cada bloque bajo EIP-1559.

Beneficio Neto	Ganancia obtenida en una operación tras restar todos los costos, incluyendo <i>gas</i> y primas de préstamos <i>flash</i> .
<i>Bundle (Flashbots Bundle)</i>	Grupo de transacciones enviadas juntas a <i>Flashbots</i> para ejecución privada, protegiendo de ataques <i>MEV</i> .
Bóveda (<i>Vault</i>)	Contrato que almacena y administra fondos, como BalancerVault en Balancer.
Caché	Almacenamiento temporal de datos para acelerar el acceso y procesamiento.
Callback	Función que es llamada automáticamente por otro contrato al completarse una acción (ejemplo: Aave llama a <code>executeOperation</code> tras otorgar un <i>flash loan</i>).
Chainlink	Protocolo de oráculos que proporciona datos externos seguros a contratos inteligentes, especialmente precios de activos.
CoinGecko / GeckoTerminal	Servicios de consulta de precios, liquidez y datos de mercado para activos <i>DeFi</i> y criptomonedas.
Comisión por Préstamo <i>Flash (Premium)</i>	Tarifa que se paga por usar un préstamo <i>flash</i> , típicamente un pequeño porcentaje del monto prestado.
Comisión de Gas (Gas Fee)	Costo pagado a mineros/validadores por ejecutar operaciones y contratos en la <i>blockchain</i> .
Contrato Inteligente (<i>Smart Contract</i>)	Programa autoejecutable almacenado en la <i>blockchain</i> , con reglas fijas e inmutables tras su despliegue.
Control de Acceso (<code>Ownable</code> , <code>onlyOwner</code>)	Mecanismo que restringe ciertas funciones a un propietario o administrador del contrato.
Curve	<i>DEX</i> especializado en intercambios eficientes de <i>stablecoins</i> con bajo <i>slippage</i> .
<code>callStatic</code>	Llamada a función de contrato para simular su ejecución sin gastar gas ni modificar el estado.
DAI	<i>Stablecoin</i> descentralizada vinculada 1:1 al dólar estadounidense.
<i>Decentralized Exchange (DEX)</i>	Plataforma de intercambio de criptomonedas sin intermediarios, operada mediante contratos inteligentes.

<i>DeFi (Decentralized Finance)</i>	Ecosistema de aplicaciones financieras que operan sobre <i>blockchain</i> públicas y que eliminan intermediarios tradicionales (como bancos o casas de bolsa) mediante contratos inteligentes.
Despliegue (<i>Deploy</i>)	Acción de subir y activar un contrato inteligente en la <i>blockchain</i> .
DexAggregator.sol	Contrato inteligente que centraliza la lógica de swaps entre diferentes <i>DEXes</i> .
DexType (enum)	Enumeración que clasifica los tipos de <i>DEXes</i> en el sistema (V2, V3, Curve, etc).
Dotenv	Herramienta para gestionar variables de entorno fuera del código fuente.
EIP-1559	Actualización de Ethereum que introduce la tarifa base y el mecanismo de quemado de gas para optimizar costos.
Ethers.js	Librería JavaScript/TypeScript para interactuar con Ethereum.
ETH (Ether)	Criptomoneda nativa de Ethereum, usada para pagar comisiones de gas.
Ethereum Mainnet	Red principal de Ethereum, donde las transacciones tienen valor real.
<i>EVM (Ethereum Virtual Machine)</i>	Máquina virtual que ejecuta contratos inteligentes en Ethereum.
Evento (<i>Event</i>)	Registro de información importante durante la ejecución de contratos inteligentes, útil para monitoreo y auditoría.
ERC20	Estándar de tokens fungibles en Ethereum.
Elkfinance	<i>DEX</i> adicional integrado en el sistema, basado en modelo <i>AMM</i> .
Factory	Contrato que despliega y registra nuevos pares de liquidez en <i>DEXes</i> . En Uniswap V2 y protocolos similares. Es el encargado de crear los contratos de cada pool dado un par de tokens.
<i>Fee (Gas Fee, Flash Loan Fee)</i>	Comisión pagada por operación de red o por uso de servicios <i>DeFi</i> como préstamos <i>flash</i> .
Préstamo <i>Flash (Flash Loan)</i>	Préstamo sin garantía tomado y devuelto en una sola transacción atómica; si no se repaga, la transacción se revierte.

<i>Flashbots / Flashbots Bundle</i> Provider	Infraestructura que permite enviar transacciones privadas para evitar <i>MEV</i> y <i>front-running</i> en <i>Ethereum</i> .
Fork (Bifurcación de red)	Copia exacta o división de una <i>blockchain</i> para pruebas o cambios.
<i>Front-running</i>	Estrategia donde <i>bots</i> anticipan y adelantan transacciones para capturar ganancias a costa de otros usuarios.
<i>Gas / Gas Fee</i>	Unidad de costo de cómputo en <i>Ethereum</i> ; el gas fee es el monto pagado por consumir gas.
Generador de Claves (<code>generate-flashbots-key.ts</code>)	Script que genera una clave privada exclusiva para uso con <i>Flashbots</i> .
Gwei	Unidad para medir tarifas de gas en <i>Ethereum</i> . 1 ETH equivale a mil millones (1,000,000,000) de Gwei.
Hardhat	Entorno de desarrollo, pruebas y despliegue de contratos inteligentes en <i>Ethereum</i> .
Hooks (Uniswap V4)	Funcionalidad avanzada de Uniswap V4 para modificar la lógica de pools a través de 'enganches' personalizados.
Historial de Gas (<code>gasFee.json</code>)	Archivo que almacena el historial de tarifas de gas para análisis y optimización.
ICurveRouter	Interfaz para interactuar con el <i>router</i> de Curve desde contratos inteligentes.
<i>IFlashLoanSimpleReceiver</i>	Interfaz estándar de Aave para contratos que desean recibir préstamos <i>flash</i> simples.
IMaverickRouter, INewDexRouter, etc.	Interfaces para integración con <i>routers</i> específicos de diferentes <i>DEXes</i> .
IPool / IPoolAddressesProvider	Interfaces de Aave que gestionan pools y direcciones clave del protocolo.
IPancakeRouter, IPancakeQuoter	Interfaces para interactuar con PancakeSwap desde contratos inteligentes.
Interface (Interfaz)	Definición estándar de funciones y estructuras de un contrato para interoperabilidad.

Initial Balance	Saldo inicial disponible para operaciones (despliegue, ejecución, etc).
Intermediary Token (Token Intermediario)	Token usado como paso intermedio en rutas de arbitraje para maximizar rentabilidad.
Integración (<i>off-chain/on-chain</i>)	Conexión entre lógica ejecutada fuera de la <i>blockchain</i> y contratos inteligentes en la <i>blockchain</i> .
Infura / Alchemy (RPC Provider)	Servicios que proveen acceso remoto a nodos de <i>Ethereum</i> .
Iniciador (Initiator)	Entidad que inicia la operación de un préstamo <i>flash</i> .
<code>isV2Like</code>	Variable booleana que indica si un <i>DEX</i> opera bajo la arquitectura de Uniswap V2 (<i>AMM</i> clásico).
<code>isV3Like</code>	Variable booleana que indica si un <i>DEX</i> funciona bajo el modelo de Uniswap V3 (<i>AMM</i> con rangos).
JavaScript	Lenguaje de programación usado en scripts <i>off-chain</i> .
JSON	Formato de almacenamiento de datos estructurados usados por el sistema.
KyberClassic, KyberElastic	<i>DEXes</i> soportados por el sistema, con diferentes modelos de liquidez.
Liquidez (<i>Liquidity</i>)	Cantidad de fondos disponibles en un <i>pool</i> de intercambio, afecta el <i>slippage</i> .
<i>Pool</i> de Liquidez (<i>Liquidity Pool</i>)	Contrato que almacena activos de usuarios para facilitar intercambios en un <i>DEX</i> .
Mainnet	Red principal y oficial de <i>Ethereum</i> .
Mapeo (<i>Mapping</i>)	Estructura de datos en Solidity que asocia claves (<i>keys</i>) con valores (<i>values</i>) eficientemente dentro de <i>smart contracts</i> .
<i>MEV</i> (Maximal Extractable Value)	Valor que puede extraer un minero/validador al reordenar, insertar o censurar transacciones en un bloque.
<i>Mempool</i>	Conjunto de transacciones pendientes de ser incluidas en un bloque de <i>blockchain</i> .
Modulo (<i>Module</i>)	Componente funcional y reutilizable de software.
Monitor (<i>monitor function</i> , monitoreo)	Lógica para observar y registrar eventos, estados o métricas.

Node.js	Entorno de ejecución de JavaScript fuera del navegador, usado para scripts <i>backend</i> .
Node_modules	Carpeta donde npm almacena dependencias del proyecto.
NPM (Node Package Manager)	Gestor de paquetes para JavaScript/ <i>TypeScript</i> .
NonReentrant / ReentrancyGuard	Patrón y librería de seguridad para prevenir ataques de reentrada en funciones críticas.
Off-chain	Lógica o datos procesados fuera de la <i>blockchain</i> .
On-chain	Lógica o datos que residen y ejecutan dentro de la <i>blockchain</i> .
OpenZeppelin	Biblioteca líder de contratos inteligentes auditados y herramientas de seguridad para Solidity.
Ownership (transferOwnership)	Transferencia de control administrativo de un contrato inteligente a otra cuenta.
Oráculo (Oracle)	Servicio que provee datos externos (por ejemplo, precios) a la <i>blockchain</i> .
PancakeSwap	<i>DEX</i> basado en <i>AMM</i> , originalmente desplegado en BNB Chain, con presencia en Ethereum a través de contratos inteligente específicos compatibles con la EVM.
Plugin	Extensión que agrega funcionalidades extra a <i>frameworks</i> como Hardhat, por ejemplo, @nomicfoundation/hardhat-ethers para integrar tipos Typescript automáticamente.
Pool (Liquidity Pool)	Es un contrato inteligente que almacena una reserva de fondos en criptomonedas para permitir que los usuarios realicen intercambios de forma automática y descentralizada, sin depender de un libro de órdenes tradicional.
Premium (Prima de Préstamo Flash)	Es la comisión porcentual que el usuario paga al protocolo prestamista por el uso temporal de capital, la cual debe liquidarse junto con el monto principal dentro de la misma transacción atómica.
Provider (RPC, Flashbots, Alchemy)	Servicio que facilita la conexión y envío de transacciones a la <i>blockchain</i> .

<i>Quoter</i>	Contrato o función que permite obtener una cotización de precio en un <i>DEX</i> antes de hacer el swap real.
Registro (Registro de <i>DEXes</i> , <code>setupAllDexes</code>)	Proceso y funciones para almacenar e inicializar <i>DEXes</i> soportados por el sistema.
Repago	Acción de devolver fondos prestados (préstamo <i>flash</i>) y pagar la prima correspondiente.
Retiro de Fondos (<code>rescueTokens</code> , <code>rescueETH</code>)	Funciones administrativas para recuperar fondos atascados.
Route (Ruta de Swap)	Camino definido de intercambios para convertir un token en otro a través de <i>DEXes</i> .
<i>Router</i>	Contrato en <i>DEX</i> que enruta operaciones de intercambio entre pools de liquidez.
RPC (Remote Procedure Call)	Protocolo que permite ejecutar funciones en nodos remotos de <i>Ethereum</i> .
SafeERC20	Biblioteca de OpenZeppelin para operaciones seguras con tokens ERC20.
<i>Script</i>	Archivo automatizado para ejecutar tareas repetitivas o programadas en el sistema.
<i>SDK (Software Development Kit)</i>	Conjunto de herramientas y librerías para facilitar el desarrollo de software.
<code>SetupAllDexes()</code>	Función para configurar e inicializar todos los <i>DEXes</i> soportados en el sistema.
<i>Slippage</i> (Deslizamiento)	Diferencia entre el precio esperado y el ejecutado en una operación de intercambio, causado por variación de liquidez o velocidad del mercado.
<i>Stablecoin</i>	Criptomoneda con valor anclado a un activo externo, como el USD (ej. USDC, DAI).
<i>Swap</i>	Intercambio de un token por otro en un <i>DEX</i> .
SushiSwap	<i>DEX</i> basado en Uniswap con mejoras y recompensas adicionales para proveedores de liquidez.
Testnet	Red de pruebas de <i>Ethereum</i> sin valor real, usada para desarrollo y <i>testing</i> .

<i>Token</i>	Representación digital de un activo en la <i>blockchain</i> .
Transacción Atómica	Operación que se ejecuta completamente o no se ejecuta nada, asegurando consistencia en <i>blockchain</i> .
Transacción Revertida	Operación anulada por la <i>blockchain</i> si alguna condición falla.
TypeChain	Herramienta que genera tipos <i>TypeScript</i> a partir de contratos Solidity.
<i>TypeScript</i>	Lenguaje que extiende JavaScript con tipado estático, usado en scripts automatizados.
Uniswap V2 / V3 / V4	Versiones sucesivas del <i>DEX</i> líder en <i>Ethereum</i> , cada una con diferentes modelos de liquidez y arquitectura.
USDC	<i>Stablecoin</i> centralizada vinculada 1:1 al dólar.
Wallet	Billetera digital para almacenar y firmar transacciones de criptomonedas.
Wei	Unidad más pequeña de Ether (1 Ether = 10^{18} Wei).
WETH	Tokenizado de ETH en formato ERC20 para mayor compatibilidad con DEXes.
Zero Address	Dirección especial “0x000...000” usada en <i>Ethereum</i> como valor nulo.

CAPÍTULO 7: REFERENCIAS

- [1] Kurniawan, I Gede Wahyu Antara *et al.*, “The Rise of Decentralized Finance (DeFi): Opportunities for Disruption in Traditional Financial Models”, *Journal of Education, Social & Communication Studies*, vol. 2, núm. 2, pp. 128–136, may 2025, [En línea]. Disponible en: https://www.researchgate.net/publication/391397676_The_rise_of_decentralized_finance_DeFi_Opportunities_for_disruption_in_traditional_financial_models
- [2] A. Sari, M. A. Gavcar, S. Aplay, y A. Ozsoy, “Automated Trading in Cryptocurrency Markets: Strategies, Impacts, and Future Directions”, en *2025 IEEE International Conference on Consumer Electronics (ICCE)*, Las Vegas, NV, USA: IEEE, ene. 2025, pp. 1–6. doi: 10.1109/ICCE63647.2025.10930129.
- [3] I. Makarov y A. Schoar, “Trading and arbitrage in cryptocurrency markets”, *Journal of Financial Economics*, vol. 135, núm. 2, pp. 293–319, feb. 2020, doi: 10.1016/j.jfineco.2019.07.001.
- [4] A. Alexander, “Latency Arbitrage in Cryptocurrency Markets: Analyzing Execution Speeds & Liquidity Dynamics”, 2025, *SSRN*. doi: 10.2139/ssrn.5143158.
- [5] B. Chandler, P. Stiles, y J. Blinken, “DeFi Flash Loans: What ‘Atomicity’ Makes Possible - Why Does This Innovation Not Already Exist in Traditional Finance?”, *SSRN Journal*, 2022, doi: 10.2139/ssrn.4116909.
- [6] M. H. Nguyen, B. N. Thanh, H. Pham, y T. T. T. Pham, “The determinants of funding liquidity risk in decentralized lending”, *Global Finance Journal*, vol. 64, p. 101055, mar. 2025, doi: 10.1016/j.gfj.2024.101055.
- [7] K. Qin, L. Zhou, B. Livshits, y A. Gervais, “Attacking the DeFi Ecosystem with Flash Loans for Fun and Profit”, el 20 de marzo de 2021, *arXiv*: arXiv:2003.03810. doi: 10.48550/arXiv.2003.03810.
- [8] D. Wang *et al.*, “Towards A First Step to Understand Flash Loan and Its Applications in DeFi Ecosystem”, en *Proceedings of the Ninth International Workshop on Security in Blockchain and Cloud Computing*, Virtual Event Hong Kong: ACM, may 2021, pp. 23–28. doi: 10.1145/3457977.3460301.
- [9] P. Daian *et al.*, “Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges”, el 10 de abril de 2019, *arXiv*: arXiv:1904.05234. doi: 10.48550/arXiv.1904.05234.
- [10] R. Kanojia, “Flash Loan Arbitrage Bot”, *SSRN Journal*, 2023, doi: 10.2139/ssrn.4447220.
- [11] Comisión Nacional del Sistema de Ahorro para el Retiro (CONSAR), “Informe al Congreso – 1er Trimestre 2025”, Comisión Nacional del Sistema de Ahorro para el Retiro (CONSAR), Ciudad de México, México, Informe trimestral al H. Congreso de la Unión, may 2025. [En línea]. Disponible en:

https://sil.gobernacion.gob.mx/Archivos/Documentos/2025/05/asun_4901802_20250527_1748357535.pdf

- [12] A. R. Hevner, S. T. March, J. Park, y S. Ram, “Design Science in Information Systems Research”, *MIS Quarterly*, vol. 28, núm. 1, pp. 75–106, mar. 2004, doi: 10.2307/25148625.
- [13] K. Peffers, T. Tuunanen, M. A. Rothenberger, y S. Chatterjee, “A Design Science Research Methodology for Information Systems Research”, *Journal of Management Information Systems*, vol. 24, núm. 3, pp. 45–77, dic. 2007, doi: 10.2753/MIS0742-1222240302.
- [14] Mankins, John C., “Technology Readiness Levels: A White Paper”, NASA, Office of Space Access and Technology, Advanced Concepts Office, Washington, DC, abr. 1995. [En línea]. Disponible en:
https://space.se.spacegrant.org/SEMModules/Technology%20Mods/Mankins_trl.pdf
- [15] J. Yli-Huumo, D. Ko, S. Choi, S. Park, y K. Smolander, “Where Is Current Research on Blockchain Technology?—A Systematic Review”, *PLoS ONE*, vol. 11, núm. 10, p. e0163477, oct. 2016, doi: 10.1371/journal.pone.0163477.
- [16] M. El Hajj y I. Farran, “The Cryptocurrencies in Emerging Markets: Enhancing Financial Inclusion and Economic Empowerment”, *JRFM*, vol. 17, núm. 10, p. 467, oct. 2024, doi: 10.3390/jrfm17100467.
- [17] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System”, *SSRN Journal*, 2008, doi: 10.2139/ssrn.3440802.
- [18] C. Vanmathi, A. Farouk, S. M. Alhammad, R. Mangayarkarasi, S. Bhattacharya, y M. S. B. Kasyapa, “The Role of Blockchain in Transforming Industries Beyond Finance”, *IEEE Access*, vol. 12, pp. 148845–148867, 2024, doi: 10.1109/ACCESS.2024.3468611.
- [19] H. Taherdoost, “Smart Contracts in Blockchain Technology: A Critical Review”, *Information*, vol. 14, núm. 2, p. 117, feb. 2023, doi: 10.3390/info14020117.
- [20] G. Tripathi, M. A. Ahad, y G. Casalino, “A comprehensive review of blockchain technology: Underlying principles and historical background with future challenges”, *Decision Analytics Journal*, vol. 9, p. 100344, dic. 2023, doi: 10.1016/j.dajour.2023.100344.
- [21] S. A. S. Sawayz y A. Kavitha, “Review of Academic Research on Blockchain Technology and Smart Contract”, *SSRN Journal*, 2025, doi: 10.2139/ssrn.5195574.
- [22] N. M. Alshahrani, M. L. Mat Kiah, B. B. Zaidan, A. H. Alamoodi, y A. Saif, “A Review of Smart Contract Blockchain Based on Multi-Criteria Analysis: Challenges and Motivations”, *Computers, Materials & Continua*, vol. 75, núm. 2, pp. 2833–2858, 2023, doi: 10.32604/cmc.2023.036138.
- [23] A. Gangwal, H. R. Gangavalli, y A. Thirupathi, “A Survey of Layer-Two Blockchain Protocols”, el 26 de julio de 2022, *arXiv*: arXiv:2204.08032. doi: 10.48550/arXiv.2204.08032.

- [24]M. B. Saif, S. Migliorini, y F. Spoto, “A Survey on Data Availability in Layer 2 Blockchain Rollups: Open Challenges and Future Improvements”, *Future Internet*, vol. 16, núm. 9, p. 315, ago. 2024, doi: 10.3390/fi16090315.
- [25]G. Wood, “Ethereum: A Secure Decentralised Generalised Transaction Ledger – Yellow Paper”. 2014. [En línea]. Disponible en: <https://ethereum.github.io/yellowpaper/paper.pdf>
- [26]S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han, y F.-Y. Wang, “Blockchain-Enabled Smart Contracts: Architecture, Applications, and Future Trends”, *IEEE Trans. Syst. Man Cybern., Syst.*, vol. 49, núm. 11, pp. 2266–2277, nov. 2019, doi: 10.1109/TSMC.2019.2895123.
- [27]W. Zou *et al.*, “Smart Contract Development: Challenges and Opportunities”, *IEEE Trans. Software Eng.*, vol. 47, núm. 10, pp. 2084–2106, oct. 2021, doi: 10.1109/TSE.2019.2942301.
- [28]C. D. Clack, V. A. Bakshi, y L. Braine, “Smart Contract Templates: foundations, design landscape and research directions”, el 15 de marzo de 2017, *arXiv*: arXiv:1608.00771. doi: 10.48550/arXiv.1608.00771.
- [29]V. Buterin y F. Vogelsteller, “ERC-20: Token Standard”, Ethereum Improvement Proposals. Consultado: el 11 de diciembre de 2025. [En línea]. Disponible en: <https://eips.ethereum.org/EIPS/eip-20>
- [30]B. K. Meister y H. C. W. Price, “Gas fees on the Ethereum blockchain: from foundations to derivative valuations”, *Front. Blockchain*, vol. 7, p. 1462666, nov. 2024, doi: 10.3389/fbloc.2024.1462666.
- [31]A. Kuhlman y A. Wicaksana, “Smart contract optimization for gas fee reduction with static solidity optimizer”, *Discov Appl Sci*, vol. 7, núm. 8, p. 888, ago. 2025, doi: 10.1007/s42452-025-07537-6.
- [32]H. Kim y D. Kim, “Optimal Gas Fee Minimization in DeFi: Enhancing Efficiency and Security on the Ethereum Blockchain”, *IEEE Access*, vol. 12, pp. 173810–173823, 2024, doi: 10.1109/ACCESS.2024.3495714.
- [33]V. Buterin, E. Conner, R. Dudley, M. Slipper, I. Norden, y A. Bakhta, “EIP-1559: Fee market change for ETH 1.0 chain”, Ethereum Improvement Proposals, abr. 2019. [En línea]. Disponible en: <https://eips.ethereum.org/EIPS/eip-1559>
- [34]T. Roughgarden, “Transaction Fee Mechanism Design”, 2021, *arXiv*. doi: 10.48550/ARXIV.2106.01340.
- [35]M. Z. Haider, T. Noreen, M. Salman, M. D. D. Assunção, y K. Zhang, “V-ZOR: Enabling Verifiable Cross-Blockchain Communication via Quantum-Driven ZKP Oracle Relays”, en *2025 7th International Conference on Blockchain Computing and Applications (BCCA)*, Durbovnic, Croatia: IEEE, oct. 2025, pp. 765–771. doi: 10.1109/BCCA66705.2025.11229691.

- [36] C. Li, Z. Sun, J. X. Yuu, y X. Wang, “The Walls Have Ears: Unveiling Cross-Chain Sandwich Attacks in DeFi”, el 19 de noviembre de 2025, *arXiv*: arXiv:2511.15245. doi: 10.48550/arXiv.2511.15245.
- [37] R. Xu, J. Zhu, L. Yang, Y. Lu, y L. D. Xu, “Decentralized finance (DeFi): a paradigm shift in the Fintech”, *Enterprise Information Systems*, vol. 18, núm. 9, p. 2397630, sep. 2024, doi: 10.1080/17517575.2024.2397630.
- [38] V. Gramlich, T. Guggenberger, M. Principato, B. Schellinger, y N. Urbach, “A multivocal literature review of decentralized finance: Current knowledge and future research avenues”, *Electron Markets*, vol. 33, núm. 1, p. 11, dic. 2023, doi: 10.1007/s12525-023-00637-4.
- [39] J. Castillo León y A. Lehar, “What data have told us about decentralized finance”, *Journal of Corporate Finance*, vol. 96, p. 102916, ene. 2026, doi: 10.1016/j.jcorpfin.2025.102916.
- [40] K. Qin, L. Zhou, Y. Afonin, L. Lazzaretti, y A. Gervais, “CeFi vs. DeFi -- Comparing Centralized to Decentralized Finance”, 2021, *arXiv*. doi: 10.48550/ARXIV.2106.08157.
- [41] J. Xu, K. Paruch, S. Cousaert, y Y. Feng, “SoK: Decentralized Exchanges (DEX) with Automated Market Maker (AMM) Protocols”, *ACM Comput. Surv.*, vol. 55, núm. 11, pp. 1–50, nov. 2023, doi: 10.1145/3570639.
- [42] V. Mohan, “Automated market makers and decentralized exchanges: a DeFi primer”, *Financ Innov*, vol. 8, núm. 1, p. 20, dic. 2022, doi: 10.1186/s40854-021-00314-5.
- [43] T. Bitterli y F. Schär, “Decentralized Exchanges: The Profitability Frontier of Constant Product Market Makers”, 2023, *arXiv*. doi: 10.48550/ARXIV.2302.05219.
- [44] Á. Cartea, F. Drissi, y M. Monga, “Decentralised finance and automated market making: Execution and speculation”, *Journal of Economic Dynamics and Control*, vol. 177, p. 105134, ago. 2025, doi: 10.1016/j.jedc.2025.105134.
- [45] H. Adams, N. Zinsmeister, y M. Salem, “Uniswap v3 Core”. marzo de 2021. [En línea]. Disponible en: <https://uniswap.org/whitepaper-v3.pdf>
- [46] Z. Fan, F. Marmolejo-Cossío, D. J. Moroz, M. Neuder, R. Rao, y D. C. Parkes, “Strategic Liquidity Provision in Uniswap v3”, 2021, *arXiv*. doi: 10.48550/ARXIV.2106.12033.
- [47] H. R. Varian, “The Arbitrage Principle in Financial Economics”, *Journal of Economic Perspectives*, vol. 1, núm. 2, pp. 55–72, nov. 1987, doi: 10.1257/jep.1.2.55.
- [48] K. Duan, Z. Li, A. Urquhart, y J. Ye, “Dynamic efficiency and arbitrage potential in Bitcoin: A long-memory approach”, *International Review of Financial Analysis*, vol. 75, p. 101725, may 2021, doi: 10.1016/j.irfa.2021.101725.
- [49] E. Dávila, D. Graves, y C. Parlatore, “The Value of Arbitrage”, *Journal of Political Economy*, vol. 132, núm. 6, pp. 1947–1993, jun. 2024, doi: 10.1086/728453.

- [50] M. Muck, T. Schmidl, y J. Wolf, “Wish or reality? On the exploitability of triangular arbitrage in cryptocurrency markets”, *Finance Research Letters*, vol. 73, p. 106508, mar. 2025, doi: 10.1016/j.frl.2024.106508.
- [51] J. F. Muth, “Rational Expectations and the Theory of Price Movements”, *Econometrica*, vol. 29, núm. 3, p. 315, jul. 1961, doi: 10.2307/1909635.
- [52] M. Avellaneda y J.-H. Lee, “Statistical arbitrage in the US equities market”, *Quantitative Finance*, vol. 10, núm. 7, pp. 761–782, ago. 2010, doi: 10.1080/14697680903124632.
- [53] L. Zhou, K. Qin, C. F. Torres, D. V. Le, y A. Gervais, “High-Frequency Trading on Decentralized On-Chain Exchanges”, en *2021 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA: IEEE, may 2021, pp. 428–445. doi: 10.1109/SP40001.2021.00027.
- [54] Y. Wang, Y. Chen, H. Wu, L. Zhou, S. Deng, y R. Wattenhofer, “Cyclic Arbitrage in Decentralized Exchanges”, el 14 de enero de 2022, *arXiv*: arXiv:2105.02784. doi: 10.48550/arXiv.2105.02784.
- [55] Y. Zhang, T. Yan, J. Lin, B. Kraner, y C. Tessone, “An Improved Algorithm to Identify More Arbitrage Opportunities on Decentralized Exchanges”, el 24 de junio de 2024, *arXiv*: arXiv:2406.16573. doi: 10.48550/arXiv.2406.16573.
- [56] I. Grishchenko, M. Maffei, y C. Schneidewind, “A Semantic Framework for the Security Analysis of Ethereum Smart Contracts”, en *Principles of Security and Trust*, vol. 10804, L. Bauer y R. Küsters, Eds., en *Lecture Notes in Computer Science*, vol. 10804. , Cham: Springer International Publishing, 2018, pp. 243–269. doi: 10.1007/978-3-319-89722-6_10.
- [57] S. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, y W. Knottenbelt, “SoK: Decentralized Finance (DeFi)”, en *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, Cambridge MA USA: ACM, sep. 2022, pp. 30–46. doi: 10.1145/3558535.3559780.
- [58] E. Frangella y L. Herskind, “Aave V3 Technical Paper”. enero de 2022. [En línea]. Disponible en: https://github.com/aave/aave-v3-core/blob/master/techpaper/Aave_V3_Technical_Paper.pdf
- [59] M. Bartoletti, J. Chiang, T. Junttila, A. Lluch Lafuente, M. Mirelli, y A. Vandin, “Formal Analysis of Lending Pools in Decentralized Finance”, en *Leveraging Applications of Formal Methods, Verification and Validation. Adaptation and Learning*, vol. 13703, T. Margaria y B. Steffen, Eds., en *Lecture Notes in Computer Science*, vol. 13703. , Cham: Springer Nature Switzerland, 2022, pp. 335–355. doi: 10.1007/978-3-031-19759-8_21.
- [60] Y. Cao, C. Zou, y X. Cheng, “Flashot: A Snapshot of Flash Loan Attack on DeFi Ecosystem”, el 1 de febrero de 2021, *arXiv*: arXiv:2102.00626. doi: 10.48550/arXiv.2102.00626.
- [61] P. Daian *et al.*, “Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability”, en *2020 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA: IEEE, may 2020, pp. 910–927. doi: 10.1109/SP40000.2020.00040.

- [62]G. Angeris, H.-T. Kao, R. Chiang, C. Noyes, y T. Chitra, “An analysis of Uniswap markets”, el 9 de febrero de 2021, *arXiv*: arXiv:1911.03380. doi: 10.48550/arXiv.1911.03380.
- [63]L. Zhou, K. Qin, A. Cully, B. Livshits, y A. Gervais, “On the Just-In-Time Discovery of Profit-Generating Transactions in DeFi Protocols”, el 3 de marzo de 2021, *arXiv*: arXiv:2103.02228. doi: 10.48550/arXiv.2103.02228.
- [64]N. Atzei, M. Bartoletti, y T. Cimoli, “A Survey of Attacks on Ethereum Smart Contracts (SoK)”, en *Principles of Security and Trust*, vol. 10204, M. Maffei y M. Ryan, Eds., en Lecture Notes in Computer Science, vol. 10204. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 164–186. doi: 10.1007/978-3-662-54455-6_8.
- [65]Z. Liao *et al.*, “SmartAxe: Detecting Cross-Chain Vulnerabilities in Bridge Smart Contracts via Fine-Grained Static Analysis”, *Proc. ACM Softw. Eng.*, vol. 1, núm. FSE, pp. 249–270, jul. 2024, doi: 10.1145/3643738.
- [66]B. Öz *et al.*, “Cross-Chain Arbitrage: The Next Frontier of MEV in Decentralized Finance”, *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 9, núm. 3, pp. 1–33, dic. 2025, doi: 10.1145/3771566.
- [67]A. Narayanan, *Bitcoin and cryptocurrency technologies: a comprehensive introduction*. Princeton: Princeton University Press, 2016.
- [68]G. Caldarelli y J. Ellul, “The Blockchain Oracle Problem in Decentralized Finance—A Multivocal Approach”, *Applied Sciences*, vol. 11, núm. 16, p. 7572, ago. 2021, doi: 10.3390/app11167572.
- [69]F. Schär, “Decentralized Finance: On Blockchain- and Smart Contract-Based Financial Markets”, *Federal Reserve Bank of St. Louis Review*, vol. 103, núm. 2, pp. 153–174, 2021, doi: 10.20955/r.103.153-74.
- [70]K. Qin, L. Zhou, B. Livshits, y A. Gervais, “Attacking the DeFi Ecosystem with Flash Loans for Fun and Profit”, en *Financial Cryptography and Data Security*, vol. 12674, N. Borisov y C. Diaz, Eds., en Lecture Notes in Computer Science, vol. 12674. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2021, pp. 3–32. doi: 10.1007/978-3-662-64322-8_1.