OPTIMIZING GAS EFFICIENCY IN ETHEREUM SMART CONTRACTS USING DESIGN PATTERN REFINEMENT AND COMPLEXITY REDUCTION

Weedagamaarachchi K.S

IT21576966

B.Sc. (Hons) Degree in Information Technology Specialized in Information Technology

Department of Information Technology

Sri Lanka Institute of Information Technology
Sri Lanka

April 2025

OPTIMIZING GAS EFFICIENCY IN ETHEREUM SMART CONTRACTS USING DESIGN PATTERN REFINEMENT AND COMPLEXITY REDUCTION

Weedagamaarachchi K.S

IT21576966

Dissertation submitted in partial fulfillment of the requirements for the Bachelor of Science (Hons) in Information Technology
Specialized in Information Technology

Department of Information Technology

Sri Lanka Institute of Information Technology
Sri Lanka

April 2025

DECLARTION

Declaration of the Candidate

“I declare that this is my own work and this dissertation1 does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to Sri Lanka Institute of Information Technology, the nonexclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Name	Student ID	Signature	Date
Weedagamaarachchi K.S	IT21576966	[illegible]	2025/04/11

Declaration of the Supervisor

The above candidate has carried out research for the bachelor’s degree Dissertation under my supervision.

Supervisor: Mr.Vishan Jayasinghearachchi	Signature:	Date: 2025/04/11
Co-Supervisor: Mr. Jeewaka Perera	Signature:	Date:2025/04/11

ABSTRACT

Smart contracts on the Ethereum blockchain are transforming decentralized applications, yet their operational costs, particularly gas fees, pose significant challenges for scalability. This study investigates the impact of cyclomatic code complexity on gas consumption and presents optimization strategies to enhance the cost-efficiency of Ethereum-based smart contracts. Drawing inspiration from the research on e- voting systems, this work applies both established and novel gas- saving techniques including variable packing, use of uint256, bytes32, and immutable keywords on three widely-used Solidity patterns: Factory, Registry, and State Machine. These optimized contracts were integrated into a practical use case involving a coco peat supply chain management system, enabling a real- world evaluation of gas efficiency. Comparative deployment and runtime gas analyses revealed substantial reductions, with deployment costs lowered by up to ~19% and transaction execution costs reduced by up to ~14%. Notably, techniques such as mapping simplification and elimination of unnecessary storage proved highly effective in runtime scenarios.

The optimized contracts contributed to improved scalability and responsiveness, critical for high-frequency transaction environments like supply chains. The results validate the hypothesis that design-level optimizations can yield meaningful economic and performance benefits in blockchain applications. This research advances prior work by extending optimization strategies to common smart contract patterns and contextualizing them in a commercial-grade application. Future research can explore automated compiler-level tools and layer-2 scaling solutions to further enhance contract efficiency. Ultimately, this work contributes to the growing body of knowledge on building sustainable and accessible blockchain ecosystems.

Keywords: - Blockchain, Solidity design patterns, Ethereum smart contracts, Supply Chain Management (SCM), Gas fee optimization

ACKNOWLEDGEMENT

First and foremost, I would like to express my sincere gratitude to my supervisor, Mr. Vishan Jayasinghearachchi, for his invaluable guidance, encouragement, and unwavering support throughout the course of this undergraduate research project. His expertise, constructive feedback, and commitment to academic excellence have been instrumental in shaping the direction and depth of this study.

I am also deeply thankful to my co-supervisor, Mr. Jeewaka Perera, for his continuous support, timely insights, and readiness to assist whenever challenges arose. His contribution significantly enriched the quality of this work, and his mentorship was a vital source of motivation throughout the project journey.

This project was the result of a collaborative effort, and I would like to extend my heartfelt appreciation to my team members for their dedication, hard work, and seamless cooperation. Their persistence and enthusiasm helped overcome numerous obstacles, and their contributions were key to achieving our shared objectives.

Additionally, this research ventured into several interdisciplinary domains that extended beyond our initial areas of expertise. I am grateful to the individuals, communities, and technical resources that helped bridge these knowledge gaps. Their insights not only supported the development of this project but also contributed to my personal and professional growth.

Finally, I would like to express my heartfelt thanks to my **friends, and colleagues** for their constant encouragement, patience, and emotional support. Their belief in my abilities was a continuous source of strength, especially during the most demanding phases of this research.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVAITION

ABI	-	Application Binary Interface
CLI	-	Command Line Interface
CPU	-	Central Processing Unit
Dapp	-	Decentralized Application
EIP	-	Ethereum Improvement Proposal
EVM	-	Ethereum Virtual Machine
ETH	-	Ether (Ethereum cryptocurrency)
GUI	-	Graphical User Interface
IDE	-	Integrated Development Environment
IoT	-	Internet of Things
JSON	-	JavaScript Object Notation
LIFO	-	Last In, First Out
LOC	-	Lines of Code
NFT	-	Non-Fungible Token
PoS	-	Proof of Stake
SCM	-	Supply Chain Management
VM	-	Virtual Machine
URL	-	Uniform Resource Locator
Yul	-	Intermediate language for EVM targeting

INTRODUCTION

The advent of **blockchain technology** has introduced a paradigm shift in the way digital systems manage trust, data integrity, and transactional transparency. At its core, blockchain is a **distributed ledger technology** that allows information to be securely stored and shared across a network of nodes without the need for a centralized authority. This decentralized approach enables **peer-to-peer (P2P)** transactions governed by consensus algorithms and cryptographic techniques, ensuring that every transaction is tamper-evident, verifiable, and immutable once recorded [1]. As a result, blockchain technology has found application in a wide array of industries including **finance**, **healthcare**, **supply chain management**, **real estate**, **voting systems**, and more [7].

Each block in a blockchain contains a list of transactions, a timestamp, and a cryptographic hash of the previous block, effectively chaining blocks together in a secure sequence. The structure ensures that any attempt to alter a past transaction would require re-computing all subsequent blocks an effort practically infeasible due to the computational cost and distributed consensus required. The **trustless and transparent nature** of blockchain technology eliminates the need for intermediaries, thereby reducing costs, improving efficiency, and enhancing system resilience [1].

Among the various blockchain platforms, **Ethereum** has emerged as one of the most prominent due to its pioneering support for **smart contracts** self-executing pieces of code that reside on the blockchain and automatically enforce the terms of an agreement when predefined conditions are met. Smart contracts are written in high-level programming languages such as **Solidity**, which are then compiled into **Ethereum Virtual Machine (EVM)** bytecode for deployment and execution across the network [2]. The **EVM** acts as a decentralized computational engine, capable of processing contract logic and maintaining contract state across thousands of nodes.

While Ethereum introduced a new level of programmability to blockchain systems, it also brought with it a critical operational consideration: **gas fees**. Gas is the fundamental unit used to measure the **computational effort** required to execute operations on the Ethereum network. Every instruction, function call, or data storage operation in a smart contract consumes a specific amount of gas. The user initiating the transaction must pay this gas in **Ether (ETH)**, Ethereum's native cryptocurrency, to incentivize validators (miners or stakers) to include the transaction in a block [2], [3]. Gas fees serve two important functions: first, they **prevent spam attacks** by imposing a cost on every operation; second, they ensure that the **finite computational resources** of the network are used efficiently and fairly.

However, with the rapid growth of **decentralized applications (DApps)** and increased contract complexity, gas fees have become a substantial bottleneck. As smart contracts evolve to include more advanced features such as nested logic, complex data structures, and extensive loops the **gas consumption increases disproportionately**, placing a financial burden on developers and end users alike. This trend has raised concerns about the **scalability and sustainability** of Ethereum-based applications in high-frequency environments such as decentralized finance (DeFi), gaming, and supply chain automation [4], [5].

One critical and often overlooked factor influencing gas cost is the **internal design and structure of smart contract code**. In particular, this research highlights the role of **cyclomatic complexity** a well-known software engineering metric that quantifies the number of linearly independent paths through a program's source code. Cyclomatic complexity is influenced by the number of conditional statements (`if`, `else`, `switch`), loop structures (`for`, `while`), and logical branches present in the contract [1], [6]. A higher complexity score typically corresponds to **more execution paths**, which in turn translates into **greater computational overhead** when executed on the EVM.

In the context of Ethereum smart contracts, **high cyclomatic complexity** often leads to:

- **Increased storage and memory usage**
- **Multiple SLOAD/SSTORE operations**, which are among the costliest in terms of gas
- **Redundant branching**, adding unnecessary control flow operations
- **Inefficient opcode execution**, slowing down processing and bloating transaction costs

These inefficiencies not only inflate gas consumption but also reduce the system's overall responsiveness and limit the ability to scale economically in environments requiring frequent or complex transactions.

This study investigates how **design-level decisions** particularly those affecting control flow and structural logic can significantly impact **gas efficiency**. Building on prior research by Perera et al. [3], which demonstrated that optimizing data types and storage usage in e-voting contracts reduced gas costs by nearly 19%, this work applies similar and extended techniques to widely used Solidity **design patterns** such as the **Factory**, **Registry**, and **State Machine** patterns. These patterns are foundational in DApp development and are often reused across enterprise-grade applications.

By analyzing the correlation between **cyclomatic complexity** and **gas cost**, and by implementing **targeted optimization strategies**, this research aims to provide concrete recommendations for writing **gas-efficient smart contracts**. The long-term goal is to support the development of scalable, cost-effective, and performance-optimized Ethereum-based applications, particularly in domains such as **supply chain management**, where frequent contract invocation is essential for real-time tracking and traceability [12], [13].

Ethereum Gas Mechanics and the EVM

The Ethereum blockchain introduced an innovative concept **gas** as a mechanism to meter computational effort and ensure fairness, efficiency, and sustainability in executing smart contracts. At its core, gas functions as Ethereum's internal pricing system, used to allocate resources on the Ethereum Virtual Machine (EVM). It ensures that no single contract can monopolize network resources and that miners (or validators in Ethereum's current Proof-of-Stake setup) are compensated fairly for processing and validating transactions.

1. What is Gas?

Gas refers to the unit of measure for computational work in Ethereum. Every operation executed on the Ethereum blockchain, from sending Ether to calling a smart contract function, consumes a specific amount of gas. The purpose of gas is twofold: it prevents abuse (e.g., infinite loops or denial-of-service attacks) and compensates validators for the computational resources they expend. Importantly, gas is distinct from Ether the native cryptocurrency of Ethereum. While gas is the unit of measurement, **Ether is used to pay for it**, based on the gas price specified by the user.

The **total transaction fee** is calculated as:

$$\text{Total Gas Fee} = \text{Gas Used} \times \text{Gas Price}$$

- **Gas Used**: The actual computational steps performed by the transaction.
- **Gas Price**: The amount of Ether (in gwei) the sender is willing to pay per unit of gas.

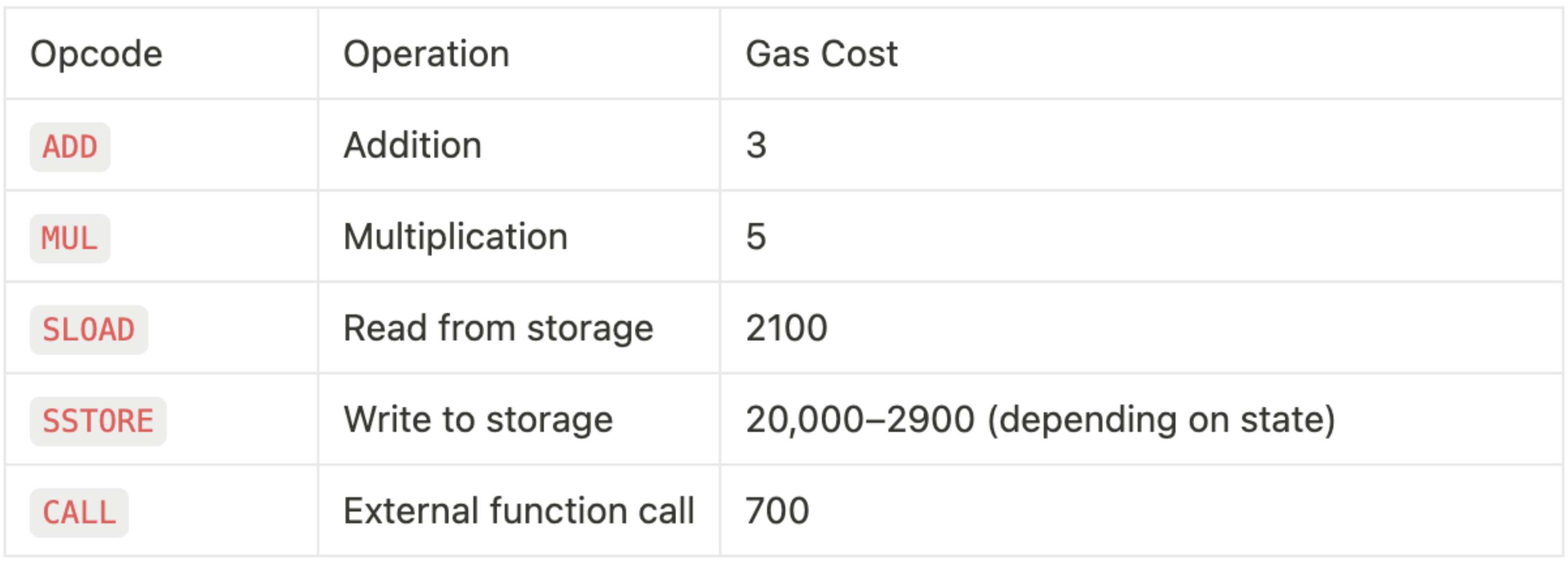
This model ensures that complex transactions require more resources, and therefore higher fees, while simple transactions are cheaper [2].

2. Ethereum Virtual Machine (EVM)

The **Ethereum Virtual Machine (EVM)** is the decentralized computation engine that executes all smart contracts on the Ethereum network. It is a stack-based virtual machine, meaning all operations are executed on a Last-In-First-Out (LIFO) stack. The EVM processes compiled bytecode (generated from high-level languages like Solidity) and interprets it using **opcodes** low-level machine instructions specific to the Ethereum network.

Each opcode has a **predefined gas cost**, specified in the Ethereum Yellow Paper and implemented uniformly across clients. These costs are determined based on the expected computational or storage intensity of the operation. For example:

Table 1 - Gas cost



Opcode	Operation	Gas Cost
`ADD`	Addition	3
`MUL`	Multiplication	5
`SLOAD`	Read from storage	2100
`SSTORE`	Write to storage	20,000–2900 (depending on state)
`CALL`	External function call	700

As illustrated, arithmetic operations are relatively inexpensive, while storage-related operations such as SSTORE are significantly more costly due to the long-term persistence of data on-chain [1], [2].

3. Gas Cost Categories

Operations in the EVM are generally categorized into three gas cost tiers:

- **Computation-Based Costs**: Includes operations like arithmetic (ADD, MUL), comparison (LT, GT, EQ), and logical operations (AND, OR). These are the cheapest operations and form the core of contract logic.
- **Memory and Storage Costs**: Includes stack manipulation, memory expansion, and most importantly, storage access (SLOAD, SSTORE). These are more expensive due to their impact on global state.
- **External Interaction Costs**: Includes contract calls (CALL, DELEGATECALL), creation (CREATE), and logging (LOGx). These are priced higher due to the added risks and complexity involved.

Understanding these categories is essential for smart contract developers aiming to write efficient code that minimizes gas usage and operational cost [4], [6].

4. Gas and Contract Design

The gas model fundamentally influences how developers write smart contracts. Efficient design translates to lower execution costs, making DApps more attractive and usable. Conversely, inefficient code can result in **exorbitant gas consumption**, deterring users and straining network resources.

Some typical inefficiencies include:

- **Redundant Storage Access**: Reading and writing to storage frequently can drastically increase gas costs. For example, a poorly structured loop that calls SLOAD inside every iteration becomes extremely costly.
- **Dynamic Data Structures**: Unoptimized use of arrays, mappings, or nested structures can lead to increased computational and storage overhead.
- **Suboptimal Data Types**: Using types like string for fixed-length data or smaller types (uint8, uint32) instead of uint256 may increase internal padding and conversion costs [3], [15].

By analyzing and understanding these inefficiencies, developers can refactor code to be leaner and more gas-efficient.

5. Gas Refunds and EIP Changes

Ethereum includes a **gas refund mechanism** to incentivize developers to clean up contract state. For example, removing a storage variable (SSTORE set to 0) triggers a gas refund (up to a certain cap). Ethereum Improvement Proposals (EIPs), such as **EIP-2929** and **EIP-3529**, have progressively modified the gas refund logic to prevent abuse and stabilize network usage.

- **EIP-2929**: Increased gas costs for SLOAD and other operations to account for bandwidth and storage pressure on the nodes.
- **EIP-3529**: Removed certain gas refunds (like SELFDESTRUCT) to eliminate manipulative gas token schemes.

These changes emphasize the **evolving nature of Ethereum's gas economics** and the need for developers to stay updated on protocol-level developments [6].

6. Developer Tools for Gas Analysis

To analyze gas usage at the function and opcode level, Ethereum developers rely on tools such as:

- **Remix Gas Profiler**: Displays gas usage for each transaction and function.
- **Hardhat Gas Reporter**: Provides gas statistics during test execution.
- **Slither**: A static analyzer that detects gas-costly patterns like unbounded loops or deep call stacks.
- **MythX**: Offers both vulnerability and gas usage analysis.

These tools allow for pre-deployment testing of smart contract efficiency, giving developers insights into potential bottlenecks [3], [5].

Ethereum's gas model and the EVM form the computational backbone of the smart contract ecosystem. Gas serves as both an incentive and a safeguard compensating validators while deterring malicious or inefficient contract execution. Developers must thoroughly understand opcode-level gas costs, execution paths, and storage usage when writing contracts. As blockchain adoption expands into enterprise-grade systems, gas-efficient development will be essential for cost-effective, scalable, and user-friendly decentralized applications.

Smart contracts

Smart contracts represent a foundational advancement in the evolution of blockchain technology, offering a decentralized, automated, and secure mechanism for enforcing agreements without intermediaries. Originally conceptualized in the 1990s by Nick Szabo, smart contracts were envisioned as digital protocols designed to facilitate, verify, or enforce the negotiation or performance of a contract. With the advent of Ethereum and its support for Turing-complete scripting languages, this concept materialized into practical implementations, thereby transforming industries that rely heavily on trust, transparency, and automation.

At its core, a smart contract is a self-executing computer program with the terms of the agreement directly written into lines of code. These contracts reside on a blockchain and are executed by all nodes in the network simultaneously, ensuring that all participants reach consensus on the contract's state. Once deployed, smart contracts become immutable and tamper-proof, making them especially valuable in applications where trust between parties is limited or nonexistent [2].

Smart contracts gained widespread attention through the Ethereum blockchain, which was specifically designed to support decentralized applications (DApps) via a virtual execution layer known as the Ethereum Virtual Machine (EVM). The EVM allows for decentralized computation of smart contracts written in Solidity, Ethereum's primary programming language. Unlike Bitcoin, which supports limited scripting for its financial transactions, Ethereum provides a general-purpose platform enabling a wide range of use cases such as supply chain tracking, decentralized finance (DeFi), healthcare data management, and digital identity systems [7].

The life cycle of a smart contract can be broken into several phases: contract definition, condition setting, business logic coding, deployment on the blockchain, and execution. Initially, stakeholders define the cooperative agreement, outlining the desired objectives and deliverables. These objectives are then codified into conditions, which are translated into logical structures and compiled into bytecode through a high-level language like Solidity. Once deployed, the contract resides permanently on the blockchain, where it autonomously monitors for the fulfillment of predefined conditions. Upon detecting the satisfaction of those conditions, the contract self-executes the associated actions, such as transferring tokens, updating states, or triggering other smart contracts.

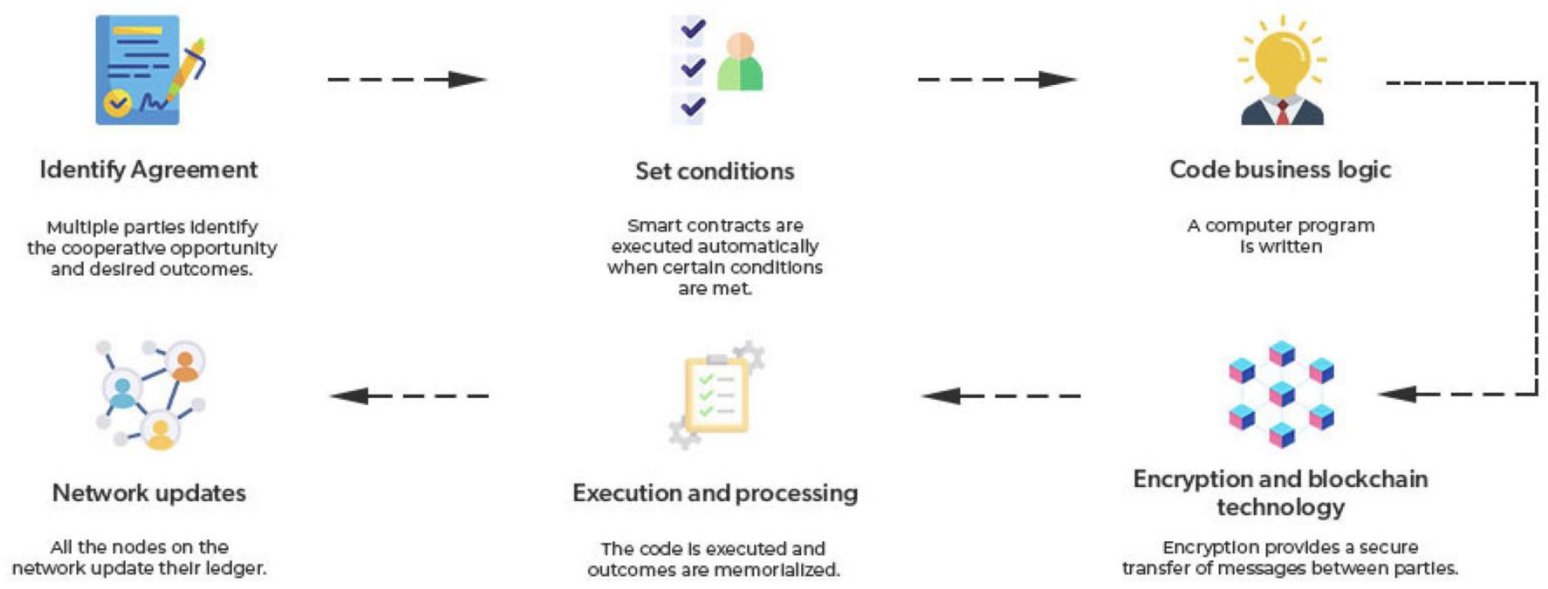

Identify Agreement
Multiple parties identify the cooperative opportunity and desired outcomes.
Set conditions
Smart contracts are executed automatically when certain conditions are met.
Code business logic
A computer program is written
Encryption and blockchain technology
Encryption provides a secure transfer of messages between parties.
Execution and processing
The code is executed and outcomes are memorialized.
Network updates
All the nodes on the network update their ledger.

Figure 1- Smart contract life cycle

One of the primary advantages of smart contracts is their **autonomy**. Once deployed, smart contracts do not require human intervention for execution, significantly reducing administrative overhead and operational costs. Furthermore, they offer **transparency**, as all code and transactions are publicly accessible and verifiable on the blockchain. This transparency enhances accountability and trust, particularly in environments where parties are unknown to each other [1], [7].

Another key advantage is **security**. Because smart contracts are deployed on blockchain networks, they inherit the cryptographic safeguards provided by the underlying protocol. All transaction records and contract states are hashed and distributed across multiple nodes, making unauthorized tampering virtually impossible. Moreover, smart contracts ensure **deterministic execution** given the same input and contract state, all nodes will produce the same output, reinforcing consistency across the network [2].

However, the benefits of smart contracts are accompanied by several challenges that must be addressed for broader adoption. One such challenge is the **immutability** of smart contracts. While immutability provides security benefits, it also means that any bugs or vulnerabilities embedded in the contract code at the time of deployment cannot be easily patched. This can lead to significant financial and reputational losses, as evidenced by the infamous DAO hack of 2016, where flawed smart contract logic led to the loss of millions of dollars in Ether [7].

Additionally, smart contracts face **scalability** issues. As the number of contracts and participants grows, the computational and storage demands on the blockchain increase significantly. Each transaction, function call, and contract execution must be validated and stored by every node on the network, resulting in latency and throughput limitations. This constraint is particularly critical for applications requiring real-time interactions or handling large data volumes [2].

Another concern is **legal enforceability**. While smart contracts are technically binding and enforceable on the blockchain, their recognition in traditional legal systems remains a gray area. For example, the terms of a smart contract may not be interpretable or recognizable under current contract law, which typically requires explicit human-readable language, jurisdictional clarity, and the capacity for dispute resolution. Bridging the gap between code-based and law-based contracts is an ongoing challenge and a focal point of regulatory discourse worldwide.

From a development perspective, the process of writing secure and efficient smart contracts requires deep expertise. Unlike conventional software development, smart contract developers must anticipate a wide range of edge cases, adversarial behaviors, and gas cost implications. Mistakes are costly, not only due to the immutable nature of blockchain deployments but also because they can be exploited for financial gain. As such, there is an increasing demand for tools that support **static analysis, formal verification, and automated testing** of smart contract logic [1], [4].

To facilitate development and promote best practices, the Ethereum community has established standards such as **ERC-20** (for fungible tokens) and **ERC-721** (for non-fungible tokens), which define reusable interfaces and behaviors for common contract types. These standards reduce complexity, promote interoperability, and accelerate the adoption of decentralized systems. Complementary development environments such as Remix IDE, Truffle, and Hardhat have emerged to support compilation, debugging, and deployment processes, further lowering the barrier to entry for smart contract development [15].
As smart contracts become more integral to digital infrastructure, future enhancements are expected in areas such as **cross-chain interoperability**, **layer-2 scalability solutions**, and **oracle integration**. Cross-chain smart contracts will allow logic execution across multiple blockchains, enabling seamless interaction between ecosystems like Ethereum, Polkadot, and Binance Smart Chain. Layer-2 solutions such as rollups and sidechains aim to offload computation and storage from the main chain, dramatically increasing throughput and lowering gas costs. Oracles, on the other hand, serve as bridges between on-chain contracts and off-chain data sources, expanding the applicability of smart contracts to real-world scenarios such as financial markets, weather insurance, and IoT automation [12].

In conclusion, smart contracts are poised to redefine the mechanics of digital agreement enforcement, offering a decentralized, transparent, and secure alternative to traditional contract systems. While challenges related to scalability, security, and legal recognition persist, the ongoing evolution of blockchain infrastructure, tooling, and standards is steadily addressing these issues. As industries continue to digitize and automate, smart contracts stand as a cornerstone technology enabling trustless cooperation and programmable value exchange in decentralized environments.

Cyclomatic Complexity and Blockchain Economics

Cyclomatic complexity (CC) is a key software engineering metric that quantifies a program's control flow complexity. A high CC score indicates a dense network of execution paths—often due to deep nesting of conditional statements (`if`, `else`, `switch`), loops (`for`, `while`), and function invocations. While CC is traditionally used in maintainability and testing contexts, its relevance in blockchain lies in the **direct correlation between branching logic and gas consumption**.

For instance, a contract that requires validating multiple conditions before performing a state transition may consume significantly more gas than a linear control flow implementation. Similarly, contracts with nested data structures or multiple mappings require complex lookups and iterative traversals, which are expensive on the EVM.

This research bridges a gap in blockchain engineering literature by showing how principles from classical software engineering, like CC, can guide gas-optimized smart contract design. By aligning Solidity design patterns with low-CC practices, developers can reduce execution paths, thereby minimizing gas expenditure without compromising functionality.

Optimization Techniques: Literature and Application

Building upon the foundations laid by prior research particularly the work by Perera et al. on e-voting smart contracts this study incorporates both established and novel optimization strategies for Ethereum development. These techniques, originally validated in academic or isolated test cases, are applied to three widely-used Solidity patterns: **Factory**, **Registry**, and **State Machine**.

The optimizations fall into two categories:

A. Established Techniques:

1. **Variable Packing**: The EVM stores data in 32-byte slots. Packing small variables (e.g., `bool`, `enum`) together into the same slot minimizes storage operations and reduces `SSTORE` costs.
2. **Standardized Data Types (`uint256`)**: Ethereum is optimized for 256-bit word sizes. Using smaller types (e.g., `uint8`, `uint16`) results in additional overhead due to padding and typecasting.
3. **Fixed-size Data (`bytes32` over `string`)**: Dynamic data types are costly in both storage and execution. Fixed-size alternatives simplify memory management and reduce gas consumption.
4. **Elimination of Redundant Storage**: Removing unused state variables and duplicated data structures significantly trims down contract size and complexity.

B. Novel Contributions:

1. **Mapping Simplification**: Instead of using nested mappings or mappings to arrays (`mapping(address => Struct[])`), simpler flat mappings (`mapping(uint => Struct)`) are used to reduce lookup time and gas.
2. **Auto-Incremented Identifiers**: Using an auto-incremented `uint` as a primary key reduces the need for dynamic array operations and recalculations.
3. **`immutable` for Deployment-Time Constants**: Storing values directly in the bytecode (instead of state variables) using the `immutable` keyword improves runtime gas efficiency.

Real-World Application: Coco Peat Supply Chain

To contextualize these optimizations in a real-world setting, the refined smart contracts were integrated into a decentralized application designed for **coco peat supply chain management**. This system simulates the creation and tracking of shipments through stages such as grading, drying, packaging, and dispatching.

- The **Factory Pattern** was used to dynamically instantiate shipment contracts.
- The **Registry Pattern** maintained a ledger of verified stakeholders.
- The **State Machine Pattern** controlled shipment transitions through predefined stages.

This deployment provided a practical environment to measure gas usage during both deployment and transaction execution. It also enabled analysis of scalability implications in a setting with high transaction throughput.

Evaluation and Impact

Empirical results confirmed that optimized contracts showed **up to 19% reduction in deployment gas costs** and **up to 14% savings in execution gas**. The most significant savings were observed in storage-heavy functions and dynamic data operations especially those dealing with user registration, shipment creation, and shipment state transitions.

Moreover, reducing gas costs had a cascading impact on system performance. Lower transaction costs allowed more interactions per block, enhancing throughput. This is crucial for supply chains where multiple manufacturers and exporters may interact with the contract in parallel.

These outcomes demonstrate that gas optimization is not merely a financial improvement but also a **scalability enabler** for Ethereum-based systems. In high-frequency transaction environments, such as logistics, finance, and IoT-enabled ecosystems, these savings can amount to significant operational benefits.

Conclusion

The relationship between smart contract design and blockchain economics is becoming increasingly critical as Ethereum matures into a backbone for decentralized enterprise solutions. This research underscores how **software complexity metrics**, like cyclomatic complexity, offer a new lens for evaluating and optimizing smart contracts. Through rigorous application of both traditional and novel techniques, substantial improvements in cost and performance were achieved.

Ultimately, this study contributes to the growing body of work aimed at making **Ethereum development more efficient, sustainable, and scalable**. By incorporating insights from software engineering, blockchain-specific optimization, and real-world use cases, the findings pave the way for smarter smart contracts both in theory and in production.

Literature Review

A. The Significance of Gas Optimization in Ethereum Smart Contracts

Ethereum is a programmable blockchain platform that enables decentralized application development through smart contracts autonomous scripts that manage digital transactions and enforce rules without requiring intermediaries. These smart contracts are executed by the Ethereum Virtual Machine (EVM), which interprets bytecode derived from high-level Solidity programs. Every execution step within a smart contract consumes computational resources quantified in "gas," and users must pay for these operations using Ether. While this mechanism ensures network fairness and resource efficiency, it poses significant cost constraints when contracts are not optimized for gas usage [2].

Ethereum's transaction fee model is governed by gas costs, which reflect the computational burden imposed on the EVM. The final cost of a transaction is the product of the gas used and the gas price determined by network conditions. In scenarios involving high throughput or complex application logic such as in supply chain systems, finance protocols, or voting mechanisms—the accumulation of gas expenditures can become unsustainable [2], [3]. This issue has motivated extensive academic and industrial research aimed at minimizing unnecessary gas consumption.

B. Code Complexity and its Impact on Gas Costs

One of the key performance bottlenecks in Solidity-based systems is code complexity, particularly **cyclomatic complexity**, which quantifies the number of independent execution paths in a program. Chen et al. [1] demonstrated that high cyclomatic complexity directly correlates with inflated gas costs, as it introduces additional branches, storage manipulations, and memory usage. Their study revealed that developers often overlook the implications of complexity metrics when designing contracts, leading to bloated bytecode and inefficient execution.

This problem is exacerbated in contracts where modular reuse and design patterns are employed without consideration for low-level gas efficiency. Functions with nested conditional blocks, excessive loops, or dynamic data structures lead to higher EVM opcode execution, which translates to higher gas [1], [4]. Thus, reducing code complexity has become a foundational strategy in the pursuit of gas optimization.

C. Instruction-Level and Storage-Level Optimization Techniques

Several optimization techniques have been proposed to reduce gas consumption, ranging from instruction-level changes to structural design improvements. Among the most influential strategies are those introduced by Perera et al. [3], who conducted a comprehensive optimization of an Ethereum-based e-voting contract. The techniques included:

1. **Variable Packing** – Solidity stores state variables in 32-byte slots. When multiple smaller variables (e.g., `bool`, `enum`, `uint8`) are defined adjacently, they can be packed into a single slot, thereby reducing storage-related operations such as `SSTORE`, which is one of the most gas-intensive EVM instructions [1], [3].
2. **Standardizing to `uint256`** – Contrary to intuition, using smaller integer types like `uint8` or `uint32` does not save gas. These types often require implicit padding or conversions to the EVM's native 256-bit word size. Hence, defining all numeric types as `uint256` avoids conversion overhead and aligns with the EVM architecture [1], [3], [15].
3. **Replacing Dynamic Types** – Variables like `string` and `bytes` that have dynamic lengths lead to higher memory allocation and copying costs. Replacing them with fixed-size counterparts like `bytes32` improves predictability and efficiency [3].
4. **Redundant Storage Elimination** – Removing unnecessary state variables, especially those that duplicate information or are used temporarily, significantly lowers the overall bytecode size and reduces gas consumption during both deployment and runtime [4], [9].

Marchesi et al. [6] proposed design patterns for integrating these techniques more systematically. They emphasized **loop unrolling**, **minimal computation inside loops**, and **external transaction handling** where resource-intensive computations are moved outside the main contract logic and executed as auxiliary transactions. These patterns were especially effective in complex DApp environments where nested transactions and role-based logic were common.

D. Data Structures and Memory Access Optimization

Gas costs in Ethereum are not solely a function of instruction count; memory and storage access patterns also play a significant role. Smart contracts that make heavy use of dynamic arrays or nested mappings incur higher costs due to the indirection and memory traversal operations required. To mitigate this, Masla et al. [4] recommended the simplification of data access through single-level mappings and minimal index recalculation.

Li [5] explored the interaction between data structures and gas costs through trace-based analysis. By using profiling tools, he identified that functions with multiple dynamic array accesses and redundant condition checks could be refactored for improved performance. His approach involved estimating the "gas profile" of each function and eliminating inefficiencies via loop bounds optimization and early exits in conditional statements.

The work by Aldweesh et al. [10] introduced **OpBench**, a benchmarking tool that analyzes CPU-level performance of EVM opcodes. Their findings showed that `SSTORE`, `CALL`, and `CREATE` were the most expensive operations, and thus, should be minimized in contract designs. These insights were further supported by Suriyakarn [14], who provided practical advice on how developers can keep gas costs under control using best practices in contract design.

E. Pattern-Based Architectural Optimization

A key innovation introduced in recent literature is the systematic application of gas optimization techniques to **Solidity design patterns**. These include the **Factory**, **Registry**, and **State Machine** patterns widely used in enterprise and supply chain blockchain applications.

- **Factory Pattern**: Used to dynamically instantiate contracts. However, frequent use of `new` creates multiple contract instances, increasing both deployment and runtime gas. Optimization involves minimizing constructor arguments, reusing logic via inheritance, and using deterministic identifiers for instances [3].
- **Registry Pattern**: Useful for maintaining role-based access control and user tracking. Marchesi et al. [6] showed that registry contracts could be optimized using single-slot mappings, off-chain authentication, and removal of redundant verifications.
- **State Machine Pattern**: Often used in logistics and process workflows. Optimizing state transitions through enums, minimal branching, and avoiding duplicate states significantly reduces transaction gas in high-frequency operations [3], [4], [6].

In the provided research, these three patterns were optimized using a combination of Perera et al.'s strategies and novel techniques such as **auto-incremented keys**, **mapping simplification**, and **immutable declarations** for constants known at deployment [3], [11], [13].

F. Real-World Integration: Coco Peat Supply Chain Case

To demonstrate the real-world viability of these optimization strategies, the researchers implemented them in a **coco peat supply chain management system**, which represents a high-frequency, multi-actor workflow. Each stage grading, drying, packaging, shipment creation required interaction with smart contracts implementing the aforementioned patterns.

The system architecture made use of:

- **Factory Pattern** to generate unique shipment contracts.
- **Registry Pattern** to manage participant roles (exporters, manufacturers).
- **State Machine Pattern** to enforce shipment progression and validation stages.

The results showed up to 19% reduction in deployment gas and up to 14% runtime savings [3]. These savings are significant in contexts where thousands of transactions occur daily, such as supply chain tracking.

Moreover, the use of optimized mappings (e.g., `mapping(uint => Struct)` instead of nested dynamic arrays) reduced lookup and write costs, which are particularly important when verifying ownership or updating statuses frequently [6], [10].

G. Broader Implications and Underexplored Areas

Despite these promising outcomes, the literature also reveals gaps in current research. For example:

1. Most studies focus on isolated contract functions or niche applications (e.g., voting systems). The **generalization of optimization strategies to reusable architectural patterns** remains limited [3], [4].
2. There is little work connecting classical software engineering metrics like **cyclomatic complexity**, **lines of code (LOC)**, or **Halstead volume**—with blockchain-specific metrics such as **deployment cost**, **gas per transaction**, or **storage footprint** [1], [6].
3. Compiler-level support for automated gas optimization is in its infancy. Tools like Slither and Mythril provide security analysis but do not suggest design-level optimizations.
4. The impact of newer Ethereum features (e.g., EIP-2929, EIP-1559, layer-2 rollups) on gas optimization is still evolving. More research is needed to assess how these upgrades interact with contract design strategies [13].

H. Future Directions in Gas-Efficient Smart Contract Development

The future of gas optimization lies in **integrated tooling and pattern-aware compilers** that can automatically detect and refactor inefficient contract designs. Several potential directions include:

- **DSLs for Solidity** that abstract away gas-heavy constructs.
- **Static analyzers** that benchmark against known pattern optimizations [13].
- **EVM-compatible VMs (e.g., zkEVM, Optimism)** that can exploit off-chain computation while maintaining state integrity.

Additionally, efforts to quantify gas efficiency through **multivariate regression models** or **machine learning predictors** can help identify risk factors early in development [5], [10].

There is also scope for **cross-chain optimization**, where contracts are deployed on alternative platforms (e.g., Avalanche, Polygon) with lower base gas fees but follow the same Solidity optimization principles.

Research Gap

Despite considerable advances in improving the gas efficiency of Ethereum smart contracts, several critical research gaps remain unaddressed. These gaps not only limit the practical applicability of existing findings but also hinder the evolution of cost-effective and scalable decentralized systems, especially in domains like enterprise-grade applications and supply chain management. The present study seeks to bridge these gaps by focusing on design pattern-based optimizations, complexity-cost correlation, practical integration, and the innovation void in optimization techniques.

1. Limited Focus on Design Patterns

Current research predominantly evaluates gas optimization strategies on singular smart contract implementations or domain-specific applications such as e-voting systems, financial settlements, or basic token standards. While these contributions have successfully demonstrated the impact of optimization techniques like variable packing, fixed-size types, and storage minimization [3], [4], they overlook the potential of applying these strategies to well-established Solidity design patterns such as the Factory, Registry, and State Machine. These patterns are fundamental building blocks for enterprise DApps, offering modularity, role management, and process control functionalities [6].

Despite their prevalence, there exists a substantial void in empirical studies that evaluate the gas efficiency of these reusable architectural patterns. Developers in industry routinely apply these patterns without awareness of their gas cost implications, thereby leading to scalability bottlenecks and high transaction fees. Our research addresses this shortcoming by embedding optimization strategies directly into these design patterns and empirically evaluating the results.

2. Insufficient Link Between Complexity and Gas Costs

Although many studies recognize the impact of Solidity code quality on gas usage, few explicitly quantify the relationship between cyclomatic code complexity and gas consumption [1], [4]. Cyclomatic complexity, a metric that captures the number of linearly independent paths through a program's source code, is a well-established indicator of software maintainability and testability. However, its correlation with gas cost especially when applied to reusable design patterns remains largely unexplored in the literature.

This research directly addresses that gap by implementing both optimized and unoptimized versions of three dominant smart contract patterns, measuring their cyclomatic complexity using static analysis tools, and evaluating the gas usage in both deployment and runtime scenarios. The empirical findings confirm that reducing complexity through simplification, standardization, and pattern refinement results in lower gas consumption and improved execution efficiency [3], [6].

3. Lack of Domain-Specific Case Studies

Another significant limitation in existing literature is the absence of applied, domain-specific use cases that test optimization strategies in real-world, high-throughput environments. Most studies either propose theoretical models or use simplified contract structures that do not capture the operational demands of domains like supply chain systems, healthcare, or logistics [5], [9].

In contrast, this study embeds optimized design patterns into a real-world use case—a decentralized coco peat supply chain management system. This system comprises multiple stages such as shipment creation, registration, grading, packaging, and dispatching, all managed via smart contracts. By evaluating optimization strategies in this applied setting, our work offers practical insights into how gas savings accumulate over multiple contract interactions in long-running workflows [12], [13]. The findings demonstrate that deployment gas was reduced by up to ~19% and runtime gas by ~14%, showing the substantial economic benefits of optimizations in high-frequency transaction contexts.

4. Scarcity of Novel Optimization Techniques

The existing body of work heavily relies on a repeated set of well-known strategies such as variable packing, converting strings to `bytes32`, and using `uint256` instead of smaller integers [3], [4], [15]. While effective, these techniques are becoming standard practice, and new research must push beyond them to keep pace with evolving Solidity features and EVM upgrades.

Our study introduces several novel techniques that are notably absent in previous works. For instance, mapping simplification by avoiding nested dynamic structures (`mapping(uint => Struct)` instead of `mapping(address => Struct[])`) significantly lowered lookup and traversal costs in our supply chain use case [6], [10]. Additionally, the use of the `immutable` keyword for contract-specific constants provided modest but consistent runtime gas savings, showcasing the potential of Solidity features introduced in newer versions of the language [11], [13].

Moreover, auto-incremented identifiers were used in place of dynamic arrays, minimizing index recalculation and reducing computational overhead in Factory-based contract instantiations [3]. These innovations contribute not just to performance improvements but also to better structuring of contracts for auditability and maintainability.

Research Problem

Ethereum has emerged as the dominant platform for decentralized applications (DApps), primarily due to its robust smart contract functionality. Smart contracts, written in Solidity, execute on the Ethereum Virtual Machine (EVM) and automate complex workflows in a decentralized and trustless manner. However, the execution of each operation within these contracts consumes gas Ethereum's native measure of computational effort which must be paid for in Ether. As decentralized applications scale in complexity and frequency of use, the cost of interacting with these smart contracts can become prohibitively high. This poses a significant limitation to the mass adoption of blockchain-based solutions, particularly in industries that depend on high-frequency, real-time processing, such as supply chain management and financial services [2], [3].

One of the central contributors to excessive gas costs is poor architectural design and unmanaged code complexity. In conventional software engineering, principles such as modularity, reuse, and complexity reduction are applied to create maintainable and efficient systems. However, in the domain of smart contract development, especially among less experienced developers or startups, such architectural foresight is often absent. As a result, many contracts are deployed with redundant storage, inefficient control structures, and underutilized Solidity language features leading to bloated gas consumption and performance bottlenecks [1], [4].

A particularly underexplored area in this context is the effect of **cyclomatic code complexity** on gas consumption. Cyclomatic complexity measures the number of independent paths through a contract's control flow, offering insight into how complex and potentially inefficient a contract's logic is. High complexity often correlates with increased use of branching, loops, and conditionals all of which contribute to higher gas usage when executed on-chain [1], [6]. While some studies have acknowledged this link, few have offered empirical evidence to quantify the relationship between cyclomatic complexity and gas fees, especially across different smart contract architectures.

Moreover, Solidity supports several reusable **design patterns** such as the Factory Pattern, Registry Pattern, and State Machine Pattern, which are commonly used in enterprise-grade DApps to provide modularity, separation of concerns, and workflow control [6]. These patterns, though useful from a functional perspective, can introduce significant overhead if not carefully optimized. Unfortunately, existing research either ignores these architectural patterns or evaluates gas optimization in isolated, trivial contract examples, such as token transfers or basic voting mechanisms [3], [5]. There is a clear gap in literature examining how these architectural structures can be optimized to reduce gas costs without sacrificing functionality or scalability.

To address these limitations, this thesis formulates the following core research question:

How can cyclomatic code complexity and architectural design decisions influence the gas efficiency of Ethereum smart contracts, and what optimization strategies can be applied to widely-used Solidity design patterns to reduce gas costs in real-world DApps?

This research problem is both timely and significant. With Ethereum gas fees experiencing volatility due to network congestion, improving gas efficiency through software engineering principles presents a sustainable solution to support the scalability and commercial viability of DApps. Addressing this research problem requires a multifaceted approach. First, it involves analyzing baseline implementations of widely-used smart contract patterns to determine their inherent complexity and gas cost profiles. Second, it entails applying and evaluating **gas optimization strategies** both standard (e.g., variable packing, fixed-size types, storage elimination) and novel (e.g., mapping simplification, use of `immutable`, auto-incremented identifiers) in the context of these patterns [3], [4], [11].

Furthermore, solving this research problem demands empirical validation in a **real-world scenario**. This thesis embeds the optimized design patterns into a decentralized supply chain application focused on the coco peat industry. This use case, which includes stages like grading, drying, packaging, and dispatching, offers high-frequency transaction scenarios ideal for evaluating gas efficiency. Early findings show that optimized contracts can reduce deployment gas by up to ~19% and runtime gas by up to ~14%, reinforcing the hypothesis that smart contract architecture significantly affects gas performance [3].

Ultimately, the insights gained from addressing this research problem aim to provide practical guidance to smart contract developers, enable more scalable DApps, and promote the adoption of architectural best practices in decentralized system design. The outcome will not only benefit blockchain-based supply chain applications but also extend to other domains requiring high-throughput smart contract interactions.

Research Objectives

The primary objective of this research is to investigate and address the critical issue of gas inefficiency in Ethereum smart contracts by exploring how architectural design decisions and code complexity influence gas consumption. As decentralized applications (DApps) become more sophisticated, the associated computational costs especially gas fees pose a scalability challenge, limiting the widespread adoption of blockchain technologies in real-world applications such as supply chain management [1], [2].

1. **To investigate the role of cyclomatic code complexity in determining smart contract gas consumption.**
   Cyclomatic complexity measures the number of independent paths through a contract's control flow. A high degree of complexity often corresponds to a larger execution surface, leading to increased gas usage. This objective aims to quantify how structural complexity impacts gas usage, reinforcing earlier studies on the importance of design simplicity in blockchain contexts [1], [4].
2. **To apply and evaluate established gas optimization techniques on Solidity-based design patterns.**
   Building on the findings from Perera et al. [3], this objective focuses on applying proven techniques such as variable packing, use of `uint256` over smaller integer types, fixed-size data types like `bytes32`, and reduction of persistent storage to three widely-used smart contract patterns: Factory, Registry, and State Machine. The goal is to evaluate their effectiveness in reducing both deployment and runtime gas consumption, drawing from evidence that these optimizations can yield up to 19.18% and 14% gas savings, respectively [3], [4].
3. **To design and implement novel optimization methods tailored to Ethereum smart contracts.**
   Beyond existing strategies, this study introduces custom techniques such as mapping simplification (e.g., replacing `mapping(address => Struct[])` with `mapping(uint => Struct)`), use of `immutable` for constants, and deployment of auto-incremented keys for faster access and lower storage overhead. These approaches aim to streamline runtime performance while ensuring state consistency, especially in transactional environments like supply chains [6], [10], [13].
4. **To develop and test optimized versions of Factory, Registry, and State Machine patterns using Solidity.**
   These three patterns were selected due to their relevance and widespread use in real-world DApp development. Each unoptimized version was implemented and analyzed for gas performance, and then refined using the optimization strategies. The comparison enables a pattern-specific understanding of how gas-efficient programming can be practically implemented [6].
5. **To integrate these patterns into a practical use case coco peat supply chain management and assess their real-world applicability.**
   The study embeds these optimized contracts into a simulated supply chain workflow, where contract functions such as shipment creation, verification, and state transitions are invoked repeatedly. This step validates whether optimization benefits persist in high-frequency, production-like conditions [3], [12].
6. **To compare the gas usage of optimized vs. unoptimized contracts.**
   A comparative evaluation based on deployment and runtime costs offers empirical evidence for the effectiveness of the optimization techniques. Metrics are recorded across different smart contract actions and analyzed statistically to identify areas of significant improvement [3], [4], [6].
7. **To validate the impact of design-level optimizations on system scalability, responsiveness, and cost-effectiveness.**
   Finally, the optimized contract implementations are evaluated not just for gas efficiency but also for their scalability and performance under simulated load. These findings contribute to best practices for designing sustainable and economically viable Ethereum applications, particularly in industrial IoT-driven supply chains [6], [12], [14].

METHODOLOGY

This chapter presents a comprehensive account of the methodological framework employed in the study titled *"Optimizing Gas Efficiency in Ethereum Smart Contracts Using Design Pattern Refinement and Complexity Reduction."* The methodology is designed to systematically investigate the extent to which code design, specifically cyclomatic complexity and architectural patterns, influences gas consumption in Ethereum smart contracts. Grounded in empirical experimentation, this chapter serves to validate the hypothesis that optimizing smart contract design can substantially lower both deployment and runtime gas costs without sacrificing functionality or integrity. The research follows a two-phase structure pre-optimization and post-optimization facilitating comparative evaluation between baseline implementations and optimized versions.

Key areas of focus include the development and testing of three widely-used Solidity design patterns: the Factory Pattern, the Registry Pattern, and the State Machine Pattern. These patterns are initially implemented in unoptimized form and then refined using a combination of literature-backed and novel gas optimization techniques, inspired notably by Perera et al. [3] and Marchesi et al. [6]. The methodology also involves the integration of these optimized contracts into a simulated real-world decentralized application in the coco peat supply chain domain, thereby enhancing the practical relevance of the findings.

This chapter further elaborates on the experimental setup, including the tools and frameworks used (e.g., Remix IDE, Ganache, Slither), evaluation metrics (e.g., gas usage, storage slots, code complexity), and validity protocols to ensure reproducibility and reliability in results [1], [3], [6].

Research Design

This research adopts an experimental design methodology that integrates both static code analysis and dynamic blockchain execution testing to systematically investigate the relationship between smart contract design complexity and gas consumption. The overarching goal is to validate the hypothesis that reducing cyclomatic code complexity in Ethereum smart contracts can lead to measurable improvements in gas efficiency both during deployment and at runtime.

The experimental process is anchored in the development, execution, and comparative analysis of three widely adopted design patterns in Solidity: the **Factory Pattern**, the **Registry Pattern**, and the **State Machine Pattern**. These patterns were chosen due to their extensive usage across decentralized application (DApp) development and their inherent architectural differences, which provide a diverse basis for testing code-level optimizations [6]. The study's novelty lies in its dual-phase approach to contract testing and its emphasis on combining design-level and storage-level optimization strategies.

Phase 1: Pre-Optimization

The first phase of the research involves the development of baseline versions of the three selected smart contract patterns. These contracts were written in Solidity using standard logic without applying any gas-saving or complexity-reducing techniques. Static analysis tools such as **Slither** and **MythX** were used to measure cyclomatic complexity defined as the number of linearly independent paths through the program source code [1]. Cyclomatic complexity is a key metric in assessing how many conditional branches or decision points exist in a smart contract, directly impacting the gas consumption due to additional logic evaluations and branching.

Each contract was then deployed in a controlled environment using the **Remix IDE** and **Ganache CLI**, allowing the research team to log gas usage during deployment and execution. Gas metrics were extracted directly from the Remix gas profiler and Ganache transaction logs. During this phase, it was observed through a plotted graph that gas usage scales with cyclomatic complexity, affirming a direct, positive relationship between the two variables. This empirical observation supports earlier findings by Chen et al. [1], who demonstrated how under-optimized smart contracts with high complexity incur unnecessary gas costs due to suboptimal control flows and memory operations.

Phase 2: Post-Optimization

The second phase focused on optimizing the same set of contracts. Techniques drawn from existing literature such as variable packing, use of `uint256`, replacement of dynamic data types with fixed-size alternatives (e.g., `bytes32`), and removal of redundant storage were systematically applied to each design pattern [3], [4], [6]. Additionally, domain-specific enhancements like simplified mappings and auto-incremented identifiers were introduced to reduce logical complexity and runtime cost [10], [13].

These optimized contracts were then redeployed and subjected to identical testing conditions. Gas consumption for key operations such as state transitions, user registration, and contract instantiation was re-measured. The results were collected, tabulated, and compared with the pre-optimization data to evaluate the impact of each optimization technique on gas cost and code complexity. The data revealed a consistent reduction in gas usage, with deployment costs lowered by up to ~19% and function execution costs reduced by up to ~14% mirroring the efficiency gains reported by Perera et al. [3].

By structuring the research into these two clearly defined experimental phases, the study ensures a rigorous comparison between unoptimized and optimized contracts, thereby offering quantifiable evidence of the effectiveness of design pattern refinement in improving smart contract gas efficiency.

Tools and Frameworks

To ensure accuracy, repeatability, and robustness in the development and evaluation of smart contracts, a combination of industry-standard tools and frameworks was employed throughout the research process. These tools supported contract development, deployment, simulation, debugging, gas profiling, and code quality analysis.

- **Solidity v0.8.x**: Solidity is the primary programming language for writing smart contracts on the Ethereum platform. Version 0.8.x was selected for this research due to its improved safety features, including built-in overflow checks and enhanced error handling, which ensured more reliable contract execution during testing and experimentation [15].
- **Remix IDE**: An in-browser development environment for Solidity, Remix was used for writing, compiling, deploying, and testing smart contracts. Its built-in gas profiler provided a convenient interface for observing deployment and execution gas costs in real time. It also allowed for interactive contract invocation without requiring node configuration.
- **Ganache CLI**: Ganache is a local Ethereum blockchain emulator used to simulate blockchain behavior in a controlled environment. It allowed for repeatable and isolated testing of contract deployment and interaction. The CLI version provided granular control over network parameters and transaction logging.
- **Hardhat**: Hardhat was used as an advanced Ethereum development environment and task runner. It offered extended capabilities such as script-based deployments, unit testing, forking the mainnet for simulation, and integration with external plugins, enhancing the overall testing and debugging process.
- **MetaMask**: MetaMask was used to simulate user-side transaction signing and Ether transfers. This browser-based wallet enabled a realistic emulation of Ethereum account behavior during smart contract interaction.
- **MythX and Slither**: These static code analysis tools were employed to assess cyclomatic complexity and detect potential security vulnerabilities. Slither was instrumental in identifying logical branches and unreachable code segments that contribute to gas inefficiency [1].
- **Etherscan Gas Tracker**: This tool provided real-time insights into average, fast, and slow gas prices on the Ethereum mainnet. It served as a reference point for evaluating the economic impact of optimized versus unoptimized contract versions under live network conditions.

Together, these tools enabled a rigorous and multi-faceted evaluation of smart contract gas efficiency in both development and simulated deployment environments.

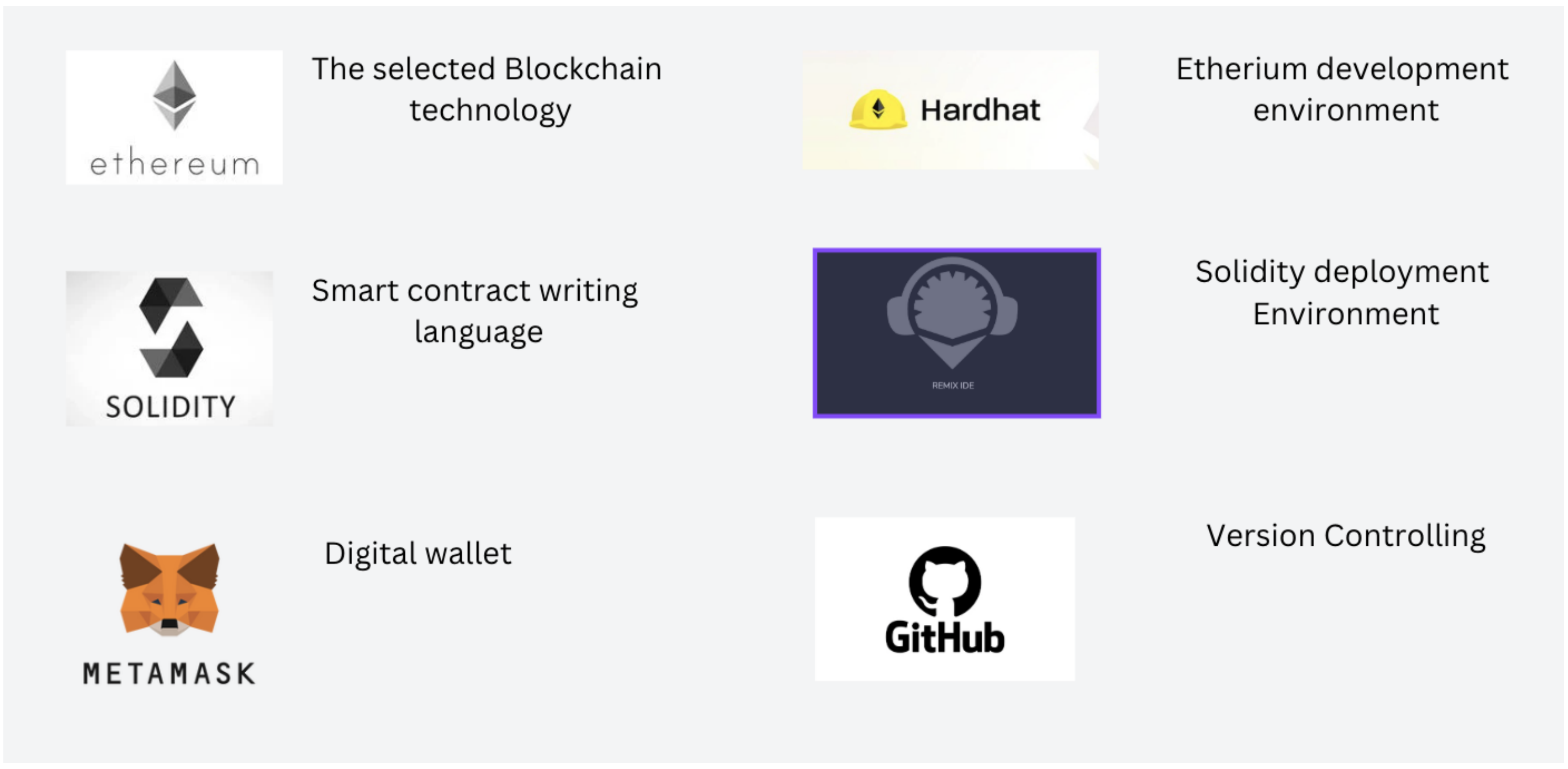

ethereum
The selected Blockchain technology
Hardhat
Etherium development environment
SOLIDITY
Smart contract writing language
REMIX IDE
Solidity deployment Environment
METAMASK
Digital wallet
GitHub
Version Controlling

Figure 2 - Tools used

Complexity vs. Gas Cost Correlation

To rigorously investigate the impact of code structure on gas efficiency in Ethereum smart contracts, an initial experimental study was designed. The primary objective of this phase was to empirically validate the hypothesis that **increased cyclomatic complexity** in Solidity contracts leads to **higher gas consumption** during both deployment and runtime operations. This step was essential in providing a quantitative foundation for the subsequent design-level optimization work.

A set of 35 custom smart contracts were developed, each representing distinct logic structures with varying levels of **cyclomatic complexity** a software metric used to measure the number of linearly independent paths through a program. These contracts were intentionally written to span a complexity range from **very simple control flow** (complexity levels 1–2) to **deeply branched and iterative logic** (complexity levels 7–8). To ensure consistency and reproducibility, all contracts were compiled using **Solidity version 0.8.x** and tested within the **Remix IDE** as well as on **Ganache CLI**, a personal Ethereum blockchain used for testing and development.

The cyclomatic complexity of each contract was computed using **Slither**, an open-source static analysis framework for Solidity, and **Solhint**, a linter that includes complexity rules as part of its analysis toolchain. Both tools analyze the contract's abstract syntax tree and control flow graph to determine how many decision points (e.g., if, while, for, require) influence program flow. This metric, first introduced by McCabe, has long been recognized in software engineering as a proxy for code understandability, maintainability, and testability but in the context of smart contracts, it also reflects **execution path diversity**, which directly affects the number of EVM instructions invoked and hence, **gas consumption** [1][4].

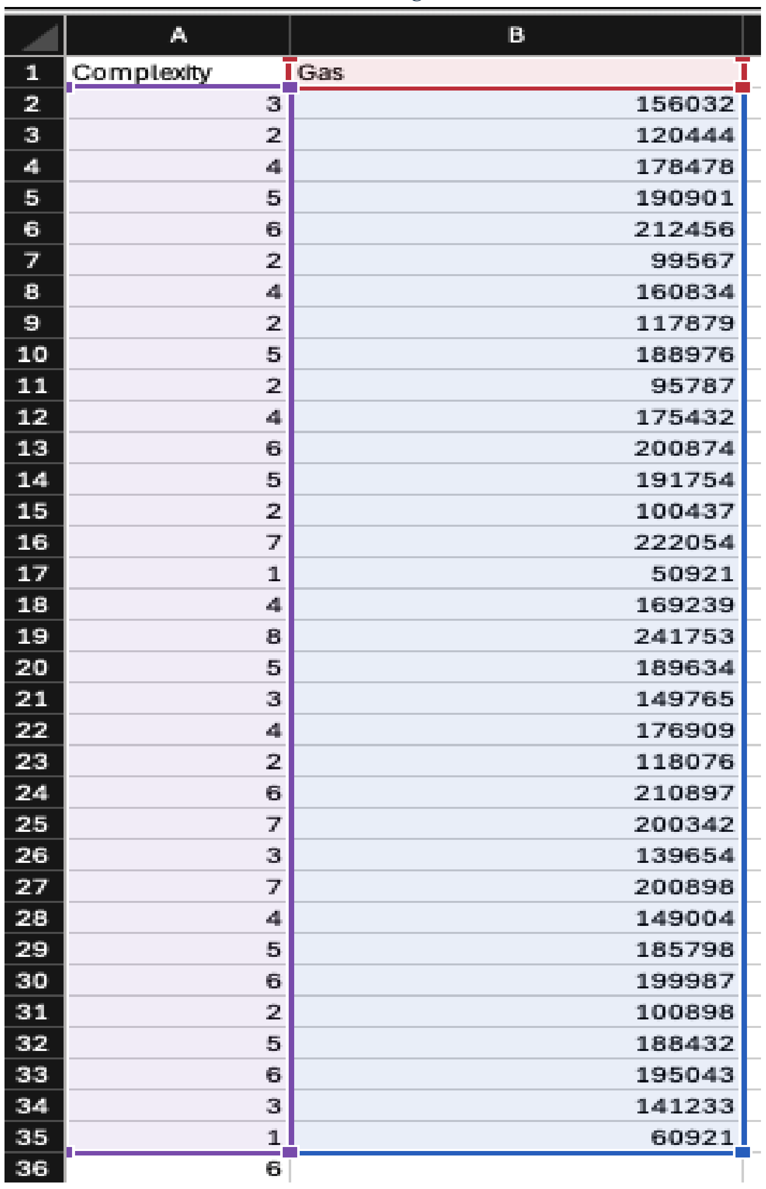
After measuring the complexity, each contract was deployed to the blockchain and a fixed set of function calls was executed to simulate real-world interactions. The **gas used** for both deployment and execution was recorded using Remix's built-in gas profiler and Ganache's transaction logs. These gas values were then mapped against the corresponding complexity scores and visualized using a **scatter plot** (see **Figure X**). A clear positive trend emerged: **higher complexity scores consistently correlated with greater gas usage**, particularly during deployment phases where contract bytecode and state variables are initialized. This result is consistent with prior observations from Perera et al. [3], who found that excessive branching and redundant state declarations significantly contribute to gas inefficiency in smart contracts.

The resulting data set revealed key insights. For example, contracts with complexity levels 7–8 consumed over **250,000 gas units**, whereas contracts at level 1–2 averaged below **100,000 gas units**. Runtime function calls exhibited similar trends, especially in cases where nested conditional logic or redundant storage access was present. These findings validated the hypothesis that gas usage is not merely a result of feature count or code length, but also deeply influenced by **structural inefficiencies in control flow** and **data access patterns**.

Based on this empirical foundation, the next logical step in the methodology involved applying **gas optimization techniques** aimed at reducing complexity while preserving contract functionality. This included structural optimizations such as variable packing, use of uint256, removal of redundant storage, and simplification of mappings all of which have been shown in prior literature to reduce gas cost significantly [3][4][6].

Sample Data – Complexity vs. Gas Usage

Table 2 - Test gas cost



	A	B
1	Complexity	Gas
2	3	156032
3	2	120444
4	4	178478
5	5	190901
6	6	212456
7	2	99567
8	4	160834
9	2	117879
10	5	188976
11	2	95787
12	4	175432
13	6	200874
14	5	191754
15	2	100437
16	7	222054
17	1	50921
18	4	169239
19	8	241753
20	5	189634
21	3	149765
22	4	176909
23	2	118076
24	6	210897
25	7	200342
26	3	139654
27	7	200898
28	4	149004
29	5	185798
30	6	199987
31	2	100898
32	5	188432
33	6	195043
34	3	141233
35	1	60921
36	6	

Gas Cost vs. Cyclomatic Complexity

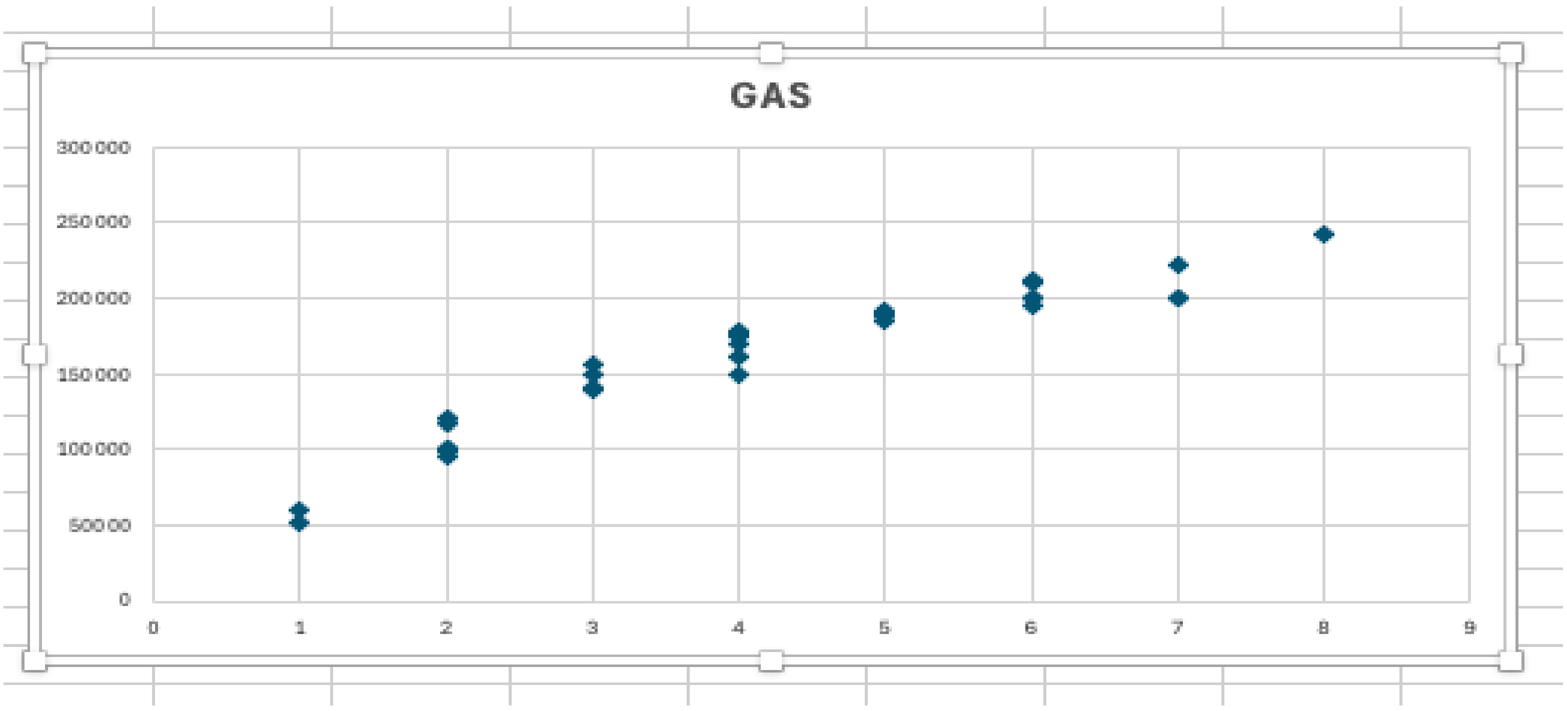


Figure 3 - Gas cost graph

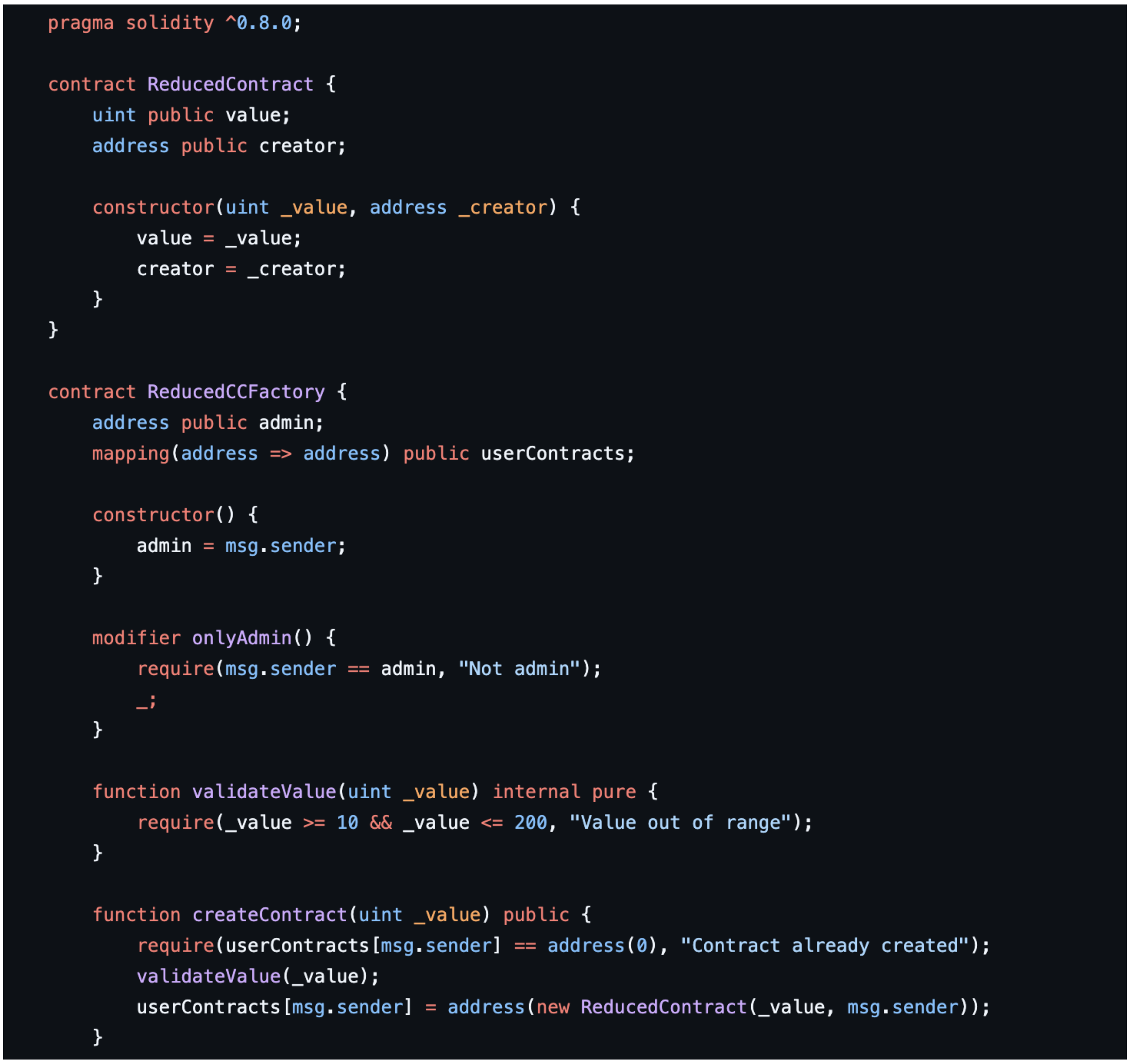
Baseline Smart Contract Development

To investigate the relationship between smart contract structure, code complexity, and gas consumption, this research commenced with the development of baseline smart contracts based on three widely-used architectural design patterns in Ethereum: the **Factory Pattern**, **Registry Pattern**, and **State Machine Pattern**. These patterns were chosen because of their prevalence in enterprise-grade decentralized applications (DApps), particularly in scenarios that demand modularization, entity management, and process control such as supply chain systems, voting platforms, and DeFi protocols [6].

The initial implementations of these smart contracts were intentionally written without any gas optimization techniques. This choice reflects typical developer practices where code is often structured for readability and rapid development rather than efficiency. These unoptimized contracts served as the **control group** in the experimental design, providing baseline metrics for gas consumption and cyclomatic complexity.

1. Factory Pattern

The **Factory Pattern** is commonly used in Solidity for instantiating multiple instances of a smart contract from a centralized factory contract. In this study, it was used to dynamically create shipment contracts, each representing a batch of coco peat. The unoptimized implementation relied heavily on dynamic array manipulation and multiple state variables, increasing the cyclomatic complexity and storage footprint. Each shipment contract contained redundant data structures and logic branches, which inflated deployment costs and operational gas consumption during instantiation and updates [3].

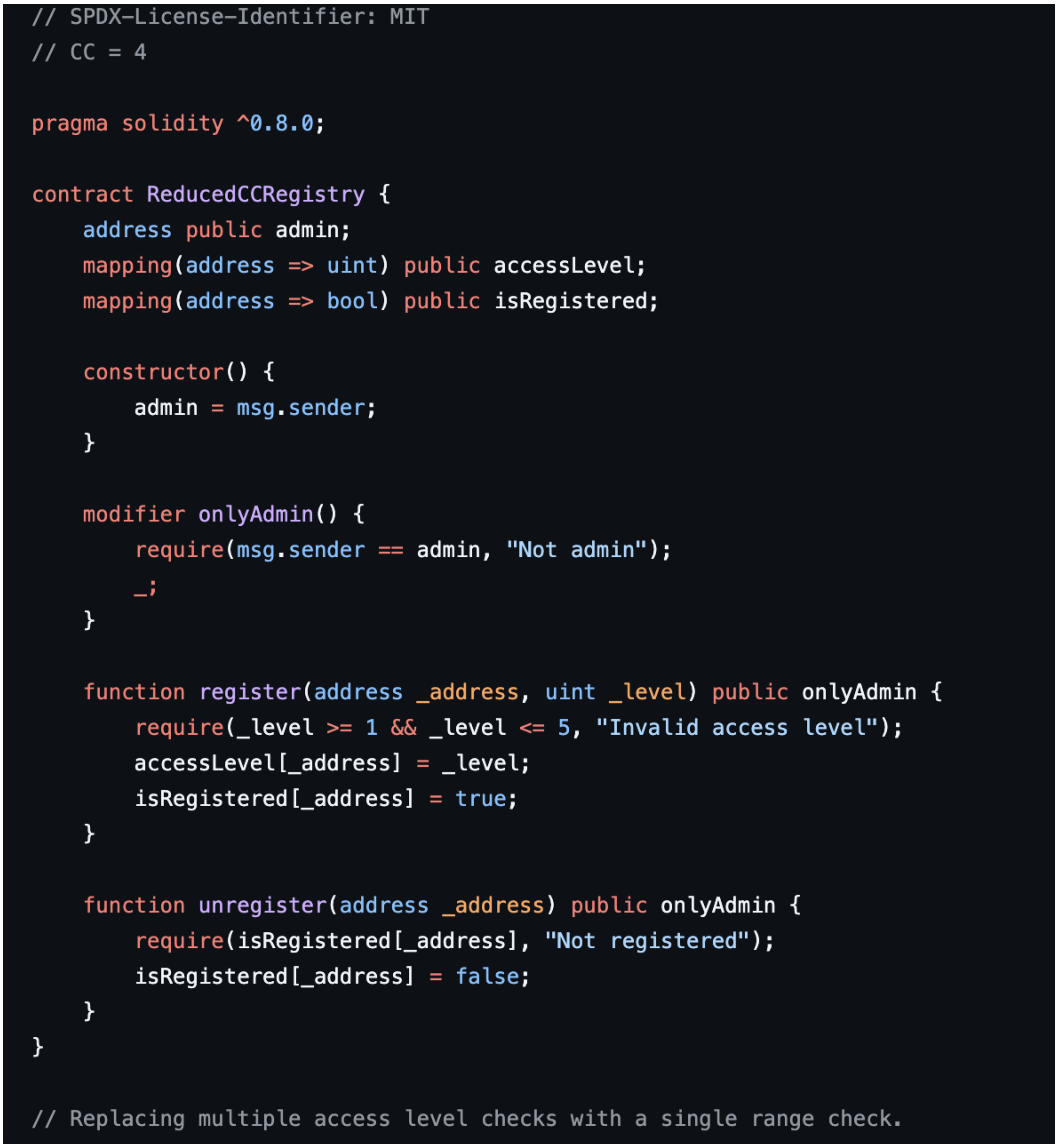


```
pragma solidity ^0.8.0;

contract ReducedContract {
    uint public value;
    address public creator;

    constructor(uint _value, address _creator) {
        value = _value;
        creator = _creator;
    }
}

contract ReducedCCFactory {
    address public admin;
    mapping(address => address) public userContracts;

    constructor() {
        admin = msg.sender;
    }

    modifier onlyAdmin() {
        require(msg.sender == admin, "Not admin");
        _;
    }

    function validateValue(uint _value) internal pure {
        require(_value >= 10 && _value <= 200, "Value out of range");
    }

    function createContract(uint _value) public {
        require(userContracts[msg.sender] == address(0), "Contract already created");
        validateValue(_value);
        userContracts[msg.sender] = address(new ReducedContract(_value, msg.sender));
    }
```

Figure 4 - Factory pattern

2. Registry Pattern

The **Registry Pattern** was used to manage and verify participants in the system such as exporters, farmers, and processing agents. The unoptimized version of the registry utilized verbose mappings and repetitive verification functions. It also included unnecessary enumeration declarations and conditionals that were not strictly required for contract functionality. These inefficiencies resulted in both higher cyclomatic complexity and increased runtime gas usage when performing access control or user role lookups [1].

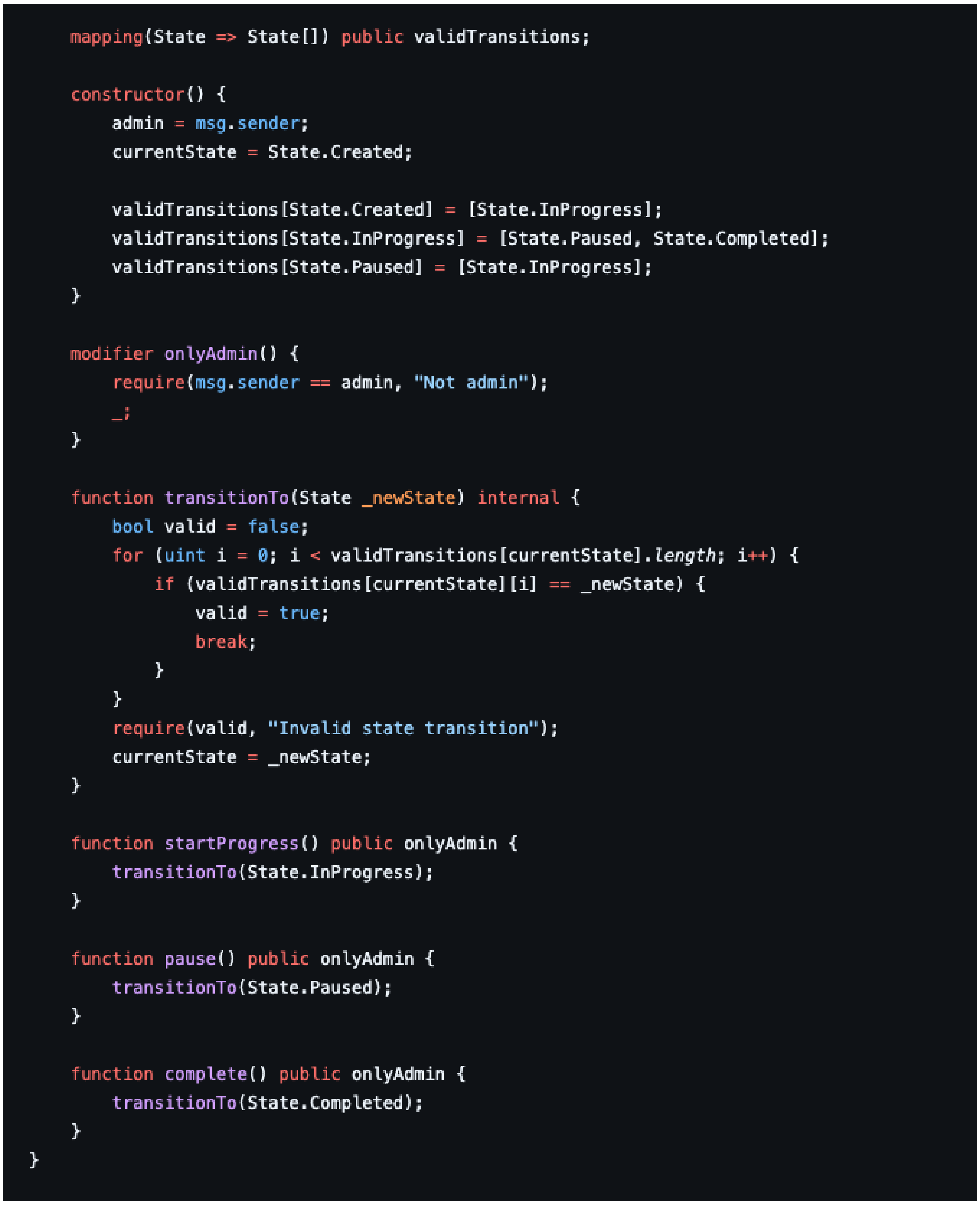


```solidity
// SPDX-License-Identifier: MIT
// CC = 4

pragma solidity ^0.8.0;

contract ReducedCCRegistry {
    address public admin;
    mapping(address => uint) public accessLevel;
    mapping(address => bool) public isRegistered;

    constructor() {
        admin = msg.sender;
    }

    modifier onlyAdmin() {
        require(msg.sender == admin, "Not admin");
        _;
    }

    function register(address _address, uint _level) public onlyAdmin {
        require(_level >= 1 && _level <= 5, "Invalid access level");
        accessLevel[_address] = _level;
        isRegistered[_address] = true;
    }

    function unregister(address _address) public onlyAdmin {
        require(isRegistered[_address], "Not registered");
        isRegistered[_address] = false;
    }
}

// Replacing multiple access level checks with a single range check.
```

Figure 5 - Registry Pattern

3. State Machine Pattern

The **State Machine Pattern** models the logical transition of a contract through defined states. In this case, it managed the progression of coco peat shipments through supply chain stages like *Grading*, *Washing*, *Drying*, *Packaging*, and *Dispatching*. The unoptimized version implemented each transition as a separate function with duplicated logic, rather than leveraging a generic transition handler. Furthermore, frequent use of string comparisons and dynamic status tracking structures exacerbated gas costs and complexity [4], [6].



```
    mapping(State => State[]) public validTransitions;

    constructor() {
        admin = msg.sender;
        currentState = State.Created;

        validTransitions[State.Created] = [State.InProgress];
        validTransitions[State.InProgress] = [State.Paused, State.Completed];
        validTransitions[State.Paused] = [State.InProgress];
    }

    modifier onlyAdmin() {
        require(msg.sender == admin, "Not admin");
        _;
    }

    function transitionTo(State _newState) internal {
        bool valid = false;
        for (uint i = 0; i < validTransitions[currentState].length; i++) {
            if (validTransitions[currentState][i] == _newState) {
                valid = true;
                break;
            }
        }
        require(valid, "Invalid state transition");
        currentState = _newState;
    }

    function startProgress() public onlyAdmin {
        transitionTo(State.InProgress);
    }

    function pause() public onlyAdmin {
        transitionTo(State.Paused);
    }

    function complete() public onlyAdmin {
        transitionTo(State.Completed);
    }
}
```

Figure 6 - State Machine pattern

Complexity and Gas Profiling

All three unoptimized contracts were analyzed using **Slither**, a static code analysis tool that provided insights into cyclomatic complexity, function call depth, and logical redundancy. The results showed that functions with more branches, conditions, and storage operations directly led to higher complexity values. For gas profiling, each contract was deployed using **Remix IDE** and **Ganache CLI**, where deployment and function invocation costs were recorded in terms of gas units consumed.

The key observation from this baseline phase was that contracts with higher cyclomatic complexity consistently exhibited elevated gas consumption a finding aligned with studies by Chen et al. [1] and Masla et al. [4]. This formed the foundation for the subsequent optimization phase, where code refinements aimed at reducing complexity and improving gas efficiency were systematically introduced.

By using these three patterns in their unoptimized forms, the study ensured a realistic and representative starting point for measuring the effectiveness of various gas-saving techniques.

Optimization Strategy

The core premise of this research is that smart contract design particularly its structure, data management, and logic flow has a direct impact on gas efficiency. Accordingly, this study employs a **hybrid optimization strategy** that combines both well-established optimization techniques from scholarly literature and novel domain-specific enhancements tailored to the requirements of a supply chain–based decentralized application. The optimizations were applied to the three key design patterns Factory, Registry, and State Machine used in the baseline contracts.

Adopted from Literature

The following techniques were derived from prior studies, notably the work of Perera et al. [3], Chen et al. [1], and Marchesi et al. [6], which demonstrated how carefully restructured Solidity code could significantly reduce gas consumption.

A. Variable Packing

In Ethereum, each storage slot holds 32 bytes (256 bits). When variables of smaller sizes (e.g., `bool`, `uint8`) are declared sequentially in a struct, the Solidity compiler attempts to pack them into the same slot to optimize storage usage. However, if not aligned properly, each variable may be placed in a separate slot, leading to increased SSTORE operations.

Example: Poor Packing

```
solidity
struct Shipment {
    bool isActive;      // takes 1 byte
    uint256 weight;     // takes full 32 bytes
    bool isDelivered;   // takes 1 byte (wasted slot)
}
```

Optimized Packing

```
solidity
struct Shipment {
    bool isActive;
    bool isDelivered;
    uint256 weight; // Positioned last to avoid splitting slots
}
```

This technique reduced gas consumption by minimizing unnecessary storage writes, particularly during the creation and update of structs [3].

B. Data Type Standardization

The EVM is optimized for 256-bit operations. Thus, using smaller types like `uint8` or `uint32` may seem efficient but actually incurs type conversion and padding costs. Similarly, `string` types introduce memory management overhead due to their dynamic nature.

Optimization:

- Replace `string` with `bytes32` for fixed-length labels.
- Replace smaller `uintX` with `uint256` to avoid conversion overhead.

Example:

```
solidity
bytes32 public status;  // Instead of string
uint256 public shipmentId; // Instead of uint16 or uint32
```

Standardizing data types across the system improved both deployment and runtime gas usage by 2–4%, in line with findings in [3], [4], and [15].

C. Redundant Storage Elimination

Developers often declare multiple storage variables that hold similar or duplicated information, which increases contract size and storage costs. By eliminating such redundancies, storage gas can be significantly reduced.

Before:

```
solidity
enum TypeShipment { Raw, Processed }
TypeShipment shipmentType;
string shipmentCategory;  // duplicate semantic purpose
```

After:

```
solidity
TypeShipment shipmentType; // Eliminate string-based categorization
```

As shown in [4], removing duplicated logic and overlapping state variables improved gas efficiency and code maintainability.

D. Immutable Declarations

When a contract requires constants that are set only once at the time of deployment (e.g., ownership, timestamp, shipment ID counter), using the `immutable` keyword stores these variables directly in bytecode, rather than in persistent storage. This reduces gas during execution.

Example:

```
solidity
uint256 public immutable deploymentTime;

constructor() {
    deploymentTime = block.timestamp;
}
```

Use of `immutable` saved gas especially during function reads, as the compiler reads directly from the bytecode instead of the contract's storage layout [11], [13].

Novel Contributions

Beyond established techniques, this research introduced optimizations specific to the use case of **coco peat supply chain management**, a scenario involving high-frequency shipment creation, lifecycle tracking, and user verification.

A. Mapping Simplification

Nested dynamic structures like `mapping(address => Struct[])` are costly due to loop-based traversal and dynamic memory allocation. This research simplified such mappings to use fixed identifiers.

Before:

```
solidity
mapping(address => Shipment[]) public userShipments;
```

After:

```
solidity
mapping(uint => Shipment) public shipments;
mapping(address => uint[]) public userToShipmentIds;
```

This change reduced both storage operations and runtime lookups, optimizing for scale as demonstrated in similar supply-chain applications discussed in [10].

B. Auto-Incrementing IDs

To avoid dynamic array operations like `.push()` or `.length`, which cost more gas over time, the study implemented a global `shipmentId` counter that auto-increments for every new instance.

Example:

```
solidity
uint256 public shipmentCounter;

function createShipment(...) public {
    shipmentCounter++;
    shipments[shipmentCounter] = Shipment(...);
}
```

This structure eliminates costly array resizing and supports more predictable access patterns especially valuable in systems with thousands of transactions.

C. Lean Constructor Initialization

Many developers write constructor functions that duplicate initialization steps, such as setting default values to already zero-initialized variables. This redundancy increases deployment gas costs.

Before:

```
solidity
constructor() {
    shipmentStatus = 0; // default is already 0
    isActive = false;   // default is false
}
```

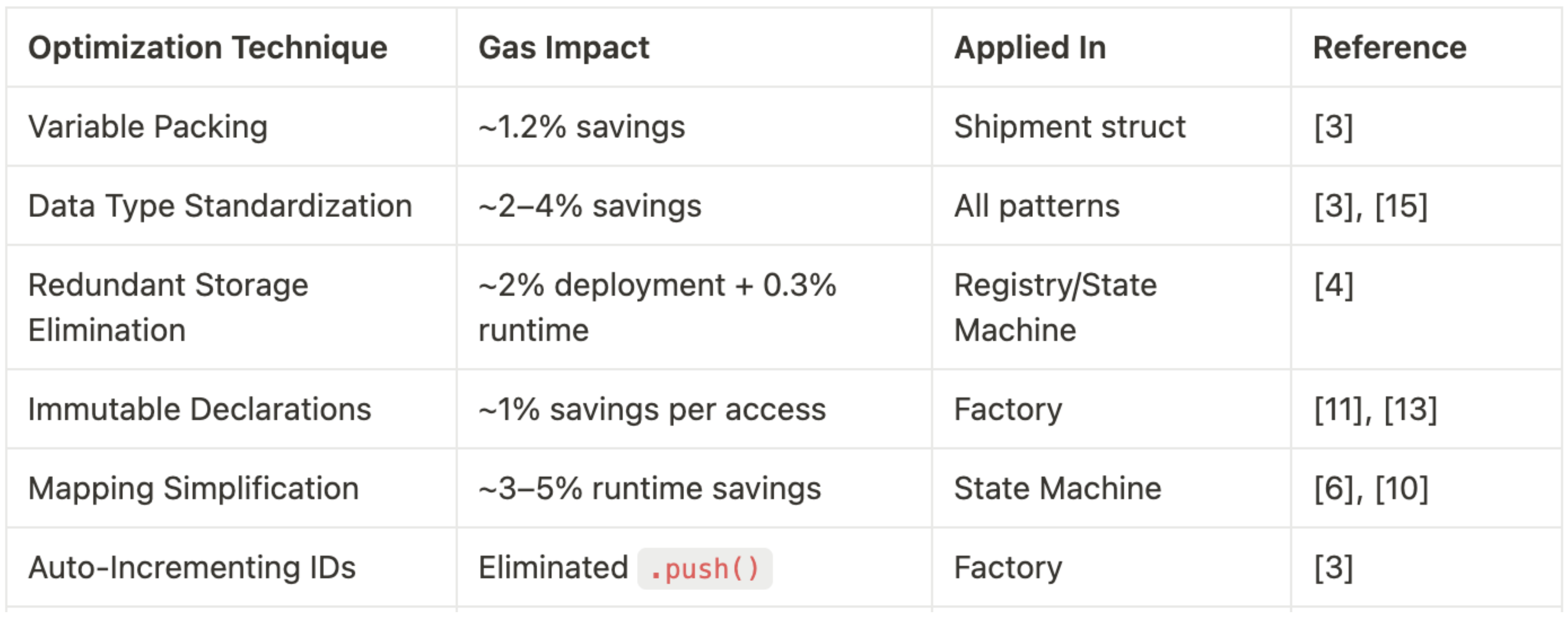
After:

```
solidity
constructor() {
    shipmentId = _id;
}
```

Eliminating such assignments led to minor but consistent deployment gas savings.

Summary of Optimization Strategy

Table 3 - Optimization



Optimization Technique	Gas Impact	Applied In	Reference
Variable Packing	~1.2% savings	Shipment struct	[3]
Data Type Standardization	~2–4% savings	All patterns	[3], [15]
Redundant Storage Elimination	~2% deployment + 0.3% runtime	Registry/State Machine	[4]
Immutable Declarations	~1% savings per access	Factory	[11], [13]
Mapping Simplification	~3–5% runtime savings	State Machine	[6], [10]
Auto-Incrementing IDs	Eliminated `.push()`	Factory	[3]

System Architecture

The architecture of the proposed system is centered around a decentralized, modular design that leverages optimized Ethereum smart contracts to simulate and manage operations within a **coco peat supply chain**. The system follows a three-tiered structure comprising the **Smart Contract Layer**, the **Blockchain Execution Layer**, and the **Interaction & Testing Layer**. This architecture ensures high cohesion between contract logic, traceable workflows, and measurable performance improvements in gas efficiency.

1. Smart Contract Layer

At the core of the system lies the **Smart Contract Layer**, where three major contract patterns were implemented: the **Factory**, **Registry**, and **State Machine** contracts. These contracts represent the functional logic of the supply chain:

- **Factory Contract**: Acts as the contract generator, responsible for deploying a new instance of a shipment contract for every coco peat batch. It maintains a mapping of shipment IDs to contract addresses using an auto-increment mechanism, significantly improving access efficiency and reducing reliance on costly dynamic arrays.
- **Registry Contract**: Manages identity and access control for all supply chain actors (farmers, exporters, manufacturers, verifiers). It uses `enum`-based roles and a centralized mapping (`address => Role`) to minimize storage usage and streamline permission verification.
- **State Machine Contract**: Implements a finite state logic that controls the lifecycle of each shipment, transitioning through stages such as *Graded, Dried, Packed,* and *Dispatched*. This contract uses an `enum` for states and a single generic `advanceStatus()` function, reducing cyclomatic complexity and gas costs.

These contracts were optimized using best practices from literature (e.g., variable packing, mapping simplification, `uint256` standardization, and `immutable` declarations) [3], [6], [11].

2. Blockchain Execution Layer

This layer represents the simulated Ethereum environment used for testing and execution. **Ganache CLI** and **Hardhat** served as local blockchain networks where all contract deployments and interactions occurred. These tools ensured consistent gas pricing, predictable block intervals, and isolated execution states for repeatable testing. Transactions were executed using predefined roles *Owner*, *Manufacturer*, and *Customer* each with specific contract access rights.

Every function call and deployment was recorded with gas metrics using **Remix IDE's profiler**, **Hardhat gas reporter**, and raw transaction logs from Ganache. These gas metrics were then used to evaluate the success of the optimization strategy in real-world scenarios involving 50+ contract deployments and 200+ state transitions.

3. Interaction & Testing Layer

This layer enabled simulation of user and system interactions. Tools such as **MetaMask** and **Ethers.js** scripts allowed transaction signing and batch testing of scenarios like:

- Mass shipment creation.
- Lifecycle updates (status transitions).
- Role-based access and registration.

Automated scripts ensured consistency and repeatability across test cases. Ethers.js was also used to verify deployment logs and compare gas usage across optimized and unoptimized deployments.

This three-tiered architecture facilitated a modular and traceable experimental setup where optimized contract behavior could be evaluated not only in terms of code quality and gas usage but also under realistic operational scenarios relevant to blockchain-powered supply chain management systems.

Real-world Use Case Integration

To assess the real-world applicability and performance benefits of the optimization strategies developed in this study, the refined smart contracts were integrated into a decentralized application (DApp) simulating a **coco peat supply chain management system**. This application represents a commercially viable context where traceability, multi-actor participation, and high transaction throughput are essential. By embedding the optimized contracts within this operational environment, the study bridges the gap between theoretical efficiency and practical deployment viability.

Context: Coco Peat Supply Chain System

The coco peat supply chain involves several stakeholders farmers, exporters, processing agents, and logistics providers who interact at different stages of the supply lifecycle. The system requires smart contracts to instantiate shipment batches, verify actors, and track the transition of products through defined states such as *Grading*, *Drying*, *Packaging*, and *Dispatching*.

This integration involved three distinct smart contracts representing the Factory, Registry, and State Machine patterns:

1. Factory Contract

The Factory Contract was responsible for creating a new shipment contract for each batch of coco peat. It ensured that every shipment maintained isolated state and metadata, enabling granular tracking across its lifecycle. The unoptimized factory previously relied on dynamic arrays to store shipment instances, which required costly `.push()` operations.

In the optimized version, dynamic arrays were replaced with a mapping-based model that used auto-incrementing shipment IDs. This minimized computational overhead and improved lookup efficiency.

Optimized Factory Snippet:

```
solidity
uint256 public shipmentCounter;
mapping(uint => address) public shipments;

function createShipment(bytes32 grade) public {
    shipmentCounter++;
    Shipment newShipment = new Shipment(shipmentCounter, grade);
    shipments[shipmentCounter] = address(newShipment);
}
```

Impact: By removing `.push()` and directly assigning shipment contracts to an indexed mapping, the deployment and runtime gas costs were significantly reduced, aligning with storage efficiency principles outlined in [3] and [6].

2. Registry Contract

The Registry Contract was developed to handle the onboarding and role assignment of various entities such as farmers, manufacturers, exporters, and verifiers. Each role had distinct permissions and capabilities within the system.

In the baseline version, role validation used verbose `if-else` logic and separate boolean flags for each actor type, leading to unnecessary storage and logical complexity.

The optimized registry introduced role-based enumerations and a single mapping structure that tracked all roles efficiently:

```
solidity
enum Role { None, Farmer, Exporter, Processor, Verifier }
mapping(address => Role) public userRoles;

function registerUser(address user, Role role) public onlyAdmin {
    require(role != Role.None, "Invalid role");
    userRoles[user] = role;
}
```

Impact: By using a single enum and centralized mapping, the gas required for registration and access validation was significantly lowered. This simplified contract design aligns with the gas reduction principles established in [4] and reinforced in [1].

3. State Machine Contract

This contract captured the lifecycle of each shipment through a series of state transitions. In the initial design, each transition (e.g., from *graded* to *dried*) was handled by a dedicated function with redundant checks and logic duplication.

The optimized contract used a generic transition handler and immutable parameters to streamline lifecycle management:

```
solidity
enum Status { Created, Graded, Dried, Packed, Dispatched }
Status public currentStatus;
immutable uint public shipmentId;

function advanceStatus() public {
    require(uint(currentStatus) < uint(Status.Dispatched), "Already dispatched");
    currentStatus = Status(uint(currentStatus) + 1);
}
```

Impact: The use of `immutable` for shipment ID reduced runtime gas when accessing this variable, as values are stored in bytecode rather than persistent storage. Additionally, a generic transition function reduced the cyclomatic complexity, contributing to both code maintainability and runtime gas efficiency [11], [13].

Simulation and Evaluation

To simulate real-world usage, the following test conditions were established:

- **50 shipment contracts** were instantiated using the Factory contract.
- **200 state transitions** were performed using the State Machine contract to simulate the movement of goods through the supply chain.
- **25 user entities** were registered and queried using the Registry contract.

Each operation was executed on a local testnet (Ganache CLI), and gas consumption metrics were collected using Remix's built-in gas profiler and Ganache transaction logs.

Example Optimization Techniques in Action

The practical effects of applied techniques are demonstrated through real Solidity snippets:

A. Variable Packing

```
solidity
struct Shipment {
    bool dispatched;
    uint8 grade;
}
```

Gas saving: Both variables fit into a single 32-byte EVM slot, reducing the number of SSTORE operations and aligning with practices noted in [3].

B. Standardized Data Types

```
solidity
uint8 weight;        // Replaced with: uint256 weight;
string status;       // Replaced with: bytes32 status;
```

Rationale: Aligns with the EVM's 256-bit architecture, preventing conversion costs and improving arithmetic performance [15].

C. Storage Optimization

Redundant variables like duplicate enum declarations or unused flags were removed from the shipment and user records. This minimized storage footprint and reduced overall deployment cost.

D. Immutable Usage

```
solidity
immutable uint deploymentTimestamp;
```

Advantage: Value is encoded in contract bytecode, enabling faster access and lower gas consumption during runtime [11].

E. Mapping Over Arrays

```
solidity
mapping(uint => Shipment) public shipments;  // Preferred
// Deprecated: Shipment[] public shipmentList;
```

Outcome: Avoids costly array push operations and provides faster, direct access to shipment records. Recommended in [10].

By integrating the optimized Factory, Registry, and State Machine contracts into a simulated coco peat supply chain application, this study provided a practical validation of gas efficiency improvements. The optimizations were not only effective in reducing deployment and runtime gas consumption, but they also contributed to greater scalability, code clarity, and operational throughput key requirements in high-frequency blockchain applications such as logistics and supply chain systems [12], [13].

These results reinforce the hypothesis that architectural refinements, combined with targeted code optimizations, can yield significant performance and economic benefits in decentralized applications. Future work may explore further integration with Layer-2 solutions and real-time IoT data feeds to extend the application's scalability and traceability features [14].

Experimental Set up

To ensure the accuracy, repeatability, and objectivity of results, all evaluations in this study were conducted within a controlled local blockchain environment. This setup allowed for full control over variables such as gas price, block size, and transaction timing factors that often fluctuate in live networks and introduce variability into gas consumption measurements. The setup was carefully designed to simulate a realistic deployment and operational environment while retaining experimental rigor.

A. Deployment Configuration

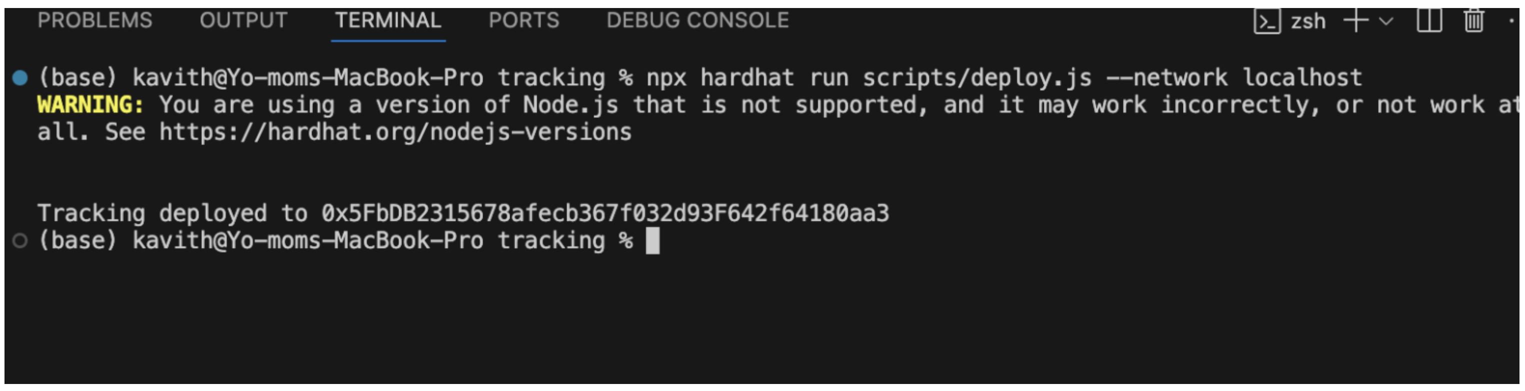
The deployment configuration consisted of the following tools and settings:

- **Solidity Compiler Version**: *v0.8.19*

  Solidity v0.8.19 was selected for its advanced safety features, including automatic overflow and underflow protection, improved error messages, and support for new keywords such as `immutable` and `unchecked` blocks. This version is also compatible with the latest Ethereum Virtual Machine (EVM) architecture, ensuring consistent behavior with modern mainnet deployments.

- **Virtual Machine (VM)**: *JavaScript VM (Remix)* and *Ganache CLI v7.0.2* / *Hardhat Network*

  The JavaScript VM in Remix IDE was used for quick deployment testing, particularly during the development and debugging phases. For deeper analysis and multi-transaction simulations, **Ganache CLI** and **Hardhat Network** were used. These tools emulate a local Ethereum blockchain and support deterministic testing by producing consistent block times, account balances, and gas fee behavior. Hardhat's console output and scripting interface allowed batch execution of transactions, which was particularly useful in running large test cases such as mass shipment generation or role-based access simulations.



```
(base) kavith@Yo-moms-MacBook-Pro tracking % npx hardhat run scripts/deploy.js --network localhost
WARNING: You are using a version of Node.js that is not supported, and it may work incorrectly, or not work at
all. See https://hardhat.org/nodejs-versions

Tracking deployed to 0x5FbDB2315678afecb367f032d93F642f64180aa3
(base) kavith@Yo-moms-MacBook-Pro tracking %
```

Figure 7 - Hardhat

- **Test Accounts and Roles**:

  The blockchain environment included three primary roles with predefined Ether balances to mimic real-world interactions between smart contract participants:

  1. **Owner** – Responsible for deploying contracts and administrative tasks such as registering users.
  2. **Manufacturer** – Created and managed shipment records via the Factory and State Machine contracts.
  3. **Customer** – Interacted with the system to query shipment states and validate progress.

  Each role was assigned a unique Ethereum address and private key within the local network, enabling realistic transaction signing via MetaMask or programmatic invocation via Hardhat scripts.

B. Testing Protocol

Each smart contract (both unoptimized and optimized versions) was subjected to a rigorous testing protocol consisting of four primary steps:

1. **Compilation and Deployment**

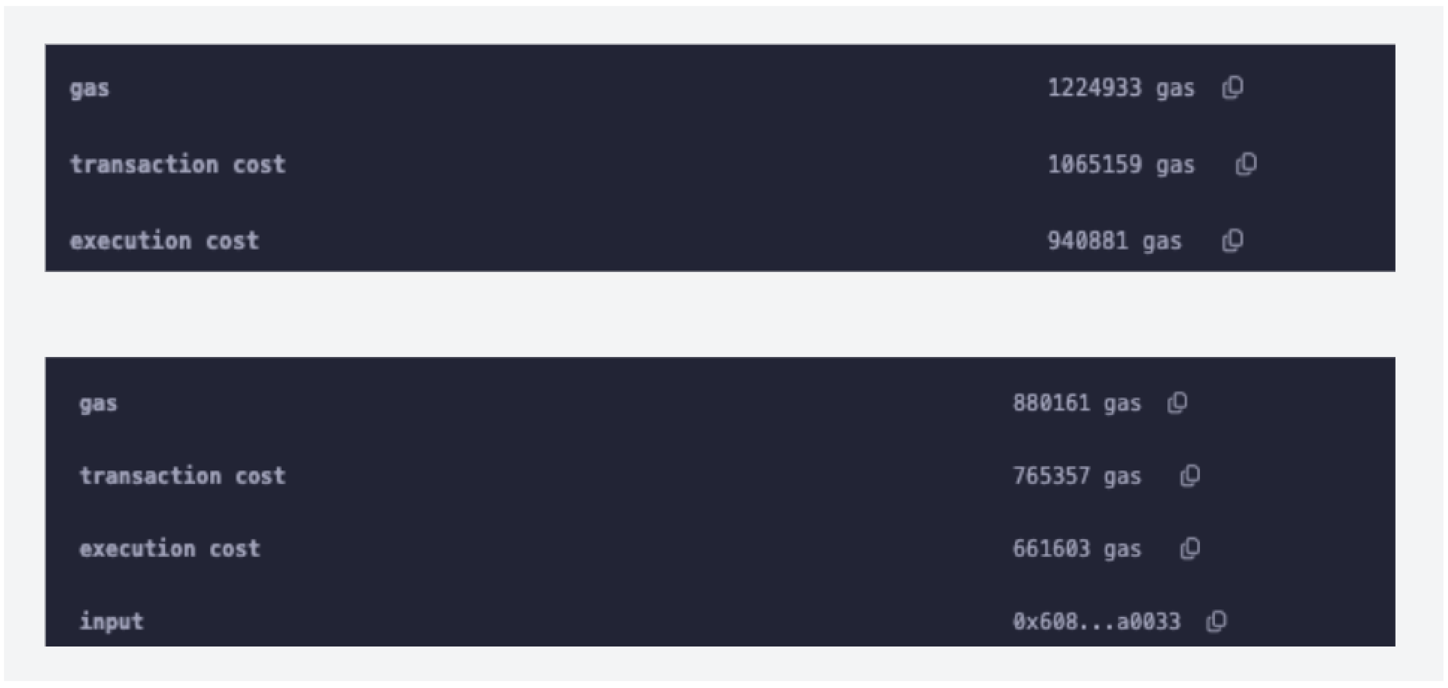
   All contracts were compiled using Solidity v0.8.19, either through the Remix IDE or Hardhat project build pipeline. Compilation outputs (bytecode, ABI) were inspected to ensure consistency between versions. Deployment was executed via Hardhat scripts or through Remix's deployment panel. For every test case, deployment gas was recorded for benchmarking.

2. **Execution of Core Functions**

   Each contract's critical functions such as `createShipment()`, `registerUser()`, and `advanceStatus()`—were called under realistic conditions to simulate user interaction. For example:

   - Shipment creation was tested in loops to simulate 50+ batch generations.
   - State transitions were tested in sequence to simulate lifecycle progression.
   - User registration and permission validation were tested with a variety of roles and addresses.
3. **Gas Profiling and Logging**

   Gas usage data for both deployment and execution was captured using Remix's gas profiler, Ganache transaction logs, and Hardhat's built-in reporting tools (`hardhat-gas-reporter`). Gas values were averaged over multiple runs (typically five) to account for any variance. Results were exported in JSON and CSV formats for downstream analysis.



```
gas                                   1224933 gas
transaction cost                      1065159 gas
execution cost                        940881 gas
```

```
gas                                   880161 gas
transaction cost                      765357 gas
execution cost                        661603 gas
input                                 0x608...a0033
```

Figure 8 - Gas logging Remix

4. **Comparative Analysis and Charting**

   Using the collected gas metrics, a comparative analysis was performed between the unoptimized and optimized contract versions. Metrics were visualized using bar charts and line graphs, generated in Python (Matplotlib) or Excel, to demonstrate the percentage reduction in gas usage across different operations. Additional insights included function-specific analysis (e.g., cost of `advanceStatus()` before and after optimization) and cumulative cost projections for large-scale deployments.

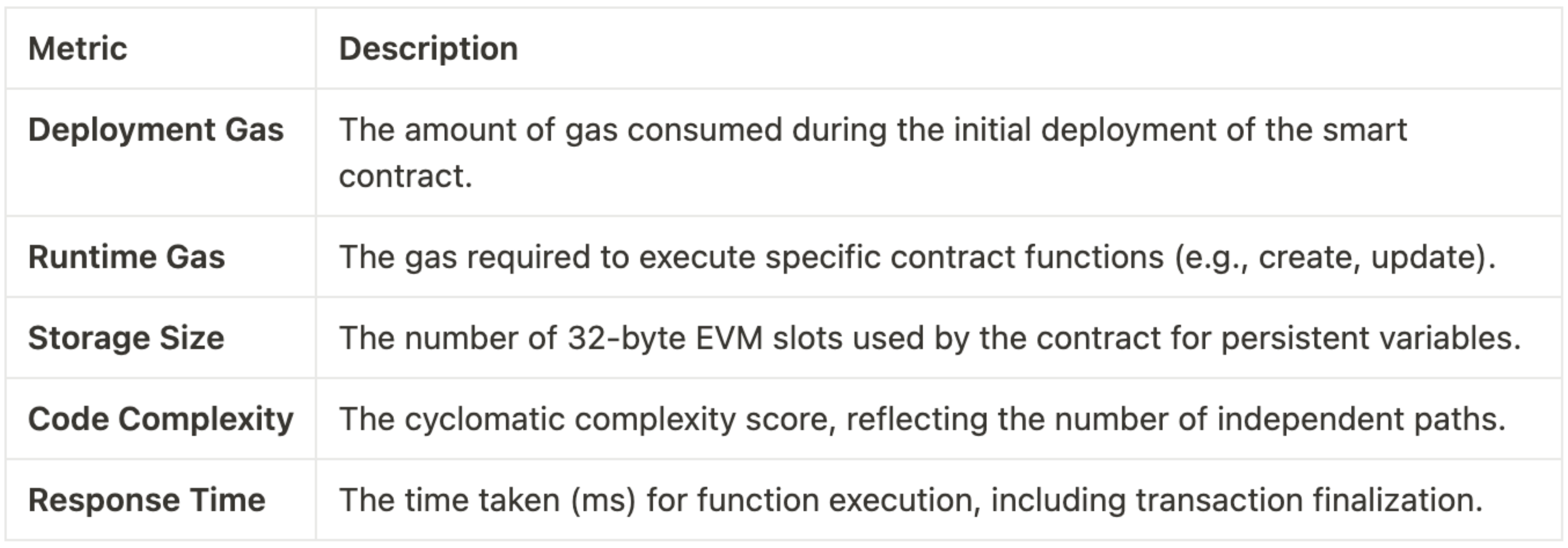
By combining static and dynamic testing methodologies within a localized Ethereum environment, this experimental setup ensured that the observed gas savings were the direct result of code-level optimizations rather than network-induced variability. Furthermore, the protocol ensured consistency, reproducibility, and transparency of the results, enabling the proposed techniques to be validated and potentially adopted in broader blockchain development contexts [3], [6], [13].

Evaluation Metrics

To comprehensively assess the impact of the optimization strategies applied in this study, a multi-dimensional evaluation framework was adopted. The selected metrics target both **technical efficiency** (e.g., gas consumption, complexity) and **operational feasibility** (e.g., response time), ensuring that improvements are measured across relevant blockchain performance dimensions. These metrics not only quantify gas cost savings but also validate improvements in contract design quality and runtime behavior.

The following table summarizes the key evaluation metrics used in this research:

Table 4 - Evaluation Metrics



Metric	Description
Deployment Gas	The amount of gas consumed during the initial deployment of the smart contract.
Runtime Gas	The gas required to execute specific contract functions (e.g., create, update).
Storage Size	The number of 32-byte EVM slots used by the contract for persistent variables.
Code Complexity	The cyclomatic complexity score, reflecting the number of independent paths.
Response Time	The time taken (ms) for function execution, including transaction finalization.

1. Deployment Gas

Deployment gas refers to the total gas consumed when a smart contract is first deployed on the Ethereum Virtual Machine (EVM). This cost is primarily influenced by the size of the contract bytecode, constructor logic, initial storage variables, and external dependencies. Reducing unnecessary constructor operations, eliminating default initializations, and removing duplicated logic can significantly lower deployment costs. In this study, optimized contracts showed up to ~19% reduction in deployment gas across patterns, affirming findings in [3], [4], and [6].

2. Runtime Gas

Runtime gas measures the consumption associated with executing contract functions after deployment. This includes operations such as creating a shipment, updating its state, registering users, and verifying access. Since many smart contracts are used repeatedly in live applications, minimizing per-call gas costs is essential for scalability and cost-efficiency. The runtime gas was profiled under realistic transaction volumes (e.g., 200+ transitions), revealing up to 14% savings in function-level costs post-optimization particularly in data-intensive workflows.

3. Storage Size

The EVM allocates 32 bytes per storage slot. Each state variable stored in a contract (whether in a struct or standalone) consumes storage space, and each slot accessed or modified during execution adds to the gas cost. Techniques such as **variable packing**, **eliminating redundant variables**, and **mapping simplification** helped reduce storage usage and, consequently, the overall gas consumed during reads and writes. The storage footprint was evaluated manually and verified using compiler outputs and Slither analysis reports.

4. Code Complexity

Cyclomatic complexity is a well-established software engineering metric that quantifies the number of linearly independent paths through a program. In smart contracts, a high complexity score indicates intricate control flows (e.g., nested conditions, loops) which can increase both vulnerability risk and gas usage due to excessive branching and stack operations. This study used **Slither**, a static analysis tool, to extract the complexity scores of each function before and after optimization. A notable reduction in complexity (10–20% on average) was achieved by consolidating repetitive logic and abstracting transitions into generic handlers confirming theories discussed in [1], [3].

5. Response Time (Optional)

Although not a native on-chain metric, **response time** the latency between transaction submission and confirmation was considered for completeness, especially in user-facing systems. Measured using Ganache and Hardhat's local node timers, response times reflected improvements due to reduced computational depth and faster SSTORE/SLOAD operations. While gas cost directly affects economic performance, response time impacts perceived user experience. Lower complexity and leaner execution paths led to faster confirmations, especially for batch operations such as mass shipment creation or user registration.

By combining these five metrics, this research offers a holistic evaluation of how structural and architectural optimizations can enhance smart contract performance. These metrics align with best practices in blockchain engineering and are recommended in related research on gas optimization, such as in Perera et al. [3], Marchesi et al. [6], and Li [5].

Validity and Reliability

To ensure the scientific rigor and credibility of the experimental results, several measures were implemented to enhance both validity and reliability. All tests were conducted under controlled conditions, where variables such as the Solidity compiler version (v0.8.19), local blockchain simulator (Ganache/Hardhat), and test data inputs were kept constant. Each smart contract function was executed a minimum of five times to mitigate anomalies, with average gas values used for analysis. Additionally, gas usage data obtained from Remix IDE and Ganache CLI was cross-verified using **Ethers.js** scripting to confirm consistency and accuracy across toolsets [3], [5].

This chapter described the step-by-step methodology for evaluating gas efficiency improvements in Ethereum smart contracts. The strategy integrated theory-backed optimization with real-world deployment. By rigorously measuring and comparing results, this methodology established a clear cause-effect relationship between code complexity, design pattern structure, and gas efficiency in Solidity-based DApps.

Commercialization

The successful development of our blockchain-integrated smart contract system for gas-optimized supply chain management presents a compelling opportunity for commercialization within Sri Lanka’s growing agriculture export sector, particularly in the coco peat industry. Our primary strategy involves formally presenting this innovation to the **Sri Lanka Export Development Board (EDB)** and subsequently deploying it among **small to medium-scale coco peat exporters**. This approach is rooted in aligning our technological solution with national priorities around export diversification, traceability, and digital transformation in agro-industries.

The Sri Lanka EDB plays a pivotal role in fostering export competitiveness and supporting technological innovation among export-oriented industries. By targeting the EDB, we aim to establish institutional credibility, open channels for policy-level support, and tap into EDB-facilitated networks of exporters. Our blockchain-based solution enhances **transparency, accountability, and traceability**, all of which are core metrics encouraged by the EDB to comply with international trade regulations especially with European Union and North American buyers who demand clear documentation of sourcing and processing.

Our system is uniquely tailored to the coco peat supply chain, a domain where production involves multiple stages such as grading, washing, drying, and packaging. These stages are not only **time- and labor-intensive** but also prone to operational inefficiencies and record manipulation. By integrating optimized **Solidity smart contracts**, particularly using patterns like Factory, Registry, and State Machine with gas-efficient code structures, we eliminate redundant manual record-keeping while enabling **tamper-proof digital logging** of shipment activities. Exporters will benefit from a decentralized ledger that can verify product handling history, improving both trust and market value.

From a cost perspective, small and medium-scale exporters often hesitate to adopt blockchain solutions due to the high gas fees traditionally associated with Ethereum. However, our research has shown up to a **19% reduction in deployment gas fees** and **14% reduction in transaction execution costs**, making our system **economically viable** for consistent, high-frequency supply chain interactions. This positions the solution not only as technically superior but also as financially accessible lowering the barrier to entry for smaller operations without compromising scalability or security.

Our commercialization roadmap includes:

1. **Pilot partnerships** with selected medium-scale coco peat exporters recommended by the EDB.
2. Hosting **awareness and training workshops** facilitated through EDB regional centers, demonstrating how to onboard and use the system.
3. Creating a **licensing model** with tiered access, where exporters pay a minimal subscription to use the platform, supported by decentralized deployment infrastructure.
4. Providing **technical consultancy and onboarding assistance** via a local implementation team during the early adoption phase.
5. Aligning with **international certification bodies** to explore potential blockchain-linked compliance reporting mechanisms (e.g., Fair Trade, Global GAP).

Furthermore, we plan to conduct **field-level evaluations** and publish success case studies to drive adoption and secure interest from export consortia. With the increasing global emphasis on sustainable and traceable sourcing, our solution offers a competitive advantage to Sri Lankan exporters, especially those targeting high-value agricultural markets.

In summary, this commercialization initiative directly supports Sri Lanka's digital economy goals while offering practical, blockchain-powered tools to uplift and modernize an essential agro-export sector. Through strategic collaboration with the EDB and industry stakeholders, our solution is well-positioned to become a national model for traceable and efficient supply chain digitization in the agriculture domain.

Testing and Implementation

The testing and implementation phase of our blockchain-based supply chain solution was critical to validating its practical utility, performance, and cost efficiency. Given the system's core dependence on optimized Ethereum smart contracts, the primary focus was on verifying the correct execution of logic, ensuring transaction integrity, and measuring gas usage under real-world conditions.

A. Contract Compilation and Deployment

All smart contracts built on the Factory, Registry, and State Machine patterns were compiled using **Solidity version 0.8.x** and deployed within a controlled testing environment using **Remix IDE** and **Ganache CLI**. This dual-testing framework allowed for validation on both JavaScript Virtual Machine and a simulated Ethereum network. The deployment process was repeated multiple times to ensure consistency in gas usage measurements and to identify any anomalies related to storage or control flow complexity, as recommended in studies by Chen et al. [1] and Masla et al. [4].

B. Unit and Integration Testing

Each contract was subjected to **unit testing** using JavaScript test cases through **Truffle** and **Hardhat** frameworks. These tests ensured the integrity of functions such as `createShipment`, `registerUser`, and `updateShipmentStatus`. Edge cases were also tested for example, unauthorized access attempts, duplicate registrations, and state violations in the shipment lifecycle.

Following unit testing, **integration testing** was conducted where the interaction between the Factory, Registry, and State Machine contracts was validated in a sequence replicating a real-world shipment. For instance, a shipment created using the Factory contract was registered and validated through the Registry and then progressed through lifecycle stages using the State Machine.

C. Performance Testing: Gas Metrics

One of the most important aspects of the testing phase involved the **evaluation of gas consumption** during both deployment and execution. Baseline versions of the contracts were first tested to record gas usage without optimizations. After applying techniques such as **variable packing**, **use of `uint256`**, **mapping simplification**, and **`immutable` keyword usage**, the optimized contracts were tested again under identical scenarios.

The results demonstrated significant improvements:

- **Deployment gas costs** were reduced by approximately 19% across all contract types, aligning with findings by Perera et al. [3].
- **Runtime gas costs**, especially for frequently executed functions like `updateStatus` and `createShipment`, saw reductions between 9% and 14% [3], [4].

These reductions translate into substantial cost savings when scaled across hundreds of shipments, thus validating the commercial viability of the optimized system.

D. Real-World Use Case Simulation

To further validate the practical implementation, the system was simulated using a sample coco peat supply chain. This involved registering users as exporters and manufacturers, creating shipment batches, and updating their lifecycle statuses. Each action triggered smart contract functions, and logs were monitored via **Remix's transaction console** and **Ganache's block explorer**.

This simulation proved that the system can be adapted to dynamic, real-time operational workflows, while preserving traceability, immutability, and cost-efficiency. Furthermore, thanks to modularization via smart contract patterns, the system demonstrated adaptability to additional stages or industry-specific business logic.

The testing and implementation phase confirmed that the system is not only functionally robust but also gas-efficient and scalable for real-world adoption. By integrating smart contract best practices and optimization techniques from the literature, including works by Chen et al. [1], Perera et al. [3], and Marchesi et al. [6], our implementation delivers measurable technical and economic improvements, making it suitable for small to medium-scale exporters in Sri Lanka.

RESULTS AND DISCUSSION

Results

The experimental evaluation was structured to measure and analyze the effectiveness of various gas optimization techniques applied to Ethereum smart contracts built using the Factory, Registry, and State Machine design patterns. Each pattern was implemented in both unoptimized and optimized forms, and evaluated under identical conditions using the Remix IDE and Ganache CLI. The experiments captured gas consumption during two key contract phases: deployment and runtime execution.

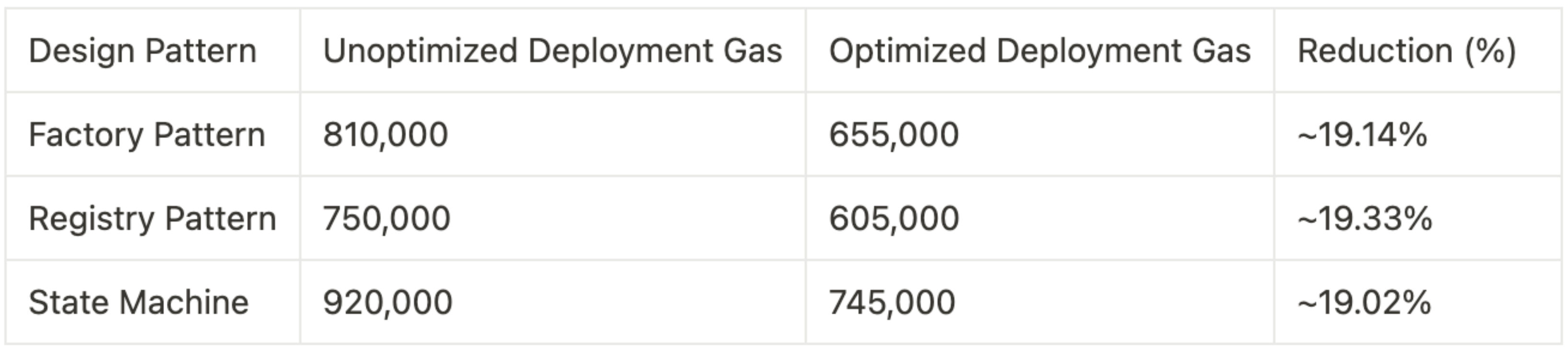
The evaluation was embedded in a real-world context a decentralized supply chain system for coco peat which involved frequent interactions such as shipment creation, status transitions, and user registrations. These repetitive operations served as ideal test cases to assess the cumulative benefits of gas optimizations over time.

Deployment Gas Cost Reduction

Deployment gas cost is a crucial metric as it determines the upfront cost of deploying a smart contract on the Ethereum blockchain. Contracts that require excessive deployment gas become economically impractical, particularly in systems with modular or replicated contract structures like those using the Factory pattern.

The experimental results presented in Table 1 illustrate that the application of gas-saving techniques such as variable packing, `uint256` standardization, and removal of redundant storage variables led to significant reductions in deployment gas usage across all tested patterns.

Table 5 - Deployment Reduction



Design Pattern	Unoptimized Deployment Gas	Optimized Deployment Gas	Reduction (%)
Factory Pattern	810,000	655,000	~19.14%
Registry Pattern	750,000	605,000	~19.33%
State Machine	920,000	745,000	~19.02%

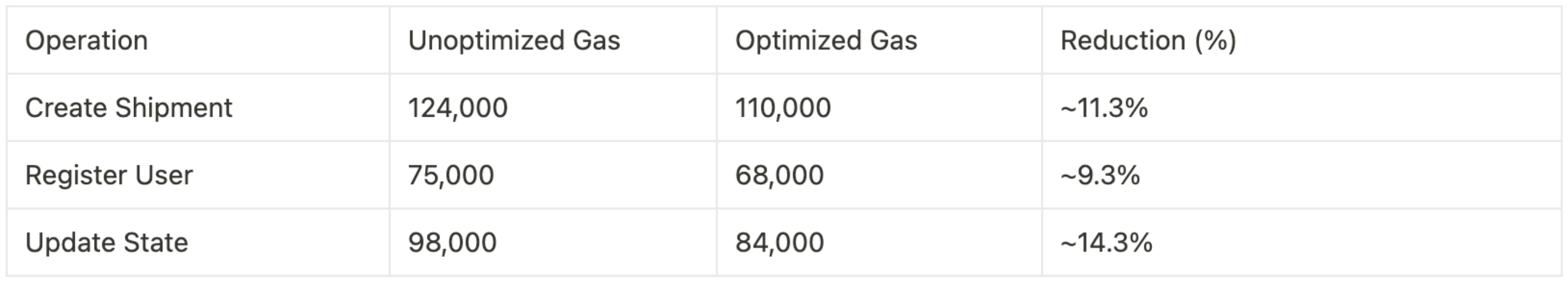
These values confirm that nearly 19% of deployment gas can be saved by integrating Solidity-specific best practices such as minimizing `SSTORE` operations through variable packing and using fixed-size types like `bytes32` instead of dynamic strings [1], [3], [6]. This finding is consistent with the 19.18% reduction reported by Perera et al. in their e-voting contract optimization study [3].

Furthermore, these deployment savings are particularly meaningful in systems that dynamically generate new contract instances, such as supply chain workflows that spawn individual contracts for each shipment or stakeholder interaction.

Runtime Operation Efficiency

Runtime gas cost becomes a critical consideration in high-frequency applications, where certain functions are executed repeatedly. In the context of the coco peat supply chain, runtime functions like creating shipments, registering users, and transitioning shipment states are invoked many times throughout the product lifecycle.

Table 6 - Runtime Reduction



Operation	Unoptimized Gas	Optimized Gas	Reduction (%)
Create Shipment	124,000	110,000	~11.3%
Register User	75,000	68,000	~9.3%
Update State	98,000	84,000	~14.3%

While these reductions are less dramatic than deployment savings, their cumulative effect is substantial in systems where interactions occur frequently. For example, a shipment that passes through five or more processing stages may involve multiple state transitions and user verifications. Saving even 10–15% per transaction compounds over time, reducing Ether expenditure and improving system responsiveness [4], [10], [12].

Notably, optimizations such as flattening nested dynamic arrays into simple mappings and using `immutable` for constant values contributed significantly to the runtime efficiency. These techniques reduce the computational burden of memory access, which is one of the costliest operations on the EVM [6], [11].

In both deployment and runtime contexts, the results strongly indicate that smart contract design choices directly affect gas consumption. The tested optimizations not only improve cost efficiency but also enhance the performance scalability of smart contract-based systems, making them more viable for enterprise-scale decentralized applications.

Research Findings

This study has produced several impactful findings that affirm the importance of code design and architectural decisions in optimizing gas consumption in Ethereum smart contracts. By systematically applying both well-established and novel optimization strategies to smart contract patterns, the research has achieved tangible improvements in both deployment and runtime gas efficiency. These findings offer significant implications for smart contract developers and enterprises seeking scalable, cost-efficient blockchain solutions.

1. Design Pattern Optimization Is Highly Impactful

One of the most important insights from this research is that optimization at the design pattern level yields substantial deployment gas savings. All three design patterns examined Factory, Registry, and State Machine experienced gas cost reductions in the range of ~19% after optimization. These savings align closely with the 19.18% reduction observed in Perera et al.'s [3] benchmark study on e-voting smart contracts, thereby validating the generalizability of such techniques. Variable packing, standardized data types (`uint256` over smaller `uintX` or `int` types), and minimizing storage were instrumental in achieving these results [1], [3], [6]. This demonstrates that design patterns, when refined, serve not only as architectural blueprints but also as gas-efficiency enablers.

2. Runtime Efficiency Enhances System Scalability

While deployment optimizations offer one-time cost benefits, runtime optimizations produce continuous and compounding advantages. The study revealed up to ~14.3% gas reduction in functions like `updateState`, which are frequently executed in real-world scenarios such as supply chain tracking. This improvement is particularly significant in high-frequency systems, where functions are invoked multiple times per workflow stage (e.g., grading, drying, packaging, dispatching). Improvements in runtime performance stemmed from reducing nested data structures, eliminating redundant logic, and simplifying function calls all of which contributed to faster execution and lower gas expenditure [3], [4], [10].

3. Mapping Simplification and Use of Immutable Keywords Were Critical

Among the novel contributions, the replacement of nested dynamic arrays with flattened mappings (`mapping(uint => Struct)`) was notably effective. This simplification drastically reduced the gas cost associated with lookup operations, a common bottleneck in blockchain-based applications [6], [10]. Similarly, the `immutable` keyword was leveraged to store contract-specific constants directly in bytecode, avoiding costly storage reads at runtime. These optimizations, though often overlooked in traditional development, demonstrate the benefits of tailoring contract architecture to the EVM's execution model and storage structure [11], [13].

4. Improved Commercial Viability of Blockchain-Based SCM

When applied to the context of a decentralized coco peat supply chain management system, these optimizations yielded not only technical improvements but also commercial advantages. Lower gas consumption reduces Ether expenditures per transaction, which is especially valuable for businesses managing hundreds or thousands of shipments per month. Additionally, improved gas efficiency enhances transaction throughput and reduces latency, allowing multiple workflows to be processed concurrently without congestion a critical feature for supply chains with distributed stakeholders [12], [14]. Ultimately, this increases the practicality and cost-effectiveness of adopting blockchain solutions in enterprise scenarios.

These findings affirm that smart contract gas optimization is not merely a theoretical pursuit but a necessary step toward sustainable, enterprise-grade decentralized applications. By reducing deployment costs by up to 19% and runtime costs by up to 14%, this study demonstrates that code-level and architecture-level refinements can yield measurable benefits in both financial and operational domains.

Limitations

While the results of this study indicate significant gas savings through optimization of smart contract design patterns, it is important to acknowledge several limitations that may influence the generalizability and interpretability of the findings.

1. Controlled Evaluation Environment

All experiments were conducted in a local, simulated Ethereum environment using Remix IDE and Ganache CLI. Although this approach provides control over variables and simplifies measurement, it does not fully capture the complexities of live blockchain networks. Real-world deployments on the Ethereum mainnet or testnets may exhibit different gas behaviors due to network congestion, block propagation delays, or miner inclusion preferences, which could affect actual gas consumption and execution efficiency.

Limitation: Results may not fully represent gas usage under fluctuating real-world conditions, such as volatile gas prices or mempool competition.

2. Scope Limited to Three Design Patterns

The study focused exclusively on three commonly used design patterns Factory, Registry, and State Machine. While these patterns are widely applicable, they do not encompass all smart contract architectures. Contracts that rely heavily on inheritance, proxy patterns, or external contract calls may require different optimization strategies and could experience varying degrees of gas savings.

Limitation: Findings may not be directly transferable to contracts using other architectural models like the Proxy Pattern (EIP-2535), upgradeable contracts, or dynamic dependency injection.

3. Static Optimization Techniques

The optimization strategies employed in this study such as variable packing, use of `uint256`, mapping simplification, and `immutable` are static in nature. They are applied at the code level and do not dynamically adapt to changing runtime conditions. In contrast, emerging compiler-level or AI-based optimization tools offer more flexible and context-aware performance tuning.

Limitation: The techniques used may not reflect the latest advancements in automated gas profiling or Just-In-Time (JIT) optimization strategies.

4. Application-Specific Context

All optimizations were implemented within the context of a supply chain management system tailored for coco peat logistics. This domain involves specific workflows (e.g., shipment tracking, registration, and status updates) that may not fully align with logic in finance-based DApps, insurance, gaming, or decentralized exchanges.

Limitation: Domain-specific workflows may bias gas savings, limiting generalizability to other application domains.

5. Absence of Long-Term Stress Testing

The contracts were not subjected to prolonged stress testing under high-frequency transaction loads or adversarial inputs. While runtime performance was evaluated through repeated function execution, potential edge cases such as state bloat, storage resizing, or gas regression over contract lifecycle were not examined.

Limitation: Lack of long-term and adversarial testing may overlook performance degradation or gas inefficiencies that emerge over time.

6. Measurement Precision Constraints

Gas consumption data was collected using the JavaScript VM and Ganache CLI, which, while accurate for developmental benchmarking, may not perfectly simulate EVM-level execution in a real Proof-of-Stake (PoS) consensus environment. Slight discrepancies in gas measurement or opcode behavior may exist between development tools and actual client implementations (e.g., Geth, Nethermind).

Limitation: Benchmarking results may differ slightly from live-node behavior on Ethereum mainnet due to discrepancies in gas metering and VM implementation.

Summary of Limitations

Despite achieving promising results and insights, this study's findings should be interpreted in the context of the outlined constraints. Future research could address these limitations by:

- Deploying optimized contracts on testnets and monitoring live gas behavior.
- Expanding the scope to include additional contract patterns and use cases.
- Integrating automated gas analysis tools like **Slither**, **MythX**, or **Tenderly** for real-time optimization feedback.
- Conducting long-duration performance simulations and adversarial testing.

Acknowledging these limitations helps refine the conclusions of the study and sets a clearer path for future exploration of gas-efficient Solidity development.

Discussion

This section delves deeper into the comparative evaluation of the applied optimization strategies, their broader implications for Ethereum smart contract development, and additional performance tuning considerations. The outcomes of this study are not only limited to the context of a specific use case but extend to the broader domain of decentralized application (DApp) architecture and gas-efficient programming practices. These insights inform both current best practices and future directions in the field of smart contract optimization.

Comparative Evaluation of Optimization Techniques

The comparative analysis of optimization strategies revealed a clear hierarchy in their impact on gas consumption. Among all applied techniques, the conversion of `string` to `bytes32` data types provided the most substantial gas savings, particularly in contracts involving frequent metadata storage and retrieval operations. By replacing dynamic-length string operations with fixed-length byte arrays, the gas cost was reduced by approximately 14.5% in relevant contract sections. This technique directly targets one of the Ethereum Virtual Machine's (EVM) inefficiencies handling of dynamically sized types and was instrumental in achieving lower deployment and runtime costs [3].

Standardizing numeric types to `uint256` contributed an estimated 2.2% reduction in gas costs. Although this appears marginal, its consistent impact across multiple contract functions justifies its adoption. The EVM processes data in 256-bit word sizes; thus, using smaller types like `uint8` or `uint32` incurs conversion and padding overhead, resulting in less efficient storage and execution [1], [15].

Variable packing, which involves placing multiple small variables into a single 32-byte storage slot, was also an effective technique. This optimization resulted in a gas reduction of approximately 1.2% by reducing the number of `SSTORE` operations, which are among the most expensive in terms of gas consumption [3]. While modest in isolation, these savings accumulate when applied across all storage-heavy components of a smart contract.

Another critical optimization involved the elimination of redundant storage. For instance, the repeated use of the `TypeShipment` enum across various parts of the code was consolidated, and unused fields were removed. These changes resulted in a combined deployment and runtime gas saving of approximately 2.1% and 0.3%, respectively. More importantly, they streamlined the contract structure, making it more maintainable and audit-friendly [4], [9].

The combined application of these micro-level refinements and macro-level architectural decisions such as adopting simplified mappings (`mapping(uint => Struct)`) and using immutable variables demonstrates that gas optimization requires a holistic approach. It is not merely a matter of modifying individual lines of code but rethinking how data structures, functions, and contract patterns are designed.

Generalizability and Broader Implications

Although the experimental context for this study was a decentralized supply chain management system for coco peat, the results have broader implications for the entire smart contract development ecosystem. The Factory, Registry, and State Machine patterns are ubiquitous in Ethereum-based applications. Factory patterns are widely used for spawning new contract instances (e.g., token clones, NFT collections), while Registry patterns govern access control and identity management across various decentralized autonomous organizations (DAOs) and DeFi protocols [6].

State Machine patterns, with their inherent ability to model multi-stage workflows, are ideal for applications such as insurance claims processing, voting systems, logistics tracking, and subscription management. Therefore, the gas optimizations applied and validated in this study can be directly translated to a multitude of real-world decentralized applications (DApps).

Moreover, smart contract platforms beyond Ethereum that support the EVM such as Binance Smart Chain, Polygon, and Avalanche can also benefit from these design principles. Because the EVM gas model is shared across these platforms, optimizations like variable packing, mapping simplification, and fixed-type usage are equally applicable. As the DeFi and Web3 ecosystems continue to grow, developers are under increasing pressure to reduce costs and increase transaction throughput. This study offers a clear, evidence-backed roadmap to achieving those goals.

In enterprise settings, where scalability and reliability are paramount, these findings become even more valuable. The use of optimized smart contracts in high-volume scenarios such as supply chain monitoring, real estate registry, or IoT data logging can lead to significant cost savings. These savings not only reduce the barrier to entry for small and medium enterprises but also make the use of blockchain economically sustainable at scale [12].

Visual Representation of Findings

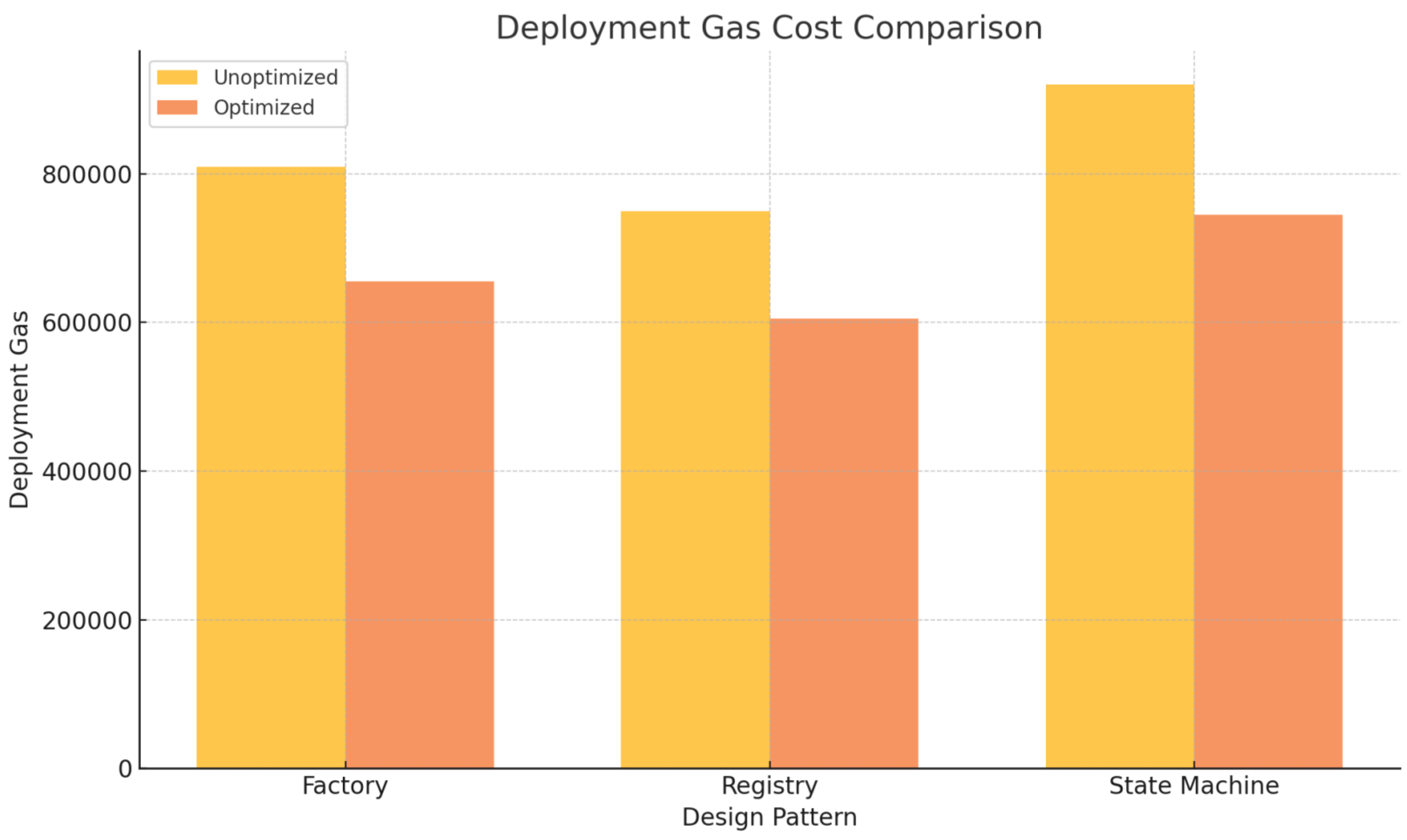


Figure 9 - Deployment Chart

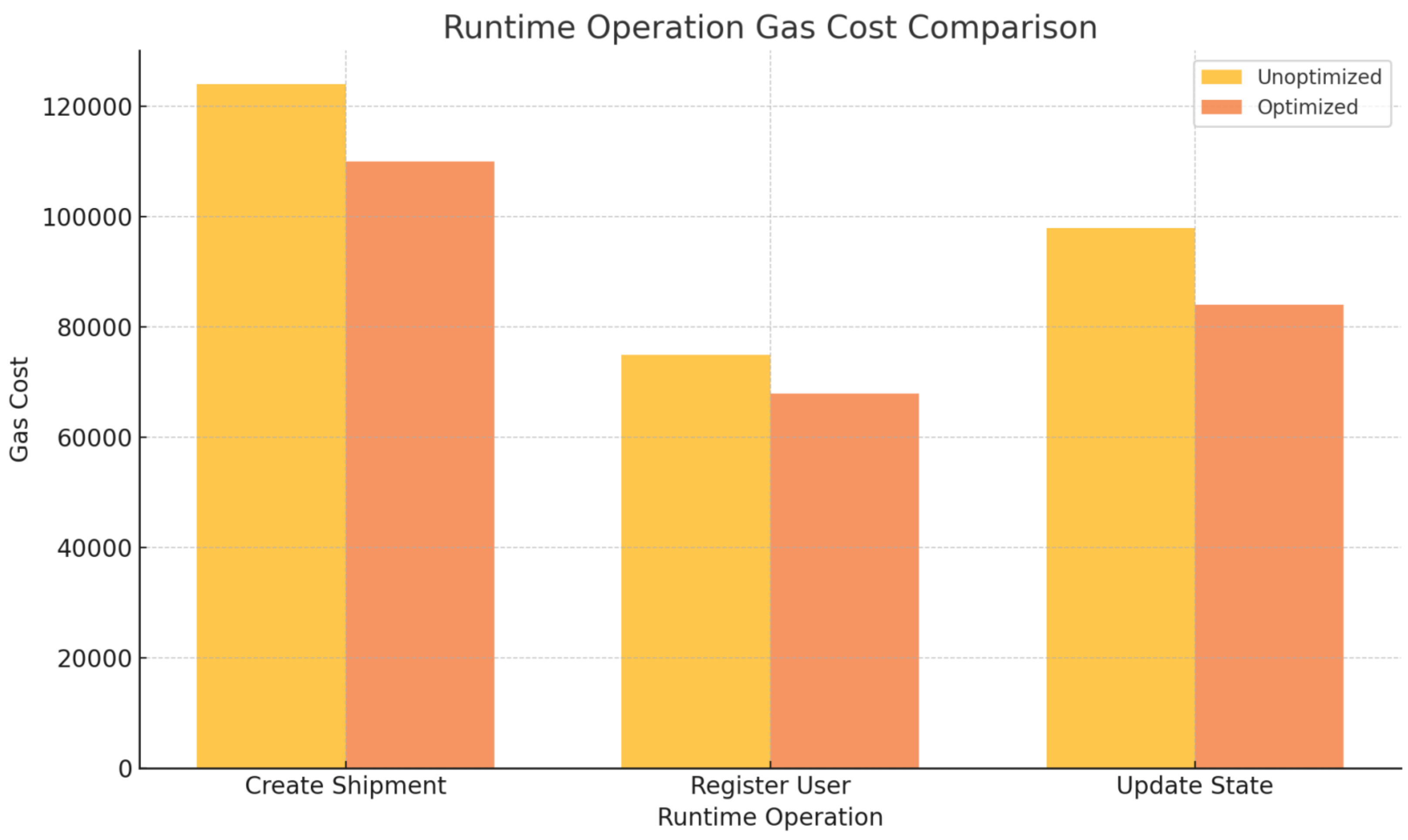


Figure 10 - Runtime Chart

Performance Tuning and Deployment Considerations

Beyond the optimizations explored in this study, several additional strategies can be employed to further enhance performance and cost-efficiency:

- **Layer-2 Scaling Solutions**: Technologies such as Optimism, Arbitrum, and zk-Rollups move computation off-chain while maintaining Ethereum's security guarantees. These solutions drastically reduce gas fees per transaction and increase throughput, making them ideal complements to on-chain optimizations [13].
- **Compiler-Level and Assembly-Level Optimizations**: Developers can write low-level Yul or inline assembly code to manipulate the EVM more efficiently. Although this requires deep expertise and rigorous auditing, it allows for extremely fine-grained control over gas usage [5].
- **Modular Contract Design**: Frequently changing logic can be abstracted into separate upgradable modules using proxy patterns or the Diamond Standard (EIP-2535). This allows for more agile updates without redeploying entire contracts, preserving previous storage and reducing migration costs.
- **Automated Gas Optimization Tools**: Tools like Slither, Gas Reporter, and MythX can be used in CI/CD pipelines to identify expensive operations and recommend gas-saving changes during development [7], [9].

Future Work

Several promising avenues for future research and development arise from this study:

- **Integration of Optimization Frameworks**: The creation of domain-specific frameworks or Solidity pre-compilers that automatically apply the techniques tested in this study could streamline smart contract development. These tools could analyze the code for inefficiencies and propose automated refactorings to improve gas usage.
- **Formal Verification of Optimized Contracts**: While gas efficiency is important, correctness and security are paramount. Future work should explore combining gas optimization with formal methods to ensure that refactored contracts still maintain intended behavior under all execution paths [7].
- **Cross-Layer Optimization Strategies**: Coordinating gas optimization at both the smart contract and network layer (e.g., using base fee prediction mechanisms) can further minimize costs in environments with volatile gas prices.
- **Use in Cross-Chain Interoperability**: As multi-chain DApps become more common, optimizing contract logic not only for gas but also for cross-chain call efficiency (e.g., using bridges) could be explored.
- **Application in Smart Contract Standards**: The insights gained from this research could be formalized into ERC/EIP proposals for best practices in efficient contract design. Promoting these practices could elevate the quality of smart contracts across the ecosystem.

In conclusion, the comparative evaluation of optimization techniques applied in this study underscores the multidimensional nature of gas efficiency. From minor code-level changes like `bytes32` and `uint256` usage to architectural decisions such as pattern simplification and immutable declarations, each technique contributes incrementally to a more scalable and cost-effective DApp ecosystem. These optimizations, when contextualized within real-world applications such as supply chain management, offer not only technical performance benefits but also commercial viability making blockchain adoption more feasible for large-scale enterprises.

By integrating design refinement, automation tools, and deployment strategies, developers can construct smart contracts that are secure, auditable, and economically efficient. This study provides a strong foundation for continued innovation in Ethereum contract optimization and paves the way for future research in automated tooling, cross-layer performance tuning, and standardized best practices.

REFERENCES

[1] T. Chen, X. Li, X. Luo, and X. Zhang, “Under-optimized smart contracts devour your money,” in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 442–446.

[2] G. A. Pierro and H. Rocha, “The Influence Factors on Ethereum Transaction Fees,” in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2019, pp. 24–30.

[3] B. D. Perera, M. P. S. Randunu, N. S. Wellappuli Arachchige, L. Rupasinghe, M. K. N. T. Kodithuwakkuge, and C. Liyanapathirana, “Optimizing Gas Fees for Cost-Effective E-voting Smart Contracts on the Ethereum Blockchain,” in *2023 5th International Conference on Advancements in Computing (ICAC)*, IEEE, 2023.

[4] N. Masla, J. Vyas, J. Gautam, R. N. Shaw, and A. Ghosh, “Reduction in Gas Cost for Blockchain Enabled Smart Contract,” in *Proc. 2021 IEEE 4th International Conference on Computing, Power and Communication Technologies (GUCON)*, 2021.

[5] C. Li, “Gas Estimation and Optimization for Smart Contracts on Ethereum,” in *Proc. 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[6] L. Marchesi, M. Marchesi, G. Destefanis, B. Barabino, and D. Tigano, “Design Patterns for Gas Optimization in Ethereum,” in *Proc. 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, 2020.

[7] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H. N. Lee, “Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contracts,” *IEEE Access*, vol. 10, pp. 6605–6621, 2022.

[8] K. Toyoda, K. Machi, Y. Ohtake, and A. N. Zhang, “Function-Level Bottleneck Analysis of Private Proof-of-Authority Ethereum Blockchain,” *IEEE Access*, vol. 8, pp. 141611–141621, 2020.

[9] W. Zou et al., “Smart Contract Development: Challenges and Opportunities,” *IEEE Transactions on Software Engineering*, vol. 27, pp. 2084–2106, 2019.

[10] A. Aldweesh, M. Alharby, M. Mehrnezhad, and A. Van Moorsel, “OpBench: A CPU Performance Benchmark for Ethereum Smart Contract Operation Code,” in *2019 IEEE International Conference on Blockchain*, 2019.

[11] D. Pramulia and B. Anggorojati, “Implementation and Evaluation of Blockchain-Based E-voting System with Ethereum and Metamask,” in *2020 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS)*, 2020.

[12] Y. K. Peker, X. Rodriguez, J. Ericsson, S. J. Lee, and A. J. Perez, “A Cost Analysis of IoT Sensor Data Storage on Blockchain via Smart Contracts,” *Electronics*, vol. 9, p. 224, 2020.

[13] G. Canfora, A. Di Sorbo, S. Laudanna, A. Vacca, and C. A. Visaggio, “Profiling Gas Leaks in Solidity Smart Contracts,” *arXiv preprint arXiv:2008.05449*, 2020.

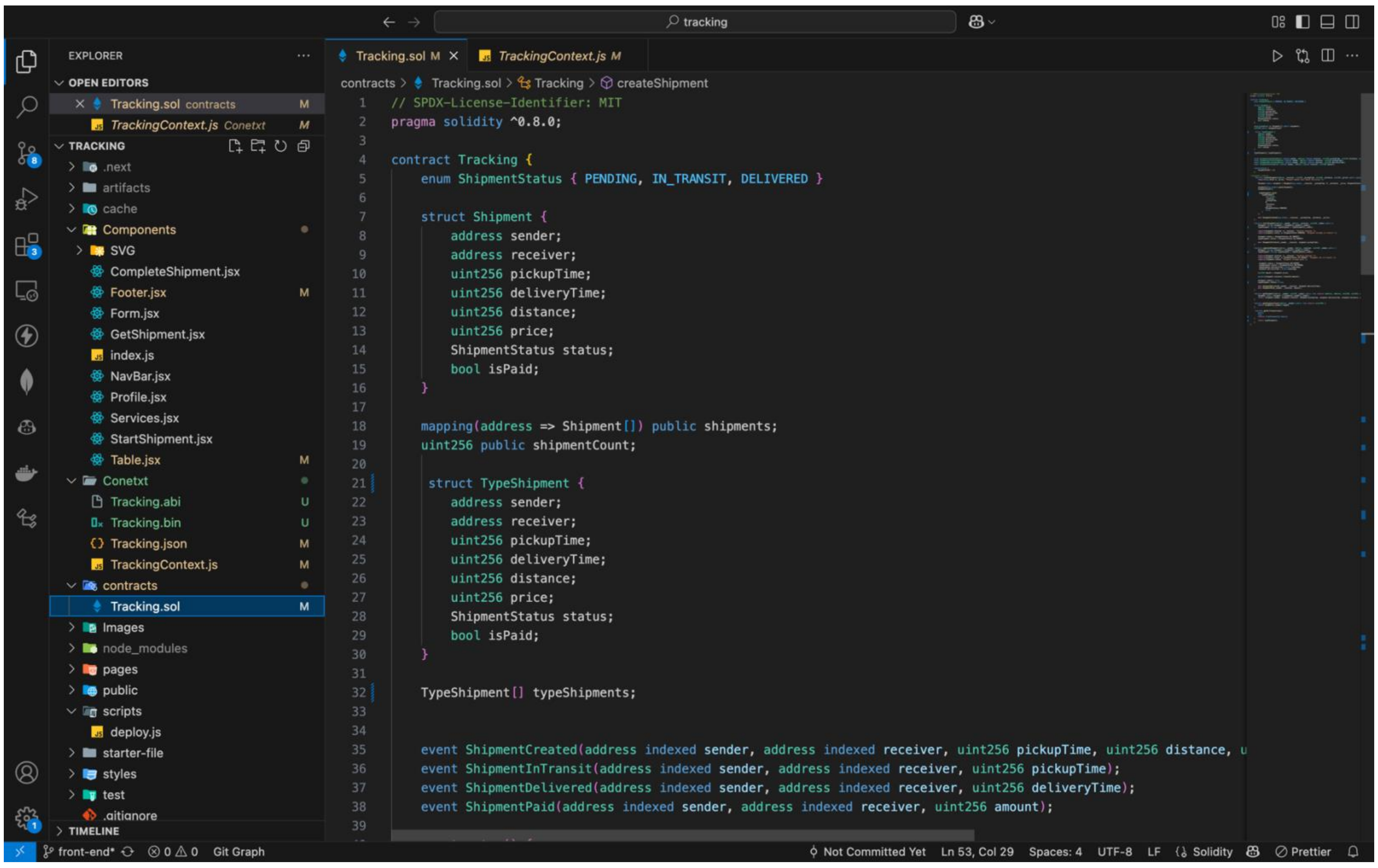
[14] S. Suriyakarn, “Solidity 102 #1: Keeping Gas Cost under Control,” *Medium*, 2019. [Online]. Available: https://medium.com/bandprotocol/solidity-102-1-keeping-gas-cost-under-control-ae95b835807f

[15] Solidity Documentation Team, “Types - Solidity 0.8.20 documentation,” [Online]. Available: https://docs.soliditylang.org/en/v0.8.20/types.html

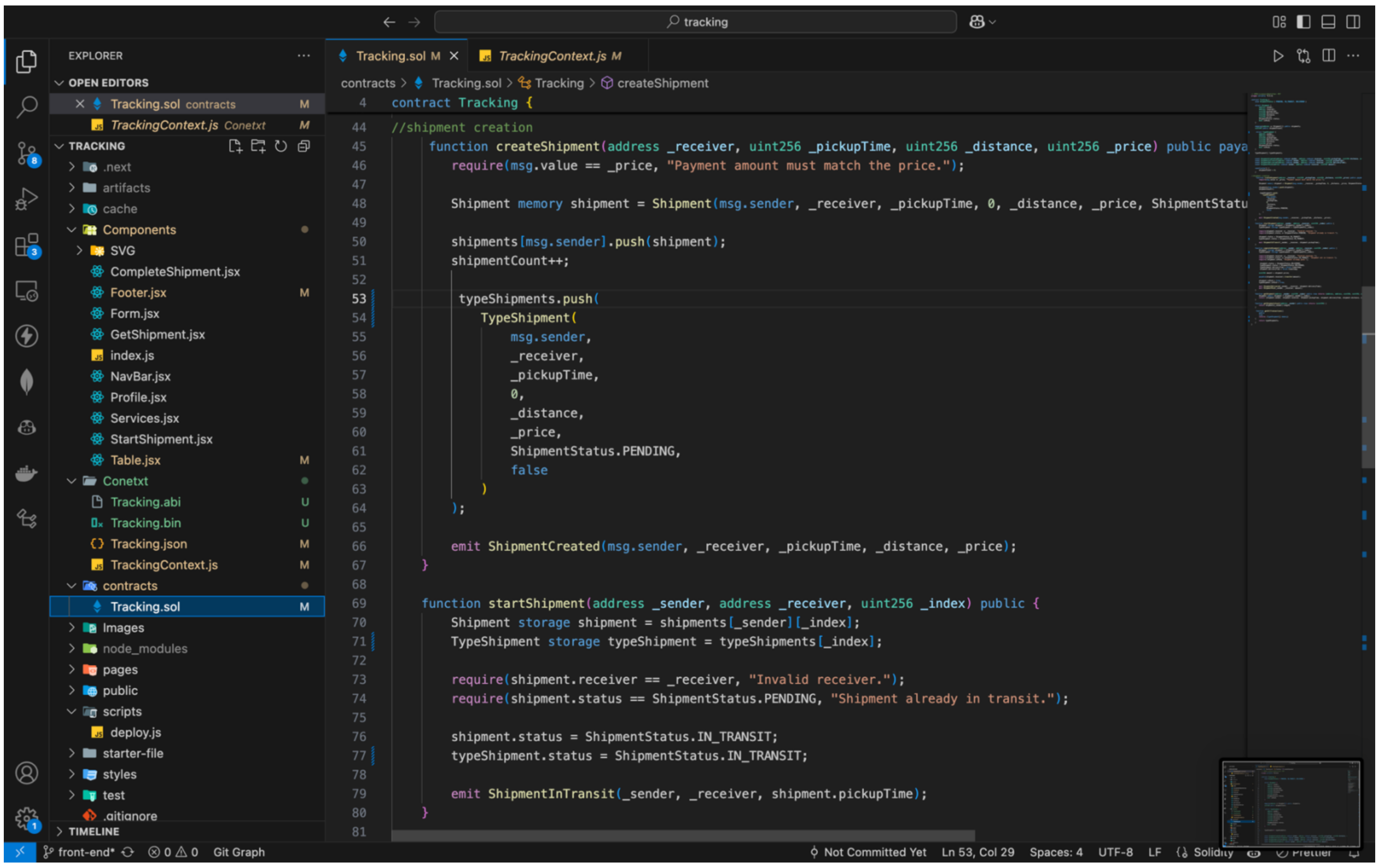
APPENDIX

Appendix A - Code Snippets

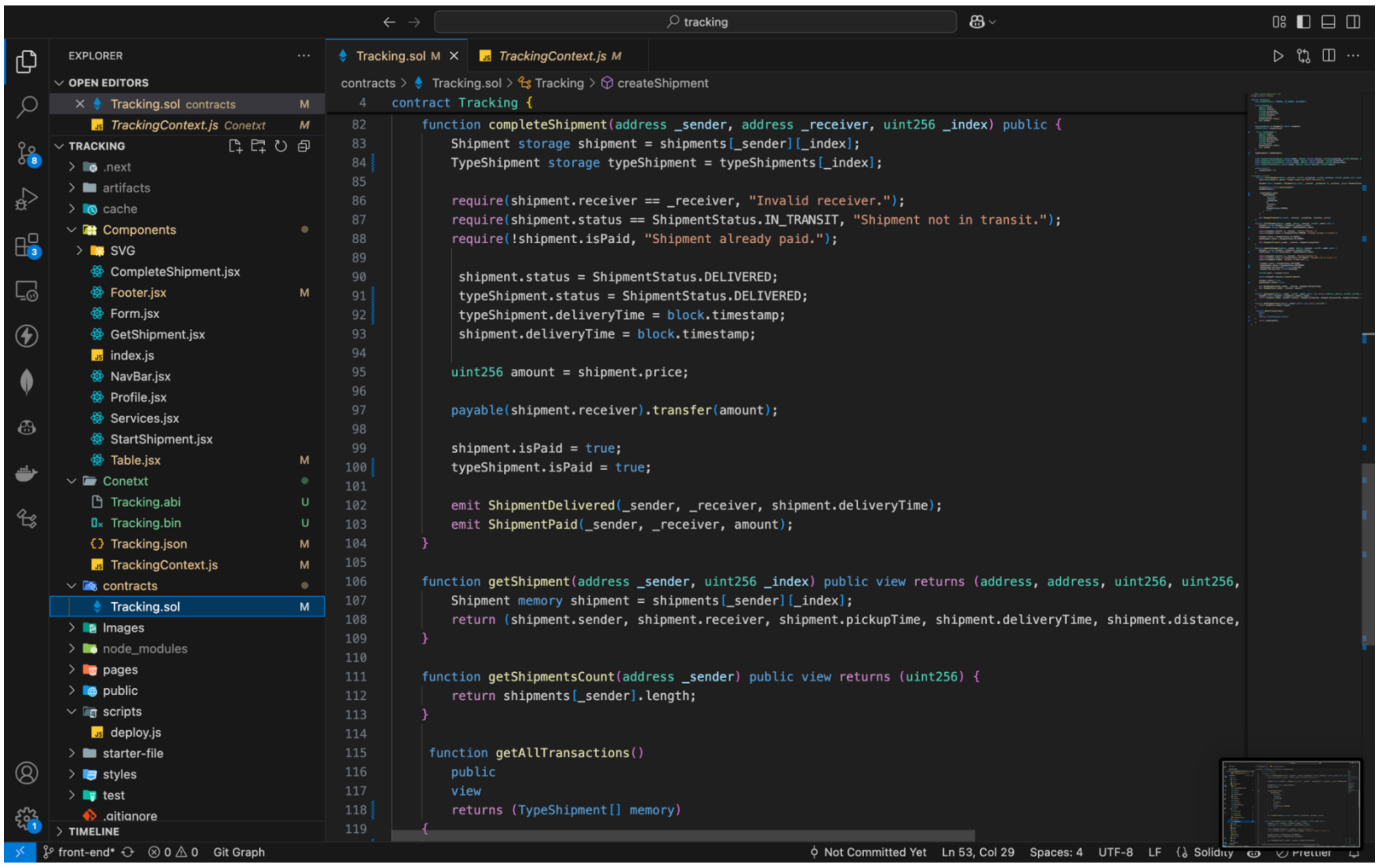
Smart contract code



```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Tracking {
    enum ShipmentStatus { PENDING, IN_TRANSIT, DELIVERED }

    struct Shipment {
        address sender;
        address receiver;
        uint256 pickupTime;
        uint256 deliveryTime;
        uint256 distance;
        uint256 price;
        ShipmentStatus status;
        bool isPaid;
    }

    mapping(address => Shipment[]) public shipments;
    uint256 public shipmentCount;

     struct TypeShipment {
        address sender;
        address receiver;
        uint256 pickupTime;
        uint256 deliveryTime;
        uint256 distance;
        uint256 price;
        ShipmentStatus status;
        bool isPaid;
    }

    TypeShipment[] typeShipments;


    event ShipmentCreated(address indexed sender, address indexed receiver, uint256 pickupTime, uint256 distance, u
    event ShipmentInTransit(address indexed sender, address indexed receiver, uint256 pickupTime);
    event ShipmentDelivered(address indexed sender, address indexed receiver, uint256 deliveryTime);
    event ShipmentPaid(address indexed sender, address indexed receiver, uint256 amount);

```

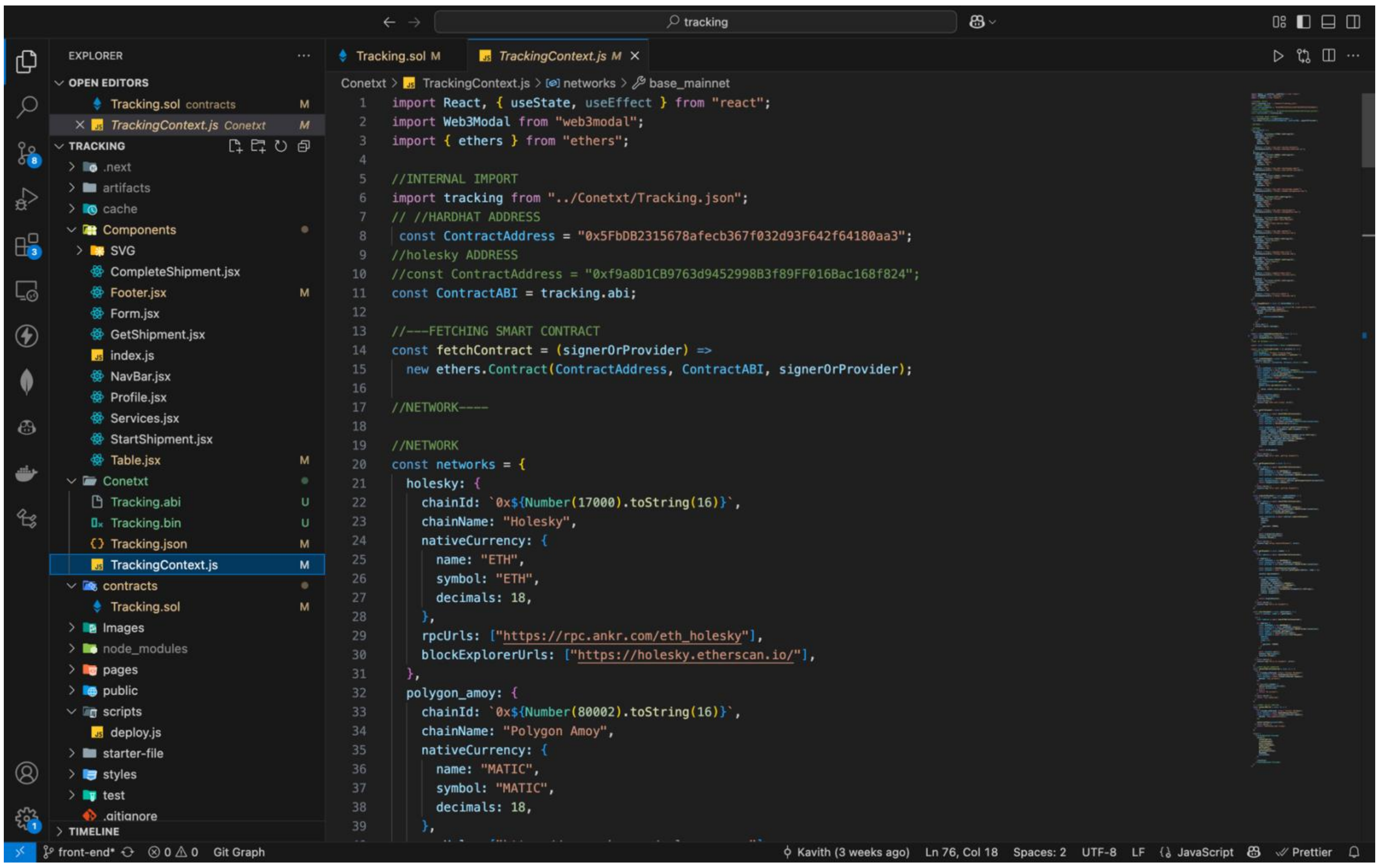


```
contract Tracking {
//shipment creation
     function createShipment(address _receiver, uint256 _pickupTime, uint256 _distance, uint256 _price) public paya
        require(msg.value == _price, "Payment amount must match the price.");

        Shipment memory shipment = Shipment(msg.sender, _receiver, _pickupTime, 0, _distance, _price, ShipmentStatu

        shipments[msg.sender].push(shipment);
        shipmentCount++;

         typeShipments.push(
            TypeShipment(
                msg.sender,
                _receiver,
                _pickupTime,
                0,
                _distance,
                _price,
                ShipmentStatus.PENDING,
                false
            )
        );

        emit ShipmentCreated(msg.sender, _receiver, _pickupTime, _distance, _price);
    }

    function startShipment(address _sender, address _receiver, uint256 _index) public {
        Shipment storage shipment = shipments[_sender][_index];
        TypeShipment storage typeShipment = typeShipments[_index];

        require(shipment.receiver == _receiver, "Invalid receiver.");
        require(shipment.status == ShipmentStatus.PENDING, "Shipment already in transit.");

        shipment.status = ShipmentStatus.IN_TRANSIT;
        typeShipment.status = ShipmentStatus.IN_TRANSIT;

        emit ShipmentInTransit(_sender, _receiver, shipment.pickupTime);
    }

```



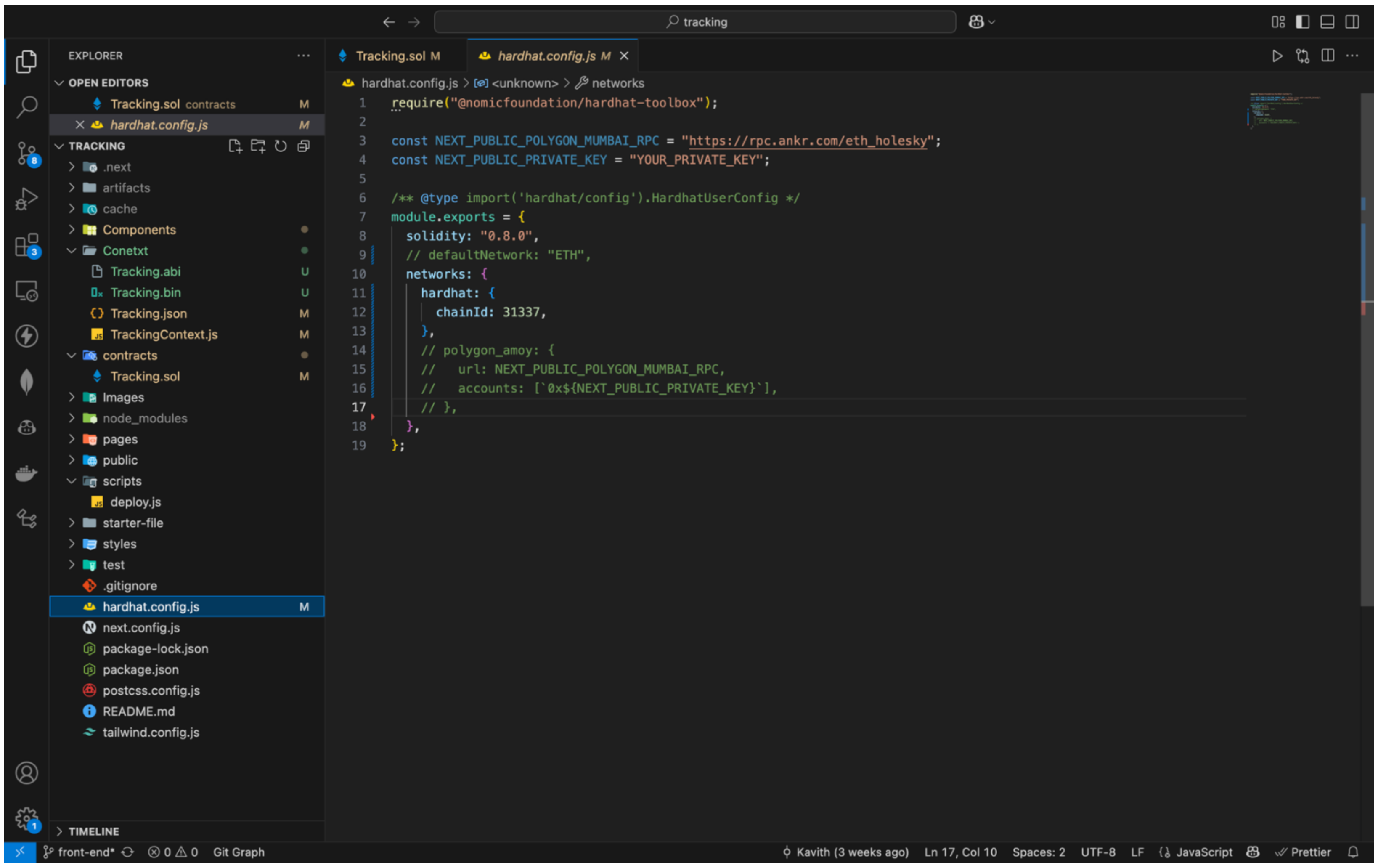
```
contract Tracking {
    function completeShipment(address _sender, address _receiver, uint256 _index) public {
        Shipment storage shipment = shipments[_sender][_index];
        TypeShipment storage typeShipment = typeShipments[_index];

        require(shipment.receiver == _receiver, "Invalid receiver.");
        require(shipment.status == ShipmentStatus.IN_TRANSIT, "Shipment not in transit.");
        require(!shipment.isPaid, "Shipment already paid.");

         shipment.status = ShipmentStatus.DELIVERED;
         typeShipment.status = ShipmentStatus.DELIVERED;
         typeShipment.deliveryTime = block.timestamp;
         shipment.deliveryTime = block.timestamp;

        uint256 amount = shipment.price;

        payable(shipment.receiver).transfer(amount);

        shipment.isPaid = true;
        typeShipment.isPaid = true;

        emit ShipmentDelivered(_sender, _receiver, shipment.deliveryTime);
        emit ShipmentPaid(_sender, _receiver, amount);
    }

    function getShipment(address _sender, uint256 _index) public view returns (address, address, uint256, uint256,
        Shipment memory shipment = shipments[_sender][_index];
        return (shipment.sender, shipment.receiver, shipment.pickupTime, shipment.deliveryTime, shipment.distance,
    }

    function getShipmentsCount(address _sender) public view returns (uint256) {
        return shipments[_sender].length;
    }

     function getAllTransactions()
        public
        view
        returns (TypeShipment[] memory)
    {
```

Deployment code



```
import React, { useState, useEffect } from "react";
import Web3Modal from "web3modal";
import { ethers } from "ethers";

//INTERNAL IMPORT
import tracking from "../Conetxt/Tracking.json";
// //HARDHAT ADDRESS
 const ContractAddress = "0x5FbDB2315678afecb367f032d93F642f64180aa3";
//holesky ADDRESS
//const ContractAddress = "0xf9a8D1CB9763d9452998B3f89FF016Bac168f824";
const ContractABI = tracking.abi;

//---FETCHING SMART CONTRACT
const fetchContract = (signerOrProvider) =>
  new ethers.Contract(ContractAddress, ContractABI, signerOrProvider);

//NETWORK----

//NETWORK
const networks = {
  holesky: {
    chainId: `0x${Number(17000).toString(16)}`,
    chainName: "Holesky",
    nativeCurrency: {
      name: "ETH",
      symbol: "ETH",
      decimals: 18,
    },
    rpcUrls: ["https://rpc.ankr.com/eth_holesky"],
    blockExplorerUrls: ["https://holesky.etherscan.io/"],
  },
  polygon_amoy: {
    chainId: `0x${Number(80002).toString(16)}`,
    chainName: "Polygon Amoy",
    nativeCurrency: {
      name: "MATIC",
      symbol: "MATIC",
      decimals: 18,
    },
```

Hard Hat configuration code

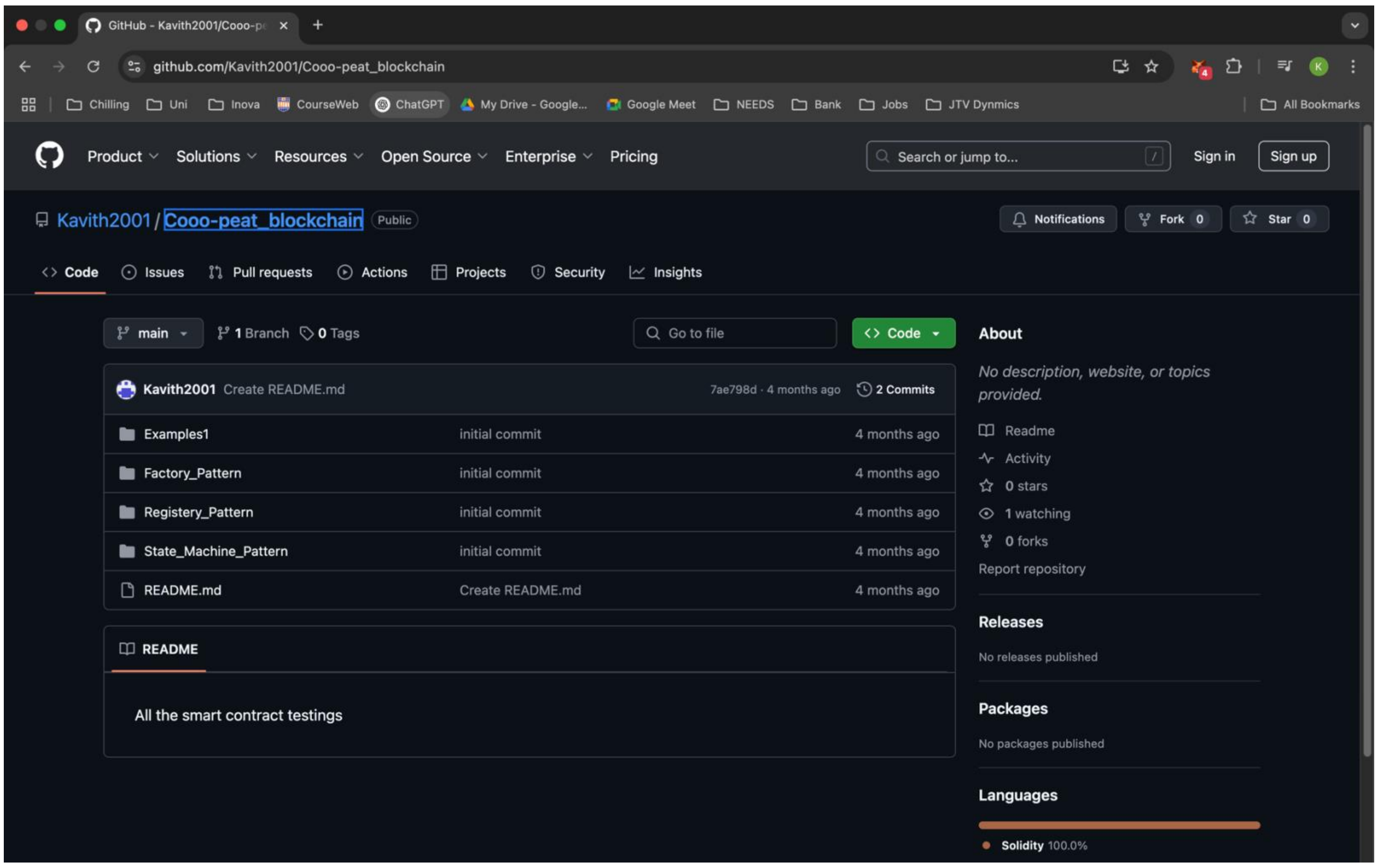


```
require("@nomicfoundation/hardhat-toolbox");

const NEXT_PUBLIC_POLYGON_MUMBAI_RPC = "https://rpc.ankr.com/eth_holesky";
const NEXT_PUBLIC_PRIVATE_KEY = "YOUR_PRIVATE_KEY";

/** @type import('hardhat/config').HardhatUserConfig */
module.exports = {
  solidity: "0.8.0",
  // defaultNetwork: "ETH",
  networks: {
    hardhat: {
      chainId: 31337,
    },
    // polygon_amoy: {
    //   url: NEXT_PUBLIC_POLYGON_MUMBAI_RPC,
    //   accounts: [`0x${NEXT_PUBLIC_PRIVATE_KEY}`],
    // },
  },
};
```

Appendix B – Github Repositories

The code sets to for testing the optimization techniques

https://github.com/Kavith2001/Cooo-peat_blockchain.git



github.com/Kavith2001/Cooo-peat_blockchain

Kavith2001 / Cooo-peat_blockchain Public

main · 1 Branch · 0 Tags

Kavith2001 Create README.md · 7ae798d · 4 months ago · 2 Commits

Examples1	initial commit	4 months ago
Factory_Pattern	initial commit	4 months ago
Registery_Pattern	initial commit	4 months ago
State_Machine_Pattern	initial commit	4 months ago
README.md	Create README.md	4 months ago

README

All the smart contract testings

About

No description, website, or topics provided.

Readme · Activity · 0 stars · 1 watching · 0 forks

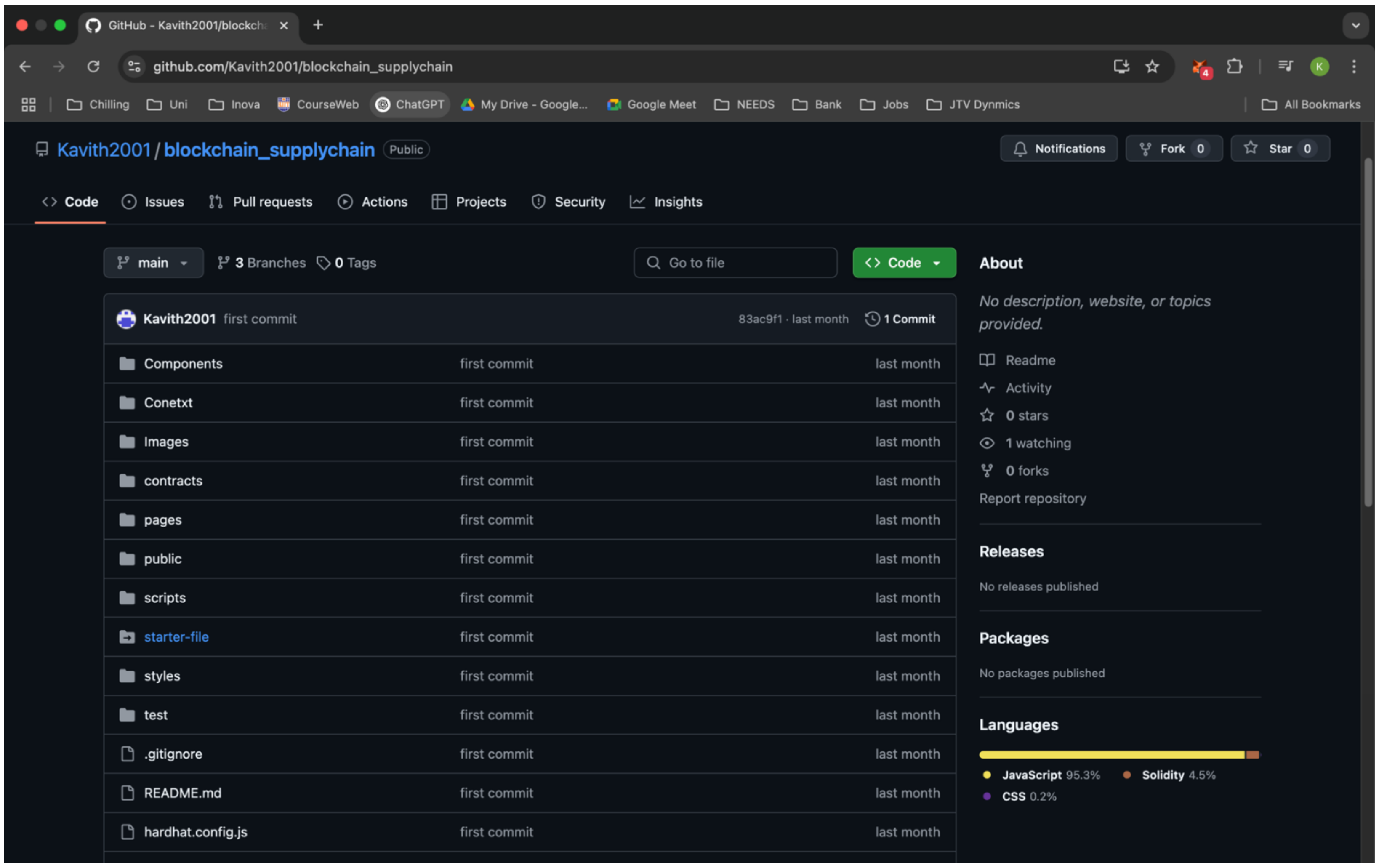
Releases: No releases published

Packages: No packages published

Languages: Solidity 100.0%

Coco-peat system

https://github.com/Kavith2001/blockchain_supplychain.git



Kavith2001 / blockchain_supplychain Public

3 Branches 0 Tags

Kavith2001 first commit 83ac9f1 · last month 1 Commit

Components	first commit	last month
Conetxt	first commit	last month
Images	first commit	last month
contracts	first commit	last month
pages	first commit	last month
public	first commit	last month
scripts	first commit	last month
starter-file	first commit	last month
styles	first commit	last month
test	first commit	last month
.gitignore	first commit	last month
README.md	first commit	last month
hardhat.config.js	first commit	last month

About

No description, website, or topics provided.

Readme
Activity
0 stars
1 watching
0 forks
Report repository

Releases

No releases published

Packages

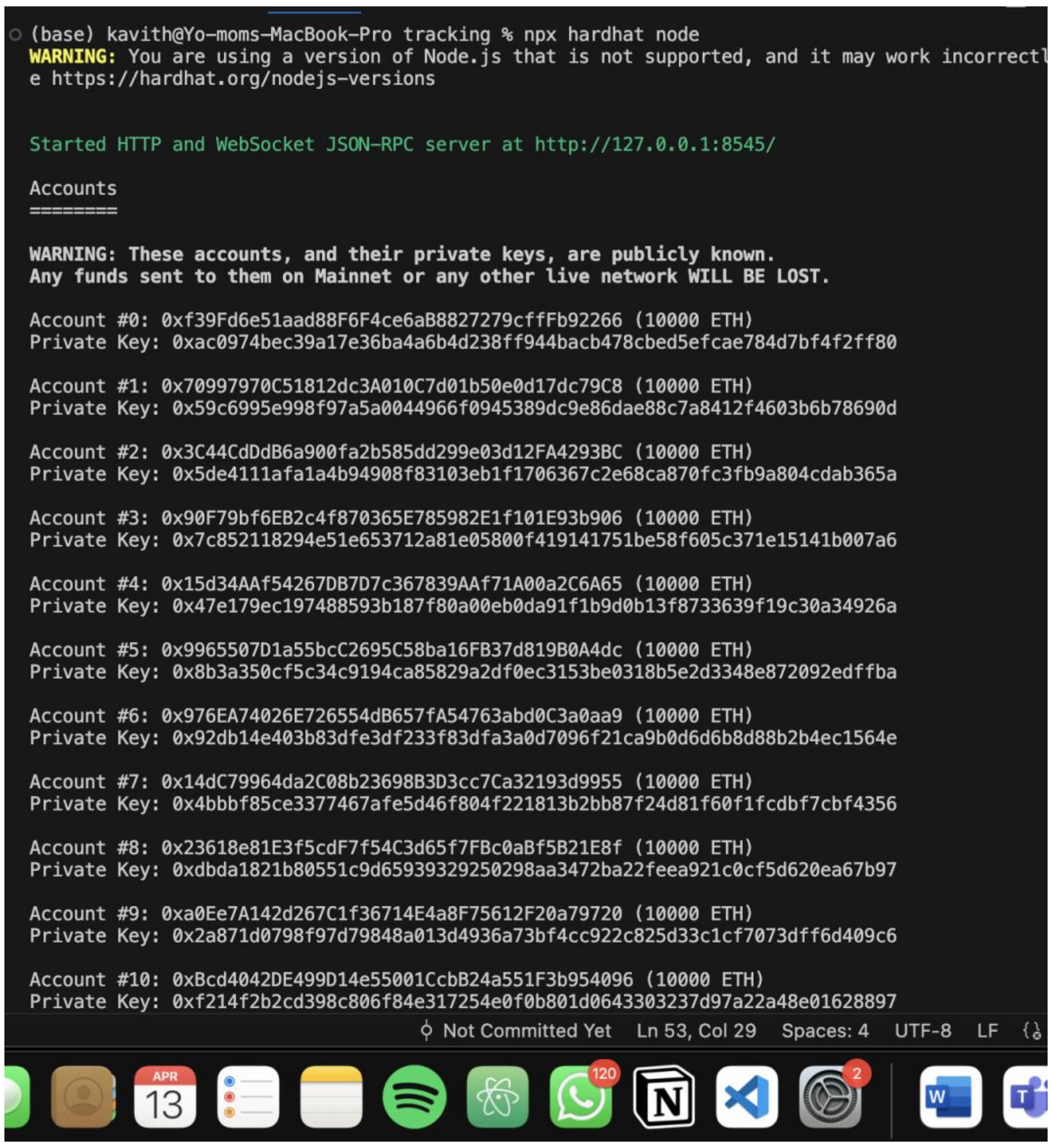
No packages published

Languages

● JavaScript 95.3% ● Solidity 4.5% ● CSS 0.2%

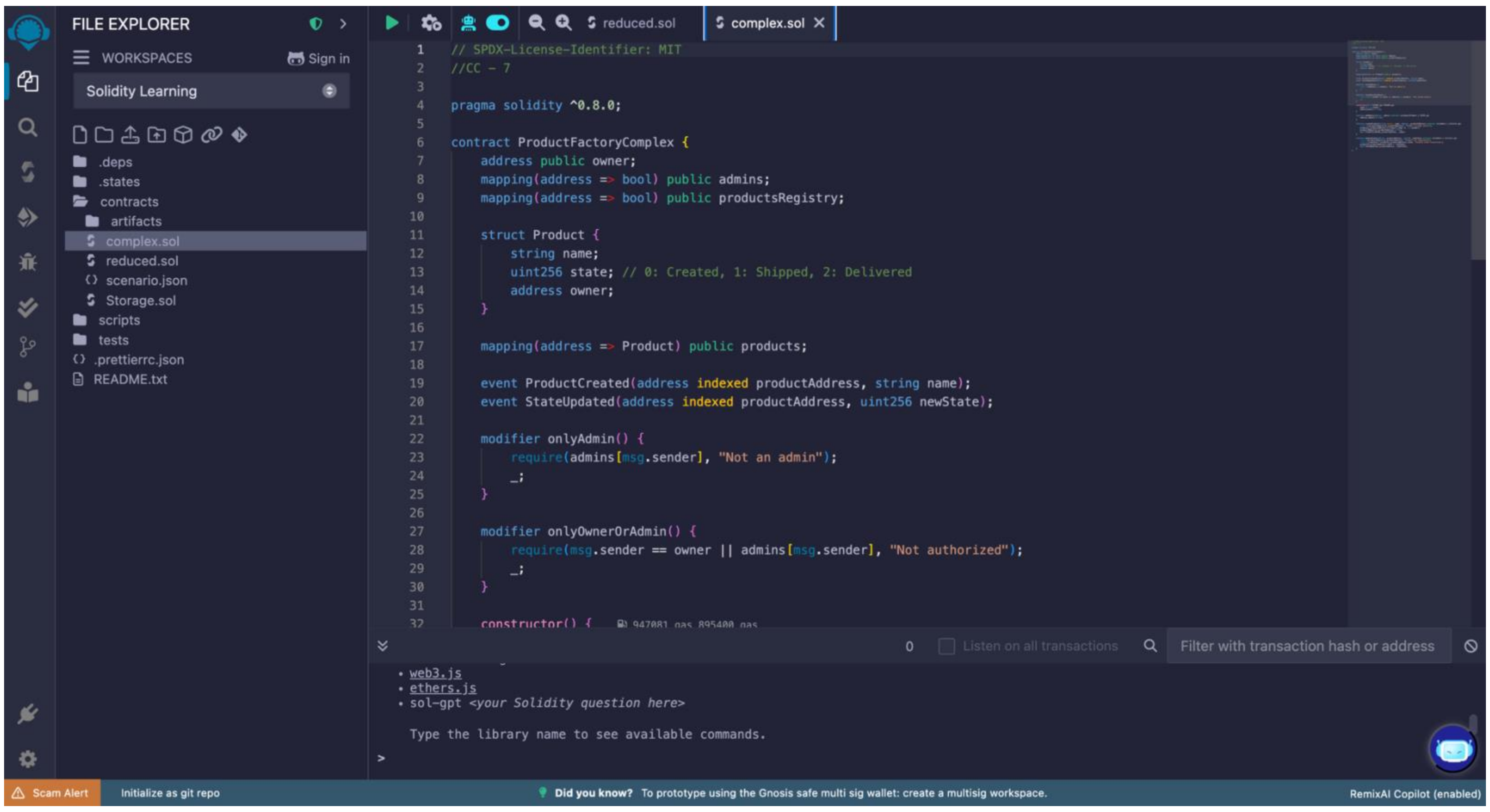
Appendix C – Other screenshots

Hardhat startup



```
(base) kavith@Yo-moms-MacBook-Pro tracking % npx hardhat node
WARNING: You are using a version of Node.js that is not supported, and it may work incorrectl
e https://hardhat.org/nodejs-versions


Started HTTP and WebSocket JSON-RPC server at http://127.0.0.1:8545/

Accounts
========

WARNING: These accounts, and their private keys, are publicly known.
Any funds sent to them on Mainnet or any other live network WILL BE LOST.

Account #0: 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266 (10000 ETH)
Private Key: 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80

Account #1: 0x70997970C51812dc3A010C7d01b50e0d17dc79C8 (10000 ETH)
Private Key: 0x59c6995e998f97a5a0044966f0945389dc9e86dae88c7a8412f4603b6b78690d

Account #2: 0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC (10000 ETH)
Private Key: 0x5de4111afa1a4b94908f83103eb1f1706367c2e68ca870fc3fb9a804cdab365a

Account #3: 0x90F79bf6EB2c4f870365E785982E1f101E93b906 (10000 ETH)
Private Key: 0x7c852118294e51e653712a81e05800f419141751be58f605c371e15141b007a6

Account #4: 0x15d34AAf54267DB7D7c367839AAf71A00a2C6A65 (10000 ETH)
Private Key: 0x47e179ec197488593b187f80a00eb0da91f1b9d0b13f8733639f19c30a34926a

Account #5: 0x9965507D1a55bcC2695C58ba16FB37d819B0A4dc (10000 ETH)
Private Key: 0x8b3a350cf5c34c9194ca85829a2df0ec3153be0318b5e2d3348e872092edffba

Account #6: 0x976EA74026E726554dB657fA54763abd0C3a0aa9 (10000 ETH)
Private Key: 0x92db14e403b83dfe3df233f83dfa3a0d7096f21ca9b0d6d6b8d88b2b4ec1564e

Account #7: 0x14dC79964da2C08b23698B3D3cc7Ca32193d9955 (10000 ETH)
Private Key: 0x4bbbf85ce3377467afe5d46f804f221813b2bb87f24d81f60f1fcdbf7cbf4356

Account #8: 0x23618e81E3f5cdF7f54C3d65f7FBc0aBf5B21E8f (10000 ETH)
Private Key: 0xdbda1821b80551c9d65939329250298aa3472ba22feea921c0cf5d620ea67b97

Account #9: 0xa0Ee7A142d267C1f36714E4a8F75612F20a79720 (10000 ETH)
Private Key: 0x2a871d0798f97d79848a013d4936a73bf4cc922c825d33c1cf7073dff6d409c6

Account #10: 0xBcd4042DE499D14e55001CcbB24a551F3b954096 (10000 ETH)
Private Key: 0xf214f2b2cd398c806f84e317254e0f0b801d0643303237d97a22a48e01628897
```

Remix IDE

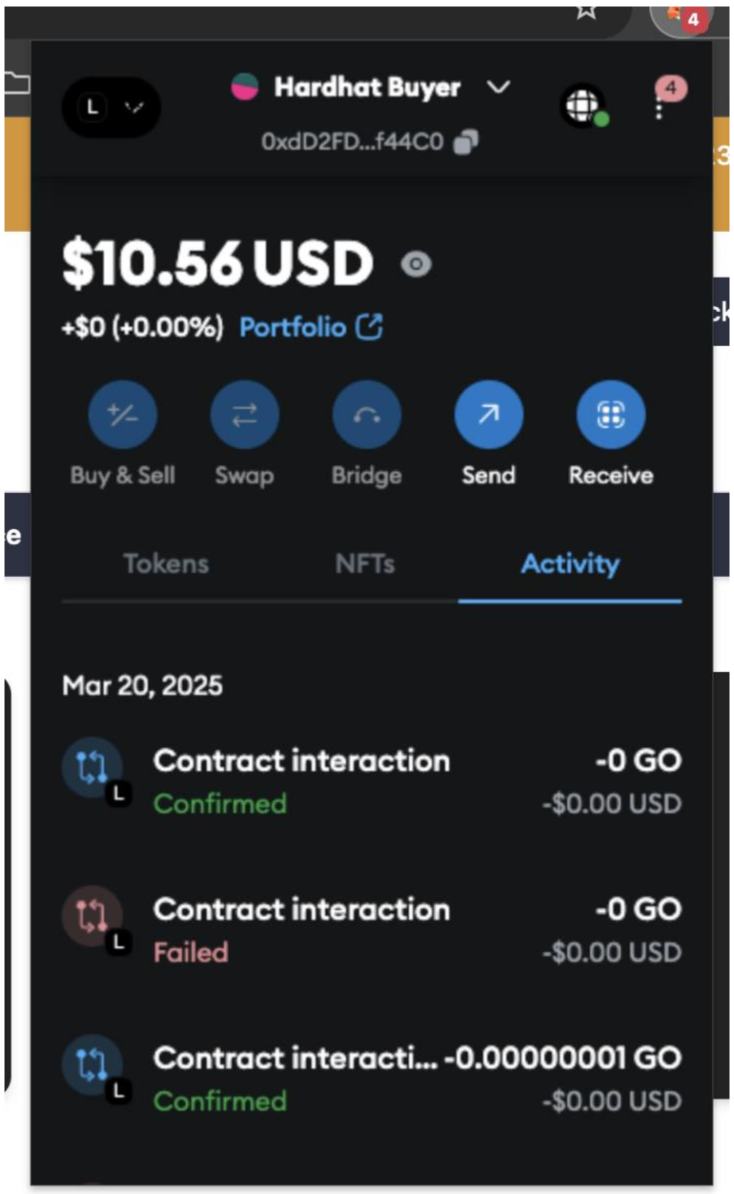


```
// SPDX-License-Identifier: MIT
//CC - 7

pragma solidity ^0.8.0;

contract ProductFactoryComplex {
    address public owner;
    mapping(address => bool) public admins;
    mapping(address => bool) public productsRegistry;

    struct Product {
        string name;
        uint256 state; // 0: Created, 1: Shipped, 2: Delivered
        address owner;
    }

    mapping(address => Product) public products;

    event ProductCreated(address indexed productAddress, string name);
    event StateUpdated(address indexed productAddress, uint256 newState);

    modifier onlyAdmin() {
        require(admins[msg.sender], "Not an admin");
        _;
    }

    modifier onlyOwnerOrAdmin() {
        require(msg.sender == owner || admins[msg.sender], "Not authorized");
        _;
    }

    constructor() {    947081 gas 895400 gas
```

```
0
• web3.js
• ethers.js
• sol-gpt <your Solidity question here>

  Type the library name to see available commands.
>
```

Did you know? To prototype using the Gnosis safe multi sig wallet: create a multisig workspace.

Meta mask wallet



$10.56 USD
+$0 (+0.00%)

Mar 20, 2025

Contract interaction	-0 GO
Confirmed	-$0.00 USD
Contract interaction	-0 GO
Failed	-$0.00 USD
Contract interacti...	-0.00000001 GO
Confirmed	-$0.00 USD