# Gas-Efficient Smart Contract Design: Quantifying Refactoring Impact on EVM Execution Costs

Nur Haliza Abdul Wahab[1]*, Juniardi Nur Fadila[2], Nur Faszha Razali[3], Keng Yinn Wong[4]
Department of Computer Science, Faculty of Computing, University Technology Malaysia, Johor, Malaysia [1, 2]
Digital Trust Centre, Nanyang Technological University, Singapore [1]
Department of Informatics Engineering, UIN Maulana Malik Ibrahim, Malang, Indonesia [2]
Kolej Kemahiran Tinggi MARA, Petaling Jaya, Selangor, Malaysia [3]
Faculty of Mechanical Engineering, University Technology Malaysia, Johor, Malaysia [4]

*Abstract*—**High transaction costs remain a major barrier to the scalability of Ethereum-based decentralized applications (DApps), particularly when smart contracts are computationally inefficient. Although the Solidity compiler optimizer can reduce bytecode size and improve some low-level patterns, it does not fully address structural inefficiencies in storage layout and state mutation. This study introduces controlled empirical research on the topic of manual smart contract refactoring approaches with the aim of quantifying their impact on gas usage and execution cost in the Ethereum Virtual Machine (EVM). The Remix Integrated Development Environment (IDE) and a synchronized Go-Ethereum (Geth) node (version 1.13.5) were configured to create a controlled experimental environment. This environment was connected to the Sepolia Testnet to approximate conditions similar to the Ethereum Mainnet. The role of high-cost storage operations such as SSTORE was analyzed using opcode-level transaction traces, which were collected using debug_traceTransaction. The proposed refactoring plan implies the alignment of storage slots by systematically packing the variables and data location optimization (calldata and memory) to minimize unnecessary memory allocation. The experiments show gas reductions of up to 40.68% for storage-intensive functions, with an average reduction of 28.5% across all evaluated test cases. Moreover, the findings at the opcode level have shown that it is possible to reduce the costs of unnecessary storage writes without impacting the correct functional performance of the execution. Overall, the findings show that storage-aware manual refactoring is a viable strategy for improving runtime efficiency and reducing the execution cost of Layer-1 smart contracts.**

***Keywords—Ethereum smart contract; solidity refactoring; gas optimization; Ethereum Virtual Machine (EVM); opcode tracing***

## I. Introduction

Ethereum is a widely used decentralized application platform whose smart contracts are executed through the Ethereum Virtual Machine (EVM) [1], [2]. However, the wider use of Ethereum is still constrained by the gas cost of executing smart contracts [3], [4]. Although gas helps prevent computational abuse, high gas requirements remain a practical barrier to scalability [5], [6]. Ethereum has adopted Proof of Stake (POS) and implemented EIP-1559, which stabilizes market fees. But even the gas that is used by a smart contract is still heavily reliant on the computational and storage requirements of the contract itself. This implementation expense has a direct impact on the cost-effectiveness of Decentralized Applications (DApps) [7], [8].

Despite these protocol-level changes, contract-level code complexity still affects execution cost. This is especially important for high-frequency DApps (such as decentralized order books or real-time gaming), because even marginal inefficiencies in state management will make a project economically unviable for mainstream adoption [9], [10], [11].

While standard Solidity compiler optimizer *'solc --optimize'* as well as automated tools like *'GasReducer'*, *'sOptimize'* make good progress on reducing the size of bytecode using techniques like constant folding and function inlining, they don't explicitly take developer-defined architectural choices, in terms of storage layout and data-handling [12], [13], [14]. Since the compiler has no idea of the organization of storage specific to the contract, it is not able to automatically implement structural optimizations like packing storage slots, such as storage-slot packing and calldata-based parameter handling [15], [16]. This study examines whether manual refactoring can improve runtime gas efficiency in selected Solidity patterns. This motivates attention to storage-aware refactoring strategies.

To address such an efficiency difference, a rigid comparative empirical analysis is made in this study between the traditional automated methods of optimization and advanced manual refactoring techniques. The method applied in this study to examine the cost of execution in detail is the opcode-level transaction tracing. This is especially so in applications that are highly interactive in the blockchain. Based on this, the aims of the study are the following:

- Implement a manual refactoring framework for Solidity contracts, focusing on storage slot alignment (Variable Packing) and data location calldata vs. Memory.
- Conduct comparative empirical research to quantify the gas savings when deployed and run-time execution with respect to the case of the baseline and optimized versions.
- Determine the deterministic behavior of EVM through trace fine-grained opcodes with a synchronized private Geth node.

To further direct the empirical study, the following research questions are derived in this study:

*Corresponding author.
This research was supported by a UTM Fundamental Research Grant Q.J130000.3828.23H38).

- RQ1: Does storage-slot alignment and data-location optimization of manual refactoring decrease gas consumption of storage-intensive Solidity functions when controlled by EVM execution?
- RQ2: Which optimization strategy contributes more directly to gas reduction at the opcode level: variable packing or calldata-based parameter handling?
- RQ3: How significant are the observed gas savings in terms of meaningful economic benefits to high-frequency cases of decentralized application?

These research questions are intended to provide a more analytical framework for the structure of the assessment of the technical and economic impact of manual smart contract optimization.

This study empirically evaluates the manual refactoring workflow for reducing EVM execution cost under controlled test conditions. By presenting at the opcode level evidence of runtime gas savings up to 40.68%, this study shows that manual interventions in storage-slot alignment and data-location management can improve execution efficiency and lower transaction-level cost for the evaluated smart contract functions:

- An opcode-level empirical framework for analyzing gas optimization in Ethereum smart contracts within a controlled EVM experimental environment.
- A manual Solidity refactoring methodology targeting storage access patterns and calldata optimization to reduce execution costs.
- Experimental validation demonstrating gas reductions of up to 40.68% through structural code refactoring under controlled Sepolia Testnet conditions.

## II. Literature Review

Previous studies on the efficiency of Ethereum smart contracts can be classified into four categories: gas-fee dynamic, EVM resource pricing, automatic optimization tools, and experimental validation environments [17], [18], [19]. However, there is little evidence in the literature on the impact of manual refactoring on execution cost in controlled EVM environments at the level of opcodes [11], [18]. This gap indicates the need for regulated empirical assessment of manual refactoring utilizing low-level execution traces.

### A. Economic Scalability and Gas Volatility

Recent protocol changes have not removed the importance of contract-level gas efficiency [12], [20]. Some studies say that the cost of gas is a big problem for DApps since bugs in contract code make transactions more expensive for users [21], [22].

Under EIP-1559, transaction fees are divided into a protocol-determined base fee and a priority fee [23]. Although fee levels vary with network conditions, developers still influence overall transaction cost through the gas consumption of contract execution [16], [17]. Because fee levels are externally determined, developer influence lies primarily in reducing contract-level gas consumption.

To frame optimization more clearly, gas mechanisms can be categorized according to their functions within the blockchain system [24]. In Ethereum, gas serves two main purposes: it acts as a pricing mechanism that discourages computational abuse, and it helps mitigate unbounded execution and denial-of-service risks [9], [23]. The given mechanism can be further divided into a two-tiered taxonomy of source control:

*1) Protocol control (external):* This includes the Base Fee calculated by the EIP-1559 protocol depending on block congestion [6], [23]. Although L2 networks like Polygon or Optimistic Rollups solutions have reduced fee taxonomies due to compression of the data, the logic behind the execution is still governed by the EVM rules [25].

*2) Developer source control (internal):* This is the strategic control of storage and use of computational resources at the implementation level [3], [4]. Although fee-market charges under EIP-1559 include a Base Fee and a Priority Fee, developers primarily influence total gas usage through contract-level efficiency, as illustrated in Fig. 1. Even though developers are not in control of the Base Fee in the market, they have ultimate control over gas limit through the efficiency of their code [21].

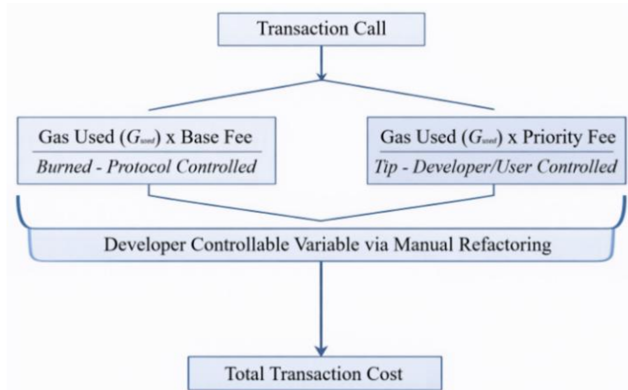



Fig. 1. Conceptual framework of Ethereum gas fee calculation.

Prior studies show that private Geth nodes support controlled and reproducible measurement of EVM execution behavior, making them suitable for fine-grained gas analysis [5], [26].

### B. EVM Operating Cost Calculation Mechanism

The deterministic aspect of EVM calculation of costs is a requirement before one can detect structural inefficiencies in smart contracts [10], [18]. Reducing code inefficiency remains important for improving execution efficiency. Each operation is metered to support network security and to prevent unbounded execution, including risks associated with infinite looping [10], [27]. The total gas used by a transaction $G_{total}$ is a combination of computational, storage, and memory resources usage, that is controlled by Eq. (1):

$$G_{total} = G_{intrinsic} + G_{execution} + G_{memory} \quad (1)$$

TABLE I. GAS-INTENSIVE EVM OPCODES AND OPTIMIZATION TARGETS

| Opcode | Typical Gas Cost | Resource Category | Optimization Relevance | Refactoring Target |
|---|---|---|---|---|
| SSTORE | 20,000 (New) / 2, 900 (Dirty) | Persistent storage writes | Very High | Variable Packing |
| SLOAD | 2, 100 (Cold) / 100 (Warm) | Persistent storage read | High | Reduce repeated reads |
| MLOAD / MSTORE | 3 | Memory | Low | Temporary caching |
| CALLDATACOPY | 3 + 3 x words | Input data copy | Moderate | Calldata vs. Memory |
| ADD / SUB | 3 | Stack Arithmetic | Low | Basic Logic |

where:

- $G_{instrinsic}$: The fixed 21,000 units of gas to pay the base cost per transaction contained in the block.
- $G_{execution}$: The total cost of the opcodes that have been performed at runtime. This consists of state operations of high cost, such as *SSTORE* (up to 20,000 gas) and of low cost, such as *ADD* (3 gas).
- $G_{memory}$: Costs that are paid in quadratic proportion to the increase in memory used in contract execution.

Analysis through Eq. (1) reveals that the most impactful optimization vectors lie within $G_{execution}$ and $G_{memory}$. For instance, [20] highlight that identifying costly bytecodes and replacing them with gas-efficient patterns is highly required to mitigate these expenses. Techniques such as Variable Packing directly target the reduction of $G_{execution}$ by minimizing the expensive of storage operations, while shifting data location to *CALLDATA* mitigates the $G_{memory}$ expansion penalty.

However, as mentioned by [5], the issue of linking these technical assurances to particular audit statements is the focus of research. Table I has been synthesized as comparative technical specifications of these opcodes, which are resource-intensive.

Table I summarizes the gas-expensive EVM opcodes that are most relevant for this study. In particular, *SSTORE* dominates state-changing cost, making storage-slot packing a key target for reducing deployment and runtime overhead. By contrast, input-copy operations such as *CALLDATACOPY* highlight the importance of parameter handling through calldata to avoid unnecessary memory-expansion costs. Collectively, these characteristics of the opcodes give the technical underpinning for the refactoring strategies that are evaluated in this work.

On top of this classification, Section III introduces the experimental refactoring workflow that is used to baseline smart contract implementations.

## C. Manual Refactoring vs Automated Tools

Compiler-level optimization can improve common low-level patterns, but it does not usually perform structural changes such as storage-slot reorganization or context-specific data-layout decisions [28].

In order to overcome these shortcomings, different specific automated tools have been produced. *GasReducer* was presented in [15] as a tool that recognizes 24 code anti-patterns using the bytecode representation, and substitutes them with more efficient operation sequences.

TABLE II. COMPARISON OF SMART CONTRACT GAS OPTIMIZATION APPROACHES AND THEIR LIMITATIONS

| Optimization Approach / Study | Primary Focus | Core Technique / Mechanism | Strengths / Contributions | Critical Limitation (Efficiency Gap) |
|---|---|---|---|---|
| Solidity Compiler Optimizer (--optimize) [29]. | Bytecode size reduction, basic runtime optimization | Deduplication, jumpdest removal, peephole optimizations | Provides baseline automated improvements with minimal developer effort | Cannot restructure storage layout or apply variable packing; limited awareness of opcode-level cost (e.g., *SSTORE*) |
| GasReducer [30] | Anti-pattern detection at bytecode level | Replaces inefficient opcode sequences (24 patterns) | Effective for removing known inefficiencies automatically | Pattern-based; lacks semantic context and cannot perform deep architectural refactoring |
| sOptimize [31]. | Dead code and redundancy elimination | Control Flow Graph (CFG) + lazy annotation | Improves execution efficiency through redundant instruction removal | Limited ability to optimize storage-heavy functions or calldata transitions |
| GASOL [2], [18], [32]. | Minimizing memory/storage operations | SMT-based optimization encoding | Formal approach to reduce expensive memory and storage usage | Computationally complex; does not provide developer-guided storage slot packing |
| Slither Refactoring Framework [33] | Static detection of storage inefficiencies | Static analysis of contract structure | Identifies refactoring opportunities and vulnerability patterns | Context-blind: cannot apply dynamic optimizations such as calldata vs. memory restructuring |
| GasOptiScan (AST-based tools) [34]. | Loop and syntax-level optimization | Abstract Syntax Tree (AST) parsing for pattern matching | Useful for detecting loop-fusible or structural patterns | Cannot optimize calling-context-dependent transitions (e.g., memory → *CALLDATA*) |
| Audit Formalization Gap [35]. | Verification and guarantees mapping | Linking optimization guarantees to audit assertions | Highlights need for stronger runtime-focused validation | Automated tools prioritize deployment reduction over runtime opcode efficiency |
| Manual Refactoring Framework Evaluated in This Study | Runtime execution efficiency at opcode level | Variable packing, calldata optimization, stack caching, opcode tracing via debug_traceTransaction | Provides opcode-level empirical evidence of runtime gas savings through storage-aware restructuring | Requires deeper EVM expertise and manual developer intervention (higher barrier to entry) |

TABLE III. COMPARATIVE ANALYSIS OF EXPERIMENTAL ENVIRONMENTS FOR GAS VALIDATION

| Blockchain Network | Consensus Mechanism | Avg. Transaction Cost (USD) | Throughput (TPS) | Cost Implication for Developers |
|---|---|---|---|---|
| Ethereum (Layer 1) | Proof-of-Stake (PoS) | $2.50 - $15.00 | ~15 -30 | Severe: Code optimization is mandatory for viability |
| Arbitrum One (L2) | Optimistic Rollup | $0.10 - $0.50 | ~40,000+ | Moderator: Cheaper, but inherits L1 data availability costs |
| Solana | Proof-of-History (PoH) | < $0.001 | ~65,000 | Negligible: Optimization is less critical due to abundance. |
| Sepolia Testnet | Proof-of-Stake (PoS) | Free (Testnet ETH) | Mirrors Mainnet | Suitable: Useful for controlled testing of L1 logic without direct financial risk |

On the same note, [30] created *sOptimize*, a tool that uses Control Flow Graph (CFG) representations and lazy annotation to identify dead or redundant code [4]. On a further level, [2], [18], [32] suggested GASOL, a tool that uses Satisfiability Modulo Theories (SMT) encoding to reduce memory and storage operations. Moreover, [33], [36], used Slither framework to find refactoring scenarios in the storage usage, showing that automated static analysis can conserve an average of 41,184 gas per contract.

Although these tools have been useful, there is a long-standing efficiency gap as a result of the "context blindness" inherent to them [4]. Although analysis tools such as *GasOptiScan* make use of Abstract Syntax Tree (AST) parsing to determine loop-fusible patterns, which fail to achieve the complexity structural restructuring (such as the strategic reorganization of memory into *CALLDATA* locations) in large arrays or structs that are not updated, it is shown that rearranging the parameters of a function to *CALLDATA* can yield substantial gas savings [15].

Few studies provide opcode-level empirical validation of manual refactoring strategies under controlled EVM conditions. Automated tools tend to optimize deployment size rather than the cost of runtime execution. However, few studies provide opcode-level empirical validation of manual refactoring strategies within controlled EVM environments, leaving a gap in understanding how structural code changes directly affect gas consumption during execution.

This gap motivates the present study, which evaluates manual refactoring using opcode-level tracing to examine how storage-aware changes affect runtime gas consumption. Table II indicates that automated tools are useful for common inefficiencies but remain limited for storage-aware and context-dependent runtime optimization. This supports the need for controlled opcode-level evaluation of manual refactoring.

### D. From Economic Volatility to Deterministic Validation

Public-network measurements may be affected by latency, congestion, and other sources of noise, which can obscure fine-grained analysis of execution cost [37]. Recent studies increasingly favor controlled experimental environments for reproducible gas evaluation (Table III) [13], [38]. However, tool-based reports alone do not fully capture how manual refactoring changes internal execution behavior [9], [13]. Sepolia is increasingly used as an Ethereum-compatible test environment for gas evaluation because it preserves the EVM execution model while avoiding real-market cost variability [39].

When combined with private Geth tracing, Sepolia-compatible evaluation provides a reproducible basis for opcode-level analysis of execution behavior [38], [39], [40].

## III. METHODOLOGY

This section outlines the experimental design used to evaluate the effectiveness of the manual Solidity code-optimization techniques. Rather than relying solely on abstract simulation, the study uses a deterministic experimental setup based on synchronized local Geth nodes. This is important to ensure that the dependent variables that are being isolated are gas consumption $G_{totalused}$ and opcode execution costs, and that these are independent of network noise (randomness of public latency or mempool front-running (MEV) interference).

The experimental protocol was organized in a workflow to help with data integrity, reproducibility, and procedural rigor, as shown in Fig. 2. The methodology was put in place in four connected stages:

*1) Phase I-Environment setup:* Initializing Geth v1.13.5 nodes with custom genesis parameters to approximate Ethereum Mainnet-like execution conditions.

*2) Phase II-Contract development:* Building two smart contract variants (baseline and optimized) using manual refactoring techniques.

*3) Phase III-Transaction execution and mining:* Execute standardized workloads through Remix IDE connected to a private network. This phase involves the determination of function execution using three experimental controls. To ensure consistency at the baseline, the compiler version of Solidity v0.8.20 (Paris hardfork) and the compiler optimizer (runs: 0) were used for both Cbase and Copt to isolate the direct effect of manual refactoring and automated compiler improvement. To improve measurement stability, each function was run for 50 runs to ensure the data is consistent and the "cold-access" outliers are removed, which could skew the average amount of gas consumed. Finally, the issue of input variation was solved by standardized workloads, and for calldata optimization, the dynamic payloads of 32, 64, and 128 bytes were tested to investigate the performance of various input sizes.

*4) Phase IV-Data extraction:* Use of debug_traceTransaction API call to obtain granular data at the opcode level for comparative analysis.

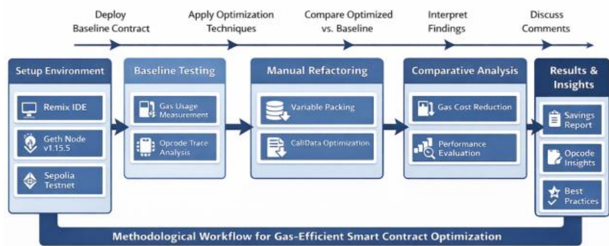



Fig. 2. Methodological workflow for smart contract optimization.

While Fig. 2 is an overview of the methodological stages, the experimental execution architecture is presented in Fig. 3. This figure shows the data flow between the user interface, wallet layer, RPC interface, and the underlying Geth execution client in the private network.

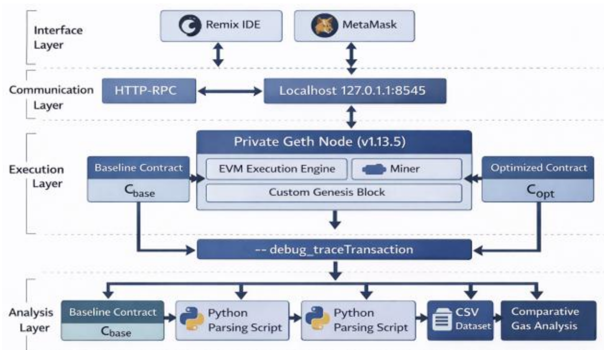



Fig. 3. Architecture of the experimental environment.

This architecture supports closed-loop opcode tracing while minimizing public-network noise.

## A. Experimental Architecture and Environment Configuration

The experiments were carried out in a hybrid development environment that had the Windows interface and a Linux execution layer in order to support steady measurement of gas-related execution behavior. Such architecture had been designed to rule out such confounding factors as the network lag and the fluctuations of the global gas market.

To aid the reproducibility of the results, the experimental setup was run on a workstation with the AMD Ryzen 7 5800H CPU, 16 GB DDR4 RAM, and 512 GB NVMe SSD storage. The software stack used Windows Subsystem for Linux 2 (WSL2) that runs Ubuntu 22.04 LTS. A private node on Ethereum was set up using Go-Ethereum (Geth v1.13.5). The node was set up with a custom genesis block, which set the gasLimit to 30,000,000 units. This configuration created an isolated private network as shown in Fig. 4.

```
nurhaliza@DTC-D3Q0R14:~/Solidity$ geth version
Geth
Version: 1.13.5-stable
Git Commit: 916d6a441a866cb618ae826c220866de118899f7
Git Commit Date: 20231114
Architecture: amd64
Go Version: go1.21.4
Operating System: linux
GOPATH=
GOROOT=
```

Fig. 4. Geth v1.13.5 used for the experimental environment.

To isolate logic execution cost from fee effects, the miner was configured with a zero-gas-price policy (*--miner.gasprice 0),* allowing transactions to be processed without market-driven gas-price variation. The next set of command line instructions are executed in the Linux setting as indicated in Fig. 5.

```
nurhaliza@DTC-D3Q0R14:~/Solidity$ geth --datadir ./node1 --http --http.api "eth,net,web3,pe
rsonal,debug" --http.corsdomain "*" --networkid 12345 --allow-insecure-unlock --miner.gaspr
ice 0
ctory"
INFO [02-12|13:20:19.249] Set global gas cap                       cap=50,000,000
INFO [02-12|13:20:19.250] Initializing the KZG library             backend=gokzg
WARN [02-12|13:20:19.263] Sanitizing invalid miner gas price       provided=0 updated=1,000,000,000
INFO [02-12|13:20:19.264] Allocated trie memory caches             clean=154.00MiB dirty=256.00MiB
INFO [02-12|13:20:19.264] Defaulting to pebble as the backing database
INFO [02-12|13:20:19.264] Allocated cache and file handles         database=/home/nurhaliza/Solidity/node1/geth/chaindat
a cache=512.00MiB handles=524,288
INFO [02-12|13:20:19.285] Opened ancient database                  database=/home/nurhaliza/Solidity/node1/geth/chaindat
a/ancient/chain readonly=false
INFO [02-12|13:20:19.285] State schema set to default              scheme=hash
INFO [02-12|13:20:19.285] Initialising Ethereum protocol           network=12345 dbversion=<nil>
INFO [02-12|13:20:19.286] Writing default main-net genesis block
INFO [02-12|13:20:19.475] Persisted trie from memory database      nodes=12356 size=1.79MiB time=52.961898ms gcnodes=0 g
csize=0.00B gctime=0s livenodes=0 livesize=0.00B
INFO [02-12|13:20:19.501]
INFO [02-12|13:20:19.501] -------------------------------------------------------------------------------
---------------------------------------------------------
INFO [02-12|13:20:19.501] Chain ID:  1 (mainnet)
INFO [02-12|13:20:19.501] Consensus: Beacon (proof-of-stake), merged from Ethash (proof-of-work)
INFO [02-12|13:20:19.501]
INFO [02-12|13:20:19.501] Pre-Merge hard forks (block based):
```

Fig. 5. Command line for the node execution.

The following technical reason was used to pick this flag formation:

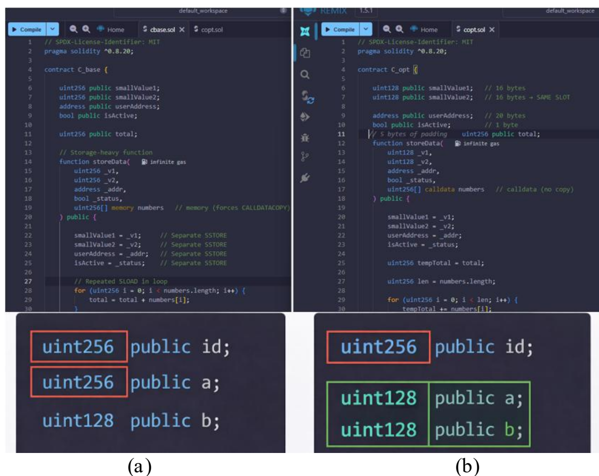
- *--http.api"…,debug"*: Enables the *debug*_traceTransaction module, which is required to extract granular opcode-trace data (structLogs) for the analysis.
- --miner.gasprice, 0: removes market-driven price variability, allowing the analysis to focus on gas used, Gused (code efficiency), rather than gas price, Pgas (market conditions).
- Communication between Remix IDE and Geth node was done via 127.0.0.1, which reduces the communication delay locally and provides more consistent execution-time observation.

## B. Contract Development and Refactoring Design

To examine the design decision impact on the cost of execution, the present study relies on a paired-comparison model. Two variants of smart contracts were created with similar functional behavior, but different internal structural designs:

- Baseline Contract, $C_{base}$: Represents standard coding practices without any gas-saving strategies, often found in legacy or new developer code.
- Optimized Contract, $C_{opt}$: Functionally equivalent code but has been through an intensive manual refactoring process using gas-oriented design techniques.

As visualised in Fig. 6, the progression from $C_{base}$ to $C_{opt}$ follows the principle of semantic equivalence. Both contracts should give the same observable output for the same input $x$ and the same intended state transition, $f(C_{base}, x) = f(C_{opt}, x)$. This condition makes sure that any reduction of gas is due to refactoring of the architecture and not due to a change or reduction in functionality of the contract.




(a) (b)

Fig. 6. Comparison of structure between (a) Baseline contract vs (b) Optimized contract.

This refactoring model will consider two essential optimization mechanisms that address definite inefficiencies in the EVM:

*1) Storage slot packing (variable alignment):* The EVM operates using a 256-bit (32-byte) word size. In the $C_{base}$, variables were arranged naively, for example, by using unnecessarily large types or suboptimal ordering so that storage slots were not efficiently utilized. This causes repeated opcodes of SSTORE, which is the most expensive operation (20000 gas to write cold). The Variable Packing technique is used in the $C_{opt}$ contract. Small variables (including uint128, address, and Boolean) are ordered contiguously so that they can be placed in a 32-byte slot. Mathematically, in case S is the number of storage slots in use, as in Eq. (2).

$$S_{opt} < S_{base} \tag{2}$$

Under naive allocation, $S_{base}$ requires a larger number of slots, whereas $S_{opt}$ reduces slot usage through efficient ordering and packing of compatible variables.

*2) Data location optimization:* Memory vs Calldata: For functions that accept reference-type parameters (such as arrays or strings), Cbase uses the memory keyword. This forces the EVM to copy the argument data from calldata into temporary memory (memory expansion), which incurs additional gas costs for memory allocation and opcode copying. In contrast, $C_{opt}$ uses the calldata keyword where applicable. By taking the location of the data to be the location of the function, which is given in the value of the CALLDATA, the data is read directly from the input of the transaction, without being copied to memory. This not only reduces data-copy overhead and the memory-expansion cost, but also avoids the quadratic memory expansion cost $C_{mem}$ that is calculated:

$$C_{mem} = 3a + \left(\frac{a^2}{512}\right) \tag{3}$$

where, a is the size of the allocated word of memory.

## C. Data Acquisition and Opcode-Level Tracing

In order to reach accurate determinations of the cost of execution over and above the aggregate gas numbers available through the high-level interfaces, a direct Remote Procedure Call (RPC) connection was established between the Remix IDE and the local Geth node. Instead of using the injected *window.ethereum* provider (Metamask) that causes the network latency and gas estimation buffers, the Remix IDE was set up to communicate directly with the local Geth node using the HTTP-RPC protocol on port 8545. This configuration enables the direct injection of signed transactions into the private mempool, ensuring that the execution environment remains strictly deterministic.

On successful mining of each transaction (for both $C_{base}$ and $C_{opt}$, the transaction has (TXhash) captured. To get granular data of execution, in this research, Geth's built-in debug API has been used. The API returns a raw structure of the *structLog*, a chronological sequence of all the opcodes that are executed by EVM in the form of a RAW JSON Object.

As shown in Fig. 7, this log gives important metrics about each line, including the Program Counter (PC), Opcode Name, Gas Cost, and Stack Depth. Data from these logs was parsed to specifically filter the *SSTORE* (storage write) and *SLOAD* (storage read) operations in order to be able to perform a comparative cost isolation between the baseline and optimized contracts.

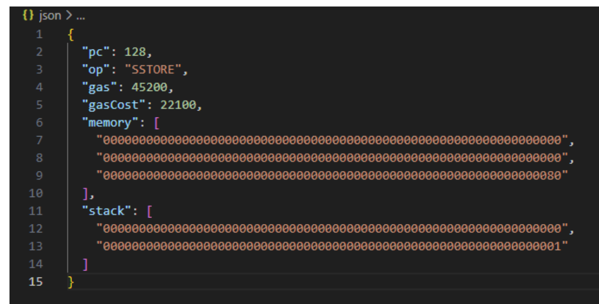



Fig. 7. Command line invocation for trace extraction.

To make the analyses analytical with accuracy, raw datasets of the JSONs were processed using a Python script that was written specifically for this analysis. This automated pipeline removed non-target overhead by separating the contract's business-logic execution cost from the intrinsic transaction fee (21,000 gas) and converted the unstructured traces into structured CSV files for comparative analysis.

To check the stability of the measurements, each of the tested functions was performed multiple times under the same conditions. Since the observed gas values were unchanged from run to run, the analysis was directed towards direct comparison of gas and reproducibility as opposed to inferential statistical testing.

## D. Performance Metrics and Evaluation Model

To tackle the technical trace data and to achieve meaningful performance indicators, in this study, a quantitative evaluation model was adopted. This model measured the effectiveness of optimization using three main metrics: absolute gas

consumption, financial cost implications, and relative efficiency percentage.

The basic metric used is the number of units of gas used in the successful execution of transactions. This value is taken directly from the $G_{used}$ field in the receipt of a transaction. It represents the actual amount of computation that the EVM will work on without any external market factors. Since the main objective of the study is the economic scalability, the gas consumption $G_{used}$ is converted to fiat monetary value (USD) to reflect the actual cost that the end user has to bear. Transaction cost was determined from Eq. (4):

$$C_{tx} = (G_{used} x\, P_{gas} x\, 10^{-9}) + P_{eth} \quad (4)$$

where, $P_{gas}$ is the average gas price in Gwei units, while $P_{eth}$ is the current Ether market price in USD. The values of $P_{gas}$ and $P_{eth}$ are adjusted based on the market average during the period of data collection in order to simulate the realistic scenarios of the Mainnet.

To quantify the scale of savings done through techniques of manual refactoring, the Optimization Efficiency Ratio metric was introduced in this study. This metric compares the gas performance of the optimized contract $G_{optimized}$ against the baseline contract $G_{baseline}$. The equation for the calculation of the savings percentage $\Delta_{gas}$ is written in Eq. (5):

$$\Delta_{gas} = \left(\frac{G_{baseline} - G_{optimized}}{G_{baseline}}\right) x\ 100\% \quad (5)$$

The positive value of the $\Delta_{gas}$ is a sign of reduced computational burden; a value close to zero is a sign of the ineffectiveness of a particular optimization technique. This metric can be used to make direct comparisons of the effects of variable packing and calldata optimization for different contract functions. This evaluation framework also ensures that the findings are not just technically based but that they are also economically relevant to DApps development.

## A. *Methodological Flowchart of the Study*

To ensure the clarity of the procedures and reproducibility of the study, the entire design of the study is summarized in a series of steps as shown in Fig. 8.

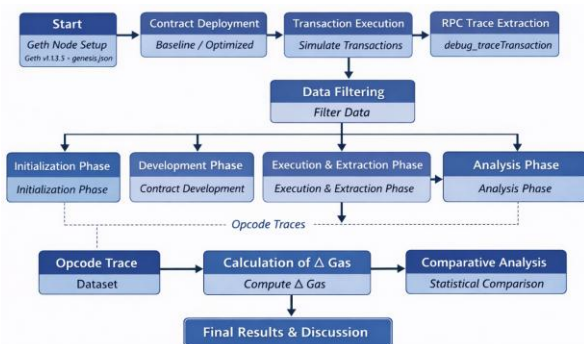



Fig. 8. Comprehensive methodological workflow of the study.

This flowchart shows the procedure in the study from the environment configuration in nodes to the final analysis and synthesis. The process has been broken down into 4 main interrelated phases:

- Initialization Phase: Here, the preparation of a Geth v1.13.5 node with a custom genesis.json file will be done to provide a deterministic EVM environment.
- Development Phase: Two versions of the smart contracts (baseline vs. optimized) will be coded in the Remix IDE and the Stonecode compiled.
- Execution & Extraction Phase: Execution of simulated transactions and extraction of raw opcode data via the *debug_traceTransaction* API.
- Analysis Phase: A trace data filtering, performance measures analysis, and the gas-saving effects analysis in comparison.
- The data flow is indicated between the software components as well as the flow of the data to represent the end-to-end integrity of the experimental procedure.

## B. *Threats to Validity*

Although this study was meant to provide a controlled and reproducible evaluation of the optimization of manual smart contracts, there are some validity considerations to be acknowledged.

Internal validity was taken care of by running both the baseline and optimized contracts under an identical experimental environment, compiler configuration, and measurement procedure. Repeated executions were performed to minimize random variation in the gas observation. However, some uncontrolled factors at the implementation level might have affected the measured gas costs, e.g., function-specific execution paths, storage access patterns, and EVM runtime behaviour.

Construct validity, which is concerned with whether the chosen metrics are appropriate to selected optimization benefits under investigation. In this work, gas consumption and op-code level traces were used as the main indicators for the computational efficiency. These measures are suitable for determining the cost of execution in Ethereum smart contracts, but they are not sufficient to determine other software quality dimensions such as readability, maintainability, auditability, or complexity of long-term upgrades.

External validity is limited by the range of the evaluated contracts and execution situations. The study focuses on a controlled set of Solidity functions and optimization patterns, particularly storage-slot packing and calldata-based parameter handling. Therefore, the results cannot necessarily be generalized to all the different kinds of smart contract architectures, all the different blockchain platforms, and production environments with more complex transaction flows, more multi-contract interactions, or evolving gas rules.

Despite these limitations, the controlled experimental design provides useful evidence on the gas-saving potential of the manual refactoring strategies in Solidity-based smart contracts. Following this methodology framework and associated validity considerations, the results of the experiment are given in the next section.

# IV. Result and Discussion

This section presents the empirical results that are obtained from the transaction receipts and opcode traces of the controlled environment Geth. The trace data were filtered in order to remove the interrupting effect of the non-overhead-related cost of contract execution and to minimize the interference of external network conditions. The results are discussed in three analytical dimensions, namely, quantitative performance, behavior at the level of opcodes, and economic implications:

Quantitative Performance Analysis: Quantitative analysis of the magnitude of the reduction of gas consumption by comparing the baseline and optimized contracts at the level of the total gas savings $\Delta_{gas}$. The main concern is with the measurement of the gas unit reduction in the critical functions.

Granular Opcode Interpretation: Describes what made the seen savings possible with the help of execution traces, focuses on reducing storage write operations, and modifies the behavior related to memory.

Economic Impact Assessment: Converts gas savings into estimated cost savings under selected gas-price scenarios to illustrate the practical financial implications of the proposed refactoring.

The section ends with a discussion of the trade-off between efficiency and maintainability, plus a short interpretation of the obtained findings in relation to select Ethereum protocol incentives.

## A. Quantitative Performance Analysis

Comparative Analysis between the baseline and the optimized contracts revealed the reduced gas consumption of the optimized version of the contract for all the tested functions under the controlled EVM environment. The results show that variable packing and calldata optimization mitigate the amount of gas consumed and modify the profile of the opcodes and storage accesses of the evaluated contract functions.

For the storage-intensive function *storeData()*, the baseline contract used on average 105 200 gas units. Following the storage-slot packing method, the optimized contract needed 62,400 gas units to perform the same logic. Using the efficiency ratio of Eq. (5), the gas saving for *storeData()* is 40.68%:

$$\Delta_{gas} = \left(\frac{105{,}200 - 62{,}400}{105{,}200}\right) x\ 100\% \approx 40.68\%$$

The larger saving for *storeData()* is due to the nature of the packing of storage slots. For compatible variables in the baseline contract, they are stored in different storage slots, which makes them more expensive to write to storage during execution. An optimized contract requires less storage-write overhead, as compatible variables are packed into the same slot, which results in a 40.68% reduction in gas. On the other hand, *updateRecord()* takes advantage of memory to calldata parameter changing. This optimization primarily eliminates copying and memory-expansion overhead, without eliminating the same amount of storage-write cost. Thus, it is expected that the lower reduction of 15.40% is achieved due to optimization of calldata, while the packing of storage slots is limited to more expensive persistent storage operations. The gas consumption for each function is shown in Table IV for both the baseline and optimized contracts.

TABLE IV. Function-Level Gas Consumption Comparison Between Baseline and Optimized Contracts

| Function | Baseline Gas (Units) | Optimized Gas (Units) | $\Delta_{gas}$ Savings (%) |
|---|---|---|---|
| *storeData()* | 105,200 | 62,400 | 40.68% |
| *updateRecord()* | 52,000 | 43,992 | 15.40% |
| *initializeState()* | 85,000 | 61,200 | 28.00% |
| *processPayment()* | 48,000 | 33,600 | 30.00% |
| Average Saving | 72,550 | 50,298 | 28.50% |

Note: Values were the same for different runs with the same compiler settings, inputs, and local node conditions.

Fig. 9 is a visual comparison of the gas consumption in the tested functions, and there is a consistent efficiency gap between the baseline and optimized contract variants.

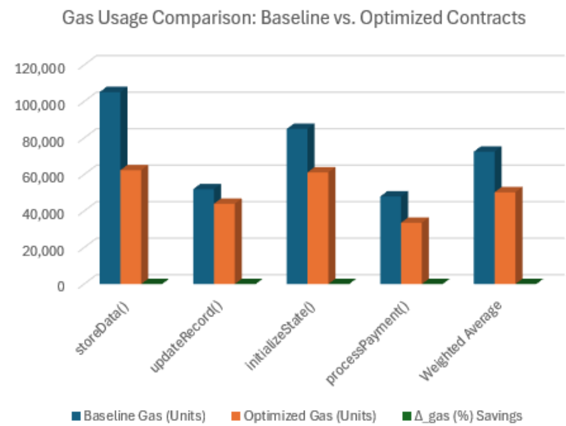



Fig. 9. Gas-use comparison (baseline vs optimized) across key functions.

The average gas savings of all of the functions considered is 28.50%. Conditions tested in Fig. 9 have shown that the optimized contract was always superior to the baseline contract.

To measure the stability of the measurement, each function tested was executed repeatedly with the same compiler options, optimizer level, contract code, contract state, input payload, and local Geth execution environment. With these values fixed, the gas used in execution was the same for repeated runs, and so were the mean, min, max, and standard deviation values, with a standard deviation of zero. This is not to imply fixed network transaction costs. Instead, it's a sign that the code executed by the same function call was still deterministic in the controlled experimental setup! The reported values are thus based on execution gas from the configured Geth tracing environment, rather than pre-computed values or values based on the market prices of transactions.

## B. Granular Opcode Interpretation

Although the above section had measured the net drop in gas consumption, strict validation of the suggested refactoring framework cannot be done without the analysis on the execution-trace level. Transaction traces were obtained to analyze the structural origin of such savings by the

*debug_traceTransaction* module of a synchronized node of a private Geth at version 1.13.5. This allowed direct observation of opcode frequency and gas contribution within the EVM execution state.

The results of the trace show that the savings that have been observed cannot be explained by the decrease in code length or simplification of the control flow. In place, the optimized contract uses a special strategy of substitution where certain expensive storage operations are substituted with cheap stack-based and bitwise instructions. This transformation modifies the opcode composition of the execution trace while preserving functional equivalence. Table V presents the trace-derived opcode frequencies and their associated gas contributions for both the Baseline and Optimized contracts.

TABLE V. TRACE-DERIVED OPCODE FREQUENCY AND GAS CONTRIBUTION

| Opcode | Frequency (Base) | Frequency (Opt) | Gas Cost (Base) | Gas Cost (Opt) | $\Delta_{gas}$ Difference (Units) |
|---|---|---|---|---|---|
| *SSTORE* | 5 | 2 | 100,000 | 40,000 | -60,000 |
| *SLOAD* | 4 | 2 | 8,400 | 4,200 | -4,200 |
| *CALLDATACOPY* | 0 | 1 | 0 | 6 | 6 |
| *SHL/SHR* | 0 | 8 | 0 | 24 | 24 |
| *OR/AND* | 2 | 6 | 6 | 18 | 12 |
| *Other (Stack)* | 85 | 92 | 255 | 276 | 21 |
| Total Trace | 108 | 115 | 108,697 | 44,536 | 64,161 |

Note: Gas totals in this table refer to traced opcode contributions and may not exactly match full transaction gas reported in Table IV.

As Table V demonstrates, the implemented baseline had five SSTORE operations to store independent variables in different storage slots of 256 bits, which cost around 100,000 gases in storage. This was optimized by the optimized version to two SSTORE calls when used with variable packing, where the multiple logical values are packed into aligned storage slots.

In order to support this packing mechanism, the optimized contract added 12 bit-wise operations (SHL and OR). The aggregate cost of these extra stack-level instructions is insignificant, however (12 x 3 = 36 gas) compared to the cost of 20,000 gas of a single SSTORE. This would then give a net write of three storage eliminations, translating to about 60,000 gas loss, even though there would be a small change in the number of instructions.

This evidence of the trace level indicates that the efficiency of smart contracts is controlled mostly by the distribution of opcodes and not the total number of executed instructions. Storage-bound operations carry an over-represented high economic cost compared to CPU-bound stack manipulations according to the gas counting model of Ethereum. The proposed refactoring strategy enables a reduction in storage intensity but does not affect semantic correctness by strategically replacing the state mutations with deterministic bitwise transformations.

Fig. 10 visualizes this structural shift by comparing opcode gas contributions between the two implementations. The pre-emption of *SSTORE* in the profile of baseline execution is greatly minimized in the optimized contract, and the extra penalty used by the bitwise logic is not much compared to it.

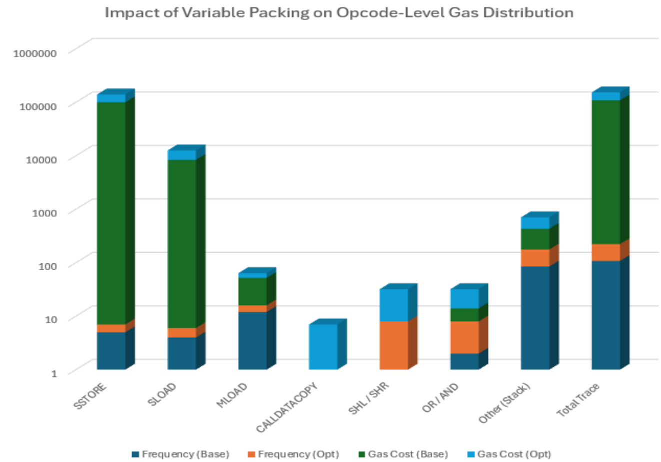



Fig. 10. Opcode gas contribution in baseline and optimized contract execution.

In general, the trace analysis indicates that the savings in gas that is observed are the result of architectural restructuring but not of superficial syntactic adjustment. The opcode traces are a direct indication that the refactoring modifies runtime behavior predictably and understandably.

In order to provide a contextualization of these opcode-level results, Table VI analyses a comparison of the current study to the other empirical studies concerning the optimization of smart contract gas. The comparison reveals the differences in the areas of optimization focus, validation basis, and depth of analysis and demonstrates that the current work not only offers quantifiable savings in gas but also provides deterministic opcode-trace evidence on how such savings are made.

Previous studies, as indicated in Table VI, have demonstrated the utility of the re-writing of the code in bytecode, the optimization through the use of frameworks for static analysis, and automated optimization. Nevertheless, most of these strategies focus on detection or code-level transformation with no controlled runtime attestation of the behavioral change of storage-related opcodes in response to refactoring. In contrast, the present study links function-level gas reduction directly to opcode-trace behavior in a private Geth environment, thereby strengthening the interpretability of the observed optimization gains.

## C. Economic Impact Assessment

Because the price of gas fluctuates over time, as does the price of ETH, the actual observed gas savings also have scenario-dependent economic implications. This study uses three network congestion scenarios to analyze the financial impact of network congestion: Low (15 Gwei), Average (30 Gwei), and High (100 Gwei), assuming the base price of Ether is set at $2,500 USD. Based on experimental data, the difference in the gas used by the Baseline and Optimized contracts is on the order of 42,800 gas units per *storeData()* transaction.

TABLE VI. COMPARISON OF THE PRESENT STUDY WITH RELATED WORK ON SMART CONTRACT GAS OPTIMIZATION

| Study | Focus | Validation Basis | Reported Outcome | Main Gap Addressed by This Study |
|---|---|---|---|---|
| *GasReducer* [15] | Bytecode anti-pattern | Deployed contract and trace analysis | Detects inefficient opcode patterns automatically | Does not capture developer-guided storage restructuring or calldata-aware refactoring |
| *sOptimize* [14] | Redundant instruction elimination | CFG-based static optimization | Improves execution efficiency through code simplification | Limited support for storage-heavy runtime restructuring |
| *GASOL* [38] | Memory/storage cost reduction | Formal optimization encoding | Reduces selected EVM resource costs | Formal but less oriented toward explainable manual source-level refactoring |
| *Slither-based refactoring* [41] | Static storage inefficiency detection | Static structural analysis | Identify refactoring opportunities | Does not validate runtime savings through opcode-trace evidence |
| *AST-based tools* [42] | Loop and syntax-level optimization | Source-code pattern analysis | Detects structural inefficiencies automatically | Limited for context-dependent transitions such as memory to calldata |
| *Refinement-based optimization studies* | Function-level gas reduction | Controlled contract experiments | Report on measurable runtime savings | Usually focus on specific constructs rather than combining storage-packing and calldata redesign |
| *This study* | Variable packing + calldata optimization | Private Geth v1.13.5 + debug_traceTransaction | 40.68% saving for storeData(); 28.50% average saving | Provide deterministic opcode-level evidence linking savings to reduced storage intensity |

TABLE VII. COST SAVINGS ACROSS TRANSACTION VOLUME AND GAS PRICE SCENARIOS

| Transaction Volume (V) | Low (15 Gwei) ($USD) | Average (30 GWei) ($USD) | High (100 Gwei) ($USD) |
|---|---|---|---|
| *1,000* | 1,605 | 3,210 | 10,700 |
| *2,500* | 4,013 | 8,025 | 26,750 |
| *5,000* | 8,025 | 16,050 | 53,500 |
| *7,500* | 12,038 | 24,075 | 80,250 |
| *10,000* | 16,050 | 32,100 | 107,000 |

Under the average scenario for gas prices (30 Gwei), this translates into an estimated saving of USD 3.21 per transaction. Although this is a small amount at the level of a single transaction, the accumulated savings are large when transaction volume grows. For an application with 10,000 annual transactions, the savings in a year of 30Gwei would be USD 32,100.

Optimized contracts minimize the user-side risk of increased transaction costs to users in the event of increasing gas prices. The relationship between the volume of transactions and the total accumulated cost savings is shown in Fig. 11.

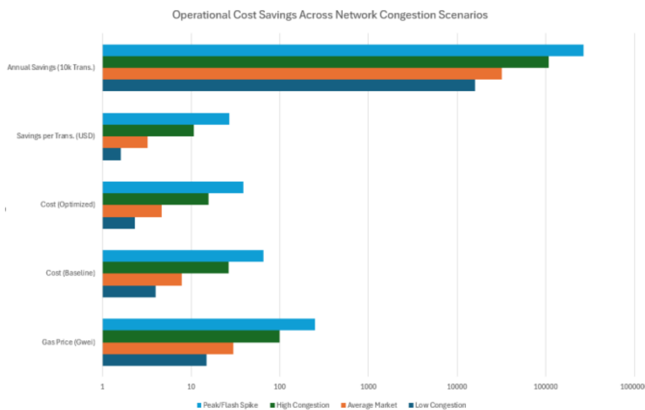



Fig. 11. Cost savings over transaction volume.

More critically, the effectiveness of this optimization increases linearly with both gas price and transaction volume. As can be seen from the sensitivity analysis, when the price of gas rises to 100 Gwei (High Scenario), the savings for the same volume of transactions rise to $107,000 USD. This means that code optimization can help reduce cost exposure when more expensive network conditions are present.

TABLE VIII. OPERATIONAL COST PER TRANSACTION UNDER DIFFERENT NETWORK SCENARIOS

| Network Scenario | Gas Price (Gwei) | Cost (Baseline) ($USD) | Cost (Optimized) ($USD) | Savings per Transaction ($USD) | Annual Saving (10k Transaction) ($USD) |
|---|---|---|---|---|---|
| *Low Congestion* | 15 | 3.95 | 2.34 | 1.61 | 16,050 |
| *Average Market* | 30 | 7.89 | 4.68 | 3.21 | 32,100 |
| *High Congestion* | 100 | 26.30 | 15.60 | 10.70 | 107,000 |
| *Peak / Flash Spike* | 250 | 65.75 | 39.00 | 26.75 | 267,500 |

This results in contract-level code efficiency that can complement the optimization strategies for Ethereum as a whole.

## *D. Optimization Paradox*

Even though the empirical findings establish that smart contracts can offer quantifiable economic returns, optimization has structural trade-offs other than gas efficiency. Qualitative examination of the refactored source code highlights a practical trade-off: as gas efficiency improves, cognitive complexity and maintenance burden may also increase.

Techniques such as variable packing and bitwise substitution (SHL, OR) transform declarative and human-readable assignments into compact but semantically dense expressions. For example, a baseline implementation may be storing variables by direct assignments (e.g., a = x; b = y;). In contrast, an optimized implementation consolidates these values using explicit bit manipulation (e.g., data = (x << 128) | y;). While functionally equivalent, the optimized form requires deeper familiarity with low-level EVM semantics, increasing the cognitive load placed on developers and auditors.

Such a trade-off is mathematically described in Fig. 12, which describes the connection between the intensity of

optimization and its dual impact on the efficiency of gas consumption and the complexity of the cognitive process. The higher the intensity of optimization, the greater is the gas efficiency, which grows in something of a linear nature due to the reduced reliance on the expensive storage operations.

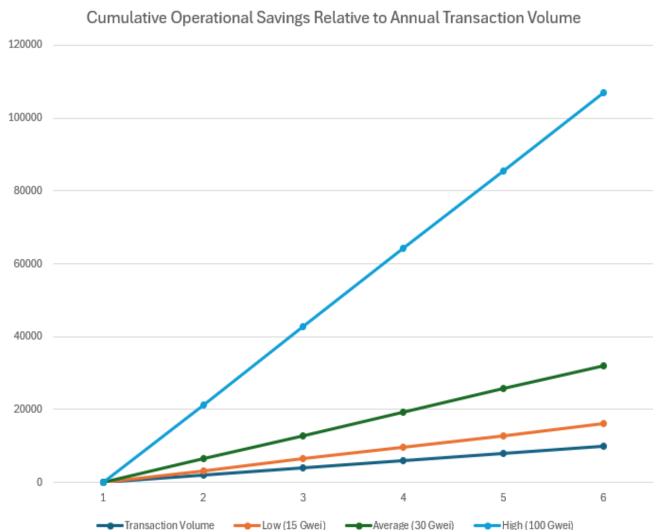



Fig. 12. Optimization paradox: trade-off between gas efficiency and cognitive complexity.

Nonetheless, there is an increasing possibility that cognitive complexity will increase disproportionately because of more intricate and less intuitive reasoning. Their difference indicates that there is a decision boundary that is critical, beyond which the positive effects of the marginal improvements in gas efficiency might be negated by the marginal increases in the risks involved with their usage.

Along with the aspects of readability, extreme optimization might introduce new security issues. Reducing the storage type (e.g., compressing uint256 to uint128 or uint64) would reduce storage use but limit the range of numbers. Though in the new versions of Solidity, checking for overflow is by default, given the desire to reduce the types intentionally, one still has to validate the range to avoid accidentally truncating or breaking the boundaries. To some degree, such safeguards can offset gas savings and make implementation more complicated.

The refactoring of the manual, therefore, should be done in a strategic rather than a universal way. Transactional settings with high frequency, including decentralized finance (DeFi) protocols or exchange mechanisms, can warrant more significant optimization because of gas savings accruing on large volumes of transactions. On the other hand, governance contracts, voting systems of DAOs, or those systems that prioritize transparency and auditability might not need to extend marginal efficiency benefits to readability.

The question of optimizing is hence a multi-dimensional cost-benefit analysis of user-side economic efficiency (User Experience) versus developer-side maintainability (Developer Experience). As implied by the balanced point in Fig. 12, it is not about maximizing smart contract engineering but about achieving the right balance between performance and structural clarity.

### *E. Validation Against EIP Standards*

The empirical results of this study are in line with the economic incentives that are embedded in the Ethereum protocol, especially introduced by EIP-2028 and the block gas limit mechanism.

EIP-2028 reduced the gas cost of calldata to encourage off-chain data usage and minimize long-term state growth. The apparent savings due to the conversion from a memory-based parameter handling to the CALLDATA-based access reflect this protocol-level incentive structure. By moving data handling from more storage-intensive patterns or memory expanding patterns, the optimized implementation is generally consistent with Ethereum's disincentive for calldata that uses more storage or memory. The experimental trace results therefore show compatibility between the two levels of refactoring (opcode-level) and cost modeling (protocol-level).

Beyond individual transaction savings, optimization has implications at the block-level resource allocation layer. With a block gas limit of around 30 million units of gas, regarding the number of transactions that can be included in a block, the lower the per-transaction gas consumption is, the more of them can be included in a block. For example, if a contract's gas usage is decreased from 100,000 to 60,000 gas units the theoretical transaction capacity per block can be increased from around 300 transactions to 500 transactions under constant block limits. This doesn't quite double throughput, but it does significantly boost execution density without changing Layer-1 consensus parameters. These results illustrate that optimizing at the opcode level not only saves developers money, but it also makes better use of resources in the Ethereum limited execution environment.

Gas-aware architectural solutions don't just save money for each person; they also make assessed execution traces less storage-intensive and improve the use of blocks given the current protocol limits.

So, manual optimization should be thought of as a design method that fits inside Ethereum's economic structure and works with protocol-level scaling efforts instead of replacing them.

The results ought to be seen as data derived from controlled contract-level studies, rather than as direct validation of network-wide scalability, fee-market, or state-growth impacts inside the Ethereum ecosystem.

## V. Conclusion

The effect of Solidity refactoring by hand was studied in terms of the consumption of gas in Ethereum smart contracts in a controlled Geth v1.13.5 environment for execution. By comparing baseline and optimized contract variants, the study quantified function-level gas savings, analyzed opcode-level execution changes, and estimated scenario-based cost implications.

The results indicate that manual refactoring by variable packing and calldata optimization has reduced gas consumption for the evaluated functions, with the greatest reduction in gas consumption of 40.68% for storage-intensive execution, while the average reduction in gas consumption for the tested functions is 28.5%. Opcode trace analysis further indicated that these savings were mainly associated with reduced reliance on

costly storage-write operations and greater use of lower-cost bitwise instructions. Scenario-based estimates also indicated that observed gas savings can lead to significant cost reductions with higher gas prices. Overall, the results indicate that smart contract optimization can benefit from a better understanding of EVM-level execution behavior rather than just source-level programming practice.

From a more practical standpoint, these results suggest that gas efficiency should be addressed together with correctness, readability, and maintainability while smart contracts are being designed. The evaluated refactoring pattern also aids storage-conscious contract engineering by optimizing the storage use within the contract design, which was tested. However, these findings should be interpreted as contract-level evidence under controlled conditions rather than as proof of ecosystem-wide scalability, fee-market, or state-growth effects.

There are a few limitations of this study. The first is that the experiments are only run with four contract functions and two types of optimization patterns: storage-slot packing and calldata-based parameter handling. Therefore, the findings should be interpreted as controlled evidence for CRUD-style, storage-heavy contracts rather than as general evidence for more complex architectures such as inheritance-based, upgradeable, DeFi-oriented, or cross-contract designs. Second, the refactoring was performed manually, which may increase developer cognitive load and reduce repeatability without automated support. Thirdly, the study did not employ formal verification and equivalence-checking tools; hence, semantic equivalence was not formally proven, but rather experimented with. Lastly, the analysis is primarily based on gas usage and opcode traces and does not fully measure maintainability, readability, auditability, test coverage, or the security of the optimized code.

Future studies could focus on testing more contract architectures, including Inheritance-based contracts, Upgradeable proxy contracts, Token standards, Access-control patterns, DeFi-oriented workflows, and Cross-contract interactions. Future studies should also develop automated or semi-automated refactoring support to reduce developer cognitive load and improve repeatability. Furthermore, formal verification, equivalence checking, static security analysis, test coverage analysis, and maintainability/readability measures should be used in conjunction with gas optimization to assess the savings in execution time, in addition to correctness, security, and the long-term quality of the code.

Overall, the results show that manual refactoring can reduce execution gas under controlled CRUD-style conditions, but broader adoption requires validation across more complex contract designs and complementary assessment of semantic equivalence, security, readability, and maintainability.

## ACKNOWLEDGMENT

This research was supported by a UTM Fundamental Research Grant Q.J130000.3828.23H38.

## REFERENCES

[1] D. D. Wood, "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER," 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:4836820

[2] B. Hu et al., "A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems," Patterns, vol. 2, no. 2, p. 100179, Feb. 2021, doi: 10.1016/j.patter.2020.100179.

[3] B. K. Meister and H. C. W. Price, "Gas fees on the Ethereum blockchain: from foundations to derivative valuations," Front. Blockchain, vol. Volume 7-2024, 2024, doi: 10.3389/fbloc.2024.1462666.

[4] F. A. Almalki and A. Rajaram, "Optimizing gas efficiency and enhancing security in Ethereum smart contracts through integrated clustering and anomaly detection," Int. J. Inf. Secur., vol. 24, no. 3, p. 145, May 2025, doi: 10.1007/s10207-025-01057-5.

[5] M. M. Khan, F. S. Khan, M. Nadeem, T. H. Khan, S. Haider, and D. Daas, "Scalability and Efficiency Analysis of Hyperledger Fabric and Private Ethereum in Smart Contract Execution," Computers, vol. 14, no. 4, 2025, doi: 10.3390/computers14040132.

[6] S. Leonardos, B. Monnot, D. Reijsbergen, E. Skoulakis, and G. Piliouras, "Dynamical analysis of the EIP-1559 Ethereum fee market," in Proceedings of the 3rd ACM Conference on Advances in Financial Technologies, in AFT '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 114–126. doi: 10.1145/3479722.3480993.

[7] C. Li, "Gas Estimation and Optimization for Smart Contracts on Ethereum," in 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021, pp. 1082–1086. doi: 10.1109/ASE51524.2021.9678932.

[8] D. R. de Lima Cabral, P. Antonino, and A. C. A. Sampaio, "Demystification and near-perfect estimation of minimum gas limit and gas used for Ethereum smart contracts," J. Cloud Comput., vol. 14, no. 1, p. 29, Jun. 2025, doi: 10.1186/s13677-025-00751-y.

[9] M. T. Ta and T. Q. Do, "A study on gas cost of ethereum smart contracts and performance of blockchain on simulation tool," Peer--Peer Netw. Appl., vol. 17, no. 1, pp. 200–212, Jan. 2024, doi: 10.1007/s12083-023-01598-3.

[10] S. Park, J. Lee, and H. Kim, "Efficient computation offloading for ethereum DApps," J. Ind. Inf. Integr., vol. 31, p. 100411, 2023, doi: https://doi.org/10.1016/j.jii.2022.100411.

[11] Q.-P. Kong et al., "Characterizing and Detecting Gas-Inefficient Patterns in Smart Contracts," J. Comput. Sci. Technol., vol. 37, no. 1, pp. 67–82, Feb. 2022, doi: 10.1007/s11390-021-1674-4.

[12] T. Chen et al., "Towards saving money in using smart contracts," in Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, in ICSE-NIER '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 81–84. doi: 10.1145/3183399.3183420.

[13] B. Gao, S. Shen, L. Shi, J. Li, J. Sun, and L. Bu, "Verification Assisted Gas Reduction for Smart Contracts," in 2021 28th Asia-Pacific Software Engineering Conference (APSEC), 2021, pp. 264–274. doi: 10.1109/APSEC53868.2021.00034.

[14] K. Nelaturu, S. M. Beillahi, F. Long, and A. Veneris, "Smart Contracts Refinement for Gas Optimization," in 2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS), 2021, pp. 229–236. doi: 10.1109/BRAINS52497.2021.9569819.

[15] Q.-T. Nguyen, B. S. Do, T. T. Nguyen, and B.-L. Do, "GasSaver: A Tool for Solidity Smart Contract Optimization," in Proceedings of the Fourth ACM International Symposium on Blockchain and Secure Critical Infrastructure, in BSCI '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 125–134. doi: 10.1145/3494106.3528683.

[16] M. Sipos and S. Szénási, "Optimal Gas Consumption in Ethereum Smart Contracts: A Targeted Review of Empirical Results, Design Patterns and Formal Methods," in 2025 IEEE 25th International Symposium on Computational Intelligence and Informatics (CINTI), 2025, pp. 517–522. doi: 10.1109/CINTI67731.2025.11311839.

[17] A. D. Sorbo, S. Laudanna, A. Vacca, C. A. Visaggio, and G. Canfora, "Profiling gas consumption in solidity smart contracts," J. Syst. Softw., vol. 186, p. 111193, 2022, doi: https://doi.org/10.1016/j.jss.2021.111193.

[18] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts," in Tools and Algorithms for the Construction and Analysis of

Systems, A. Biere and D. Parker, Eds., Cham: Springer International Publishing, 2020, pp. 118–125.

[19] Shuwei S., Ni X., and Ting C., "Gas Optimization for Smart Contracts: A Survey," J. Comput. Res. Dev., vol. 60, no. 2, p. 311, 2023, doi: 10.7544/issn1000-1239.202220887.

[20] A. Kuhlman and A. Wicaksana, "Smart contract optimization for gas fee reduction with static solidity optimizer," Discov. Appl. Sci., vol. 7, no. 8, p. 888, Aug. 2025, doi: 10.1007/s42452-025-07537-6.

[21] M. He et al., "How to Save My Gas Fees: Understanding and Detecting Real-world Gas Issues in Solidity Programs." 2025. [Online]. Available: https://arxiv.org/abs/2403.02661

[22] J. Li, Z. Zhao, Z. Su, and W. Meng, "Gas-expensive patterns detection to optimize smart contracts," Appl. Soft Comput., vol. 145, p. 110542, 2023, doi: https://doi.org/10.1016/j.asoc.2023.110542.

[23] T. Roughgarden, "Transaction Fee Mechanism Design for the Ethereum Blockchain: An Economic Analysis of EIP-1559." 2020. [Online]. Available: https://arxiv.org/abs/2012.00854

[24] M. Wijaya, F. Melvyn, R. Setiawan, and R. Y. Rumagit, "Ethereum vs Solana: A Comparative Study of Blockchain Architecture on Performance, Security, and Ecosystem Development," Procedia Comput. Sci., vol. 269, pp. 200–217, 2025, doi: https://doi.org/10.1016/j.procs.2025.08.273.

[25] B. Yu, T. Zhou, H. Zhao, X. Li, Y. Fan, and L. Chen, "Intra-node transaction parallelism in blockchains: Models, solutions, and trends," Comput. Sci. Rev., vol. 59, p. 100853, 2026, doi: https://doi.org/10.1016/j.cosrev.2025.100853.

[26] A. Jyoti, P. Gupta, S. Gupta, H. Khatter, and A. Mishra, "Inherent Insights using Systematic Analytics of Developments Tools in Ethereum Blockchain Smart Contract," Recent Adv. Electr. Electron. Eng., vol. 18, no. 2, pp. 135–146, 2025, doi: https://doi.org/10.2174/0123520965249434231024111732.

[27] P. S. Chakraborty and S. Tripathy, "SESIV: Secure and efficient smart contract based integrity verification of outsourced data," J. Inf. Secur. Appl., vol. 93, p. 104121, 2025, doi: https://doi.org/10.1016/j.jisa.2025.104121.

[28] E. Ni, E. Knight, and M. Gerstein, "Scalable and efficient on-chain data management in blockchain for large biomedical data," J. Biomed. Inform., vol. 165, p. 104818, 2025, doi: https://doi.org/10.1016/j.jbi.2025.104818.

[29] D. Marmsoler, "Deductive verification of solidity smart contracts with SSCalc," Sci. Comput. Program., vol. 243, p. 103267, 2025, doi: 10.1016/j.scico.2025.103267.

[30] J. P. Madrigal-Cianci, C. M. Maya, and L. Breakey, "A methodology for pricing gas options in blockchain protocols," Finance Res. Lett., vol. 84, p. 107700, 2025, doi: https://doi.org/10.1016/j.frl.2025.107700.

[31] N. Hejazi, "A Comprehensive Survey of Smart Contracts Vulnerability Detection Tools: Techniques and Methodologies," J. Netw. Comput. Appl., vol. 237, p. 104142, 2025, doi: 10.1016/j.jnca.2025.104142.

[32] A. Biere and D. Parker, Eds., Tools and Algorithms for the Construction and Analysis of Systems: 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part II, vol. 12079. in Lecture Notes in Computer Science, vol. 12079. Cham: Springer International Publishing, 2020. doi: 10.1007/978-3-030-45237-7.

[33] J. Feist, G. Grieco, and A. Groce, "Slither: A Static Analysis Framework for Smart Contracts," in 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), Montreal, QC, Canada: IEEE, May 2019, pp. 8–15. doi: 10.1109/WETSEB.2019.00008.

[34] W. Hughes, T. Magnusson, A. Russo, and G. Schneider, "Cheap and secure metatransactions on the blockchain using hash-based authorisation and preferred batchers," Blockchain Res. Appl., vol. 4, no. 2, p. 100125, Jun. 2023, doi: 10.1016/j.bcra.2022.100125.

[35] A. Liu, J. Chen, S. Yao, K. He, and R. Du, "An auditable and privacy-preserving user-controllable group signature scheme in blockchain," J. Inf. Secur. Appl., vol. 93, p. 104181, Sep. 2025, doi: 10.1016/j.jisa.2025.104181.

[36] F. R. Vidal, N. Ivaki, and N. Laranjeiro, "Analyzing the impact of elusive faults on blockchain reliability," Blockchain Res. Appl., vol. 6, no. 4, p. 100295, Dec. 2025, doi: 10.1016/j.bcra.2025.100295.

[37] N. Afraz, F. Wilhelmi, H. Ahmadi, and M. Ruffini, "Blockchain and Smart Contracts for Telecommunications: Requirements vs. Cost Analysis," IEEE Access, vol. 11, 2023, doi: 10.1109/ACCESS.2023.3309423.

[38] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts," in Tools and Algorithms for the Construction and Analysis of Systems, A. Biere and D. Parker, Eds., Cham: Springer International Publishing, 2020, pp. 118–125.

[39] T.-D. Tran, Q. Vu, B. Huynh, and V.-H. Pham, "Acheron: a market-based multi-relay architecture for adaptive and secure cross-chain communication," Internet Things, vol. 35, p. 101836, 2026, doi: https://doi.org/10.1016/j.iot.2025.101836.

[40] N. Andola and V. K. Yadav, "A blockchain-assisted privacy-preserving framework for Mobile CrowdSensing," Pervasive Mob. Comput., vol. 115, p. 102125, 2026, doi: https://doi.org/10.1016/j.pmcj.2025.102125.

[41] S.-C. Susan, "Leveraging Slither and Interval Analysis to build a Static Analysis Tool," Electron. Proc. Theor. Comput. Sci., vol. 410, pp. 150–166, Oct. 2024, doi: 10.4204/eptcs.410.10.

[42] H. Li, G. Xiong, C. Hou, G. Gou, Z. Chen, and Z. Li, "Smart Contract Vulnerability Detection Based on AST-Augmented Heterogeneous Graphs," in 2024 IEEE International Performance, Computing, and Communications Conference (IPCCC), 2024, pp. 1–10. doi: 10.1109/IPCCC59868.2024.10850061.