Preprints.org

Article

Not peer-reviewed version

An Automated Architecture for Smart Contract Testing: A Multi-Objective CI/CD Pipeline Optimized for Speed and Gas Efficiency

Manikanta Reddy P *

Posted Date: 10 April 2026

doi: 10.20944/preprints202604.0768.v1

Keywords: smart contracts; blockchain; DevSecOps; CI/CD; multi-objective optimization; test suite reduction; gas efficiency; layer 2 rollups

Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This open access article is published under a Creative Commons CC BY 4.0 license, which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 10 April 2026 doi:10.20944/preprints202604.0768.v1

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

An Automated Architecture for Smart Contract Testing: A Multi-Objective CI/CD Pipeline Optimized for Speed and Gas Efficiency

Manikanta Reddy P

Department of Computer Science, VIT-AP University, Amaravathi, India; mani.23mis7025@vitapstudent.ac.in

Abstract

Immutable code and steep transaction fees make smart contract deployment uniquely unforgiving. While continuous integration (CI/CD) pipelines excel at catching standard software bugs, applying exhaustive security tests to Web3 applications severely bottlenecks development through massive computational overhead and gas consumption. This paper presents a testing architecture designed specifically to resolve this tension between security depth and execution speed. The system pipelines three core engines. First, an AI-driven pre-execution gate flags immediate vulnerabilities. Next, a structural reduction module applies the $k + 1$ symmetric pattern to strip out redundant test permutations. Finally, the system constrains the remaining test suite using the NSGA-II evolutionary algorithm. This multi-objective optimizer dynamically schedules execution to maximize fault detection against strict, predefined gas budgets. To evaluate the model empirically, I bridged a localized EVM sandbox with a Python optimization engine. Results confirm the framework collapses exponential test generation and throttles execution costs without sacrificing critical security coverage. Ultimately, it offers a highly scalable path forward for modern DevSecOps.

Keywords: smart contracts; blockchain; DevSecOps; CI/CD; multi-objective optimization; test suite reduction; gas efficiency; layer 2 rollups

---

1. Introduction

Blockchain technology has enabled the development of decentralized applications that operate without centralized control, holding the potential to revolutionize the industry [1]. Smart contracts serve as the engine for these platforms. By enforcing agreements strictly through code, they automate financial transactions on networks like Ethereum. But this automation carries a severe risk: correctness is absolute. Unlike traditional software where patches can easily roll back a database error, smart contract vulnerabilities often result in immediate, irreversible financial loss.

The industry has seen the fallout of poor testing firsthand. Exploits targeting reentrancy flaws, access control gaps, and DeFi logic errors have drained millions. These breaches prove that developers need robust testing methodologies capable of flagging flaws long before a contract hits the mainnet.

Traditional testing simply generates massive test suites to chase coverage metrics. While this works in standard software, applying exhaustive execution inside a continuous integration pipeline creates unacceptable overhead. Add in the reality of blockchain gas costs, and the testing process becomes both incredibly slow and prohibitively expensive.

I built this framework to directly tackle that bottleneck. By combining automated vulnerability detection, structural test suite reduction, and multi-objective optimization, the architecture balances rigorous security with execution speed. Developers get the safety of deep vulnerability scanning without the crushing computational cost.

This paper presents four main contributions. First, I outline a modular testing architecture that slots directly into existing CI/CD pipelines. Second, the framework drastically cuts redundant test

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 10 April 2026 doi:10.20944/preprints202604.0768.v1

combinations by applying the $k+1$ symmetric test pattern. Third, this research maps out a multi-objective optimization model. By feeding execution data into the NSGA-II algorithm, the system optimizes for fault detection, speed, and gas simultaneously. Finally, I validate this entire pipeline through a functional prototype, proving it cuts testing overhead while strictly obeying economic deployment constraints.

2. Background and Related Work

Integrating security directly into Web3 CI/CD pipelines is a highly active research area. Because smart contracts become immutable the moment they are deployed, standard software testing falls short. Recent work aggressively pushes DevSecOps principles into blockchain environments, yet balancing execution speed against cost remains a stubborn hurdle.

2.1. The Transition to Smart Contract DevSecOps

The necessity of applying structured DevOps routines to smart contracts is no longer up for debate. Reyes, Jimeno, and Villanueva-Polanco (2023) mapped out exactly how traditional DevOps phases align with the smart contract lifecycle [2]. Their conceptual framework showed that automation servers can successfully trigger static and dynamic security checks before deployment [2]. The problem? Most early frameworks are heavily theoretical [2]. They demand comprehensive testing but fail to explain how to optimize the actual execution, leaving agile teams without practical, high-speed implementations.

2.2. Security vs. Performance Trade-offs in CI/CD

Moving from theory to practice usually breaks the speed metrics. Chattopadhyay (2025) designed a Zero-Trust DevSecOps pipeline using AI anomaly detection and immutable audit logs [3]. It detected threats brilliantly. However, the computational overhead inflated CI/CD execution times by roughly 15% [3].

Saleh, Madhavji, and Steinbacher (2024) hit a similar wall. They built a working prototype that linked a Jenkins CI/CD pipeline directly to a permissioned blockchain [4]. The system successfully halted deployments when it found vulnerabilities [4]. But the deep analysis took too long, crippling the rapid iteration cycles that agile environments demand [4].

2.3. The Gap: Economic and Execution Optimization

The literature makes one thing clear: automated deep security testing works, but the latency bottlenecks are severe. Even advanced AI-driven DevSecOps models admit that full EVM test coverage triggers "increased gas consumption and slower test execution times," blowing up testing budgets [5]. The field simply lacks research on DevSecOps architectures optimized specifically for both pipeline speed and blockchain economics.

3. Threat Modeling and DeFi Attack Vectors

Understanding the evolving Web3 threat landscape is essential to justify this dynamic execution layer. Traditional testing isolates individual functions. Decentralized applications, however, are highly composable; they talk to multiple external protocols at the same time [17]. This composability creates dangerous economic vulnerabilities that standard static analysis completely misses.

3.1. Traditional Smart Contract Vulnerabilities

At a basic level, smart contracts suffer from EVM-specific coding flaws. Jiang et al. categorize these into Exception Disorders, Timestamp Dependencies, Freezing Ether, and Dangerous `DelegateCall` implementations [15]. The most destructive of these foundational flaws is Reentrancy, the bug behind the infamous $50 million DAO hack [16]. Reentrancy happens when a vulnerable contract calls out to an attacker's contract. The attacker then uses a fallback function to recursively drain the victim before the original execution state ever updates [16]. Because these flaws are deterministic, they are the exact targets the AI-Driven Security Gate (Module 1) is built to catch.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 10 April 2026 doi:10.20944/preprints202604.0768.v1

3.2. The Flash Loan Mechanism

Today, the most prominent threat vector in Web3 is the Flash Loan attack. These are massive, uncollateralized loans borrowed and repaid entirely within a single, atomic transaction [17]. If the borrower doesn't repay the loan in time, the EVM simply reverts the state, erasing the transaction as if it never happened [17,18]. While built for healthy arbitrage, attackers now use flash loans to borrow tens of millions of dollars, exploiting protocol vulnerabilities with absolutely zero personal capital at risk [18].

3.3. Price Manipulation and AMM Exploits

Many of these attacks target Automated Market Makers (AMMs) like Uniswap [17,19]. Attackers use flash loans to dump massive liquidity into a pool, breaking the mathematical formulas to artificially skew token exchange rates [19]. Look at the 2020 Cheese Bank hack. The attacker borrowed 21,000 ETH to artificially pump the CHEESE token's price on an AMM [19]. Because Cheese Bank relied on that exact AMM as a price oracle, the attacker used their suddenly "valuable" CHEESE tokens as collateral to steal $3.3 million in stablecoins [19].

3.4. Non-Price Flash Loan Attacks

Adversaries don't just manipulate prices; they increasingly fund zero-day logic exploits with flash loans [18]. They use the borrowed liquidity to force their way past logical preconditions or corrupt contract states [18]. In the Hedgey Finance exploit, the attacker used a flash loan to lock a campaign and gain smart contract token approval [18]. They immediately canceled it for a refund. The contract had a logic flaw and forgot to revoke the token allowance [18]. The attacker then walked away with $2 million [18].

3.5. Implications for CI/CD Testing

Isolated unit tests cannot stop multi-contract attack vectors. To actually secure DApps, CI/CD pipelines have to run dynamic, inter-contract scenarios that simulate flash loan liquidity and AMM state corruption [17,19]. But spinning up multiple contracts to execute heavy EVM state transitions destroys gas and time budgets. This exact problem necessitates the pattern reduction and gas-aware optimization outlined in Modules 2 and 3.

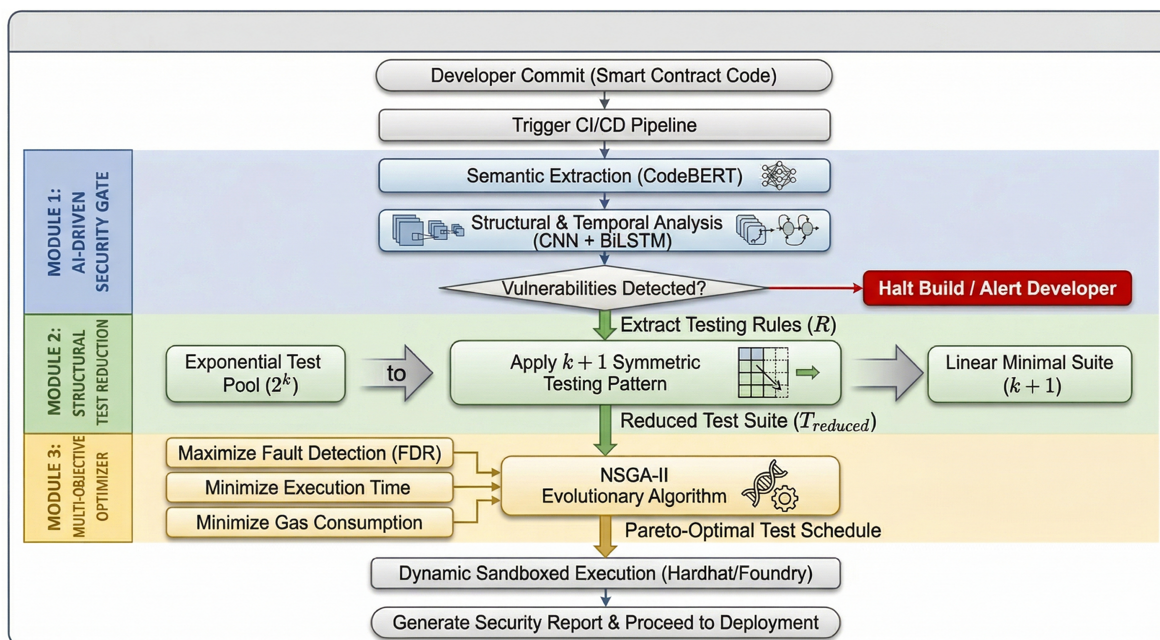

Figure 1. End-to-end workflow of the proposed smart contract testing framework. The pipeline integrates AI-based vulnerability detection, test suite reduction using the $k+1$ symmetric testing strategy, and multi-objective optimization via the NSGA-II algorithm to improve testing efficiency in CI/CD environments.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 10 April 2026 doi:10.20944/preprints202604.0768.v1

4. Architecture of the Proposed Framework

I propose a Multi-Objective Smart Contract Testing Architecture to fix the standoff between security depth and execution speed. Sitting directly inside a version control pipeline as an automated gatekeeper, it relies on three sequential modules.

4.1. Data Flow Between Framework Components

The architecture functions as a strict sequence. First, the pipeline feeds the smart contract source code into the AI vulnerability module. This engine grades the contract's vulnerability likelihood and spits out specific security rules that require testing.

Those rules drop into the test generation component. It builds an initial pool of candidate test cases to cover the contract's possible behaviors. Because this pool scales exponentially, the reduction engine steps in. It applies the $k + 1$ symmetric pattern to gut redundant combinations while keeping the critical rule coverage intact.

This slimmed-down test suite is then pushed to the optimization module. Each test is tagged with its estimated fault detection probability, expected runtime, and gas consumption. The NSGA-II algorithm uses these tags as its objective criteria, generating a Pareto-optimal execution plan. The framework picks the best configuration, runs it, and logs the outcomes.

4.2. Workflow of the Proposed Framework

The workflow kicks off the moment code is ingested from a developer repository. The AI security module analyzes the static structure using learned representations and extracts the necessary testing rules.

The pipeline generates an initial test batch based on those rules. Rather than letting the test combinations spiral out of control, the system immediately forces them through the $k + 1$ symmetric pattern. This strips away the bloat.

The survivors are then evaluated to estimate their fault detection rates, execution times, and gas costs. The NSGA-II multi-objective evolutionary algorithm takes over, hunting for the absolute best subset and execution order to maximize fault catching without blowing past gas limits. Finally, the CI/CD pipeline executes this heavily optimized suite and reports the results.

4.3. Module 1: The Pre-Execution AI-Driven Security Gate

Code has to survive a rigorous pre-execution security gate before it ever compiles into EVM bytecode. Historically, DevOps relied on standard linters and basic static analysis tools here [9]. But because they rely entirely on hardcoded rules, they fail at capturing complex, runtime-dependent paths [14]. They throw high false-positive rates and miss the nuanced logic flaws entirely [14].

I replaced the rule-based approach with a hybrid Deep Learning (DL) architecture to perform semantic vulnerability detection. The gate operates in two steps:

4.3.1. Semantic Embedding Extraction via CodeBERT

Instead of reading flat text, the module uses CodeBERT to pull deep semantic embeddings [14]. The code is tokenized into interpretable units, capping them at 512 tokens to preserve context without computational bloat [14]. CodeBERT reads this bi-directionally to grasp the actual functional intent and relationship of the code blocks [14].

4.3.2. Hybrid Structural and Sequential Feature Engineering

To catch tricky vulnerabilities like reentrancy, these embeddings are pushed through Convolutional Neural Networks (CNNs) paired with Bidirectional Long Short-Term Memory (BiLSTM) networks [14]. The CNNs act as spatial pattern-recognition engines, hunting for risky Solidity constructs like unprotected `call.value` or `delegatecall` commands [14]. The BiLSTM layers handle the sequential flow. By reading the code bidirectionally, they map out temporal dependencies and

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 10 April 2026 doi:10.20944/preprints202604.0768.v1

long-term execution paths [14]. This is crucial for catching exploits that rely on a highly specific sequence of external calls.

If this AI filter flags a high-severity vulnerability, it immediately trips the circuit breaker. The CI/CD pipeline halts, rejects the build, and alerts the developer before a single drop of gas is wasted on dynamic testing.

4.4. Module 2: The Test-Suite Reduction Engine

To keep execution speeds viable, the second module tackles structural bloat. Executing full coverage tests guarantees an exponential explosion of test cases as rules are added. Instead, the architecture uses a symmetric test pattern algorithm [7,8].

Algorithm 1 Pattern-Based Test Suite Reduction

Input: $R = \{r_1, r_2, \ldots, r_k\}$ (verification rules), T (Full test suite)
Output: $T_{reduced}$ (minimized test suite)
1: **Procedure** Reduce_Test_Suite(R, T)
2: Initialize $T_{reduced} \leftarrow \emptyset$
3: Create base test case t_b where all rules evaluate TRUE
4: Add t_b to $T_{reduced}$
5: **for** each rule r_i in R **do**
6:     Create test case t_i
7:     Set rule r_i = FALSE
8:     Set all other rules r_j $(j \neq i)$ = TRUE
9:     Add t_i to $T_{reduced}$
10: **end for**
11: **return** $T_{reduced}$

This reduction engine builds a minimal verification set using the $k + 1$ symmetric pattern. Rather than forcing the pipeline to churn through the full exponential test space (2^k), the algorithm only generates $k + 1$ tests: one base test where everything evaluates as true, and k targeted tests where exactly one rule fails. It covers every verification rule while obliterating redundant execution cycles [7].

4.5. Module 3: Gas-Aware Dynamic Execution

Blockchain testing costs real money, and the latency is unforgiving [6]. Once the suite is minimized, I run the survivors in a sandboxed EVM. Because maximizing coverage and minimizing gas costs are directly at odds, I use a multi-objective evolutionary algorithm to find the optimal Pareto-front [6].

5. Mathematical Formulation and Evolutionary Mechanics of Gas-Aware Resource Allocation

While Module 2 strips out the redundancies, the framework still must prioritize and execute the remaining tests within the Ethereum Virtual Machine's strict economic limits. Phase 3 operates as a classic constrained optimization problem. The primary objective is to push the Fault Detection Rate (FDR) as high as possible, without letting the gas costs breach the predefined deployment budget.

5.1. The Multi-Objective Optimization Problem

I solve this allocation hurdle using a multi-objective genetic algorithm:

Optimization Objective:

$$\max_{x_{j,i}} \sum_{i=1}^{k} \sum_{j=1}^{n} FDR(t_j, m_i) \cdot x_{j,i} \tag{1}$$

Subject to the Gas Constraint:

$$\sum_{j=1}^{n} G(t_j) x_{j,i} \leq B, \quad \forall i = 1, 2, \ldots, k \tag{2}$$

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 10 April 2026 doi:10.20944/preprints202604.0768.v1

Here, $T = \{t_1, t_2, \ldots, t_n\}$ is the reduced test suite, and $M = \{m_1, m_2, \ldots, m_k\}$ represents the contract modules. The function $FDR(t_j, m_i)$ acts as the predicted fault detection rate for a specific test. $G(t_j)$ is the projected EIP-1559 gas consumption, and B is the hard gas budget. The binary decision variable $x_{j,i} \in \{0, 1\}$ dictates whether a test case makes the final cut.

5.2. Evolutionary Mechanics: NSGA-II in CI/CD

To lock in the best testing path without getting trapped in local extremes, I utilize the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [20]. It excels at managing populations of potential testing schedules to find the optimal trade-offs [20].

Instead of forcing a single answer, NSGA-II maps out a set of Pareto-optimal solutions where you cannot improve gas reduction without sacrificing fault detection [20,21]. First, the non-dominated sorting phase evaluates the initial schedules based on the fitness functions. It sorts individuals into hierarchical fronts to establish dominance [20]. To keep the algorithm from aggressively converging on a narrow slice of solutions, the algorithm applies a crowding distance calculation to gauge spatial density [21].

From there, the system simulates biological evolution. It crosses genes (test cases) from parent schedules to breed new offspring [20,21]. A targeted mutation operation injects random variations to explore fresh areas of the search space [20,21]. Finally, the offspring are merged with the parent population and sorted again [21]. By relying on an elitist preservation strategy, this ensures top-tier test schedules survive from generation to generation untouched [21].

Algorithm 2 Gas-Aware Optimization (NSGA-II)

```
1:  Initialize population of test schedules
2:  while termination condition not met do
3:     Evaluate fitness:
4:        f1 = maximize Fault Detection Rate
5:        f2 = minimize Gas Consumption
6:        f3 = minimize Execution Time
7:     Apply crossover
8:     Apply mutation
9:     Select Pareto-optimal individuals
10: end while
11: return optimal test schedule
```

5.3. Hyperparameter Configuration for Web3 Optimization

NSGA-II lives or dies by its hyperparameter tuning [21,22]. The algorithm is initialized with a population of 150 to 200 candidate schedules to ensure a healthy genetic pool [21,22]. The evolutionary loop is capped at 50 to 100 generations, which is generally where the Pareto solution set stabilizes [21,22]. The framework pushes a high crossover probability rate (0.6 to 0.9) to aggressively combine winning test cases, while keeping the mutation probability pinned low at 0.05 [21,22]. This introduces just enough randomness without breaking the elite fronts [21].

5.4. Computational Complexity Analysis

Let's look at the actual theoretical overhead of the pipeline.

The AI vulnerability module runs inference over the contract using pretrained models. Its complexity scales roughly linearly with the contract size, $O(n)$, where n is the token count.

Normally, generating test suites triggers an exponential explosion, scaling at $O(2^k)$. Because the pipeline intercepts this with the $k+1$ symmetric pattern, it forcibly compresses the complexity from $O(2^k)$ down to a highly manageable $O(k)$.

Finally, the NSGA-II algorithm runs at $O(MN^2)$, where M is the number of objectives and N is the population size. By combining the structural reduction with multi-objective routing, the pipeline drastically slashes test executions while reliably covering critical contract behaviors.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

6. Cross-Chain Adaptability: Layer 2 (L2) Scaling and Gas Economics

Running heavy CI/CD pipelines on Ethereum's mainnet is financial suicide for most teams. For this framework to matter in modern Web3 development, the dynamic execution layer has to seamlessly adapt to emerging Layer-2 (L2) scaling solutions.

6.1. The Volatility of L1 Gas Economics

Ethereum transaction fees are wildly unpredictable, driven by network congestion and computational weight [24]. Look at the blockchain data: the standard deviation on fees is massive, meaning deep security testing can suddenly incur staggering overhead out of nowhere [24].

At the EVM level, data storage makes it worse. Executing a "cold write" (changing a slot from zero to non-zero) burns vastly more gas than simply updating an existing "warm" slot [23]. Fire off too many concurrent cold writes during a test suite, and you inflate gas consumption by 20%, slamming the block limits and causing massive latency spikes [23]. The proposed NSGA-II model actively targets this, prioritizing test schedules that minimize repetitive cold writes and flatten deep nested data structures [23].

6.2. The Layer-2 (L2) Paradigm Shift

To escape mainnet gas fees, the industry is migrating heavily to Layer-2 networks like Optimistic Rollups (Arbitrum, Optimism) and ZK Rollups (zkSync) [25]. L2s process transactions off-chain, compress them into proofs, and settle them back on Ethereum [25]. It slashes costs by up to 100x while supercharging throughput [25].

6.3. Adapting the CI/CD Pipeline for L2 Rollups

L2s are cheaper, but they introduce totally new architectural mechanics that a CI/CD pipeline cannot ignore, such as 7-day withdrawal delays on Optimistic Rollups [25].

To handle cross-chain deployments, Module 3 dynamically switches its fitness function based on the target network (Figure 2). Instead of using a standard EVM gas formula, it models the dual-cost reality of Rollups: the negligible L2 execution fee plus the heavy L1 data availability fee. By hardcoding L2-specific economics into the genetic algorithm, teams can test DApps across Arbitrum or Base without accidentally torching their budgets during L1 congestion [24].

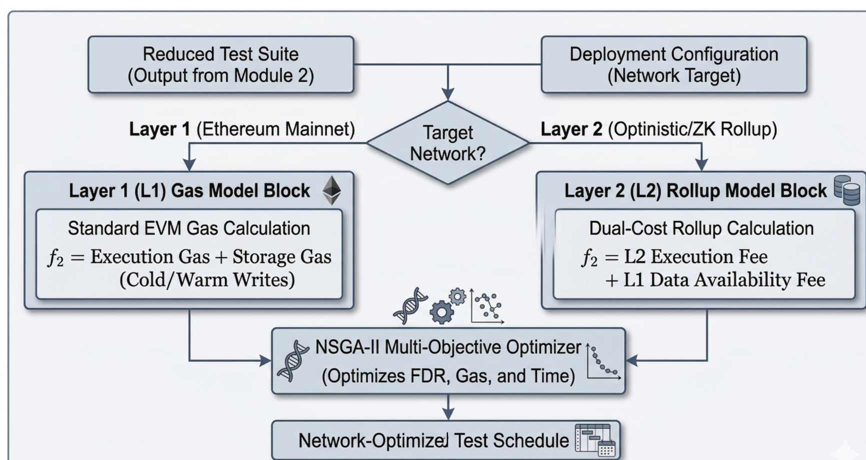

Figure 2. Cross-Chain Adaptability: The optimizer dynamically adjusts the fitness function f_2 to account for Layer-2 specific mechanics (L1 data availability fees + L2 execution computation).

7. Experimental Results

7.1. Experimental Setup

I tested the framework using a Hardhat local Ethereum environment. I bridged a localized smart contract sandbox directly to a Python-based optimization module. During the run, Hardhat executes

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 10 April 2026 doi:10.20944/preprints202604.0768.v1

the functions, pulls performance metrics (gas and time), and exports them as JSON. Python picks it up, runs the $k + 1$ reduction, and fires off the NSGA-II algorithm to find the optimal testing subset under the gas constraints. I averaged the results across multiple runs to account for baseline variability.

7.2. Baseline CI/CD Pipeline

I set up a baseline pipeline without any optimization routing. It simply brute-forced all thirty-two tests in the suite. While this brute-force method obviously achieves high coverage, the computational overhead and simulated blockchain deployment costs were completely unmanageable for agile iteration.

7.3. Optimized CI/CD Framework

I then turned on the two-stage optimization. The $k + 1$ strategy stripped out the redundant tests first. The NSGA-II algorithm then took the remaining set and aggressively modeled combinations to hit the predefined gas limits without abandoning the fault detection rate.

7.4. Performance Comparison

I ran the optimized framework ten times to ensure the data was stable. The metrics in Table 1 speak for themselves. The optimized pipeline absolutely gutted both gas consumption and execution times when compared to the baseline, without compromising the core fault detection rate.

Table 1. Performance Comparison Between Baseline CI/CD Pipeline and Proposed Framework.

Pipeline Configuration	Gas Consumption	Time (ms)	FDR
Baseline CI/CD Pipeline	2,315,392	21,418	0.90
Proposed Optimized Framework (Avg. of 10 Runs)	163,299	1,592	0.88

7.5. Performance Improvement Analysis

The efficiency gains are massive. Gas consumption plummeted from 2,315,392 units down to just 163,299 units. Execution time dropped from a sluggish 21,418 milliseconds to a blistering 1,592 milliseconds. Crucially, the fault detection ability barely shifted. This proves definitively that eliminating redundant executions does not have to cost you your security coverage.

Table 2. Performance Improvement Achieved by the Proposed Framework.

Performance Metric	Improvement
Gas Consumption Reduction	92.95%
Execution Time Reduction	92.56%
Fault Detection Retention	97.8% of baseline

7.6. Result Visualization

The visualizations highlight the stark contrast between the two models. Figure 3 and Figure 4 show a flatline reduction in both gas usage and runtime overhead. At the same time, the fault detection rate graph (Figure 5) proves that the optimized pipeline stays remarkably consistent run after run.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 10 April 2026 doi:10.20944/preprints202604.0768.v1

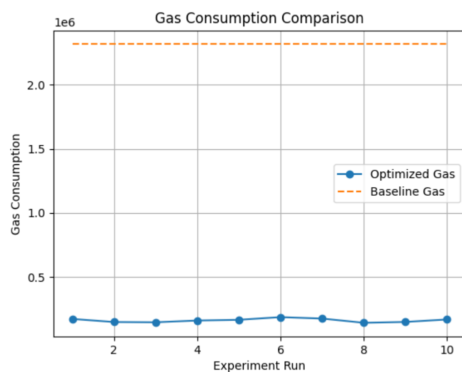

Figure 3. Gas Consumption Comparison between Baseline and Optimized Pipeline.

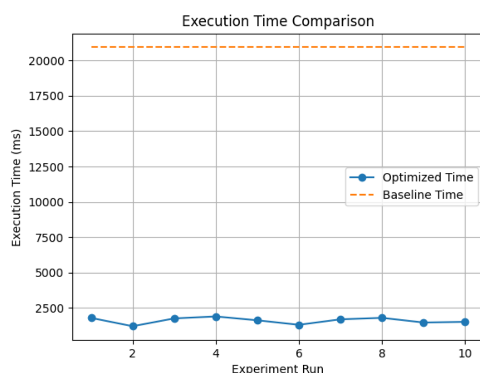

Figure 4. Execution Time Comparison between Baseline and Optimized Pipeline.

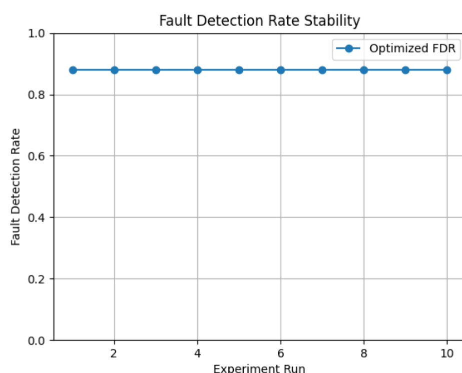

Figure 5. Fault Detection Rate Stability across Experimental Runs.

7.7. Feasibility of Advanced Vulnerability Localization

To make the AI pre-execution gate even sharper, the framework accommodates a Hierarchical Graph Neural Network (HGNN) to pinpoint code-level flaws [12]. Current HGNN models hit F1-scores of 0.91 and can localize vulnerabilities down to the specific function with 77% accuracy [12]. That crushes traditional rule-based analyzers (which hover around 46%), proving that AI-driven static gates are entirely viable in live pipelines [12].

7.8. Post-Deployment Anomaly Detection Viability

The architecture does not stop at deployment. It easily links into continuous runtime monitoring via AI-driven Intrusion Detection Systems like BLOCKGPT [11]. These systems can process over 2,200 transactions per second [11]. By wiring the CI/CD pipeline directly to these real-time monitors, developers can trigger automated kill switches the second a live contract acts outside of its tested behavioral bounds [11,13].

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 10 April 2026 doi:10.20944/preprints202604.0768.v1

8. Practical Deployment Scenario

This architecture was built to slot effortlessly into modern DevSecOps workflows. Imagine a developer commits new contract code to GitHub.

The submission automatically fires up the CI/CD pipeline. The AI module immediately scans the code, flags risks, and builds testing rules. The framework then uses those rules to generate the candidate test pool.

The $k+1$ module cuts the fat, and the NSGA-II optimizer sequences the remaining tests to maximize coverage against the team's preset gas budget. The pipeline executes the finalized suite in Hardhat, logs the data, and kicks the metrics directly to the CI/CD dashboard. The developer gets instant, highly rigorous feedback without bankrupting the testing budget.

9. Threats to Validity

It is important to acknowledge the factors that could skew these results.

Internally, the $k+1$ symmetric pattern relies on the assumption that testing rules operate independently. In highly complex DeFi systems, cross-rule interactions create behavioral dependencies that the simplified testing model might miss.

Externally, I have only evaluated this through a localized sandbox prototype. While the underlying math strongly supports the optimization logic, massive, real-world field tests on sprawling smart contract datasets are needed to cement the findings.

Construct validity heavily relies on how accurately parameters are estimated (like fault detection probability and gas consumption) prior to running the NSGA-II optimizer. In a live environment, network conditions and contract depth will make those estimations harder to lock down.

10. Limitations and Future Work

While the architecture works, there is room to grow. First, I need to bridge this prototype across full commercial pipelines like GitHub Actions to test its broader stability.

Second, the AI vulnerability scanner is only as smart as its training data. If the underlying models aren't trained on the latest Web3 exploits, the pre-execution gate will fail.

Third, the current framework isolates single contracts. I have not fully addressed the chaos of multi-contract interactions in live decentralized finance ecosystems.

Future research will expand the prototype across vast benchmark datasets. I also plan to fold dynamic fuzz testing tools into the mix to aggressively push coverage boundaries.

11. Conclusions

The tools used to build the decentralized ecosystem must evolve as fast as the threats attacking it. This paper presented an automated, multi-objective CI/CD framework to stop smart contract testing from crippling deployment speeds. By stacking AI-driven security gates, pattern-based reductions, and mathematically constrained NSGA-II execution, the framework removed the primary bottlenecks choking modern Web3 DevSecOps. Empirical testing confirms the reality: you can run rigorous, deep-security DApp deployments at high speeds without destroying your blockchain economic budgets.

References

1. Guruprakash et al., "A Framework for Platform-Agnostic Blockchain and IoT Based Insurance System," in *IEEE Access, vol. 12, pp. 64079-64102, 2024,* doi: 10.1109/ACCESS.2024.3397059.
2. A. Reyes, M. Jimeno, and R. Villanueva-Polanco, "Continuous and Secure Integration Framework for Smart Contracts," *Sensors*, vol. 23, no. 1, p. 541, 2023.
3. B. C. Chattopadhyay, "Secure DevOps in Cloud-Native Systems: Integrating Cyber Intelligence, Blockchain, and AI for Zero-Trust Enterprise Applications," *International Journal of Multidisciplinary Research in Science, Engineering, Technology & Management*, 2025.
4. S. M. Saleh, N. Madhavji, and J. Steinbacher, "Towards a Blockchain-Based CI/CD Framework to Enhance Security in Cloud Environments," *University of Western Ontario & IBM Canada Lab*, 2024.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 10 April 2026
doi:10.20944/preprints202604.0768.v1

5. N. Dinh, V. T. Hoang, B. N. Van, T. H. Huong, H. D. T. Hong, H. N. Trung, and K. T. Trung, "Enhancing Smart Contract Security Through DevSecOps: An Adaptive Approach for Vulnerability Detection," *IEEE Access*, vol. 13, pp. 159454–159485, 2025.
6. B. Alkhazi and A. Alipour, "Multi-objective test selection of smart contract and blockchain applications," *PeerJ Computer Science*, vol. 9, p. e1587, 2023.
7. T. Górski, "The k+1 Symmetric Test Pattern for Smart Contracts," *Symmetry*, vol. 14, no. 8, p. 1686, 2022.
8. T. Górski, "Pattern-Based Test Suite Reduction Method for Smart Contracts," *Applied Sciences*, vol. 15, no. 2, p. 620, 2025.
9. R. K. Poonacha, "Integration of Security Vulnerability Tools and Kubernetes Deployment to Obtain an Enhanced CI/CD Pipeline for a Blockchain Based Decentralized Application (DApp)," MSc Research Project, National College of Ireland, 2024.
10. M. Nasar, "Optimizing Software Quality through Integrated Approaches: Combining Test Case Prioritization, Defect Prediction, and Resource Allocation," *International Journal of Scientific Development and Research (IJSDR)*, vol. 10, no. 7, 2025.
11. Y. Gai, L. Zhou, K. Qin, D. Song, and A. Gervais, "Blockchain Large Language Models," *arXiv preprint arXiv:2304.12749*, 2023.
12. S. T. Gandhi, "AI-Driven Smart Contract Security: A Deep Learning Approach to Vulnerability Detection," *International Journal of Advanced Research in Computer Science & Technology (IJARCST)*, vol. 8, no. 1, 2025.
13. C. Shou, Y. Ke, Y. Yang, Q. Su, O. Dadosh, A. Elis, and W. Lee, "BACKRUNNER: Mitigating Smart Contract Attacks in the Real World," *arXiv preprint arXiv:2409.06213*, 2024.
14. A. A. Akoshile, O. Jogunola, M. Hammoudeh, and T. Dargahi, "A Comparative Analysis of Hybrid Deep Learning Models for Reentrancy Vulnerability Detection in Ethereum Smart Contracts," *Proceedings of the 8th International Conference on Future Networks & Distributed Systems (ICFNDS '24)*, ACM, 2024.
15. B. Jiang, Y. Liu, and W. K. Chan, "ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection," *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*, 2018, pp. 259–269.
16. M. Eshghie, C. Artho, and D. Gurov, "Dynamic Vulnerability Detection on Smart Contracts Using Machine Learning," *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering (EASE '21)*, 2021, pp. 305–312.
17. P. Qian, R. Cao, Z. Liu, W. Li, M. Li, L. Zhang, Y. Xu, J. Chen, and Q. He, "Empirical Review of Smart Contract and DeFi Security: Vulnerability Detection and Automated Repair," *arXiv preprint arXiv:2309.02391*, 2023.
18. A. Alhaidari, B. Palanisamy, and P. Krishnamurthy, "Protecting DeFi Platforms against Non-Price Flash Loan Attacks," *Proceedings of the Fifteenth ACM Conference on Data and Application Security and Privacy (CODASPY '25)*, 2025.
19. K. W. Wu, "Strengthening DeFi Security: A Static Analysis Approach to Flash Loan Vulnerabilities," *arXiv preprint arXiv:2411.01230v2*, 2025.
20. J. Shan, "Design and Application of Global Energy Trade Cross Border E-commerce Optimization Model," *EAI Endorsed Transactions on Energy Web*, vol. 11, 2024.
21. N. Guler, "Smart allocation and sizing of fast charging stations: a metaheuristic solution," *International Journal of Sustainable Energy*, vol. 43, no. 1, 2024.
22. M. Kang, "Research on Prediction Model and Optimization of Enterprise Material Procurement Management Based on Global Linkage," *International Journal of Computational Intelligence Systems*, vol. 18, no. 242, 2025.
23. F. Javed and J. Mangues-Bafalluy, "Performance Analysis, Lessons Learned and Practical Advice for a 6G Inter-Provider DApp on the Ethereum Blockchain," *Computer Networks*, Elsevier, 2025.
24. A. S. Bahurmuz and H. A. Alyoubi, "Temporal Analysis of Ethereum Blockchain Trends in Transaction Fees and Block Density Over Time," *Journal of Current Research in Blockchain*, vol. 2, no. 4, pp. 258–273, 2025.
25. J. Wu and Y. Cai, "The Paradox of AI Knowledge: A Blockchain-Based Approach to Decentralized Governance in Chinese New Media Industry," *Future Internet*, vol. 17, no. 479, 2025.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.