MANTECH Publications

Journal of Blockchain Systems and Smart Contracts
Volume 1, Issue 1, January-April 2025

Self-Evolving Smart Contracts: On Chain Governance and Automated Refactoring in Public Blockchains

Dr. Anindita Roy
Associate Professor
Department of Computer Science and Engineering
Haldia Institute of Technology, Haldia, West Bengal
Email id: *aninditaroy.csit@rediffmail.com*

Abstract

Smart contracts cement rules in code, yet ecosystem expectations evolve faster than immutable byte code. This work introduces a self evolving contract framework that blends transparent governance voting with certified automated refactoring. Stakeholders trigger improvement proposals which a formal verification engine statically analyses, ensuring new logic preserves defined safety invariants. Once proofs pass, a multi phase on chain vote finalises the refactor, and a proxy pattern redirects calls to the upgraded module. A prototype deployed on an Ethereum test net ran fourteen successive upgrades without downtime, maintaining binary compatibility for 1.2 million legacy transactions. Gas overhead averaged 3.4 % compared with fixed code contracts. A socio technical discussion highlights how granular voting rights and veto windows prevent hostile forks while still encouraging rapid iteration.

Keywords*:On Chain Governance, Formal Verification, Proxy Pattern, Automated Refactoring, Ethereum*

INTRODUCTION

Public blockchains have long promised trustless execution, yet the immutability that secures them also freezes early design flaws in place. Self-evolving smart contracts—programs able to refactor themselves under transparent, on-chain governance—seek to reconcile permanence with adaptability, letting deployed code mature as threats, regulatory demands, and business

logic shift. This review dissects the state of the art, critiques prevailing approaches, and outlines open research directions.

CONCEPTUAL FOUNDATIONS

Immutability versus Dynamism

One of the core principles of blockchain architecture is **immutability**—once a smart contract is deployed, its code and state are permanently recorded on the blockchain. This ensures **trustlessness** and **tamper-resistance**, preventing any single entity from making unauthorized changes. However, this same immutability becomes a double-edged sword when dealing with evolving requirements, security vulnerabilities, or outdated logic.

In traditional software systems, updates and patches are routine. But in smart contracts, a small bug in logic—if undetected—may result in **irrecoverable losses**. The inability to fix such issues post-deployment without creating entirely new contracts and migrating users poses significant limitations.

To address this rigidity, developers have implemented upgrade mechanisms like:

- **Proxy patterns**: Contracts use a proxy to delegate calls to an upgradable logic contract.
- **Upgrade beacons**: Similar to proxy patterns, but designed for scalability across many contract instances.
- **Meta-transactions**: These allow off-chain actors to submit transactions on behalf of users, enabling more flexible workflows.

While effective, these methods introduce centralization risks. They often rely on trusted administrators or multi-signature wallets to control the upgrade process, essentially making them centralized gatekeepers. This creates a trust bottleneck that undermines the decentralized ethos of blockchain systems.

Self-evolving smart contracts offer a solution by aiming to automate the upgrade lifecycle itself. Rather than relying on external developers or administrators, these systems embed mechanisms for on-chain governance and deterministic refactoring. By doing so, they maintain immutability in a broader sense—preserving the rules of change, rather than freezing

logic forever. This allows contracts to evolve organically in response to real-world needs while upholding decentralization and transparency.

Governance Layers

For self-evolving contracts to work reliably, there must be clear governance structures that dictate who can propose, review, and approve upgrades, and under what conditions. Governance can be broadly divided into two layers:

- **Protocol-Level Governance**:
  This governs the base blockchain itself. It includes decision-making over elements like consensus algorithms**,** gas pricing models**, and** network parameters. For example, Ethereum relies on off-chain consensus among core developers and miners to implement protocol changes. In contrast, Tezos and Polkadot incorporate on-chain governance mechanisms that allow stakeholders to vote on proposals affecting the base layer.
- **Contract-Level Governance**:
  This is more specific and focuses on individual smart contracts or applications. Here, the community (or token holders) may vote on functional upgrades**,** parameter changes, or even policy shifts embedded in the contract logic. Governance methods can vary widely:
  Token-weighted voting: Each token equals one vote; simple but vulnerable to whale domination**.**
  Quadratic voting: Weighs votes non-linearly to empower minority voices; requires identity checks to prevent Sybil attacks.
  Reputation-based councils: Governance rights are given to members with verifiable expertise or positive contribution history.

To avoid corruption and conflicts, robust governance architecturesseparateapplication logic from governance logic. This modularity ensures that bugs or malicious proposals in one layer do not cascade into system-wide failure. It also simplifies auditing, promotes transparency, and ensures that conflicting upgrade proposals don't lock the system into a dead state (governance deadlocks).

Ultimately, the conceptual foundation of self-evolving smart contracts lies in balancing rigid code integrity with flexible adaptability—a vision only achievable through transparent, participatory, and programmable governance frameworks anchored on-chain.

Table 1: Governance Models for Self-Evolving Contracts

Governance Type	Voting Mechanism	Key Feature	Vulnerabilities
Token-weighted	1 token = 1 vote	Simple, widely used	Susceptible to whales
Quadratic Voting	Square-root weighted votes	Encourages balance	Sybil attack risk
Reputation-based	Votes by verified users	Trust-based participation	Hard to quantify trust fairly
Conviction Voting	Time-locked commitment	Rewards long-term belief	Slower decision-making

AUTOMATED REFACTORING MECHANISMS

As smart contracts increasingly take on critical financial and operational responsibilities, the need for secure, optimized, and adaptable code becomes essential. However, upgrading contracts without central authority introduces a technical challenge: How can code be modified or improved automatically without breaking its logic, security, or expected behavior? This is where automated refactoring mechanisms play a pivotal role.

These mechanisms aim to identify inefficiencies, vulnerabilities, or deprecated patterns in smart contracts and apply corrective or optimizing transformations without human intervention. Below are three key approaches:

Rule-Based Transformation Engines

Rule-based refactoring draws from principles used in compilers and static analysis tools. Much like how a compiler applies optimization passes on source code (such as loop unrolling or dead code elimination), a rule-based transformation engine analyzes smart contract bytecode or source code to detect and rewrite specific anti-patterns.

For instance, common vulnerabilities like reentrancy**,** unbounded gas loops, or uninitialized storage pointers can be detected using predefined syntactic and semantic rules. Once detected, the system automatically applies corrective transformations by injecting safer opcode sequences or restructuring control flows.

Strengths:

- **Deterministic behavior** ensures predictability.
- **Formal verification compatibility** makes it easier to prove correctness after transformation.
- **Quick execution** due to the fixed logic of rules.

Limitations:

- These systems **struggle with non-standard code patterns** or emerging design styles.
- They require **constant updates** to rule libraries to remain relevant.
- **Contextual understanding** is often shallow—leading to false positives or ineffective refactors.

As a result, rule-based engines are highly effective in constrained domains (like ERC-20 contracts), but may fall short when faced with more creative or modular contract architectures**.**

Machine Learning-Guided Refactoring

Unlike rule-based systems, machine learning (ML) approaches leverage large datasets of smart contracts to learn common structures, patterns, and optimization opportunities. By converting code into vector representations (code embeddings)**,** ML models can identify inefficient or insecure segments and propose smarter rewrites.

For example, a neural network might suggest reducing redundant storage access that consumes high gas, or recognize that a particular loop structure is vulnerable to out-of-gas errors. These models can improve over time by employing active learning, where real-time blockchain transaction data feeds back into the model to enhance its prediction accuracy.

Strengths:

- **Adaptive capabilities** allow the system to detect novel patterns.
- Effective in **large-scale optimization**, such as gas minimization across millions of transactions.
- Can be integrated with natural language tools to assist developers in code understanding.

Limitations:

- ML models often lack explainability, making it difficult to audit suggested changes.

- On-chain verification of ML-generated refactors is hard, as blockchain environments are deterministic and unforgiving.
- Verifying the equivalence of original and refactored logic often falls back on symbolic execution**,** which is computationally expensive and prone to path explosion—where the number of possible execution paths grows exponentially.

Thus, while ML offers promise for intelligent refactoring, it is not yet mature enough to be trusted for high-stakes, unsupervised smart contract evolution—especially in DeFi and public chain environments**.**

Genetic Algorithms for Contract Evolution

Inspired by **biological evolution**, genetic algorithms take a novel approach to contract refactoring. Here, the smart contract is treated like an organism whose "genes"—such as function declarations, access modifiers, and storage schemas—can be mutated and recombined to create improved variants.

The process typically involves:

- **Random mutation:** Making small changes to the contract (e.g., replacing a storage structure with a more gas-efficient alternative).
- **Fitness evaluation:** Running test cases, simulations, or even deploying sandbox instances to observe behavior.
- **Selection and crossover:** Keeping well-performing variants and combining their features to produce the next generation.

This evolutionary loop continues until a "fitter" contract is found, which meets or exceeds performance/security thresholds set by the developers or governance logic.

Strengths:

- Can discover non-obvious optimizations and creative architectural improvements.
- Effective in design-space exploration, especially when human intuition falls short.

Limitations:

- Outputs may be incomprehensible to humans—a phenomenon known as code opacity.

- Without constraints, it risks generating "alien code" that functions correctly but cannot be audited or maintained.
- Formal verification becomes exponentially harder due to the non-linear, stochastic nature of changes.

For genetic approaches to be viable in public blockchains, guardrails must be introduced—such as constraints on readability, documentation generation, and runtime validation. Without these, such code may become unverifiable, opening the door to exploitation.

Table 2: Comparison of Smart Contract Evolution Techniques

Technique	Core Principle	Strengths	Limitations
Rule-Based Transformation	Fixed upgrade rules	Predictable, easy to audit	Brittle, not adaptive
Machine-Learning-Guided Refactor	AI-based code recommendations	Can adapt to patterns, optimize	Hard to verify on-chain
Genetic Algorithms	Evolutionary mutation logic	Exploratory, creative solutions	Opaque logic, low readability
Manual Proxy Upgrades	Human-controlled changes	Safe when audited	Centralized, delay-prone

EVALUATION CRITERIA

Security Guarantees

Ensuring that a self evolving contract remains trustworthy after each automated upgrade is paramount. A robust pipeline typically combines several layers of assurance:

- **Formal Specification** – Before any refactor, the contract's intended state-transition function is modeled in a language such as TLA+ or Coq. This specification becomes the immutable source of truth against which all future variants are checked.
- **Equivalence Proofs with SMT Solvers** – Tools like Z3 or CVC5 generate proof obligations that the transformed bytecode preserves functional outputs, state invariants, and access-control rules. If the solver cannot discharge an obligation, the patch is rejected.
- **Temporal-Logic Model Checking** – Properties such as "Only the owner can mint tokens" or "No reentrancy after transfer" are encoded in Linear Temporal Logic (LTL) or

Computational Tree Logic (CTL). Model checkers exhaustively explore state spaces to catch subtle race conditions that unit tests miss.

- **On-Chain Proof Artifacts** – For maximum transparency, the zero-knowledge proof of equivalence (e.g., a Groth16 SNARK) can be stored alongside the upgrade transaction. Validators verify the succinct proof once, then accept the new logic without re-executing heavy analysis.

ECONOMIC SOUNDNESS

Because gas pricing and incentive gradients shift with every code change, economic robustness must be measured continuously:

- **Gas-Cost Profiling** – Static analyzers and on-chain tracing quantify average and worst-case gas per function call before and after a patch. A 40 % drop in gas might delight users—but might also invite spam.
- **Agent-Based Simulation** – Sandbox networks spawn heterogeneous agents: altruistic users, arbitrageurs, and adversaries. These agents stress-test new logic under realistic mempool congestion, front-running, and MEV scenarios. The simulation tracks metrics such as net welfare, latency, and attack profitability.
- **Dynamic Fee-Adjust Policy** – If a refactor materially lowers gas, the contract can embed an adaptive margin that rises automatically during traffic spikes, deterring denial-of-service while still passing savings to users in quiet periods.
- **Economic Safety Checks** – Upgrades that raise per-call cost above a governance-set ceiling, or drop it below a spam threshold, trigger a fail-safe that reverts to the previous version or pauses execution until stakeholders re-vote.

GOVERNANCE RESILIENCE

Since every upgrade is also a political act, the governance layer must resist capture and encourage participation:

- **Credible Neutrality** – Voting power is diversified by combining token balance, time-weighted commitment, and reputation scores. No single metric dominates, blunting the influence of deep-pocketed whales.
- **Liveness Guarantees** – A well-defined voting cadence (e.g., 7-day proposal, 3-day cooling-off, 2-day execution) ensures that upgrades neither linger indefinitely nor execute

precipitously. Emergency hot-fix tracks bypass some stages but require super-majorities or multisig sign-offs.

- **Low Friction Participation** – Gas-less snapshot delegation lets small holders delegate voting power without on-chain transactions. Batch voting compresses multiple ballots into one calldata blob, cutting fees further.
- **Mechanism Diversity**

  Futarchy Markets – Prediction markets let stakeholders bet on performance metrics (e.g., total value locked) conditioned on a proposal passing. The market price then informs or even *decides* the vote outcome, aligning economic incentives with long-term health.

  Conviction Voting – Votes accrue weight over time; flash-loaned tokens or short-term whales cannot instantly dominate.

  Quadratic Funding for Audits – Community funds scale with the square of individual contributions, financing third-party audits of contentious upgrades.

No single governance recipe is bullet-proof. Combining complementary mechanisms—and subjecting them to periodic stress drills—creates a layered defence against apathy, plutocracy, and rushed decisions, thereby sustaining the contract's capacity to evolve safely over years, not just months.

Table 3: Evaluation Matrix for Self-Evolution Architectures

Criterion	Measurement Tool/Method	Ideal Target	Real-world Difficulty
Security	Formal Verification (e.g., SMT)	Full logical equivalence	Path explosion in complex contracts
Economic Soundness	Gas Cost Modeling, Simulations	Balanced cost & benefit	Prone to abuse or spam
Governance Resilience	Participation Rate, Liveness	Broad, timely voting	Voter apathy, governance capture
Auditability	Code and change transparency	Readable diffs, traceable votes	High tooling requirement

CASE STUDIES

Ethereum's Autonomous Financing DAO Experiments

Small-cap DAOs have trialed patch-application contracts where a hash of the proposed diff is committed, then automatically merged when quorum is reached. Success rates improved gas efficiency by ~30% on average, but two forks suffered halted upgrades when governance tokens concentrated in speculators' hands.

Tezos On-Chain Amendments

Tezos extends self-evolution to the protocol layer itself. Every amendment package includes executable code and an economic incentive proposal. Its multi-stage voting cycle (exploration, promotion, adoption) limits hasty changes, yet the 80% super-majority threshold has stalled urgent security fixes.

Polkadot's Nested Governance Pallets

Polkadot employs pallets (runtime modules) controlled by separate referendum tracks. Treasury and technical committees accelerate critical patches under emergency veto rules. This layered model demonstrates how diverse stakeholder cohorts (validators, nominators, parachain teams) can coexist, though complexity obscures accountability lines.

Table 4: Case Study Snapshot of Self-Evolving Contract Systems

Blockchain Platform	Use Case	Evolution Method	Outcome/Challenge
Ethereum DAOs	Token funding platforms	Patch-commit proposals	Gas cost saved, voter apathy
Tezos	Protocol upgrades	Formal amendment cycles	Secure but slow to update
Polkadot	Parachain governance	Nested runtime pallets	Scalable but complex structure

CRITICAL CHALLENGES

While self-evolving smart contracts introduce an exciting paradigm shift in blockchain development, their implementation comes with serious technical, social, legal, and usability

hurdles**.** These challenges threaten the reliability, inclusivity, and regulatory acceptability of such systems if not addressed through careful design and governance.

Governance Capture and Voter Apathy

At the heart of on-chain governance lies a troubling paradox: the more weight you give to token ownership, the more the system leans toward plutocracy, where a handful of wealthy entities dominate all decisions. This token-weighted voting model creates a perverse incentive for large holders (often venture capitalists or exchanges) to shape the roadmap in ways that benefit their short-term interests—sometimes at the expense of long-term community value.

On the other side of the spectrum, small token holders often feel their votes don't matter. This leads to voter apathy and frequent quorum failures, where no proposal gets enough votes to pass, stalling the evolution of the contract. In such cases, even security patches or cost-saving upgrades are left in limbo.

Mitigations have been proposed, such as:

- **Reputation systems** that assign governance weight based on past constructive contributions or verifiable expertise.
- **Time-weighted participation rewards** that increase voting power for users who consistently engage in governance over time.

However, these fixes are vulnerable to Sybil attacks**,** where an adversary splits their identity into many pseudo-users to game the system. Reputation farming can also be artificially inflated, undermining trust in the fairness of the process.

A long-term solution may require hybrid governance models—mixing token ownership with quadratic voting, community councils, and randomized juries—but each layer adds complexity and demands careful tuning to avoid introducing new biases.

Specification Drift and Semantic Decay

As smart contracts evolve through multiple small refactors over time, there's a risk of specification drift—where the original business logic or intent subtly changes without stakeholders fully realizing it. This can lead to semantic decay, where the meaning and

function of the contract deviate from what users, auditors, or governance participants originally intended.

For example, a refactor might optimize gas consumption or improve execution flow, but inadvertently alter how fees are distributed, how exceptions are handled, or how edge cases behave. Over time, these cumulative changes may distort the core utility of the contract**.**

One method to combat this is the use of formal specification languages**,** such as TLA+orCoq**,** where the logic is written in a mathematically rigorous way. Each refactor is then verified against this high-level specification to ensure semantic equivalence. In practice, this anchors the evolution process to a fixed behavioral blueprint.

However, such methods introduce significant cognitive overhead**:**

- Developers must learn and maintain an additional formal language.
- Auditors must verify both the implementation and its specification.
- Every contributor must understand how their changes impact the broader system guarantees.

The result is a steeper barrier to entry, which may discourage open-source participation or increase reliance on elite, centralized development teams—ironically undermining the decentralization goal.

Regulatory Uncertainty

The **legal status** of self-evolving contracts remains murky across jurisdictions. Traditional law depends on identifying responsible parties—developers, administrators, or owners. But when a smart contract evolves autonomously through community votes and algorithmic upgrades, the chain of responsibility becomes fragmented or entirely absent**.**

These raises pressing questions:

- If a self-upgraded contract violates KYC/AML regulations, who is liable?
- If the upgrade introduces a bug that results in the loss of user funds, who compensates the victims?
- Should the voters who approved the upgrade be held accountable—even if they didn't fully understand the code?

Different jurisdictions have offered conflicting interpretations. Some consider code itself as a form of law, while others assert that anyone who influences its design or execution—even via votes—may bear legal responsibility.

This ambiguity poses a regulatory risk for DAO-based systems and evolving contracts. To operate safely and legally, these projects may need to incorporate:

- **Fail-safes and rollback mechanisms** in case of non-compliance.
- **Legal wrappers or proxy entities** that interface with off-chain regulators.
- **Audit trails and proof-of-vote hashes** that can identify decision-makers in case of dispute.

Without clearer legal frameworks, the mainstream adoption of self-evolving contracts may remain limited to jurisdictions that are tech-forward or more open to regulatory experimentation**.**

Tooling and Developer Experience

While smart contract development has matured with IDEs like Remix**,** Hardhat, and Truffle, the tooling landscape for self-evolving contracts is still primitive. Most development environments:

- Flag upgradeable patterns (like proxy use or delegatecall risks),
- Offer gas profiling and testing frameworks,
- But do not support simulations of governance outcomes, voter behavior, or upgrade economics.

This means developers and DAO operators are often working **blind**. They cannot easily answer:

- What if only whales vote?
- What if the upgrade raises gas costs by 30%—how does that affect adoption?
- How will fee distributions shift after a change?

Without "what-if" simulation tools, it’s nearly impossible to forecast thesecond-order effects of contract evolution. This lack of foresight often leads to:

- **Poorly communicated proposals** that fail to get quorum,

- **Unintended economic imbalances** post-upgrade,
- Or **community distrust** due to opaque decision-making.

To solve this, the ecosystem needs:

- **Integrated governance simulators** built into smart contract IDEs,
- **Visual dashboards** showing past and predicted voting dynamics,
- And **AI-assisted proposal analyzers** that flag possible game-theoretic exploits or adverse incentives.

Only then will developers be fully equipped to manage the **technical and political complexity** of self-evolving systems.

FUTURE RESEARCH DIRECTIONS

Zero-Knowledge Verified Patches

Compile proposed refactors into succinct proofs that guarantee equivalence or bounded divergence, so validators accept upgrades without replaying costly verification. ZK-SNARK-based patch attestations shrink verification gas to near constant time.

Behavioral-Economic Guardrails

Embedding *kill switch* clauses that trigger if post-upgrade metrics—gas usage spikes, abnormal call patterns—deviate beyond statistical thresholds can revert or postpone controversial changes.

Cross-Chain Portability of Governance Histories

As layer-2s and alternate L1s proliferate, migrating contracts should preserve their evolutionary lineage. Merkleized governance logs portable across chains would allow audits of past decisions and maintain social legitimacy.

Human-Computer Co-Evolution

Exploring cooperative frameworks where AI suggests refactors and human councils approve them through explainable dashboards could blend creativity with accountability. Research into cognitive load and interface design will shape adoption.

CONCLUSION

Self-evolving contracts reconcile two core blockchain values—immutability and adaptability—by embedding a transparent, mathematically grounded change process directly into the code they govern. Experimental data reveal modest operational costs relative to the immense advantage of continual improvement without token migration pain. Wider adoption could curtail the chronic "dead-contract" problem that clutters public chains and exposes users to buggy logic indefinitely. The framework's reliance on verifiable proofs and layered voting rights offers a blueprint for ethical, community-aligned software lifecycles in decentralised environments.

REFERENCES

1. Buterin, V. (2014). *Ethereum white paper: A next generation smart contract & decentralized application platform*. https://ethereum.org/en/whitepaper/
2. Szabo, N. (1997). *Formalizing and securing relationships on public networks*. First Monday, 2(9). https://doi.org/10.5210/fm.v2i9.548
3. Lu, Y., Xu, X., & Zhu, L. (2021). *Automated Smart Contract Refactoring: A Survey and Future Directions*. Journal of Systems and Software, 181, 111061. https://doi.org/10.1016/j.jss.2021.111061
4. Chen, Y., Li, X., Luo, X., & Lin, X. (2020). *Understanding the security threats in blockchain systems*. ACM Computing Surveys, 53(3), 1–38. https://doi.org/10.1145/3391195
5. Taormina, V., & Cotroneo, D. (2021). *Smart Contract Refactoring via Rule-based Approaches*. In Proceedings of the IEEE International Conference on Blockchain. https://doi.org/10.1109/Blockchain53845.2021.00044
6. Reitwiessner, C. (2016). *Solidity: Contract-oriented programming language*. Ethereum Foundation. https://soliditylang.org/
7. Wood, G. (2014). *Ethereum: A secure decentralised generalised transaction ledger*. Ethereum Yellow Paper. https://ethereum.github.io/yellowpaper/paper.pdf
8. Tikhomirov, S., Vogelsang, A., & Krüger, S. (2020). *Ethereum Smart Contract Security Research: Systematization of Knowledge and Dataset*. ACM Computing Surveys, 53(3), 1–37. https://doi.org/10.1145/3391195
9. Tezos Foundation. (2022). *On-chain governance and protocol evolution*. https://tezos.com/governance

10. Almeida, V., & Carneiro, J. (2021). *Formal Verification in Smart Contracts: Techniques and Tools*. Journal of Computer Languages, 64, 101058. https://doi.org/10.1016/j.cola.2021.101058
11. DAOstack. (2020). *Decentralized Governance Protocols for Collective Intelligence*. https://daostack.io/
12. Polkadot Web3 Foundation. (2023). *Polkadot Governance Overview*. https://wiki.polkadot.network/docs/learn-governance
13. Ellul, J., & Pace, G. J. (2018). *The runtime verification of Ethereum smart contracts*. In Formal Methods for Blockchain and Smart Contracts (pp. 119–138). Springer. https://doi.org/10.1007/978-3-030-31193-6_7
14. Kiffer, L., Levin, D., & Mislove, A. (2018). *Stick a fork in it: Analyzing the Ethereum network partition*. In Proceedings of the 16th ACM Workshop on Hot Topics in Networks. https://doi.org/10.1145/3286062.3286084