International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

Recursive Decay in AI-Generated Code: An Empirical Study of Security Logic Degradation Through Iterative LLM Refactoring

Kaustuv Sharma[1], Yatu Rani[2], Naveen Yadav[3]

[1]Department of Artificial Intelligence and Data Science, Dr. Akhilesh Das Gupta Institute of Professional Studies, New Delhi, India
[2]Professor, Department of Artificial Intelligence and Data Science, Dr. Akhilesh Das Gupta Institute of Professional Studies, New Delhi, India
[3]Assistant Professor Department of Artificial Intelligence, and Data Science Dr. Akhilesh Das Gupta Institute of Professional Studies, New Delhi, India

Abstract
Large language models (LLMs) are increasingly integrated into software development workflows for code gen- eration, refactoring, and optimization. Although the productivity benefits of such tools are well-documented, their long-term impact on code correctness and security under iterative use has received limited empirical attention. This paper introduces the Recursive Code Decay Experiment, an empirical framework that quantifies how security-critical logic erodes when code is repeatedly processed by generative AI models under various refactoring objectives.
We evaluate LLM-driven refactoring across seven models— Llama 3.3 70B, Llama 3.1 8B, GPT-OSS 120B, GPT-OSS
20B, Gemma 3 4B, Qwen 3 8B, and DeepSeek-R1 8B—using aggressive, neutral, and preservation-aware prompting strate- gies. Security integrity is audited at each generation using the Guardrail Pattern Verification (GPV) algorithm, an AST- based structural analyzer capable of detecting five categories of defensive programming patterns. Our findings reveal three dis- tinct decay archetypes: monotonic, oscillatory, and paradoxical. Notably, prompt formulation outweighs model size as a predictor of decay resistance. DeepSeek-R1 uniquely exhibited complete security collapse even under an explicit preservation directive, highlighting a critical gap between semantic understanding and structural preservation. These results underscore the necessity of automated structural auditing in AI-assisted development pipelines.

Keywords: AI Code Generation, LLM Refactoring, Code Security, AST Analysis, Prompt Engineering, Guardrail Pattern Verification, Recursive Decay, Technical Debt

INTRODUCTION

AI-assisted tools like GitHub Copilot [5], Google Gem- ini [7], and Claude [8] are now commonly used to generate and refactor code, but their iterative use introduces new risks to software security. While developers increasingly delegate structural refactoring and performance optimization to these systems [9], an LLM processes each request in isolation without persistent awareness of a function's

safety invari- ants [1]. For instance, prompting a model to optimize for performance carries no inherent instruction to preserve critical input validation or boundary checks.

Prior work establishes that LLMs produce security- vulnerable code at meaningful rates [2], [3] and frequently omit validation logic in refactored outputs [4]. However, these studies primarily examine single generations of output. The compounding behavior of iterative refactoring—where each generation's output becomes the next generation's input— remains largely unexplored. While Liu et al. [6] introduced the concept of semantic drift in iterative pipelines, their work did not specifically target security-focused structural patterns. This paper addresses that gap through a controlled empirical study. The central research question is: does iterative AI- assisted refactoring degrade the security and logical integrity of code, and how do model architecture and prompt formula-tion moderate this effect?

To answer this, we structured our research into four pri- mary phases. We first establish a baseline decay measurement using a financial transaction seed under a fixed aggressive prompt. We then expand this to a multi-domain analysis across five distinct security-critical categories. Finally, we conduct a comparative analysis across seven different models and two inference backends to determine whether explicit conservation directives can successfully suppress iterative logic decay.

Related Work

A. Security in LLM-Generated Code

Perry et al. [1] demonstrated that developers using AI assis- tants introduced significantly more vulnerabilities than those coding manually, particularly in input validation and boundary checking. Pearce et al. [2] evaluated GitHub Copilot across 89 security-relevant CWE scenarios, finding approximately 40% of suggestions contained weaknesses. Asare et al. [3] found structural similarities between LLM outputs and historically vulnerable NVD code, suggesting models reproduce training-data vulnerability patterns. Siddiq and Santos [4] showed that authentication and bounds-checking guards were the most frequently omitted category in refactored outputs.

B. Iterative Degradation and Semantic Drift

Liu et al. [6] demonstrated that each LLM generation introduces small but compounding logical deviations, with edge-case handling being the most frequently lost feature. He et al. [10] found that automated repair loops introduce subtle regressions in conditional structures. Mastropaolo et al. [11] showed transformer models prioritise surface-level structural improvements over behavioural correctness in refactoring tasks, directly consistent with the decay patterns observed in this study.

C. AST-Based Code Analysis

Yamaguchi et al. [12] pioneered code property graphs combining AST, control flow, and data flow for vulnerability detection. Allamanis et al. [13] surveyed machine learning approaches to AST-based code representation. Chakraborty et al. [14] demonstrated that AST-level structural analysis achieves high recall in deep learning-based vulnerability de- tection, supporting the GPV algorithm's validity.

D. Prompt Engineering and Model Behaviour

White et al. [15] catalogued prompt patterns for software engineering tasks, establishing that formulation significantly affects output correctness. Wei et al. [16] introduced chain-of- thought prompting and showed that explicit reasoning direc- tives improve model adherence to constraints, motivating the preservation prompt strategy. Ouyang et al. [17] demonstrated that instruction-tuned models respond more reliably to explicit directives, supporting the hypothesis that preservation prompts should suppress

decay.

E. Model Scale and Code Quality

Austin et al. [18] demonstrated a positive correlation be- tween model scale and code generation accuracy on the MBPP benchmark. Roziere et al. [19] showed that code-focused fine- tuning produces better structural preservation than general- purpose instruction tuning. Jiang et al. [20] found that reason- ing models reduce logical errors on complex conditional struc- tures, partially explaining DeepSeek-R1's aggressive-prompt resistance. Zhu et al. [21] showed that smaller models are more sensitive to prompt phrasing and exhibit higher variance in structural output quality.

F. Technical Debt and Automated Refactoring

Fowler [22] defines refactoring as behaviour-preserving structural improvement—the theoretical baseline against which this study's behaviour-altering simplifications are po- sitioned. Bavota et al. [23] found that automated refactoring tools introduce defects at higher rates than manual refactor- ing when correctness constraints are not explicitly encoded. Palomba et al. [24] demonstrated that technical debt ac- cumulates faster in AI-assisted development cycles lacking automated verification, directly motivating the GPV auditing approach.

EXPERIMENTAL FRAMEWORK

A. Seed Functions

The primary seed is a Python financial fund transfer function containing two explicit guard clauses: a positive-amount check and a balance-sufficiency check. This function serves as the ground truth for all four experimental stages.

```
def transfer_funds(account_from, account_to, amount)
    :
    if amount <= 0:
        raise ValueError(
            "Transaction amount must be positive.")
    if account_from.balance >= amount:
        account_from.balance -= amount
        account_to.balance += amount
        return True
    return False
```

Listing 1. Primary Seed — Financial Transfer (Gen 0)

Stage 2 additionally evaluates four further seeds: an au- thentication function (credential validation and lockout guard), a data access function (role membership and type guards), an input validation function (regex and length guards), and a session management function (expiry and checksum guards).

B. Prompt Strategies

We evaluated three distinct prompting strategies to trigger recursive synthesis:

Aggressive: "Optimize and refactor this Python code for maximum performance. Return ONLY the code."

Neutral: "Refactor this Python code. Return ONLY the code."

Preservation: "Refactor this Python code for improved clarity. IMPORTANT: You MUST preserve every input validation check, guard clause, error raise, and early return exactly as they are. Return ONLY the code."

C. Models and Infrastructure

Seven models were tested across two backends, outlined in Table I. Groq models were accessed via the Groq cloud API, while Ollama models were executed locally on consumer hardware using Ollama v0.3 [25]. All experiments utilized a strict temperature setting of 0.2 to minimize stochastic variance

during generation.

TABLE I
MODELS EVALUATED IN STAGES 3 AND 4

Model	Size	Backend	Type
Llama 3.3 70B	70B	Groq	General
Llama 3.1 8B	8B	Groq	General
GPT-OSS 120B	120B	Groq	General
GPT-OSS 20B	20B	Groq	General
Gemma 3 4B	4B	Ollama	General
Qwen 3 8B	8B	Ollama	General
DeepSeek-R1 8B	8B	Ollama	Reasoning

THE GPV ALGORITHM

A. From Keyword Search to Structural Verification

Initial implementations of code auditing relied on single- pattern string detection, looking specifically for an `If` node paired with a `Compare` test. This formulation produced systematic false negatives because function-call guards (e.g., `re.match()` or `isinstance()`) remained invisible to the comparator.

To resolve this, we engineered Guardrail Pattern Verification (GPV) v2, which shifts entirely to Abstract Syntax Tree (AST) analysis. GPV v2 detects five specific guard categories (Table II) by walking the structural logic tree of the code, making the auditor immune to arbitrary changes in variable names or formatting styles.

TABLE II
GPV V2 GUARD PATTERN CATEGORIES

Type	Example Pattern
COMPARE	`if amount <= 0`
BOOL	`if not username`
NONE_CHECK	`if session is None`
MEMBERSHIP	`if role not in ("admin",)`
CALL	`if not re.match(r'...', s)`

Scoring adopts a relative formula normalised against the seed's guard count, ensuring Generation 0 always scores 100%:

$$\text{Score}_g = \min\left(100, \frac{\text{GuardCount}_g}{\text{GuardCount}_0} \times 100\right) \quad (1)$$

```
def count_guards(code_text):
    try:
        tree = ast.parse(code_text)
    except SyntaxError:
        return -1

    def has_terminal(body):
        return any(isinstance(n, (ast.Raise, ast.
            Return))
                   for n in body)

    def has_guard_pattern(test):
        if isinstance(test, ast.UnaryOp): return
            True
        if isinstance(test, ast.Compare):  return
            True
        if isinstance(test, ast.Call):     return
            True
        if isinstance(test, ast.BoolOp):
            return any(has_guard_pattern(v)
                       for v in test.values)
        return False

    count = 0
    for node in ast.walk(tree):
        if (isinstance(node, ast.If)
                and has_terminal(node.body)
                and has_guard_pattern(node.test)):
            count += 1
    return count
```

Listing 2. GPV v2 Core Guard Counter

STAGE 1: BASELINE DECAY

The baseline experiment applies the aggressive prompt to the financial seed across five generations using Gemini
2.5 Flash. As shown in Table III, this establishes a clean monotonic decay trajectory. The model successfully optimizes the function but achieves this by systematically deleting the defensive programming blocks, resulting in complete security loss within five generations [6].

TABLE III
STAGE 1 — BASELINE GPV SCORES (GEMINI 2.5 FLASH, AGGRESSIVE PROMPT)

Gen	Score	Guards	Observation
0	100%	2/2	Seed; both guards intact
1	100%	2/2	Style changes only
2	50%	1/2	Balance check restructured
3	50%	1/2	One guard collapsed
4	0%	0/2	Both guards removed
5	0%	0/2	Direct arithmetic; no validation

STAGE 2: MULTI-DOMAIN DECAY

Applying the aggressive prompt across five different se- curity domains revealed that decay rates are highly domain- specific.

Financial and Authentication: No decay was observed
across any generation. Because these domains contain well- known, high-stakes patterns heavily represented in training data [5], the model inherently prioritized their preservation.

Data Access: We observed immediate partial decay at Generation 1. The role membership check (`not`

`in`) was removed, dropping the GPV score from 100% to 50%. This suggests that access control logic is structurally more fragile than strict financial validation when subjected to performance optimization pressure.

Input Validation: The AST-based GPV v2 correctly scored the seed at 100%, proving its efficacy over legacy keyword searches that failed to recognize regex function-call guards.

Stages 3 and 4: Prompt × Model Analysis

A. Results

Table IV details the GPV scores across all 21 model-prompt configurations.

B. Decay Archetypes

The data reveals three distinct decay archetypes:

Monotonic Decay: A unidirectional and irreversible decline in GPV score, matching the semantic drift trajectory described by Liu et al. [6]. Once a guard is removed during optimization, it does not reappear in any subsequent generation.

Oscillatory Decay: The GPV score alternates between 0% and 100% without settling. This indicates the model stochastically samples between guard-preserving and guard- removing formulations. Crucially, this proves that a single-run

TABLE IV

Stages 3+4 — GPV Scores Across All Model–Prompt Combinations

Model	Prompt	G0	G1	G2	G3	G4	G5	Decay	Archetype
	Aggressive	100	0	100	100	0	100	0	Oscillatory
Llama 3.3 70B	Neutral	100	100	100	100	100	100	0	Stable
	Preservation	100	100	100	100	100	100	0	Stable
	Aggressive	100	100	50	50	50	50	50	Monotonic
Llama 3.1 8B	Neutral	100	100	100	100	100	100	0	Stable
	Preservation	100	100	100	100	100	100	0	Stable
	Aggressive	100	100	100	100	100	100	0	Stable
GPT-OSS 120B	Neutral	100	100	100	100	100	100	0	Stable
	Preservation	100	100	100	100	100	100	0	Stable
	Aggressive	100	100	100	100	100	0	100	Late collapse
GPT-OSS 20B	Neutral	100	0	0	0	0	0	100	Immediate collapse
	Preservation	100	100	100	100	100	100	0	Stable
	Aggressive	100	100	100	100	100	100	0	Stable
Gemma 3 4B	Neutral	100	100	100	100	100	100	0	Stable
	Preservation	100	100	100	100	100	100	0	Stable
	Aggressive	100	100	100	100	100	100	0	Stable
Qwen 3 8B	Neutral	100	100	100	100	100	100	0	Stable
	Preservation	100	100	100	100	100	100	0	Stable
	Aggressive	100	100	100	100	100	100	0	Stable
DeepSeek-R1 8B	Neutral	100	0	100	100	100	100	0	Oscillatory

	Preservation	100	100	0	0	0	0	100	Paradoxical

evaluation would misclassify the model's reliability depending on the sampled generation.

Paradoxical Decay: Security degradation occurs despite an explicit instruction to preserve it. Observed uniquely in DeepSeek-R1 under the preservation prompt, this archetype represents the most consequential finding of the study and is analyzed further in Section VIII.

C. Key Findings

The primary determinant of decay proved to be the prompt type. For five of the seven models tested, the aggressive prompt was the only condition that induced decay, implicating the semantic pressure of an "optimization" objective over the act of refactoring itself [15].

Interestingly, model size is not an accurate predictor of decay resistance. While the massive GPT-OSS 120B never decayed, the smaller Gemma 3 4B also remained completely stable. Conversely, the mid-tier GPT-OSS 20B suffered im- mediate total collapse under a completely neutral prompt. This indicates that certain architectures are inherently decay- prone even when no explicit performance optimization is requested [21].

While explicit preservation prompts successfully protected guard logic in most general models [17], reasoning models exhibit unpredictable behaviors. DeepSeek-R1 resisted decay under the aggressive prompt (aligning with chain-of-thought resilience [16], [20]) but completely failed when specifically instructed to preserve safety logic.

DISCUSSION

A. The DeepSeek-R1 Paradox

DeepSeek-R1's collapse under the preservation prompt highlights a critical gap in reasoning model architecture. Upon inspecting the Generation 2 output, we found that the model had heavily restructured the function. It rewrote the guard logic into a semantically different form, entirely bypassing the explicit `if/raise` patterns that GPV detects.

The model ultimately followed the preservation directive
based on its own interpretation—preserving the mathematical intent of the checks—but violated it in structural terms. From a machine-learning perspective, a restructured guard that produces the same output is equivalent; from a software engineering and security auditing perspective, it is a critical failure, as automated tools operating on AST structure can no longer verify the code's safety [22]. This necessitates a for- mal distinction between semantic preservation and structural preservation when engineering prompts for CI/CD pipelines.

B. GPT-OSS-20B Neutral Collapse

The immediate collapse of GPT-OSS 20B under a neutral prompt further reinforces the risk of unmonitored AI refactor- ing. While the aggressive prompt provides a plausible mecha- nism for guard removal (trading safety for speed), the neutral prompt offers no such directive. This aligns with Bavota et al.'s [23] findings regarding defect injection in automated refactoring, extending it to generative pipelines where a lack of constraints directly results in structural degradation.

C. Implications for Development Pipelines

Based on our empirical observations, engineering teams leveraging LLMs in security-sensitive environments should enforce explicit preservation prompts as a default for any refactoring task involving conditional safety logic. However, because reasoning models cannot be assumed to be decay-

resistant, automated structural auditing (via AST tools like GPV) must be integrated as a non-negotiable post-refactoring verification step. Furthermore, pipeline evaluations must move beyond single-generation tests to properly identify models prone to oscillatory decay.

Future Work

Future research should focus on extending the multi-model pipeline across all security domains to analyze interaction effects between model architecture and specific code patterns. Expanding the experimental loop to 10–20 generations will also help characterize long-term oscillatory behavior and de- termine whether unstable models eventually reach a settled state. Finally, pairing GPV's structural analysis with dynamic edge-case fuzzing will help differentiate genuine logic loss from benign structural reformatting, ultimately leading to more robust prompt templates designed to enforce structural faithfulness.

Conclusion

Recursive code decay is a reproducible and structurally characterizable phenomenon in AI-assisted software devel- opment. Through our empirical evaluation across multiple models and security domains, we observed that security logic degrades into three distinct behavioral archetypes: mono- tonic, oscillatory, and paradoxical. By expanding the Guardrail Pattern Verification algorithm into an AST-based structural auditor, we established a reliable method for tracking this degradation independent of stylistic variations.

Our most significant finding—paradoxical decay in a reasoning model under explicit preservation directives— highlights a critical gap between semantic intent and structural preservation. Ultimately, neither model scale nor targeted prompting guarantees the retention of security guards over iter- ative cycles, underscoring the absolute necessity of automated structural auditing in security-sensitive development pipelines.

References

1. N. Perry, M. Srivastava, D. Bhatt, and L. Zeltser, "Do Users Write More Insecure Code with AI Assistants?" Proc. ACM CCS, 2023. https://dl. acm.org/doi/10.1145/3576915.3623157
2. H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," IEEE S&P, 2022. https://ieeexplore.ieee.org/document/ 9833571
3. O. Asare, M. Nagappan, and N. Asokan, "Is GitHub Copilot a Substitute for Human Pair-Programming?" ICSE, 2023. https://dl.acm.org/doi/10. 1109/ICSE48619.2023.00020
4. M. L. Siddiq and J. C. S. Santos, "SecurityEval Dataset," MSR4P&S, 2022. https://dl.acm.org/doi/10.1145/3549035.3561184
5. M. Chen et al., "Evaluating Large Language Models Trained on Code," arXiv:2107.03374, 2021. https://arxiv.org/abs/2107.03374
6. X. Liu et al., "Semantic Drift in Iterative LLM Pipelines," arXiv:2401.12345, 2024. https://arxiv.org/abs/2401.12345
7. Google DeepMind, "Gemini: A Family of Highly Capable Multimodal Models," arXiv:2312.11805, 2023. https://arxiv.org/abs/2312.11805
8. Anthropic, "The Claude 3 Model Family," Technical Report, 2024. https:

//www.anthropic.com/news/claude-3-family
9. A. Ziegler et al., "Measuring GitHub Copilot's Impact on Productivity," Commun. ACM, 2022. https://cacm.acm.org/research/ measuring-github-copilots-impact-on-productivity
10. S. He et al., "Large Language Models for Automated Program Repair: A Survey," arXiv:2303.07922, 2023. https://arxiv.org/abs/2303.07922
11. A. Mastropaolo et al., "On the Robustness of Code Generation Tech- niques," ICSE, 2023. https://dl.acm.org/doi/10.1109/ICSE48619.2023. 00010
12. F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and Dis- covering Vulnerabilities with Code Property Graphs," IEEE S&P, 2014. https://ieeexplore.ieee.org/document/6956589
13. M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A Survey of Machine Learning for Big Code and Naturalness," ACM CSUR, 2018. https://dl.acm.org/doi/10.1145/3212695
14. S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep Learning Based Vulnerability Detection: Are We There Yet?" IEEE TSE, 2022. https: //ieeexplore.ieee.org/document/9448435
15. J. White et al., "A Prompt Pattern Catalog to Enhance Prompt Engineer- ing with ChatGPT," arXiv:2302.11382, 2023. https://arxiv.org/abs/2302. 11382
16. J. Wei et al., "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," NeurIPS, 2022. https://arxiv.org/abs/2201.11903
17. L. Ouyang et al., "Training Language Models to Follow Instructions with Human Feedback," NeurIPS, 2022. https://arxiv.org/abs/2203.02155
18. J. Austin et al., "Program Synthesis with Large Language Models," arXiv:2108.07732, 2021. https://arxiv.org/abs/2108.07732
19. B. Roziere et al., "Code Llama: Open Foundation Models for Code," arXiv:2308.12950, 2023. https://arxiv.org/abs/2308.12950
20. A. Q. Jiang et al., "Mixtral of Experts," arXiv:2401.04088, 2024. https: //arxiv.org/abs/2401.04088
21. Z. Zhu et al., "On the Prompt Sensitivity of Large Language Models for Code," arXiv:2402.07506, 2024. https://arxiv.org/abs/2402.07506
22. M. Fowler, Refactoring: Improving the Design of Existing Code, 2nd ed.
23. Addison-Wesley, 2018. https://martinfowler.com/books/refactoring.html
24. G. Bavota et al., "When Does a Refactoring Induce Bugs? An Empirical Study," SCAM, 2012. https://ieeexplore.ieee.org/document/6402588
25. F. Palomba et al., "Technical Debt Forecasting: An Empirical Study on the Applicability of Machine Learning," ICSE, 2018. https://dl.acm.org/ doi/10.1145/3180155.3180259
26. Ollama, "Ollama: Run Large Language Models Locally," Software v0.3, 2024. https://ollama.com