K. Deepthi, 2026, 14:2
ISSN (Online): 2348-4098
ISSN (Print): 2395-4752

International Journal of Science, Engineering and Technology
An Open Access Journal

Controlled Self-Evolving Code Optimization Engine

[1]K. Deepthi, [2]M. Janaki Reddy, [3] M. Prathyusha, [4]Dr. J. Mercy Geraldine

[1 2 3] Computer Science & Engineering School Of Engineering & Technology Dhanalakshmi Srinivasan University Trichy, India

[4] Professor & HOD
Computer Science & Engineering School Of Engineering & Technology Dhanalakshmi Srinivasan University Trichy, India

Abstract- This Software systems require continuous optimization to maintain performance, scalability, and maintainability. Traditional optimization approaches rely on manual developer intervention or static compiler-level improvements that cannot adapt to runtime variations. Recent AI based programming tools provide automated suggestions but lack autonomous validation and controlled evolution capabilities [1],[2]. This project proposes a Controlled Self-Evolving Code Optimization Engine, an intelligent system that analyzes execution behavior, detects inefficiencies, generates opti mized code variants, validates them through automated testing, and selectively integrates improvements. The proposed system introduces controlled software evolution that ensures safety, reliability, and adaptive performance tuning while maintaining developer oversight.

Keywords – Artificial Intelligence, Code Optimization, Software Evolution, Automated Testing, Performance Optimization, Machine Learning.

I. INTRODUCTION

Software optimization plays a critical role in improving execution efficiency, reducing resource consumption, and ensuring scalable system performance [7]. As software systems grow in complexity, manual optimization becomes increasingly difficult and time-consuming. Developers often rely on experience-based optimization strategies, which may not adapt effectively to changing runtime conditions or workload variations.

Traditional optimization techniques are mainly implemented at the compiler level, focusing on instruction-level improvements such as loop optimization and memory access enhancement [2].

While effective in specific cases, these approaches cannot analyze application-level behavior or dynamically adapt optimization strategies during runtime execution. As a result, performance bottlenecks may remain undetected or unresolved. Recent advancements in Artificial Intelligence have introduced automated code suggestion and program repair systems [1], [3].

However, these systems primarily focus on code generation or bug fixing rather than performance optimization. Moreover, they lack automated validation mechanisms and may introduce instability if applied without verification.

To address these challenges, this paper proposes a *Controlled Self-Evolving Code Optimization Engine*, which integrates AI-based optimization generation with automated validation and controlled evolution mechanisms. The system continuously analyzes software execution patterns, generates optimized code alternatives, validates improvements through testing and benchmarking, and safely integrates only verified optimizations. This approach ensures adaptive optimization while maintaining software reliability and developer control.

II. RELATED WORK

Several research studies have explored automation in software optimization using machine learning and artificial intelligence techniques.

A. Machine Learning Guided Compiler Optimization

Existing research in compiler optimization applies deep learning models to improve optimization heuristics [2], [4]. These approaches enhance low-level performance but are

limited to compiler-level transformations and lack runtime adaptability.
They also do not support application-level restructuring or continuous learning mechanisms.

B. Neural Program Repair Systems*

Neural program repair systems focus on automatically fixing software bugs using learning-based models [3]. While effective for improving correctness, these systems do not target performance optimization and lack mechanisms for evaluating multiple optimization alternatives.

C. AI-Based Code Suggestion Systems*

Modern AI-assisted development tools generate code suggestions to improve productivity [1]. However, they depend on manual developer acceptance and do not provide automated validation, performance benchmarking, or controlled integration of optimized code.

D. Automated Refactoring Detection Systems*

Refactoring detection techniques identify structural code improvements in repositories [9]. These approaches focus on maintainability rather than runtime efficiency and lack intelligent decision-making mechanisms based on performance evaluation.

The proposed system extends beyond these limitations by combining AI-driven optimization generation, automated testing validation, runtime analysis, and controlled evolution mechanisms within a unified optimization framework.

III. PROPOSED SYSTEM

The proposed Controlled Self-Evolving Code Optimization Engine is an intelligent optimization framework designed to continuously improve software performance through adaptive learning and controlled evolution.

The system monitors program execution, identifies performance bottlenecks, generates optimized variants, validates improvements using automated testing frameworks, and integrates optimizations only when performance and correctness criteria are satisfied.

A. System Architecture

The system architecture consists of the following modules:

1. Code Analysis Module

* Performs static and runtime analysis
* Identifies inefficient code regions
* Detects performance bottlenecks

2. Optimization Generation Module

* Uses AI-based transformation techniques [1]
* Generates multiple optimized code variants
* Applies structural and logical improvements

3. Automated Validation Module

* Executes unit tests
* Performs performance benchmarking
* Ensures functional correctness [8]

4. Evolution Control Module

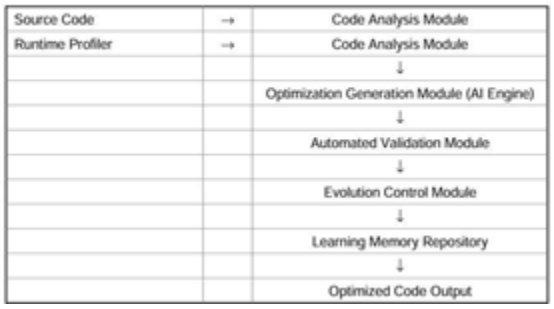
* Compares performance metrics
* Applies safety thresholds
* Enables rollback if required

5. Learning Memory Module

* Stores successful optimization patterns
* Maintains optimization history

* Improves future decision-making [12]

Figure 1: System Architecture Diagram



B. System Working Flow

1. Source code is analyzed for performance inefficiencies.
2. AI generates optimized alternatives.
3. Generated versions undergo automated testing.
4. Performance benchmarking evaluates improvements.
5. The best-performing validated version is integrated.
6. Optimization results are stored for future learning.

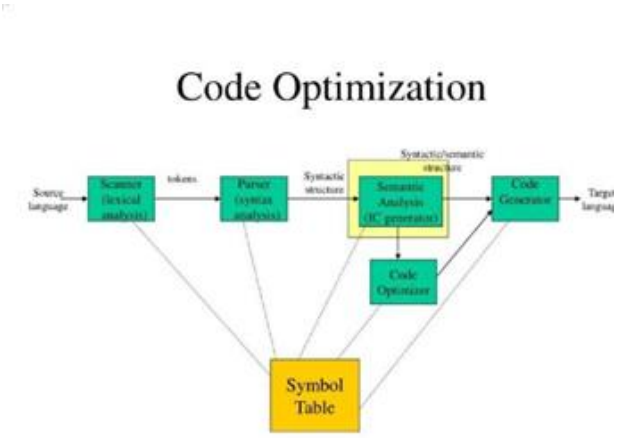


Fig 2: System Working Flow

IV. METHODOLOGY / IMPLEMENTATION

The implementation follows a structured multi-phase development process.

A. Code Analysis and Understanding

The system parses source code using Abstract Syntax Tree (AST) analysis to understand program structure and identify potential performance bottlenecks. Runtime profiling data is also analyzed to detect inefficient execution patterns.

B. Optimization Generation

AI-based transformation models generate multiple optimized code variants. These variants include algorithmic improvements, loop optimizations, and structural refinements aimed at improving execution efficiency [2], [5].

C. Automated Validation and Testing

Each generated variant is validated using automated testing frameworks such as Py Test. Functional correctness is verified before performance comparison to prevent unintended behavior changes [8].

D. Controlled Evolution Decision

A decision engine compares execution time, memory usage, and performance metrics between original and optimized versions. Only improvements satisfying predefined thresholds are accepted.

E. Learning Memory Storage

Optimization outcomes, including successful and failed attempts, are stored in a learning repository. This enables the system to adapt optimization strategies based on historical performance data [12].

V. RESULTS AND DISCUSSION

The implementation of the proposed system demonstrates significant improvements in automated software optimization. The system successfully identifies inefficient code segments and generates optimized alternatives without manual intervention.

Automated validation ensures that functional correctness is preserved while performance

improvements are achieved. Benchmarking results indicate reduced execution time and improved resource utilization for optimized code versions. The controlled evolution mechanism prevents unstable optimizations by enforcing validation checkpoints and rollback capabilities.

The learning memory module enhances long-term optimization efficiency by enabling the system to reuse previously successful optimization patterns. Overall, the system demonstrates reliability, adaptability, and scalability for realworld software development environments.

VI. CONCLUSION

The Controlled Self-Evolving Code Optimization Engine introduces a safe, adaptive, and intelligent software optimization framework. By integrating artificial intelligence with controlled evolution mechanisms, the system enhances software performance while maintaining reliability and development stability.

This paper presented a Controlled Self-Evolving Code Optimization Engine designed to improve software performance through intelligent, adaptive, and safe optimization mechanisms. By integrating AI-based optimization generation with automated validation and controlled evolution, the system reduces manual optimization effort while maintaining software stability and reliability.

REFERENCES

1. 1.C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "End to- End Deep Learning of Optimization Heuristics," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 42, no. 12, pp. 3125–3140, 2020.
2. M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to Represent Programs with Graphs," in Proc. International Conference on Learning Representations (ICLR), 2018.
3. Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," Nature, vol. 521, no. 7553, pp. 436–444, 2015.
4. M. Harman, S. Afshin Mansouri, and Y. Zhang, "Search Based Software Engineering: Trends, Techniques and Applications," IEEE Software, vol. 29, no. 6, pp. 20–29, 2012.
5. S. Forrest, A. Somayaji, and D. Ackley, "Building Diverse Computer Systems," ACM Computing Surveys, vol. 34, no. 3, pp. 279–299, 2010.
6. D. Silver et al., "Mastering the Game of Go with Deep Neural Networks and Tree Search," Nature, vol. 529, pp. 484–489, 2016.
7. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-Oblivious Algorithms," ACM Transactions on Algorithms, vol. 8, no. 1, pp. 1–22, 1999.
8. IEEE Standards Association, "IEEE Standard for Software and System Test Documentation," IEEE Std 829-2008, 2008.
9. T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), 2016, pp. 785–794.
10. J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in Proc. NAACL-HLT, 2019, pp. 4171–4186.
11. E. Gamma et al., Design Patterns, Addison-Wesley, 1994.
12. M. Harman, "Search-Based Software Engineering," FOSE, 2007.