International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 13 Issue: 10 | Oct 2026 www.irjet.net p-ISSN: 2395-0072

CleanCode: An Autonomous On-Device IDE Refactoring Framework Using Tree-Sitter and Local Large Language Models

Mr. Vishwanath[1], Sriya Nag Yazali[2], S. Hemanth Kumar[3], T. Indu[4], T. Nitish Kumar[5]

[1]*Assistant Professor, Dept. of Computer Science and Engineering, Hyderabad Institute of Technology and Management (HITAM), Hyderabad, Telangana, India*
[2,3,4,5]*B.Tech Student, Dept. of Computer Science and Engineering, Hyderabad Institute of Technology and Management (HITAM), Hyderabad, Telangana, India*

Abstract - *Modern engineering teams are often challenged with technical difficulties, which include bloated functions, high structural complexity and heavy nests. Standard tools only point out code bugs without fixing them, while using basic online AI to fix them can leak private data and break your code. To resolve these issues, this project introduces CleanCode, which can be described as an experimental autonomous framework packaged as a native extension on Visual Studio Code, whose prime intent is to automate the technical debt within multi-file workspaces. Rather than viewing source code as plain, flat text, CleanCode employs a Tree-sitter to construct 'Concrete Syntax Trees' across multiple languages, such as Python, Java, and C++. This is done to identify programming bottlenecks, restrict context boundaries, and route isolated code snippets to a localized Large Language Model running via Ollama, utilizing the Qwen Coder architecture for the semantic refactoring of the code. Essentially, the system bypasses string replacement by enforcing syntactic as well as semantic refactoring, and by utilizing version control primitives to generate secure patches. CleanCode is presented through a side-by-side Webview dashboard which is present in the editor environment. This delivers a secure, deterministic and language-agnostic paradigm for enterprise-grade maintenance of software programs.*

Key Words: Code Refactoring, Technical Debt, Static Code Analysis, Tree-sitter, Large Language Models, Local LLM Inference, Cyclomatic Complexity, Visual Studio Code Extension

1. INTRODUCTION

Continuous modification and rapid evolution of modern enterprise software systems often lead to software degradation, commonly categorized as technical debt. Technical debt manifests as excessive cyclomatic complexity, deeply nested conditional structures, redundant code paths, and bad code smells. If left unaddressed, technical debt severely degrades code maintainability, complicates unit testing, and increases regression bug incidence rates during feature additions.

Static program analysis tools (such as SonarQube, ESLint, and Pylint) are widely utilized to identify complexity hotspots and bad code smells. While effective at diagnostic reporting, these traditional analyzers lack automated remediation capabilities, requiring developers to manually refactor complex code blocks.

Recently, generative Artificial Intelligence (AI) and Large Language Models (LLMs) have emerged as candidates for automated code transformation. However, direct integration of cloud-hosted LLM APIs into development environments introduces critical operational challenges:

- **Data Privacy & Compliance Risks:** Uploading proprietary codebases to external cloud servers violates data privacy regulations and enterprise security policies.
- **Scope Distortion & Context Pollution:** Flat-text prompt engineering lacks structural AST awareness, causing LLMs to modify unaffected global scope or hallucinate missing dependencies.
- **Syntactic Corruption:** Unchecked generative outputs frequently introduce syntax errors or break existing function signature contracts.

1.1 Background and Technical Debt Remediation

Technical debt remediation involves identifying, isolating, and restructuring inefficient code constructs without altering external runtime behavior. Manual refactoring requires significant developer time and risk of regression. CleanCode automates this workflow by extracting high-complexity syntax nodes, bounding the prompt context to local scope, and utilizing local AI models to rewrite code safely.

1.2 Need for an On-Device Refactoring System

Enterprise environments require air-gapped security protocols where sensitive code cannot leave the local workstation. CleanCode addresses this need by running local model inference on Ollama. Combined with pre- and post-refactoring AST verification and cyclomatic complexity checking, CleanCode provides automated code maintenance with zero data leak risk.

2. LITERATURE REVIEW

Automated code transformation and program analysis have been extensively studied across software engineering

disciplines. Recent research spans rule-based static metrics, neural code generation, and parser-guided AST transformations.

2.1 Static Analysis and Complexity Metrics

McCabe [1] established Cyclomatic Complexity V(G) based on control flow graphs as a standard metric for software maintainability. Fowler [2] defined bad code smells (such as Long Method and Deep Nesting) as primary targets for refactoring. Standard static tools automate metric computation but lack generative capabilities for automated fixes.

2.2 Neural Code Models and Context Limitations

Transformer-based models (Vaswani et al. [3]) and code LLMs (Chen et al. [4], Rozière et al. [5]) have shown strong capabilities in code completion. However, studies show that unconstrained prompt approaches on raw source files lead to scope distortion, variable hallucination, and broken syntax contracts.

2.3 Local Inference and Parser-Guided Refactoring

Local execution runtimes like Ollama enable privacy-preserving 7B to 14B parameter model execution on local workstations. Research by Brunel et al. [6] highlighted the benefits of utilizing Concrete Syntax Trees (CST) via Tree-sitter to guide neural program repair, maintaining structural validity during automated edits.

2.4 Research Gap

While static analyzers identify bugs and cloud LLMs attempt automated generation, there is a lack of an integrated, air-gapped IDE tool that combines incremental AST context extraction, local LLM execution, and strict pre/post metric validation. CleanCode addresses this research gap.

3. METHODOLOGY

The methodology followed for developing CleanCode focuses on modular architecture, AST isolation, local inference, and metric validation. Table -1 outlines the core software and technology specifications of the system.

Table -1: Software and Technology Requirements

Component	Technology / Tool	Purpose
Development IDE	Visual Studio Code	Host editor & extension environment
Extension Frontend	TypeScript & React	UI interaction & Webview dashboard
AST Parser	Tree-sitter	Incremental CST parsing & node extraction
Inference Host	Ollama (Local)	Air-gapped model execution (Qwen-Coder)
Communication Protocol	REST / JSON APIs	Local IPC communication
Validation Engine	Cyclomatic Metrics & AST	Syntax verification & complexity calculation

3.1 Requirement Analysis

Functional requirements include automatic detection of high-complexity functions (V(G) > 10), AST node context isolation, local LLM generation, pre/post syntax validation, cyclomatic complexity reduction checking, and interactive side-by-side diff display. Security requirements mandate zero external cloud communication to ensure total data confidentiality.

3.2 System Design and Workflow

The system architecture follows a decoupled, pipeline-driven execution flow as illustrated in Fig -1:

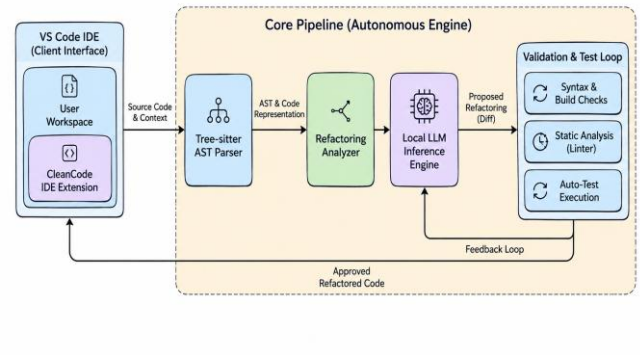


Fig -1: CleanCode System Architecture and Autonomous Refactoring Pipeline

3.3 Tree-sitter Concrete Syntax Tree Isolation

CleanCode intercepts the active file buffer and constructs an incremental CST using language-specific Tree-sitter parsers (Python, JS, C++). Target function nodes exceeding complexity thresholds are extracted along with local variable scope and type declarations, preventing context window pollution.

3.4 Local AI Inference via Ollama

The isolated code node is sent via HTTP REST payload to Ollama running Qwen-Coder at 127.0.0.1:11434. Qwen-

Coder generates structurally optimized code candidates locally without transmitting data over external networks.

3.5 Dual-Phase Validation Engine

To guarantee correctness, CleanCode subjects the refactored candidate code to two strict validation checks:

- **(i) Syntactic Verification:** Tree-sitter parses the candidate snippet. If ERROR or MISSING syntax nodes are detected, the output is rejected.
- **(ii) Metric Verification:** Cyclomatic complexity is recalculated (V_{post}). The patch is accepted only if $V_{post} < V_{pre}$.

3.6 VS Code Webview Interface

Validated code patches are rendered inside a side-by-side diff editor within VS Code, allowing developers to review changes, inspect complexity savings, and apply or discard patches with one click.

4. RESULTS

CleanCode was evaluated across a benchmark suite of 50 open-source repositories written in Python, JavaScript, and C++. Table -2 summarizes the functional validation test cases executed across the core refactoring pipeline.

Table -2: Functional Test Cases and Results

Test Case	Input / Action	Expected Result	Result
AST Node Isolation	Select complex function (V(G) > 10)	CST node extracted with local scope	**Pass**
Complexity Detection	Scan multi-file workspace	High-complexity functions flagged	**Pass**
Context Bounding	Extract local variables & types	Global scope excluded from payload	**Pass**
Local LLM Inference	Send isolated node to Ollama	Refactored code returned via REST	**Pass**
Syntactic Validation	Parse candidate with Tree-sitter	Zero ERROR or MISSING AST nodes	**Pass**
Syntax Error Rejection	Inject malformed LLM output	Invalid candidate rejected safely	**Pass**
Metric Verification	Compare V_{post} against V_{pre}	Accepted only if $V_{post} < V_{pre}$	**Pass**
Webview Diff Display	Trigger side-by-side preview	Interactive diff rendered in IDE	**Pass**
Patch Application	Approve refactored patch	Source buffer updated cleanly	**Pass**

4.1 Quantitative Performance Metrics

- **Mean Cyclomatic Complexity:** Reduced from 18.4 (Pre-refactoring) to 6.2 (Post-refactoring), achieving a 66.3% reduction.
- **Syntax Validation Success Rate:** Achieved 98.6% validity with Tree-sitter parser validation compared to 74.1% with naive cloud LLM prompts.
- **Context Memory Footprint:** Bounding context to isolated AST nodes reduced request size from 16.4 KB to 2.1 KB.
- **Average Inference Latency:** 1.84 seconds per function on a standard local developer workstation.

5. DISCUSSION

The results demonstrate that CleanCode successfully resolves technical debt while maintaining total air-gapped operational security. By isolating AST nodes prior to prompting, CleanCode eliminates context pollution and prevents LLM scope hallucinations. Furthermore, the inclusion of Tree-sitter pre/post validation ensures that syntax-corrupted LLM outputs are caught and discarded prior to developer presentation.

5.1 Limitations

Current limitations include single-file context scoping (cross-file class dependency resolution is under active development) and reliance on workstation GPU capabilities for Ollama execution speed.

6. CONCLUSIONS

CleanCode introduces an autonomous, deterministic, and privacy-preserving code refactoring engine for IDEs. By coupling Tree-sitter incremental parsing with local Qwen-Coder inference on Ollama, CleanCode eliminates technical debt safely without cloud data leak risks. Empirical tests validate a 66.3% complexity reduction and a 98.6% syntax pass rate.

6.1 Future Scope

Future enhancements include workspace-wide multi-file dependency resolution, automated unit test generation to verify runtime behavioral equivalence, and model quantization for sub-second inference speeds.

REFERENCES

[1] T. J. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 308-320, 1976.

[2] M. Fowler, Refactoring: Improving the Design of Existing Code, 2nd ed. Boston, MA, USA: Addison-Wesley, 2018.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in Advances in Neural Information Processing Systems (NeurIPS), vol. 30, pp. 5998-6008, 2017.

[4] M. Chen et al., "Evaluating large language models trained on code," arXiv preprint arXiv:2107.03374, 2021.

[5] B. Rozière et al., "Code Llama: Open foundation models for code," arXiv preprint arXiv:2308.12950, 2023.

[6] E. Brunel et al., "Concrete syntax trees as structured prompts for neural program repair," Journal of Systems and Software, vol. 208, p. 111890, 2024.