International Journal of Research in Engineering and Science (IJRES)
ISSN (Online): 2320-9364, ISSN (Print): 2320-9356
www.ijres.org Volume 11 Issue 12 ǁ December 2023 ǁ PP. 202-208

Enhancing Code Refactoring with AI: Automating Software Improvement Processes

Anbarasu Arivoli
Email: anbarasuarivoli@gmail.com
Company: Target, Minneapolis, MN

ABSTRACT
Refactoring is a standard practice in software development aimed at improving code structure without altering its behavior. While essential for long-term maintainability, the process often demands significant time and manual effort, especially in large or legacy codebases. Recent advances in artificial intelligence offer practical methods to support and automate various stages of refactoring, from identifying code smells to applying design improvements. This paper reviews the use of machine learning models for detecting structural issues in code, explores AI techniques that recommend or implement design pattern enhancements, and compares AI-assisted refactoring with traditional manual approaches. The paper also outlines a proposed solution framework that combines automated smell detection, intelligent code transformation, and developer validation. Practical considerations such as accuracy, workflow integration, and balancing automation and control are discussed. The findings highlight how AI tools, when used carefully, can help teams improve code quality more consistently and efficiently.
Keywords: *AI-driven refactoring, code smells, machine learning, software design patterns, software maintenance*

I. INTRODUCTION

Refactoring is a common technique in software development used to enhance the internal structure of code while keeping its external functionality unchanged. This process supports better readability, simplifies complex logic, and promotes easier maintenance over time. As projects grow, the need for refactoring increases due to issues that build up over time, such as duplicated logic, poor modularity, and lack of adherence to design principles.

These issues are often described as code smells—surface-level symptoms that suggest deeper structural problems. Examples include overly large classes, long methods, and unnecessary module coupling. When left unaddressed, these problems lead to technical debt, making the codebase harder to work with and more error-prone.

Manual refactoring remains standard, typically relying on developers' experience and static analysis tools. However, this process is time-consuming and difficult to apply consistently across large or fast-changing codebases. Developers may deprioritize refactoring in favor of feature development, leading to further accumulation of technical debt.

To address these limitations, researchers and practitioners have begun to apply artificial intelligence techniques to assist with refactoring tasks. Machine learning models can be trained to detect code smells, while pattern recognition and program transformation tools can help automate or guide code restructuring. These developments point toward a more efficient approach to ongoing code improvement.

This paper explores how AI can support automated refactoring, focusing on identifying code smells, suggesting design improvements, and comparing AI-based methods with traditional approaches. A solution framework is proposed, combining detection, transformation, and developer feedback. The paper also considers the practical implications of integrating such tools into development workflows.

II. LITERATURE REVIEW

2.1 Machine Learning in Code Smell Identification

Conventional approaches to detecting code smells typically depend on static analysis tools guided by predefined rules. These tools use predefined thresholds on software metrics such as cyclomatic complexity, class length, method length, and coupling. Tools like PMD, Checkstyle, and SonarQube are widely used in practice, but their effectiveness can vary across projects and languages. One of the main challenges with these approaches is the generation of false positives and the inability to adapt to the nuances of different codebases [1].

Machine learning methods have been introduced to address these limitations. Supervised learning techniques have been used to classify code fragments as smelly or clean based on labeled datasets. Researchers have applied decision trees, support vector machines, and ensemble classifiers to detect smells like Long Method,

God Class, and Feature Envy. A systematic review by Azeem et al. [2] concluded that ML classifiers when trained on representative data, perform better than static rules in terms of both precision and recall.

More recently, deep learning approaches have gained attention. Convolutional neural networks (CNNs) and autoencoders have been trained to detect code anomalies directly from token sequences or syntax trees. These models learn latent features from the code, reducing the need for manual metric selection. Additionally, graph neural networks (GNNs) have been used to capture structural relationships within the code. For instance, Tufano et al. [3] demonstrated using deep learning models to detect and classify code smells with competitive accuracy compared to human annotators. The key advantage of these models lies in their ability to learn complex patterns and adapt to different project contexts without hardcoded thresholds.

2.2 AI for Improving Software Design Patterns

AI has also been applied to identify and suggest design pattern-based improvements. These techniques range from recommendation systems to automated pattern applications using evolutionary computation. Recommendation systems analyze code to detect patterns of usage or misuse and suggest improvements aligned with known object-oriented design principles.

Search-based software engineering (SBSE) techniques have been effective in this space. Ouni et al. [4] applied genetic algorithms to explore combinations of refactoring operations that improve cohesion and reduce coupling. These approaches evaluate refactoring sequences against fitness functions derived from quality metrics and have successfully improved multiple aspects of code design simultaneously.

Reinforcement learning has also been applied to automate the application of design patterns. In this approach, an agent learns sequences of code transformations that lead to improved designs, receiving rewards based on metrics like modularity or readability. For example, a study by Mahmoudi et al. [5] showed how a Q-learning agent could learn to introduce patterns like Strategy or Singleton where applicable. These experiments demonstrate AI's potential to detect structural issues and suggest architectural-level changes that improve long-term code maintainability.

2.3 Benefits of AI-Based Refactoring

AI-based refactoring tools provide several practical benefits. First, they allow for faster analysis of large codebases, identifying issues across thousands of files within minutes. This efficiency supports regular, ongoing maintenance rather than ad hoc clean-up. Second, AI tools offer consistency by applying uniform rules and learned patterns across the entire codebase, reducing subjective decision-making between developers.

Empirical studies support these claims. In a large-scale case study, Zaidman et al. [6] reported that AI-assisted refactoring reduced manual effort by over 40% on average in enterprise-grade systems. Similarly, Palomba et al. [7] found that automated detection and transformation tools improved maintainability metrics and reduced the density of post-release bugs. By automating repetitive and low-level refactorings, AI enables developers to focus on more complex logic and architectural concerns.

Additionally, these tools can act as educational aids. Developers reviewing AI suggestions may learn new patterns or better practices they were unaware of. Over time, this supports skill development within the team and contributes to cleaner, more consistent codebases.

2.4 Limitations and Drawbacks of AI-Based Refactoring

Despite the advantages, several limitations restrict the widespread adoption of AI-based refactoring. A common concern is the trustworthiness of automated changes. Developers may hesitate to accept suggestions that could introduce subtle bugs or break existing functionality. Bharadwaj and Parker [8] highlighted this issue, noting that 14% of AI-generated code fixes in open-source projects introduced new vulnerabilities, even while resolving others.

Another challenge lies in semantic correctness. AI models may recommend structurally valid changes that alter the application's intended behavior. This risk is exceptionally high in systems with limited test coverage or unclear business logic. For safety-critical applications, such as those in healthcare or aviation, developers are less likely to rely on tools without strong correctness guarantees.

Training data is another concern. Many AI models are trained on public repositories, which may not represent proprietary codebases' structure, style, or requirements. As a result, the suggestions can be misaligned with team conventions or specific application architectures. Fine-tuning models for each codebase requires additional time, data, and technical expertise.

Integration into existing workflows also presents hurdles. Developers may be reluctant to adopt tools that produce too many suggestions or interrupt their work. Without thoughtful integration—such as surfacing suggestions in pull requests or offering in-editor feedback—AI refactoring tools may be ignored. McIntosh et al.

[9] emphasized the importance of usability in automated development tools and noted that lack of seamless integration often leads to poor adoption, regardless of technical capability.

2.5 Comparative Analysis: AI-Driven vs. Manual Refactoring

Manual and AI-assisted refactoring approaches differ significantly in identifying issues and applying improvements. While manual refactoring benefits from developer intuition and contextual understanding, it is time-intensive and may lack consistency across teams. AI tools, on the other hand, offer speed and standardization but require careful oversight.

Aspect	Manual Refactoring	AI-Driven Refactoring
Speed and Scalability	Time-consuming; hard to scale across large codebases	Fast analysis; scalable to large projects
Consistency	Varies between developers; subjective decisions	Consistent suggestions based on models and rules
Expertise Requirement	Depends on the developer's experience	Learns from prior data; requires limited intervention once configured
Accuracy and Trust	Generally high, given direct human control	Needs validation; susceptible to incorrect or misaligned suggestions
Adaptability	The developer can adjust based on the context	It may require tuning or retraining to align with project-specific standards
Integration Effort	Part of the normal development process	Needs IDE or pipeline integration to work effectively
Refactoring Scope	Effective at complex, high-level restructuring	Effective at small to medium refactorings; still limited in complex architectural work

Table 1: Comparison of Manual vs. AI-Driven Refactoring

While each approach has strengths and weaknesses, a hybrid model that combines automated detection and suggestions with developer review and final decision-making appears to be the most effective in current practice.

III. PROBLEM STATEMENT

Code refactoring is vital in maintaining software quality, yet many teams struggle to apply it consistently and effectively. As codebases grow in size and complexity, the challenges associated with identifying design issues, planning structural changes, and safely implementing those changes increase. While tools exist to support manual refactoring, several limitations affect their efficiency and adoption. The sections below outline the key problems that limit traditional refactoring efforts and present the gaps that automated, AI-assisted approaches aim to address.

3.1 Difficulty in Consistently Identifying Code Smells

Manual detection of code smells depends heavily on developer experience, familiarity with the codebase, and available time. In practice, teams working under tight deadlines may overlook structural issues that do not directly affect functionality. Furthermore, developers often differ in interpreting smell definitions or applying coding standards, leading to inconsistent assessments of what needs improvement.

Traditional static analysis tools offer support by flagging potential code smells based on fixed thresholds, such as class size or method complexity. However, these tools frequently produce false positives or miss smells from deeper architectural issues. They also lack adaptability, providing the same recommendations regardless of project type, coding style, or domain-specific requirements. As a result, even when such tools are available, developers may find their output too generic or unreliable to act on consistently.

The inability to reliably detect and prioritize code quality issues contributes to the gradual degradation of the codebase. Structural problems accumulate, making future modifications more costly and error-prone. Addressing this limitation requires a detection approach that is both comprehensive and context-aware, capable of recognizing patterns beyond basic metric violations.

3.2 Manual Refactoring is Labor-Intensive and Often Deferred

Once code smells are identified, resolving them involves considerable manual effort. Even small-scale refactorings, such as breaking up a long method or renaming a poorly labeled variable, require attention to detail and verification to ensure correctness. More considerable changes, such as reorganizing a class hierarchy or introducing a design pattern, involve additional coordination, testing, and review.

Because these tasks do not produce immediate functional benefits, they are often deprioritized in favor of new feature development or bug fixing. Over time, this deferral leads to increased technical debt—code that is more difficult to understand, extend, or debug. Teams may intend to return and refactor later, but the effort required becomes more daunting as the backlog grows.

In large or rapidly changing projects, this problem is compounded. Developers may hesitate to refactor code they did not write or do not fully understand, fearing unintended side effects. Without automation, the time cost of analyzing, planning, and safely applying changes remains high. This contributes to a reactive rather than proactive approach to code quality, where refactoring is only undertaken once problems affect delivery timelines or stability.

3.3 Ensuring Correctness and Preventing Regression

Refactoring, by definition, should not alter the external behavior of the software. However, maintaining this guarantee is not always straightforward. Structural changes can affect internal state management, introduce timing issues, or create subtle regressions that escape notice during testing. Even minor errors in refactoring can lead to significant consequences in production environments.

Manual verification relies on the presence of sufficient unit and integration tests. In codebases with limited test coverage, developers must either write new tests or manually validate changes, increasing the time and effort involved. Inconsistent testing practices further increase the risk of introducing regressions during refactoring.

AI-based systems, if used without safeguards, can exacerbate these concerns. Automatically generated suggestions may appear syntactically correct but fail to account for business rules, edge cases, or implicit dependencies. Without mechanisms to evaluate semantic equivalence or test the transformed code, there is a risk of breaking existing functionality. Building trust in refactoring tools—whether manual or automated—requires confidence in their ability to preserve the application's intended behavior.

3.4 Difficulty Integrating Automation into Development Workflows

While there is growing interest in automation, integrating refactoring tools into day-to-day development workflows remains challenging. Many teams already use linters, code formatters, and static analyzers, but these tools often stop short of suggesting or applying structural improvements. Refactoring tools that operate outside the development environment or require separate execution steps are less likely to be used consistently.

Developers are also wary of tools that generate a high volume of suggestions, particularly if those suggestions lack context or disrupt the flow of work. Tools that do not learn from past feedback or cannot be configured to match team standards are more likely to be ignored. In distributed teams or large organizations, alignment on what constitutes acceptable refactoring is often unclear, further limiting tool adoption.

To be effective, automation must be accessible, non-intrusive, and adaptable. Developers need to understand why a change is suggested, evaluate its impact quickly, and apply or reject it with minimal disruption. Even technically capable tools may fail to gain traction without thoughtful integration and user control.

3.5 Gaps That AI Must Address

Given these challenges, the potential role of AI in refactoring is significant. However, for AI-based tools to provide real value, they must address the following gaps:

Contextual Detection

Go beyond metric-based rules by learning from problematic and well-structured code examples within the same or similar projects.

Efficient Refactoring Suggestions:

Generate improvements that save time without overwhelming developers with unnecessary changes or stylistic preferences.

Semantic Awareness

Ensure recommended transformations preserve intended behavior and are verifiable through tests or static analysis.

Customizability

Adapt to project-specific conventions and workflows rather than applying one-size-fits-all solutions.

Workflow Compatibility

Integrate with tools developers already use, including IDEs, code review systems, and CI/CD pipelines.

Closing these gaps would allow AI to become a practical extension of the development process, offering assistance where needed while allowing human judgment to guide the final decisions. This combination of automation and oversight represents a balanced approach to improving code quality at scale.

IV. PROPOSED SOLUTION

This section presents a practical framework for integrating AI into the code refactoring. The framework is designed to support software teams by automating the detection of code smells, generating context-aware

refactoring suggestions, and providing mechanisms for validation and feedback. The goal is to assist developers in improving code quality while maintaining control over structural changes.

4.1 Overview of AI-Driven Refactoring Framework

The proposed framework comprises four core components: a machine learning-based smell detection module, an automated refactoring engine, a human validation and feedback loop, and workflow integration mechanisms. Together, these components allow the system to learn from code history, identify refactoring opportunities, generate and validate improvements, and adapt over time.

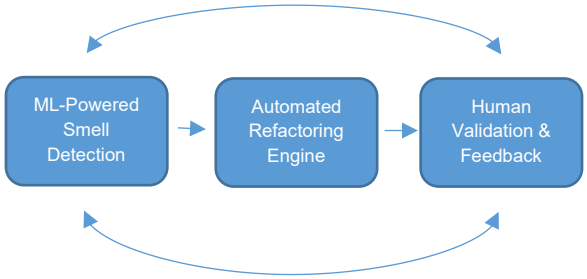

Figure 1: Architecture of the AI-driven Refactoring System

The ML detection module processes code to identify potential smells using trained models.

1. The **refactoring engine** generates possible improvements and applies changes in a testable environment.
2. The **validation loop** enables developers to review, accept, or reject suggestions, contributing to future learning.
3. The system is integrated into development environments and pipelines to support continuous operation.

This architecture aims to automate routine structural improvements while keeping final decisions in the hands of developers.

4.2 ML-Powered Code Smell Detection Module

At the system's foundation is the module responsible for identifying refactoring opportunities. This module uses supervised learning models, trained on labeled code samples, to classify segments of code based on the likelihood of containing certain types of smells. Models can be trained to recognize patterns associated with Long Methods, God Classes, Feature Envy, and other common issues.

Beyond classification, the system incorporates structural context through graph-based representations. Graph neural networks (GNNs) are used to model relationships between classes, methods, and dependencies, which allows for more accurate detection of architectural smells. Historical data, including version history and commit patterns, can be included to improve prioritization. For example, files frequently requiring changes or bug fixes may be weighted more heavily when evaluating potential smells.

The detection output includes a ranked list of refactoring candidates, each with a confidence score. Developers can use this ranking to address high-priority issues first, or the list can be passed directly to the automated engine for processing.

4.3 Automated Refactoring Engine

Once a smell is identified, the next component proposes and, when configured to do so, applies a refactoring. The engine supports both rules-based transformations and AI-generated code edits. Static patterns and syntax transformations are sufficient for routine refactorings, such as method extraction, variable renaming, or redundant code removal.

The system uses AI models trained to suggest structural changes for more complex tasks, such as introducing design patterns or reorganizing classes. For example, if repeated switch-case logic is detected across classes, the system may recommend the Strategy pattern and generate corresponding scaffolding code. These suggestions are prepared in a sandboxed environment, where they are compiled and tested to ensure they do not break existing functionality.

Suggested refactoring includes a short explanation, referencing the identified issue and outlining the intended improvement. This transparency helps developers understand and evaluate the change. Developers can configure thresholds to control which changes are auto-applied and which surface as suggestions for review.

Validation is handled through automated testing and comparison with previous behavior. If a project maintains a unit or integration test suite, the engine runs these tests to confirm behavioral consistency. If tests are unavailable, the system flags the refactoring as "unverified" and routes it for manual inspection.

4.4 Developer Validation and Feedback Loop

The role of the developer is central to the framework. All changes proposed or applied by the system can be reviewed in standard version control systems. Suggested refactorings may be delivered as pull requests, where the developer can review the changes line-by-line, provide comments, or reject the proposal.

To support continuous improvement, the system logs developer decisions. If a suggestion is repeatedly rejected, the model is updated to deprioritize similar suggestions in the future. Accepted suggestions reinforce model behavior and may be added to a growing internal dataset for retraining.

In more advanced configurations, a lightweight reinforcement learning mechanism can be introduced. In this setup, the system receives feedback on each proposed change (accepted or rejected), adjusts its weights accordingly, and gradually aligns itself with the team's style, preferences, and refactoring priorities.

This loop ensures that AI assistance improves over time and reduces the volume of incorrect or irrelevant suggestions, addressing a key concern for developer trust and adoption.

4.5 Integration into Development Workflow

For the system to provide value, it must integrate cleanly into software teams' tools and processes. The framework supports several integration points:

IDE Integration

The detection and suggestion components can run in supported editors, flagging smells and offering quick fixes as developers write or review code.

Version Control Hooks

Suggestions can be surfaced during code reviews by integrating with platforms like GitHub or GitLab. Pull requests may include comments or patches generated by the system.

CI/CD Pipeline Integration

The system can run as a step in the build process, identifying new smells in each commit and optionally blocking changes introducing high-risk issues.

Scheduled Runs

For ongoing maintenance, the system can periodically scan the entire codebase, generate reports, or submit batches of suggested refactorings.

Teams can configure the aggressiveness of suggestions, the types of refactorings offered, and whether changes are applied automatically or flagged for review. The system can also ingest project-specific guidelines or be fine-tuned on internal codebases to reflect unique standards.

This integration strategy ensures the system supports day-to-day work rather than interrupting it and adapts to the working style of each team.

V. CONCLUSION

This paper examined the role of artificial intelligence in automating and improving the software refactoring process. Reviewing current research and tools shows that AI can support key tasks such as code smell detection, design pattern application, and low-level code transformation. Machine learning models can recognize structural issues more consistently than traditional rule-based tools, while AI-assisted engines can generate suggestions that reduce developer workload and help address technical debt more efficiently.

Despite these benefits, important challenges remain. Trust in automated refactoring systems continues to be a barrier, particularly when the proposed changes are applied without full context. Ensuring semantic correctness is another area where AI tools still fall short. Developers need confidence that the behavior of the software will remain intact after a refactoring is applied. Integration into real-world workflows also requires more attention. For AI systems to be functional, they must align with how developers write, review, and deploy code.

The most effective approach appears to combine automated assistance with developer oversight. A hybrid model—where AI provides suggestions or applies changes that a human then validates—strikes a balance between efficiency and accuracy. Over time, these systems can be trained better to match a team's preferences and coding standards, increasing the usefulness of their recommendations.

Future research should focus on improving verification methods that confirm behavioral equivalence after code changes, possibly through automated test generation or formal methods. Expanding the scope of AI-based refactoring from local code smells to more considerable architectural improvements is another promising direction. Finally, making AI tools more transparent and interpretable will be critical to building developer trust and encouraging broader adoption.

By addressing these issues, AI-assisted refactoring can become a dependable part of modern software development, helping teams maintain cleaner, more maintainable codebases with less manual effort.

REFERENCES

[1]. **Marinescu, R.** (2004). Detection strategies: Metrics-based rules for detecting design flaws. *Proceedings of the 20th IEEE International Conference on Software Maintenance*, 350–359.

[2]. **Azeem, M., Palomba, F., Shihab, E., & Wang, Q.** (2019). Machine learning techniques for code smell detection: A systematic literature review. *Journal of Systems and Software*, 155, 65–95.

[3]. **Tufano, M., Watson, C., Bavota, G., Poshyvanyk, D., & Di Penta, M.** (2020). Deep learning similarities from different representations of source code. *Empirical Software Engineering*, 25(3), 1874–1914.

[4]. **Ouni, A., Kessentini, M., Sahraoui, H., & Inozemtseva, L.** (2017). Search-based software refactoring: A systematic literature review. *Information and Software Technology*, 81, 159–175.

[5]. **Mahmoudi, M., Shiri, M. E., & Rabiee, H. R.** (2020). Automatic code refactoring through Q-learning. *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 664–668.

[6]. **Zaidman, A., Van Rompaey, B., Demeyer, S., & Van Deursen, A.** (2010). Studying the relation between refactorings and software defect prediction. *Software Quality Journal*, 18(4), 383–417.

[7]. **Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., & De Lucia, A.** (2015). Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, 41(5), 462–489.

[8]. **Bharadwaj, R., & Parker, I.** (2023). Double-edged sword of LLMs: mitigating security risks of AI-generated code. In *Disruptive Technologies in Information Sciences VII* (Vol. 12542, pp. 141–146). SPIE.

[9]. **McIntosh, S., Kamei, Y., Adams, B., & Hassan, A. E.** (2014). An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5), 2146–2189.

[10]. **Chui, M., Hall, B., Mayhew, H., Singla, A., & Sukharevsky, A.** (2022). *The state of AI in 2022—and a half decade in review*. McKinsey & Company.

[11]. **Amazon Web Services.** (2021). *Improving the CPU and latency performance of Amazon applications using Amazon CodeGuru Profiler*. AWS DevOps Blog.

[12]. **Cheirdari, F., & Karabatis, G.** (2018). Analyzing false positive source code vulnerabilities using static analysis tools. In *2018 IEEE International Conference on Big Data (Big Data)* (pp. 4782–4788). IEEE.