ACE: Automated Technical Debt Remediation with Validated Large Language Model Refactorings

Adam Tornhill
CodeScene
Malmö, Sweden
adam.tornhill@codescene.com

Markus Borg
CodeScene and Lund University
Malmö, Sweden
markus.borg@codescene.com

Nadim Hagatulah
Lund University
Lund, Sweden
nadim.hagatulah@cs.lth.se

Emma Söderberg
Lund University
Lund, Sweden
emma.soderberg@cs.lth.se

Abstract

The remarkable advances in AI and Large Language Models (LLMs) have enabled machines to write code, accelerating the growth of software systems. However, the bottleneck in software development is not writing code but understanding it; program understanding is the dominant activity, consuming approximately 70% of developers' time. This implies that improving existing code to make it easier to understand has a high payoff and – in the age of AI-assisted coding – is an essential activity to ensure that a limited pool of developers can keep up with ever-growing codebases.

This paper introduces Augmented Code Engineering (ACE), a tool that automates code improvements using validated LLM output. Developed through a data-driven approach, ACE provides reliable refactoring suggestions by considering both objective code quality improvements and program correctness. Early feedback from users suggests that AI-enabled refactoring helps mitigate code-level technical debt that otherwise rarely gets acted upon.

CCS Concepts

• **Software and its engineering → Software maintenance tools**; **Integrated and visual development environments**.

Keywords

software engineering, maintainability, code quality, refactoring, AI assistants

ACM Reference Format:
Adam Tornhill, Markus Borg, Nadim Hagatulah, and Emma Söderberg. 2025. ACE: Automated Technical Debt Remediation with Validated Large Language Model Refactorings. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3696630.3730565

FSE Companion '25, Trondheim, Norway

ACM ISBN 979-8-4007-1276-0/2025/06
https://doi.org/10.1145/3696630.3730565

1 Introduction

High-quality code is a competitive business advantage, enabling companies to ship features faster and with fewer defects. Yet, this advantage remains largely untapped as Technical Debt (TD) and poor code continue to consume up to 42% of developers' time [26]. Despite these alarming numbers, research suggests that only 7% of organizations systematically track TD [18]. Instead, software organizations routinely prioritize new feature development over refactoring [1, 8].

This issue could intensify with the advent of AI-assisted development. On the one hand, early empirical studies show that AI assistants can increase developer speed. A pioneering paper by Peng *et al.* claims that GitHub Copilot makes software developers 56% faster [22]. Similarly, a recent randomized control trial at Google reports a 21% improvement in task completion speed when developers used AI-enhanced features for code [21]. On the other hand, neither of these studies evaluated the impact on code quality — which leaves us wondering whether a new kind of TD generator may soon be deployed at scale in industry.

Several studies warn about the risks of naïvely applying AI assistants in software development. For example, Yetiştiren *et al.* report that GitHub Copilot generates correct code only 31% to 65% of the time [32]. Furthermore, an analysis by Harding and Kloster of 150 million changed lines of code reveals "disconcerting trends for maintainability" when adopting AI assistants for code [11]. This implies that the software developers of tomorrow might face a growing mountain of code of unknown quality, much of it written by machines rather than humans.

The growth of AI-generated code of unknown quality has substantial implications for software maintenance, which typically accounts for over 90% of a typical product's lifecycle costs [6]. Moreover, research shows that developers spend about 57-70% of their time understanding existing code, i.e., program comprehension is the dominant programming activity [19, 31]. When the code is plagued by TD, the situation is exacerbated. Consequently, optimizing for software maintainability is crucial for any long-term software development project.

At its core, optimizing for software maintainability means refactoring code. Refactoring is the process of improving the design of existing code without changing its behavior [10]. This seemingly simple definition has two important implications for automated refactoring:

- A change is only a refactoring if it improves the design. However, "improve" has been largely subjective. To automate refactoring, we need an objective standard for assessing improvements.
- A change is not a refactoring if it fails to preserve the behavior of the original code, such as when it introduces a bug. To automate refactoring, we need confidence that the tool adheres to this requirement.

These two implications have guided our solution-oriented work. To address the first, we employ CodeHealth, a code-level quality metric shown to correlate strongly with defect density and development velocity [5, 27]. By comparing the CodeHealth before and after refactoring, we can objectively assess whether the design improved or not.

The second refactoring implication is particularly concerning when using Large Language Models (LLM) to increase the level of automation. Given the tendency of LLMs to hallucinate and produce incorrect code [32], a reliable and trustworthy refactoring tool needs to complement an LLM with guardrails. That way, the results can be validated and incorrect refactoring attempts discarded — this is in line with many implementations of automatic program repair [13].

The remainder of this tool paper is organized as follows. Section 2 describes design rationales for the ACE development and Section 3 presents an overview of closely related work. Section 4 describes an overview of ACE, whereas Section 5 describes inner workings such as prompts and validation. Finally, Section 6 shares early user feedback before Section 7 concludes the paper.

2 Background

TD was first introduced by Ward Cunningham in 1992 as a metaphor for "not-quite-right code" leading to extra work for the next developer. A few years later, the term became popular in the rapidly growing agile software development movement. Unfortunately, it was often broadly used to describe arbitrary quality issues. The research community became interested in the phenomenon with a first international workshop on TD in 2010 and several attempts at defining TD followed. A joint effort at a Dagstuhl seminar in Germany in 2016 resulted in a definition [2] that has stood the test of time: "In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability."

Several researchers have surveyed tools for maintainability prediction and TD management over the last five years. In 2020, Lenarduzzi *et al.* identified 60 maintainability prediction tools in a literature review [15], but 35 were disregarded as outdated research prototypes. They concluded that the most widely adopted tools are commercial.

Focusing on TD measurement, a large number of researchers presented an overview and comparison of commercial tools in 2021 [3]. The study included tools that calculate TD principal or TD interest. Nine tools were investigated, and the authors reported SonarQube as the most widely used tool based on online discussions. More recently, Lenarduzzi *et al.* conducted a detailed comparison of six tools, including SonarQube, across 47 Java projects [14]. Their findings showed a low level of agreement between the tools and highlighted that most of the tools suffer from many false positives.

CodeScene is another commercial tool supporting TD management, although it was not included in the above surveys. The tool uses a relatively small set of code smells (compared to the tools investigated by Lenarduzzi *et al.* [14]) to compute its CodeHealth metric, which it uses for TD identification. Recently, Borg *et al.* reported that CodeHealth produces substantially fewer false positives than SonarQube in Java projects [4]. They performed a competitive benchmarking study using the Maintainability Dataset created by Schnappinger *et al.* as the ground truth [24], which constitutes the most reliable dataset with human-annotated file-level maintainability assessments.

No matter what tool is used to identify refactoring targets, acting on the output has so far mostly been left to the organizations using the tool. The main barriers to action are urgency and skill. First, improving existing code always competes with the time allocated for building new features [8]. Second, refactoring is an acquired skill, and junior developers might lack the expertise to effectively improve the code [30].

ACE was developed to tackle these barriers. By reliably automating code refactoring, ACE has the potential to amplify software organizations in several ways. Specifically, ACE can:

- Reduce time spent on manual refactoring.
- Simplify TD remediation, freeing developers to focus on the creative work of adding new features.
- Elevate team skill levels by providing examples of effective code improvements.

As a baseline for ACE, we have presented a large-scale benchmarking study of popular LLMs on real-world refactoring tasks [28]. The study involved more than 100,000 real-world CodeHealth issues in open-source codebases. Various LLMs were prompted to refactor the specific code smells, and the preservation of behavior was evaluated using the accompanying test suites. In the study, we also assessed if the CodeHealth indeed improved after the refactoring.

The study revealed that the best-performing LLM generated functionally correct refactorings only 37% of the time. This finding was corroborated in a recent paper on using LLMs for extract method refactorings by Pomian *et al.*, which reported that LLMs provide valid and useful suggestions in 24% of their attempts [23]. These results make it clear that we cannot use out-of-the-box AI models or tools that merely wrap an LLM API to reliably refactor code. Instead, a more promising approach is to use generative AI to create a pool of potential solutions and then apply validation guardrails.

ACE implements these guardrails by validating the AI-generated output and assigning a confidence level to the refactorings. By discarding incorrect solutions, 98% of the remaining AI-generated refactorings improve CodeHealth while retaining the original behavior [28]. That is, ACE elevates the precision from 37% to 98%. This precision, however, comes at a cost in terms of reduced recall. Out of the box, an LLM always provides output, resulting in 100% recall. ACE reaches a recall of 52%, thanks to its Contextual LLM Selection discussed in Section 4. Thus, ACE enables confident refactoring of more than half of the detected code smells.

3 Related Work on Refactoring

Recent research in automated code refactoring explores a variety of approaches that use LLMs to improve code quality by removing code smells and reducing TD. Pomian *et al.* introduce EM-Assist [23], a refactoring tool that exploits LLMs to automatically suggest and perform extract method refactorings. They evaluated EM-Assist on 1,752 real-world extract method scenarios, finding that 76.3% of the LLM-generated suggestions were hallucinations. To address this, the authors used static analysis techniques to validate simple program structures and program slicing to further improve the suggested candidates. While EM-Assist achieves higher recall than traditional tools such as JDeodorant and GEMS, it remains reactive, requiring developers to manually identify long methods for refactoring, and the tool is limited to Java and Kotlin, as a plugin for the IntelliJ IDEA.

In a broader empirical study, Liu *et al.* showed that targeted prompting allows LLMs such as GPT-4 to detect 86.7% of real-world refactoring opportunities across 20 different Java projects [17]. However, 7.7% of the proposed refactorings introduced errors, necessitating the need for safeguards. The author proposes RefactoringMirror, which is a strategy that first identifies the refactoring change the LLM proposes by using a refactoring detection tool, e.g. ReExtractor, and then reapplies the suggested refactoring proposal details via a refactoring engine, such as the one provided by IntelliJ IDEA.

Another metric-driven refactoring tool is LiveRef [9], that visually flags identified code smells with colors, through an aggregation of more than 20 software maintainability metrics, such as number of Lines of Code (LoC) and cyclomatic complexity. The color-coded severity indicator allows developers to prioritize refactoring opportunities. The authors' empirical evaluation showed that developers who used LiveRef applied refactorings more frequently and improved maintainability metrics faster than those who relied on manual methods. However, similar to EM-Assist, LiveRef is limited to Java and the IntelliJ IDEA as a plugin and has a long analysis latency on the order of seconds.

Shirafuji *et al.* demonstrates a method of using an LLM, GPT-3.5-turbo, with few-shot prompting to refactor Python programs into simpler, more readable, and maintainable versions [25]. Their method generated 10 candidate solutions per program, with 95.68% yielding at least one valid solution, determined by unit tests to ensure functional correctness. The refactorings achieved a 17.35% reduction in cyclomatic complexity and a 25.84% decrease in LoC, showing that a few-shot prompting method can guide an LLM to produce refactorings that improve the original code. However, the study was limited to introductory Python problems and focused primarily on reducing complexity, even when the original code is already in a good state. Unlike ACE, which uses CodeScene to automatically identify methods with code smells as candidates for refactoring.

ACE differentiates itself by being a more self-contained tool while also expanding its applicability. Similarly to LiveRef, ACE uses a deterministic and metric-driven detection method, but is based on the CodeHealth metric, to identify code smells and refactoring opportunities with minimal latency. In contrast to RefactoringMirror, ACE applies mostly self-contained validations to LLM-generated refactoring suggestions to mitigate hallucinations without fully relying on external tooling. Furthermore, ACE increases the scope of automated code refactoring by supporting multiple programming languages and aiming to support multiple IDEs, which makes ACE a more robust tool for TD remediation.

4 ACE Overview

In developing ACE, we leverage CodeScene output in two ways. First, we select a subset of the code smells that build up the CodeHealth score as refactoring targets. Second, we use CodeHealth as a fundamental part of the validation step described in Section 5.2. As CodeScene is available as an IDE extension in VS Code, ACE follows suit and is currently available within the same ecosystem. At the time of writing, the release of ACE as a plugin for IntelliJ is imminent. A complete ACE use case is available as a screencast[1].

Figure 1 shows an overview of ACE. The back-end architecture, alongside a set of LLMs, is based on two key components:

(1) Contextual LLM Selection: The benchmarking study revealed that LLMs have different strengths and weaknesses [28]. ACE takes advantage of this by using a machine learning model to select the most appropriate LLM based on the properties of the code to be refactored. The multi-LLM approach improves ACE's recall and decouples its application logic, facilitating the integration of new LLMs as they become available. The current version of ACE uses LLMs provided by OpenAI, Anthropic (Claude), Google, and the LLaMA family of models.
(2) LLM Validation: The responsibility of the validator is to ensure that the changed code is indeed a refactoring given the two implications presented in Section 1. The validator does this by a combination of static code analysis techniques.

Developers interact with ACE through their IDE. There are two main interaction points for ACE, each one of them supporting a separate use case in the space of TD management:

(1) Refactor Declining CodeHealth: When a local quality gate detects a decline in CodeHealth, ACE offers refactoring suggestions as shown in Figure 2.
(2) Proactive Code Improvement: Developers can proactively improve existing code on-demand as shown in Figure 4.

Most developers do not actively scan codebases for refactoring opportunities. Instead, a refactoring tool must address a specific need at the right time [20]. For ACE, the local quality gate was a natural choice: it triggers when a developer lowers the CodeHealth. At that moment, the code change is fresh in the developer's mind, and offering automated refactoring has two advantages. First, developers might not know how to refactor the code effectively. Second, due to the pervasive time pressure in the software industry, code quality is often sacrificed for speed [16] – a micro-optimization that harms the macro progress. ACE overcomes these barriers, helping organizations manage TD by establishing a quality baseline for existing code, aligning with the "Boy Scout Rule" [29]. Via the quality gate, refactoring recommendations are delivered non-intrusively to avoid disrupting the developer's workflow.

Another common refactoring scenario arises when a developer wants to make a change but finds the existing code too complicated. Similarly, the team has decided to invest time into actively paying down their TD. In both situations, the developers look to elevate

[1] https://www.youtube.com/watch?v=7KQ1oXysFvc

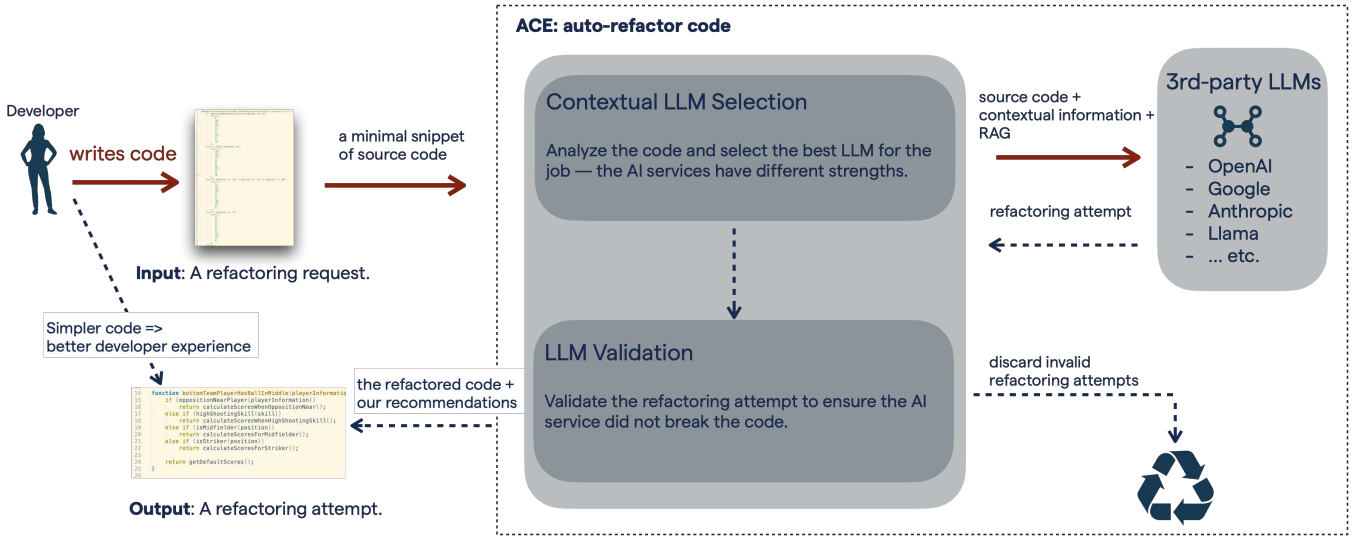

Figure 1: The high-level architecture of the ACE tool for reliable LLM-enabled refactoring.

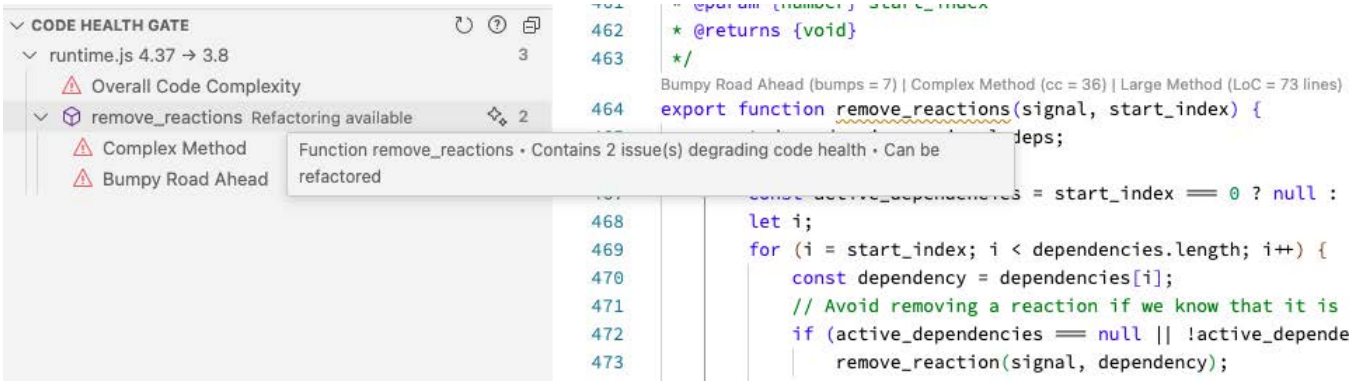

Figure 2: ACE offers automated refactoring of issues discovered in the CodeHealth quality gate.

CodeHealth for a specific piece of code. ACE supports this use case by presenting code smells in the IDE and offering to fix them automatically.

Finally, it is worth emphasizing that ACE leaves the developer in control. While the refactorings are automated, the developer decides whether to accept them. To simplify the decision, ACE presents a summary of its review and the corresponding validation results. Even if a refactoring is not perfect, it can serve as a useful starting point, allowing developers to make minor tweaks and fill in the gaps themselves.

5 ACE Architecture and Implementation

This section describes key components in the ACE architecture and implementation. Furthermore, we explain which code smells are currently targeted.

5.1 LLM Prompting Strategy

ACE constructs prompts dynamically based on the specific code smell and its context. Input to this customization includes: i) programming language, allowing language-specific guidance (e.g., handling the "this" keyword in JavaScript), ii) source code of the function to be refactored, and iii) type of function, whether it is standalone or a class function, iv) code smell category, informing the refactoring strategy (e.g., applying extract method for complex conditionals), and v) code smell location within the function. These pieces of information are combined into a layered prompt that guides the LLMs to perform minimal and focused code changes to remove the code smell.

5.2 LLM Validation Layer

The dynamic prompting strategy and combining multiple AI services only explain parts of ACE's refactoring capabilities. The other fundamental component is the validation layer in ACE, whose high-level decision flow is shown in Figure 3, which assigns a confidence level to each LLM-based refactoring. Note that ACE is a proprietary tool, and the exact details cannot be disclosed.

Low-confidence refactoring attempts are likely to break the code or make it worse. These are discarded by the ACE service and

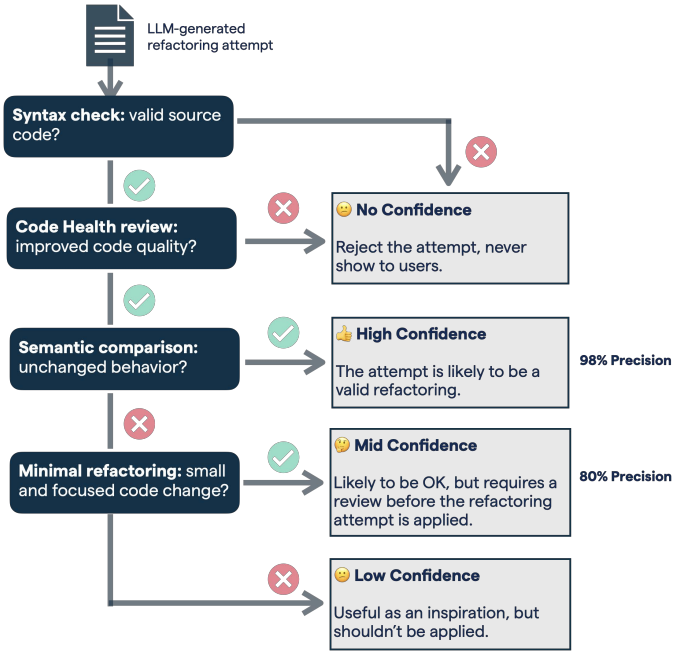

Figure 3: Flowchart for the validation layer for assigning a confidence level to each automated refactoring.

never shown to users. Mid- and high-confidence refactorings are presented to the user together with the rationale for the decisions. This rationale comes from the validation layer (see Figures 4 and 5).

As shown in Figure 3, there are multiple validation steps in ACE. The first step is syntactic validation, where a static analysis tool (e.g., ESLint for JavaScript) is used to verify that the LLM-generated refactoring output is syntactically valid. This step involves checking the original code and the refactored code for differences reported by the analysis tool. If the refactored code fails the check or introduces any additional warnings compared to the original code, then ACE discards the refactoring suggestion or lowers its confidence score. This step makes sure that the LLM-generated refactoring solution does not introduce any syntactic errors or warnings.

The second step validates that the targeted code smell has been resolved and that the overall CodeHealth has been improved by the LLM-generated refactoring suggestion. This step compares the code smell and CodeHealth score of the original and the refactored code for differences. If the targeted code smell remains or if the CodeHealth score is lower in the refactoring suggestion, then ACE discards that refactoring suggestion. If the code smell is removed and the CodeHealth score is improved, but a new, less severe code smell is introduced by the refactoring, then the confidence score will be lowered. This step ensures that the LLM-generated refactoring resolves the targeted code smell from the original code and increases the overall CodeHealth score without reducing the code quality.

The final steps in ACE's validation process involve semantic validation. These steps compare the original and refactored code to verify that the behavior is preserved and that only the expected minimal changes were introduced. During development, several recurring patterns were identified where AI-generated refactorings can break semantic correctness. These patterns are now captured and used to assess equivalence and assign a corresponding confidence level. If the semantic comparison confirms that the expected changes were made, a high confidence level is assigned. If minor but unintended changes are found, the result is marked with mid confidence. More substantial or unclear deviations result in low confidence. The expected changes are defined through a combination of code smell-specific validations and general language-specific rules, such as detecting empty method stubs or avoiding redundant re-implementations of existing logic.

It is important to note that assessing semantic equivalence is a largely unsolved research problem in the area of automated program repair [12]. Consequently, ACE never aimed at solving semantic equivalence in general – which is an undecidable problem according to Rice's theorem. Rather, the validation is specifically tailored to the set of code smells identified via the CodeHealth metric. This allows ACE to constrain the validation process to a finite set of structural changes that can be learned by our internal machine learning models. The quality of ACE's data lake, containing hundreds of thousands of refactorings with known ground truths, is the secret sauce, not the algorithms.

5.3 Supported Code Smells and Programming Languages

ACE is an evolving product. In its first release, the tool supports automated refactoring of five common code smells, all part of the larger aggregated CodeHealth:

- Complex Conditional: expressions inside a branch (e.g., if, for, while) containing multiple, logical operators such as AND or OR.
- Complex Method: many conditional statements inside a function, measured using cyclomatic complexity.
- Deep Nested Logic: the nesting depth as measured by the number of if-statements and/or loops inside other program branches.
- Bumpy Road: a function with sequential chunks of Deep Nested Logic.
- Large Method: large functions as measured by the lines of code.

ACE currently supports automated refactoring of JavaScript, TypeScript, and Java. Our first priority has been on the dominant front-end languages, followed by Java. Next, we plan to extend ACE to cover additional back-end languages such as C#, C, and C++.

6 Early Feedback: Dogfooding and Limitations

ACE was evaluated by two audiences to ensure a product-market fit. First, our development team used ACE internally. Although the precision and recall numbers were promising, subjective assessments from developers – the target audience – were crucial during development. Once the internal team found ACE valuable, the next step was to test it with external users in a controlled Alpha release.

One pilot user, an experienced tech lead, commented *"I'm surprised with ACE's suggestions, as they are quite similar to what I would have come up with myself, saving me time and cognitive effort."* Another software developer, a self-described "AI skeptic," remarked *"I like how ACE breaks down complex components into atomic parts with appropriate naming."*

Figure 4: Example of a high confidence extract method refactoring.

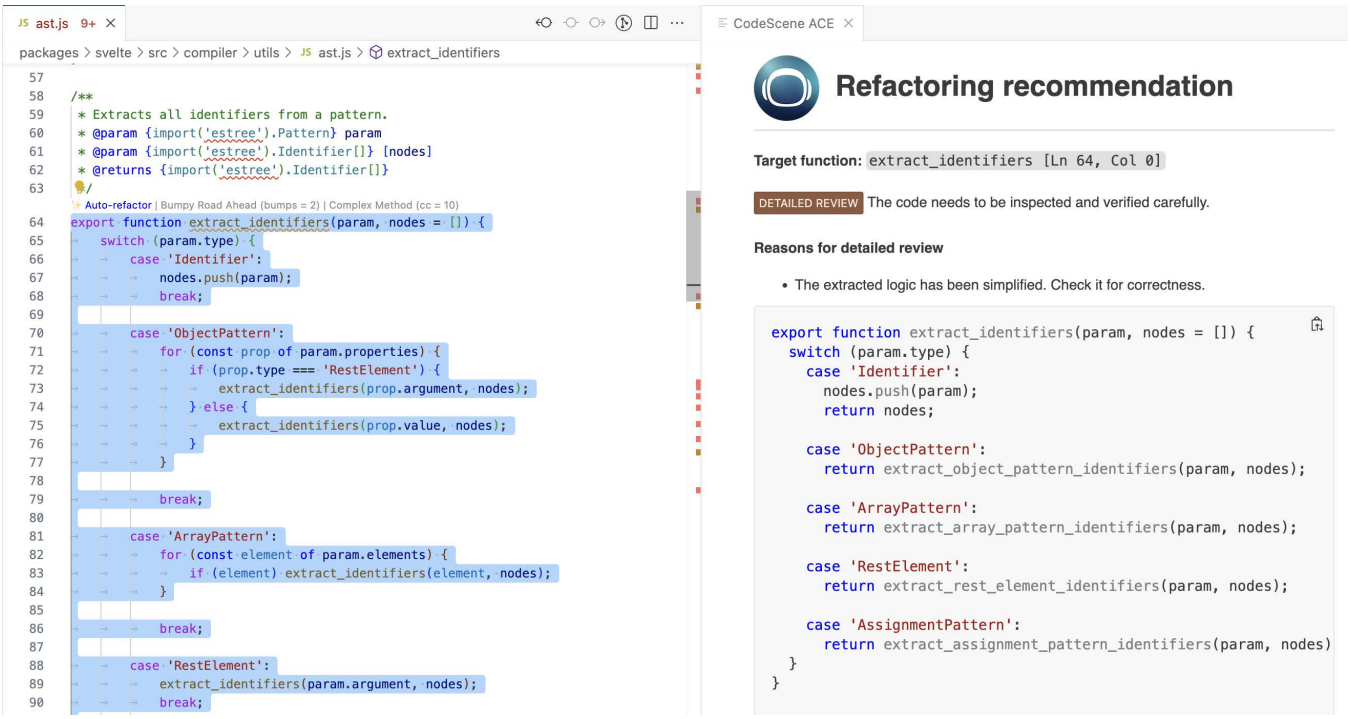

Figure 5: Example of a mid confidence refactoring that simplifies cyclomatic complexity.

In fact, one of the key advantages of using LLMs for refactoring is their ability to name and decompose convoluted elements. Since approximately 70% of code consists of identifiers, good naming is paramount for readability and maintainability [7].

The Alpha testing phase highlighted some early limitations that now inform the ACE roadmap:

- **Applicability:** JavaScript and TypeScript are traditional front-end technologies. Much TD is on the back-end, necessitating support for more programming languages. This motivated us to implement support for Java, and more back-end languages are in the works.
- **LLM Scaling:** Our initial data showed a drop in LLM output quality for functions larger than 70 LoC, resulting in lower recall as function size increased. Our recall improvements stretched the maximum refactorable function length to 130 lines of code, and we aim to continue expanding this limit.

- **Scope:** The first ACE version focuses on function-level code smells (e.g., Complex Conditional and Bumpy Road) to avoid breaking APIs. As a result, far from all code quality problems can currently by automatically fixed. Expanding the refactoring scope of ACE is an important direction for future work.

7 Conclusion

ACE is a tool designed to automate refactoring and improve the design of existing code. By combining the creative capabilities of LLMs with a robust validation layer, ACE delivers high-confidence refactorings with 98% precision, outperforming out-of-the-box LLMs.

Introducing an AI coding assistant that mitigates complex code smells safely allows users to optimize code for understanding – the most human-intensive aspect of coding – not just the occasional task of writing new code. An even more compelling potential of ACE is its promise of automated TD remediation. Every business manager is aware of TD, but few prioritize it – and even fewer manage their debt actively [18].

ACE is a proprietary solution, but free to download and try. The tool currently supports automated refactoring for five code smells, with the capability to detect 25. Future versions of ACE will expand its refactoring scope. Similarly, an important future direction is to expand the number of supported programming languages to cover additional back-end languages like C#, C++, and PHP.

Acknowledgments

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) and partly by the Competence Centre NextG2Com funded by the VINNOVA program for Advanced Digitalisation with grant number 2023-00541.

References

[1] Areti Ampatzoglou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Paris Avgeriou. 2015. The Financial Aspect of Managing Technical Debt: A Systematic Literature Review. *Info. and Softw. Technology* 64 (2015), 52–73.

[2] Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, and Carolyn Seaman. 2016. Managing Technical Debt in Softw. Engineering (Dagstuhl Seminar 16162). *Dagstuhl Reports* 6, 4 (2016), 110–138.

[3] Paris C. Avgeriou, Davide Taibi, Apostolos Ampatzoglou, Francesca Arcelli Fontana, Terese Besker, Alexander Chatzigeorgiou, Valentina Lenarduzzi, Antonio Martini, Athanasia Moschou, Ilaria Pigazzini, Nyyti Saarimaki, Darius Daniel Sas, Saulo Soares de Toledo, and Angeliki Agathi Tsintzira. 2021. An Overview and Comparison of Technical Debt Measurement Tools. *IEEE Softw.* 38, 3 (2021), 61–71.

[4] Markus Borg, Marwa Ezzouhri, and Adam Tornhill. 2024. Ghost Echoes Revealed: Benchmarking Maintainability Metrics and Machine Learning Predictions Against Human Assessments. In *Proc. of the 40th Int'l. Conf. on Softw. Maintenance and Evolution*. 278–288.

[5] Markus Borg, Ilyana Pruvost, Enys Mones, and Adam Tornhill. 2024. Increasing, Not Diminishing: Investigating the Returns of Highly Maintainable Code. In *Proc. of the 7th Int'l. Conf. on Technical Debt*. 21–30.

[6] Sayed Mehdi Hejazi Dehaghani and Nafiseh Hajrahimi. 2013. Which Factors Affect Softw. Projects Maintenance Cost More? *Acta Informatica Medica* 21, 1 (2013), 63–66.

[7] Florian Deissenboeck and Markus Pizka. 2006. Concise and Consistent Naming. *Softw. Quality Journal* 14, 3 (2006), 261–282.

[8] Neil Ernst, Rick Kazman, and Julien Delange. 2021. *Technical Debt in Practice: How to Find It and Fix It.* The MIT Press, Cambridge, Massachusetts.

[9] Sara Fernandes, Ademar Aguiar, and André Restivo. 2023. LiveRef: a Tool for Live Refactoring Java Code. In *Proc. of the 37th IEEE/ACM Int'l. Conf. on Automated Softw. Engineering* (Rochester, MI, USA). Article 161.

[10] Martin Fowler. 2018. *Refactoring: Improving the Design of Existing Code* (2 ed.). Addison-Wesley Professional.

[11] William Harding and Matthew Kloster. 2024. *Coding on Copilot.* Technical Report. GitClear. https://gitclear-public.s3.us-west-2.amazonaws.com/Coding-on-Copilot-2024-Developer-Research.pdf

[12] Thanh Le-Cong, Duc-Minh Luong, Xuan Bach D. Le, David Lo, Nhat-Hoa Tran, Bui Quang-Huy, and Quyet-Thang Huynh. 2023. Invalidator: Automated Patch Correctness Assessment Via Semantic and Syntactic Reasoning. *IEEE Trans. Softw. Eng.* 49, 6 (2023), 3411–3429.

[13] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (2019), 56–65.

[14] Valentina Lenarduzzi, Fabiano Pecorelli, Nyyti Saarimaki, Savanna Lujan, and Fabio Palomba. 2023. A Critical Comparison on Six Static Analysis Tools: Detection, Agreement, and Precision. *Journal of Systems and Softw.* 198 (2023), 111575.

[15] Valentina Lenarduzzi, Alberto Sillitti, and Davide Taibi. 2020. A Survey on Code Analysis Tools for Softw. Maintenance Prediction. In *Proc. of the 6th Int'l. Conf. in Softw. Engineering for Defence Applications*. Cham, 165–175.

[16] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A Systematic Mapping Study on Technical Debt and Its Management. *Journal of Systems and Softw.* 101 (2015), 193–220.

[17] Bo Liu, Yanjie Jiang, Yuxia Zhang, Nan Niu, Guangjie Li, and Hui Liu. 2024. An Empirical Study on the Potential of LLMs in Automated Software Refactoring. (2024). https://doi.org/10.48550/arXiv.2411.04444

[18] Antonio Martini, Terese Besker, and Jan Bosch. 2018. Technical Debt Tracking: Current State of Practice: A Survey and Multiple Case Study in 15 Large Organizations. *Science of Computer Programming* 163 (2018), 42–61.

[19] Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time. In *Proc. of the 23rd Int'l. Conf. on Program Comprehension*. 25–35.

[20] E. Murphy-Hill and G. Murphy. 2014. Recommendation Delivery. In *Recommendation Systems in Softw. Engineering*, M. Robillard, W. Maalej, R. Walker, and T. Zimmermann (Eds.). Springer, 223–242.

[21] Elise Paradis, Kate Grey, Quinn Madison, Daye Nam, Andrew Macvean, Nan Zhang, Ben Ferrari-Church, and Satish Chandra. 2024. How Much Does AI Impact Development Speed? An Enterprise-based Randomized Controlled Trial. http://arxiv.org/abs/2410.12944

[22] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot.

[23] Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Timofey Bryksin, and Danny Dig. 2024. Next-Generation Refactoring: Combining LLM Insights and IDE Capabilities for Extract Method. In *Proc. of the 40th Int'l. Conf. on Softw. Maintenance and Evolution*.

[24] Markus Schnappinger, Arnaud Fietzke, and Alexander Pretschner. 2020. Defining a Softw. Maintainability Dataset: Collecting, Aggregating and Analysing Expert Evaluations of Softw. Maintainability. In *Proc. of the 36th Int'l. Conf. on Softw. Maintenance and Evolution*. 278–289.

[25] Atsushi Shirafuji, Yusuke Oda, Jun Suzuki, Makoto Morishita, and Yutaka Watanobe. 2023. Refactoring Programs Using Large Language Models with Few-Shot Examples . In *Proc. of the 30th Asia-Pacific Softw. Engineering Conf.* 151–160.

[26] Stripe. 2018. *The Developer Coefficient: Softw. engineering efficiency and its $3 trillion impact on global GDP.* Technical Report. https://stripe.com/reports/developer-coefficient-2018

[27] Adam Tornhill and Markus Borg. 2022. Code Red: The Business Impact of Code Quality - A Quantitative Study of 39 Proprietary Production Codebases. In *Proc. of the 5th Int'l. Conf. on Technical Debt*. 11–20.

[28] Adam Tornhill, Markus Borg, and Enys Mones. 2024. *Refactoring vs Refactoring: Advancing the State of AI-automated Code Improvements.* Technical Report. CodeScene. https://tinyurl.com/refuctoring

[29] Roberto Verdecchia, Philippe Kruchten, Patricia Lago, and Ivano Malavolta. 2021. Building and evaluating a theory of architectural technical debt in software-intensive systems. *Journal of Systems and Softw.* 176 (2021), 110925.

[30] Yi Wang, Christian Wagner, and Rachael Ip. 2009. An Empirical Investigation of the Key Factors for Refactoring Success in an Industrial Context. In *Proc. of the 15th Americas Conf. on Information Systems (577)*.

[31] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2018. Measuring program comprehension: a large-scale field study with professionals. In *Proc. of the 40th Int'l. Conf. on Softw. Engineering*. 584.

[32] Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. 2022. Assessing the Quality of GitHub Copilot's Code Generation. In *Proc. of the 18th Int'l. Conf. on Predictive Models and Data Analytics in Softw. Engineering*. 62–71.