Whitepaper

Refactoring vs Refuctoring:

Advancing the state of AI-automated code improvements

By Adam Tornhill, Markus Borg, PhD & Enys Mones, PhD

Summary

This report is the conclusion of a benchmark study of the most popular Large Language Models (LLMs) and their ability to generate code for refactoring tasks. We aim to illustrate the current standards and limitations, and seek to show new methodologies with higher confidence results.

CodeScene

Introduction

The remarkable advances in AI promised a coding revolution, spawning tools to help us write code faster. Yet the true gains elude us. The crux? The majority of a developer's time isn't writing but understanding and maintaining existing code.

This whitepaper explores this new frontier by investigating AI support for improving existing code. We do that via two important contributions:

- First, we benchmark the performance of the most popular Large-Language Models (LLM) on refactoring tasks for improving real-world code. We find that existing AI solutions only deliver functionally correct refactorings in 37% of the cases.
- Second, as a response to the poor performance of LLMs, we introduce a novel innovation for fact-checking the AI output and augmenting the proposed refactorings with a confidence level. By rejecting incorrect solutions, 98% of the remaining AI-generated refactorings improve the code while retaining the original behavior.

This level of precision exceeds that of even human experts, highlighting the utility of fact-checked AI. By applying this innovation, software organizations get a viable way forward for automating improvements to existing code, including auto-mitigations of technical debt.

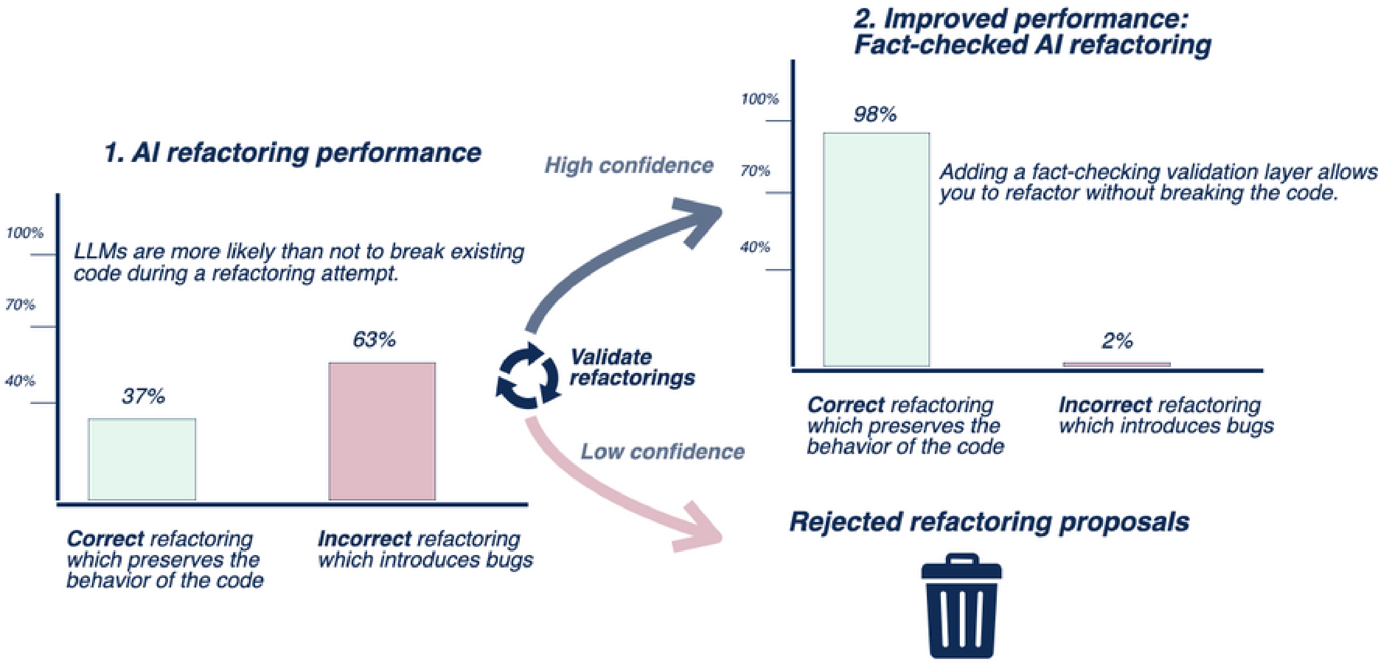


Figure 1: State-of-the-art generative AI breaks the code in 63% of all refactoring attempts (left). Fact-checking the AI allows us to reject the majority of all broken refactoring attempts.

Benchmark: AI performance on code refactoring

As exciting as the AI revolution is, we are far from realizing the claimed productivity breakthrough. At least for non-trivial coding tasks. (See our Forbes article for why AI-assisted coding is still in its infancy).

Specifically, two main barriers remain to be conquered before AI can truly disrupt the way we work with source code:

1. **Optimize for software maintenance** which accounts for more than 90% of a software product's life cycle costs. Specifically, 70% of developers' time is spent on program understanding, meaning that any improvements that make the existing code easier to grasp will have a high return on investment.

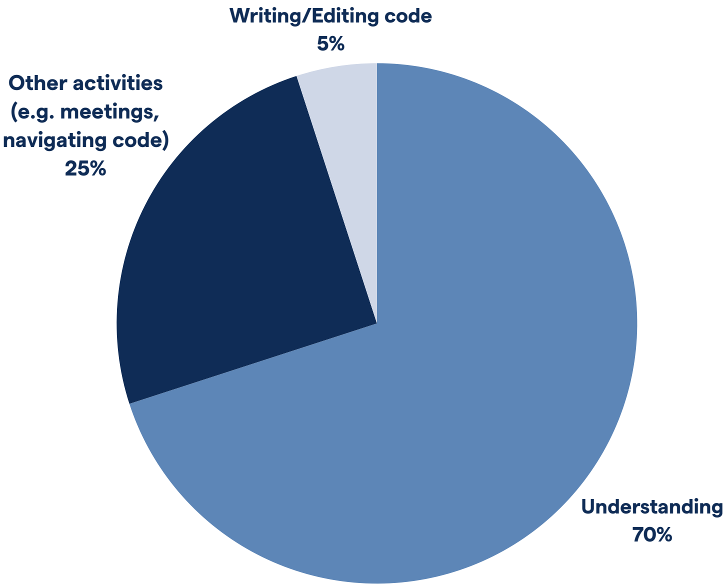


Figure 2: The majority of a developer's time is spent trying to understand the existing system (data from Minelli, et. al., 2015) 1

1. https://ieeexplore.ieee.org/abstract/document/7181430

2. **Improve AI precision to the level of a human expert**. The disappointing 37% correctness score of today's AI solutions simply isn't good enough for refactoring production code. Rather, the hit-or-miss success ratio adds to the problem by increasing developers' cognitive load as we have to scrutinize all AI refactorings with great care to sort out the good from the bad. Reviewing code is arguably a harder task than writing it.

These two factors indicate that innovation in tooling to support improving existing code – without breaking it – is a more important direction than focusing on optimizing the less significant code-writing process. Let's discuss why.

Are we refactoring or just breaking code?

Refactoring is defined as improving the design of existing code without changing its behavior. It's a simple definition, but with some important implications:

- It's not a refactoring unless we improve the design. "Improve" has been largely subjective. To automate refactoring, we need a gold standard to make improvements objectively measurable.
- It's not a refactoring if we fail to preserve the behavior of the original code, e.g. we introduce a bug. To automate refactoring, we need confidence that the machine adheres to this assumption.

Unless these two conditions are met, a code change is simply not a refactoring. For the purpose of this article, we will use the term refuctoring to refer to the process of changing existing code while – involuntarily – altering the program's behavior.

Benchmarking criteria: a gold standard for code improvements

This study uses the Code Health metric as a proxy for code quality. Code Health is the only code-level metric with a proven business impact in terms of development velocity and post-release defects. (See the Code Red paper for details).

The Code Health metric is a particularly good fit when refactoring code as the measure is based on factors known to make code harder to understand and riskier to maintain in terms of defect introduction:

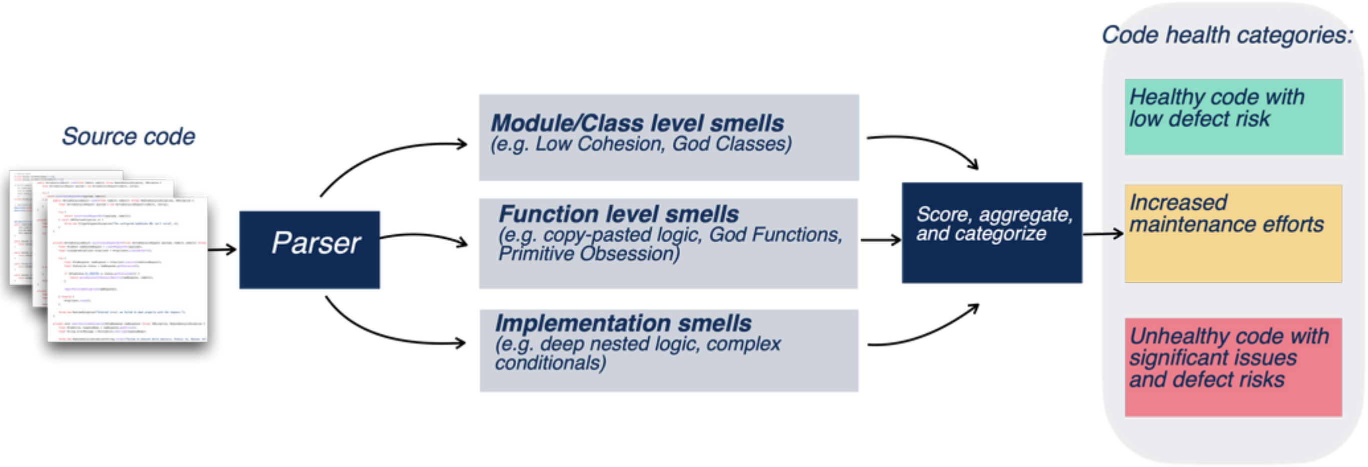

Source code
Parser
Module/Class level smells
(e.g. Low Cohesion, God Classes)
Function level smells
(e.g. copy-pasted logic, God Functions, Primitive Obsession)
Implementation smells
(e.g. deep nested logic, complex conditionals)
Score, aggregate, and categorize
Code health categories:
Healthy code with low defect risk
Increased maintenance efforts
Unhealthy code with significant issues and defect risks

Figure 3: Code Health is a language-neutral, aggregated code quality metric based on a combination of 25 code smells.

The detailed Code Health scores used in this study go from 10.0 (healthy code) all the way down to 1.0 (a maintenance nightmare / spaghetti code / high technical debt). As such, the Code Health metric offers an objective assessment of any code changes: did the Code Health improve – a refactoring that makes the code easier to understand – or was the code merely changed without getting objectively better?

AI performance on code refactoring

To evaluate how well current AI platforms perform, we collected more than 100,000 real-world code smells, and pointed state-of-the-art AI models at these targets to refactor the code. We used the CodeScene tool to identify Code Health issues in codebases. We then evaluated the correctness of the attempted refactoring by running the code's automated tests, as well as making sure the code quality improved by re-assessing the Code Health.

For this benchmarking study, we focused on code in JavaScript and TypeScript. LLM performance varies across programming languages, so we chose to start with two popular and well-supported languages. We also centered the refactorings on four common Code Health issues: Complex Conditionals, Deep Nested Logic, Bumpy Road, and Complex Method. (See the docs for descriptions of these code smells).

Using this data, we measure the ratio of refactoring vs refuctoring for a series of popular AI models:

AI model	**Valid code?** (check the syntax of the refactored code)	**Code Health improved?** (did the code change by the AI mitigate the code smell?)	**Valid refactoring?** (do the tests still pass after the AI changed the code?)
PaLM 2 code	99.93%	68.75%	37.29%
GPT-3.5	100%	69.89%	30.26%
PaLM 2t	100%	66.54%	34.73%
phind-codellama-34B-v2*	100%	78.76%	18.14%

Table 1: Benchmarking of refactoring correctness for a series of popular Large Language Models. *A fine-tuned model based on CodeLlama 34B.

As the preceding table shows, using an out-of-the-box AI is very much a hit-or-miss situation. In fact, with the best-performing model only giving a 37% probability of success, it's more likely that the attempted refactoring will break your code than not.

Notes on GPT4 Performance

During our research, we also made some benchmarks using GPT4. These tentative studies indicate that GPT4 seems to perform marginally better. However, those potential improvements are offset by GPT4 being significantly slower and an order of magnitude more expensive. Without a drastic gain, GPT4-based refactoring doesn't seem to be a viable alternative either.

Is fragile code an acceptable new normal?

Our research findings align with a 2023 study which found that popular AI-assistants like Copilot and CodeWhisperer only deliver functionally correct code in 31% - 65% of the cases. Generating new code is arguably a simpler task than refactoring complex code, which explains the higher-end of those numbers. However, those research findings are within the same ballpark; the performance of those AI-assistants reflects the performance of the underlying LLM.

It's safe to assume that a human developer shipping code which breaks 60-80% of the time would be asked to look for new challenges. Promptly.

How does an AI err?

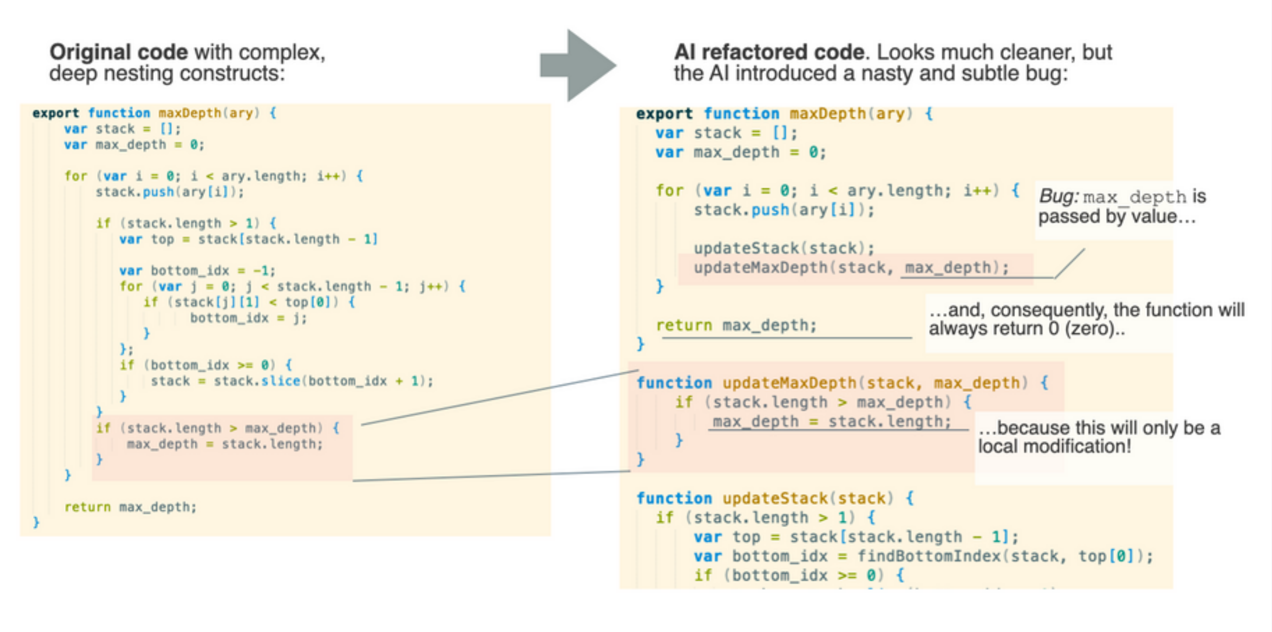
Strictly speaking, an AI based on LLMs doesn't have a concept of "correctness"; some tokens (e.g. output) are more probable than others, but the AI itself doesn't execute the code, nor does it "prove" its response in any meaningful way.

This spells trouble as the nature of programming is a much more constrained environment than a conversation between humans. For example, when generating a natural text, choosing one synonym over another usually doesn't make or break a text. (e.g. consider "gear" and "equipment"). Not so in code – a single character can alter the behavior of any program.

During our research we inspected a lot of AI-generated code. Some common patterns are that the AI:

- Drops entire branches, e.g. throwing away an if-block. Note that these bugs can easily become a cybersecurity threat in case important input validation gets removed.
- Inverts boolean logic, e.g. a && b becomes !(a && b).
- In JavaScript – which is notorious for its complex rules for the this-keyword – the AI often mistreats the function-level this by extracting the literal expression to a new function, a classic refuctoring pitfall.

All of these failures are subtle and not obvious to the human eye during a code inspection.



Original code with complex, deep nesting constructs:

```
export function maxDepth(ary) {
    var stack = [];
    var max_depth = 0;

    for (var i = 0; i < ary.length; i++) {
        stack.push(ary[i]);

        if (stack.length > 1) {
            var top = stack[stack.length - 1]

            var bottom_idx = -1;
            for (var j = 0; j < stack.length - 1; j++) {
                if (stack[j][1] < top[0]) {
                        bottom_idx = j;
                }
            };
            if (bottom_idx >= 0) {
                stack = stack.slice(bottom_idx + 1);
            }
        }
        if (stack.length > max_depth) {
            max_depth = stack.length;
        }
    }

    return max_depth;
}
```

AI refactored code. Looks much cleaner, but the AI introduced a nasty and subtle bug:

```
export function maxDepth(ary) {
  var stack = [];
  var max_depth = 0;

  for (var i = 0; i < ary.length; i++) {
      stack.push(ary[i]);

      updateStack(stack);
      updateMaxDepth(stack, max_depth);
  }

  return max_depth;
}

function updateMaxDepth(stack, max_depth) {
    if (stack.length > max_depth) {
        max_depth = stack.length;
    }
}

function updateStack(stack) {
  if (stack.length > 1) {
      var top = stack[stack.length - 1];
      var bottom_idx = findBottomIndex(stack, top[0]);
      if (bottom_idx >= 0) {
```

Bug: `max_depth` is passed by value…

…and, consequently, the function will always return 0 (zero)..

…because this will only be a local modification!

Figure 4: Example on an AI-introduced, subtle bug that takes a great deal of effort to track down.

These findings also indicate that we should be more cautious about how and when we apply AI-assisted coding. What if we could add a safety net around the AI? That would let us reap the benefits of automation by outsourcing the mechanics to an AI while still giving us – as software people – some guarantees that we are refactoring, not refuctoring. Welcome to the future – there's a light on the horizon.

Tools for inspiration...but use with care!

The AI-assistance of today can still be helpful, despite their frequent errors. In particular, an AI-assistant like Copilot or CodeWhisperer can be useful as the starting point for new code, serving as an inspiration and a coach. However, the burden is still on you to verify that the code is correct and – just as important – that it's code you and your team can maintain going forward.

Into the future: improving automated refactoring by fact-checking the AI

Given the low correctness of the stochastic AI models, it becomes strikingly clear that we cannot use out-of-the-box AI models or tools that merely wrap an LLM API. Instead, a more promising approach is to use generative AI to come up with a pool of potential solutions and then add a fact-checking layer around the AI. That way, we get the benefits of automation while retaining a certain level of guarantee that a proposed solution is a refactoring rather than a refuctoring.

CodeScene's research team took on this challenge by creating a layered fact-checking model:

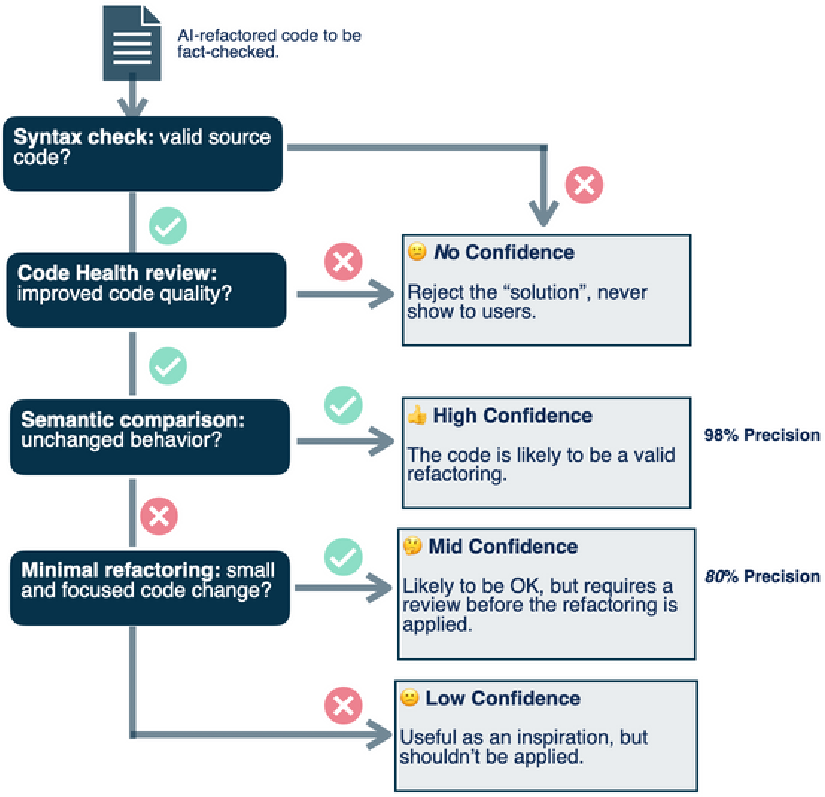

AI-refactored code to be fact-checked.
Syntax check: valid source code?
Code Health review: improved code quality?
Semantic comparison: unchanged behavior?
Minimal refactoring: small and focused code change?
No Confidence
Reject the "solution", never show to users.
High Confidence
The code is likely to be a valid refactoring.
98% Precision
Mid Confidence
Likely to be OK, but requires a review before the refactoring is applied.
80% Precision
Low Confidence
Useful as an inspiration, but shouldn't be applied.

Figure 5: A schematic overview of the layered model for fact-checking AI-refactored code.

Benchmarking: improving AI correctness with a fact-checking model

To evaluate the fact-checking model, we re-ran the benchmarking study described in Table 1 above. This makes it possible to compare the correctness gained from the fact-checking model:

	Correct code smell refactorings			
Solution	**Complex Conditional**	**Deep Nested Logic**	**Bumpy Road**	**Complex Method**
Raw GPT-3.5	33.7%	26.0%	26.3%	28.2%
AI with CodeScene's fact-checking	96.7%	98.4%	97.8%	98.9%

Table 2: Benchmarking data showing how the confidence in refactoring can be improved to 98%. GPT-3.5 performance added for comparison.

Table 2 shows that the layered fact-checking model is a massive improvement over GPT-3.5 – and any other commercially available LLM – with respect to correctness. An LLM without fact-checking will always give you an answer, be it correct or not. CodeScene's fact-checking model is able to validate the proposed code changes, and reject 98% of the incorrect refactorings.

Before we discuss the disruptive potential that this level of AI performance enables, we need to look behind the model to understand how the AI fact-checking is possible at all.

Data as the secret sauce

The main challenge in the fact-checking is to ensure semantic equivalence between the original code and the refactored code. This is a largely unsolved research problem in academia where the problem is studied in the area of Automatic Program Repair (APR).

Potential solutions like formal methods and code similarity metrics haven't been able to reliably verify semantic correctness between a given piece of code and its fixed/refactored counterpart. So how did the CodeScene team pull this off? There are two unfair advantages at our disposal plus one critical constraint that we chose:

First, when building the fact-checking model we had access to our data lake consisting of +100,000 real-world JavaScript refactoring samples with a known ground truth (i.e. semantically equivalent or not). This made it possible for our algorithms to observe and learn patterns in successfully refactored code.

Second – and in fact a basis and pre-requisite for #1 – the data lake was built up using the automated code review capabilities of the CodeScene tool. This automated code review is deterministic and driven by the Code Health metric. This step is crucial as it directly influences the quality of the data; poor sample quality, and it won't be possible to achieve these levels of accuracy.

Third, it's important to point out that we didn't attempt to solve semantic equivalence in general. Doing so would be futile at best. Instead, we limited the fact-checking to the set of code smells identified via the Code Health metric. That way, we could be more specific in our AI prompts as well as constraining the fact-checking model to a finite number of structural changes that can be learned by our in-house models.

Summary

This benchmarking study shows that AI is nowhere near replacing humans in a coding context; today's AI is simply too error-prone, and far from a point where it is able to securely modify existing code. However, by introducing a novel fact-checking model for the AI output, we can elevate generative AI to a point where it is genuinely useful as several complex code smells can be mitigated safely. This allows us to optimize for understanding – the dominant and most human-intensive aspect – not just the narrow task of writing new code.

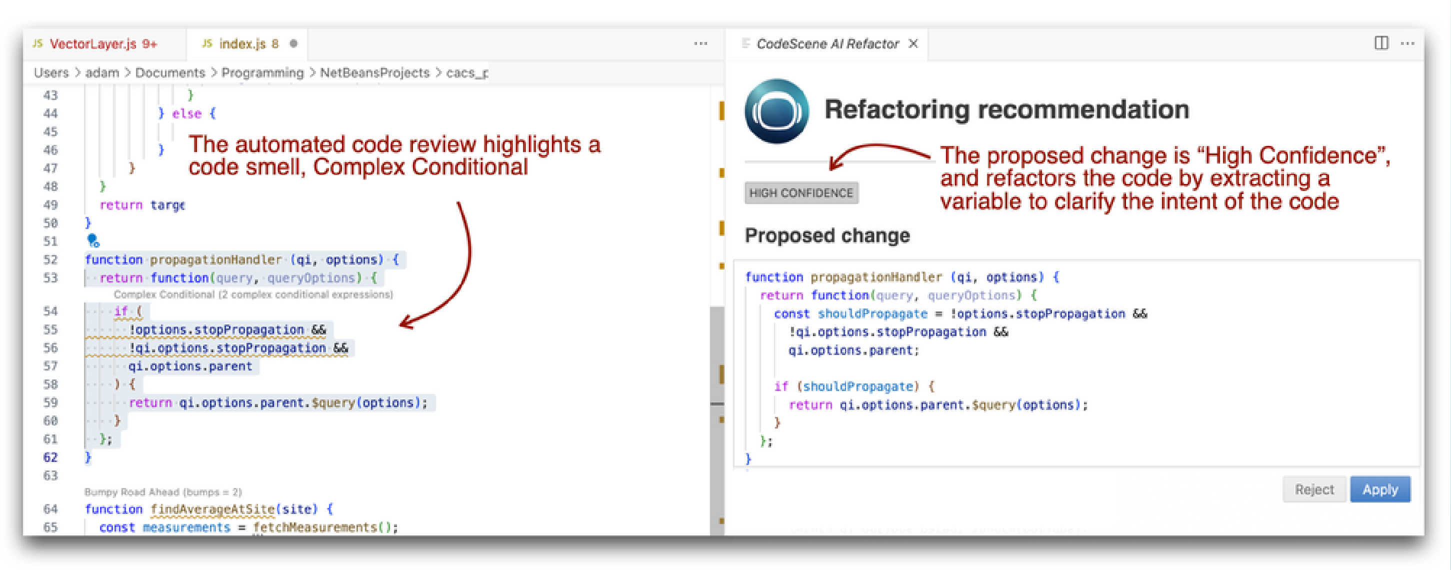
Perhaps the most intriguing possibility is the progress in technical debt mitigation made possible via this innovation. Every business manager is aware of technical debt, but few prioritize it – and even fewer manage it actively. Traditionally, there's been a hard trade-off between improving existing code vs. adding the next big feature. Predictably, improvements get the back seat despite hard numbers showing how a healthy codebase is a competitive advantage. Now that the process can be automated to a large degree, companies can finally start to reap these benefits without having to put feature development on hold.

During our research, we also couldn't help to reflect on the fact that these benchmarks on AI-assisted programming re-emphasize solid engineering practices like unit testing, code quality gates, and continuous code reviews. Those practices were always important, but perhaps even more so in the age of AI where humans need to understand and verify machine-generated code.

In our study, we too used commonly available AI models which we augmented with specific domain data to improve their refactoring performance. Yet, the key to our breakthrough wasn't AI augmentation or magic prompt engineering, but rather the ability to provide a confidence indication for each refactoring with respect to its semantic equivalence to the original code. Knowing the confidence of a proposed refactoring is a time saver.

Try the Automated Refactoring on your own code

The fact-checking innovation described in this whitepaper will be available to the general public via CodeScene. Sign-up for the beta testing waitlist at https://codescene.com/ai.


VectorLayer.js 9+
index.js 8
Users > adam > Documents > Programming > NetBeansProjects > cacs_p
CodeScene AI Refactor
Refactoring recommendation
HIGH CONFIDENCE
Proposed change
Reject
Apply

```
        }
      } else {

      }
    }
  }
  return targe
}

function propagationHandler (qi, options) {
  return function(query, queryOptions) {
Complex Conditional (2 complex conditional expressions)
    if (
      !options.stopPropagation &&
      !qi.options.stopPropagation &&
      qi.options.parent
    ) {
      return qi.options.parent.$query(options);
    }
  };
}

Bumpy Road Ahead (bumps = 2)
function findAverageAtSite(site) {
  const measurements = fetchMeasurements();
```

The automated code review highlights a code smell, Complex Conditional

The proposed change is “High Confidence”, and refactors the code by extracting a variable to clarify the intent of the code

```
function propagationHandler (qi, options) {
  return function(query, queryOptions) {
    const shouldPropagate = !options.stopPropagation &&
      !qi.options.stopPropagation &&
      qi.options.parent;

    if (shouldPropagate) {
      return qi.options.parent.$query(options);
    }
  };
}
```

About the authors

Adam Tornhill is the founder and CTO of CodeScene. Adam is a programmer who combines degrees in engineering and psychology. He’s also the author of the best-selling Your Code as a Crime Scene as well as multiple other technical books.

Markus Borg, PhD, is a senior researcher at the intersection of software engineering and applied AI. He is a principal researcher at CodeScene and an adjunct associate professor at Lund University, Sweden.

Enys Mones, PhD, is the Lead Data Scientist at CodeScene who also enjoys doing basic research. A theoretical physicist by training, his focus is applying mathematical models to understand human-computer interaction.

CodeScene

Next generation code analysis

www.codescene.com