Software Engineering Assignment

Lirong Yi, lirongy@chalmers.se

July 9, 2025

1 Introduction

My research lies at the intersection of Software Engineering (SE) and Artificial Intelligence (AI), with a particular focus on guiding Large Language Models (LLMs) to generate code that is not only functionally correct but also efficient in terms of execution time and memory usage. While models like GPT, Gemini, and Claude have made significant progress in automated code generation, studies show that their outputs are often inefficient. As LLM-generated code becomes more integrated into real-world software systems, these inefficiencies can lead to degraded application performance, increased operational costs, and greater energy consumption.

The core problem addressed in my work is the disconnect between the correctness and performance of LLM-generated code. My goal is to explore techniques that steer LLMs toward generating code that satisfies both functional and non-functional requirements. To that end, I am developing a benchmark suite composed of real-world Java programming tasks. This benchmark supports a systematic evaluation of LLM-generated code by comparing it against expert-written implementations. The evaluation includes both runtime behavior and resource usage, as well as functional correctness. This dual focus aims to identify typical inefficiency patterns and inform the development of new software engineering techniques for improving LLM-driven code generation.

2 Robert's Lectures

2.1 Data Validation

Data validation ensures that inputs, outputs, and intermediate data meet expected formats and quality standards. In traditional software systems, it involves checking user inputs or system-generated data to ensure they follow required rules. In the context of AI and machine learning, the scope is broader—it includes the quality, relevance, and consistency of training data, as well as runtime data used by models.

This concept is integral to my research. First, prompts to LLMs are a form of input data and must be validated—not just syntactically, but in terms of their ability to elicit correct and efficient code. One goal of my project is to investigate how to craft prompts that incorporate performance-related guidance. Second, I rely on a curated dataset of real-world Java performance problems and corresponding expert-written fixes, which serve as the ground truth for evaluating LLM output. Ensuring the correctness of these examples is essential, as they form the foundation for my benchmark. Finally, the code produced by LLMs must also be validated, not only for syntax and functional correctness but for runtime efficiency. My benchmark thus acts as a comprehensive validation framework, assessing the quality of LLM-generated code on both functional and performance criteria.

2.2 Quality Assurance (QA)

Quality Assurance (QA) involves ensuring that software systems meet defined quality standards across the entire development lifecycle. Rather than just identifying defects, QA focuses on building processes that prevent them.

This concept underpins my research. While traditional QA targets code written by humans, my work applies QA principles to the code generation process of LLMs. I treat performance—measured by execution time and memory usage—as a key quality attribute, alongside functional correctness. By creating a standardized Java benchmark, I provide a structured way to evaluate and improve the quality of code generated by LLMs. My benchmark acts as a QA framework: it highlights inefficiencies, identifies common issues, and guides prompt engineering or fine-tuning strategies that can lead to better LLM behavior. The ultimate aim is not just assessment but prevention—reducing the occurrence of inefficient code through improved generation strategies.

3 Guest Lectures

3.1 Requirements Engineering

Julian's lecture emphasized the importance of capturing both functional and non-functional requirements early in the software development lifecycle. This is especially relevant when considering LLMs as coding assistants or autonomous developers.

Current LLMs are generally optimized to meet functional requirements (e.g., producing syntactically correct and compilable code) but often ignore non-functional ones such as performance, memory usage, or platform compatibility. My research treats performance as a first-class requirement and explores how to encode it into prompts or provide feedback mechanisms that align LLM behavior with these goals. This connection to requirements engineering is valuable, as it provides a structured lens through which to think about aligning LLM-generated code with broader system goals.

3.2 Behavioral Software Engineering

Behavioral Software Engineering explores how cognitive and social factors influence software development. One of its key insights is that developers are not always rational—they rely on heuristics, are influenced by habits, and may make decisions based on trust rather than analysis.

This perspective is crucial in understanding how developers interact with LLM-generated code. Even when the code compiles and passes basic tests, developers may trust it without evaluating its efficiency. This can lead to performance regressions being overlooked. My research addresses this issue by providing clear performance evaluations of generated code. In future tools, I plan to present this information in a developer-friendly format—e.g., highlighting bottlenecks or suggesting faster alternatives—so that developers can make informed decisions without cognitive overload. This aligns with the human-centered focus of behavioral software engineering and aims to bridge the gap between AI tools and developer behavior.

4 Data Scientists versus Software Engineers

After reading chapters ”Introduction” and ”From Models to Systems” from the Machine Learning in Production book, here's my view on the differences between data scientists and software engineers and how these roles might change in the future.

4.1 Differences Between Data Scientists and Software Engineers

I agree with the book's distinction between the two roles. Data scientists focus on exploring data and building accurate models, often in a lab setting. Software engineers, on the other hand, build systems that are reliable, fast, and maintainable in real-world environments.

The book's example of Sidney's start-up shows this clearly. Sidney created a great model for transcription, but turning it into a working product required solving engineering problems like low latency, scalability, and smooth deployment. A good model alone isn't enough; it has to be part of a well-built system.

4.2 The Future of These Roles

I don't think these roles will fully merge into one. Instead, I believe they will overlap more as people from each side learn skills from the other.

Data scientists will need to write cleaner, production-ready code. Software engineers will need to understand that machine learning models can be unpredictable and design systems to handle failures.

Specialized roles like Machine Learning Engineers (MLEs) are growing. MLEs combine software skills with ML knowledge to help bring models into production. We'll likely see more roles like MLOps Engineers and Data Engineers working alongside data scientists and software engineers.

In summary, data scientists and software engineers will stay distinct, but the wall between them is coming down. Collaboration and new hybrid roles will help turn great models into successful products.

5 Paper Analysis

5.1 Rule-Based Assessment of Reinforcement Learning Practices Using Large Language Models

This paper by Ntentos, Warnett, and Zdun, introduces a rule-based framework for assessing the quality of reinforcement learning (RL) training code. It tackles a practical issue in AI engineering, how to ensure RL pipelines follow best practices, by automating conformance checks using both heuristic methods and LLMs. Their 31 architectural rules target areas like checkpointing, hyperparameter management, and configuration. The study finds that LLM-based detectors outperform heuristic ones, especially for complex or non-standard code patterns. Importantly, they design these detectors to be modular and traceable, making LLMs more explainable and suitable for software engineering workflows.

The paper strongly relates to my research on guiding LLMs to generate efficient Java code, albeit from a different perspective. While my work focuses on using LLMs for code generation, the paper uses LLMs for code assessment. The key connections are:

Automated Assessment of a Quality Attribute: The paper assesses RL code for conformance to architectural best practices. My research assesses LLM-generated code for a different, non-functional quality attribute: performance efficiency. The underlying principle is identical: using an automated approach to verify that a piece of code meets a specific quality standard. The benchmark suite I am developing is, in essence, a specialized "detector" for performance issues.

Codifying Rules: The paper defines 31 explicit, binary rules for what constitutes good RL practice. My research implicitly does the same for performance. My goal is to identify the characteristics of efficient code (e.g., optimal algorithm choice, proper data structure usage) and use them to guide LLMs. The paper's formal rule-based structure provides a model for how I could formalize these performance characteristics into a concrete set of rules.

LLMs as both Generator and Assessor: The paper establishes the capability of LLMs to act as effective code analyzers. This creates a fascinating synergy with my research. One could envision a system where one LLM generates the Java code (my research), and a second, specialized LLM assesses that generated code for efficiency issues (the paper's concept), creating a closed-loop quality control system.

To illustrate the real-world value, consider a project like Cursor, a developer assistant platform that uses LLMs to provide real-time code suggestions, refactorings, and intelligent diagnostics directly inside an IDE. Cursor depends on generating and modifying code that developers can trust, not only to be correct but also to be efficient and maintainable. The rule-based assessment framework from the paper could be integrated into Cursor to validate any code snippets involving machine learning pipelines, flagging suboptimal RL practices before they reach production. For example, if a developer accepts an LLM-generated RL training loop, the system could immediately alert them if it lacks robust checkpointing logic or uses fixed hyperparameters.

The paper also inspires several long-term adaptations to my research project. Instead of relying on implicit notions of efficiency, I could develop a formal catalog of "Performance Rules for LLM-Generated Java," modeled directly on the paper's Table I. These rules would range from simple (e.g., "R-P01: Avoid creating objects inside a tight loop") to complex (e.g., "R-P09: For the given data access pattern, a HashMap should be used instead of a linear search"). This would make my research more rigorous and my guidance mechanisms more explicit.

5.2 Mutation-based Consistency Testing for Evaluating the Code Understanding Capability of LLMs

The paper talks about a new way to test if LLMs really understand code, not just write it. Here are the main ideas.

Core Idea 1: A New Testing Method (MCT). The paper made a new method called Mutation-based Consistency Testing, or MCT. The main job of MCT is to check if an LLM can find small conflicts between code and the text that describes the code (like comments). For example, if a comment says "this function adds two numbers", but the code actually subtracts them, can the LLM see this problem?

Core Idea 2: Using Code Mutation. To make these conflicts for testing, the method uses something called "code mutation". This is a technique from software testing. It makes small changes in the working code, like changing a + to a -, or deleting a line. These small changes create mutants. The mutants are used to test if the LLM is smart enough to see that the code no longer matches the description.

This is very important for Software Engineering for AI. Now many people use AI tools like Copilot to help them write code. But we need to be sure these tools really understand what they are doing. If an LLM just copies patterns without deep understanding, it can make very hard-to-find bugs. This MCT method gives us a way to do quality check on the LLM's understanding ability. It helps us trust these AI tools more and helps researchers to build better, more reliable AI for software development. It's like a deeper health check for the AI's brain.

This paper is very related to my own research, even though it does a different thing. My research is about making LLMs write efficient code. This paper is about testing if LLMs understand code. We both want to improve the quality of LLMs for software engineering. The paper looks at the quality of understanding. I look at the quality of generation, specifically performance. It's like we are checking two different skills of the same student. The paper checks the student's reading skill, I check the student's writing skill. My research uses a benchmark with real-world Java performance problems to test LLMs. The paper uses a systematic way (mutation) to create test cases. Both of our works use a structured, engineering way to measure the

LLM’s ability. This is better than just guessing if the LLM is good or not. We both use data and clear steps to get results.

The paper gives me many ideas on how to make my own research better and bigger in the long term. My project is about guiding LLMs to generate efficient code. The paper uses mutation for testing. I can use the same spirit for guiding generation. My research is about guiding, and one way to guide is to show bad examples. These bad examples can be like ”negative mutants”. By showing the LLM these ”anti-patterns”, I am not just telling it what to do, but also very clearly what not to do. This can be a very powerful way to guide the LLM to the correct, efficient solution. This directly adapts the paper’s core idea of using mutants from a testing tool to a teaching tool.

6 Research Ethics & Synthesis Reflection

I scanned the CAIN conference website using terms like LLM, code generation, and code quality. I filtered papers by titles, abstracts, and key sections to ensure relevance.

I wrote my analysis in my own words, using the LLM only to help with the writing and clarity, not to generate ideas.