Thesis Report

Luca Beltrami

June 2025

ABSTRACT

In recent years, Artificial Intelligence has become more sophisticated, useful, and almost necessary for a wide range of tasks across many sectors of society. Its rapid development is already transforming many professions, raising important questions: How similar is AI-generated work to human work? What are the key differences?

In the field of informatics, these questions are particularly relevant, as a growing portion of code is now written or assisted by tools like ChatGPT, GitHub Copilot, or DeepSeek. My initial goal was to compare code written by humans with code written by AI, using the "AI code detector" tool developed by OpenAI. However, during the research, it became evident that this detector was not reliable enough to distinguish AI-generated code from human-written code with sufficient accuracy.

Due to this limitation, the focus of the work shifted: instead of comparing human and AI code directly, I concentrated on analyzing code generated by AI. This was done by selecting a dataset of AI-generated code and running Semgrep, a static analysis tool, to extract meaningful patterns and insights about its structure, practices, and potential vulnerabilities.

This project aims to provide a deeper understanding of how AI writes code and to build a valuable dataset that could support future research on AI-generated software development.

1. Introduction

Nowadays, Artificial Intelligence (AI) plays an increasingly important role in our everyday tasks, and software development is no exception. AI tools such as ChatGPT, GitHub Copilot, and DeepSeek are widely used to assist in coding: they can generate code snippets, automate repetitive processes, and support developers in understanding and modifying existing code. This innovation is transforming how developers work, making certain tasks faster and more accessible, even to those with less experience.

However, this progress comes with potential risks. AI-generated code can introduce bugs, inefficiencies, or subtle vulnerabilities particularly in more complex or context dependent scenarios where deeper semantic understanding is required. While AI performs well on short, well-defined tasks, its performance may degrade on larger, multi-layered programming problems that demand abstract reasoning or domain specific knowledge.

Motivated by this growing presence of AI in software development, our initial goal for this thesis was to explore how AI-generated code differs from human-written code, focusing on dimensions such as structure, clarity, maintainability, and adherence to best practices. To investigate this, we aimed to build a dataset of code samples and use OpenAI's "AI code detector" to classify whether each piece of code was written by a human or by an AI. This would have allowed a comparative study between the two styles of code.

However, during the development of this project, we observed that the AI code detector did not provide reliable or consistent results. The tool often misclassified code or gave ambiguous confidence levels, making it unsuitable for a rigorous analysis of stylistic and structural differences between human and AI code.

As a result, the focus of the thesis shifted. Instead of attempting a direct comparison, we decided to concentrate on characterizing code that we knew had been generated by AI. We gathered a dataset of AI-generated code samples written in nine popular programming languages (Python, JavaScript, Java, TypeScript, C++, Go, PHP, C, and C#), primarily from commits associated with known AI-generated contributions on GitHub. To analyze this dataset, we used Semgrep, a static code analysis tool that detects code patterns, potential vulnerabilities, and deviations from best practices.

By applying Semgrep to the AI-generated code, we were able to extract quantitative data about common coding issues, unsafe patterns, and structural choices made by AI systems. Although this approach no longer allowed for a direct comparison with human-written code, it still provided valuable insights into the characteristics and limitations of current AI-generated software.

Through this process, the thesis aims to contribute to the understanding of how AI writes code in real world contexts, and to provide a curated dataset that could be useful for further research in this rapidly evolving area.

1.1. Report Structure

In Section 2: "Related Work", I will explore and critically evaluate existing research that investigates the similarities and differences between human-written and AI-generated code. Several recent studies have attempted to assess the quality, readability, and maintainability of code produced by large language models. Some works focus more on patterns and common programming idioms, while others investigate code correctness and vulnerability rates. Furthermore, I will examine studies from the software engineering and machine learning communities that discuss the implications of AI assistance in professional and educational contexts.

In Section 3: "Study Design", I will dive deeper into the question: "To what extent is AI-generated and manually written code different?" To approach this, I first curated two datasets, one consisting of code written entirely by human developers, and another composed of code generated by AI systems (primarily ChatGPT and Copilot). I focused on commonly used programming languages and ensured comparable code functionality across both sets. I then identified specific dimensions of analysis, including code structure, naming conventions, comment usage, indentation and formatting consistency, function length, and others. The study also considers contextual factors such as code origin (e.g., educational, professional, open source) and task complexity to ensure a fair comparison.

In Section 4: "Results Discussion", I interpret the results of the comparative analysis between human-written and AI-generated code. Key patterns emerged that underscore the distinct tendencies of each source. For instance, AI-generated code tended to use more generic variable and function names, often closely reflecting the prompt given, while human-written code showed more context-aware and meaningful naming conventions. In terms of structure, AI code exhibited consistent indentation and formatting due to the model's training on style-conformant code, whereas human code showed greater stylistic variability. Furthermore, AI-generated code was more likely to include inline comments. However, some AI-generated solutions also demonstrated elegance in handling edge cases, suggesting that, in certain scenarios, the AI mimics best practices learned from large codebases. The discussion also touches on the limitations of AI-generated code, including occasional lack of modularity, redundancy, and superficial understanding of context. Overall, the findings highlight differences that have implications for code review practices, AI tool adoption, and programming pedagogy.

In Section 5: "Conclusion and Future Work", I reflect on the evolution of the project and its key contributions. The original goal was to compare AI-generated code with human-written code using OpenAI's AI Code Detector. However, due to the detector's unreliability particularly its high false positive rate on older or highly structured code the study pivoted toward analyzing code strongly suspected to be AI-generated. To support this, a GitHub mining pipeline was developed using a curated list of AI-related n-grams, which led to the collection of AI-generated commits. These commits were analyzed using Semgrep to identify vulnerabilities, revealing frequent security issues such as SQL injection, XSS, and poor cryptographic practices, especially in Python and JavaScript frameworks like Flask and Express. The analysis highlighted that AI-generated code often lacks contextual understanding and tends to replicate insecure patterns.

The chapter also outlines several promising directions for future research. These include returning to the original comparative analysis with improved classification tools, refining the commit mining process, expanding the dataset, and incorporating more advanced vulnerability detection techniques. Additionally, future studies could explore model-specific labeling and develop risk scoring systems to assess the security of AI-generated code more systematically. These suggestions aim to bridge the gap between functionally correct and secure AI-generated code, ultimately advancing the safe integration of generative AI in software engineering workflows.

2. Related Work

The paper [1] examines how AI-generated code compares to human-written code in terms of software metrics and bug occurrences. The authors focus on Java solutions to 90 LeetCode problems, equally split across easy, medium, and hard levels. They generated code using GitHub Copilot and selected the top voted human-written solutions from LeetCode. Static code analysis tools such as SpotBugs and PMD were used to identify bugs, while software metrics were extracted using the Understand tool. Four metrics were prioritized: AvgCountLineCode, AvgCyclomatic, CountLineCodeDecl, and CountLineCodeExe. The study aimed to answer three questions: (1) What relationships exist between metrics and bugs in AI vs. human code? (2) How do code size and complexity differ? (3) How does bug density compare? Two key metric-bug relationships were found

in both types of code. The bug `DLS_DEAD_LOCAL_STORE` positively correlated with both `CountLineCodeDecl` and `CountLineCodeExe`. The bug `AvoidLiteralsInIfCondition` correlated with `AvgCyclomatic`.

AI-generated code tended to be longer than human-written code in most cases. Cyclomatic complexity was evenly distributed between both. Bug density was generally lower in AI-generated code, though correctness was often an issue, especially with harder problems. Only 5 of 30 hard problems were correctly solved by AI. In contrast, human-written code was assumed to be correct. The study suggests that AI-generated code may have less errors in syntax but more limited in logic for complex tasks. The authors note that correctness was not directly factored into the bug-metric correlation analysis. They acknowledge threats to validity, such as reliance on LeetCode problems and a single AI tool. The study provides insights into strengths and weaknesses of AI coding and helps inform developers about when and how AI can be trusted in software development.

The authors of the paper [2] created the FormAI dataset, consisting of 112,000 compilable C programs generated using GPT-3.5-turbo. These programs were written to perform a wide variety of tasks with diverse complexity. Each generated program was designed to be independent and self-contained, without requiring external input or dependencies. To assess security, the team used the ESBMC (Efficient SMT-based Bounded Model Checker), a formal verification tool. ESBMC checks each program for vulnerabilities such as buffer overflows, integer overflows, null pointer dereferencing, and more. The tool labels each vulnerability with type, line number, function name, and CWE (Common Weakness Enumeration) identifier. Over 197,800 vulnerabilities were found in total across the dataset. 51.24% of the programs were found to contain at least one vulnerability. The most common issues were Buffer Overflow, Arithmetic Overflow, Array Bounds Violations, and Dereference Failures. On average, each vulnerable file contained multiple issues, highlighting the compounding risk in AI-generated code. The study shows that even simple prompts to LLMs can produce syntactically correct but unsafe code. A dedicated effort was made to ensure code diversity using prompt combinations from 200 task types and 100 coding styles. About 98% of programs successfully compiled with the GNU C compiler; ESBMC classified 106,138 programs. The dataset enables detailed classification and benchmarking of static and dynamic vulnerability scanners. It can also support training machine learning models in secure code generation and vulnerability prediction. The team discovered and helped fix bugs in ESBMC, Clang, and CBMC using the generated code. The authors conclude that LLMs like GPT-3.5 frequently produce vulnerable code, necessitating caution and improved safeguards when using AI for software development.

The paper [3] conducts a thorough empirical study on the reliability and quality of code generated by ChatGPT (GPT-3.5) for programming tasks. The authors created a benchmark of 2,033 LeetCode problems (easy, medium, hard) and used ChatGPT to generate 4,066 code snippets in Python and Java. They first evaluated the correctness of the generated code using LeetCode's test suites. About 66% of Python and 69% of Java solutions passed all tests. However, accuracy declined with increasing task difficulty and for tasks introduced after 2021—likely due to the model's limited training data. They found common code issues, even in correct solutions. Around 47% of the programs had maintainability issues, and 27% had incorrect outputs. Static analysis tools (e.g., Pylint, Flake8, PMD, Checkstyle) revealed issues like unused variables, poor structure, and excessive complexity. Runtime and compilation errors were also cataloged, especially for newer

or more complex problems. Results showed that ChatGPT could self-repair 20–60% of the issues, with static feedback being more effective for maintainability bugs and simple prompts performing better for logic or performance issues. However, repairs sometimes introduced new issues, especially when using vague prompts. The paper concludes that while ChatGPT shows potential for code generation, its output often requires human oversight and revision, especially for complex or newer tasks.

The authors of [4] evaluates the correctness and quality of code generated by ChatGPT (GPT-3.5), GPT-4, and Google Bard. The authors selected three complex Java programming problems from university exams to ensure they were not part of the LLMs' training data. The generated code was tested using 32 custom test cases and analyzed with SonarCloud to assess metrics like lines of code, cyclomatic complexity, cognitive complexity, and code smells. GPT-4 achieved the best results, passing 29/32 test cases, followed by GPT-3.5 with 28/32, and Bard with 22/32. Bard failed to produce a correct solution for one entire problem. GPT-4 also produced less complex code. None of the solutions showed bugs, vulnerabilities, or duplications. All models required minimal but necessary additional prompting to make the code runnable. The study highlights that while LLMs can generate functional code, human oversight remains essential. GPT-4 showed moderate advantages over the other models in both correctness and quality. The paper concludes that LLMs adhere fairly well to coding standards but cannot yet fully replace human developers.

Instead, in the paper [5], the authors investigates the security implications of using generative AI, specifically ChatGPT, compared to traditional developer resources like StackOverflow. The authors selected 108 Java code snippets from SO that addressed security-related questions and then used the same questions to prompt ChatGPT (GPT-3.5-turbo) to generate equivalent code. After compiling the snippets from both sources, they analyzed them using GitHub's CodeQL, a static analysis tool that detects vulnerabilities based on Common Weakness Enumerations (CWEs). The results showed that ChatGPT-generated code contained 248 vulnerabilities, while StackOverflow code had 302, marking a statistically significant 20% reduction in the number of vulnerabilities for ChatGPT. Additionally, ChatGPT produced 19 different CWE types compared to 22 in SO, and the overlap in specific vulnerabilities between the two sources was only 25%. Both platforms had a total of 274 unique vulnerabilities, with the most common being related to weak cryptographic practices and resource management issues. Interestingly, ChatGPT code had more instances of hard-coded credentials, whereas SO code had more issues with pseudo-random number generation. Despite ChatGPT's relatively better performance, the authors emphasized that neither source can be trusted blindly, as both contribute to insecure code. They advocate for the consistent use of secure development practices, such as static analysis tools, to mitigate risks. The study also raises questions about why LLMs like ChatGPT, trained on internet data including SO, might produce less vulnerable code.

3. Study Design

3.1. Data Collection

As the starting point for the practical phase of my research, I explored and evaluated effective methods for identifying code that was likely generated by artificial intelligence.

This is an inherently complex and rapidly evolving area of study, primarily because AI models particularly those designed to assist with programming are continuously improving in their ability to produce code that closely resembles human-written logic in both structure and style. The boundaries between human-generated and AI-generated code are becoming increasingly blurred, which makes the challenge of distinguishing between the two technically demanding.

To ground my investigation in real world data, I turned to GitHub, which serves as one of the largest and most diverse repositories of publicly available source code globally. With millions of developers contributing to open-source and private projects alike, GitHub offers a rich landscape for studying trends in software development and for collecting samples of code that may reflect AI influence.

My initial focus was on developing a reliable strategy for retrieving code samples from GitHub that were highly likely to have been authored, or at least assisted, by AI tools. A crucial part of this step involved defining and validating a set of phrases referred to as n-grams that would act as filters when searching through commit messages. In this context, n-grams are short sequences of n words, in the mining process I used 3- and 4-grams (e.g., "used ChatGPT to", "generated using Copilot") which are likely to appear when developers reference AI-assisted programming tools in their commit logs.

Before finalizing the list of n-grams, I conducted a manual evaluation phase. I tested 30 candidate phrases by running preliminary GitHub searches and manually inspecting the returned commits. With a manual analysis I decided on 22. This involved reading the commit messages and associated code changes to assess whether the commit had indeed been written with the assistance of tools like ChatGPT or GitHub Copilot. In total, this manual process led to the identification of 753 commits that matched such indicative patterns and showed clear signs through comments, descriptions, or behavioral traits of AI involvement. This manual inspection proved to be the most effective initial strategy, as it allowed me to refine my set of n-grams and ensure they produced high quality matches when applied at scale.

Once I had selected the following n-grams as the mining queries: used chatgpt to, generated by chatgpt, changes of chatgpt, chatgpt to generate, help of chatgpt, co-authored by chatgpt, co-authored-by: chatgpt, coauthored by chatgpt, chatgpt to implement, chatgpt to write, chatgpt to create, used copilot to, generated by copilot, changes of copilot, copilot to generate, help of copilot, co-authored by copilot, co-authored-by: copilot, coauthored by copilot, copilot to implement, copilot to write, copilot to create. I implemented a mining pipeline using the GitHub API. The pipeline queried public commits containing any of the selected phrases and collected structured metadata for each match. The results were saved in .jsonl format (JSON Lines), where each line represented a different commit.

For each commit, the following fields were stored:

Link_to_commit: a direct URL to the commit on GitHub, allowing traceability and further manual review.

n-gram matched: the specific phrase that triggered the match, useful for validating the filtering logic.

n_lines_longer_change: the number of lines added in the longest chunk added in the commit (added minus removed), which serves as a proxy for the extent of modification.

n_files_impacted: the number of files touched by the commit, providing context for the scope of the change.

longest_added_chunk: the size (in lines) of the largest contiguous block of newly added code, which is particularly important because this chunk was later used as input for the AI Code Detector.

These fields were selected not only to capture the nature of the code changes, but also to facilitate later stages of analysis like storing the n-gram matched for further analysis.

3.1.1 AI-generated code

To test the effectiveness of the OpenAI AI Code Detector, I randomly sampled 20 commits from the n-gram-filtered dataset, focusing on those that contained strong indicators of AI involvement (e.g., commit messages explicitly mentioning ChatGPT or Copilot). From each commit, I extracted the largest newly added code chunk and submitted it to the AI detector, which returned a confidence score indicating how likely the code was to have been generated by an AI system.

The results showed that most commits received high confidence scores, typically ranging from 85% to 98%, confirming that the detector correctly identified these code fragments as AI-generated. These scores were accompanied by an explanation of the decision, including syntactic and stylistic patterns observed in the code.

However, a few cases received significantly lower scores (10%–60%). This may be explained by post generation modifications made by developers, human-style formatting that diluted AI-identifiable traits, or instances in which the commit message referenced AI tools without actual AI involvement in the code itself. In these edge cases, the phrase match could be a false positive.

Overall, these findings suggest that the AI Code Detector aligns well with human-reported AI use, especially when metadata (like commit messages) clearly indicates such involvement. However, the variability in confidence scores also highlights the limitations of relying on the tool in isolation.

3.1.2 Manually written code

To establish a baseline for comparison, I selected a second set of 20 commits known to be human-written. Specifically, these commits were randomly drawn from code pushed prior to 2020, a period before tools like ChatGPT and GitHub Copilot were available to the public. This ensured that the code had not been influenced by modern generative AI tools.

Using the same methodology, I extracted the longest added code chunk from each commit and submitted it to the AI Code Detector. Surprisingly, a significant portion of these human-written samples received high confidence scores from the detector, with some even exceeding 85% or 90%.

While a few commits were correctly assigned low scores (as low as 3%, 4%, or 12%), the presence of the majority of the data resulting in false positives raises concerns about the tool's reliability. In these cases, the AI detector appears to confuse regular coding

patterns with generative AI outputs. This makes it unsuitable as a standalone classifier for determining authorship and underscores the need for additional, context-aware analysis.

3.2. Detecting code quality issues using Semgrep

Given the limitations of the AI Code Detector, especially in ambiguous or borderline cases, I integrated an additional layer of analysis using Semgrep a static analysis tool that can scan code for specific patterns, vulnerabilities, and compliance with coding standards.

To prepare the input for Semgrep, I first revisited the JSON file generated during the data collection phase. Although this file contained metadata and code snippets, it only included the changed chunks of code, not the full source context. Since Semgrep is designed to analyze full source files for structural patterns and vulnerabilities, I needed to obtain the complete version of each file affected by a commit.

To address this, I built a script that retrieved the full file content associated with each commit. Each file was then saved in a separate location, allowing Semgrep to scan them individually. This step was crucial, as partial code fragments would limit the effectiveness and accuracy of static analysis tools.

The Semgrep pipeline was configured to apply a set of predefined rules tailored to detect common vulnerabilities, insecure coding patterns, and anti-patterns across multiple programming languages. These included checks for the use of insecure functions, lack of input validation, poor exception handling, and other security-relevant issues. The results of each analysis were stored in a structured format for later comparison and review.

Semgrep serves two key purposes in this context:

It enhances the interpretability of code analysis by applying transparent, rule based checks.

It enables investigation into whether AI-generated or human-written code differs in terms of adherence to software quality standards and security best practices.

By examining what types of patterns emerge across code samples, I was able to build a more nuanced understanding of coding practices, regardless of authorship. Moreover, the combination of metadata based filtering, confidence scoring from the AI detector, and rule based static analysis provides a more robust framework for studying the influence of generative AI on modern software development.

3.3. Replication Package

The data mining and analysis for this thesis were primarily conducted using Python. The full project with the code is available on GitHub at: https://github.com/Loca0307/Thesis.

4. Results discussion

4.1. OpenAI code detector

I evaluated 40 GitHub commits using the OpenAI AI code detector to assess its ability to distinguish between AI-generated code and human-written code. The analysis was divided into two groups:

- **Group A (n = 20)**: Commits retrieved using an n-gram based mining strategy designed to identify AI-generated code.
- **Group B (n = 20)**: Random commits from before 2020, a period preceding the widespread availability of AI coding tools.

The AI detection scores (expressed as percentage likelihood of being AI-generated) are summarized below.

Comparison of Group A and Group B

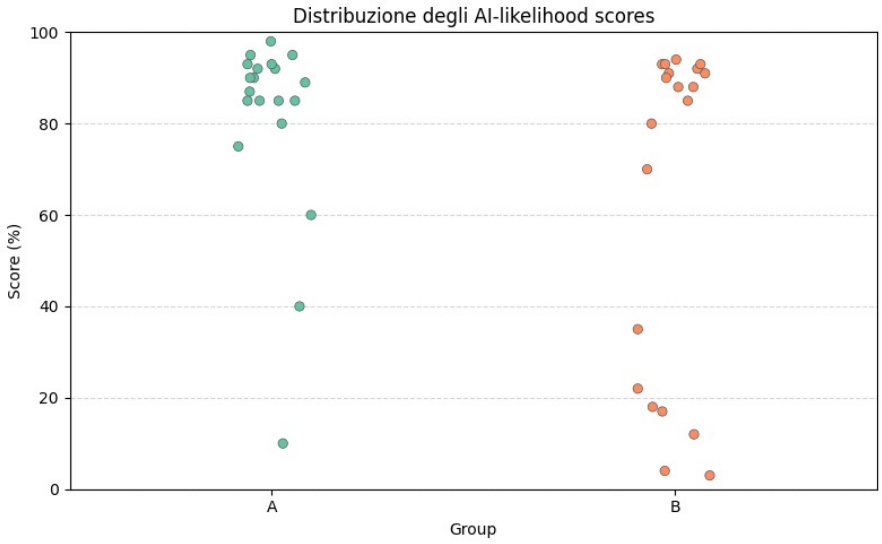

Figure 1: Distribution of AI-likelihood scores for Group A (targeted commits) and Group B (random pre-2020 commits)

The scatter plot visualizes the AI-likelihood scores for two groups of commits. Group A includes targeted commits identified through an n-gram mining strategy, while Group B consists of random commits made before 2020, which are presumed to be human-written.

The scores in **Group A** are generally high and clustered near the top end of the scale. Specifically, 17 out of 20 commits score above 75%, suggesting that the mining strategy effectively identifies code likely to be AI-generated. A few notable outliers (scores around 10%, 40%, and 60%) indicate some imprecision of this method of analysis.

In contrast, **Group B** displays a wider and more varied distribution. While several commits received low scores (e.g., below 20%), which aligns with expectations for human-written code, many others surprisingly scored above 85%. These represent potential false

positives, as all commits in this group were made before the widespread use of generative AI.

The visual comparison highlights a general separation between the two groups but also reveals an overlap that suggests limitations in the model's precision.

4.1.1 Analysis

The results highlight two main findings:

- **Effectiveness of the n-gram mining strategy**: Group A commits had predominantly high AI-likelihood scores, indicating that the chosen n-grams were successful in capturing stylistic or structural features associated with AI-generated code. However, some commits in this group received low scores (e.g., 10%, 40%), suggesting that the strategy is not flawless and may also include non-AI code.
- **Limitations of the AI code detector**: A substantial number of commits from Group B despite predating the rise of AI coding assistants were classified with high AI-likelihood scores. Specifically, 50% (10 out of 20) of these commits were above 85%, clearly representing false positives. This undermines the reliability of the detector in distinguishing AI-generated code from human-written code, especially in older commits or when humans adopt efficient, structured coding styles.

4.1.2 Summary Statistics

Group	Mean Score	Std Dev	Scores $\geq$ 85%
Group A	81.3%	$\sim$22.0	17/20
Group B	63.1%	$\sim$36.2	10/20

Table 1: Summary statistics for detection scores

While Group A scores are both higher and more consistent, Group B shows a broad spread, reflecting detector uncertainty and a high false positive rate.

These findings suggest caution when relying solely on automated AI code detectors for attribution. The high rate of false positives in Group B implies that human-written code can frequently resemble AI-generated output, particularly in clean, modular, or repetitive code. Similarly, the presence of a few low-scoring commits in Group A indicates that AI code may evade detection under certain stylistic choices or formatting.

4.2. Semgrep analysis

Based on the findings presented in Table 4 table, the analyzed file contains a total of 55 issues distributed across various severity levels: 29 warnings, 16 errors, 2 critical issues, and 8 informational messages. Although the number of critical issues is relatively low—only 2 out of 55—suggesting that the file does not pose an immediate high-risk threat, the high volume of warnings and errors indicates notable problems in terms of code quality, maintainability, and adherence to secure coding practices. Compared to the average open source file, which typically contains significantly fewer findings (around 2–5), this file demonstrates a worse than average performance.

Table 2: Findings by Severity

Severity	Count
WARNING	29
ERROR	16
CRITICAL	2
INFO	8

The Semgrep analysis of the AI-generated looking code commits reveals a range of security weaknesses spanning multiple CWE categories. Notably, there is a high concentration of critical vulnerabilities associated with data integrity and injection risks, such as CWE-89 (SQL Injection) with 8 occurrences and CWE-79 (Cross-site Scripting - XSS) with 7, suggesting frequent improper handling of user input and failure to neutralize special elements. The presence of CWE-798 (Hard coded Credentials) and CWE-522 (Insufficiently Protected Credentials), totaling 10 findings, raises serious concerns about insecure authentication practices. Issues like CWE-134 (Externally Controlled Format String) and CWE-502 (Deserialization of Untrusted Data) further indicate potential exploitation vectors that could lead to remote code execution. While lower-frequency issues such as CWE-611 (XXE), CWE-939, and CWE-330 appear less often, their impact remains significant due to their exploitability in specific contexts. Overall, the analysis suggests that AI-generated looking code commits may replicate patterns that lead to common but severe vulnerabilities.

Table 3: Findings by CWE

CWE	Count
CWE-798: Use of Hard coded Credentials	4
CWE-611: Improper Restriction of XML External Entity Reference	1
CWE-78: Improper Neutralization of Special Elements in OS Command	3
CWE-939: Improper Authorization in Handler for Custom URL Scheme	1
CWE-89: Improper Neutralization in SQL Command	8
CWE-489: Active Debug Code	3
CWE-502: Deserialization of Untrusted Data	3
CWE-676: Use of Potentially Dangerous Function	2
CWE-79: Improper Neutralization of Input in Web Page Generation	7
CWE-319: Cleartext Transmission of Sensitive Information	4
CWE-1333: Inefficient Regular Expression Complexity	1
CWE-352: Cross-Site Request Forgery (CSRF)	2
CWE-522: Insufficiently Protected Credentials	6
CWE-134: Use of Externally-Controlled Format String	4
CWE-330: Use of Insufficiently Random Values	1
CWE-116: Improper Encoding or Escaping of Output	1
CWE-22: Improper Limitation of a Pathname	2
CWE-338: Use of Cryptographically Weak PRNG	1

The analysis of technologies associated with the detected vulnerabilities reveals clear trends in where security issues are most concentrated. Python and JavaScript stand out

as the most problematic technologies, each appearing in 11 findings. Their widespread usage and dynamic nature likely contribute to the higher risk of introducing security flaws, especially in web related contexts. Within the Python ecosystem, frameworks like Flask (6 findings), Flask-WTF (2), and WTForms (2) were often involved, indicating that web input handling and form validation in Python projects are common sources of vulnerability. Similarly, Express (9 findings) and other Node.js based technologies (Node.js: 4, Nodejs: 3, MySQL2: 2) were frequently flagged, reflecting the complexity of securing JavaScript backends.

On the other end, technologies like Firebase, C, and C# were only associated with a single vulnerability each, suggesting either limited usage in the analyzed commits or a lower incidence of detectable security issues in those ecosystems. It's worth noting, however, that a low count doesn't necessarily imply better security—it may also reflect less coverage by Semgrep rules or fewer code examples.

Overall, the data implies that dynamic, high-level, web-focused languages and frameworks (especially in Python and JavaScript) are more prone to misuses and oversights that lead to vulnerabilities.

Table 4: Findings by Technology

Technology	Count
mysql	3
sql	3
secrets	4
python	11
flask	6
flask-wtf	2
prestodb	2
web	2
wtforms	2
pytorch	3
sqlalchemy	3
node.js	4
javascript	11
firebase	1
nodejs	3
browser	5
typescript	2
express	9
mysql2	2
java	3
go	2
csharp	1
c	1

The distribution of vulnerabilities by class provides insight into the types of security issues most prevalent in the analyzed AI-generated commits. SQL Injection and Cryptographic

Issues are tied as the most common vulnerability classes, with 8 instances each. This highlights a significant concern in how AI-generated code handles input sanitization and cryptographic practices—two areas where subtle errors can lead to severe security flaws. Close behind is Cross-Site Scripting (XSS) with 7 findings, suggesting that AI-generated front-end code often fails to properly sanitize output rendered in browsers.

Other notable classes include Hard coded Secrets, Mishandled Sensitive Information, and Improper Validation, each with 4 findings, indicating a recurring issue with securely managing credentials and enforcing correct input assumptions. Insecure Deserialization, Command Injection, and Active Debug Code were also relatively frequent, each appearing 3 times, suggesting ongoing risks in how data and commands are handled at runtime, especially in backend systems.

Conversely, Improper Authorization, XML Injection, Improper Encoding, and Denial-of-Service (DoS) each had only 1 detection, possibly due to less representation in the codebase or fewer applicable Semgrep rules. However, their presence, even in small numbers, emphasizes that AI-generated code can still introduce subtle, high-impact bugs across a broad spectrum of vulnerability types.

In summary, the prevalence of injection-based issues (SQLi, XSS, Command Injection), insecure cryptographic practices, and poor secret management suggest that AI-generated code tends to struggle with fundamental security principles like input/output sanitization and secure data handling. These results reinforce the need for robust post generation security review, especially when using AI tools in production software development.

Table 5: Findings by Vulnerability Class

Vulnerability Class	Count
Hard coded Secrets	4
XML Injection	1
Command Injection	3
Improper Authorization	1
SQL Injection	8
Active Debug Code	3
Insecure Deserialization	3
Dangerous Method or Function	2
Cross-Site-Scripting (XSS)	7
Mishandled Sensitive Information	4
Denial-of-Service (DoS)	1
Cross-Site Request Forgery (CSRF)	2
Cryptographic Issues	8
Improper Validation	4
Improper Encoding	1
Path Traversal	2

The statistics presented in Table 8 offer a summary of the distribution of individual vulnerability findings per class. The minimum number of findings is 1, while the maximum is 8, indicating that some vulnerability classes appear only once, whereas others are considerably more common. The median value of 1.0 shows that over half of the vulnerability

classes were detected just once. However, the average of 1.72 findings per class indicates a moderate skew caused by a few high frequency vulnerabilities, such as SQL Injection and Cryptographic Issues.

This pattern implies that while AI-generated code introduces a wide variety of vulnerability types, most occur infrequently perhaps due to a combination of diverse project contexts and scattered weaknesses. Still, the consistently recurring critical issues (like SQLi and XSS) highlight systemic problems that AI systems may be particularly prone to replicating. This underscores the importance of targeted security tooling and developer oversight when integrating AI-generated code

Table 6: Finding Statistics

Finding Stats	Count
Minimum	1
Maximum	8
Median	1.0
Average	1.72

5. Conclusions and Future Work

This thesis set out with the goal of comparing AI-generated code to human-written code, driven by the rising influence of tools like ChatGPT and GitHub Copilot in modern software development. The motivation stemmed from an interest in understanding whether AI-generated code differs significantly from human code in terms of structure, style, quality, and security. Initially, the plan was to use OpenAI's AI Code Detector to automatically classify code samples and analyze the differences between the two categories.

However, during the early stages of experimentation, it became clear that the AI Code Detector lacked the accuracy and reliability needed for meaningful classification. It frequently produced false positives, with many human-written code samples, especially older or highly structured ones receiving high AI-likelihood scores. This observation led to a shift in focus: rather than comparing code across authorship categories, the research narrowed its scope to analyzing code already known (or strongly suspected) to have been generated by AI.

To this end, a robust GitHub mining pipeline was developed. Using a curated set of AI-related n-grams, the system identified 750 commits likely associated with AI-generated code. These commits were then processed to retrieve the full files for analysis using Semgrep, a static analysis tool capable of identifying security vulnerabilities, unsafe patterns, and other code quality issues. The results revealed a wide spectrum of findings. The most frequent vulnerabilities included SQL injection, XSS, and poor cryptographic practices, with frameworks like Flask and Express being commonly involved. Technologies such as Python and JavaScript were the most represented among vulnerable files, possibly due to their prevalence and flexibility, but also highlighting areas where AI-generated code is most prone to misuse. The analysis also showed that while some vulnerabilities occurred only once, others recurred frequently, with an average of 1.72 findings per class. These findings suggest that AI-generated code often replicates insecure patterns and lacks the contextual awareness required for secure software development.

Despite the shift in research direction, the project succeeded in producing a curated dataset of AI-generated code, identifying prevalent security concerns, and testing the limits of AI code detection tools. It demonstrated that while AI can produce functionally correct code, it is still far from reliably secure or comparable to human developers in handling edge cases and subtle vulnerabilities.

5.1. Future Work

Several directions can improve and extend this research:

- **Complete the original comparative analysis:** With a more reliable or updated AI code detector, it would be possible to revisit the initial goal of comparing AI vs. human code across various dimensions (e.g., maintainability, readability, and structure).
- **Improve the n-gram mining process:** While effective, the current set of n-grams may still include false positives. Future work could refine this strategy by incorporating additional signals such as file metadata, time periods, or author patterns.
- **Enhance vulnerability coverage:** Semgrep rules could be extended or complemented with dynamic analysis tools (e.g., fuzzing or runtime monitoring) to uncover vulnerabilities that static analysis may miss.
- **Broaden the dataset:** The current study focused on 750 commits. Expanding this to thousands of AI-generated and human-written commits could improve statistical significance and reveal deeper trends.
- **Incorporate model-aware labeling:** Future datasets could include code explicitly labeled with the LLM model used to generate it (e.g., GPT-3.5 vs GPT-4), allowing for more granular analysis of differences in quality and security across model versions.
- **Develop risk scoring mechanisms:** Based on the types and frequency of detected vulnerabilities, a scoring system could be developed to assess the relative risk of AI-generated code, helping practitioners decide when human review is most critical.

By pursuing these improvements, future studies can help close the gap between functional and secure AI-generated code, contributing to safer and more reliable use of generative AI in software engineering.

6. References

References

[1] A Comparative Analysis between AI Generated Code and Human Written Code: A Preliminary Study https://www.computer.org/csdl/proceedings-article/bigdata/2024/10825958/23ykKzCP4RO

[2] FormAI: A Dataset of 112K Verified Compilable C Programs Generated by GPT-3.5 for Security Analysis https://arxiv.org/abs/2307.02192

[3] Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues https://arxiv.org/abs/2307.12596

[4] Studying the Quality of Source Code Generated by Different AI Generative Engines: An Empirical Evaluation https://www.mdpi.com/1999-5903/16/6/188

[5] Just another copy and paste? Comparing the security vulnerabilities of ChatGPT generated code and StackOverflow answers https://arxiv.org/abs/2403.15600