Università della Svizzera Italiana

Luca Beltrami

Context

Artificial Intelligence is rapidly transforming the way software is written. LLM like ChatGPT and Copilot are now commonly used to assist developers by generating code, automating repetitive tasks, and offering suggestions for modifying or understanding existing code.

This has made software development faster and more accessible. Even individuals with limited programming experience can now generate functional code in seconds.

- AI-generated code is based on pattern prediction, not true understanding
- The output often looks syntactically correct, but that can be misleading
- It may contain hidden bugs, inefficiencies, or security vulnerabilities
- These issues are especially common in tasks that require deep semantic reasoning

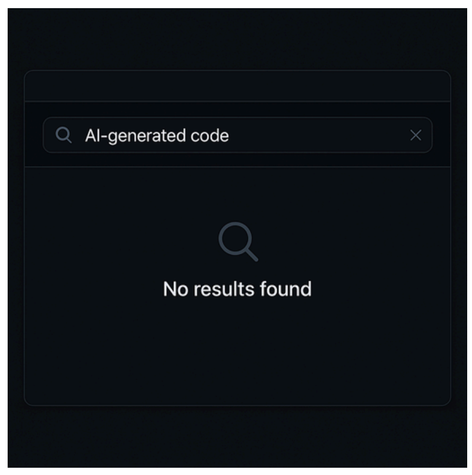
While AI performs well on simple, isolated tasks, its performance degrades significantly in larger or more complex projects. Tasks involving abstract thinking, consistent structure, or multi-file integration often reveal the limitations of current models.
To ensure the safe adoption of AI in software development, we must critically analyze the code these systems produce. We need to understand not just what AI can generate, but how reliable and secure that code is in practice.

Problems and Limitations

During the project, the biggest issue was to find a reliable method to search and analyse code. Then some limitations occured during the data collection and analysis process.

- No clear usual method for analysis
- The code produced by AI tools often closely resembled human-written code in structure and style, making reliable classification non-trivial
- Public code repositories rarely label code as “AI-generated”, hard to find confirmed examples to validate detection methods
- Limited dataset of code

I needed to test different methods to find the best one to extract and analyse code AI-generated code, as precisely and efficiently possible.



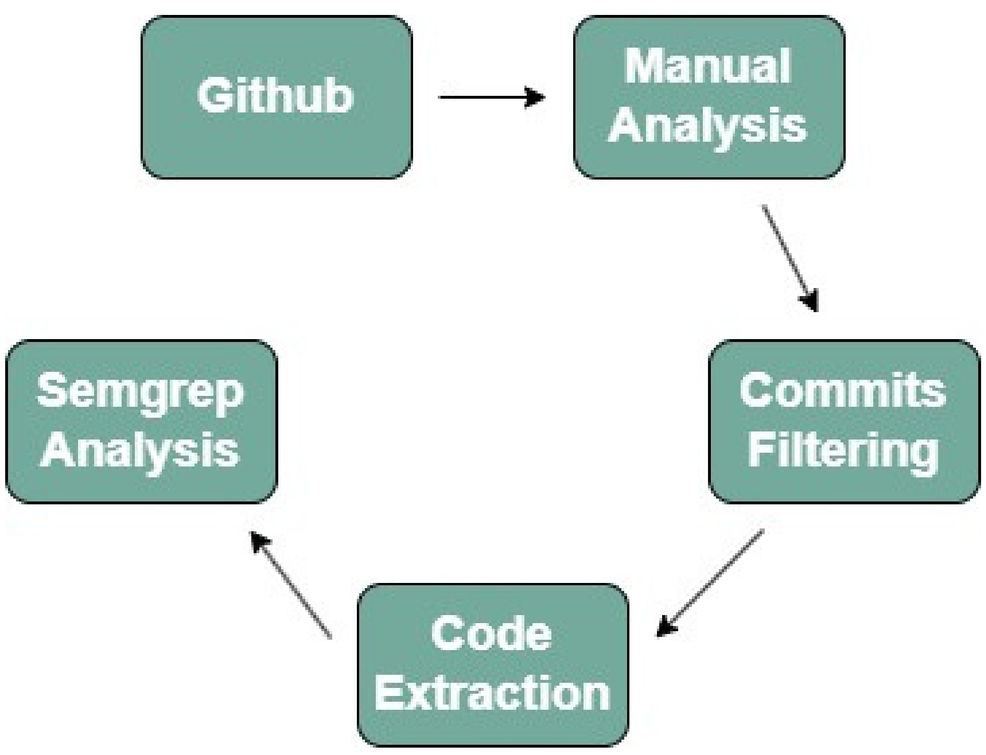
The analysis tools are very limited and potentially costly (e.g. OpenAI code detector’s API). Moreover they have limited accuracy and limited flexibility for the analysed code.

- Static analysis had limited accuracy
- Most static analysis tools expect complete programs, many AI-generated code snippets are partial or lacked context
- OpenAI’s code detector was unreliable (false positives)

Study Design

To evaluate how AI-generated code behaves in practice, I designed a data-driven analysis pipeline. After realizing that distinguishing between AI and human code with the OpenAI code detector was unreliable, I shifted the focus to analyzing only code that was very likely generated by AI.

- Search for GitHub commits mentioning AI terms
- Extract relevant n-grams from commit messages for filtering
- Use these n-grams to automatically filter AI-related commits
- For each commit, extract the longest chunk of code added
- Run Semgrep on the diff to analyze code vulnerabilities



- 750 GitHub commits mined
- 22 n-grams
- Metadata collected for each commit
  - Commit link
  - Matched n-gram
  - Number of files changed
  - Longest added code block
- Applied on 40 commits (AI-tagged and pre-2020 human commits) — but results were inconsistent

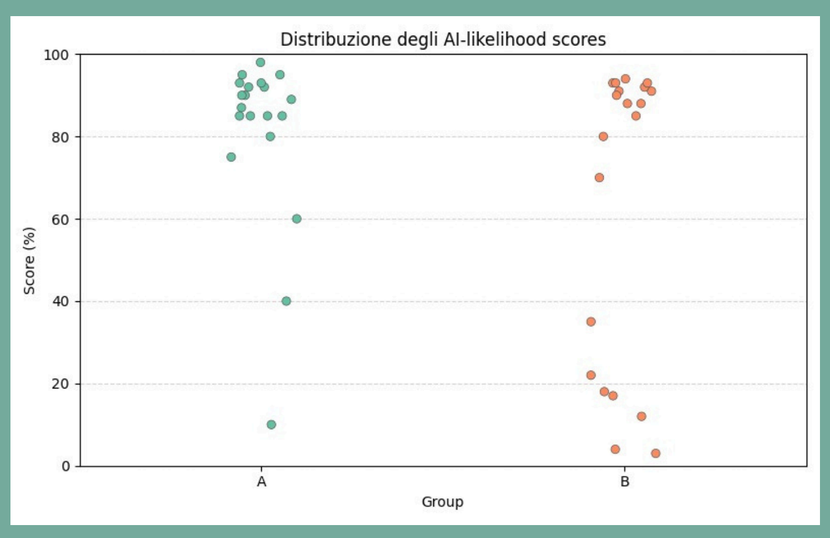
So I decided to change my analysis tool and use Semgrep, a static analysis tool that detects bugs, insecure code patterns, and violations of best practices.

- Retrieved the full source files and scanned them using Semgrep
- Languages analyzed: Python, JavaScript, Java, C++, Go, PHP, C, C#, TypeScript

Results

To sum up, from the code I extracted 2 types of analysis: with OpenAI code detector (promising but not reliable to distinguish between AI and Human code), and with Semgrep (which points out the vulnerabilities of the code)

As shown in the graph below, altough the OpenAI code detector did point to some differences in the results between the two sets of code, the rate of false positive, meaning makes it a unreliable tool for this task.



Most affected languages:

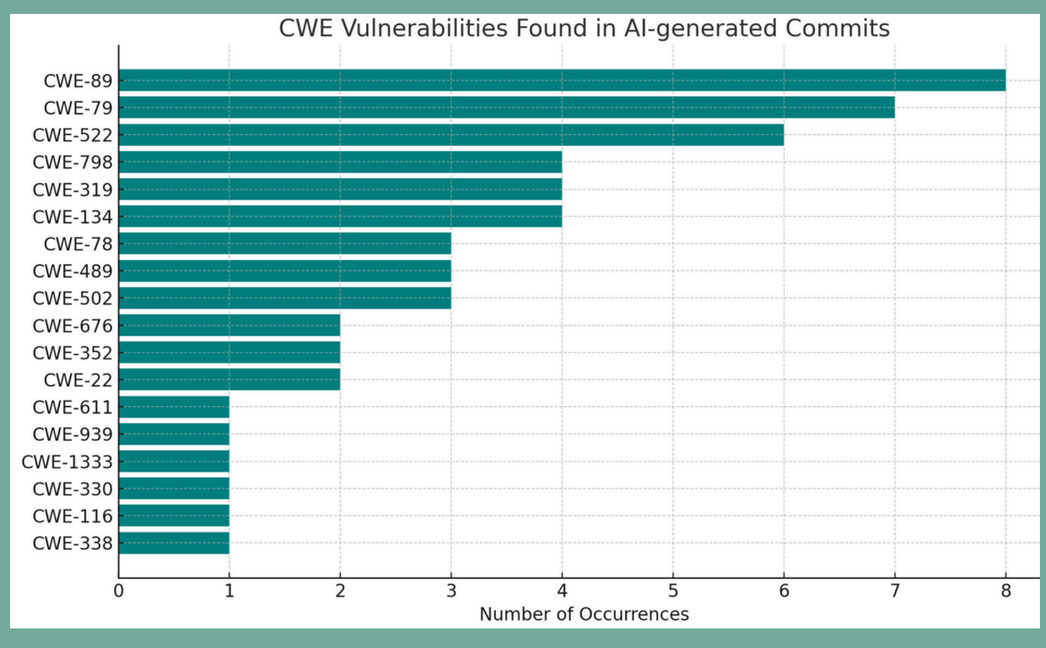
- Python (11 findings), especially Flask/WTF forms
- JavaScript (11 findings), especially Express and Node.js

Statistic	Value
Minimum	1.0
Maximum	8.0
Median	1.0
Average	1.72

Severity	Count
WARNING	29
ERROR	16
CRITICAL	2
INFO	8

As shown in the graph below, the most common vulnerability was the SQL injection vulnerability(CWE-89) that can cause:

- Unauthorized data access
- Data modification or deletion
- Full database compromise



Università della Svizzera italiana