How to Compare the Security of Code Written by Humans to LLM-generated Code

Jasmine Egli[1], Rebecca Balebako[1,*]

[1]Zurich University of Applied Sciences

Abstract

Large language models (LLMs) are rapidly transforming how software is created and maintained. Comparing LLM-generated code against human-written standards is essential to determine whether these new tools uphold or erode the security baselines established by professional developers. Yet, we lack a standardized method for empirically comparing the security of code produced through human-LLM collaboration against LLM-only, or traditional human-only methods. To facilitate this, we propose an automated framework for conducting comparative studies across human-only, LLM-only, and hybrid conditions. Our approach automates the logging of prompts, timing, and experimental settings, measuring outcomes through multi-dimensional static and dynamic quality analysis. We provide an open-source implementation of this framework to ensure that future researchers can conduct reproducible, species-fair experiments. Importantly, we validate the framework via a feasibility study, providing an experimental blueprint for "species-fair" comparisons between human and AI subjects. By sharing lessons learned, we establish a foundation for empirical research on human and LLM-generated code for software security.

Keywords

Security, Secure Code, LLMs, Quality of LLM-generated Software Artifacts

1. Introduction

Large language models (LLMs) have fundamentally altered software development. While LLM tools offer significant efficiency gains , security remains a primary, unresolved concern [1]. Current literature offers disparate conclusions on the safety of LLM-generated code, often because outcomes are artifacts of inconsistent evaluation methods and contexts.

There is relatively sparse literature on how humans and LLMs compare in producing secure code. To limit confounding factors, comparing human and LLM-written code requires a "species-fair" approach [2]. Empirical research that attempts to compare the security of LLM versus human code should follow best practices in Developer Centered Security research and best practices in LLM evaluation, while striking the balance in what is a fair comparison.

Furthermore, evaluations of human and LLM-written code are often confounded by non-deterministic outputs, varying task constraints, and a reliance on subjective or high-friction manual reviews. Without a reproducible and multi-dimensional measurement pipeline, it is difficult to determine whether observed security flaws are merely artifacts of inconsistent experimental design. To address these inconsistencies, future empirical work must employ precise, species-fair designs that account for these influences. There is an urgent need for an automated, deterministic framework that can bridge this gap and enable researchers to isolate the variables that influence code security across human, LLM, and hybrid generation paradigms

Our developed framework proposes a path to bridging the gap. It offers researchers a method to compare the functionality and security of human-written code and LLM-generated code. This framework can help designers, security researchers, and tool developers identify conditions and variables that influence code security across different code generation paradigms. By providing humans and LLMs with identical exercises and executing the results in identical environments, and then utilizing code

Joint Proceedings of the STAF 2026 Workshop, LLM4SE Rennes, France, June 29- July 3, 2026

*Corresponding author.

0009-0002-0160-5709 (J. Egli); 0000-0001-9862-0790 (R. Balebako)

quality metrics to understand code security, the framework allows for a direct assessment of how human logic compares to the capabilities of modern models.

We evaluate the framework on curated human code and LLM-generated code to demonstrate the feasibility of this approach. We provide lessons learned for experimental design, task creation, and security metrics to ensure reproducibility. Our contribution lies in the identification of experiment design issues, providing a roadmap for future researchers to align LLM secure programming capabilities with commensurate human skill levels in a way that is scientifically defensible.

1. Instructional Symmetry: Identifying the prompt engineering necessary to maintain a fair comparison between human and LLM instructions.
2. Model Reliability: Documenting how specific model behaviors can break automated evaluation pipelines.
3. Analysis of Experimental Attrition: Categorizing the ”leaky pipeline” where samples are lost to malformed outputs or execution failures, providing a blueprint for the sample over-provisioning required for statistical validity.

The next sections are as follows. In the Related Work section, we give an overview of relevant work in evaluating secure code. In the Framework Implementation section, we describe the design decisions made in designing the framework tool and outcomes. In the Feasibility Study we describe the feasibility design and lessons learned. Finally, in the section Discussion, we cover limitations, future work, and ethics.

2. Related Work

2.1. Measuring the security of code

Despite its complexity, quantifying security is essential for comparative code evaluation. The focus of this work is on source code, as opposed to broader systems of security, as this is an area LLMs are currently helpful.

2.1.1. Run-time Analysis for Errors and Correctness

Dai et al. [3] examine LLM-generated code, and show the need to measure both functionality and security of code. Correctness include error rates and functionality. Code should run without failure and it should complete the required task.

Frequent runtime errors can indicate fragile code. High error rates during execution suggest a lack of robust input validation, which is the primary entry point for injection attacks. [4] demonstrated that traditional software fault metrics are strongly correlated with security vulnerabilities, suggesting that “buggy” code is statistically more likely to be “vulnerable” code.

2.1.2. Static Analysis for Code Quality

Static analysis automates code evaluation against a predefined rule set, enabling scalable, fast, repeatable quality checks. However, static analysis cannot include contextual nuances, runtime behavior, and emergent properties. Despite these limits, static analysis integrates into automated pipelines and has relevance to code security [5, 6, 7]. Key metrics include *Cyclomatic Complexity:* (McCabe), which measures the number of linearly independent paths through a program’s source code and acts as predictor of security vulnerabilities [4]. *Mean function length:* or lines of code per function is another indirect measure of security. Long functions tend to “leak” state, where variables intended for one task are accidentally reused for another, leading to memory corruption or sensitive data exposure [8, 9, 10].

2.2. Developer Centered Security

Code security is shaped by environment in which the code is produced. Developer-Centered Security (DCS) is a subset of the usable security community. This field include over a decade of work to understand what conditions influence secure code, and offer insights into the contexts in which developers produce secure (or insecure) code [11] [12].

Current empirical evaluations of LLM-assisted coding in DCS have yielded fragmented results, with findings on security and productivity varying significantly across different study designs. Sandoval et al. [13] used a randomized control trial to understand how novice programmers were impacted by LLMs. Overall, they found LLMs assistants had small "but consistent" benefits to functionality of the code without adversely impacting the security of the code. Perry et al. [14] discovered that participants with access to an LLM assistant wrote less secure code than those without access to an assistant. Belozerov et al. [15] explored the capabilities of ChatGPT in detecting security issues but concluded that LLM-generated code tends to have more vulnerabilities than human-written code.

2.3. Comparing Human to LLM code

Direct comparisons of code by humans and by LLMs are often confounded by asymmetrical testing conditions, necessitating a more balanced approach to evaluating human and LLM-generated software. Firestone [2] advocates for "species-fair" evaluations, emphasizing the necessity of equivalent constraints when benchmarking human and machine performance. Drawing parallels from computer vision, the author argues that direct assessments are only valid when both agents operate under symmetrical environmental and task-based parameters.

Previous research on comparing humans and LLMs have not relied on species-fair comparisons. Molison et al. [16] integrated Python solutions from open source repositories to test various LLM configurations to measure maintainability, reliability, and remediation effort. However, the LLM prompts included zero-shot, few-shot, and fine-tuning. Arguably this would be species-fair if humans also had multiple attempts to write the code. Cotroneo et al. [17] analyzed over 500,000 code samples in Python and Java to compare human developers to LLMs, also using static analysis for code security and measuring code complexity. The LLMs were prompted to generate code by giving them the docstring and function signature that was extracted from the human code. This arguably means the LLMs started with a different task than the humans. Licorish et al. [18] used the same prompt for humans and LLMs, therefore being more "species-fair." With a dataset of 72 software engineering tasks, they used static analysis tools to measure coding standards, complexity, and security vulnerabilities. However, only one human was responsible for the full human-generated code dataset, which introduces concerns about representativeness of human coders. The findings across these papers are contradictory in regards to whether LLMs or humans are "better" or "more secure", and some were task dependent.

3. Framework Implementation

3.1. Creating and Collecting Code

Comparing human-created code to LLM-generated code requires acquiring code from humans and prompting LLMs to generate code. Researchers may ask participants and LLMs to write code for different tasks. We refer to these human-language instructions as "exercises." For example, an simple exercise might be to "Write python to generate the first 10 values in Fibonacci sequence of numbers." For LLMs, the exercise is often called a "prompt." Our framework is designed to track and compare results for multiple exercise definitions, and assume the same exercise is given to both humans and LLMs. After a human or LLM receives the exercise, they should generate executable code.

3.2. Framework Design

The architecture overview of the implementation is displayed in Figure1. In general, this framework resembles many of the related work examples, with additional emphasis in reproducibility and species-fair design. The framework serves a dual purpose: first, as a standardized interface for code generation across various LLMs, and second, as a deterministic execution and evaluation pipeline for security measurement. Key properties of a deterministic system include: (1) clear definitions of input and output variables, (2) precisely known initial conditions, (3) cause-effect relationships, (4) predictability, (5) continuity, (6) reproducibility, (7) modeling, and (8) time independence. This architecture ensures that our measurements of code quality remain reproducible to the extent possible.

The framework is implemented in Python and available on Github [19]. All artifacts during the execution of the pipeline are persisted in a centralized SQLite database.

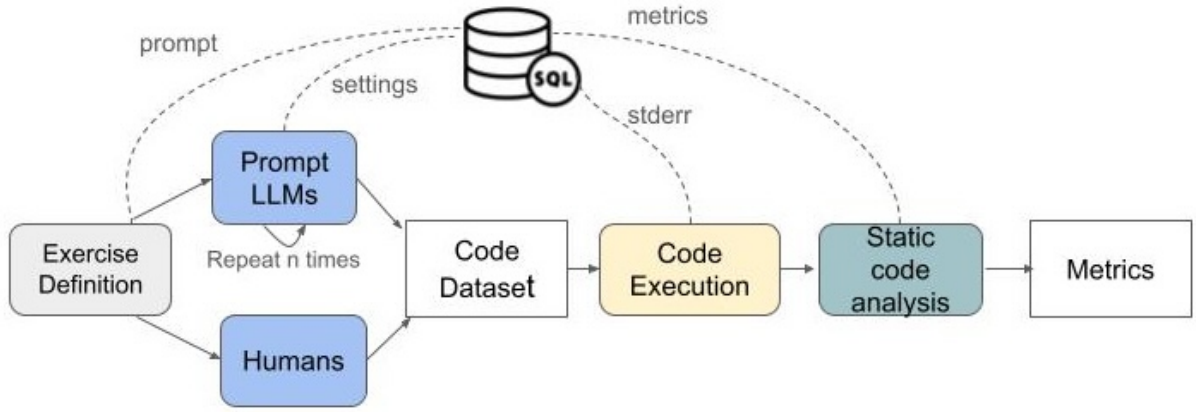

Figure 1: Framework design: Exercise definition as prompt for LLMs and humans, the output is executed in a container, and static analysis is run. All settings and environment variables are saved the database.

Exercise definition: The programming exercises (tasks or prompts describing what code to write) are stored as version text-files in a git repository and then imported into the prompt table in the database. This allows clear exercise inputs over runs that are identical and documented.

Code generation: Automated model invocation with strictly defined input-output parameters ensures a traceable audit trail for all generated code artifacts. For LLMs, we used OpenAI API with fixed request parameters (client, model, temperature, top p, max tokens, frequency penalty and presence penalty) and defined response parameters (timestamp, output text, response id, request id, finish reason, latency ms, output tokens, total tokens). The parameters were fixed across models and defined in **??**. Generated code is captured in a defined output format to minimize variability.

This step is repeated for a fixed number of times, creating a number of code outputs for each exercise and model, before moving to the next step in the pipeline.

Code execution: To ensure consistent initial conditions, both human and LLM-generated code are executed in isolated Podman containers with a fixed set of local dependencies. We further mitigate environmental non-determinism by restricting tasks to self-contained logic that does not rely on external network services.

Code evaluation: As described in Table 1 , we used a collection of metrics to measure correctness, clarity, and security of the code output. A rule-based assessment was performed with static Ruff linter [20].

3.2.1. Reproducibility and Auditing

The pipeline captures metadata to create traceable artifacts linking specific prompt inputs to quality outputs. Metatdata includes: model parameters, timestamps, execution logs, and security results. This

Table 1
Quality Metrics and Criteria

Quality Criteria	Metric	Description
Correctness	Pass (1) / fail (0)	Output of the code execution is evaluated automatically to determine if the output is correct or not.
Clarity	Cyclomatic complexity	Measures independent paths through the code sample. High complexity indicates hard-to-understand logic.
	Function length	Counts the number of lines in a function. Large functions are harder to understand.

structured environment ensures the reproducibility of deterministic properties, such as initial conditions and cause-effect relationships.

However, while the framework facilitates characterization through repeated trials, absolute predictability remains hindered by the intrinsic non-determinism of LLM internals [21] and external service variability. To preserve the integrity of these trials, execution and evaluation results remain inaccessible to the models, preventing the LLMs from learning across multiple iterations and ensuring that each generation is independent.

3.2.2. Containers for Security and Reproducibility

We use Podman (an open-source, daemonless container engine) for reproducibility [22], but they serve a secondary purpose in protecting the researchers' environment. Evaluating code security requires executing potentially malicious or malformed logic. We avoid these pitfalls by routing all generated code through a rootless Podman containerization layer.

3.3. Framework Results

The framework implements four primary controls to isolate variables identified in Developer-Centered Security (DCS) literature while adhering to the species-fair doctrine [2].

3.3.1. Species Fair Integrity

Environmental Symmetry: To ensure fair constraints, we utilize Podman-based containerization. This guarantees that both human and LLM-generated code are evaluated within identical operating systems, library sets, and hardware allocations.

Instruction Parity: Functional requirements and security constraints must be mirrored exactly between human instructions and LLM system prompts. This prevents prompt-tuning bias, where one agent receives more architectural guidance than the other.

Contextual Calibration: DCS research highlights participant expertise as a critical determinant of security. The framework allows researchers to map model capabilities (e.g., base vs. reasoning models) to commensurate human experience levels (e.g., students vs. seniors).

Functional Quality Gates: To avoid the common pitfall of evaluating the security of non-functional code, the pipeline enforces sequential evaluation. Code must pass functional unit tests before undergoing security analysis, ensuring results are not confounded by syntax or logical failures.

Isolation and Traceability: Each iteration generates a unique audit trail linking prompts to outcomes. The environment resets for every trial to prevent data leakage or cross-iteration learning, ensuring independent results.

3.3.2. Research Utility and Framework Benefits

The framework provides a flexible, extensible foundation for comparative studies across diverse code-generation paradigms.

Multi-dimensional Assessment: The pipeline automates the evaluation of execution, code clarity, and security. We leverage the Ruff linter (800+ rules), allowing researchers to toggle security patterns relevant to their specific exercises (e.g., omitting SQL injection checks for mathematical tasks).

Reproducibility : By minimizing measurement variance in inherently non-deterministic LLM environments, the framework provides a stable baseline.

Operational Efficiency: Designed for accessibility, the toolchain utilizes ubiquitous technologies (Python, SQLite, Podman). During our feasibility study, OpenAI API costs were minimal (approx. $10), and the containerized architecture protected the host machine from potentially malicious code execution.

Privacy: To protect human participants , the framework is a "closed loop" It does not feed code back into LLMs for training or prompts, preventing the proliferation of private information (such as humans including their identifiers in the code).

4. Feasibility Study

To demonstrate the framework's application in assessing both LLM-generated and human-authored code, we executed a feasibility study focused on Python security and algorithmic challenges.

4.1. Exercise selection

Thirteen coding exercises were chosen as prompts. These prompts cover Python security exercises and Advent of Code (AoC) 2024 challenges.[23]. These tasks span difficulty levels from beginner to expert and cover diverse problem types (algorithmic puzzles, parsing, data-structure task) as well as security-focused cases. These challenges require both syntactic correctness and algorithmic reasoning, expose security and robustness issues, and allow deterministic validation of correctness via predefined input/expected output pairs. Where available, property-based unit tests were used to verify and stabilize basic building blocks of the code base [24].

4.1.1. Human Code Data sources

A set of human reference solutions was included in execution and quality assessment steps (see Figure 1). For w3resource exercises, only one online solution was available. For Advent of Code 2024, hundreds of solutions exist on Github. Approximately five high-ranking Python solutions were selected from the public leader board and extracted from their repositories.

4.1.2. LLM Code Generation

Exercises were copied from the challenge websites and used as prompts. For this process input data are embedded in the prompt text file, as the LLMs cannot load external directories.

Five LLM model variants were compared: gpt-4.1, gpt-4o-mini, gpt-5.1, gpt-5-mini, and gpt-5-nano. According to the OpenAI documentation, these models differ in coding reasoning, determinism, latency, cost per run, error profile or failure modes. [25]

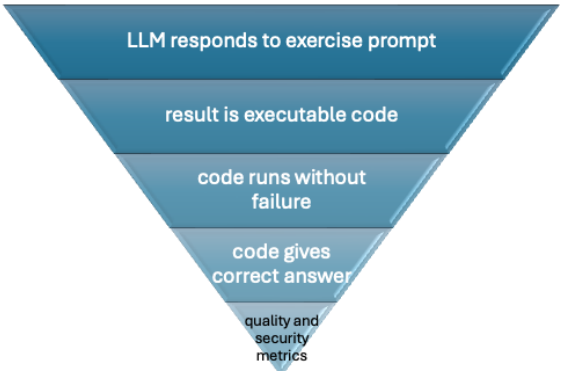

Figure 2: Cascading errors: Results at each step may be disqualified from inclusion in the next analysis.

Each exercise and model combination can executed in the automated pipeline, as described in 1, multiple times. Prompt ingestion, parametrization (temperature, top p, max tokens, frequency penalty and presence penalty), and input token number were stable across runs. All requests to generate code completed without error.

4.1.3. Handling Errors

In this analysis, code is analyzed in multiple steps, as shown in Figure 2. For example, the LLMs did not always output executable code; sometimes they just returned the answer or human text intermingled with code snippets. We did not do further execution or testing on non-code results. In these cases, the sample was removed from further testing. Some of the executable code failed to run, or ran and produced incorrect answers. If it failed, the analysis stopped for that sample. The assessment for cyclomatic complexity, function length and security metrics was done for correct code results only.

4.2. Feasibility Study Results

We utilize a pilot sample to examine the nuances of species-fair task constraints and automated security evaluation. Rather than providing definitive security conclusions, these preliminary results stress-test the evaluation pipeline, ensuring the framework can reliably support the complex, species-fair research it was built to facilitate.

4.2.1. Data Analysis

Each exercise was generated 60 times per model to minimize within-task variance and approximate independent draws [26]. 60 repetitions were intended strike a balance between capturing variability in model output and practical resource and time constraints. However, as described previously, we were not able to run all samples. In our case, the total test should have 3,900 data points (5 models * 13 exercises * 60 times), but we were only able to execute 3,832 samples (98%) of LLM-generated code, and 2,615 produced correct output (67%). Furthermore, some models never produced correct code for some exercises. This leads to a skewed dataset which violates many common distribution assumptions. Assumptions for linear regression, ANOVA, and poison regression did not hold due to the dispersion and distribution of these errors.

The selection of top-rated human code samples for this feasibility study introduced an inherent ceiling effect, as all human-authored entries executed without functional errors. These samples are not intended to represent a statistically random distribution of developer performance. Rather, the impact of this selection process reinforces our primary thesis: that future comparative research requires a standardized framework built from the ground up to account for and mitigate such experimental biases.

4.2.2. Correctness

For each exercise, the "correct" answer was a pre-defined string. We verified whether the executed code generated this correct answer. Correctness varied greatly across the exercises. Some exercises were solved consistently across all models, others showed more model-dependent performance. Correctness was near-100% for gpt-4.1, strong but task-sensitive for gpt-4o-mini, and more variable for the gpt-5 family (with gpt-5.1 out-performing gpt-5-mini/nano).

To account for the bounded nature and unequal variance of the correctness scores (0 to 60), we employed a Binomial Generalized Linear Model (GLM) using a logit link function to model the correctness. The relative influence of prompt id and model was then quantified by comparing the increase in model deviance upon the removal of each variable, identifying the factor that contributed most significantly to the model's overall fit. We find the exercise itself was more predictive of correct output than the different models.

Predictor	Δ Deviance	df	p-value	Influence Rank
Prompt ID	1693.59	12	< 0.001	1st
Model	397.81	4	< 0.001	2nd

Table 2
Analysis of Deviance for Correctness: The higher variance for the prompt id indicates that the exercise had more influence on correctness than the model.

4.2.3. Failure Mode

Analysis of failure modes revealed two dominant error types: input/output and structural errors. (1) "No output" error type meaning that the executed code did not print a correct standard output. Analysis of the code showed that this error is caused by its improper input handling. The generated solutions assumed external files (e.g., password.txt), expected manual input, omitted input assignment, or printed natural-language answers instead of executable code. (2) Algorithmic and structure errors: The LLM generated functioning code but the output was wrong ("WrongLogic"), or the script was aborted due to "IndexError", "RecursionError" or"NameErrors".

4.2.4. Clarity

Code clarity is characterized by cyclomatic complexity and function length. Comparative analysis of correctly solved runs highlighted distinct stylistic differences between models and human authors. See Table 3. Human-authored references generally exhibited lower McCabe cyclomatic complexity than the models, despite the functions being longer. The GPT-4 series produced less complex solutions than the GPT-5 series on harder tasks. Similar to McCabe cyclomatic complexity, we assessed the quality metric "function length". Typical mean values varied from 4 up to 20 lines, with CLT standard errors ranging from ±0.3 to ±2.0 depending on sample size (n).

model	gpt-4.1	gpt-4o-mini	gpt-5-mini	gpt-5-nano	gpt-5.1	human
McCabe	4.6 ± 0.1	3.3 ± 0.0	5.2 ± 0.1	4.2 ± 0.1	4.5 ± 0.1	2.9 ± 0.2
Function Length	9.2 ± 0.3	9.3 ± 0.1	7.4 ± 0.7	13.5 ± 0.4	6.4 ± 0.1	10.7 ± 0.2

Table 3
Complexity Scores: Mean value of cyclomatic complexity (McCabe) with CLT standard error for correctly solved challenges only across different models and human generated references. Mean value of function length with CLT standard error for correctly solved challenges across different models and human generated references.

4.2.5. Security

The total number of security issues reported was small, limiting the statistical conclusion. However, the data suggests that security error types were clustered by exercise rather than the model family, and similar errors occurred within an exercise across models. While the counts were small for the human-authored code, it appears that errors that the models commonly make in an exercise were also seen in the human-generated code.

Figure 3 summarizes security rule violations across models and human-generated code. No issues were flagged for prompt 2 or 8.

5. Discussion

There is still work to do to understand where humans versus LLMs should be writing code for security. This work does not claim to offer a final answer that one or the other is "better." Instead, we offer

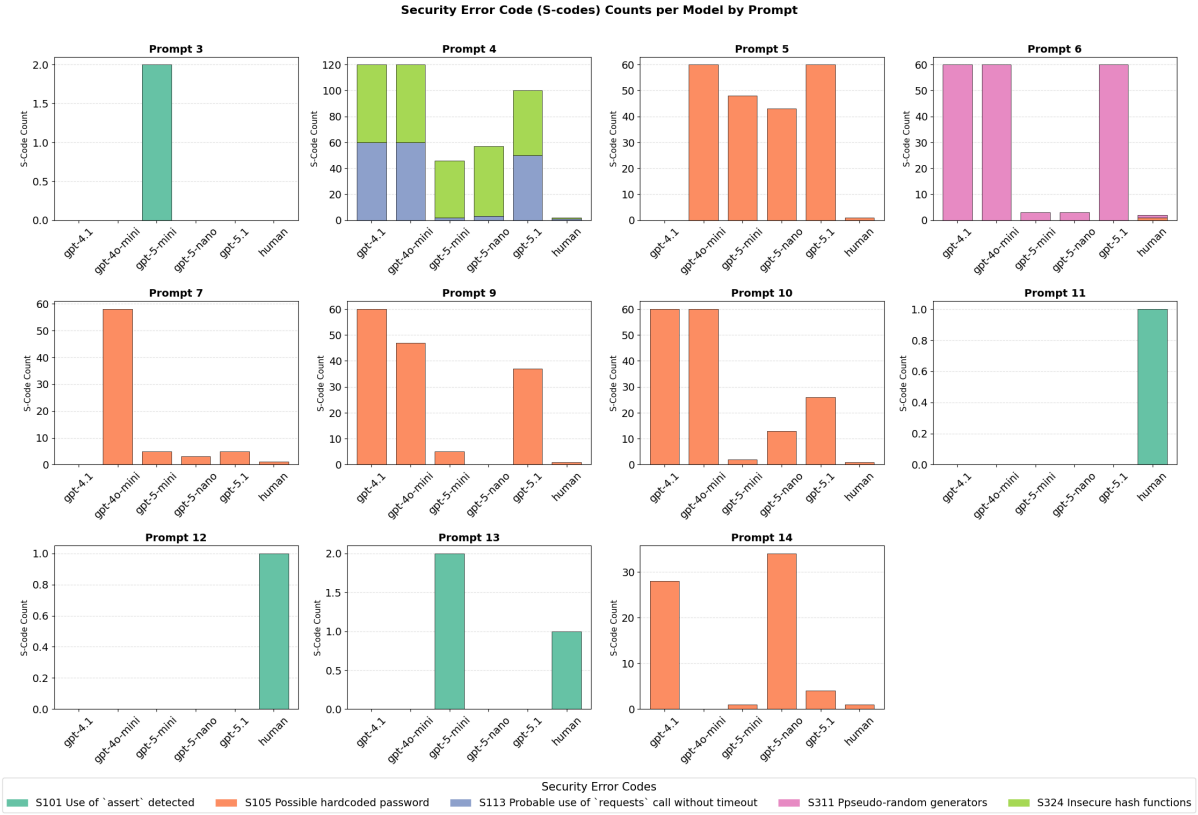

Figure 3: Security Errors: The types of security errors flagged by Ruff per model and prompt.

a framework so that more work can be done. The framework is designed to eliminate confounding factors, ensuring that the results reflect true security differences rather than artifacts of a mismatched experimental design.

This framework prioritizes a species-fair way to compare code written by humans versus LLM by giving LLMs and humans the same prompts and evaluating them on the same metrics. Such a prompting style may seem suboptimal for either the humans or the LLMs, but allows for comparable results. In this manner, we can compare apples to apples to better understand strengths and weaknesses of secure code output for similar tasks.

The feasibility study offers lessons learned for future experiments. The feasibility study can be used to help researchers estimate statistical power or anticipate experimental concerns before comput-time is expended or human subjects are recruited.

5.1. Limitations

The following limitations in the framework were observed:

- Task coverage and metric sensitivity: some quality metrics, notably security, did not reach statistical significance for the selected problems. The framework identified that current sample sizes in the literature are likely underpowered.
- While the current reference implementation is Python-specific, the containerized architecture and evaluation logic are language-agnostic.
- This work was carried out in late 2025. The rapid increase in model capability may influence future work.

5.2. Future Work

Future research should transition toward task-specific, focused experiments designed to identify the exact points in a workflow where LLMs provide the most value and where human intervention remains essential.

Exercise Selection: Our results suggest that exercise selection influences outcomes more than model choice, necessitating a focused understanding of how specific tasks elicit vulnerabilities.

Prompt Engineering: Prompt engineering is crucial in providing a fair comparison. For example, given the exercises instructions and context, humans may understand to write code, but LLMs may need this explicitly stated. However, researchers must maintain a balance between prompt optimization for LLMs and human readability.

Model Selection: The quality of models overall set a baseline performance. However, increased model reasoning capacity did not inherently translate to higher-quality code. There are opportunities for future work here.

Metric Selection: The framework supports a modular selection of metrics tailored to specific research questions. We report results separately to ensure they remain adaptable to diverse research goals. Developing a single score would require a subjective weighting system that may not align with every study.

5.3. Ethical Considerations

Common ethical concerns of this study were discussed by the researchers and considered to be acceptable. No vulnerabilities in production code or in real systems were discovered in this work. There are no direct military applications. No human subjects were used in this work. The data collection adhered to the terms of service of the respective platforms and follows established norms for the use of publicly available datasets in software engineering research [27].

Declaration on Generative AI

During the preparation of this work, the author(s) used Gemini in order to: Paraphrase and reword, improve writing style, draft code, and simulate peer review.

References

[1] L. C. Ramírez, X. Limón, A. J. Sánchez-García, J. C. Pérez-Arriaga, State of the Art of the Security of Code Generated by LLMs: A Systematic Literature Review, in: 2024 12th International Conference in Software Engineering Research and Innovation (CONISOFT 2024), 2024, pp. 331–339. doi:10.1109/CONISOFT63288.2024.00050.

[2] C. Firestone, Performance vs. competence in human–machine comparisons, Proceedings of the National Academy of Sciences 117 (2020) 26562–26571. doi:10.1073/pnas.1905334117.

[3] S.-C. Dai, J. Xu, G. Tao, Rethinking the Evaluation of Secure Code Generation, 2025. doi:10.48550/arXiv.2503.15554, arXiv:2503.15554 [cs].

[4] Y. Shin, L. Williams, An empirical model to predict security vulnerabilities using code complexity metrics, in: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, ESEM '08, Association for Computing Machinery, New York, NY, USA, 2008, pp. 315–317. URL: https://dl.acm.org/doi/10.1145/1414004.1414065. doi:10.1145/1414004.1414065.

[5] M. Pistoia, S. Chandra, S. J. Fink, E. Yahav, A survey of static analysis methods for identifying security vulnerabilities in software systems, IBM systems journal 46 (2007) 265–288.

[6] H. Choi, D. Kang, J.-Y. Choi, Static Analysis for Software Reliability and Security, in: K. Daimi, H. R. Arabnia, L. Deligiannidis, M.-S. Hwang, F. G. Tinetti (Eds.), Advances in Security, Networks, and Internet of Things, Springer International Publishing, Cham, 2021, pp. 463–470.

[7] A. Gosain, G. Sharma, Static analysis: A survey of techniques and tools, in: Intelligent Computing and Applications: Proceedings of the International Conference on ICA, 22-24 December 2014, Springer, 2015, pp. 581–591.
[8] Y. Shin, L. Williams, An empirical model to predict security vulnerabilities using code complexity metrics, in: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, 2008, pp. 315–317.
[9] S. Moshtari, A. Sami, M. Azimi, Using complexity metrics to improve software security, Computer Fraud & Security 2013 (2013) 8–17.
[10] M. J. Tehrani, S. Hashemi, Assessing vulnerability in smart contracts: The role of code complexity metrics in security analysis, arXiv preprint arXiv:2411.17343 (2024).
[11] M. Tahaei, K. Vaniea, A Survey on Developer-Centred Security, in: 2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), 2019, pp. 129–138. URL: https://ieeexplore.ieee.org/document/8802434. doi:10.1109/EuroSPW.2019.00021.
[12] Y. Acar, S. Fahl, M. L. Mazurek, You are Not Your Developer, Either: A Research Agenda for Usable Security and Privacy Research Beyond End Users, in: 2016 IEEE Cybersecurity Development (SecDev), IEEE, Boston, MA, USA, 2016, pp. 3–8. URL: http://ieeexplore.ieee.org/document/7839782/. doi:10.1109/SecDev.2016.013.
[13] G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, B. Dolan-Gavitt, Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants, 2023, pp. 2205–2222. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/sandoval.
[14] N. Perry, M. Srivastava, D. Kumar, D. Boneh, Do users write more insecure code with ai assistants?, in: Proceedings of the 2023 ACM SIGSAC conference on computer and communications security, 2023, pp. 2785–2799.
[15] V. Belozerov, P. J. Barclay, A. Sami, Secure Coding with AI, From Creation to Inspection, arXiv preprint arXiv:2504.20814 (2025).
[16] A. S. Molison, M. Moraes, G. Melo, F. Santos, W. K. G. Assuncao, Is LLM-Generated Code More Maintainable & Reliable than Human-Written Code?, 2025. doi:10.48550/arXiv.2508.00700, arXiv:2508.00700 [cs].
[17] D. Cotroneo, C. Improta, P. Liguori, Human-Written vs. AI-Generated Code: A Large-Scale Study of Defects, Vulnerabilities, and Complexity, in: 2025 IEEE 36th International Symposium on Software Reliability Engineering (ISSRE), 2025, pp. 252–263. doi:10.1109/ISSRE66568.2025.00035, iSSN: 2332-6549.
[18] S. A. Licorish, A. Bajpai, C. Arora, F. Wang, K. Tantithamthavorn, Comparing Human and LLM Generated Code: The Jury is Still Out!, 2025. doi:10.48550/arXiv.2501.16857, arXiv:2501.16857 [cs].
[19] Egli, Jasmine, LLMEvaluationTool/LLMgenerateCodeEvaluationTool at main · jaegli/LLMEvaluationTool, 2026. URL: https://github.com/jaegli/LLMEvaluationTool/.
[20] S. Gonzalez, Ruff: a game changer or Python linters., 2024. URL: https://xantygc.medium.com/ruff-a-game-changer-for-python-linters-12b1ec8c5f12.
[21] S. Ouyang, J. M. Zhang, M. Harman, M. Wang, An empirical study of the non-determinism of chatgpt in code generation, ACM Transactions on Software Engineering and Methodology 34 (2025) 1–28.
[22] D. Walsh, Podman in Action: Secure, rootless containers for Kubernetes, microservices, and more, Simon and Schuster, 2023.
[23] AoC, Advent of Code 2024, 2025. URL: https://adventofcode.com/2024/.
[24] J. Börstler, K. E. Bennin, S. Hooshangi, J. Jeuring, H. Keuning, C. Kleiner, B. MacKellar, R. Duran, H. Störrle, D. Toll, Developers talking about code quality, Empirical Software Engineering 28 (2023) 128.
[25] OpenAI, Models, 2025. URL: https://platform.openai.com/docs/models.
[26] M. A. Alvarado Gonzalez, M. B. Hernandez, M. A. Peñaloza Perez, B. Lopez Orozco, J. Tadeo Cruz Soto, S. Malagon, Do Repetitions Matter? Strengthening Reliability in LLM Evaluations, arXiv e-prints (2025) arXiv: 2509.24086.

[27] T. Kohno, Y. Acar, W. Loh, Ethical Frameworks and Computer Security Trolley Problems: Foundations for Conversations, in: Usenix Security, 2023, pp. 5145–5162. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/kohno.