Confidence, Correctness, and Security in AI-Generated Code: A Correlation Study using SWE-bench

Alvin Lee
MITB, SMU
Singapore
alvin.lee.2024@
mitb.smu.edu.sg

Chen Zhiyang
MITB, SMU
Singapore
zy.chen.2024@
mitb.smu.edu.sg

Chew Tze Hoong
MITB, SMU
Singapore
th.chew.2024@
mitb.smu.edu.sg

Sim Kim Sia
MITB, SMU
Singapore
kimsia.sim.2024@
mitb.smu.edu.sg

Abstract

Large Language Models (LLMs) are increasingly used for code generation and automated bug fixing, yet their outputs can be incorrect, miscalibrated, or insecure. We propose a systematic correlation study of three dimensions in LLM-generated patches: correctness, confidence, and security. Using a Python-only subset of SWE-bench (≈150–200 issues) and three open-source code LLMs (gpt-oss-20b, qwen3-coder-30b-a3b-instruct, devstral-small-2507), we (i) measure patch correctness against ground truth/ tests, (ii) estimate confidence via self-consistency and logit-based proxies (with CROW - Confidence-Ranked Output Weakening - as a stretch), and (iii) assess security with Semgrep, Bandit, and CodeQL, aggregating findings into a Security Risk Score (SRS). We analyze how confidence correlates with correctness and security, and whether confidence-based filtering reduces both failed and vulnerable patches. As a stretch goal, we propose a composite "trustworthiness" score that integrates these dimensions to guide evaluation and deployment. Our findings aim to inform safer IDE assistants and future benchmarks by moving beyond accuracy toward multi-dimensional reliability.

1 Introduction

The integration of large language models (LLMs) into developer workflows, through tools such as GitHub Copilot and Code Llama, has begun to reshape modern software engineering. These systems can accelerate coding tasks and improve developer productivity, yet their adoption raises fundamental questions of trustworthiness. Generated patches are not always reliable: they may be incorrect, accompanied by misleading confidence estimates, or even introduce subtle security vulnerabilities.

Recent research highlights these risks across three dimensions. First, in terms of correctness, code produced by LLMs often fails to resolve the intended issue, even when it passes superficial checks (Chen et al., 2021; Austin et al., 2021). Second, regarding confidence and calibration, LLMs are frequently miscalibrated, expressing high confidence in incorrect outputs or low confidence in correct ones (Guo et al., 2017; Spiess et al., 2025). Third, with respect to security, studies have shown that AI-generated patches can introduce vulnerabilities, including SQL injections, insecure subprocess calls, and weak cryptographic practices, that may be more severe than those in human-written fixes (Pearce et al., 2023; Kang et al., 2024).

These risks are especially concerning in real-world development workflows. Even "passing" code may be unsafe: a patch that resolves a failing test could simultaneously open a command injection pathway or leak sensitive data. Conversely, potentially correct and secure patches may be discarded if confidence thresholds are set too strictly. Understanding how confidence signals align with correctness and security outcomes is therefore central to enabling safer deployment of code LLMs.

Prior work has largely considered these axes in isolation: benchmarks measure correctness, calibration studies focus on confidence, and security audits evaluate vulnerabilities. What is missing is a joint analysis across all three dimensions on a realistic benchmark such as SWE-bench. This project addresses that gap by asking:

Do higher-confidence patches tend to be both more correct and more secure? If not, how should confidence be interpreted in practice?

As an illustrative example, consider an LLM-generated patch that replaces input validation with an eval() statement. While such a patch might resolve the immediate bug and even pass associated test cases, it introduces a severe injection vulnerability. A multi-metric evaluation framework could flag this discrepancy (high correctness but low security), guiding developers toward safer decisions such as withholding or revising the patch.

In summary, despite rapid progress in LLM code generation, no systematic study has analyzed the interaction between correctness, confidence, and security. This proposal aims to fill that gap by conducting a correlation study on SWE-bench and exploring the design of a composite benchmark that reflects the multi-dimensional trustworthiness of LLM-generated code.

2 Objectives

The primary goal of this project is to evaluate the interplay between correctness, confidence, and security in LLM-generated code patches. First, we aim to assess correctness, by measuring whether model-generated patches resolve the target issue as defined in SWE-bench, either through repository test suites or ground-truth diff comparison. Second, we will examine multiple forms of confidence estimation, including self-consistency across multiple generations, logit-based token probabilities, and as a stretch, CROW (Confidence-Ranked Output Weakening). This allows us to quantify how well different proxies align with correctness. Third, we will investigate security, running generated patches through Bandit, Semgrep, and CodeQL to identify potential vulnerabilities, and aggregating results into a normalized Security Risk Score (SRS). Fourth, we will conduct a correlation study to measure the relationships among these three dimensions, asking whether higher confidence is predictive of both correctness and security. Finally, as a stretch goal, we intend to design a composite benchmark metric, combining correctness, confidence, and security into a single trustworthiness score that could serve as a future evaluation standard.

3 Related Works and Literature Review

Correctness Benchmarks. Early benchmarks such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) introduced unit-testable tasks for evaluating functional correctness of code generated by LLMs. These datasets demonstrated that models could generalize to short, synthetic programming problems, but they remain limited in complexity and context. More recently, SWE-bench (Kang et al., 2023) extended evaluation to real-world GitHub issues by pairing natural language issue descriptions with developer-written patches and full repository test suites. This shift enables a more ecologically valid measure of correctness, though it also increases variance and difficulty, making it an ideal setting for studying trustworthiness of model outputs.

Confidence and Calibration. Calibration has been studied extensively in classification tasks, with Guo et al. (2017) introducing the Expected Calibration Error (ECE) as a standard metric. While these ideas transfer naturally to LLMs, recent work shows that code generation models are often poorly calibrated, giving high confidence to incorrect patches (Spiess et al., 2025). Beyond log-probabilities, Wang et al. (2023) proposed self-consistency for reasoning tasks, where agreement among multiple generations provides a more robust confidence signal. Extensions such as CROW (Ulmer et al., 2024) aim to improve output ranking through adversarial weakening. These methods suggest promising directions for estimating when a generated patch is trustworthy, but they have not yet been systematically connected to security properties.

Security of LLM code. Several studies have shown that LLMs produce code with non-trivial vulnerabilities. Pearce et al. (2023) highlighted insecure patterns such as SQL injection and improper subprocess usage in Python code generated by Codex. Kang et al. (2024) further demonstrated that on SWE-bench, AI-generated patches can introduce new vulnerabilities not present in the original developer fixes, underscoring the potential risks of deploying such code directly. More broadly, Khoury et al. (2023) examined security vulnerabilities in GitHub Copilot suggestions, finding that up to 40% of generated code snippets contained exploitable weaknesses. Sandoval et al. (2023) provided a taxonomy of LLM-induced vulnerabilities, categorizing issues such as insecure cryptography, input sanitization failures, and privilege escalation risks. Nguyen et al. (2024) investigated the reliability of static analysis tools like Semgrep and CodeQL when applied to LLM outputs, concluding that tool coverage is complementary but still incomplete. Collectively, these works highlight the pressing need for frameworks that combine correctness testing, calibration analysis, and automated vulnerability scanning to evaluate code LLMs in a holistic manner.

Gaps. Prior work has primarily studied these axes in isolation or in pairs, for example, correctness with calibration, or correctness with security. However, no existing evaluation framework has systematically integrated all three dimensions - correctness, confidence, and security - into a single analysis. This project addresses this gap by examining their correlations on SWE-bench and exploring the design of a composite benchmark that reflects the multi-dimensional trustworthiness of code LLMs.

4 Methodology

Dataset & Sampling. We will use a subset of SWE-bench, restricted to Python projects to ensure compatibility with static analysis tools. From the full dataset, we will sample approximately 150–200 issues stratified by project and issue type. Issues without reproducible test environments will be excluded. This choice balances coverage with feasibility, given the constraints of model inference and static analysis at scale.

Models and Generation Protocol. We will evaluate three open-source code LLMs: gpt-oss-20b, qwen3-coder-30b-a3b-instruct, and devstral-small-2507. For each SWE-bench issue, we will prompt models with the issue title, description, and surrounding context, and generate ten candidate patches using diverse decoding parameters (temperatures 0.6, 0.8, and 1.0 with fixed top-p). Generations will be cached with seeds and, where available, token log-probabilities to support reproducibility and confidence estimation.

Correctness Evaluation. Generated patches will first be applied to the target repository. If unit tests are available, we will execute them in a sandboxed environment with timeouts to label candidates as pass or fail. In cases without runnable tests, we will fall back to diff-based similarity against ground-truth patches and manually audit a small subset. For each issue, we will compute pass@N metrics and categorize errors into syntax, runtime, or logical failures.

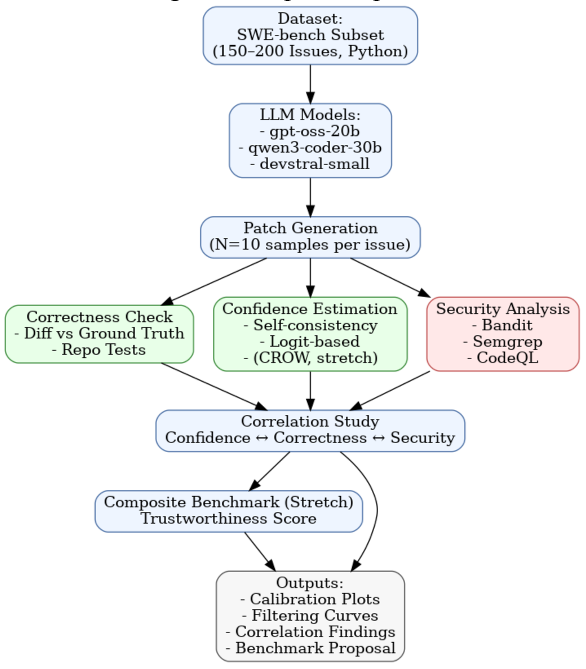
Confidence Estimation. We will experiment with multiple confidence proxies. The primary approach is self-consistency, where confidence is quantified as the fraction of passing candidates among the ten generations. As a test-free proxy, we will also compute mean pairwise similarity among candidates (e.g., edit distance or AST-based comparison). Additionally, when log-probabilities are accessible, we will calculate average token log-probs as a logit-based confidence measure. Finally, as a stretch goal, we will attempt to implement a simplified version of CROW to evaluate robustness of confidence rankings under controlled perturbations.

Security Analysis. All generated patches will be scanned using Semgrep, Bandit, and CodeQL. We will aggregate results into a Security Risk Score (SRS), weighting high-severity findings more heavily than medium or low ones, and normalizing by lines of code. To reduce false negatives, we will also report results under max-rule (flagging a patch if any tool identifies a high-severity issue).

Analysis. We will compute correlations between the three dimensions: confidence versus correctness, confidence versus security, and correctness versus security. Statistical measures will include Pearson and Spearman coefficients, AUROC and AUPRC for discriminative power, and Expected Calibration Error (ECE) with reliability diagrams for calibration quality. We will also generate filtering curves by sweeping confidence thresholds to evaluate how many correct and secure patches would be shown or withheld.

Stretch Goal: Composite Benchmark. If time permits, we will propose a composite "trustworthiness score" that integrates correctness, confidence, and security into a single evaluation metric. The score will take the form of a weighted combination of normalized values, and we will explore the sensitivity of different weightings. This will provide a proof-of-concept for a future SWE-bench-style benchmark that evaluates not only accuracy but also calibration and safety.

Figure 1: Proposed Pipeline



5 Plan of Execution

The project will be completed within an eight-week timeline by a team of four members. In Week 1, we will finalize our SWE-bench subset, focusing on Python projects, and setup the required infrastructure, including model inference environments and static analysis tools. In Week 2 and 3, we will run the three selected LLMs on approximately 150–200 SWE-bench issues, generating multiple candidate patches per issue, and perform initial correctness validation via ground-truth comparisons and repository tests. In Week 3 and 4, we will implement and compute the confidence metrics, producing calibration curves and evaluating the predictive power of self-consistency and logit-based measures. In Week 5 and 6, we will conduct the security analysis by running Semgrep, Bandit, and CodeQL across all generated patches, aggregate findings into Security Risk Scores, and complete the correlation study between correctness, confidence, and security. In Week 7 and 8, we will synthesize results into plots and tables, write the project report, and prepare our presentation. If time permits, this final week will also include the development of a composite trustworthiness score and the release of a prototype harness that integrates correctness, confidence, and security evaluation into a single pipeline.

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*. https://arxiv.org/abs/2107.03374.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, et al. 2021. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732*. https://arxiv.org/abs/2108.07732.

Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. 2017. On Calibration of Modern Neural Networks. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*.

Claudio Spiess, Sam Witteveen, and Christoph Treude. 2025. Calibration and Correctness of Language Models for Code. In *Proceedings of the 47th International Conference on Software Engineering (ICSE)*

Yiming Kang, William Held, Andrew Zhao, Gabriel Synnaeve, and Jason Weston. 2023. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *arXiv preprint arXiv:2310.06770*. https://arxiv.org/abs/2310.06770.

Shengjie Li, Yuntao Bai, Kexin Wang, Tianle Wang, and Yuandong Tian. 2024. HonestCoder: Showing LLM-Generated Code Selectively Based on Confidence. *arXiv preprint arXiv:2410.03234*. https://arxiv.org/abs/2410.03234.

Michael Pearce, Kevin Markham, Dawn Song, and others. 2023. Examining the Security Implications of Large Language Models. In *Proceedings of the 32nd USENIX Security Symposium*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *Proceedings of the 11th International Conference on Learning Representations (ICLR)*.

Fabian Ulmer, Steffen Eger, and Iryna Gurevych. 2024. APRICOT: Calibrating Black-Box Language Models with Output-Only Confidence Estimation. *arXiv preprint arXiv:2403.05973*. https://arxiv.org/abs/2403.05973.

Khoury, R., Parnin, C., & Aldrich, J. 2023. How Secure is Code Generated by ChatGPT? An Empirical Study on GitHub Copilot and ChatGPT. *arXiv preprint arXiv:2304.09655*.

Sandoval, J., Roziere, B., & Lample, G. 2023. A Taxonomy of Security Vulnerabilities in Large Language Model-Generated Code. *arXiv preprint arXiv:2312.02985*.

Nguyen, A., Chen, J., & Bui, T. 2024. Evaluating Static Analysis Tools for Detecting Vulnerabilities in LLM-Generated Code. arXiv preprint arXiv:2405.01792.