Assessing the Security of Vibe Coding: Baseline vs. Security-Oriented Prompts in LLM Code Generation

Runtong He[1], Huishan Lai[1], Jingxue Chen[2], and Chunhua Su[1]

[1] University of Aizu, Aizuwakamatsu, Japan
{m5281012, m5281022, chsu}@u-aizu.ac.jp
[2] University of Electronic Science and Technology of China, Chengdu, China
1015725207@qq.com

Abstract. Large Language Models (LLMs) are increasingly used in software development through so-called "vibe coding," where developers specify tasks in natural language and rely on the model to produce executable code. While this paradigm lowers barriers to entry and accelerates prototyping, it raises concerns about security. Prior studies show that a substantial fraction of AI-generated code contains exploitable vulnerabilities, and functional correctness does not guarantee safety. This paper investigates whether security-oriented prompting improves the security of LLM-generated code and whether these effects generalize across models and sampling variability. We design ten representative Python tasks inspired by OWASP Top 10 and CWE categories, and evaluate outputs from GPT-OSS 20B with ten random seeds and from a cross-model comparison between GPT-OSS 20B and Gemma-3 27B. Outputs are analyzed using static analysis (Bandit) and lightweight runtime probes. Results show that default prompts consistently yield insecure code, with vulnerabilities present in most tasks. Security-informed prompts reduce both the prevalence and severity of weaknesses, particularly in command execution, deserialization, and secrets management. However, residual risks persist, such as insecure file upload handling and incomplete authentication checks, and improvements are not uniform across models. These findings suggest that prompt engineering can mitigate but not eliminate vulnerabilities.

Keywords: Large Language Models · Vibe coding · Software security · Prompt engineering

1 Introduction

Large Language Models (LLMs) have transformed software development by enabling developers to express intent in natural language while delegating most of the implementation to AI systems. This paradigm—often referred to as "vibe coding"—has been promoted in both industry and academia as lowering the barrier to entry and reshaping how non-experts and professionals alike engage with programming tasks [8,19,29,1]. By shifting from syntax-level editing to intent mediation, developers can rapidly prototype applications, test ideas, and improve productivity.

Despite these benefits, concerns about the security and reliability of AI-generated code are well-documented. Early studies revealed that GitHub Copilot often suggests insecure code, with insecure patterns recurring across common programming tasks [23,25]. Subsequent large-scale empirical analyses demonstrated that up to one third of Copilot-generated code integrated into repositories contains security weaknesses, covering dozens of CWE classes [7]. These findings highlight that functional correctness does not imply security by default. Moreover, surveys and systematic reviews confirm recurring risks in AI-assisted coding workflows and warn that novice developers may unknowingly introduce vulnerabilities by over-trusting model outputs [26,6]. These weaknesses often map onto categories in the Common Weakness Enumeration (CWE)[15] and the OWASP Top 10[22], which are widely used to classify and prioritize software security risks.

Beyond code-level issues, generative AI introduces broader cybersecurity and privacy concerns. Recent work has documented adversarial uses of LLMs, such as jailbreaks, prompt injection, and malicious code generation [10,2]. In data sharing and privacy-sensitive domains, AI has been shown to undermine anonymization techniques or facilitate sensitive information leakage [12,16]. These risks extend the scope of "secure-by-design" considerations to include data protection, model governance, and compliance with ethical and regulatory frameworks.

To address these challenges, researchers have begun developing datasets, benchmarks, and frameworks for secure code generation. The SecurityEval dataset provides curated examples of vulnerabilities across multiple CWE categories, allowing systematic evaluation of whether AI models tend to produce insecure code [28]. CWEVAL goes further by introducing outcome-driven metrics that jointly assess functionality and security, exposing gaps where LLMs succeed functionally but fail to avoid vulnerabilities [24]. Industry efforts complement these academic initiatives, including repository-level "rules files" that encode organizational security policies directly into AI coding assistants [13,30].

In this paper, we aim to provide empirical evidence on the extent to which vibe coding can be made safer under realistic settings. Our study asks whether explicitly security-oriented prompts can reduce vulnerabilities in LLM-generated code, and whether such improvements remain consistent across different random seeds and model families. By building on prior evidence of insecure defaults and incorporating benchmarks inspired by CWE and OWASP categories, we aim to

provide a clearer picture of how vibe coding practices can be strengthened in realistic development settings.

Research Questions

This study is guided by the following research questions:

1. **RQ1:** To what extent does default vibe coding with LLMs produce insecure code across representative tasks?
2. **RQ2:** Can security-oriented prompting reduce vulnerabilities across different task categories?
3. **RQ3:** What limitations remain when relying on prompting alone for secure code generation, and how do results vary across model families?

Threat Model

Our threat model assumes a developer relying on an LLM assistant to generate source code. The developer may be security-unaware, providing only functional requirements, or may explicitly request security-oriented implementations. The adversary is not an external attacker, but rather the systemic risk that insecure defaults propagate into production code. We further assume that the LLM is a black-box model, with no access to its training data or internal weights. Thus, our analysis targets the *interface level*, where developer prompts interact with model outputs. While we do not model an active external attacker in this study, the same insecure defaults could be exploited by adversaries if deployed in production.

Contributions

This paper makes the following contributions. First, we design ten representative Python programming tasks grounded in OWASP Top 10 and CWE categories to systematically evaluate vibe coding security. Second, we empirically compare default prompts with security-oriented prompts, measuring vulnerability prevalence and severity across tasks with extended multi-seed experiments. Third, we include a cross-model comparison (GPT-OSS 20B vs. Gemma-3 27B) to examine whether improvements generalize across model families. Fourth, we show that while security-informed prompts reduce risks, residual vulnerabilities persist, underscoring the need for defense-in-depth. Finally, we release the task set, evaluation scripts, and source code in a public GitHub repository to support replication and foster further research.[3]

[3] https://github.com/huiishan99/vibe-sec-experiment

2 Related Work

This section reviews prior studies relevant to secure code generation with LLMs. We cover four areas: (i) the rise of AI-assisted and vibe coding workflows, (ii) evidence of security risks in model-generated code, (iii) benchmarks and evaluation methods, and (iv) mitigation strategies including governance and verification.

2.1 AI-assisted Coding and Vibe Coding

The notion of "vibe coding" has emerged as an interaction paradigm where developers primarily specify intent in natural language and delegate most of the implementation to LLMs. Recent reports and industry explainers describe this workflow as reducing entry barriers and accelerating prototyping, while also reshaping expectations of collaboration and software development practices [8]. Academic surveys highlight similar trends in scientific computing: LLMs can assist with code drafting, scaffolding, and reproducible workflows, though quality control and reproducibility remain open challenges [18,3].

2.2 Security Risks in LLM-generated Code

Empirical evidence consistently shows that LLM-generated code often contains exploitable weaknesses. In a large-scale study of GitHub projects, approximately 25–33% of Copilot-generated code merged into repositories was found to contain security weaknesses spanning 38 CWE classes [7]. Earlier analyses also reported that LLM suggestions frequently included insecure code across common tasks [23,25]. In educational contexts, systematic reviews show that students using AI assistants may overlook boundary conditions and security considerations, introducing vulnerabilities into their code [26,6]. At the ecosystem level, new threats such as data poisoning against training corpora and backdoor insertion in public code have been demonstrated [4,2]. Controlled studies further suggest that developers may write more insecure code when assisted by LLMs, especially under time pressure [25].

2.3 Benchmarks and Evaluation Methods for Secure Code Generation

Several benchmarks have been developed to assess whether code generation models achieve both functional correctness and security. The SecurityEval dataset provides 130 Python examples covering 75 CWE types, allowing for fine-grained evaluation of insecure code tendencies [28]. More recently, CWEVAL highlighted limitations in prior benchmarks and introduced outcome-driven evaluation metrics, such as func@k and func-sec@k, to jointly assess functionality and security across multiple programming languages [24]. Unlike SecurityEval, which focuses on curated CWE coverage, CWEVAL emphasizes outcome-driven evaluation combining functionality and security, thus highlighting cases where models succeed syntactically but remain vulnerable. Systematic reviews further consolidate

knowledge on recurring vulnerabilities and the limitations of current evaluation practices [17].

2.4 Risk Mitigation and Engineering Governance

Practical mitigation strategies span governance, automated analysis, and formal verification. At the governance level, industry has proposed repository-level "rules files" that encode secure prompting and organizational best practices into AI-assisted tools such as Copilot and Cursor [13,30]. Static analysis tools remain the most widely used mechanism to audit generated code; recent empirical studies of Copilot-produced code relied on such methods for CWE categorization [7]. For stronger assurance, formal verification has been applied to LLM-generated programs in restricted domains. For example, studies explored verifying LLM outputs with Ada/SPARK in safety-critical contexts [5], and others proposed using verifiable languages like Dafny as intermediate targets for code generation[11,14]. Although these methods apply to restricted domains, they illustrate pathways for integrating verifiability into vibe coding workflows in the long term. Finally, research emphasizes that secure LLM-based coding must also incorporate privacy safeguards, given evidence of adversarial uses of generative AI in cybersecurity and data sharing scenarios [10,12].

2.5 Summary

In summary, while AI-assisted coding improves accessibility and productivity, a persistent gap remains: functional correctness does not imply security. Existing mitigation strategies range from governance to formal verification, but they are often decoupled from the developer's interaction with the model. This motivates our focus on prompt-level interventions combined with lightweight automated evaluation as a pathway to strengthen the security of vibe coding in practice. Table 1 compares representative prior studies with our approach.

Table 1. Comparison of representative prior studies on secure code generation.

Study	**Domain**	**Eval**	**Strengths / Limitations**
Copilot studies [23,25,7]	GitHub projects, common coding tasks	Static analysis, CWE mapping, manual audits	Large-scale real-world evidence; limited to proprietary Copilot
SecurityEval [28]	130 Python tasks (75 CWE)	Static analysis on curated dataset	Fine-grained CWE coverage; limited language and task diversity
CWEVAL [24]	Multi-language, outcome-driven	func@k, func-sec@k metrics	Combines functionality and security; benchmark only, not prompt-focused
Formal verification [5,11,14]	Safety-critical code (Ada/SPARK, Dafny)	Formal methods, verification tools	Strong guarantees in narrow domains; limited generality
This work	10 Python tasks (OWASP Top 10, CWE)	Bandit + runtime probes (VP, IC, SWC, RPR)	Prompt-level interventions, multi-seed/multi-model robustness; residual risks remain

3 Methods

This section describes the methodology used to evaluate secure code generation with LLMs. We outline the study design, task construction, prompt conditions, models and sampling setup, evaluation tools, and analysis pipeline.

3.1 Study Design

Our study addresses the research questions stated in Section 1. We compare two prompt conditions: a default baseline prompt without security guidance and an improved prompt with explicit security instructions. The overall experiment pipeline is shown in Figure 1.

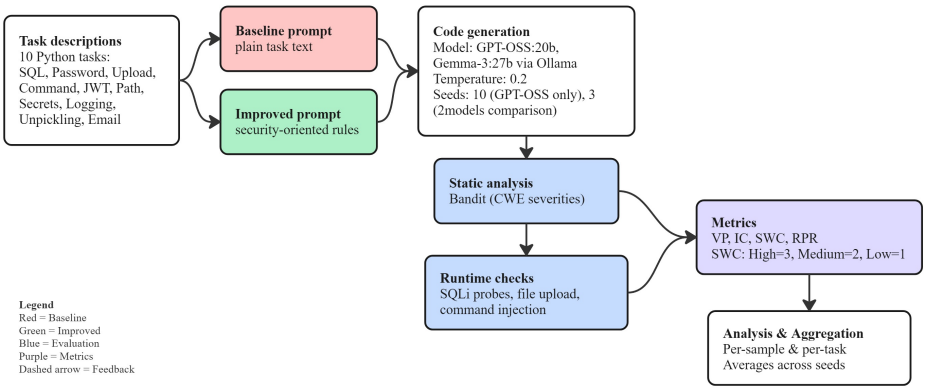

Fig. 1. Experiment pipeline. Ten tasks are prompted in two modes. Code is generated by `GPT-OSS 20B` and `Gemma-3 27B`, served via Ollama, and evaluated with Bandit and lightweight runtime probes. Metrics are VP, IC, SWC, and RPR.

3.2 Task Design

We designed ten Python programming tasks covering scenarios commonly associated with security risks, drawing inspiration from OWASP Top 10 and CWE categories [22,15]. For clarity, they are grouped into three categories:

1. **Data handling:** SQL query construction, password storage, and file upload handling.
2. **Execution and authentication:** Command execution, JWT authentication, and logging configuration.
3. **Resource and serialization:** Path handling, unsafe deserialization (via `pickle`), email validation, and secrets management.

These tasks test whether model-generated code follows secure practices for handling inputs, executing commands, managing credentials, and processing untrusted data. We selected these tasks as they represent recurring CWE categories frequently reported in prior studies of Copilot and AI-assisted coding [23,7].

3.3 Prompt Conditions

Each task was expressed in two prompt variants. The baseline prompt provided a plain natural language description without explicit security guidance, mirroring common "vibe coding" usage where the developer simply asks for functionality. The improved prompt augmented the same description with explicit security requirements, such as parameterized SQL queries, safe subprocess invocation without `shell=True`, extension whitelists for uploads, short-lived JWTs with audience checks, avoidance of unsafe deserialization, no hard-coded secrets, and minimal sensitive logging. All prompts are archived in the artifact for reproducibility. These improvements were manually designed based on widely recommended secure coding practices rather than task-specific heuristics, in order to avoid overfitting prompts to particular test cases.

3.4 Model and Generation Setup

We use two LLMs: `GPT-OSS 20B`, an open-source LLM for code generation [21], and `Gemma-3 27B`, a contemporary model of similar scale[9]. The models are served locally via Ollama [20], which simplifies deployment, ensures deterministic runs, and avoids legal or rate-limit issues associated with remote APIs.

Generation parameters are fixed across all experiments: temperature 0.2, top_p 0.9, and a context window of at least 8k tokens. Each run requests one runnable Python file per task, with model name, parameters, and random seeds logged.

3.5 Sampling and Runs

We conducted two sets of experiments. First, to improve robustness, we ran all ten tasks with ten random seeds each using GPT-OSS 20B, yielding ten baseline and ten improved samples per task. Second, to compare across models, we evaluated Gemma-3 27B and GPT-OSS 20B side by side with three random seeds each, producing three baseline and three improved samples per task per model.

For each seed, we retained the first syntactically valid sample in order to ensure determinism and avoid post-hoc selection bias. All prompts, generations, and analyzer outputs were stored in timestamped run directories.

3.6 Evaluation Tools and Checks

We applied Bandit, a static analysis tool for Python that reports findings with severity levels mapped to CWE-style categories [27]. We parsed Bandit's JSON outputs to compute per-sample and per-task aggregates.

For selected tasks, we also implemented lightweight runtime probes. For SQL queries, we tested injection payloads such as `' OR 1=1 -` to confirm that queries remained parameterized. For file uploads, we submitted files with double extensions and unusually long names to verify extension whitelists and size enforcement. For command execution, we injected separator-like inputs to confirm that the code invoked argument lists rather than expanding shell commands. These probes served as sanity checks rather than comprehensive penetration tests, and may therefore underestimate deeper logical flaws.

3.7 Metrics

We report four metrics. The vulnerability presence (VP) measures the percentage of generated samples per task containing at least one medium- or high-severity finding. The issue count (IC) records the total number of findings per sample. The severity weighted count (SWC) applies weights of High=3, Medium=2, and Low=1, following conventions used in prior static analysis practice [27]. Finally, the runtime pass rate (RPR) measures the fraction of samples passing the runtime probes, which apply only to tasks where probes were implemented.

3.8 Aggregation and Analysis

For each task and condition we average results across seeds and report VP, IC, and SWC in tables and figures. Figures include a bar chart of VP and a stacked bar chart of issue counts and severities. We report descriptive trends rather than statistical significance. The 10-seed runs enable variance reduction for GPT-OSS 20B, while the cross-model study highlights differences between model families. Section 4 presents results.

3.9 Reproducibility and Environment

Experiments were conducted on a Windows 64-bit workstation using Python 3.13 and a local Ollama installation. The pipeline is platform independent. Dependencies were managed via `requirements.txt`. Each run stored prompts, model parameters, Bandit JSON outputs, parsed CSV summaries, and human-readable logs under `eval/`. The artifact includes scripts to regenerate code, run Bandit, parse outputs, and render plots. The pipeline is deterministic given identical seeds and model builds.

3.10 Ethical and Safety Notes

Generated code was executed only in a controlled environment with sandboxed runtime probes. No generated code was deployed in production. We release prompts and analyzer outputs stripped of sensitive details for reproducibility.

4 Results

This section reports results from two sets of experiments. First, we extend the main study to ten random seeds with the GPT-OSS 20B model. Second, we conduct a cross-model comparison between GPT-OSS 20B and Gemma-3 27B under identical conditions. Together, these experiments provide evidence on the effect of security-oriented prompting and highlight differences across two models.

4.1 GPT-OSS 20B with 10 Seeds

Table 2 summarizes results across ten Python tasks with ten random seeds each, comparing baseline prompts with security-oriented prompts. Vulnerability presence (VP), issue count (IC), and severity-weighted count (SWC) are averaged across seeds.

Table 2. GPT-OSS 20B results with 10 seeds across ten tasks. VP = vulnerability presence (%), IC = issue count (mean), SWC = severity-weighted count (mean).

Task	VP (%)		IC (mean)		SWC (mean)	
	Baseline	Improved	Baseline	Improved	Baseline	Improved
SQL query	100.0	90.0	1.3	1.0	3.6	2.5
Password storage	0.0	0.0	0.0	0.0	0.0	0.0
File upload	100.0	100.0	1.8	1.1	3.8	3.1
Command execution	60.0	10.0	2.0	1.9	3.2	2.0
JWT authentication	100.0	100.0	2.0	1.0	4.0	2.0
Logging config	90.0	80.0	0.9	1.7	1.8	2.5
Path handling	0.0	0.0	0.0	0.0	0.0	0.0
Unpickling service	100.0	90.0	2.8	0.9	4.6	1.8
Email validation	40.0	100.0	0.7	1.0	1.4	2.0
Secrets management	0.0	0.0	0.0	0.0	0.0	0.0

As shown in Figure 2, security-oriented prompts reduced VP in several tasks, most notably command execution and unpickling. However, file upload and JWT authentication remained consistently insecure.

Figure 3 presents the average issue count. Security-oriented prompts lowered issue counts in high-risk tasks such as unpickling and JWT, though file upload continued to generate multiple findings.

Figure 4 shows the severity-weighted counts. Improved prompts substantially reduced severity in command execution and deserialization, yet high-severity issues persisted in file upload.

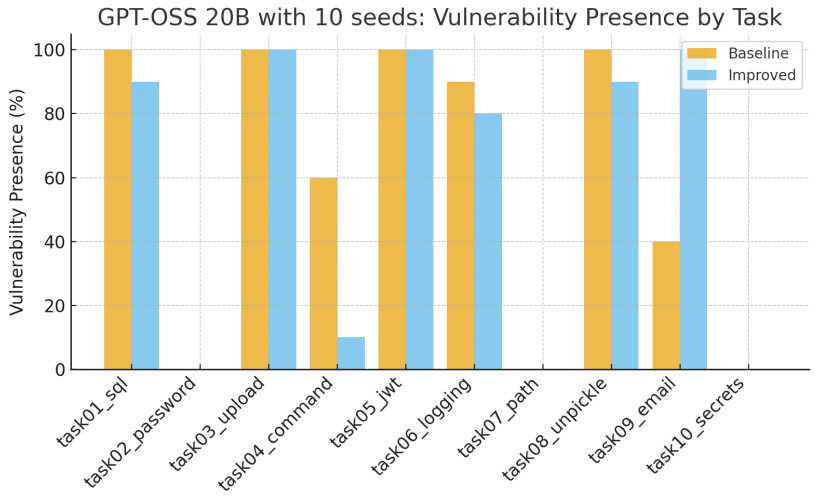

Fig. 2. Vulnerability presence (%) by task for GPT-OSS 20B with 10 seeds. Security-oriented prompts reduced risks in several tasks, but file upload and JWT authentication remained consistently insecure. Some tasks (logging, email validation) showed unstable outcomes, highlighting the limits of prompt engineering.

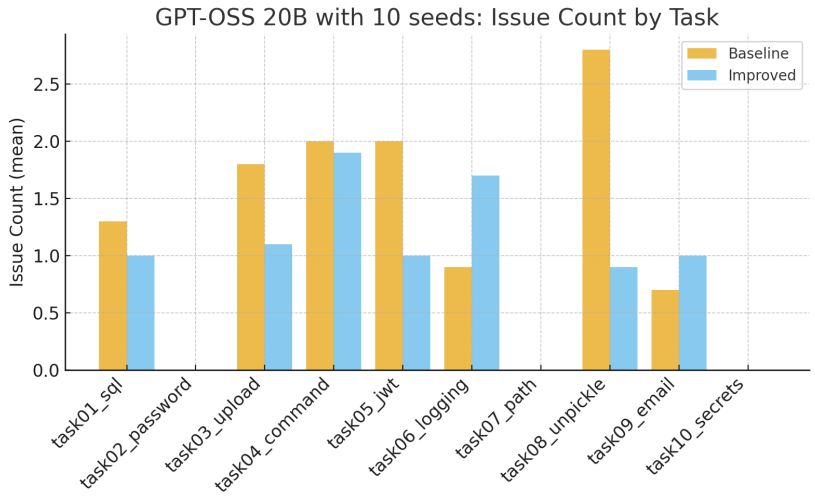

Fig. 3. Mean issue count per task for GPT-OSS 20B with 10 seeds. Security-oriented prompts lowered issue counts in unpickling and JWT, but residual vulnerabilities persisted in file upload.

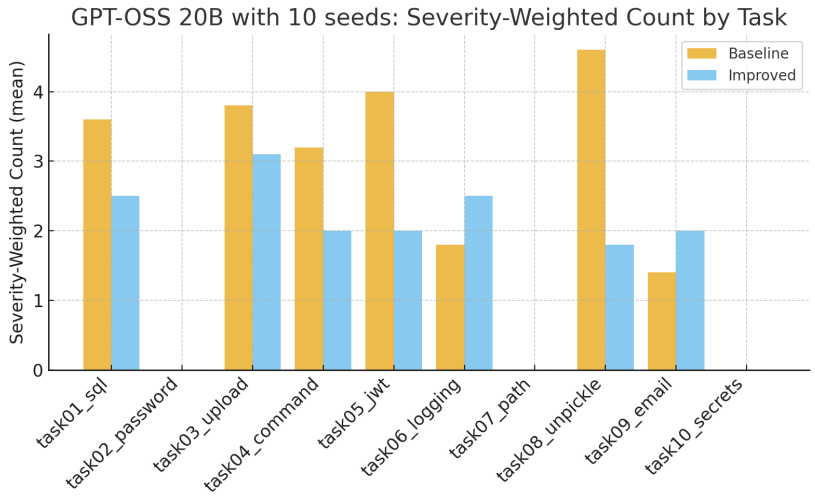

Fig. 4. Mean severity-weighted count per task for GPT-OSS 20B with 10 seeds. Improved prompts reduced severity in command execution and deserialization, though critical categories such as file upload remained problematic.

4.2 Cross-Model Comparison: GPT-OSS 20B vs. Gemma-3 27B

To assess whether findings generalize across models, we compared GPT-OSS 20B and Gemma-3 27B on the same ten tasks with three random seeds each. Results are summarized in Table 3.

As shown in Figure 5, GPT-OSS 20B produced fewer vulnerabilities than Gemma-3 27B in several categories, especially unpickling and secrets management.

Figure 6 further compares mean issue counts. GPT-OSS 20B generally showed fewer issues per task, while Gemma-3 27B tended to generate more frequent problems in logging and authentication.

Finally, Figure 7 shows the severity-weighted counts. GPT-OSS 20B outputs contained lower-severity issues overall, whereas Gemma-3 27B often produced more severe vulnerabilities, particularly in JWT authentication and unpickling.

4.3 Representative Vulnerability Fixes

Table 4 illustrates representative cases where improved prompts successfully mitigated vulnerabilities. As shown in the examples, command execution tasks replaced insecure `shell=True` calls with safer list-based invocations, and unpickling services substituted `pickle.loads()` with `json.loads()`. These examples demonstrate the potential of prompt engineering to steer model outputs toward safer implementations, though residual risks persisted in file upload handling and JWT validation.

Table 3. Cross-model comparison between Gemma-3 27B and GPT-OSS 20B (3 seeds). VP = vulnerability presence (%), IC = issue count (mean), SWC = severity-weighted count (mean).

Task	VP (%) Baseline		VP (%) Improved		IC (mean)		SWC (mean)	
	Gemma	GPT-OSS	Gemma	GPT-OSS	Gemma	GPT-OSS	Gemma	GPT-OSS
SQL query	100.0	100.0	100.0	100.0	1.0	1.7	3.0	4.3
Password storage	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
File upload	100.0	100.0	100.0	100.0	3.0	3.3	3.0	4.0
Command execution	100.0	100.0	100.0	66.7	3.3	2.0	5.3	3.3
JWT authentication	100.0	100.0	100.0	66.7	5.3	2.0	5.3	4.7
Logging config	100.0	33.3	100.0	66.7	3.0	0.7	3.0	0.7
Path handling	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Unpickling service	100.0	100.0	100.0	0.0	5.3	1.3	5.3	3.7
Email validation	100.0	100.0	100.0	66.7	1.7	0.7	3.3	1.3
Secrets management	100.0	100.0	100.0	33.3	2.0	0.7	4.7	1.3

Table 4. Representative examples of baseline vs. improved prompts. Security-oriented prompts replaced insecure API usage with safer alternatives.

Task	Baseline Example	Improved Example
SQL query	`query = "SELECT * FROM users WHERE id=" + user_id cursor.execute(query)`	`cursor.execute("SELECT * FROM users WHERE id=?", (user_id,))`
Command execution	`import subprocess subprocess.run("ls " + user_input, shell=True)`	`import subprocess subprocess.run(["ls", user_input], check=True)`
Unpickling service	`import pickle data = pickle.loads(request.data)`	`import json data = json.loads(request.data)`

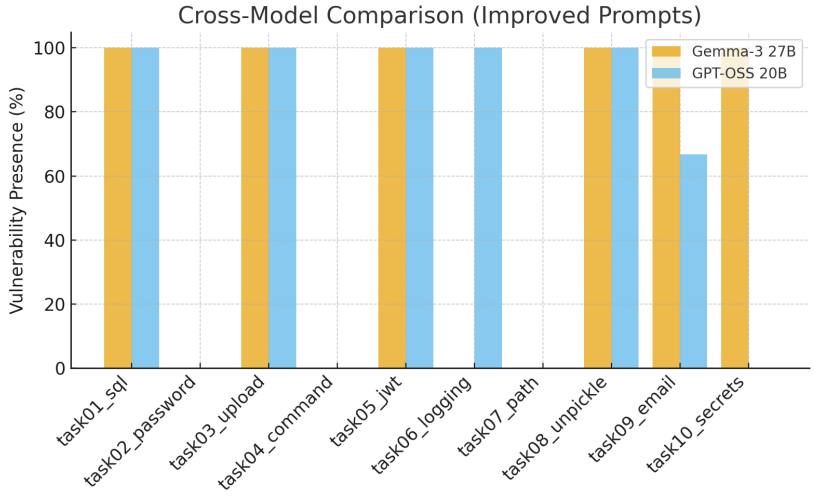

Fig. 5. Cross-model comparison of vulnerability presence across tasks under improved prompts. GPT-OSS 20B produced fewer vulnerabilities in unpickling and secrets management, while Gemma-3 27B remained insecure in logging and authentication.

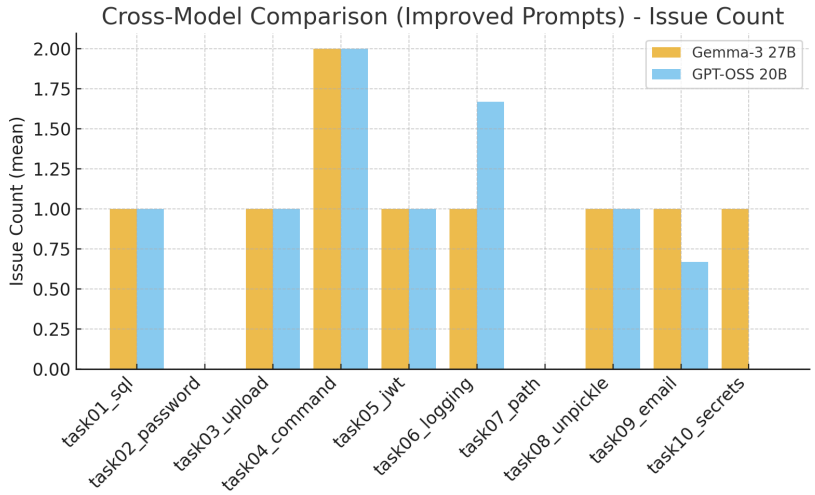

Fig. 6. Cross-model comparison of mean issue counts under improved prompts. GPT-OSS 20B showed consistently fewer issues, while Gemma-3 27B produced more problems in logging and authentication.

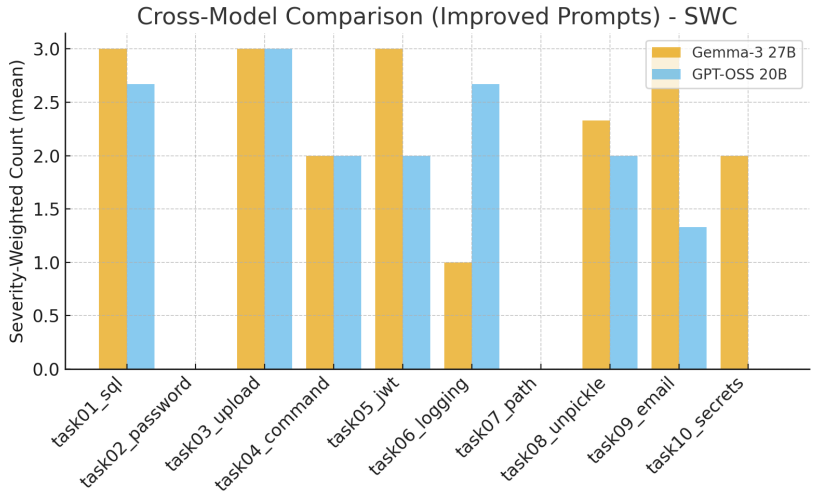

Fig. 7. Cross-model comparison of mean severity-weighted counts under improved prompts. GPT-OSS 20B outputs contained lower-severity vulnerabilities, while Gemma-3 27B exhibited more severe flaws in JWT authentication and unpickling.

5 Discussion

This section interprets the results in light of our research questions. We discuss implications for secure vibe coding, connections to prior work, potential threats to validity, and directions for future studies.

5.1 Answers to Research Questions

RQ1: Default vibe coding produced insecure code in most tasks, confirming prior findings that functional correctness does not imply security [23,7]. In our expanded 10-seed evaluation, up to 100% of generated samples contained vulnerabilities for several tasks (e.g., SQL query, file upload, JWT authentication), reinforcing the observation that insecure defaults are the norm rather than the exception.

RQ2: Security-oriented prompting reduced vulnerability presence (VP) and severity in many cases, with particularly strong improvements in command execution and unpickling. For example, VP dropped from 60% to 10% in command execution, and mean severity decreased substantially in unsafe deserialization. However, persistent risks remained in tasks such as file upload and JWT authentication, where security-oriented prompts could not eliminate vulnerabilities.

RQ3: Prompting alone was insufficient. The cross-model comparison showed that while GPT-OSS 20B often benefited more from improved prompts, Gemma-3 27B remained vulnerable in several categories (e.g., logging, JWT). These results highlight the need for layered defenses combining prompt engineering, static analysis, runtime checks, and verification [28,24].

5.2 Implications for Secure Vibe Coding

The findings emphasize that the default vibe-coding workflow, where developers adopt LLM outputs directly from functional prompts, is insecure across a broad range of tasks. Even when prompts explicitly requested secure practices, risks persisted.

Two key implications follow. First, prompting can steer model outputs toward safer APIs, but its effect varies across models and task categories. For example, GPT-OSS 20B showed notable improvements in command execution and secrets management, whereas Gemma-3 27B remained unstable in logging and authentication tasks. Second, secure adoption of vibe coding requires integration with automated defenses. Static analyzers such as Bandit, runtime probes, and possibly verification tools must be combined with prompting to provide defense-in-depth.

5.3 Relation to Prior Work

Our results are consistent with prior studies of GitHub Copilot, which demonstrated that a significant fraction of model-generated code contains vulnerabilities across common CWE categories [23,25,7]. We also align with systematic

reviews and surveys that highlight insecure defaults in AI-assisted programming and the risks of over-reliance by novice developers [26,6,18].

We extend this line of work in two ways. First, our 10-seed replication strengthens robustness claims by showing that trends hold under broader sampling, reducing the influence of stochastic variation. Second, we compare across two model families, showing that improvements from prompting are not uniform: while GPT-OSS 20B benefits more consistently, Gemma-3 27B remains vulnerable in categories where prompting alone is insufficient. This complements recent work on prompt engineering for secure code generation [24,3], clarifying both benefits and limits across models.

5.4 Threats to Validity

Several limitations should be noted.

Construct validity: We evaluated security using Bandit and lightweight runtime probes, which may miss logic-level flaws or over-report non-exploitable patterns [27]. More comprehensive evaluation combining dynamic analysis, fuzzing, or human audits could provide a fuller picture.

Internal validity: Although we expanded to 10 seeds for GPT-OSS 20B, only two LLMs were tested. Results may vary across larger model families, decoding strategies, or fine-tuned variants. Prompt sensitivity and sampling randomness remain open concerns [2].

External validity: The ten selected Python tasks are grounded in OWASP Top 10 [22] but cannot represent all real-world programming. Broader tasks, languages, and frameworks (e.g., cryptographic misuse, concurrency flaws) should be explored.

Conclusion validity: Our sample size remains moderate (10 seeds for GPT-OSS 20B, 3 seeds for Gemma-3 27B). Larger-scale replication and cross-benchmark validation (e.g., SecurityEval vs. CWEVAL) are needed for stronger statistical claims [28,24].

Hence, our claims should be interpreted as descriptive trends, not definitive statistical results.

5.5 Future Directions

Future work can build on these findings in several directions. First, broaden the task set to include other languages (e.g., Java, JavaScript) and frameworks. Second, investigate adaptive or automated prompt refinement mechanisms where prompts evolve based on vulnerability feedback from analyzers. Third, integrate formal verification and coverage-guided fuzzing to address both false positives and deeper logical flaws [5,11]. Finally, conduct user studies to examine how developers adopt or override security-oriented prompts in practice, especially under time pressure or productivity constraints [3,18].

Overall, secure vibe coding requires not only careful prompt engineering but also systematic integration with analysis and verification, bridging usability with assurance.

6 Conclusion

This paper investigated the security posture of *vibe coding*, a workflow in which developers rely on LLMs to translate natural-language task descriptions into executable code. Across ten Python tasks spanning common CWE categories, we compared baseline prompts with security-oriented prompts under a controlled evaluation pipeline supported by Bandit and lightweight runtime probes.

Our findings confirm prior work showing that LLM-generated code, when adopted without scrutiny, often contains exploitable weaknesses [23,7]. Expanding to ten random seeds with GPT-OSS 20B demonstrated that these trends hold consistently: insecure defaults remained the norm, while security-oriented prompts reduced the proportion and severity of vulnerabilities in several tasks. However, persistent classes of weaknesses were not eliminated, particularly in file upload handling and JWT authentication. A cross-model comparison with Gemma-3 27B further showed that improvements are not uniform across models: while GPT-OSS 20B benefited more consistently from improved prompts, Gemma-3 27B exhibited higher variability and continued vulnerabilities in tasks such as logging and authentication. These results echo recent surveys [26,6], underscoring that functional correctness does not guarantee security in AI-assisted development.

Several limitations should be acknowledged. Although we expanded to 10 seeds and included a second model, the evaluation still covered only two LLM families with fixed parameters, so the generality of results across larger model sets or alternative decoding strategies remains uncertain [2]. The ten Python tasks examined are grounded in OWASP Top 10 [22] but do not capture the full diversity of real-world programming. In addition, static analysis and lightweight probes offered partial coverage, potentially missing deeper logic flaws or domain-specific vulnerabilities [28]. Finally, the study did not benchmark against alternative evaluation suites such as CWEVAL [24], which limits comparability with emerging literature.

In sum, this work provides an expanded empirical basis for assessing secure prompting in vibe coding. It demonstrates measurable improvements from security-oriented prompts but also shows that prompting alone is insufficient, and that model choice influences the degree of security gains. Our results support a layered approach that combines prompting with automated analysis and verification mechanisms, moving toward more trustworthy adoption of vibe coding practices in software development.

Acknowledgments. This work was supported by JSPS Grant-in-Aid for Scientific Research Kiban (C) 23K11103.

References

1. Ben Matthews: Understanding the vibe coding trend and considerations for developers (2025), https://www.techradar.com/pro/understanding-the-vibe-coding-trend-and-considerations-for-developers, accessed 2025-08-20
2. Bhatt, M., Chennabasappa, S., Li, Y., Nikolaidis, C., Song, D., Wan, S., Ahmad, F., Aschermann, C., Chen, Y., Kapil, D., Molnar, D., Whitman, S., Saxe, J.: Cyberseceval 2: A wide-ranging cybersecurity evaluation suite for large language models (2024). https://doi.org/10.48550/arXiv.2404.13161
3. Cooper, N., Clark, A.T., Lecomte, N., Qiao, H., Ellison, A.M.: Harnessing large language models for coding, teaching and inclusion to empower research in ecology and evolution. Methods in Ecology and Evolution **15**(10), 1757–1763 (2024). https://doi.org/10.1111/2041-210X.14325
4. Cotroneo, D., Improta, C., Liguori, P., Natella, R.: Vulnerabilities in ai code generators: Exploring targeted data poisoning attacks. In: Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension. p. 280–292. ICPC '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3643916.3644416
5. Cramer, M., McIntyre, L.: Verifying llm-generated code in the context of software verification with ada/spark (2025). https://doi.org/10.48550/arXiv.2502.07728
6. Farjam, M., Meyer, H., Lohkamp, M.: A practical guide and case study on how to instruct llms for automated coding during content analysis. Social Science Computer Review **0**(0) (2025). https://doi.org/10.1177/08944393251349541
7. Fu, Y., Liang, P., Tahir, A., Li, Z., Shahin, M., Yu, J., Chen, J.: Security weaknesses of copilot-generated code in github projects: An empirical study. ACM Trans. Softw. Eng. Methodol. (2025). https://doi.org/10.1145/3716848
8. Google Cloud: What is vibe coding? (2025), https://cloud.google.com/discover/what-is-vibe-coding, accessed 2025-08-20
9. Google DeepMind and Google Research: Gemma 3 documentation (2025), https://ai.google.dev/gemma/docs/core, accessed 2025-09-14
10. Gupta, M., Akiri, C., Aryal, K., Parker, E., Praharaj, L.: From chatgpt to threatgpt: Impact of generative ai in cybersecurity and privacy. IEEE Access **11**, 80218–80245 (2023). https://doi.org/10.1109/ACCESS.2023.3300381
11. Li, Y.C., Zetzsche, S., Somayyajula, S.: Dafny as verification-aware intermediate language for code generation (2025). https://doi.org/10.48550/arXiv.2501.06283
12. Majeed, A., Hwang, S.O.: When ai meets information privacy: The adversarial role of ai in data sharing scenario. IEEE Access **11**, 76177–76195 (2023). https://doi.org/10.1109/ACCESS.2023.3297646
13. McCarthy, R.: Rules files for safer vibe coding (2025), https://www.wiz.io/blog/safer-vibe-coding-rules-files, accessed 2025-08-20
14. Misu, M.R.H., Lopes, C.V., Ma, I., Noble, J.: Towards ai-assisted synthesis of verified dafny methods. Proc. ACM Softw. Eng. **1**(FSE) (2024). https://doi.org/10.1145/3643763
15. MITRE Corporation: Common weakness enumeration (cwe) (2025), https://cwe.mitre.org, accessed 2025-08-20
16. Murdoch, B.: Privacy and artificial intelligence: challenges for protecting health information in a new era. BMC Medical Ethics **22**(1), 122 (2021). https://doi.org/10.1186/s12910-021-00687-3
17. Negri-Ribalta, C., Geraud-Stewart, R., Sergeeva, A., Lenzini, G.: A systematic literature review on the impact of ai models on the security of code generation. Frontiers in Big Data **7** (2024). https://doi.org/10.3389/fdata.2024.1386720

18. Nejjar, M., Zacharias, L., Stiehle, F., Weber, I.: Llms for science: Usage for code generation and data analysis. J. Softw. Evol. Process **37**(1) (2025). https://doi.org/10.1002/smr.2723
19. Nigel Powell: Vibe coding: How ai is making coding possible for everyone (2025), https://www.tomsguide.com/ai/vibe-coding, accessed 2025-08-20
20. Ollama Team: Ollama: Run large language models locally (2023), https://ollama.com/, accessed 2025-08-20
21. OpenAI, Inc: Gpt-oss (2025), https://github.com/openai/gpt-oss, accessed 2025-08-20
22. OWASP Foundation: Owasp top 10:2021 (2021), https://owasp.org/Top10/, accessed 2025-08-20
23. Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., Karri, R.: Asleep at the keyboard? assessing the security of github copilot's code contributions. Commun. ACM **68**(2), 96–105 (2025). https://doi.org/10.1145/3610721
24. Peng, J., Cui, L., Huang, K., Yang, J., Ray, B.: Cweval: Outcome-driven evaluation on functionality and security of llm code generation (2025), https://arxiv.org/abs/2501.08200
25. Perry, N., Srivastava, M., Kumar, D., Boneh, D.: Do users write more insecure code with ai assistants? In: CCS '23: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. p. 2785–2799. CCS '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3576915.3623157
26. Pirzado, F.A., Ahmed, A., Mendoza-Urdiales, R.A., Terashima-Marin, H.: Navigating the pitfalls: Analyzing the behavior of llms as a coding assistant for computer science students—a systematic review of the literature. IEEE Access **12**, 112605–112625 (2024). https://doi.org/10.1109/ACCESS.2024.3443621
27. PyCQA: Bandit: A security linter from pycqa (2014), https://bandit.readthedocs.io/, accessed 2025-08-20
28. Siddiq, M.L., Santos, J.C.S.: Securityeval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques. In: Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security. p. 29–33. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3549035.3561184
29. Varanasi, L.: Vibe coding is the future — just don't trust it (yet) (2025), https://www.businessinsider.com/vibe-coding-limits-use-cases-software-companies-airtable-redis-2025-7, accessed 2025-08-20
30. Warrior, S.C.: Ai security rules (2025), https://github.com/SecureCodeWarrior/ai-security-rules, accessed 2025-08-20