A.S.E: A Repository-Level Benchmark for Evaluating Security in AI-Generated Code

Anonymous ACL submission

Abstract

The increasing adoption of large language models (LLMs) in software engineering necessitates rigorous security evaluation of their generated code. However, existing benchmarks often lack relevance to real-world AI-assisted programming scenarios, making them inadequate for assessing the practical security risks associated with AI-generated code in production environments. To address this gap, we introduce A.S.E (AI Code Generation Security Evaluation), a repository-level evaluation benchmark designed to closely mirror real-world AI programming tasks, offering a comprehensive and reliable framework for assessing the security of AI-generated code. Our evaluation of leading LLMs on A.S.E reveals several key findings. In particular, current LLMs still struggle with secure coding. The complexity in repository-level scenarios presents challenges for LLMs that typically perform well on snippet-level tasks. Moreover, a larger reasoning budget does not necessarily lead to better code generation. These observations offer valuable insights into the current state of AI code generation and help developers identify the most suitable models for practical tasks. They also lay the groundwork for refining LLMs to generate secure and efficient code in real-world applications.

1 Introduction

The rapid advancement of large language models (LLMs) has greatly enhanced the AI programming ecosystem, with tools like Cursor (Cursor, 2025) and Claude Code (Anthropic, 2025) enabling developers to choose models that best fit their tasks. These AI assistants significantly improve programming efficiency, leading to a surge of AI-generated code in production environments. However, research (Siddiq and Santos, 2022; Vero et al., 2025; Peng et al., 2025; Hajipour et al., 2024; Li et al., 2025b; Fu et al., 2024b; He and Vechev, 2023; Pearce et al., 2025; Wang et al., 2024) has shown that such code can harbor security vulnerabilities, posing serious risks such as data breaches or system failures (Pearce et al., 2025; Siddiq and Santos, 2022; Fu et al., 2023; Li and Paxson, 2017). Relying on developers to ensure the security of AI-generated code can be highly challenging given the complexity of modern software systems. Therefore, *there is a pressing need to identify and utilize AI models that are capable of generating secure code*.

Despite the numerous benchmarks (Hendrycks et al., 2021; Dou et al., 2024; Chen et al., 2021; Austin et al., 2021; Li et al., 2025b; Vero et al., 2025; Peng et al., 2025; Fu et al., 2024a; Hajipour et al., 2024; Siddiq and Santos, 2022; Wang et al., 2024) developed by both academia and industry to evaluate AI-generated code, most of them (Hendrycks et al., 2021; Dou et al., 2024; Chen et al., 2021; Austin et al., 2021) primarily focus on code quality, such as syntax correctness and functional accuracy, while overlooking critical security considerations. Although some benchmarks (Li et al., 2025b; Vero et al., 2025; Peng et al., 2025; Fu et al., 2024a; Hajipour et al., 2024; Siddiq and Santos, 2022; Wang et al., 2024) attempt to address code security (as shown in Table 1), they are often *inadequate* to assess the actual security risks of AI-generated code in real-world production scenarios due to several key reasons: **(a) Limited relevance to real-world data.** These datasets are typically sourced from human-curated synthetic code snippets, which have limited relevance to the functional and security scenarios of real-world projects. **(b) Code generation tasks detached from real-world AI programming.** Their code generation tasks are typically limited to isolated code snippets, focusing solely on functional descriptions without considering the context within files or projects, which does not align with mainstream AI programming paradigm. **(c) Unreliable code evaluation methods.** Security assessments of generated code often rely on manual or LLM-based judgment, which are

unreliable and difficult to automate or reproduce consistently. This gap poses a significant challenge for both developers and organizations seeking to integrate AI-generated code securely into their systems.

To bridge this gap, we introduce A.S.E (AI Code Generation Security Evaluation), a repository-level evaluation benchmark designed to closely mirror real-world AI programming scenarios, offering a comprehensive and reliable framework for assessing the security of AI-generated code. Specifically, A.S.E has the following key design features: **(a) Real-world data source**: The dataset is derived from high-quality GitHub open-source repositories with documented CVEs. A.S.E leverages vulnerability-related code extracted from CVE patches, ensuring that the data reflects both realistic and security-sensitive scenarios. **(b) Simulation of real-world code generation tasks**: A.S.E mimics AI programming assistants like Cursor by extracting code contexts (including both intra-file and cross-file contexts) from the repository, and providing them to LLMs for code generation. **(c) High accuracy and reproducibility in code evaluation**: For each test case, corresponding to a specific CVE and repository, A.S.E designs targeted static vulnerability detection rules that can scan for the original CVE, ensuring accurate security assessment of the regenerated project.

Building on these design principles, the A.S.E benchmark includes 120 repository-level instances, consisting of 40 seed dataset collected from GitHub and 80 mutated variants generated through semantic and structural mutation techniques, such as identifier renaming and control-flow reshaping. These variants are introduced to mitigate data leakage risks, ensuring that the evaluation reflects the LLM's capabilities rather than its memorization.

Based on A.S.E, we evaluated 26 mainstream commercial or open-source models under the same experimental setup, leading to several key findings. First, existing LLMs still face significant challenges in secure coding. All models fall short in terms of security performance compared to their code quality performance (such as syntax correctness). Even the best-performing model, Claude-3.7-Sonnet, achieved only a total score of 52.79 in our evaluation. Second, A.S.E introduces significant complexity in repository-level scenarios, which presents a challenge for LLMs that typically perform well on snippet-level tasks. For example, although GPT-3 excels on SafeGenBench (Li et al., 2025b), its performance on the A.S.E. benchmark drops, falling behind many other models. Third, slow-thinking configurations, which allocate more deliberate computation or multi-step reflection, tend to underperform compared to fast-thinking configurations that rely on concise, direct decoding. This suggests that a larger reasoning budget does not necessarily lead to better code generation. These observations offer valuable insights into the current state of AI code generation, helping developers select the most appropriate models for their specific tasks. Furthermore, they provide a foundation for refining LLMs, enhancing their ability to generate secure and efficient code in real-world applications.

The main contributions of this paper are summarized as follows:

- **New repository-level benchmark from real code.** We release A.S.E, a repository-level evaluation benchmark derived from real-world GitHub repositories with documented CVEs. Unlike existing benchmarks, A.S.E is designed to closely mirror real-world AI programming tasks by leveraging vulnerability-related code from CVE patches, ensuring the data reflects both realistic and security-sensitive scenarios.
- **Automated and reproducible evaluation framework.** We develop a reproducible vulnerability-targeted evaluation framework that integrates custom vulnerability detection rules tailored to each data instance. Compared to previous work, this framework enables more automated and accurate code evaluation. It comprehensively considers the capabilities of AI-generated code, including security, quality, and generation stability.
- **Extensive experiments and findings.** We evaluate 26 mainstream commercial and open-source LLMs on A.S.E, revealing several key findings. These insights shed light on the current state of AI code generation, guiding developers in selecting the most suitable models for their tasks. Additionally, they lay the groundwork for refining LLMs to improve their ability to generate secure and efficient code in real-world applications.

2 Related Work

Benchmarks for AI-generated code evaluation generally fall into functionality-oriented and security-

oriented lines. Functionality benchmarks (Chen et al., 2021; Austin et al., 2021; Liu et al., 2024; Bogomolov et al., 2024; Ding et al., 2023; Liang et al., 2025; Li et al., 2025a; Jimenez et al., 2024) primarily measure syntactic validity and functional correctness (e.g., HumanEval (Chen et al., 2021) via unit tests), with limited emphasis on vulnerabilities. Security benchmarks (Siddiq and Santos, 2022; Vero et al., 2025; Peng et al., 2025; Hajipour et al., 2024; Li et al., 2025b; Fu et al., 2024b) explicitly evaluate security and reliability. A consolidated comparison of representative security benchmarks and A.S.E is provided in Table 1 (Appendix A).

2.1 Relevance to Real-world Scenarios

Early functionality benchmarks focus on small, self-contained tasks (Chen et al., 2021; Austin et al., 2021), while newer ones move toward long-context and repository-level settings (Liu et al., 2024; Bogomolov et al., 2024; Ding et al., 2023; Liang et al., 2025; Li et al., 2025a; Jimenez et al., 2024); SWE-Bench (Jimenez et al., 2024) is a representative example built from real projects. By comparison, many security benchmarks remain snippet-centric (Siddiq and Santos, 2022; Vero et al., 2025; Peng et al., 2025; Hajipour et al., 2024; Li et al., 2025b; Fu et al., 2024b), limiting their ability to reflect context-dependent vulnerabilities. A.S.E targets real-world CVE-derived tasks and incorporates repository context during generation.

2.2 Code Assessment Methods

Security evaluation spans manual review (Siddiq and Santos, 2022), LLM-as-judge (Bhatt et al., 2023), generic SAST (Hajipour et al., 2024; Li et al., 2025b), and test-based checks (Vero et al., 2025; Peng et al., 2025; Fu et al., 2024b). These approaches trade off scalability, robustness, and reproducibility (e.g., judge sensitivity or SAST miscalibration across projects). A.S.E follows a project- and CVE-aligned detection design to reduce ambiguity in evaluation.

3 The A.S.E Framework

This section introduces the A.S.E. framework, which includes three core components: benchmark construction (subsection 3.2), code generation task setup (**??**), and code evaluation (**??**), as shown in Figure 1. We will first highlight the key features of the A.S.E. design, followed by a detailed discussion of each of these three core components.

3.1 Design Philosophy

We aim to create a repository-level evaluation benchmark that mirrors real-world AI programming scenarios, offering a reliable framework for assessing the security of AI-generated code. To ensure accurate results, A.S.E. focuses on realistic data sources, task settings, and code assessment methods. Specifically, the core features of the A.S.E benchmark are designed around the following principles:

(i) Data Source: Real-world and Repository-Level Data Sources. To reflect the performance of large models in real-world software environments, A.S.E constructs tasks from active open-source repositories with documented CVEs and verifiable patches. It utilizes vulnerability-related code extracted from CVE patches, ensuring the data captures realistic and security-sensitive scenarios. To mitigate the risk of data leakage, A.S.E employs semantic and structural mutation techniques, such as identifier renaming and control-flow reshaping, on the collected real-world repositories. These variants help prevent data leakage and ensure that the evaluation reflects the LLM's capabilities, not its memorization.

(ii) Task Settings: Practical Simulations of AI Programming Workflows. To simulate realistic usage, A.S.E replicates AI programming assistants like Cursor by extracting code contexts—both intra-file and cross-file—directly from the repository. These contexts are then provided to LLMs for code generation, closely mimicking real-world AI programming scenarios. Moreover, the generated code is output in the form of `diff` files, allowing patches to be applied directly to the repository, further reflecting real software development practices.

(iii) Code Assessment: High Accuracy and Reproducibility Assessment. Instead of relying on manual or LLM-based judgment, which can be unreliable and difficult to reproduce consistently, A.S.E designs targeted static vulnerability detection rules for each test case, corresponding to a specific CVE and repository. These rules are tailored with dedicated source–sink definitions and taint propagation paths to successfully detect the original CVE, thereby ensuring an accurate security assessment of the regenerated project.

Following these guidelines, we introduce the A.S.E benchmark to evaluate in repository-level code generation regarding security. After that, we detail the three key steps: benchmark construction,

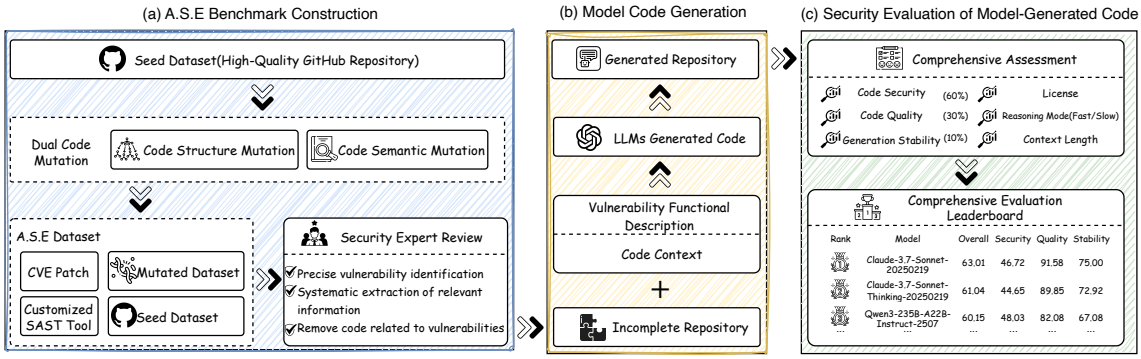

Figure 1: Overall workflow of A.S.E. (a) A.S.E benchmark construction: from high-quality GitHub seeds, we build the A.S.E dataset via dual mutations (structure/semantic), CVE patches, and a customized SAST tool, followed by expert curation. (b) Model code generation: given an incomplete repository, a vulnerability description and context guide LLMs to complete the repository. (c) Security evaluation: comprehensive assessment with security, quality and stability.

task design, and result evaluation.

3.2 A.S.E Benchmark Construction

Guided by our design philosophy, we construct the A.S.E benchmark as shown in Figure 1(a) and Figure 2. To ensure realism and adequate security expertise, we form a team of ten contributors (five Ph.D. candidates and five master's students) from top-tier universities with strong backgrounds in cybersecurity and web development. All contributors have hands-on experience in vulnerability discovery and remediation, focusing on common web issues (e.g., XSS, SQL injection, and path traversal), and are familiar with secure coding and static analysis. Benchmark construction proceeds in four stages: determining data sources, filtering candidate repositories, expert-guided refinement and quality control, and dataset expansion.

The construction of the benchmark proceeds in four stages: determining data sources, filtering candidate repository, expert-guided refinement and quality filtering, and dataset expansion.

Step 1: Determining Data Sources. We collect CVE records and their associated repositories from public vulnerability databases and enterprise-internal sources. We require accessible commit histories to locate vulnerable code precisely and to support task construction. This step yields over 100,000 raw CVE entries as the starting pool.

Step 2: Filtering Candidate Repositories. Starting from raw CVE entries, we apply a multi-stage filtering and verification pipeline to ensure both project quality and a verifiable vulnerability–fix linkage. We first retain only CVEs that (i) fall into web-relevant categories in the 2024 Top CWE list (Corporation, 2025) and (ii) provide traceable fixing-commit contexts, so that each instance is grounded in a concrete code change rather than an abstract vulnerability description. We then enforce repository quality by requiring either active monthly maintenance or a popularity threshold of over 1,000 GitHub stars, which removes abandoned projects and preserves realistic codebases with sufficient complexity. After these filters, the pool is reduced to approximately 50,000 candidate repositories.

To further strengthen evidence and reduce noise, we run multiple SAST tools (e.g., CodeQL (CodeQL, 2025) and Joern (Yamaguchi et al., 2014)) on the candidates and intersect their findings with the lines modified by the fixing commits. We keep only cases where the SAST alerts overlap with the actual modified lines, which simultaneously suppresses false positives and establishes an explicit vulnerability–fix causal chain. This design ensures that each retained instance is (i) practically observable by automated analysis and (ii) correctly anchored to the corresponding fix commit, producing 199 high-confidence candidates for subsequent expert refinement.

Step 3: Expert-Guided Refinement and Quality Control. To ensure the final benchmark is genuinely security-relevant and reflects real-world vulnerabilities that are both detectable and reproducible, we refine the dataset through expert annotation and validation. Specifically, we first conduct an initial manual review to further control data quality. At this stage, security experts remove obvious false positives introduced by static analysis tools and discard commits that modify an excessive num-

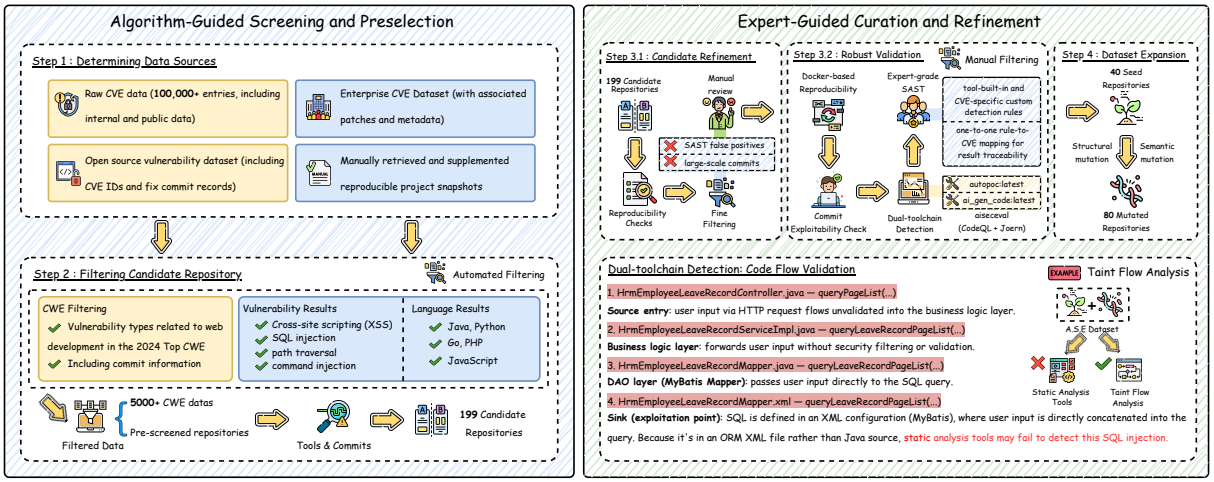

Figure 2: Overview of A.S.E benchmark construction. **Algorithm-guided screening and preselection (left):** aggregate CVE-linked sources and automatically filter repositories by web-related CWEs, vulnerability types (XSS, SQL injection, path traversal, command injection), and languages (Java, Python, Go, PHP, JavaScript). **Expert-guided curation and refinement (right):** conduct manual review, reproducibility and exploitability checks, and dual-toolchain SAST (e.g., CodeQL + Joern) with CVE-specific rules; then expand 40 seed repositories via structural/semantic mutation to 80 variants.

ber of files (e.g., more than 10), since large-scale changes obscure vulnerability localization and hinder precise labeling. Building on the cleaned candidate set, we then proceed with a fine-grained expert analysis that focuses on the vulnerabilities themselves. Security experts precisely annotate the vulnerable code regions, reconstruct the relevant execution context (e.g., source/sink signatures, API definitions, call chains), and design targeted CodeQL/Joern queries to validate taint propagation paths. Once validated, the labeled vulnerable code is removed to create a fill-in-the-code setting. By combining the functional description of the vulnerability with its extracted context, we generate structured prompts that require models to reason over repository-level structures and logic rather than isolated snippets. After final expert review, 40 repositories with verified CVE records are retained as the seed dataset, each anchored at a baseline commit that provides a stable starting point for task construction and evaluation. This expert-driven process guarantees authenticity, reliability, and reproducibility across all benchmark tasks.

Step 4: Dataset Expansion. We expand the seed tasks with semantics-preserving transformations to improve coverage and robustness. We apply (i) semantic transformations (e.g., systematic renaming and equivalent API substitution) and (ii) structural transformations (e.g., control-flow edits, call-graph refactoring, and file-layout reorganization). These changes preserve behavior while reducing surface overlap with public code that may appear in training corpora. Overall, we generate 80 additional variants from the 40 seed repositories, resulting in 120 benchmark instances.

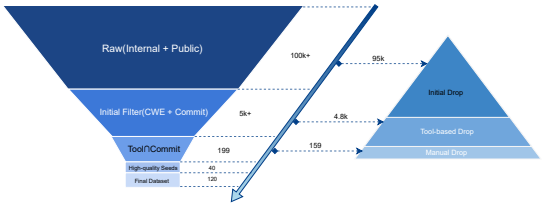

Figure 3: Benchmark Construction Funnel.

General Statistics. Figure 3 illustrates the data reduction pipeline from the initial collection of raw CVE entries to the final benchmark. The funnel chart presents the number of instances retained after each stage of filtering and refinement, showing how the dataset was progressively narrowed from a large pool of raw vulnerabilities to a carefully curated benchmark. The resulting A.S.E benchmark comprises 120 repository-level vulnerability instances and the overall composition is illustrated in Figure 4.

Specifically, the dataset targets four categories of vulnerabilities that are widely prevalent in real-world web projects, each aligned with a CWE entry: SQL Injection (29.2%, CWE-89), Path Traversal (26.7%, CWE-22), Cross-Site Scripting (25.0%, CWE-79), and Command Injection (19.2%, CWE-78). This mapping defines the dataset at the CWE

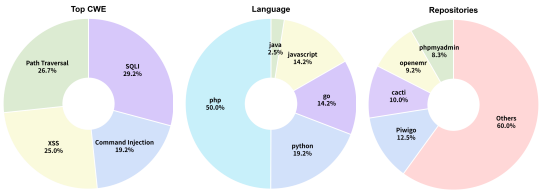

Figure 4: Statistics of A.S.E benchmark, including the distribution of top CWE categories, programming languages, and repositories.

level, ensuring that evaluation tasks align with security-critical issues that LLMs must account for when generating code in real-world development. Each category captures a distinct challenge where secure functionality requires the model not only to implement business logic correctly but also to avoid unsafe coding patterns:

- Cross-Site Scripting (XSS): evaluates whether the model can generate web logic (e.g., input/output rendering) while preventing injection of malicious scripts into trusted contexts.
- SQL Injection (SQLI): tests whether the model, when generating database operation logic, properly handles user input and avoids unsafe SQL statement construction.
- Path Traversal: examines if the model can implement file access functionality without exposing sensitive paths outside the intended directory scope.
- Command Injection: assesses whether the model can generate code involving system interactions while preventing the execution of unauthorized operating system commands.

From a language perspective, A.S.E spans five mainstream programming environments to reflect realistic multi-language software development. The distribution is concentrated in PHP (50.0%), followed by Python (19.2%), Go (14.2%), JavaScript (14.2%), and Java (2.5%). This distribution highlights the dominance of PHP in vulnerability-prone web applications while also enabling evaluation of model generalization across diverse programming languages.

We also analyze the size of the vulnerable code that define each code generation task. Specifically, the number of vulnerable lines of code (LOC) per task varies substantially, with an *average* of 35.77, a *median* of 18, and a *range* of [2–415]. These statistics characterize the functional code fragments that models are required to regenerate, highlighting the variation in task complexity—from small, localized code edits spanning only a few lines to larger segments involving multiple statements or function bodies. This diversity ensures that the benchmark captures both simple and complex generation scenarios under realistic repository-level settings.

For tooling integration, we incorporate two state-of-the-art static analysis frameworks—CodeQL and Joern—which are packaged into containerized environments to ensure reproducibility and ease of deployment. Each tool is applied to 50% of the benchmark instances, providing complementary static analysis capabilities for security evaluation. The containerization not only standardizes the execution environment across different platforms but also guarantees that results are consistent and reproducible.

3.3 Evaluation Pipeline (Overview)

A.S.E evaluates models with a repository-level, two-stage pipeline: (i) *code generation*, where an LLM produces a repository-aware patch for a masked vulnerable region using retrieved project context; and (ii) *code assessment*, where the patch is applied and evaluated along *Quality*, *Security*, and *Stability*. We defer full pipeline details, prompt composition, and metric definitions to Appendix C.

4 Experiments

We evaluate a representative set of 26 state-of-the-art LLMs on the A.S.E benchmark to examine their code security generation capabilities. This diverse selection includes both proprietary families (e.g., Claude 3.7/4, GPT-4o, Grok-3/4, Gemini 2.5) and widely-adopted open-source models (e.g., Qwen3, DeepSeek-V3/R1, GLM-4.5), covering both "fast thinking" and "slow thinking" reasoning paradigms. A list of the evaluated models and the experimental environment are provided in Appendix D.

4.1 Overall Results

For completeness and reproducibility, we report the full leaderboard in Table 2 (Appendix B). We summarize the key observations below. Table 2 reveals a substantial gap between code quality and security: while most models produce correct and useful code, none surpass the 50-point threshold on Code Security. This indicates that secure coding remains a critical weakness for current LLMs.

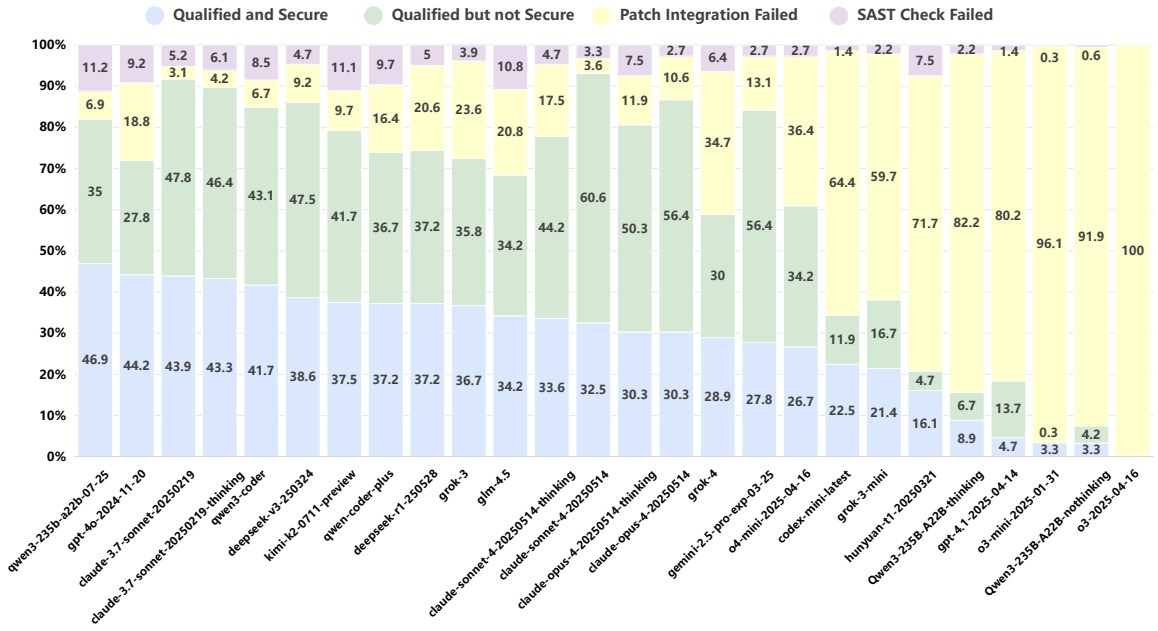

Figure 5: Attributional distribution across Code LLMs. Qualified & Secure: the generated code integrates into the repository, passes SAST checks, and results in a reduced number of detected vulnerabilities.; Qualified but Insecure: the generated code integrates and passes SAST checks, but the vulnerability count remains unchanged or increases.; Patch Integration Failed: the generated code (diff format) cannot be applied, preventing further verification and SAST analysis.; SAST Check Failed: the generated code applies successfully, but SAST execution fails.

Moreover, A.S.E effectively exposes weaknesses in secure code generation under repository-level settings, where models must resolve cross-file dependencies and handle long-context reasoning beyond snippet-level generation. Consequently, models that perform well on snippet-oriented benchmarks, such as GPT-o3 on SafeGenBench (Li et al., 2025b), can experience a noticeable drop in performance.

Among the evaluated models, Claude-3.7-Sonnet achieves the highest overall score (63.01) and a strong Code Quality score (91.58), yet its Code Security score remains below 47. Similarly, Claude-Sonnet-4 obtains the best Code Quality performance but only 34.78 in Code Security. In contrast, GPT-o3 exhibits extremely high Generation Stability (98.91) but fails almost completely in security and quality. Similar patterns appear in GPT-4.1 and Qwen3-235B-A22B, suggesting that high stability does not guarantee secure code.

Moreover, Figure 5 presents the distribution of code generation outcomes for each model, categorized into four types: qualified and secure, qualified but insecure, patch integration failed, and SAST check failed. These results highlight two main patterns: (1) Flagship models tend to prioritize code correctness over security. For example, Claude-3.7-Sonnet generates 91.7% qualified code, yet 43.8% of it remains insecure. (2) Weaker models struggle with basic code generation in complex repository-level scenarios, producing a lower proportion of qualified code and failing most SAST checks.

4.2 Analysis and Findings

In this section, we examine model performance from multiple complementary perspectives. We focus on the most critical insights regarding model categories, reasoning paradigms, task-level challenges, and benchmark robustness. Extended analyses regarding architectural effects, scaling laws, and the decoupling of stability and security are provided in Appendix E.

I. Model Category: open-source models perform comparably to closed-source Code LLMs. Our results show a narrowing performance gap between open-source and proprietary Code LLMs. While top-tier closed-source models like the Claude series maintain a slight edge in Code Quality, prominent open-source representatives such as Kimi-K2 and Qwen3-235B-A22B-Instruct demonstrate comparable overall performance and even superior generation stability. This parity suggests that state-of-the-art open-source models have become competitive in secure code generation.

II. Reasoning Paradigms: Slow-thinking can lead to security regressions. Contrary to ex-

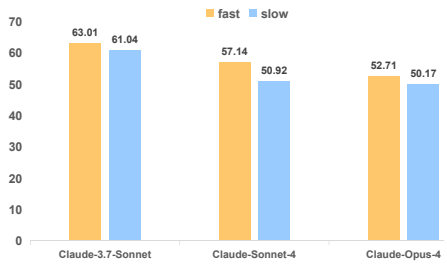

Figure 6: Overall performance comparison of fast vs. slow thinking modes in the Claude series.

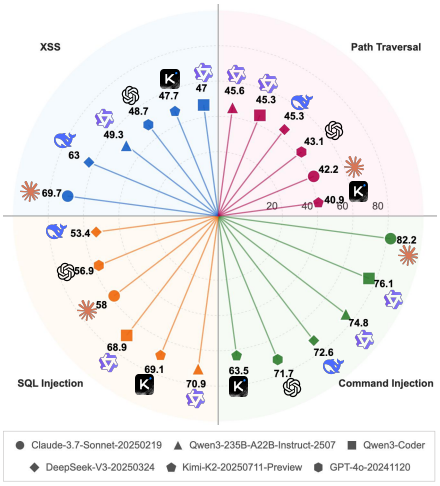

Figure 7: Detailed performance of various Code LLMs across four task categories of A.S.E benchmark.

pectations, deliberate reasoning paradigms ("slow-thinking") often underperform in Code Security compared to their "fast-thinking" counterparts. As shown in Figure 6, this trend is consistent across the Claude series, where thinking modes exhibit noticeable performance drops in security metrics. This suggests that while slow-thinking enhances complex reasoning, it may inadvertently introduce vulnerabilities. This degradation potentially stems from the generation of overly complex logic or a lack of security-specific reinforcement during the extended reasoning process.

III. Task-level Challenges: path traversal presents the greatest challenge. As shown in Figure 7,Path Traversal is consistently the most challenging task. Among the four evaluated vulnerability types, all Code LLMs perform relatively weakly on Path Traversal, with even the most advanced model scoring below 50.0. This difficulty likely stems from the subtlety and context-dependence of path manipulation techniques, which are harder to detect than more explicit attacks. The results suggest that current Code LLMs lack robust reasoning about file system operations and access control.

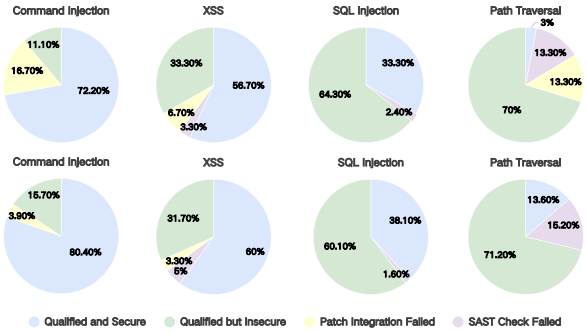

Figure 8: Detailed Attribution classification of Claude-3.7-Sonnet: Original (top) and Mutation Test (bottom).

IV. Benchmark Consistency Across Original and Mutated Datasets. We observe minimal performance variation between the original benchmark and its mutated variants, suggesting that A.S.E is robust and free from substantial data leakage. As illustrated in Figure 8, the error distributions for Claude-3.7-Sonnet remain consistent across both datasets. For Path Traversal and SQL Injection, the model frequently produces qualified yet insecure code, while for XSS and Command Injection, it tends to generate secure and qualified outputs.

5 Conclusion

In this work, we introduced A.S.E, the first repository-level benchmark designed to evaluate the security of AI-generated code. By leveraging real-world projects with documented CVEs, repository-level context, and customized static detection rules, A.S.E provides a more authentic and rigorous evaluation framework than previous snippet-oriented benchmarks. Our extensive evaluation of 26 state-of-the-art LLMs reveals a critical "quality-security gap": while current models excel in functional correctness, they frequently generate insecure code in complex, repository-level scenarios. These findings underscore the limitations of current reasoning paradigms and highlight the urgent need for security-aware model alignment. Ultimately, A.S.E serves as both a diagnostic tool for developers and a foundation for the next generation of secure-by-design LLMs. By bridging the gap between isolated code generation and real-world software engineering, A.S.E marks a substantial step toward ensuring that AI-assisted programming is not only efficient but also reliable and secure.

Limitations

Despite its contributions, A.S.E has several limitations. First, the current scope of A.S.E is restricted to web-related projects, four vulnerability categories, and five programming languages. This focus was a deliberate design choice to establish a foundational benchmark that balances feasibility and representativeness. Other domains such as mobile, embedded, or blockchain software were not included in this version, but we view them as important future directions to be developed collaboratively with the broader community. Second, the evaluation framework relies on customized static analysis rules for code security assessment. Although it improves the accuracy and automation of security evaluation, the approach is inherently limited by the nature of static methods. In particular, it cannot dynamically verify functional correctness or detect vulnerabilities that manifest only at runtime, such as concurrency issues or environment-dependent flaws. Finally, while A.S.E leverages repository-level context and patch-based evaluation to simulate real-world workflows, it cannot fully capture the diversity and unpredictability of software engineering practices in production environments. Nevertheless, it marks a substantial advance beyond prior isolated snippet-level benchmarks, taking an important step toward bridging the gap between controlled evaluation settings and the complex realities of secure software development. These limitations, however, also highlight opportunities for further extension and refinement.

References

Anthropic. 2024. Claude 3.5 sonnet technical report. https://www.anthropic.com/news/claude-3-5-sonnet.

Anthropic. 2025. Claude code homepage. https://www.anthropic.com/claude-code.

Anthropic. 2025. System card: Claude opus 4 & claude sonnet 4. Technical report, Anthropic. PDF.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models.

Manish Bhatt, Sahana Chennabasappa, Cyrus Nikolaidis, Shengye Wan, Ivan Evtimov, Dominik Gabi, Daniel Song, Faizan Ahmad, Cornelius Aschermann, Lorenzo Fontana, Sasha Frolov, Ravi Prakash Giri, Dhaval Kapil, Yiannis Kozyrakis, David LeBlanc, James Milazzo, Aleksandar Straumann, Gabriel Synnaeve, Varun Vontimitta, and 2 others. 2023. Purple llama cyberseceval: A secure coding benchmark for language models.

Egor Bogomolov, Aleksandra Eliseeva, Timur Galimzyanov, Evgeniy Glukhov, Anton Shapkin, Maria Tigina, Yaroslav Golubev, Alexander Kovrigin, Arie van Deursen, Maliheh Izadi, and Timofey Bryksin. 2024. Long code arena: a set of benchmarks for long-context code models.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code.

GitHub / CodeQL. 2025. Codeql documentation. https://codeql.github.com/docs/.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit S. Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, Luke Marris, Sam Petulla, Colin Gaffney, Asaf Aharoni, Nathan Lintz, Tiago Cardal Pais, Henrik Jacobsson, Idan Szpektor, Nan-Jiang Jiang, and 81 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *CoRR*, arXiv:2507.06261.

The MITRE Corporation. 2025. 2024 cwe top 25 cwe. https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html.

Cursor. 2025. Cursor documentation. https://docs.cursor.com/en/welcome.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, and 81 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *CoRR*, arXiv:2501.12948.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, and 80 others. 2024. Deepseek-v3 technical report. *CoRR*, arXiv:2412.19437.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion.

Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, and 1 others. 2024. Stepcoder: Improve code generation with reinforcement learning from compiler feedback. *arXiv preprint arXiv:2402.01391*.

Yanjun Fu, Ethan Baker, and Yizheng Chen. 2024a. Constrained decoding for secure code generation.

Yanjun Fu, Ethan Baker, Yu Ding, and Yizheng Chen. 2024b. Constrained decoding for secure code generation. *Preprint*, arXiv:2405.00218.

Yujia Fu, Peng Liang, Amjed Tahir, Zengyang Li, Mojtaba Shahin, Jiaxin Yu, and Jinfu Chen. 2023. Security weaknesses of copilot generated code in github. *arXiv preprint arXiv:2310.02059*.

Hossein Hajipour, Keno Hassler, Thorsten Holz, Lea Schönherr, and Mario Fritz. 2024. Codelmsec benchmark: Systematically evaluating and finding security vulnerabilities in black-box code language models.

Jingxuan He and Martin Vechev. 2023. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1865–1879.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *NeurIPS Datasets and Benchmarks*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, and 1 others. 2024. Qwen2.5-coder technical report.

Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, Aleksander Madry, Alex Baker-Whitcomb, Alex Beutel, Alex Borzunov, Alex Carney, Alex Chow, Alex Kirillov, Alex Nichol, Alex Paino, and 79 others. 2024. Gpt-4o system card. *CoRR*, arXiv:2410.21276.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues?

Frank Li and Vern Paxson. 2017. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215.

Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. 2025a. Fea-bench: A benchmark for evaluating repository-level code generation for feature implementation.

Xinghang Li, Jingzhe Ding, Chao Peng, Bing Zhao, Xiang Gao, Hongwan Gao, and Xinchen Gu. 2025b. Safegenbench: A benchmark framework for security vulnerability detection in llm-generated code.

Shanchao Liang, Nan Jiang, Yiran Hu, and Lin Tan. 2025. Can language models replace programmers for coding? REPOCOD says 'not yet'.

Tianyang Liu, Canwen Xu, and Julian J. McAuley. 2024. Repobench: Benchmarking repository-level code auto-completion systems.

OpenAI. 2025a. Model release notes. https://help.openai.com/en/articles/9624314-model-release-notes.

OpenAI. 2025b. Models: codex-mini-latest. https://platform.openai.com/docs/models/codex-mini-latest.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2025. Asleep at the keyboard? assessing the security of github copilot's code contributions. *Communications of the ACM*, 68(2):96–105.

Jinjun Peng, Leyi Cui, Kele Huang, Junfeng Yang, and Baishakhi Ray. 2025. Cweval: Outcome-driven evaluation on functionality and security of LLM code generation.

Stephen E. Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: BM25 and beyond.

Mohammed Latif Siddiq and Joanna CS Santos. 2022. Securityeval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, and 1 others. 2025. Kimi k2: Open agentic intelligence.

Tencent Hunyuan. 2025. Reasoning efficiency redefined! meet tencent's 'hunyuan-t1'—the first mamba-powered ultra-large model. https://tencent.github.io/llm.hunyuan.T1/README_EN.html.

Mark Vero, Niels Mündler, Victor Chibotaru, Veselin Raychev, Maximilian Baader, Nikola Jovanovic, Jingxuan He, and Martin T. Vechev. 2025. Baxbench: Can llms generate correct and secure backends?

Jiexin Wang, Xitong Luo, Liuwen Cao, Hongkui He, Hailin Huang, Jiayuan Xie, Adam Jatowt, and Yi Cai. 2024. Is your ai-generated code really safe? evaluating large language models on secure code generation with codeseceval. *arXiv preprint arXiv:2407.02395*.

xAI. 2025a. Grok 3 beta — the age of reasoning agents. https://x.ai/news/grok-3.

xAI. 2025b. Grok 4. https://x.ai/news/grok-4.

Fabian Yamaguchi, Nico Golde, Dan Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. *2014 IEEE Symposium on Security and Privacy*, pages 590–604.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *CoRR*, arXiv:2505.09388.

Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, and 1 others. 2025. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models.

Appendix

This appendix provides supplementary information to support the findings and methodology discussed in the main text. In Appendix D, we detail the evaluated models and the hardware/software configurations of our experimental setup. Appendix E presents an extended analysis of model performance, focusing on architectural influences, scaling laws, and the decoupling of generation stability from security. To provide a qualitative perspective, Appendix F conducts a granular case study on SQL injection tasks, illustrating representative error modes in repository-level generation. Finally, Appendix G discusses potential real-world applications of the A.S.E benchmark and outlines directions for future research.

A Comparison of Security Code Generation Benchmarks

This part provides a consolidated comparison table of representative security-oriented benchmarks and A.S.E. We summarize each benchmark along key axes that are frequently conflated in the main text: (i) task provenance (synthetic vs. derived from real-world projects/CVEs), (ii) context granularity (snippet/function vs. repository-level context), and (iii) security assessment protocol (manual review, LLM-as-judge, generic SAST, test-based checks, or customized project-specific detectors). The table is intended to support reproducibility and to clarify how A.S.E differs from prior settings beyond headline task descriptions.

B Full Leaderboard of Evaluated Models

This appendix reports the full leaderboard for all 26 evaluated LLMs on A.S.E (Table 2). We include the complete set of models and metrics to ensure transparency and reproducibility, while keeping the main text focused on high-level trends. The reported scores correspond to the evaluation protocol and experimental environment described in Appendix D.

C Evaluation Pipeline Details

A.S.E adopts a repository-level two-stage pipeline that emulates real-world patching: *Code Generation* followed by *Code Assessment*. The first stage constructs a fill-in-the-code task over a real repository state, and the second stage evaluates the integrated patch across quality, security, and stability.

C.0.1 Code Generation

For each benchmark instance, A.S.E retrieves the corresponding GitHub repository and checks out a fixed baseline commit that contains the vulnerability. Expert annotations identify the vulnerable region in the target file, which is masked and replaced by a special token `<masked>`, yielding a fill-in-the-code setting.

The model input contains two components: (i) the masked file with a functional description of the vulnerability generated by Claude-Sonnet-4 (Anthropic, 2025) and refined by experts; and (ii) repository-level context, including the project README and a set of related files retrieved via BM25 ranking (Robertson and Zaragoza, 2009). Models are instructed to output a unified-diff patch so that it can be applied automatically (e.g., via `git apply`). To assess run-to-run variability, each instance is executed three times under identical, containerized conditions.

C.0.2 Code Assessment

Given a generated patch, A.S.E first performs a *quality pre-check*, since security is meaningful only when the code can be integrated and analyzed. Specifically, we apply the patch to the baseline repository and run essential static checks (e.g., syntax verification and tool execution sanity). If integration or basic checks fail, the attempt is treated as unsuccessful for downstream security evaluation.

We then assess security by measuring whether vulnerability alerts decrease after integrating the generated patch. For each instance, we use expert-crafted static analysis rules tailored to the target CVE, explicitly modeling sources, sinks, and taint propagation patterns. Because a single rule set may produce multiple alerts within a project, we use the *relative change* in alert counts as a more robust signal than a single binary label.

Finally, to characterize variability of LLM outputs, we compute a stability score based on the consistency of results across repeated runs.

C.0.3 Metrics

Quality. Quality measures whether the generated patch is successfully integrated into the repository and passes essential static checks. A test is successful only if the patch applies cleanly and satisfies both static analysis and syntax checks:

$$\text{Quality} = \frac{1}{N}\sum_{t=1}^{N} q_t, \tag{1}$$

where N is the number of tests and $q_t = 1$ if test t merges and passes all checks, and $q_t = 0$

Table 1: Comparison of Security-Oriented Code Generation Evaluation Datasets.

Dataset	CWE Tags	Granularity (Repo/Snippet)	Provenance	Domain	Open Source	Security Eval.
A.S.E (Ours)	✓	**Repository**	Real-World Repos	Realistic Full-Web Repositories	✓	SAST
SafeGenBench	✓	Snippet	Human-Curated Synthetic	Simplified Programming Tasks	✓	SAST + LLM
BaxBench	✗	Snippet	Human-Curated Synthetic	Backend Programming Tasks	—	Test Cases
CWEval	✓	Snippet	Human-Curated Synthetic	Simplified Programming Tasks	✓	Test Cases
CODEGUARD+	✓	Snippet	Human-Curated Synthetic	Simplified Programming Tasks	—	Test Cases
CodeLMSec	✓	Snippet	Human-Curated Synthetic	Simplified Programming Tasks	✓	SAST
SecurityEval	✓	Snippet	Human-Curated Synthetic	Simplified Programming Tasks	✓	SAST + Manual

Table 2: The leaderboard of various advanced Code LLMs on the A.S.E. benchmark. ⚡ represents the fast-thinking mode and 🤔 indicates slow-thinking mode.

Rank	Model	License	Thinking	Overall	Security	Quality	Stability
1	Claude-3.7-Sonnet-20250219	Proprietary	⚡	63.01	46.72	91.58	75.00
2	Claude-3.7-Sonnet-Thinking-20250219	Proprietary	🤔	61.04	44.65	89.85	72.92
3	Qwen3-235B-A22B-Instruct-2507	Open Source	⚡	60.15	48.03	82.08	67.08
4	Qwen3-Coder	Open Source	⚡	59.31	42.69	85.16	81.54
5	DeepSeek-V3-20250324	Open Source	⚡	58.59	40.89	85.87	82.94
6	Claude-Sonnet-4-20250514	Proprietary	⚡	57.14	34.78	92.37	85.65
7	Kimi-K2-20250711-Preview	Open Source	⚡	55.29	37.82	79.90	86.25
8	GPT-4o-20241120	Proprietary	⚡	55.10	45.65	72.46	59.67
9	Qwen-Coder-Plus-20241106	Proprietary	⚡	53.55	37.98	73.78	86.27
10	Claude-Opus-4-20250514	Proprietary	⚡	52.71	31.95	85.82	77.91
11	Grok-3	Proprietary	⚡	52.18	38.64	73.54	69.41
12	DeepSeek-R1-20250528	Open Source	🤔	51.76	38.01	74.39	66.38
13	Gemini-2.5-Pro-Exp-20250325	Proprietary	⚡	51.02	29.98	84.04	78.21
14	Claude-Sonnet-4-Thinking-20250514	Proprietary	🤔	50.92	34.10	76.81	74.22
15	Claude-Opus-4-Thinking-20250514	Proprietary	🤔	50.17	30.70	79.84	77.98
16	GLM-4.5	Open Source	⚡	49.80	35.92	70.24	71.74
17	Grok-4	Proprietary	⚡	42.40	29.53	59.78	67.42
18	o4-mini-20250416	Proprietary	🤔	41.35	27.87	60.74	64.07
19	Grok-3-mini	Proprietary	⚡	30.49	22.37	38.15	56.26
20	Codex-mini-latest	Proprietary	⚡	29.71	22.96	34.68	55.29
21	Hunyuan-T1-20250321	Proprietary	🤔	21.92	15.57	20.21	65.18
22	Qwen3-235B-A22B-Thinking	Open Source	🤔	18.11	9.42	15.60	77.81
23	GPT-4.1-20250414	Proprietary	⚡	17.26	5.26	16.46	91.66
24	Qwen3-235B-A22B	Open Source	⚡	13.37	3.34	7.27	91.86
25	o3-mini-20250131	Proprietary	🤔	13.23	3.67	3.91	98.57
26	o3-20250416	Proprietary	🤔	10.22	0.36	0.36	98.91

otherwise.

Security. Security measures whether the integrated patch reduces detected vulnerabilities under the instance-specific static analysis rules:

$$\text{Security} = \frac{1}{N}\sum_{t=1}^{N} s_t, \tag{2}$$

where $s_t = 1$ if $v_{\text{after}}(t) < v_{\text{before}}(t)$ and $s_t = 0$ otherwise, and $v_{\text{before}}(t)$ / $v_{\text{after}}(t)$ denote the numbers of detected alerts before and after patch integration.

Stability. Stability measures consistency across repeated runs for the same benchmark instance. For each instance $i \in \mathcal{B}$, we compute the standard deviation over three runs, denoted as σ_i. We convert lower variation to a higher score via min–max normalization:

$$\tilde{\sigma}_i = \begin{cases} 1 - \dfrac{\sigma_i - \sigma_{\min}}{\sigma_{\max} - \sigma_{\min}}, & \text{if } \sigma_{\max} > \sigma_{\min}, \\ 1, & \text{otherwise,} \end{cases} \tag{3}$$

where $\sigma_{\min} = \min_i \sigma_i$ and $\sigma_{\max} = \max_i \sigma_i$. The stability score is:

$$\text{Stability} = \frac{1}{|\mathcal{B}|}\sum_{i \in \mathcal{B}} \tilde{\sigma}_i. \tag{4}$$

Overall. We aggregate the three dimensions with fixed weights:

$$\text{Overall} = 0.6 \times \text{Security} + 0.3 \times \text{Quality} + 0.1 \times \text{Stability}. \tag{5}$$

These weights reflect practical priorities: security is the primary objective, quality ensures feasibility of integration, and stability captures consistency without dominating the aggregate score.

D Experimental Details

D.1 Evaluated Models

We choose a total of 26 state-of-the-art LLMs, consisting of 18 proprietary models and 8 open-source models. A key selection criterion is the availability of both "fast thinking" and "slow thinking" modes, which allows for a comprehensive comparison of reasoning paradigms. For the proprietary models, our evaluation covers flagship families. This includes the Claude series (3.7-Sonnet (Anthropic, 2024), Sonnet-4 (Anthropic, 2025), Opus-4 (Anthropic, 2025)) and their "thinking" counterparts, the GPT family (GPT-4o (Hurst et al., 2024), GPT-4.1 (OpenAI, 2025a), Codex-mini (OpenAI, 2025b), and additional variants), the Grok series (Grok-3 (xAI, 2025a), Grok-4 (xAI, 2025b), Grok-3-mini (xAI, 2025a)), Gemini-2.5-Pro (Comanici et al., 2025), Qwen-Coder-Plus (Hui et al., 2024), and Hunyuan-T1 (Tencent Hunyuan, 2025). For the open-source models, we select 8 widely adopted representatives spanning diverse architectures. The set includes the Qwen3 series (Yang et al., 2025) (Qwen3-235B-A22B-Instruct, Qwen3-Coder, Qwen3-235B-A22B), DeepSeek-V3 (DeepSeek-AI et al., 2024), DeepSeek-R1 (DeepSeek-AI et al., 2025), Kimi-K2 (Team et al., 2025), and GLM-4.5 (Zeng et al., 2025).

D.2 Experiment Setup

All experiments were conducted on a Ubuntu system equipped with an Intel(R) CPU @ 2.50GHz, 16 threads, and 32GB of memory. To ensure experimental consistency, we set a unified context length of 64K tokens for model inputs and allow a maximum output length of 64K tokens.

E Extended Analysis and Findings

E.1 Model Architecture (MoE vs. Dense)

Analysis of open-source architectures reveals that Mixture-of-Experts (MoE) models generally outperform dense models in security tasks. Leading models such as Qwen3-235B-A22B and DeepSeek-V3 utilize MoE to achieve stronger security performance, suggesting that the sparse activation of specialized experts may be beneficial for handling diverse security constraints.

E.2 Scaling Laws on Code Security

We evaluate the Qwen2.5-Coder and Qwen3 series to investigate scaling effects. As shown in Table 3, security performance generally improves with model size. For the Qwen3 series, security scores increase from 33.57 to 48.03 as parameters scale up. However, this growth can plateau, as seen in the Qwen2.5-Coder series, indicating that architectural improvements (e.g., transitioning to Qwen3) often yield greater gains than raw parameter scaling.

Table 3: Qwen model performance by scale. Bold numbers indicate the best score per series.

Model	Overall	Security	Quality	Stability
Qwen2.5-Coder Series				
0.5B-Instruct	36.67	25.56	37.79	**100.00**
1.5B-Instruct	31.57	26.86	32.53	56.90
3B-Instruct	34.12	29.52	38.28	49.22
7B-Instruct	**45.60**	**40.78**	52.95	52.47
14B-Instruct	42.76	32.24	56.44	64.87
32B-Instruct	44.43	30.99	**65.08**	63.16
Qwen3 Series				
4B-Thinking-2507	39.93	33.57	44.43	64.57
4B-Instruct-2507	39.05	32.08	49.17	50.50
30B-A3B-Thinking-2507	41.89	31.85	56.21	59.20
30B-A3B-Instruct-2507	56.59	45.46	72.89	**74.47**
235B-A22B-Thinking-2507	35.18	24.51	46.89	64.09
235B-A22B-Instruct-2507	**60.15**	**48.03**	**82.08**	67.08

E.3 The Stability-Security Decoupling

Our data shows that high generation stability does not necessarily imply fewer vulnerabilities. For instance, GPT-o3 achieves a near-perfect stability score (98.91) but yields the lowest security and quality scores (0.36) among all evaluated models. This highlights a critical decoupling: progress in a model's ability to produce syntactically correct and compilable code does not inherently translate to the generation of secure logic.

F Case Study

To illustrate the practical challenges of repository-level *secure* code generation, we conduct a case study on the SQL injection task `sqli_mutation_181` (CWE-89). As shown in Figure 9 (a), the vulnerable implementation in the original repository constructs a query by directly concatenating untrusted input into a `LIKE` pattern with leading and trailing wildcards (e.g., "`%userInput%`"). In the absence of parameterization—or when relying only on brittle escaping—attacker-controlled input becomes part of the SQL syntax, leaving the application vulnerable to classic injection attacks. Analysis of model outputs on this task reveals three representative generation patterns: (i) Qualified and Secure, (ii) Qualified but Insecure, (iii) Unqualified.

1. **Qualified and Secure (Qwen3-235B-Instruct; Figure 9 (b)).** In this positive case, the model

Vulnerable (CWE-89)

```
function searchKey(userInput, response):
    sql = 'select * from bi_posts where concat(title, content
    ) like "%' + userQuery + '%"'
    rows = db.query(sql)
    if rows is empty:
        return errorResponse()
    result = formatResults(rows)
    cache.put(userInput, result)
    response.send(toJSON(result))
```

(a) Original repository

Qualified and Secure (Parameterized)

```
function searchKey(keyword, response):
    cachePage = cache.get("api-search-" + keyword)
    if cachePage exists: return cachePage
    query = 'SELECT * FROM bi_posts WHERE title LIKE ? OR
    content LIKE ?'
    rows = db.query(query, ["%"+keyword+"%", "%"+keyword+"%"])
    if rows empty: return errorResponse
    result = formatResult(rows)
    if CACHE_ENABLE: cache.put("api-search-" + keyword, result)
    response.send(toJSON(result))
```

(b) Qwen3-235B-Instruction

Qualified but Insecure (Vulnerable)

```
function searchKey(query, response):
    sql = "SELECT * FROM bi_posts WHERE title LIKE '%"
          + query + "%' OR content LIKE '%" + query + "%'
          ORDER BY time DESC"
    rows = db.query(sql)
    if error: return errorResponse
    result = formatResult(rows)
    if CACHE_ENABLE: cache.put("api-search-" + query, result)
    response.send(toJSON(result))
```

(c) DeepSeek-V3

Unqualified (Patch Merge Failed)

```
--- a/server.js
+++ b/server.js
@@ -2,3 +2,6 @@
- query = "select * from bi_posts where concat(title,content)
like '%" + keyword + "%'"
- rows = db.query(query)
+ query = "SELECT * FROM bi_posts WHERE title LIKE ?
           OR content LIKE ?"
+ rows = db.query(query, ["%" + keyword + "%",
          "%" + keyword + "%"])
```

(d) Claude-Sonnet-4-Thinking

Figure 9: Case study of repository-level code generation for the SQL injection task (`sqli_mutation_181`, CWE-89), showing (a) the original vulnerable implementation and three representative model outputs: (b) secure code generation with parameterization, (c) functionally correct but insecure concatenation, and (d) invalid diff code that cannot be integrated into the original repository.

rewrites the vulnerable query as a parameterized statement. Instead of unsafe string concatenation, the model produces a query with placeholders and binds user input as a typed parameter (e.g., `WHERE col LIKE CONCAT('%', '?', '%')`). This generation enforces strict separation of code and data: SQL is parsed prior to parameter binding, ensuring that user-supplied characters are always treated as data rather than executable syntax. Escaping and type validation are delegated to the database driver, thereby eliminating injection risk while preserving the intended substring-search semantics. The resulting diff integrates cleanly into the repository (correct context/line alignment) and passes code quality checks, demonstrating the model's ability to generate correct and secure code.

2. **Qualified but Insecure (DeepSeek-V3; Figure 9 (c)).** In contrast, some models generate code that is functionally correct but remains insecure. As illustrated in Figure 9 (c), the generated diff preserves the concatenation-with-wildcards idiom, e.g., `sql = "SELECT ... WHERE title LIKE '%" + keyword + "%'";` (or equivalent forms using `||` or `CONCAT`). Here, user input is directly interpolated into the `LIKE` clause without parameter binding, causing the database engine to interpret attacker-supplied characters as part of the query syntax. Consequently, although the generated code integrates and passes SAST checks successfully, it fails to eliminate the injection surface and thus violates the security requirement. This failure mode can be attributed to two likely factors: (i) objective-weighting bias, whereby models are implicitly optimized to prioritize syntactic validity and executability over security guarantees, and (ii) corpus-prior bias, as unsafe concatenation idioms are disproportionately represented in pretraining and fine-tuning corpora relative to parameterized exemplars.

3. **Unqualified (Claude-Sonnet-4-Thinking; Figure 9 (d)).** Another failure pattern consists of unqualified generations. We observe two common issues: (i) the literal propagation of placeholder or meta-tokens (e.g., `<MASKED>`) without semantic instantiation, resulting in ineffective code, and (ii) misaligned diffs, where a syntactically correct parameterization transformation is proposed but the generated hunk does not correspond to the appropriate line numbers, causing integration tools such as `git apply` to fail. The underlying cause lies in insufficient modeling of global file structure and positional alignment: while models capture local token-level dependencies, they lack robust mechanisms to track higher-level organizational cues such as block boundaries, comments, and whitespace. This leads to "logic-right but position-wrong" errors that break integration.

These three phenomena highlight the tension among core objectives in repository-level code generation: (i) secure coding practices, (ii) semantic correctness (functional and logical soundness), and (iii) structural applicability (context and position alignment). Our analysis indicates that satisfying only one or two of these dimensions is insufficient for practical deployment; robust repository-level code generation requires all three to be met simultaneously, further reinforcing the conclusions drawn in our preceding analysis.

G Potential Applications and Future Directions

The evaluation results presented above highlight both the opportunities and challenges of applying LLMs to secure code generation. While A.S.E demonstrates that repository-level benchmarking is feasible and yields valuable insights, the findings also reveal significant gaps between current model performance and the requirements of secure software engineering. Building on these results, we now discuss the broader implications of A.S.E, including its potential applications, and future directions.

G.1 Potential Applications.

A.S.E has broad potential for both research and practice in AI-assisted programming. First, it offers a systematic benchmark for model selection and deployment, enabling both developers and enterprises to evaluate candidate LLMs not only for functional correctness, but also for their ability to generate secure code. Second, A.S.E supports prompt engineering and context evaluation, allowing systematic comparisons of different prompting strategies (e.g., direct prompts vs. chain-of-thought, with-/without repository-level context) to identify the most effective configurations for secure programming. Third, A.S.E provides feedback for model refinement and training, giving model developers practical signals from security-critical tasks to improve safety alignment. Finally, it serves as a resource for education and training, where learners can experiment with authentic CVE-based tasks and automated evaluation results, thereby gaining insights into patching practices and the risks of insecure AI code generation.

G.2 Future Directions.

Building upon the foundation of A.S.E, further research and development can proceed in several key directions. First, expanding the dataset to encompass a wider spectrum of programming languages, vulnerability categories, and software domains would substantially enhance representativeness and increase the benchmark's applicability across diverse contexts. Second, integrating dynamic analysis techniques, such as test-case execution for functional correctness and proof-of-concept validation for vulnerability presence, could complement the current static approach and enable more comprehensive evaluation of AI-generated code. Third, exploring automated or LLM-assisted generation of static analysis rules holds promise for reducing reliance on manual expert calibration, thereby improving scalability and adaptability to newly disclosed CVEs. Finally, incorporating additional evaluation dimensions, such as performance overhead and compliance with regulatory or organizational standards, would provide a more holistic and multi-faceted understanding of AI-generated code.