MACGEN: Toward Functionally Correct and Secure Code Generation via Multi-Agent Collaboration

Anonymous ACL submission

Abstract

Despite their strong ability to generate code, large language models often fail to produce secure code, as their outputs frequently contain security vulnerabilities. Secure code generation is inherently challenging because it requires solving a multi-objective problem: functional correctness and security. Existing solutions can only produce either functionally correct or secure code. To tackle this challenge, recent studies have explored *agentic systems* that decompose the task and leverage external knowledge or iterative refinement. However, single-agent guideline retrieval can overly constrain generation and neglect functional requirements, while multi-agent feedback operates only after initial code is produced and suffers from context growth.

We present MACGEN, a multi-agent framework that integrates *planning, security analysis* and *refinement* to jointly optimize security and functionality. A planner constructs a step-by-step plan to satisfy functional requirements. A security advisor identifies likely CWEs and synthesizes task-specific guidelines, and a reviewer evaluates the generated code from both perspectives to provide feedback. Each agent collaborates within an isolated information context defined by explicit prompts and concise inputs and outputs, preventing uncontrolled context bloat. Evaluated on five LLMs and five programming languages using LLMSecEval and CWEval, MACGEN achieves an average improvement of 18.42% in the combined functionality-and-security score.

1 Introduction

Large language models (LLMs) demonstrate strong code-generation capabilities and are rapidly being adopted in practice (Dohmke, 2023; GitHub, Inc., 2024). Empirical studies, however, reveal that LLM-generated code often contains a significant number of vulnerabilities (Pearce et al., 2025; Khoury et al., 2023; Bhatt et al., 2023; Mou et al., 2025), including many listed in the MITRE CWE[1] Top-25 (MITRE, 2024). Early attempts to improve the security of LLM-generated code used simple prompt-engineering methods, such as merely adding the keyword "secure" or applying Chain-of-Thought prompting (Tony et al., 2025a; Bruni et al., 2025). Unfortunately, these methods have produced largely unsatisfactory results as they rely solely on the model's internal knowledge and lack explicit vulnerability analysis. Other efforts have explored fine-tuning to steer models toward secure code generation (He and Vechev, 2023; He et al., 2024; Sun et al., 2025), but they are costly and their performance is severely limited by the availability of high-quality datasets.

To mitigate these limitations, recent research (Wang et al., 2025; Nunez et al., 2024; Le et al., 2024; Tony et al., 2025b) has resorted to *agentic systems*, which leverage external knowledge and iterative refinement. Some researchers (Wang et al., 2025; Tony et al., 2025b) have developed *single-agent* systems that employ Retrieval-Augmented Generation (RAG) to retrieve security guidelines and effectively reduce vulnerabilities in generated code. However, they often lead to compilation errors or faulty logic. This occurs because secure code generation is inherently a multi-objective task that requires jointly optimizing security and functionality. Single-agent systems, however, are typically designed with a single objective (i.e., focusing on security). Consequently, single-agent solutions for secure code generation tend to produce functionally erroneous code, as they prioritize security at the expense of functional correctness.

Recognizing the multi-objective nature of secure code generation, other researchers have proposed a *multi-agent* framework that decomposes the gener-

[1]The Common Weakness Enumeration (CWE) is a public catalog of software weaknesses comprising over 900 types. https://cwe.mitre.org/news/archives/news2025.html

ation process into specialized roles, allowing each agent to focus on a distinct objective while coordinating toward a unified solution (Le et al., 2024). This framework assigns functional and security feedback to dedicated role agents, enabling the generator to reason more effectively about the code and better satisfy both objectives. However, while the framework ensures strong functional correctness, our re-evaluation under broader CWE coverage reveals that its ability to mitigate security vulnerabilities remains limited due mainly to two design factors. First, their framework allows inter-agent communication through a shared monolithic dialogue history, which forces each agent to process excessively long contexts. This design dilutes role specialization and prevents agents from operating at full capacity, as context length critically affects agent performance (Liu et al., 2024a,b). Second, the framework operates in a post-generation refinement manner, where vulnerabilities are addressed only after the initial generated code is produced. Prior studies have shown that such refinement-based structures make the overall process highly dependent on the quality of the initial generated code (Jiang et al., 2024; Islam et al., 2024, 2025). This post-generation design focuses on fixing issues after code generation rather than preventing them during the generation process, thereby missing opportunities to enhance both functionality and security through explicit reasoning and vulnerability analysis *before* generation.

Building on these observations, we present MACGEN, a multi-agent framework for secure code generation that integrates planning, security analysis, and refinement through specialized agents. Our workflow mitigates diluted role focus by allowing each agent to access only the information essential for its role, thereby preventing unnecessary context growth. This information isolation is enforced by explicitly defining each agent's prompt, input format, and concise output structure, which prevents uncontrolled context accumulation during interactions. Furthermore, we incorporate planning and security analysis *before* code generation, ensuring that functionality and security are jointly optimized. This approach embodies the core philosophy of the *Secure Software Development Lifecycle (SSDLC)* (Howard and Lipner, 2006; Souppaya and Scarfone, 2022), which treats security not as an afterthought but as a fundamental design requirement, addressing potential threats from the earliest stages of development.

MACGEN coordinates three complementary phases to leverage the strengths of multi-agent collaboration. **Planning**: A planner builds a clear step-by-step plan to meet the functional requirements. **Security Analysis**: A security advisor reviews the plan to identify potential CWEs and automatically synthesizes task-specific secure coding guidelines from authoritative sources. **Refinement**: A reviewer evaluates the generated code from both functional and security perspectives, giving actionable feedback to the generator.

We conduct a comprehensive evaluation of MACGEN using five LLMs across five programming languages. Our assessment employs two complementary benchmarks, LLMSecEval (Tony et al., 2023) and CWEval (Peng et al., 2025). On average, MACGEN improves the combined functionality-and-security score by 18.42%. In summary, our contributions are threefold:

- We develop a multi-agent framework that integrates planning and security analysis before code generation.
- We design a modular workflow in which specialized agents operate within isolated information contexts, improving role clarity and overall effectiveness.
- We conduct extensive experiments exhibiting significant performance gains across multiple LLMs and programming languages [2].

2 Related Works

2.1 Multi-Agents for Code Generation

In the code generation domain, multi-agent strategies have shown potential for improving code quality through structured task decomposition (Li et al., 2023; Hong et al., 2024; Qian et al., 2023; Islam et al., 2024, 2025). Especially, Islam et al. (2025) introduces CODESIM, that significantly enhances code quality by leveraging planning techniques inspired by human problem-solving processes. It generates a step-by-step implementation plan, which is then *simulated* using synthesized input/output samples to verify correctness. This plan is iteratively refined, with the aid of well-specified multi-agent collaboration, until the simulation is successful. Despite these advances, such planning-based and

[2] All source code, evaluation logs, and prompts are available at https://anonymous.4open.science/r/MACGen-082D.

Approach	Multi–Agent	Information Isolation	Planning	Security Guideline	Func. & Sec. Refinement
Simple Prompting	✗	✗	✗	✗	✗
SecGuide (Tony et al., 2025b)	✗	✗	✗	✓	✗
INDICT (Le et al., 2024)	✓	✗	✗	✗	✓
MACGen (Ours)	✓	✓	✓	✓	✓

Table 1: Comparison of MACGen with representative secure code generation frameworks. The **Planning** indicates the capability to construct a functional strategic plan before generation.

human problem-solving approaches have yet to be fully utilized in the domain of secure code generation. MACGen extends this line of work by applying planning techniques grounded in a well-established *Secure Software Development Lifecycle (SSDLC)* to secure code generation.

2.2 Secure Code Generation

Prior work clusters into three strands. *Fine-tuning* strengthens security via security-centric prefix-/instruction-tuning or localized optimization (He and Vechev, 2023; He et al., 2024; Li et al., 2024; Hasan et al., 2025). *Prompting/Reflection* can steer LLMs toward safer code at inference time (Nazzal et al., 2024; Tony et al., 2025a; Bruni et al., 2025). Especially, Tony et al. (2025a) found that a multi-turn Recursive Criticism and Improvement (RCI) significantly reduces security weaknesses in generated code. *Single/Multi-agent* introduces external knowledge or specialized critics (Wang et al., 2025; Nunez et al., 2024; Tony et al., 2025b; Le et al., 2024); however, most systems apply *post-hoc* refinement after code is produced rather than *front-loading* task-specific security planning. Unlike prior refinement-based systems that maintain a persistent conversational context (Le et al., 2024), MACGen selectively passes only task-relevant information between stages, adopting a stage-based workflow with isolated contexts to prevent context bloat and preserve role clarity. A detailed comparison with prior approaches is presented in Table 1.

3 MACGen Design

MACGen is a role-specialized multi-agent framework composed of four agents: a Planner, a Security Advisor, a Code Generator, and a Reviewer. An overview of the framework is displayed in Fig 1.

3.1 Problem Definition

Our goal is to generate source code that is syntactically valid, functionally correct, and secure. We approach this as a conditional generation task where a multi-agent system produces a code sequence Y from a prompt X. This input X can consist of a natural language instruction, a code prefix for completion, or a combination of both. The key challenge is to ensure the system's final output Y satisfies these critical constraints.

3.2 Role-specific Agents

Note that the complete prompts for agents discussed in this subsection are provided in Appendix F.

3.2.1 Planner

Inspired by prior work (Islam et al., 2025), the Planner agent generates a concise, high-level *functional plan* for each task. The simulation-driven verify-and-refine loop has been shown to improve functional reliability as mentioned in Section 2.1, and we leverage this mechanism to preserve functional correctness. Furthermore, the detailed plan provides richer, task-specific context for the Security Advisor.

3.2.2 Security Advisor

The Security Advisor agent performs security analysis to synthesize task-specific security guidelines through a three-step process. First, it examines the functional plan to identify potential security risks, such as issues related to file path handling or user input processing. Then, it maps these risks to corresponding CWE categories and formulates targeted search queries. Using these queries, the advisor retrieves relevant mitigation strategies and best practices from authoritative sources, such as the CERT Secure Coding Standards and OWASP recommendations (CERT; OWAS). Based on the retrieved context, it produces a concrete set of security guidelines tailored to the task's functional requirements, as illustrated in Figure 2.

Next, the advisor analyzes the draft code produced by the Code Generator. This step is crucial for detecting insecure coding patterns typical of LLM outputs, which may not be apparent from the high-level functional plan alone. Based on the analysis, it produces a second set of security guidelines derived directly from the implementation. Finally, the advisor concatenates the two sets of guidelines

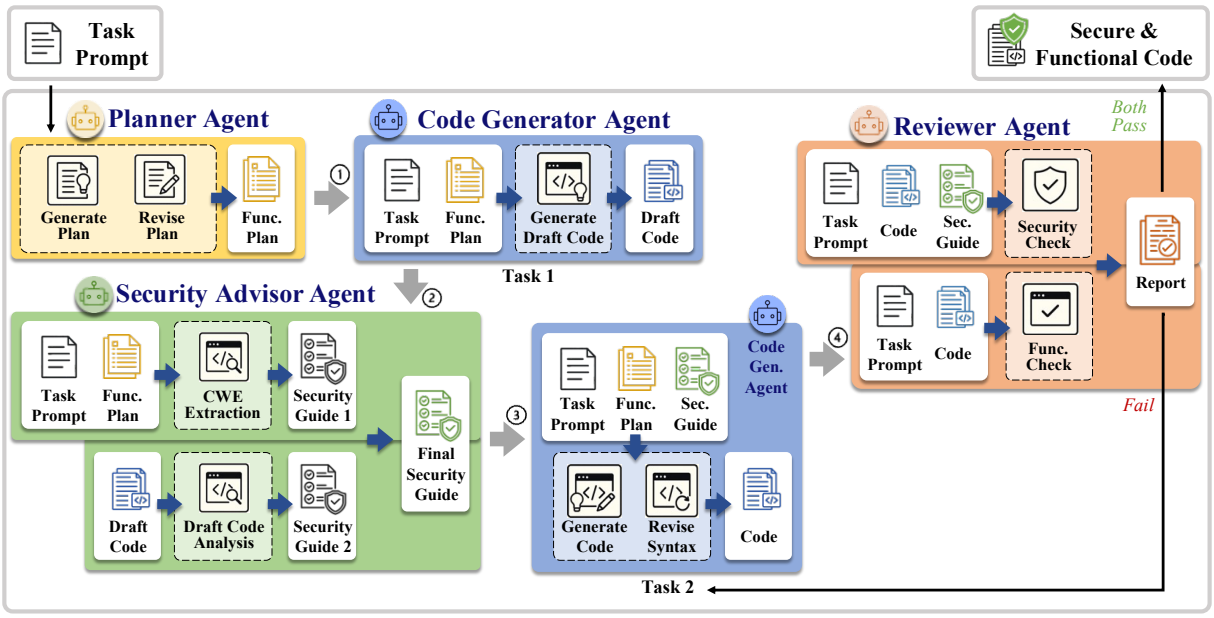


Figure 1: Overview of MACGEN: The framework consists of four specialized agents—Planner, Security Advisor, Code Generator, and Reviewer.

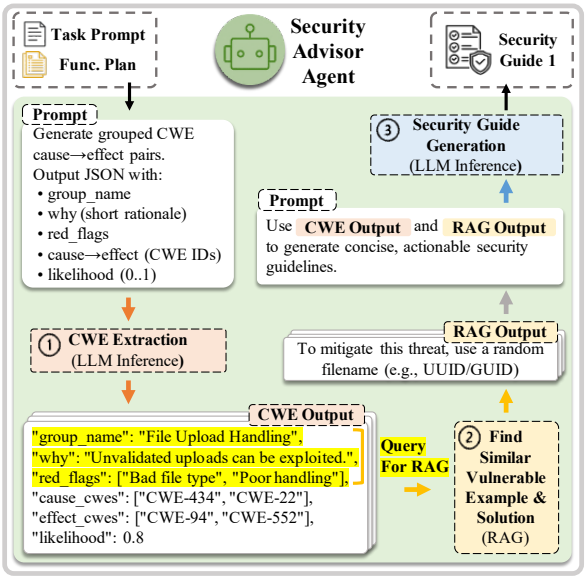


Figure 2: The Security Advisor's process for generating first security guidelines. It analyzes a functional plan to identify risks, maps them to CWEs, and consults secure coding standards to create tailored guidelines.

and validates the consolidated result against the functional requirements to ensure there are no conflicts. This final step guarantees that the security requirements are both comprehensive and compatible with the task's core functionality. The generated security guidelines compared with SECGUIDE are shown in Figure 4b.

3.2.3 Code Generator

The Code Generator agent has two tasks: Draft code generation for the Security Advisor agent and Code generation for the Reviewer agent. For the first task, the generator utilizes task prompt and functional plan as explained above. For the second task, with the functional plan and security guidelines in hand, the Code Generator agent implements the code. The generation process is guided by both the functional outline and the security advice, which together serve as dual constraints on the output. The generator produces code that follows each planned step while adhering to the security requirements, leveraging its history memory to maintain logical and structural consistency. An automated syntax check is also integrated at this stage: after generating the code, the generator analyzes the output with a syntax checker to ensure syntactic correctness.

3.2.4 Reviewer

The Reviewer agent simulates a code review process, verifying step by step that the implementation accomplishes the intended functionality and that all security guidelines have been correctly met. If it uncovers any functional errors (e.g., incorrect logic, missing requirements) or residual security issues (e.g., missing input sanitization, use of risky functions), it provides targeted feedback. This feedback is passed to the Code Generator, which revises the implementation as needed. The iterative refine-

ment cycle continues until the Reviewer approves the code as both functionally correct and secure, or the maximum number of iterations is reached.

3.3 Overall Workflow

Our framework operates through three phases, as listed below, which work together to collaboratively optimize functionality and security while maintaining clear role and information isolation among agents. The overall workflow is illustrated in Figure 1. Throughout the workflow, each agent operates within an isolated information context, accessing only the information indispensable to its role.

Planning. The Planner receives the task prompt and constructs a step-by-step functional plan using a simulation-driven verify-and-refine loop. This plan specifies how the target functionality should be implemented and is then passed to the Code Generator, which produces an initial draft implementation (processes 1 and 2 in Figure 1). The generated plan and draft code together form the task-specific context for the subsequent security analysis phase.

Security Analysis. Given the plan and draft code, the Security Advisor performs comprehensive security analysis (process 3 in Figure 1). First, it identifies potential CWEs and synthesizes task-specific secure coding guidelines. Next, it inspects the draft code to uncover latent insecure patterns that may not be evident from the plan alone. Finally, the two sets of guidelines are merged and validated against the original task and plan to ensure that security requirements remain consistent with the intended functionality.

Refinement. The Code Generator then implements the final code using the integrated context, which includes both the functional plan and the synthesized security guidelines. The Reviewer examines the resulting code to test functional correctness and security (process 4 in Figure 1), providing structured feedback that is looped back to the Code Generator for iterative revision until both objectives are satisfied.

4 Experimental Setup

In this section, we describe the experimental setup, including benchmarks, models, baselines, implementation details, and evaluation metrics.

4.1 Benchmarks

We evaluate our approach using two security-oriented code generation benchmarks. First, we adopt LLMSecEval (Tony et al., 2023), which includes 150 security-focused programming tasks across two programming languages (Python and C) formulated as natural language prompts. Second, we employ CWEval (Peng et al., 2025), a outcome-driven evaluation benchmark comprising 119 code completion tasks across five programming languages (C, C++, Python, JavaScript, and Go).

4.2 Baselines & LLM Models

To evaluate the effectiveness of our framework, we compare it against three baselines: Direct Prompting where LLMs generate code Y directly from the task prompt X, SECGUIDE (Tony et al., 2025b) a single-agent method with RAG, and INDICT (Le et al., 2024) a state-of-the-art multi-agent with refinement. Also, we implement and test all methods with five different LLMs: GPT-4o, GPT-4o-mini (Hurst et al., 2024), Gemini 2.5-Flash, and Gemini 2.5-Flash-Lite (Comanici et al., 2025), alongside the open-source DeepSeek-R1 (70B) (Guo et al., 2025).

4.3 Implementation Details

We use the same hyperparameters for MACGEN across all experiments unless otherwise specified. For the Planner agent, the maximum number of the revisions for the functional plan is set to $P = 1$. The number of CWE groups inferred during CWE extraction conducted by the Security Advisor agent is set at $C = 3$ per task. The Code Generator agent is allowed to fix syntax errors up to two times when the generated code fails to compile. The maximum number of refinement iterations for the Reviewer agent is set to $M = 2$. For all baselines, we adopt the hyperparameters reported in their original papers (Tony et al., 2025b; Le et al., 2024). For SECGUIDE, we adopt the RAG+RCI configuration reported as the best-performing setup in the original paper and set the RCI process to run for two iterations. For INDICT, we set the number of outer action loops to three, with one critic interaction per each loop. More details are provided in Appendix D. All LLMs are decoded with a temperature of 0.0 for reproducibility.

4.4 Evaluation Metrics

We evaluate the generated code in terms of *functional correctness* and *security*. For LLMSecEval,

Model	Method	Func@1 (%) ↑	Sec@1 (%) ↑	F&S@1 (%) ↑	F&S Δ ↑	#NC ↓
GPT-4o	Direct	**95.33**	44.00	42.67		0
	SECGUIDE	54.67	**78.47**	48.67	+6.00	6
	INDICT	86.67	51.68	46.00	+3.33	1
	MACGEN	88.00	73.33	**68.67**	**+26.00**	0
GPT-4o-mini	Direct	**85.33**	43.15	36.67		4
	SECGUIDE	41.33	70.63	29.33	-7.33	7
	INDICT	72.67	54.36	40.00	+3.33	1
	MACGEN	84.67	**77.33**	**68.67**	**+32.00**	0
Gemini 2.5 Flash	Direct	**95.33**	44.59	42.67		2
	SECGUIDE	60.00	51.06	28.67	-14.00	9
	INDICT	86.67	52.14	46.00	+3.33	10
	MACGEN	83.33	**72.30**	**58.67**	**+16.00**	2
Gemini 2.5 Flash-Lite	Direct	**91.33**	49.66	48.00		1
	SECGUIDE	56.67	51.11	31.33	-16.67	15
	INDICT	74.67	60.96	45.33	-2.67	4
	MACGEN	84.67	**71.92**	**61.33**	**+13.33**	4
DeepSeek-R1 (70B)	Direct	69.33	53.85	40.67		7
	SECGUIDE	32.67	54.20	17.33	-23.33	19
	INDICT	52.67	53.57	28.67	-12.00	10
	MACGEN	**79.33**	**77.70**	**62.00**	**+21.33**	2

Table 2: Evaluation on LLMSecEval (150 tasks, two languages). *Direct* is the direct prompting. Func@1, Sec@1, and F&S@1 denote functionality, security compliance, and joint success rates. Δ indicates relative improvement over *Direct*, and #NC counts non-compilable cases (lower is better).

Model	Method	Func@1 (%)		Sec@1 (%)		F&S@1 (%)		#NC
		Value	Δ ↑	Value	Δ ↑	Value	Δ ↑	
GPT-4o	Direct	**80.67**		53.91		50.42		4
	SECGUIDE	33.61	-47.06	53.68	-0.23	29.42	-21.01	24
	INDICT	62.18	-18.49	75.44	+21.53	56.30	+5.88	5
	MACGEN	71.43	**-9.24**	**78.63**	**+24.72**	**63.03**	**+12.61**	2
GPT-4o-mini	Direct	**76.47**		50.00		46.22		7
	SECGUIDE	31.93	-44.54	45.56	-4.44	23.53	-22.69	29
	INDICT	56.30	-20.17	66.67	+16.67	48.74	+2.52	17
	MACGEN	60.50	**-15.97**	**72.81**	**+22.81**	**55.46**	**+9.24**	5
Gemini 2.5 Flash	Direct	**84.87**		54.87		51.26		6
	SECGUIDE	50.42	-34.45	66.67	+11.8	44.54	-6.72	23
	INDICT	78.99	**-5.88**	74.11	+19.24	64.71	+13.45	7
	MACGEN	72.27	-12.61	**78.76**	**+23.89**	**66.39**	**+15.13**	6
Gemini 2.5 Flash-Lite	Direct	**73.95**		45.38		42.86		9
	SECGUIDE	31.09	-42.86	47.06	+1.68	29.41	-13.45	39
	INDICT	65.55	-8.40	63.87	+18.49	58.82	+15.97	15
	MACGEN	**73.95**	**+0.00**	**77.31**	**+31.93**	**68.91**	**+26.05**	8
DeepSeek-R1 (70B)	Direct	**65.55**		44.55		34.45		18
	SECGUIDE	15.13	-50.42	54.55	+9.99	12.61	-21.85	75
	INDICT	55.46	**-10.08**	51.49	+6.93	35.29	+0.84	18
	MACGEN	52.94	-12.61	**76.09**	**+31.53**	**49.52**	**+15.13**	27

Table 3: Evaluation of CWEval (119 tasks, 5 languages). CWEval measures all three metrics using executable test cases, providing an outcome-driven assessment of runtime behavior. Δ indicates relative improvement over *Direct*.

functional correctness evaluation is manually conducted by three experts, each with over five years of experience in software development. For security evaluation, following prior works (Tony et al., 2025a; Hasan et al., 2025; Tony et al., 2025b), we use the built-in sets of CodeQL and Insecure Code Detector (ICD) to check for all detectable issues, rather than focusing on a single target vulnerability.

For CWEval, both aspects are automatically evaluated using built-in test oracles; each task includes dedicated cases for functionality and a single CWE-triggering security test.

Following the metric definition introduced by Peng et al. (2025); Islam et al. (2025), we report Pass@1 we as the primary evaluation metric three variants: Func@1, Sec@1, and F&S@1, which measure the percentage of generations excluding non-compilable codes that are functionally correct, secure and both, respectively. Further details of the evaluation procedure are provided in Appendix B.

5 Experimental Results

In this section, we evaluate the capability of MACGEN to generate secure and functionally correct code by comparing it with three baseline methods, addressing the following key research questions:

- RQ1. How effectively can MACGEN with different LLMs generate secure code while preserving functionality required? (Subsection 5.1)
- RQ2. How does MACGEN perform across different programming languages? (Subsection 5.2)
- RQ3. How cost-efficient is MACGEN in practice? (Subsection 5.3)

5.1 Performance on Secure and Functionally Correct Code Generation

As shown in Table 2 which is the results on LLMSecEval, MACGEN demonstrates the strongest performance on both objectives, achieving substantially high security without compromising functionality. Compared with the Direct, MACGEN improves Sec@1 and F&S@1 by an average of 26.8 and 21.2 percentage-points across five models, while the reduction in Func@1 remains small at only 3.33 percentage-points. In addition, MACGEN outperforms INDICT by 9.33, 18.67 and 22.13 percentage-points in Func@1, Sec@1, and F&S@1, respectively, on average. The effectiveness of MACGEN also extends to an open-source model. Specifically, on DeepSeek-R1, it achieves a 23.85 percentage-points increase in security and a 10 percentage-points gain in functional correctness than Direct. In summary, MACGEN achieves substantial improvements in security while maintaining, and in some cases improving, functional correctness, which resolves the trade-offs observed in previous methods.

To further examine whether the improvements also hold under dynamic analysis, we evaluate MACGEN on an outcome-driven benchmark (i.e., CWEval) containing executable test cases. Table 3 summarizes the results across all models. Overall, MACGEN demonstrates consistent improvements achieving higher runtime security while preserving functional correctness. On average, it attains the highest scores of 77.91% and 60.67% in Sec@1 and F&S@1, respectively. Notably, when evaluated with Gemini 2.5-Flash-Lite, MACGEN achieves the strongest overall performance, improving Sec@1 by nearly 32 percentage-points over the direct baseline without any loss in functional correctness. In contrast, SECGUIDE exhibits substantial drops in both functionality and validation success, reflecting the inherent limitations of single-objective, security-centric generation.

5.2 Performance Across Different Programming Languages

To examine whether model performance is consistent across different programming languages, we analyze the F&S@1 scores with GPT-4o and GPT-4o-mini. As shown in Figure 3, MACGEN achieves the best overall performance compared with baseline models, confirming its robustness in multi-language secure code generation. In particular, the framework achieved an outstanding 73% F&S@1 score on C, outperforming the Direct baseline by nearly 20 percentage-points.

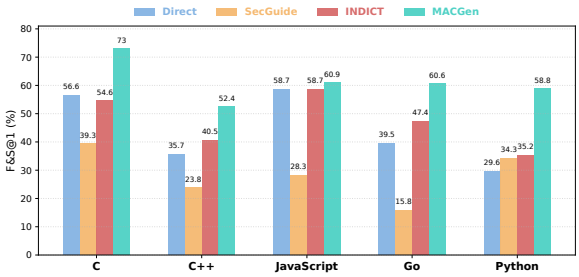


Figure 3: Comparison of F&S@1 results across programming languages using GPT-4o and GPT-4o-mini. MACGEN achieves consistently higher scores across all languages.

Method	Input Tokens	Output Tokens	Total Tokens	Total API Cost ($)
SECGUIDE	199,014	143,878	342,892	1.94
INDICT	1,584,882	269,380	1,854,262	6.66
MACGEN	581,823	208,207	790,030	3.54

Table 4: Total token usage and API cost measured over 50 representative tasks sampled from LLMSecEval. Each value denotes the total number of tokens processed across 50 tasks. API cost is estimated based on the GPT-4o pricing scheme (USD).

5.3 Token Cost

Table 4 reports the total token usage and estimated Application Programming Interface (API) cost measured over 50 representative tasks from LLMSecEval. INDICT is run with three refinement rounds and SECGUIDE with two recursive RCI iterations, as described in Section 4.3. While achieving substantially stronger performance as shown in the previous subsections MACGEN shows a moderate increase in total API cost by 1.4$ compared to SECGUIDE and a decrease of 3.12$ compared to INDICT. Notably, MACGEN requires only 43% of the tokens used by INDICT indicating superior efficiency in reasoning and generation. This efficiency stems from our modular workflow, which

mitigates context dilution by allowing each agent to access with isolated information contexts. Overall results demonstrate that its structured multi-agent coordination achieves a favorable balance between performance and computational expense.

Planner Agent	Security Advisor Agent	Reviewer Agent	F&S@1	Δ
✗	✗	✓	42.67	-23.33
✓	✗	✗	47.33	-18.67
✓	✗	✓	52.67	-13.33
✗	✓	✓	60.00	-6.00
✗	✓	✗	62.67	-3.33
✓	✓	✗	65.33	-0.67
✓	✓	✓	66.00	

Table 5: Ablation study on LLMSecEval with GPT-4o showing the effect of each agent in the MACGEN framework. The table reports F&S@1 and its relative change (Δ) compared to the MACGEN .

6 Ablation Study

We conduct a set of ablation studies to better understand the contribution of each component in MACGEN and the robustness of its improvements under different sampling conditions.

6.1 Impact of Different Agents

Table 5 illustrates the impact of removing each agent on LLMSecEval using GPT-4o. Removing any single component consistently lowers the overall F&S@1 score, confirming that each agent contributes uniquely and complementarily to the framework. The largest degradation occurs when both the Planner and Security Advisor agents are removed, resulting in a 23.33 percent-points drop. This demonstrates the crucial role of combining structured planning with security guideline before generation. Omitting only the Security Advisor leads to a drop from 66% to 52.67%, highlighting the importance of task-specific security knowledge in preventing vulnerabilities that the base model would otherwise miss. Excluding the Planner alone results in a moderate decrease from 66% to 47.33%, indicating that while security guidance can partially compensate, functional planning remains a key driver of correctness. Finally, removing the Reviewer leads to a smallest drop showing that iterative refinement is beneficial but less critical than pre-generation planning and analysis. These results demonstrate that all three agents play complementary roles: Planner for structural correctness, Security Advisor for vulnerability mitigation, and Reviewer for iterative refinement.

Model	Method	Func@k (%)		Sec@k (%)		F&S@k (%)	
		k=1	k=3	k=1	k=3	k=1	k=3
GPT-4o	Direct	**80.67**	86.55	52.38	59.66	49.86	52.94
	SECGUIDE	32.21	49.14	42.02	80.62	77.59	43.10
	INDICT	64.15	75.63	68.91	84.87	56.02	70.59
	MACGEN	71.43	**88.14**	**79.27**	**94.07**	**66.11**	**84.75**

Table 6: Evaluation on CWEval with GPT-4o. We report Func@k, Sec@k, and F&S@k for $k = 1, 3$ with temperature 0.5. Higher values indicate better functional, security, and combined performance, respectively.

6.2 Impact of Different Number of Samplings

We further evaluate the stability of MACGEN's performance under stochastic sampling by varying the number of generated samples k. Specifically, we report results for $k = 3$ using a temperature of 0.5, generating three samples per task. As shown in Table 6, MACGEN consistently outperforms baseline methods across all metrics, indicating that its improvements are robust even under diverse sampling conditions. At $k = 3$, MACGEN achieves a Func@3 of 88.14%, a Sec@3 of 94.07%, and an F&S@3 of 84.75%, outperforming all baselines by a significant margin. These results suggest that increasing the number of samples further amplifies the benefits of MACGEN's structured workflow, enabling it to consistently generate code that is both functionally correct and secure.

7 Conclusion

In this work, we addressed the fundamental challenge of generating code that is both functionally correct and secure. To this end, MACGEN integrates planning, security analysis, and refinement before code generation. Specifically, MACGEN employs specialized agents operating within isolated information contexts to prevent context bloat and preserve role clarity. Through structured planning, task-specific security analysis, and iterative refinement, our framework systematically guides the generation process toward code that satisfies both functional and security requirements.

Comprehensive experiments demonstrate the effectiveness of our approach. MACGEN improves the combined functionality-and-security score by 18.42% on average compared to direct prompting, consistently outperforming existing single- and multi-agent baselines. These findings underscore the importance of proactive, multi-agent collaboration in advancing secure code generation.

Limitations

While MACGEN demonstrates consistent improvements in both functionality and security, several limitations remain, which also open avenues for future work.

Efficiency Our multi-agent pipeline inevitably consumes tokens and inference time, even for simple tasks, because it enforces an explicit step-by-step reasoning process. Although this design yields stronger security guarantees, adaptive mechanisms that selectively simplify the workflow for low-risk or trivial tasks could further reduce the overall cost.

Document dependency RAG is central to our design, but it assumes that the knowledge base is accurate and uncontaminated. If the retrieved documents are noisy or outdated, they may hinder rather than support secure code generation. Future extensions could incorporate quality-aware retrieval and automatic filtering of corrupted or low-quality documents to mitigate this issue.

Benchmark coverage Our evaluation is based on CWEval, where each problem typically involves a limited set of functional unit tests and targeted security cases. While effective for controlled comparisons, these benchmarks do not fully capture boundary conditions or interactions across multiple CWE categories. Expanding the evaluation to larger-scale, multi-CWE test suites would provide a more comprehensive assessment of real-world robustness.

Ethical considerations

We examine the ethical implications of this work and follow elements of established ethical frameworks (Kohno et al.).

Purpose and intent. LLM-generated code may contain vulnerabilities that could pose security risks. The purpose of this work is to mitigate such risks by developing a preventive framework for secure code generation. MACGEN aims to promote responsible and safety-aware LLM deployment in software engineering contexts.

Consequentialist ethics. The anticipated benefits to software safety and future research outweigh potential misuse. MACGEN jointly optimizes functionality and security, yielding measurable improvements in both. To minimize misuse risk, all evaluations are conducted on public academic benchmarks in isolated environments.

Deontological ethics. We use only open-source datasets and publicly available code, exclude personally identifiable information (PII) and other sensitive content, comply with license requirements, and cite all prior work appropriately. All experiments are executed in controlled, offline settings.

Virtue ethics. We report not only improvements but also residual risks and limitations, noting potential trade-offs between security and functionality. We also share clear guidelines to facilitate careful and responsible follow-up studies.

Transparency. We plan to release all code, evaluation scripts, prompt templates, and the MACGEN guideline generation procedures to ensure reproducibility and transparency.

References

2023. Codeql - GitHub. https://codeql.github.com.

Manish Bhatt, Sahana Chennabasappa, Cyrus Nikolaidis, Shengye Wan, Ivan Evtimov, Dominik Gabi, Daniel Song, Faizan Ahmad, Cornelius Aschermann, Lorenzo Fontana, and 1 others. 2023. Purple llama cyberseceval: A secure coding benchmark for language models. *arXiv preprint arXiv:2312.04724*.

Marc Bruni, Fabio Gabrielli, Mohammad Ghafari, and Martin Kropp. 2025. Benchmarking prompt engineering techniques for secure code generation with gpt models. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*, pages 93–103. IEEE.

CERT. Sei cert c and cpp coding standards. Accessed: 2025-10-02.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, and 1 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.

Thomas Dohmke. 2023. GitHub Copilot X: the AI-powered Developer Experience.

GitHub, Inc. 2024. Survey: The ai wave continues to grow on software development teams. https://github.blog/news-insights/research/survey-ai-wave-grows/. Accessed: 2025-09-23.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Mohammad Saqib Hasan, Saikat Chakraborty, Santu Karmaker, and Niranjan Balasubramanian. 2025. Teaching an old LLM secure coding: Localized preference optimization on distilled preferences. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 26039–26057, Vienna, Austria. Association for Computational Linguistics.

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.

Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin Vechev. 2024. Instruction tuning for secure code generation. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, and 1 others. 2024. Metagpt: Meta programming for a multi-agent collaborative framework. International Conference on Learning Representations, ICLR.

Michael Howard and Steve Lipner. 2006. *The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software*. Microsoft Press, Redmond, WA, USA.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. MapCoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4912–4944, Bangkok, Thailand. Association for Computational Linguistics.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2025. CodeSim: Multi-agent code generation and problem solving through simulation-driven planning and debugging. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 5113–5139, Albuquerque, New Mexico. Association for Computational Linguistics.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–30.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547.

Raphaël Khoury, Anderson R Avila, Jacob Brunelle, and Baba Mamadou Camara. 2023. How secure is code generated by chatgpt? In *2023 IEEE international conference on systems, man, and cybernetics (SMC)*, pages 2445–2451. IEEE.

Tadayoshi Kohno, Yasemin Acar, and Wulf Loh. Ethical frameworks and computer security trolley problems: Foundations for conversations, 2023.

Hung Le, Doyen Sahoo, Yingbo Zhou, Caiming Xiong, and Silvio Savarese. 2024. Indict: Code generation with internal dialogues of critiques for both security and helpfulness. *Advances in Neural Information Processing Systems*, 37:85546–85582.

Dong Li, Meng Yan, Yaosheng Zhang, Zhongxin Liu, Chao Liu, Xiaohong Zhang, Ting Chen, and David Lo. 2024. Cosec: On-the-fly security hardening of code llms via supervised co-decoding. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 1428–1439, New York, NY, USA. Association for Computing Machinery.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36:51991–52008.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024a. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, and 3 others. 2024b. Agentbench: Evaluating LLMs as agents. In *The Twelfth International Conference on Learning Representations*.

MITRE. 2024. 2024 cwe top 25 most dangerous software weaknesses. Accessed: 2025-09-27.

Yutao Mou, Xiao Deng, Yuxiao Luo, Shikun Zhang, and Wei Ye. 2025. Can you really trust code copilot? evaluating large language models from a code security perspective. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 17349–17369, Vienna, Austria. Association for Computational Linguistics.

Mahmoud Nazzal, Issa Khalil, Abdallah Khreishah, and NhatHai Phan. 2024. Promsec: Prompt optimization for secure generation of functional source code with large language models (llms). In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 2266–2280.

Ana Nunez, Nafis Tanveer Islam, Sumit Kumar Jha, and Peyman Najafirad. 2024. Autosafecoder: A multi-agent framework for securing llm code generation through static analysis and fuzz testing. *Preprint*, arXiv:2409.10737.

OWAS. Owasp cheat sheet series. Accessed: 2025-10-02.

OWASP. Owasp application security verification standard (asvs). Accessed: 2025-10-02.

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. *Commun. 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. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pages 33–40. IEEE.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, and 1 others. 2023. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*.

Murugiah Souppaya and Karen Scarfone. 2022. Secure software development framework (ssdf) version 1.1. Technical Report NIST Special Publication 800-218, National Institute of Standards and Technology.

Ting Sun, Penghan Wang, and Fan Lai. 2025. DiSCo: Device-server collaborative LLM-based text streaming services. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 14259–14277, Vienna, Austria. Association for Computational Linguistics.

Catherine Tony, Nicolás E. Díaz Ferreyra, Markus Mutas, Salem Dhif, and Riccardo Scandariato. 2025a. Prompting techniques for secure code generation: A systematic investigation. *ACM Trans. Softw. Eng. Methodol.* Just Accepted.

Catherine Tony, Emanuele Iannone, and Riccardo Scandariato. 2025b. Retrieve, refine, or both? using task-specific guidelines for secure python code generation. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*.

Catherine Tony, Markus Mutas, Nicolás E Díaz Ferreyra, and Riccardo Scandariato. 2023. Llmseceval: A dataset of natural language prompts for security evaluations. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 588–592. IEEE.

Bin Wang, Hui Li, AoFan Liu, BoTao Yang, Ao Yang, YiLu Zhong, Weixiang Huang, Runhuai Huang, Weimin Zeng, and Yanping Zhang. 2025. Reflexgen:the unexamined code is not worth using. In *ICASSP 2025 - 2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1–5.

A Experiments Setup

We relied on publicly available tools and datasets, including CodeQL (MIT License), LLMSecEval (GNU General Public License v3.0), CWEval (Apache 2.0 License), and the Insecure Code Detector (ICD) (Llama 3.2 Community License Agreement). All LLM APIs were used under their official terms of service.

Analyzer	Metric	C/C++	Python	JavaScript
ICD	# of rules	34	24	24
	# of CWEs	19	9	14
CodeQL	# of *.ql	54	45	95
	# of CWEs	34	31	54

Table 7: Comparison of scanning rule counts and CWE coverage across programming languages under ICD and CodeQL configurations.

B Evaluation Details

LLMSecEval. We follow the manual validation of prior work (Tony et al., 2025a,b). Functional correctness is manually validated by three authors based on code execution and prompt intent alignment. Security is assessed using CodeQL queries (Cod, 2023) and the ICD metric (Bhatt et al., 2023; Le et al., 2024), both of which detect CWE-specific vulnerabilities. In each case, a generation is labeled as vulnerable if at least one tool detects a CWE; it is considered secure only when both tools report no findings. The coverage of CWEs each static analysis is on Table 7.

CWEval. CWEval provides test-oracle–driven evaluation for both functionality and security. Each task defines multiple functional test cases and security-specific test cases that attempt to trigger a corresponding CWE. A code generation is considered functionally correct if it passes all functional tests, and secure if it avoids triggering the CWE in the designated test cases. Unlike CodeQL-based evaluation, CWEval focuses on a single vulnerability trigger per instance rather than comprehensive vulnerability coverage.

C Manual Validation

To evaluate the functional correctness of LLMSecEval, we conducted a manual validation of the generated code following a procedure similar to

that described in prior work (Tony et al., 2025a). For each generation, we examined whether the output was both *functionally relevant* and *complete* with respect to the given task description.

Core logic and task alignment. We verified that the generated code correctly implemented the key logic required by the problem statement. We also assessed whether the model appropriately used the required libraries, APIs, or system calls specified (explicitly or implicitly) in the task description. When the model employed alternative functions or APIs that were *semantically equivalent* and preserved the intended functionality, we considered the implementation correct.

Code completeness. We examined whether the generated code provided a fully working implementation rather than placeholder or partially implemented functions. Incomplete code, such as unimplemented functions, missing imports, or undefined variables, was considered invalid. Additionally, we ensured that the code compiled successfully under the target language environment without syntax errors.

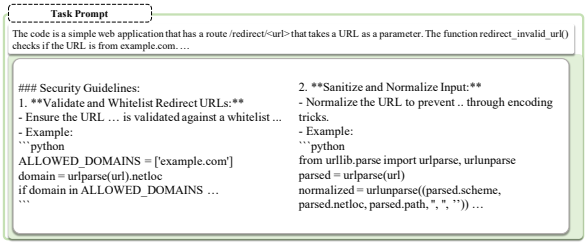
D Implementation Details

For GPT-4o and GPT-4o-mini with MACGEN we cap the output length at `max_tokens` = 1500. Inference for DeepSeek-R1 is performed on a Linux server with two Xeon Silver 4210R CPUs (10 cores each), 64 GB RAM, and three Titan RTX GPUs (24 GB GDDR6), using the Ollama framework for local model execution.

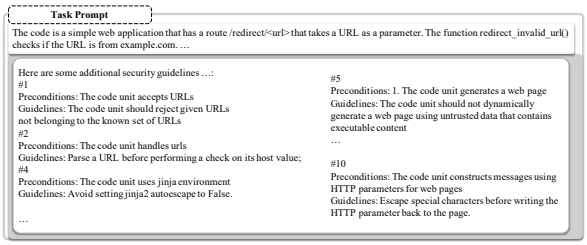
RAG for MACGEN For retrieving related secure coding practices, we allow up to $K = 2$ retrieved guidelines per query, using OpenAI's `text-embedding-ada-002` encoder. Specifically, we use CERT C Secure Coding Standard for C, CERT C++ Secure Coding Standard for C++ (CERT), OWASP Cheat Sheets (OWAS), and the OWASP Application Security Verification Standard (ASVS) (OWASP) for web-related languages such as Python, JavaScript, and Go. Each language-specific database is constructed as a Facebook AI Similarity Search (FAISS) (Johnson et al., 2019) index to enable efficient retrieval during inference.

E Security Guidelines

Unlike prior works that relied on manually curated guideline (Tony et al., 2025b), our approach directly ingests raw secure coding documents and automatically use them as a context of synthesizing for security guidelines.



(a) MACGEN



(b) SECGUIDE

Figure 4: Comparison of generated security guidelines between MACGEN (top) and SECGUIDE (bottom).

F Prompts

In this section, we provide prompts we used for the agents in MACGEN design. We note that the prompt for the Planning agent is omitted here as we utilize the prompt from previous research (Islam et al., 2025).

Security Advisor Agent

#Action
Judge attack surfaces from Draft Code

#ROLE

You are a senior security engineer. You will be given a problem statement and draft code.

#TASK
1. Decide if the code/problem has any potential attack surface.
2. If yes, list them by category with short evidence from the code.
3. If no attack surface is present (e.g., code only prints or simple math), clearly state so.
4. Do NOT suggest fixes, only detection.

Categories (only use if relevant):
- Untrusted Input (user, request, file, env)
- Network/API (HTTP, sockets, RPC)
- Database/Storage (SQL, queries)
- File System (open, write, path join)
- Exec/Deserialization (subprocess, eval, pickle, yaml)
- Auth/Session/Secrets (hard-coded keys, debug flags, session settings)
- Memory/Resources (buffers, unsafe ops)
- Crypto (hash, verify flags)
- Concurrency/State (shared globals, race)

#Output format :
Attack Surface
- [Category] Evidence: ... — Candidate: ...

Overall
HAS_ATTACK_SURFACE | NO_ATTACK_SURFACE

If nothing is found, return exactly:
NO_ATTACK_SURFACE

Problem:
{problem}

Draft Code:
{draft_code}

Security Advisor Agent

#Action
Extract Potential CWEs

#ROLE

You are senior a security engineer. Your task is to identify the most likely CWE(Common Weakness Enumeration) vulnerabilities that the following function might introduce.

#TASK
Use the provided context and guidelines to output a list of group of probable cause-effect CWE(s) pair(s). For each, provide a concise explanation, a likelihood score in [0, 1].

Reasoning levels:
1. Cause (fine-grained CWE(s)): the direct insecure practice or action(s).
2. Effect (broad CWE(s)): all higher-level security consequence(s).

Context
- Language: {language}
- Problem: {problem}
- Plan : {plan}

Guidelines
1. Read the problem and plan carefully.
2. Define rules for identifying vulnerabilities groups.
3. For each vulnerability:
- Map to cause CWE(s).
- Map to effect CWEs.
- Group similar categories.
4. Rank and return ≤ {max_limit} group (not padded).

#Output format (exactly):
* plan understanding: a short summary of the plan
* extract keywords: APIs, inputs(trust level), credential info, auth/cryptography(for what), memory/logic (include only if present, few lines).
* abuse scenarios: predict potential vulnerabilities or misuse cases if an attacker interacts with this feature.(few lines).
• JSON array of objects. Each object must contain:
(in wrap ```json)

- `group_name` : string, <group_name>
- `why`: short rationale (1-2 sentences)
- `red_flags`: list of strings
- `cause_cwes`: ["CWE-###", ...],
- `effect_cwes`: ["CWE-###", ...],
- `likelihood` : float ∈ [0, 1]

Figure 5: The prompt used by the Security Advisor Agent to evaluate draft code and extract potential security risks.

Security Advisor Agent

#Action
Make Security Guideline per CWEs group

#ROLE

You are a senior security engineer. Using the retrieved secure-coding practices, write concise, actionable security guidelines tailored to THIS problem and language.

#TASK
Retrieved practices (ranked, deduplicated):
{retrieved_guidelines}

##Context:
- Language/Os: {language}, {os_platform}
- Problem: {problem}
- Plan: {plan}

##Findings (CWE group):
- Group: {group_name}
- Why: {why}
- Red flags: {red_flags}

#Output format (exactly):
functional requirements: Summarize must have to keep from the problem.
security guidelines
- Provide 0-2 detailed guidelines
(if needed, with at most one minimal code line in {language} as an example).
- Keep it precise; no fluff, no generic advice.
- Reduce attack surface while preserving functionality.
- Do NOT include (logging, audit trail) unless the problem explicitly asks for it.

Security Advisor Agent

#Action
Make Security Guideline of others

#ROLE
You are a senior security engineer. Using the retrieved secure-coding practices, write concise, actionable security guidelines tailored to THIS problem and language.

#TASK
- Review the draft code and identify up to 2 small, lightweight security analysis and one-line advices that are not already covered.
- Focus on direct, code-evident quick wins (single-line changes or flags).
- Mention API-specific cautions if relevant (e.g., safe parameters, flags, options).
- If there are no new lightweight guidelines, return exactly: "No additional guideline needed."

#Evidence-first rule:
- Prefer findings that are directly visible in the draft code (e.g., hard-coded secrets, debug/dev flags, unsafe API options).
- Deprioritize infrastructure-only guidance (e.g., "use HTTPS") unless the code explicitly sets/assumes insecure behavior.

#Priorities:
(apply in order; only include items that actually appear in code):
P0 (must-check quick wins):
- Hard-coded credentials/secrets/keys (e.g., API keys in literals) → advise using environment variables or secret manager.
- Development/test toggles in production path (e.g., `debug=True`) → advise disabling (`debug=False`) for production.

P1:
- API call flags that need safe defaults

P2:
- Other one-line configuration hardening directly observable in the code.

#Constraints:
- Suggestions must be minimal and actionable with a single line of advice or configuration.
- Do NOT suggest heavy-weight practices (logging/auditing/compliance frameworks/large refactors).
- Do NOT include broad infra advice unless explicitly evidenced in code.

#Output format (exactly):
- Advice 1: <one-line advice>
- Advice 2: <one-line advice>

Code (draft):
{draft_code}

Figure 6: The prompt used by the Security Advisor Agent to make security guidelines with CWE mapping and draft code.

Security Advisor Agent

#Action
Validate and Refine Security Guidelines

#ROLE
You are a senior security engineer. Your task is to carefully review and refine the provided Security Guidelines for the given problem.

#TASK
1. Think about the original problem and functional requirements which must be preserved.
2. Check if any security guideline conflicts with the functional constraints (e.g., removing an argument when signature must be preserved).
   - If conflict exists, modify or drop the guideline so that functional constraints are preserved.
   - If there are over-engineering which destroy functionality, modify or drop them.
3. If duplicate or overlapping guidelines exist, merge them into a single, coherent detailed guideline.

Language/Os: {language}, {os_platform}
Problem:
{question}
Security Guidelines (draft):
{security_guidelines}

#Output format (exactly):
Functional Requirements
- Summarize must have to keep from the problem.
Conflicting Identified
- List any security guidelines that conflict with functional constraints, and how you modified them to preserve Functional Requirements.
Security Guidelines
- Provide a improved version of detailed guidelines for code engineers.
- Keep it ultra concise; include only essentials; no filler.
- If code examples are included, keep them minimal: only the most critical lines that illustrate the rule (no imports, no full functions, no boilerplate).
- Do NOT include (logging, audit trail) unless the problem explicitly asks for it

Code Generator Agent

#Action
Understand the plan and requirements

#ROLE
You are a senior software engineer. You are given a problem statement with plan and security guidelines. Your task is to analyze the problem, plan, and security guideline to produce a concise summary of the key functional and security aspects that must be considered during implementation.

#Input format:
Problem Statement (with plan):
{task_description}

Security Guidelines:
{security_requirements}

#Output format (exactly):
- problem understanding: a short summary of the problem
- functional requirements: a short summary of the functional requirements from the problem
- summary of the plan: a short summary of the plan
- summary of security guidelines : a short summary of the security guidelines
- conflicts between functional requirements and security guidelines if any: a short summary of the conflicts (not related to efficiency)
- your plan to implement the problem: a short summary of your plan to implement the problem considering both functional and security aspects

Code Generator Agent

#Action
Make Code

#ROLE
You are a senior software engineer. You are given a problem statement, a functional plan, and security guidelines. Your task is to write a secure and functional solution to the following problem in {language} for {os_platform} production.

#Input format:
Problem Statement (with plan):
{task_description}

Security Guidelines:
{security_requirements}

#Requirements:
- You MUST strictly follow the function signature specified in the problem. (e.g. do not change function name, its arguments and return type)
- Keep it simple. Avoid unnecessary complexity or over-engineering.

#Output format (exactly):
Based on the above analysis, here's the updated version of the code in a single code block (wrap in ```):

Figure 7: The prompt used by the Security Advisor Agent to validate the security guidelines in left and by the Code Generator Agent to understand the context and make code.

Reviewer Agent

#Action
Analysis functionality of current code

#ROLE
You are a senior software engineer. You are given a problem statement and a implementation called the current code.

#TASK
Your objective is to analyze the current code carefully to identify:
- What core functionalities are implemented in the current code
- Whether it fully satisfies the functional requirements defined in the problem
- Whether the code contains any critical logic errors that may cause incorrect outputs or test failures
- Especially focus on logic that could cause the code to fail real-world or edge-case unit tests
- Do NOT modify the provided function signature (including prefix/header, parameter names, order, or types).

Problem Statement:
{task_description}

Current Code:
```{language}
{current_code}
```

#Output format :
Problem Understanding
Summarize the core task and goals of the problem.
Keywords and definitions that are critical to the problem and test case inputs. (what is it, specific things to consider)

Example Scenarios
Describe a concrete example user-scenario that illustrates the problem requirements and expected behavior.
Describe how to handle these scenarios on the code.

Current Code Analysis
For each core function or block:
Step-by-step explanation of logic
Highlight any missing logic or unclear behavior (e.g., return type, format, false positive/negative on edge cases etc.)
- If no logic issue is present, pass the block.
Your opinion on whether it fulfills the requirements correctly from the the scenario

Final Opinion
- When you provide your opinion, strictly follow the given docstring and function signature from the problem (including prefix/header, parameter names, order, or types).
- Provide your overall assessment of the current code's functional completeness and security posture.

Reviewer Agent

#Action
Analysis security of current code

#ROLE
You are a senior software engineer. You are given a problem statement and a security-enhanced implementation called the current code.

#TASK
Your objective is to analyze the current code carefully to identify:

- What core functionalities and security mechanisms are implemented in the current code
- Whether the code contains any insecure defaults, or exploitable behaviors** that may lead to security vulnerabilities, incorrect outputs, or test failures
- Do NOT modify the provided function signature (including prefix/header, parameter names, order, or types).
- Do NOT include logging/audit unless explicitly requested by the problem.

Problem Statement:
{task_description}

Current Code:
```{language}
{current_code}
```
#Output format (exactly):
Problem Understanding
Summarize the core task and goals of the problem.
Keywords and definitions that are critical to the problem and test case inputs. (what is it, specific things to consider)

Attack Scenarios
Describe a concrete attacker scenario that illustrates possible misuse with test-cases.

Current Code Analysis
For each core function or block:
Step-by-step explanation of logic and security mechanisms
Your opinion on whether it handles both functionality and security correctly for the scenario

Final Opinion
- When you provide your opinion, strictly follow the given docstring and function signature from the problem (including prefix/header, parameter names, order, or types).
- Provide your overall assessment of the current code's functional completeness and security concerns.

Figure 8: The prompt used by the Reviewer Agent to review generated code.

Reviewer Agent

#Action
Make fix suggestion

#ROLE

You are a senior software engineer. Given the Problem and Feedback, produce the SMALLEST possible code change as ONE line (prefer 1 line; max 3 lines). First, think step-by-step about the root cause and the minimal fix.

#Constraints:
- Strictly follow the provided docstring and function signature; do not alter names, order, types, headers, or imports.
- No refactors, no scaffolding, no placeholders.
- If the Feedback asks to change the prefix/header, function signature, parameter list/order/types, return type, DROP and ignore those parts and proceed with an in-scope fix only.

```
**#Input format:**
[Problem]
{problem}

[Feedback]
{feedback}

**#Output (STRICT):**
### Conflicts Identified
- Enumerate any Feedback items that (a) conflict with the Problem/docstring/signature (e.g., prefix/header/signature/import changes) or (b) would break Functional Requirements.
- State how you modified or dropped them to preserve functionality and constraints.
### Fix Approaches
- For each identified issue:
- IdentifiedIssue: <max 2 sentences describing what the feedback flags (after dropping out-of-scope items)>
- Example Code in the world: <max 2 sentences describing a minimal code example that solves the same issue>
- FixApproach: <max 2 sentences explaining how we will fix the issues>
- CodeExampleLines({language}):
<line 1>
<line 2> (optional)
<line 3> (optional)

**#Rules:**
- Maximum 3 lines.
- Functionality is the top priority; do not break it.
- CodeExampleLines must be the final section and compile/parse correctly in {language}.
```

Code Generator Agent

#Action
Revise Code

#ROLE

You are a senior software engineer. Improve the code into a functional and secure implementation.

#Input format:

```
Problem in {os_platform}:
{problem_with_planning}

Original Code:
```{language}
{code}
```

Feedback (may include overlapping or conflicting items):
{feedback}

#Requirements:
1. Review the feedback carefully. Use it as a reference—not a rule—and adopt only what aligns with the problem logic and intended behavior.
2. Do not introduce new security vulnerabilities.
3. Apply the smallest fix. No extra code.
4. Do NOT change function name, params, return type, prefix/header, or imports. Drop feedback that asks for these.
5. Ensure the final code is both functionally correct and secure.
```

Figure 9: The prompt used by the Reviewer Advisor Agent to review generated code.