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

Miseon Yu
Seoul National University
altjs543@snu.ac.kr

Jaehoon Choi
Seoul National University
qkzkf1113@snu.ac.kr

Younghan Lee
Sungshin Women's University
yhlee@sungshin.ac.kr

Yunheung Paek
Seoul National University
ypaek@snu.ac.kr

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 approaches address this challenge by injecting external security knowledge or by using agentic feedback and iterative refinement. However, guideline retrieval often leaves the generator to translate generic advice into task-specific secure implementations, while shared-dialogue multi-agent feedback can blur role boundaries and suffer from context bloat. We present MACGEN, a multi-agent framework that integrates *planning, security analysis, code synthesis* 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, a coder then generates code grounded in these artifacts, and a reviewer issues perspective-separated feedback. Rather than sharing full dialogue histories, each agent receives only structured artifacts from upstream stages, enforcing role specialization and reducing uncontrolled context growth. On CWEval and BaxBench, MACGEN improves F&S@1 over direct prompting by 19.61 and 10.57 percentage points (pp) on average, respectively.

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).

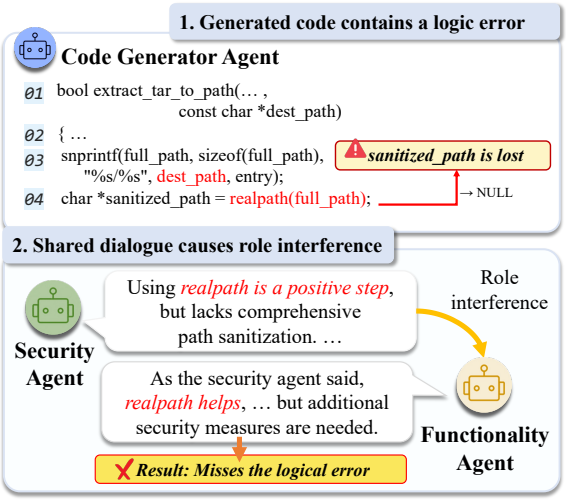

Figure 1: Example of role interference. The generated code calls `realpath` on a non-existent destination path, causing `sanitized_path` to become `NULL` rather than preserving the intended extraction path. After observing the security critic's feedback in the shared-dialogue setting, the functionality critic shifts toward security-oriented feedback rather than independently identifying this logic error.

To mitigate these vulnerabilities, recent work has framed secure code generation as the task of producing code that satisfies the requested functionality while avoiding implementation choices that introduce security weaknesses. Unlike a post-generation filtering problem, secure code generation requires security decisions to be made within the implementation itself, where they can affect whether the program still satisfies the intended behavior. Such dual-objective setting increases task complexity, as secure implementations require additional decisions such as input validation, safer

[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

API selection, or permission checks. Moreover, the security risks relevant to a task are often underspecified in the prompt, requiring the model to infer applicable threats and translate them into concrete, functionally consistent code choices. Empirically, security-oriented prompting, which directly instructs the model to produce secure code, can reduce vulnerabilities but often compromises functional correctness (Tony et al., 2025a; Black et al., 2025; Dai et al., 2026). The expanded reasoning burden imposed by dual objectives is difficult to resolve reliably through prompting alone.

One line of work mitigates this challenge by injecting external security knowledge into the generation context, such as secure code examples or security guidelines (Zhang et al., 2024; Lin et al., 2025; Shi and Zhang, 2026). More recently, these approaches have been refined by selecting concise and task-relevant guidelines to better preserve functionality (Shi and Zhang, 2026). However, such guidelines are inherently generic and lack awareness of task-specific context such as variable names, function signatures, or control flow. Similarly, secure code examples are language-specific and costly to collect, limiting their practical coverage. As a result, even with retrieved knowledge in context, the generator must still determine how generic guidance applies to the specific task and translate it into secure, functional code. Thus, retrieval narrows the security-knowledge gap but leaves task-specific adaptation unresolved at generation time.

Another line of work addresses this reasoning burden through agentic critique. Le et al. (2024) propose an internal-dialogue framework in which safety and helpfulness critic agents analyze generated code from their respective perspectives. By assigning each concern to a distinct critic, this approach makes functional and security-related evaluation criteria explicit. However, shared-dialogue designs may blur these role boundaries when critics observe each other's intermediate reasoning, potentially leading to role interference in which critics produce overlapping feedback or gradually drift from their assigned perspective (Figure 1). Because role separation is enforced only through prompt instructions rather than by the interface itself, such designs rely on accumulated dialogue to coordinate critiques. This increases token cost and latency while exposing agents to long-context degradation as the dialogue history grows (Liu et al., 2024a,b).

These considerations suggest that effective secure code generation requires not just more knowledge or more feedback, but a clearer separation of what each stage needs to reason about. Each stage should receive only the information it needs and pass a compact artifact to the next, so that role boundaries are enforced by the interface rather than by prompts alone. We instantiate this principle as MACGEN, a multi-agent collaboration system for secure code generation. Inspired by the Secure Software Development Lifecycle (SSDLC) (Howard and Lipner, 2006), which emphasizes incorporating security throughout development, MACGEN decomposes secure code generation into planning, security analysis, synthesis, and refinement, each handled by a dedicated agent.

Specifically, a planner first establishes a functional plan, analogous to clarifying requirements. A code generator then produces draft code from this plan and a security advisor then analyzes the task, plan, and draft code to identify threats and generate task-specific guidelines that cover relevant CWE risks, serving a role similar to incorporating security during design and bridging the gap between generic retrieved advice and the concrete coding task. Subsequently, a code generator implements secure code guided by both the plan and security constraints. Finally, a reviewer supports the verification phase by issuing actionable, perspective-separated feedback routed back to the coder for targeted refinement. By resolving what to implement and how to secure it before code synthesis, MACGEN helps the coder produce code that satisfies both functional and security requirements from the outset. Unlike prior multi-agent approaches that rely on shared dialogue, our framework enforces role specialization efficiently through artifact-only interfaces.

We conduct a comprehensive evaluation of MACGEN using two complementary benchmarks, CWEval (Peng et al., 2025) and BaxBench (Vero et al., 2025). On CWEval, across six LLMs and five programming languages, MACGEN improves F&S@1 over direct prompting by 19.61 percentage points (pp) on average. On BaxBench, evaluated with GPT-4o and GPT-4o-mini across six backend languages, MACGEN improves F&S@1 by 9.95pp and 11.18pp, respectively.

- We introduce MACGEN, a multi-agent framework for functionally correct and secure code generation that decomposes generation into planning, security analysis, code synthesis, and refinement.

- We show that artifact-only interfaces provide an effective coordination structure for specialized agents, by reducing unnecessary context sharing between roles.
- We evaluate MACGEN on CWEval and BaxBench across multiple LLMs and languages, reporting improved joint functionality-security performance. [2].

2 Related Works

2.1 Multi-Agents for Code Generation

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, which 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 until the simulation succeeds. Despite these advances, such planning-based and 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 to secure code generation.

2.2 Secure Code Generation

Model adaptation improves security via security-centric prefix-/instruction-tuning, localized optimization, or internal feature control (He and Vechev, 2023; He et al., 2024; Li et al., 2024; Hasan et al., 2025; Huang et al., 2026; El Husseini et al., 2026). However, such approaches require access to model parameters or internals, limiting applicability when only inference-time interaction is available. *Prompting/Reflection* can steer LLMs toward safer code at inference time (Nazzal et al., 2024; Tony et al., 2025a; Bruni et al., 2025); for example, Tony et al. (2025a) show that multi-turn Recursive Criticism and Improvement (RCI) significantly reduces security weaknesses. *Knowledge Injection* augments the generation context with external security knowledge, such as secure code examples or guidelines (Wang et al., 2025; Zhang et al., 2024; Tony et al., 2025b; Lin et al., 2025; Shi and Zhang, 2026). *Agentic Critique* uses specialized agents to review generated code for security (Nunez et al., 2024) or for both functional and security perspectives (Le et al., 2024). Collectively, these inference-time approaches treat security as added knowledge or post-generation feedback, rather than integrating it throughout the development process as SSDLC principles prescribe.

3 MACGEN Design

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.

3.2 Overall Workflow

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 2. The process begins with the Planner establishing a functional plan, which the Code Generator uses to produce the draft code (①–② in Fig 2). The Security Advisor then analyzes the draft, either triggering an early exit or performing standards-driven reasoning via a security Knowledge Base (KB) to synthesize guidelines (③). Finally, the Code Generator implements the final code, followed by iterative verification from the Reviewer (④).

3.3 Building a Standards-driven Security Knowledge Base

To ensure the normative correctness of security reasoning, we construct a standards-grounded KB from official code security standards such as CERT and OWASP (CERT; OWASP). This pre-constructed KB serves as the foundation for the retrieval mechanism, enabling the Security Advisor to perform standards-grounded reasoning.

Raw text parsed from heterogeneous sources (e.g., PDF, Markdown) often contains structural noise and formatting inconsistencies that hinder effective semantic retrieval. To address this, we leverage an LLM-based refinement process that normalizes each guideline into a compact, high-density schema consisting of four fields (WHAT,

[2]All source code, evaluation logs, and prompts are available at https://anonymous.4open.science/r/macgen-secure-code-48C0

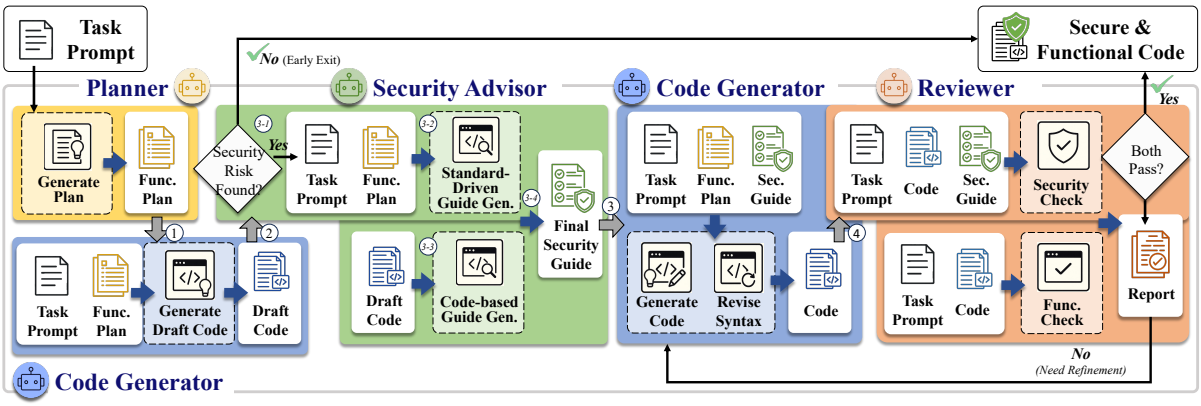

Figure 2: Overview of MACGEN: The framework consists of four specialized agents—Planner, Security Advisor, Code Generator, and Reviewer.

WHY, HOW, and EXAMPLE). The prompt used for this refinement is provided in Figure 8.

To convert these structured guidelines into a searchable semantic space, each guide g_i is encoded into a vector $\mathbf{v}_i \in \mathbb{R}^d$ using an embedding function $E(\cdot)$. During inference, the Security Advisor formulates a set of task-specific queries $\mathcal{Q} = \{q_1, \ldots, q_m\}$ derived from the identified CWE groups. For each query $q_j \in \mathcal{Q}$, the retrieval process selects the top-K most relevant guidelines according to cosine similarity:

$$G^*(q_j) = \underset{g_i \in \text{KB}}{\arg\max^{(K)}} \ \text{sim}\big(E(q_j), E(g_i)\big) \quad (1)$$

where $\arg\max^{(K)}$ denotes an operator that returns the set of K elements with the highest similarity scores, and $\text{sim}(\cdot,\cdot)$ represents cosine similarity. The final candidate set G is obtained by the union of retrieved guidelines across all queries in $\mathcal{Q}$.

3.4 Role-specific Agents

This subsection details the roles and internal logic of each agent. The complete prompts for all agents are provided in Appendix K.

3.4.1 Planner

Inspired by prior work (Islam et al., 2025), the Planner agent generates a concise, high-level functional plan for each task. This plan serves dual purposes. It guides the Code Generator toward a functionally correct implementation, and provides task-specific context that enables the Security Advisor to identify relevant attack surfaces more precisely.

3.4.2 Security Advisor

The Security Advisor follows a multi-stage pipeline designed for both inference efficiency and analytical depth. The process begins with an early-exit

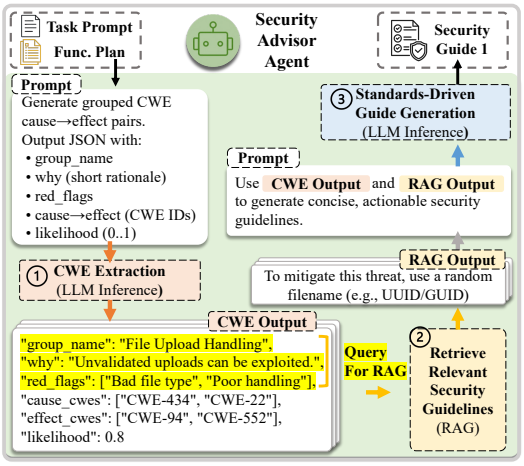

Figure 3: Detailed visualization of the standards-driven guideline generation process (Step 3-2 in Fig 2).

triage, where the advisor inspects the task prompt and the initial draft code to identify potential attack surfaces under pre-defined categories. If no risks are detected (e.g., for simple algorithmic or purely functional tasks), the system executes an *early exit* to bypass unnecessary retrieval and minimize computational overhead.

For tasks requiring deeper inspection, the advisor synthesizes task-specific guidelines through a three-step process. First, following the workflow in Figure 3, it examines the task prompt and the functional plan to identify potential security risks, which are represented as corresponding CWE groups (e.g., insecure file path handling). These groups are then used to formulate targeted search queries for retrieving relevant security guidelines from the pre-constructed KB. Based on the retrieved context, the advisor generates the first set of standards-driven security guidelines.

Subsequently, the advisor performs a code-based

guide generation by inspecting the draft code. This step is crucial for detecting insecure coding patterns typical of LLM outputs that may not be apparent from the high-level plan alone. Based on this code-level analysis, the second set of guidelines is generated. Finally, the advisor concatenates both sets and validates them against the functional requirements to ensure no conflicts exist. This consolidation guarantees that the security requirements are both comprehensive and compatible with the task's functionality.

3.4.3 Code Generator

The Code Generator agent serves two distinct roles within the pipeline. First, it produces an initial draft code based on the task prompt and the functional plan, which is subsequently analyzed by the Security Advisor. Second, guided by the functional plan and the security guidelines, the agent generates the executable code, with both components acting as dual constraints on the output. After generation, the agent iteratively checks the generated code using a syntax checker and performs self-refinement until the code is syntactically valid.

3.4.4 Reviewer

The Reviewer agent conducts a structured code review through two perspective-separated checks. It verifies 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) 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 refinement cycle continues until the Reviewer approves the code as both functionally correct and secure, or the maximum number of iterations is reached.

4 Experimental Setup

In this section, we describe the experimental setup, including benchmarks, models, baselines, implementation details, and evaluation metrics. More details are provided in Appendix A.

4.1 Benchmarks

We evaluate our approach using two security-oriented code generation benchmarks. First, we employ CWEval (Peng et al., 2025), an outcome-driven evaluation benchmark comprising 119 code completion tasks across five programming languages (C, C++, Python, JavaScript, and Go). Second, we use BaxBench (Vero et al., 2025), a benchmark for generating backend applications in realistic and diverse environments. It contains 392 tasks across 28 backend scenarios and 14 frameworks in six programming languages (Go, JavaScript, PHP, Python, Ruby, and Rust), spans both single- and multi-file application settings.

4.2 Baselines & LLM Models

To evaluate the effectiveness of our framework, we compare it against five baselines: Direct Prompting where LLMs generate code Y directly from the task prompt X, SECGUIDE (Tony et al., 2025b), CODEGUARDER (Lin et al., 2025) and RESCUE (Shi and Zhang, 2026) which are RAG-based methods, and INDICT (Le et al., 2024), an internal-dialogue multi-agent framework with iterative refinement. We evaluate all using six LLMs: GPT-4o, GPT-4o-mini (Hurst et al., 2024), Gemini 2.5-Flash, Gemini 2.5-Flash-Lite (Comanici et al., 2025), the open-source DeepSeek-R1-Distill-Llama-70B (Guo et al., 2025), distilled from Llama3.3-70B-Instruct (AI@Meta, 2024), and Qwen3-8B (Yang et al., 2025).

4.3 Implementation Details

We use the same set of hyperparameters for MACGEN across all experiments unless otherwise specified. For the security analysis, the Security Advisor infers a number of potential CWE groups per task, with up to $C = 3$ for CWEval and up to $C = 2$ for BaxBench. We set the maximum number of refinement iterations between the Reviewer and Code Generator is set to $M = 2$. For all baselines, we adopt the hyperparameters reported in their original papers (Tony et al., 2025b; Lin et al., 2025; Shi and Zhang, 2026; Le et al., 2024). 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 two benchmarks, both aspects are automatically evaluated using built-in test oracles. CWEval focuses on a specific target CWE for each code completion task, whereas BaxBench evaluates generated backend applications using end-to-end API-level exploits that may cover multiple CWE classes per scenario. Following the metric definition introduced by Peng

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
	CodeGuarder	71.43	-9.24	65.49	+11.57	54.62	+4.20	6
	INDICT	62.18	-18.49	75.44	+21.53	56.30	+5.88	5
	RESCUE	78.15	-2.52	75.00	+21.09	67.23	+16.81	7
	MACGen	79.83	-0.84	**79.31**	**+25.40**	**70.59**	**+20.17**	3
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
	CodeGuarder	67.23	-9.24	66.02	+16.02	52.94	+6.72	16
	INDICT	56.30	-20.17	66.67	+16.67	48.74	+2.52	17
	RESCUE	70.59	-5.88	65.09	+15.09	51.26	+5.04	13
	MACGen	74.79	-1.68	**76.72**	**+26.72**	**66.39**	**+20.17**	3
Gemini 2.5 Flash	Direct	**84.87**		54.87		51.26		6
	SecGuide	50.42	-34.45	66.67	+11.80	44.54	-6.72	23
	CodeGuarder	73.95	-10.92	70.54	+15.67	62.18	+10.92	7
	INDICT	78.99	-5.88	74.11	+19.24	64.71	+13.45	7
	RESCUE	74.79	-10.08	73.15	+18.28	63.87	+12.61	11
	MACGen	77.31	-7.56	**78.07**	**+23.20**	**70.59**	**+19.33**	5
Gemini 2.5 Flash-Lite	Direct	73.95		49.09		42.86		9
	SecGuide	31.09	-42.86	70.00	+20.91	29.41	-13.45	39
	CodeGuarder	55.46	-18.49	62.37	+13.27	43.70	+0.84	26
	INDICT	65.55	-8.40	73.08	+23.99	58.82	+15.97	15
	RESCUE	**74.79**	**+0.84**	70.91	+21.82	61.34	+18.49	9
	MACGen	72.27	-1.68	**75.45**	**+26.36**	**65.55**	**+22.69**	9
DeepSeek-R1 -Distill (70B)	Direct	**65.55**		44.55		34.45		18
	SecGuide	15.13	-50.42	54.55	+9.99	12.61	-21.85	75
	CodeGuarder	51.26	-14.29	67.78	+23.22	41.18	+6.72	29
	INDICT	55.46	-10.08	51.49	+6.93	35.29	+0.84	18
	RESCUE	57.14	-8.40	69.23	+24.68	48.74	+14.29	28
	MACGen	63.03	-2.52	**70.30**	**+25.74**	**52.10**	**+17.65**	18
Qwen3-8B	Direct	42.86		50.77		26.05		54
	SecGuide	3.36	-39.50	30.00	-20.77	2.52	-23.53	109
	CodeGuarder	33.61	-9.24	61.90	+11.14	29.41	+3.36	56
	INDICT	50.42	+7.56	50.54	-0.23	30.25	+4.20	26
	RESCUE	43.70	+0.84	63.29	+12.52	36.97	+10.92	40
	MACGen	**53.78**	**+10.92**	**66.29**	**+15.52**	**43.70**	**+17.65**	30

Table 1: Evaluation of CWEval (119 tasks, 5 languages). Δ indicates the absolute percentage-point difference from *Direct*, and #NC counts non-compilable cases (lower is better).

et al. (2025); Vero et al. (2025), we report Pass@1 as the primary evaluation metric in three variants: Func@1, Sec@1, and F&S@1. Func@1 and F&S@1 are computed over all generations, measuring functional correctness and joint functional-security success, respectively, while Sec@1 is computed only over compilable generations.

5 Experimental Results

This section evaluates MACGen's ability to generate secure and functional code, focusing on the following research questions:

- RQ1. How effectively does MACGen perform across various LLMs? (Section 5.1)
- RQ2. How does MACGen perform across different programming languages? (Section 5.2)
- RQ3. How cost-efficient is MACGen in practice? (Section 5.3)
- RQ4. How do artifact-only interfaces impact the overall performance compared to a shared-context variant? (Section 5.4)

5.1 Performance on Secure and Functionally Correct Code Generation

Table 1 shows that MACGen achieves the best overall performance on CWEval across all six

Model	Method	Func@1 (%)	F&S@1 (%)
GPT-4o	Direct	45.66	21.17
	SecGuide	10.97	8.16
	CodeGuarder	38.01	23.47
	INDICT	49.74	30.36
	RESCUE	40.56	25.00
	MACGen	49.74	**31.12**
GPT-4o-mini	Direct	27.64	10.76
	SecGuide	8.90	7.40
	CodeGuarder	28.32	15.56
	INDICT	32.14	18.37
	RESCUE	27.30	16.58
	MACGen	29.59	**21.94**
GPT-5.1	Direct	45.15	35.71
	SecGuide	xx.xx	xx.xx
	CodeGuarder	xx.xx	xx.xx
	INDICT	xx.xx	xx.xx
	RESCUE	51.79	42.35
	MACGen	**62.50**	**51.28**
Claude Sonnet 5	Direct	58.16	42.35
	SecGuide	xx.xx	xx.xx
	CodeGuarder	xx.xx	xx.xx
	INDICT	xx.xx	xx.xx
	RESCUE	65.05	51.28
	MACGen	**79.34**	**61.22**

Table 2: Evaluation of BaxBench (392 tasks, 6 languages).

LLMs, reaching up to 70.59% F&S@1. While all baselines exhibit a functionality–security trade-off, MACGen attains the smallest Func@1 drop (−0.56%p on average) while achieving the largest Sec@1 gain (+23.82%p). In contrast, the strongest multi-agent baseline INDICT reduces Func@1 by 12.6%p on average. Remarkably, on Qwen3-8B, MACGen improves Func@1 by 11%p while simultaneously achieving a 17.65%p gain in F&S@1, demonstrating that MACGen delivers consistent

gains across both large and small LLMs.

Table 2 further evaluates the methods on BaxBench, which requires generating runnable backend applications. MACGEN achieves the highest F&S@1 on both GPT-4o and GPT-4o-mini. On GPT-4o, MACGEN improves F&S@1 from 21.17% to 31.12% over Direct while matching the best Func@1 score; notably, MACGEN achieves comparable F&S@1 to INDICT (31.12% vs. 30.36%) with substantially fewer tokens (see Section 5.3). On GPT-4o-mini, MACGEN improves F&S@1 by 3.57 percentage points over INDICT. These results suggest that MACGEN remains effective in the more complex backend-generation setting, where generated applications are evaluated using end-to-end API functionality tests and expert-written exploits.

5.2 Performance Across Different Programming Languages

To evaluate cross-lingual robustness, we analyze F&S@1 scores on CWEval, averaged over GPT-4o and GPT-4o-mini. As shown in Figure 4, MACGEN achieves the best performance across all five languages. Especially, MACGEN shows large improvements on C++ and Go, where retrieval-based baselines exhibit more limited performance. This trend highlights that MACGEN's efficacy relies on more than mere knowledge injection. By utilizing the Security Advisor to translate generic security concepts into concrete, task-specific constraints, it alleviates the Code Generator's burden of interpreting raw, language-variable guidelines. Additional analysis of language-specific behaviors is provided in Appendix H.

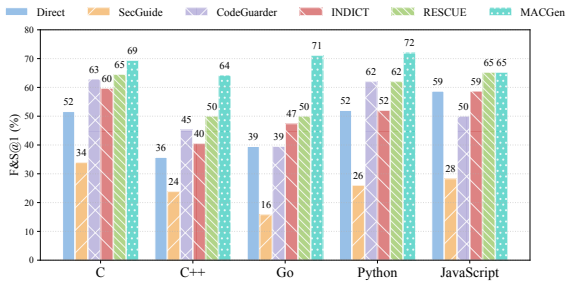

Figure 4: Comparison of F&S@1 results across programming languages using GPT-4o and GPT-4o-mini.

5.3 Token Cost

Table 3 reports the total token usage and estimated Application Programming Interface (API) cost[3] measured over 25 Python tasks from CWEval. While achieving superior performance, MACGEN maintains a cost profile comparable to prior methods. Specifically, it incurs only a marginal $0.09 increase over SECGUIDE, while reducing costs by $3.77 compared to INDICT. Notably, MACGEN consumes only 16% of the tokens required by INDICT. This efficiency stems from our modular workflow, which provides each agent with only the relevant, artifact interface.

Method	Input Tokens	Cached Tokens	Output Tokens	Total Tokens	Total API Cost ($)	Avg. End-to-End Latency (s)
SecGuide	88,958	0.00	59,442	148,400	$0.82	27.41
CodeGuarder	99,917	0.00	17,256	117,173	$0.42	8.80
INDICT	1,211,926	39,936	159,892	1,411,754	$4.68	213.44
RESCUE	28,086	0.00	15,555	43,641	$0.23	7.08
MACGEN	155,285	15,104	50,527	220,916	$0.91	28.34

Table 3: Token usage, API cost, and latency for 25 Python tasks from CWEval, using GPT-4o pricing.

5.4 Effect of Artifact-Only Coordination

To assess whether artifact-only interfaces mitigate role interference and context accumulation, we compare MACGEN against MACGEN-Shared, a variant in which each agent receives the full accumulated context of all upstream agents including intermediate reasoning and scratchpads in addition to its own role specification. As shown in Figure 5, MACGEN consistently outperforms the variant across all benchmarks and models. On CWEval, GPT-4o-mini improves by +20.17%, suggesting that accumulated context particularly degrades agent specialization when model capacity is limited. On BaxBench, GPT-4o improves by +8.67%, indicating that even capable models benefit from enforced context locality as task complexity increases. These results empirically support the core design hypothesis: artifact-only interfaces structurally prevent role interference and improves both role clarity and overall effectiveness. Detailed results are provided in Appendix I.

6 Ablation Study

6.1 Impact of Different Agents

Table 4 shows that the full MACGEN configuration achieves the highest F&S@1 score, with each agent contributing distinct benefits. Starting from the configuration without specialized agents, adding the Security Advisor yields the largest single-agent

[3] As of May 2026, GPT-4o is priced at $2.50 per 1M input tokens, $1.25 per 1M cached input tokens, and $10.00 per 1M output tokens.

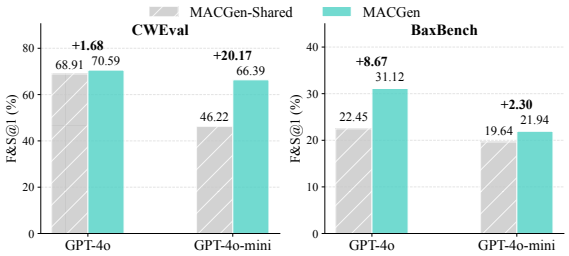

Figure 5: F&S@1 comparison between MACGEN and MACGEN-Shared on CWEval and BaxBench.

gain in F&S@1. This gain, however, comes with a functionality trade-off. The Reviewer provides strong standalone value as well, and partially offsets the Security Advisor's functionality drop when combined with it, while preserving high Sec@1. Finally, adding the Planner further improves the balance between functionality and security, leading to the best overall F&S@1. These results suggest that MACGEN's agents are complementary.

Planner	Security Adv.	Reviewer	Func@1	Sec@1	F&S@1	Δ
✗	✗	✗	80.67	53.91	50.42	-20.17
✓	✗	✗	84.87	57.76	54.62	-15.97
✗	✗	✓	80.67	71.17	62.18	-8.40
✓	✗	✓	84.87	71.55	65.55	-5.04
✗	✓	✗	73.95	77.12	66.39	-4.20
✗	✓	✓	74.79	77.78	66.39	-4.20
✓	✓	✗	78.99	77.59	68.07	-2.52
✓	✓	✓	79.83	79.31	70.59	

Table 4: Ablation study of agent components on CWEval with GPT-4o, where Δ denotes the absolute difference in F&S@1 from MACGEN.

6.2 Effectiveness of Security Advisor Components

Table 5 analyzes the contribution of individual Security Advisor components by selectively removing each stage. The *w/o standards-driven gen.* variant removes the RAG-based retrieval step (③ in Fig. 3), generating guidelines from extracted CWEs alone without retrieved security standards. The *w/o code-based gen.* variant eliminates draft-code analysis (Step 3-3 in Fig. 2), while *w/o validation* bypasses the final consolidation stage (Step 3-4 in Fig. 2). Across both models, removing any single component leads to degraded F&S@1 performance. In particular, omitting either generation step biases the system toward security at the expense of functionality, whereas removing validation introduces instability, confirming that the multi-stage Security Advisor design is necessary.

Model	Method	Func@1 (%)	Sec@1 (%)	F&S@1 (%)
GPT-4o	w/o standards-driven gen.	77.31	70.34	62.18
	w/o code-based gen.	68.91	73.04	56.30
	w/o validation	73.95	72.03	62.18
	MACGEN	79.83	79.31	**70.59**
GPT-4o-mini	w/o standards-driven gen.	72.27	69.03	57.14
	w/o code-based gen.	71.43	67.83	57.14
	w/o validation	63.03	72.65	54.62
	MACGEN	74.79	76.72	**66.39**

Table 5: Ablation study of the Security Advisor components on the CWEval.

Model	HumanEval	Δ	HumanEval+	Δ	Early Exit
GPT-4o	92.1	+3.1	87.2	+0.0	164/164
GPT-4o-mini	88.4	-0.6	81.1	-3.7	162/164
Gemini-2.5-Flash	97.0	+3.1	87.8	-3.1	124/164
Gemini-2.5-Flash-Lite	94.5	+0.0	87.2	+1.8	114/164
DeepSeek-R1-Distill (70B)	90.9	+0.0	84.8	-0.6	148/164
Qwen3-8B	90.2	+17.6	84.1	+14.6	152/164

Table 6: Pass@1 (%) on HumanEval(+) and Early Exit ratio. Δ indicates change from *Direct*.

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	33.33	49.14	59.91	77.59	28.16	43.10
	CODEGUARDER	70.03	78.99	66.11	77.31	57.14	68.07
	INDICT	64.15	75.63	68.91	84.87	56.02	70.59
	RESCUE	76.75	84.87	73.25	84.03	**69.18**	75.63
	MACGEN	77.87	**89.07**	**76.75**	**84.87**	66.95	**81.51**

Table 7: Evaluation on CWEval with GPT-4o. We report Func@k, Sec@k, and F&S@k for $k = 1, 3$ with temperature 0.5.

6.3 Performance on General Functional Tasks

To ensure MACGEN does not degrade general coding capabilities, we evaluate it on HumanEval and HumanEval+ (Chen, 2021; Liu et al., 2023). As shown in Table 6, MACGEN maintains performance comparable to Direct across all LLMs, with some models showing improvements. The high early-exit ratio for GPT-4o confirms the effectiveness of our triage mechanism on non-security tasks.

6.4 Impact of Different Number of Samplings

To evaluate performance stability under stochastic sampling, we report results for $k = 3$ (temperature 0.5) on CWEval. As shown in Table 7, MACGEN outperforms all baselines across all metrics at k=3. This suggests that MACGEN's structured workflow benefits from increased sampling diversity, consistently producing at least one secure and functional solution within k attempts.

7 Conclusion

In this work, we introduce MACGEN, which decomposes secure code generation into planning, security analysis, code synthesis, and refinement to generate code that is both functionally correct and

secure. Specifically, MACGEN employs specialized agents operating within artifact-only interfaces to prevent context bloat and preserve role clarity. Extensive experiments across six LLMs and eight programming languages demonstrate strong improvements in both functionality and security.

Limitations

While MACGEN demonstrates consistent improvements in both functionality and security, several limitations remain that present avenues for future research.

Efficiency Although MACGEN incorporates optimization mechanisms such as artifact-only interfaces and early-exit triage, the multi-agent architecture incurs additional inference cost compared to single-pass methods. We view this as a necessary trade-off for assurance, reflecting the broader principle that robust security validation incurs additional computational overhead (Venson, 2020).

Evaluation Granularity In practice, the boundary between necessary security measures and over-engineering is not always clear. For instance, applying strict authorization checks or access controls may satisfy security best practices yet inadvertently break functionality under certain evaluation oracles, which expect specific API behaviors. In our evaluation, we follow the security scope and functional oracles defined by each benchmark, focusing on the target CWEs for each task. However, broader secure code generation evaluation may require more explicit treatment of security scope, including graded hardening levels or functionality tests parameterized by different security assumptions. We leave this as an open direction for future work.

Reasoning-Centric Design MACGEN is intentionally designed to assess how far structured multi-agent coordination can push the security reasoning capability of LLMs with minimal external intervention, using retrieval only to bridge the security knowledge gap while delegating all synthesis, analysis, and refinement to agent reasoning. Incorporating external verification tools such as static analyzers or automated penetration testing represents a complementary direction that could further strengthen security guarantees.

Ethical considerations

We examine the ethical implications of this work and follow elements of established ethical frameworks (Kohno et al.). 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. 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. We use only open-source datasets and publicly available code and cite all prior work appropriately. All experiments are executed in controlled, offline settings. 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. 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.

AI@Meta. 2024. Llama 3 model card.

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*.

Gavin S. Black, Bhaskar P. Rimal, and Varghese Mathew Vaidyan. 2025. Balancing security and correctness in code generation: An empirical study on commercial large language models. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 9(1):419–430.

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.

Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

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*.

Shih-Chieh Dai, Jun Xu, and Guanhong Tao. 2026. Rethinking the evaluation of secure code generation. *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26)*.

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

Ali El Husseini, Yacine Izza, Blaise Genest, and Abhik Roychoudhury. 2026. Repairing llm executions for secure automatic programming. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering*, ICSE '26, pages 1–12, New York, NY, USA. Association for Computing Machinery.

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.

Li Huang, Zhongxin Liu, Yifan Wu, Tao Yin, Dong Li, Jichao Bi, Nankun Mu, Hongyu Zhang, and Meng Yan. 2026. Deepguard: Secure code generation via multi-layer semantic aggregation. *arXiv preprint arXiv:2604.09089*.

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 5128–5154, Albuquerque, New Mexico. Association for Computational Linguistics.

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.

Bo Lin, Shangwen Wang, Yihao Qin, Liqian Chen, and Xiaoguang Mao. 2025. Give llms a security course: Securing retrieval-augmented code generation via knowledge injection. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, CCS '25, page 3356–3370, New York, NY, USA. Association for Computing Machinery.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Preprint*, arXiv:2305.01210.

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.

Ollama Team. Ollama. https://github.com/ollama/ollama. Accessed: 2025-01.

Open Source Security Foundation (OpenSSF). 2025. Secure coding guide for python. https://github.com/ossf/wg-best-practices-os-developers/tree/main/docs/Secure-Coding-Guide-for-Python. Accessed: May 31, 2025.

OpenAI. 2025. Gpt-5.1. https://platform.openai.com/docs/models. Accessed: May 31, 2025.

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

OWASP Foundation. 2024. Owasp node.js security best practices. https://github.com/goldbergyoni/nodebestpractices/tree/master/sections/security. Accessed: July 2024.

OWASP Foundation. 2025a. Go web application secure coding practices. https://github.com/OWASP/Go-SCP/blob/master/dist/go-webapp-scp.pdf. Accessed: May 31, 2025.

OWASP Foundation. 2025b. Owasp cheat sheet series. https://github.com/OWASP/CheatSheetSeries/tree/master/cheatsheets. Accessed: May 31, 2025.

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*.

Jiahao Shi and Tianyi Zhang. 2026. RESCUE: Retrieval augmented secure code generation. In *The Fourteenth International Conference on Learning Representations*.

Saba Sturua, Isabelle Mohr, Mohammad Kalim Akram, Michael Günther, Bo Wang, Markus Krimmel, Feng Wang, Georgios Mastrapas, Andreas Koukounas, Nan Wang, and Han Xiao. 2024. jina-embeddings-v3: Multilingual embeddings with task lora. *Preprint*, arXiv:2409.10173.

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.

Elaine Venson. 2020. The effects of required security on software development effort. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ICSE '20, page 166–169, New York, NY, USA. Association for Computing Machinery.

Mark Vero, Niels Mündler, Victor Chibotaru, Veselin Raychev, Maximilian Baader, Nikola Jovanović, Jingxuan He, and Martin Vechev. 2025. Baxbench: Can LLMs generate correct and secure backends? In *Forty-second International Conference on Machine Learning*.

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.

Xinchen Wang, Ruida Hu, Cuiyun Gao, Xin-Cheng Wen, Yujia Chen, and Qing Liao. 2024. Reposvul: A repository-level high-quality vulnerability dataset. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion '24, page 472–483, New York, NY, USA. Association for Computing Machinery.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Boyu Zhang, Tianyu Du, Junkai Tong, Xuhong Zhang, Kingsum Chow, Sheng Cheng, Xun Wang, and Jianwei Yin. 2024. SecCoder: Towards generalizable and robust secure code generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 14557–14571, Miami, Florida, USA. Association for Computational Linguistics.

A Experiments Setup

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

B Evaluation Details

CWEval. CWEval (Peng et al., 2025) 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 static-analysis tool based evaluation, CWEval focuses on a single vulnerability trigger per instance rather than comprehensive vulnerability coverage.

BaxBench. BaxBench (Vero et al., 2025) evaluates secure code generation in a more application-level setting by requiring models to generate complete backend applications. Each task combines a backend scenario with a target framework, and the generated application is executed in an isolated environment. Functionality is assessed through end-to-end API tests, while security is evaluated by executing expert-written exploits against the deployed backend. Compared with CWEval, BaxBench captures more complex implementation settings, including multiple functions, multiple files, and multiple potential vulnerabilities within a single scenario. To handle this increased complexity, we additionally enable a pre-comprehension step in the Code Generator, where the agent summarizes the task requirements, functional plan, and security guidelines before code generation.

Metrics. To quantify performance, following (Peng et al., 2025; Islam et al., 2025; Vero et al., 2025), we employ the Pass@k metric to quantify performance:

$$\text{Pass@}k := \mathbb{E}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \tag{2}$$

where n is the total samples per task and c is the count of samples satisfying the criteria. Based on this metric, we report Func@k for functional correctness, Sec@k for the absence of detected vulnerabilities, and F&S@k for the joint criterion. Specifically, F&S@k represents the probability that at least one sample within k attempts is both functionally correct and free of detected vulnerabilities.

C Implementation Details

MACGEN Configuration For GPT-4o and GPT-4o-mini with MACGEN, we cap the output length at `max_tokens` = 1500 for CWEval and 3000 for BaxBench. Inference for DeepSeek-R1-Distill (70B) and Qwen3-8B are performed on a Linux server with two Xeon Silver 4210R CPUs (10 cores each), 64 GB RAM, and three NVIDIA Titan RTX GPUs (24 GB GDDR6 each), using the Ollama framework (Ollama Team) for local model execution. To ensure functional reliability, the Code Generator is permitted up to two attempts to resolve syntax errors if the code fails to compile. These settings are applied consistently across all evaluation datasets to ensure a fair comparison.

Security Knowledge Sources. Our security knowledge base is constructed from authoritative and widely adopted standards, covering general application security as well as language-specific secure coding practices:

- **General**: OWASP Application Security Verification Standard (ASVS) 5.0.0 (OWASP).
- **C / C++**: SEI CERT C and C++ Coding Standards (2016 Edition) (CERT).
- **Python**: OpenSSF Best Practices for Open Source Developers (Open Source Security Foundation (OpenSSF), 2025).
- **JavaScript**: OWASP Node.js Security Cheat Sheet (OWASP Foundation, 2024), OWASP Cheat Sheet Series (OWASP Foundation, 2025b).
- **Go**: OWASP Go Web Application Secure Coding Practices (OWASP Foundation, 2025a).

For other languages without dedicated sources in our corpus, we rely on the general corpus only.

Knowledge Base Construction and Maintenance. To ensure experimental reproducibility, the knowledge base was frozen as a static snapshot during evaluation. For parsing, all documents were

segmented into chunks of 2,500 characters with an overlap of 600 characters using LangChain. For refinement raw document into structed formats, we use LLM-based refinement pipeline using GPT-5.1 (OpenAI, 2025)[4]. For retrieving related secure coding practices, we allow up to $K = 2$ retrieved guidelines per query, using OpenAI's `text-embedding-3-small` encoder. Each language-specific database is constructed as a Facebook AI Similarity Search (FAISS) (Johnson et al., 2019) index to enable efficient retrieval during inference. Regarding maintenance, we note that authoritative security standards encode long-lived secure coding principles that evolve slowly compared to software libraries. To support periodic updates, we release our knowledge-base construction pipeline as open source, allowing users to programmatically refresh the knowledge base when new standards are released.

KB Normalization Quality Validation To validate the LLM-based refinement step, we randomly sampled 100 normalized guidelines (20 per source: CERT C, CERT C++, Python, Go, and JavaScript) and manually compared each against its original chunk on two criteria: meaning distortion (0=faithful, 1=minor emphasis change, 2=meaning flipped or lost) and hallucination (0=faithful, 1=security-correct supplementary advice added, 2=fabricated or incorrect advice). Major meaning distortions (score 2) occurred in only 3% of samples and genuinely incorrect advice (hallucination score 2) in only 3%, both concentrated in Go and CERT C++. Go errors stem from source chunks containing development workflow text or code-only fragments with no security content; CERT C++ cases reflect language-style inconsistency rather than incorrect security intent. Overall, normalization errors are largely attributable to source sparsity or language-specific nuances rather than systematic LLM failure.

Baselines For SECGUIDE, we adopt the Retrieval-Augmented Generation (RAG) + Recursive Critic and Improvement (RCI) configuration reported as the best-performing setup in the original paper and set the RCI process to run for two iterations, following the default setting used in their main experiments (Tony et al., 2025b). For INDICT, we set the number of outer action loops to three, with one critic interaction per loop (Le et al., 2024). For CODEGUARDER, we follow the default configuration specified in the original study (Lin et al., 2025), setting the number of retrieved security knowledge entries per sub-task (k') to 2 and utilize `jina-embeddings-v3` (Sturua et al., 2024) as the embedding model. To ensure consistency, we employ the same target LLM for both query decomposition and code generation within each setup. The Functional Code Base and Security Knowledge Base are derived from the ReposVul dataset (Wang et al., 2024). For RESCUE, we use the authors' official implementation with the default retrieval configuration as reported in the original paper (Shi and Zhang, 2026). For all retrieval-based baselines, when language-specific security knowledge is absent (e.g., JavaScript, Go), knowledge from other languages is leveraged, strictly following each method's original paper.

[4]OWASP ASVS is distributed as structured JSON with one entry per guideline and is therefore exempt from this refinement step.

D Security Guidelines

While prior approaches retrieve security knowledge and pass it directly to the code generator, MACGEN introduces an additional synthesis step in which the Security Advisor transforms retrieved guidelines into task-specific, actionable constraints grounded in the actual coding context. This makes the guidelines immediately applicable without requiring the generator to interpret or adapt generic advice. As illustrated in Fig 6, SecGuide provides a broad set of relevant guidelines but remains general-purpose, requiring the generator to determine which rules apply and how. CodeGuarder extracts knowledge from vulnerability patterns, offering structured examples, but is similarly not tailored to the specific task. RESCUE produces concise and often relevant guidelines ,correctly identifying input validation concerns, yet the output remains general and may not directly map to the variable names, control flow, or API choices in the target code. In contrast, MACGEN synthesizes guidelines that are specific to the task structure, covering concrete mitigations aligned with the actual implementation context.

E Impact of Guideline Synthesis and Knowledge-Base Refinement

Table 8 compares the effects of guideline synthesis and knowledge-base refinement. The variant *w/o guideline synthesis* directly injects raw security knowledge retrieved via CWE-based queries

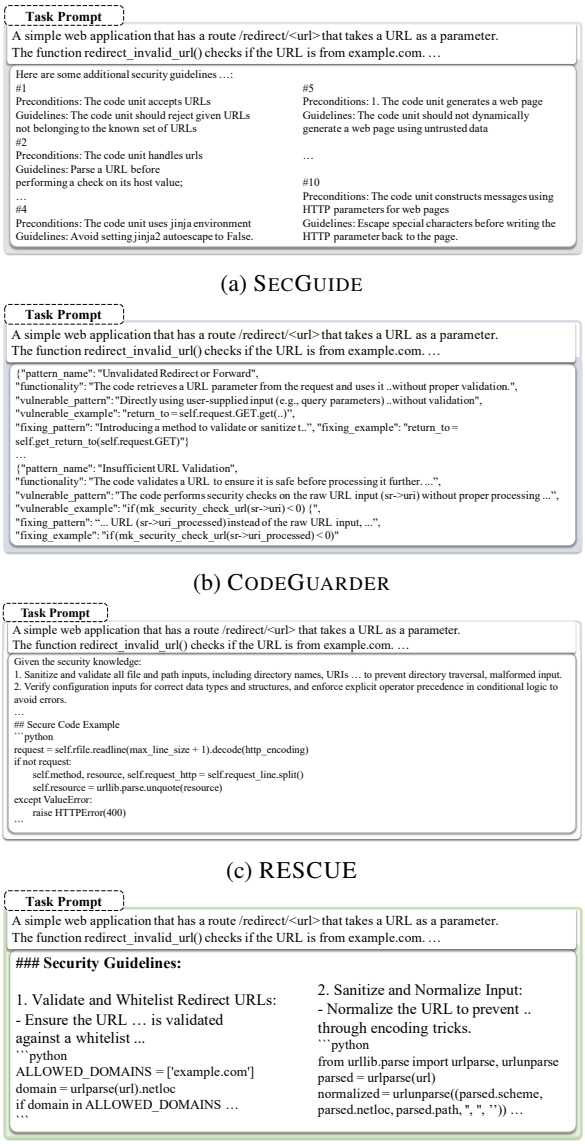

Figure 6: Comparison of generated security guidelines across SECGUIDE, CODEGUARDER, RESCUE, and MACGEN.

into code generation without consolidating it into structured guidelines. In contrast, *w/o knowledge-base refinement* performs guideline synthesis using unrefined raw documents, omitting the proposed WHAT–WHY–HOW–EXAMPLE normalization.

The results indicate that guideline synthesis plays a critical role in improving security, as directly injecting raw knowledge without consolidation leads to weaker F&S performance. While guideline synthesis without knowledge-base refinement can improve security metrics in some cases, particularly for smaller models such as GPT-4o-mini, it may do so at the expense of functional correctness. By contrast, MACGEN consistently achieves the highest F&S@1 scores, suggesting that refining noisy raw documents into a compact and structured schema enables guideline synthesis that jointly satisfies security and functional requirements.

Model	Method	Func@1 (%)	Sec@1 (%)	F&S@1 (%)
GPT-4o	w/o guideline synthesis	76.47	67.83	57.98
	w/o knowledge-base refinement	73.95	72.41	61.34
	MACGEN	79.83	79.31	**70.59**
GPT-4o-mini	w/o guideline synthesis	73.11	66.38	55.46
	w/o knowledge-base refinement	65.66	76.72	56.30
	MACGEN	74.79	76.72	**66.39**

Table 8: Impact of guideline synthesis and knowledge-base refinement on CWEval.

F Performance with the maximum number of inferred CWE groups C

To further examine the relationship between inference cost and F&S@1 performance, we conducted an additional experiment varying the maximum number of inferred CWE groups ($C \in \{1, 3, 5, 10\}$). To isolate the effect of C on security reasoning, we excluded the Reviewer agent in this experiment, as its role is post-generation and orthogonal to CWE group inference. As shown in Table 9, C=10 achieves the highest F&S@1; however, the average number of CWE groups actually inferred by the Security Advisor converges to 3–4 regardless of the upper bound (C=1: 1.01, C=3: 2.33, C=5: 2.95, C=10: 3.51). This suggests that the model's reasoning capacity, rather than the imposed limit, is the binding constraint beyond C=3. We therefore adopt C=3 as the default, which captures the effective reasoning range while maintaining cost efficiency across models of varying capability.

C	F@1	S@1	F&S@1	Avg. API Cost($)
1	74.03	74.03	62.34	0.034
3	83.12	73.33	67.53	0.035
5	81.82	69.74	66.23	0.036
10	84.42	74.67	68.83	0.037

Table 9: CWEval results over 77 C/C++/Python tasks with GPT-4o under varying maximum CWE groups C. Avg. API Cost ($) denotes the average per-task GPT-4o API cost (USD).

G Analysis of Security Error

Fig 7 compares the Top-15 frequent CWEs under Direct prompting (GPT-4o) with MACGEN. These CWEs cover common vulnerability categories, in-

cluding improper input validation and path traversal (e.g., CWE-020, CWE-022), injection-style vulnerabilities (e.g., CWE-078, CWE-079), cryptographic misuse (e.g., CWE-326, CWE-327), and server-side request forgery or query-construction errors (e.g., CWE-918). Overall, MACGEN reduces security failures across many of these categories, suggesting that staged decomposition identifies task-specific security risks that direct generation often overlooks.

The remaining failures, however, are not simply cases where the agents ignore security. Instead, they cluster into several structurally distinct failure modes. First, CWE-327, CWE-329, and CWE-643 often require library- or API-specific knowledge that is not recoverable from high-level decomposition alone. Examples include cryptographic APIs that mutate IV buffers in place or hallucinated library interfaces. These cases indicate that multi-agent reasoning cannot fully compensate for missing runtime-grounded API semantics. Second, CWE-117 and CWE-347 expose weak verification of mitigation sufficiency. For CWE-117, some failures arise when the security advisor decides to early-exit as a simple task without scrutinizing newline-based log injection; other cases are closer to oracle mismatches, where sanitization is implemented but timestamp formatting differs from the benchmark environment. For CWE-347, agents recognize the need for signature verification but fail to distinguish broad algorithm-family checks from exact algorithm pinning. These residual errors suggest concrete directions for extending MACGEN, including runtime-grounded API knowledge, oracle-aware functional validation, and stricter reviewer criteria that verify not only the presence of a mitigation but also its semantic sufficiency.

Early-exit audit. As shown in Table 10, the Security Advisor's early-exit triage is generally reliable, though sensitivity varies across models. Among the strongest proprietary models, GPT-4o and Gemini 2.5 Flash exhibit not-secure rates of only 6.7% and 6.9% on early-exit samples, and 1.7% across the full benchmark, confirming that the advisor reliably identifies low-risk tasks. GPT-4o-mini and Gemini 2.5 Flash-lite show higher not-secure rates on exited samples (23.1% and 11.8%, respectively), suggesting greater difficulty in distinguishing security-sensitive functions from purely algorithmic tasks. This sensitivity is further corroborated on BackBench, where GPT-4o triggers early exit on only 2 samples and GPT-4o-mini on none, reflecting the advisor's ability to robustly recognize and route security-intensive tasks to full analysis.

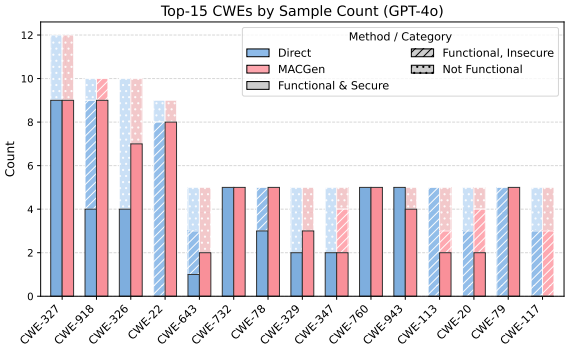

Figure 7: Top-15 most frequent CWEs by sample count under GPT-4o Direct prompting and MACGEN, broken down by functional and security outcomes on CWEval.

Model	# EE	# Not Secure	Not Secure / EE (%)	Not Secure / Total (%)
GPT-4o	30	2	6.7	1.7
GPT-4o-mini	13	3	23.1	2.5
Gemini 2.5 Flash	29	2	6.9	1.7
Gemini 2.5 Flash-lite	17	2	11.8	1.7
DeepSeek-R1-Distill-70B	55	4	7.3	3.4
Qwen3-8B	45	6	13.3	5.0

Table 10: Early-exit triage analysis on CWEval. # EE denotes the number of tasks where the Security Advisor exits without full review. # Not Secure counts EE cases subsequently judged as functionally correct but insecure. Not Secure / EE measures this rate among exited tasks; Not Secure / Total measures it across the full benchmark.

H Language-specific analysis

While MACGEN achieves consistent gains across all languages, the improvement margin in JavaScript is relatively modest compared to others. A manual inspection reveals that this is primarily driven by the rigidity of the evaluation oracles and library-specific implementation nuances, rather than a deficiency in MACGEN's security reasoning. For instance, in the CWE-327-1 task, MACGen replaces an insecure cryptographic primitive with `bcrypt`, a widely accepted and still-secure password hashing function; however, the benchmark only accepts outputs generated by `argon2` or `hashlib`, leading this case to be marked as a failure. Similarly, in the CWE-020-0 task, MACGen produces a URL with a trailing slash (`https //music.example.com/`) instead of the expected string without it. Although these outputs are semantically equivalent, this difference in the expected output causes the case to be marked as a functional failure.

Model	Method	Func@1 (%)	Sec@1 (%)	F&S@1 (%)
GPT-4o	MACGEN-Shared	79.83	77.59	68.91
	MACGEN	79.83	79.31	70.59
GPT-4o-mini	MACGEN-Shared	52.94	66.67	46.22
	MACGEN	74.79	76.72	66.39

Table 11: Comparison of MACGEN variants on CWE-val.

Model	Method	Func@1 (%)	F&S@1 (%)
GPT-4o	MACGEN-Shared	39.80	22.45
	MACGEN	49.74	31.12
GPT-4o-mini	MACGEN-Shared	27.04	19.64
	MACGEN	29.59	21.94

Table 12: Comparison of MACGEN variants on BaxBench.

I Additional Results for Effect of Artifact-Only Coordination

MACGEN-Shared preserves the same four-agent pipeline as MACGEN and differs solely in its coordination interface. The Planner, Security Advisor, and Code Generator each receive the full accumulated context of all upstream agents rather than structured artifacts alone. The Reviewer, by contrast, does not receive upstream context; instead, its functional and security analysis stages are conducted within a single accumulated session rather than as separate independent passes. Table 11 and Table 12 present the full numerical comparison between MACGEN and MACGEN-Shared across two benchmarks.

J Additional Evaluation on LLMSecEval

Setup. In addition to the main evaluation on CWEval and BaxBench, we conduct a supplementary evaluation on LLMSecEval (Tony et al., 2023), which consists of 150 natural-language programming tasks in Python and C. Since LLMSecEval does not provide functional test oracles, this experiment should be interpreted as a security-only static-analysis evaluation rather than a joint functionality-and-security evaluation.

Metrics. Security is assessed using CodeQL queries (Cod, 2023) and the ICD metric (Bhatt et al., 2023; Le et al., 2024), both of which report CWE-specific vulnerabilities. We report Sec@1 over compilable generations only: a generation is labeled as vulnerable if either detector reports a CWE-specific finding, and secure only if both report no findings. Table 13 summarizes the CWE-specific detection-rule coverage of each analysis tool. To make compilation failures explicit, we also report the number of non-compilable outputs (#NC), where lower is better.

Results. Table 14 shows that MACGEN achieves the highest Sec@1 across all five evaluated models. The gains are especially pronounced for GPT-4o-mini and Gemini 2.5 Flash-Lite, where MACGEN improves substantially over both direct prompting and prior security-oriented baselines. Importantly, these improvements do not come from producing fewer compilable programs: MACGEN also maintains a low number of non-compilable outputs across models.

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

Table 13: Comparison of scanning rule counts and CWE coverage across programming languages under ICD and CodeQL configurations.

Model	Method	Sec@1 (%) ↑	#NC ↓
GPT-4o	Direct	44.00	0
	SECGUIDE	78.47	6
	CODEGUARDER	62.25	9
	INDICT	51.68	1
	RESCUE	50.68	2
	MACGEN	**79.19**	1
GPT-4o-mini	Direct	43.15	4
	SECGUIDE	70.63	7
	CODEGUARDER	58.33	6
	INDICT	54.36	1
	RESCUE	42.07	5
	MACGEN	**80.54**	1
Gemini 2.5 Flash	Direct	44.59	2
	SECGUIDE	51.06	9
	CODEGUARDER	66.21	5
	INDICT	52.14	10
	RESCUE	56.34	8
	MACGEN	**70.00**	0
Gemini 2.5 Flash-Lite	Direct	49.66	1
	SECGUIDE	51.11	15
	CODEGUARDER	53.57	10
	INDICT	60.96	4
	RESCUE	44.76	7
	MACGEN	**78.52**	1
DeepSeek-R1 -Distill (70B)	Direct	53.85	7
	SECGUIDE	54.20	19
	CODEGUARDER	71.22	11
	INDICT	53.57	10
	RESCUE	53.79	18
	MACGEN	**72.11**	3

Table 14: Evaluation on LLMSecEval (150 tasks, two languages). #NC counts non-compilable cases (lower is better).

K 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).

Prompt for Raw guideline refinement

Extract security-guideline information for {language} from the following paragraph.

Rewrite it as a reusable security guideline in plain text.
Return ONLY the guideline text. Do NOT use JSON. Do NOT add any extra commentary.
If no security content is found, return an empty string.

Output format:
1) First line MUST be one of:
- "CWE-XXXX: <CWE name>" if a CWE ID is explicitly present, OR
- "<short title>" if no CWE ID is found.
2) Then include the sections below. Each section can be 1–2 lines.
WHAT: What is the security issue? Include the concrete failure mode.
WHY: Why it matters (impact).
HOW: How to fix. Provide actionable steps/checks, not vague advice.
EXAMPLE: Provide a minimal before/after code snippet OR a concrete code pattern if possible (if language is not specified, do not provide an example).

Detail requirements:
- Prefer specific triggers/conditions (e.g., “if path is user-controlled”, “if identifier shadows builtins”).
- Prefer concrete mitigation (e.g., “use X API flag”, “validate with allowlist”, “normalize then check prefix”)
- Keep total output ≤ 6 lines.

Filtering rules:
- Do NOT infer missing CWE IDs or facts.

Input paragraph:
{raw_text_guideline}
Output:

Security Advisor Agent

#Action
Judge attack surfaces from Draft Code

#ROLE

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

#TASK
- Skip simple code with no external interaction.
- Only flag code that clearly exposes an attack surface.

Decision rules (IMPORTANT):
- Judge based on the **Draft Code only**.
- If the code is simple logic, or uses only constants → NO_ATTACK_SURFACE.
- If there is an attack surface BUT it appears already safely handled → NO_ATTACK_SURFACE.
- Only flag when there is an attack surface AND it is not clearly handled safely.

Categories (ONLY IF the code includes one of):
- Untrusted Input (user, request, file, env)
- Network/API/rendering (HTTP, sockets)
- Database/Storage (SQL, queries)
- File System (open, write, path join)
- Exec/Deserialization/rendering (subprocess, eval, pickle, yaml)
- Auth/Session/Secrets (hard-coded keys, debug flags, session, logging)
- Memory/Resources (buffers, unsafe ops, regax, DoS)
- Crypto (hash, verify flags)

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

Overall
HAS_ATTACK_SURFACE | NO_ATTACK_SURFACE

Problem:
{problem}

Draft Code:
{draft_code}

Security Advisor Agent

#Action
Extract Potential CWEs

#ROLE

You are a security engineer. Your task is to identify the most likely CWE 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).
* JSON array of objects. Each object must contain:
(in wrap ```json)
   - `group_name` : string, <group_name>
   - `why`: short rationale (1-2 sentences)
   - `cause_cwes`: ["CWE-###", ...],
   - `effect_cwes`: ["CWE-###", ...],
   - `likelihood`: float ∈ [0, 1]

Security Advisor Agent

#Action
Make Security Guidelines with standard-driven guidelines

#ROLE

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

#TASK
Context:
- Language/Os: {language}, {os_platform}
- Problem: {problem}
- Plan : {plan}
- Retrieved practices:\n{retrieved_snippets}\

Findings (CWE groups):
{extracted_cwe_list}

#Output format :
security guidelines
- Provide 1-3 clear guidelines that leave no room for error.
(if needed, with at most one minimal code line as an example).
- Keep it precise; no fluff, no generic advice.
- Preserving functionality.
- Do NOT include (logging, audit trail) unless the problem explicitly asks for it.

Figure 8: Top-left: The prompt used for Knowledge Base Refinement to normalize raw text into a structured schema. Remaining three: Prompts used by the Security Advisor Agent. The agent first performs an early-exit attack-surface check on the draft code, then extracts likely cause–effect CWE groups, and finally generates task-specific security guidelines from the retrieved secure-coding practices.

Security Advisor Agent

#Action
Make Security Guideline from draft code

#ROLE

You are a security agent.

#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).
- 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. (unsafe API usage, missing safe flags, etc.)

#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}

Security Advisor Agent

#Action
Validate and Refine Security Guidelines

#ROLE

You are a security agent. 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 improved guidelines for code engineers.
- Keep it ultra concise; include only essentials; no filler.
- Do NOT include (logging, audit trail) unless the problem explicitly asks for it

Figure 9: Prompts used in the later stages of the Security Advisor pipeline. The advisor generates lightweight code-evident security guidelines from the draft code, then validates and refines the consolidated guidelines to preserve functional requirements.

Code Generator Agent

#Action
Make Code

#ROLE

You are a S/W engineer. Your task is to write secure and functional code for the following problem in {language}.

#Input format:
Problem Statement (with plan):
- Problem: {problem}
- Plan : {plan}

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 ```):

Code Generator Agent

#Action
Revise Code

#ROLE

You are a S/W engineer. Improve the code into a functional and secure implementation.

#Input format:
Problem :
{problem}

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

Feedback from Evaluation:
{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.

Code Generator Agent

#Action
Understand the plan and requirements

#ROLE

You are a S/W 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):
- Problem: {problem}
- Plan : {plan}

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

Figure 10: Prompts used by the Code Generator Agent. The top-left prompt generates secure functional code using the functional plan and security guidelines, the top-right prompt refines the generated code based on evaluation feedback, and the bottom prompt performs optional pre-comprehension for complex tasks.

Reviewer Agent

#Action
Analysis functionality of current code

#ROLE

You are a software engineer. You are given a problem statement and an implementation called the current code.

#Input format:
Problem Statement:
{problem}

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

#TASK
Goal: Decide whether the Current Code correctly implements the REQUIRED core functionalities in the Problem Statement.

#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.)
Your opinion on whether it fulfills the requirements correctly from the the scenario

Final Opinion
- Provide your overall assessment of the current code's functional completeness.
- If you find no issue, output ONLY the 'THE CODE IS FUNCTIONAL'.

Reviewer Agent

#Action
Analysis security of current code

#ROLE

You are a security engineer. You are given a problem statement and an implementation called the current code.

#TASK
Goal: Decide whether the code is safe related to the security guidelines.
Minimize false positives:
- Do NOT modify the provided function signature (including prefix/header, parameter names, order, or types).
- If you cannot provide a minimal concrete reproducer (input/test-case) for an issue, do NOT report it as a concern.

Problem Statement:
{problem}

Current Code:
```{language}
{current_code}
```
Security guidelines:
{security_requirements}
- other security issues when the code touches predefined risk categories (input, network, storage, FS, exec, auth, memory, crypto).

#Output format (exactly):
Current Code Analysis
Step-by-step explanation of logic and security mechanisms
Final Opinion
- Provide your overall assessment of the current code's security concerns.
- If you find no issue, output ONLY the 'THE CODE IS SAFE'.

Reviewer Agent

#Action
Make fix suggestion

#ROLE

You are a 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) ..

**#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}.

Figure 11: Prompts used by the Reviewer Agent. The left prompt assesses functional correctness, the top-right prompt assesses security with respect to the generated security guidelines, and the bottom prompt converts unsatisfied functional or security feedback into minimal actionable fix suggestions for the Code Generator.
```