Safety Sidecar: Reflection-Driven Runtime Control for Safer Agents

Bin Wang[1], **Jiazheng Quan**[2], **Xingrui Yu**[3,*], **Hansen Hu**[1], **Hao Yu**[1], **Anjun Gao**[5], **Zhenglin Wan**[6], **Hui Li**[1,*], **Ivor Tsang**[3,4]
[1]School of Computer Science, Peking University, China
[2]Key Laboratory of Multimedia Trusted Perception and Efficient Computing, Ministry of Education of China, Xiamen University, China
[3]CFAR and IHPC, Agency for Science, Technology and Research (A*STAR), Singapore
[4]College of Computing and Data Science, Nanyang Technological University
[5]Department of Computer Science and Engineering, University of Louisville, United States
[6]Department of Computer Science, National University of Singapore

Abstract

Autonomous LLM agents are increasingly deployed in complex environments as tool-using systems. However, their safety remains fragile, as minor reasoning or retrieval errors can be amplified into hazardous actions within the agentic workflow. Existing defenses, often limited to static prompts or post-hoc guardrails, fail to provide runtime intervention or cross-architecture portability. In this paper, we propose **Safety Sidecar**, a model-agnostic, plug-and-play module designed to provide standardized runtime safety control and auditability for arbitrary agent workflows. Safety Sidecar operationalizes reflection as a closed-loop controller: it dynamically monitors decision traces, retrieves evidence-based repair exemplars from a reflective memory, and enforces risk-mitigating revisions before execution. Crucially, it employs external verifiers to gate both action release and memory updates, producing a transparent, auditable trail of retrieved evidence and applied constraints.We instantiate and systematically evaluate Safety Sidecar in secure code generation—a high-stakes domain with objective vulnerability signals. Experimental results across eight CWE scenarios and four representative LLMs demonstrate that Safety Sidecar consistently improves the secure-solution rate by 2.9–11.2 percentage points while maintaining competitive functional correctness. Efficiency analysis shows the framework is practical for deployment, with reflection adding only 3.2s to end-to-end latency and a negligible average cost of 5.37×10^{-4} per scenario. Our findings position Safety Sidecar as a portable and efficient control layer for enhancing the safety, compliance, and auditability of LLM-based agents.

1 Introduction

Autonomous LLM agents are rapidly evolving from single-turn text generators into systems that execute long-horizon, multi-step tasks, invoke external tools, and make decisions that affect the real world (Chowa et al., 2025; Erdogan et al., 2025; Wang et al., 2025g). This shift tightly couples *internal reasoning* with *external actions*: agents can browse at scale, call APIs, and modify external artifacts, exhibiting compositional capabilities across tasks (Wang et al., 2025a). As a result, the design space is expanding quickly—planning, memory, retrieval, and tool orchestration are increasingly treated as first-class components (Koubaa, 2025; Zeng et al., 2025)—while the challenges of safety, control, and evaluation become substantially more acute (Lian et al., 2025). In parallel, as long-term memory, proactive planning, and environment interaction become core ingredients, agentic AI is moving from a single-model reasoner to a *distributed workflow* of cooperating modules (Wang et al., 2025b). This transformation is reshaping the generative AI community and has sparked broad discussions about agent safety and governance (Park et al., 2023; Abou Ali et al., 2025; Wang et al., 2025d).

A key lesson from early deployments is that many safety failures in agentic systems do *not* stem from the base model merely "being wrong" in isolation. Rather, the *agent workflow amplifies small deviations into executable consequences*. A minor retrieval error, a mistaken threat-model assumption, or an overly-permissive action choice can propagate through the plan–retrieve–execute loop, producing unsafe intermediate artifacts or triggering hazardous tool calls. Even strong models can mishandle APIs, misunderstand security boundaries, or generate insecure patterns. Once tool use and autonomy are introduced, adversarial techniques such as jailbreaks, prompt injection, and agent worms can further *enter the execution loop*—controlling actions, polluting context, or inducing unsafe tool invocations—thereby turning natural-language inputs and intermediate artifacts

*Corresponding Authors: Xingrui Yu, Hui Li. (E-mail: yu_xingrui@a-star.edu.sg, lih64@pkusz.edu.cn)

Findings of the Association for Computational Linguistics: ACL 2026, pages 30842–30856

into attack vectors (John et al., 2025).

Existing defenses face two structural tensions in this setting. First, *after-the-fact safeguards are often too late* (Shahriar et al., 2025; Ferrag et al., 2025). Post-hoc filtering or scanning cannot undo side effects once a tool call has executed, an external state has been modified, or sensitive resources have been accessed; moreover, polluted intermediate artifacts can be written into long-term memory and repeatedly exploited in future runs. Second, *stacking more safeguards does not generalize gracefully*. Heavy-weight rule stacks, interceptors, or bespoke validators are often tightly coupled to specific tools and task distributions; they can break when ported across agent architectures and domains, incur substantial overhead, and may erode the very autonomy that agents are built to provide. Together, these tensions point to a missing capability: we need a *workflow-native, runtime* safety control mechanism (Wang et al., 2025c)—one that intervenes *before* risky actions occur, while remaining lightweight and portable (Wang et al., 2025e; Wibowo and Polyzos, 2025; Zhan et al., 2025).

In light of this gap, we identify two core challenges for trustworthy agent workflows. **(i) Credible control at decision time:** how can we constrain an agent *at runtime*, so that risky behaviors—especially unsafe tool calls or external writes—are detected and corrected *before* they execute? **(ii) Verifiability and auditability:** how can we record the evidence and decision steps that lead to actions and artifacts, enabling transparency and accountability after the fact?

To address these challenges, we propose **Safety Sidecar** (SS) (Figure 1), a standardized, plug-and-play module that can be attached to the *tail* of arbitrary agents. Rather than acting as a passive output filter, SS turns reflection into a first-class *runtime control loop* that intervenes within the agent's decision–action cycle. Concretely, SS (1) performs lightweight self-checking to route risky steps, (2) retrieves evidence-based repair exemplars and security guidance from a dynamic reflective memory, (3) generates reflective repairs *before execution*, and (4) uses external verifiers to *gate* both action execution and memory write-back. This yields a closed loop of *risk detection* $\rightarrow$ *reflective repair* $\rightarrow$ *verification* $\rightarrow$ *memory update*, together with an auditable trail of intermediate decisions and supporting evidence.

Importantly, Safety Sidecar is designed for portability: it requires only a minimal integration interface to intercept an agent's action/output channel, and can therefore be incorporated into diverse agent architectures and application domains. To concretely validate the system-level benefits of sidecar-style runtime control, we instantiate and evaluate Safety Sidecar in secure code generation—a high-stakes testbed with strict executability constraints and objective vulnerability signals (e.g., compilation, testing, and static analysis). Experiments show that, compared to strong baseline agents, attaching Safety Sidecar consistently improves security and traceability while preserving competitive functional correctness and modest overhead.

In this paper, we make the following contributions:

- **A standardized, plug-and-play reflective *control module* for agent workflows.** We design and implement a modular component that integrates risk-aware routing, reflection-driven repair, and audit logging, enabling proactive intervention *within* the agent loop rather than relying solely on post-hoc filtering.
- **Verifier-grounded reflective memory updates for reliable self-correction.** We couple evidence retrieval with reflective repair and *gate* memory write-back using external verifiers, so that only validated repairs are retained as reusable experience, improving stability across repeated runs and reducing recurring failure patterns.
- **A fully verifiable instantiation and rigorous end-to-end evaluation in secure code generation.** We instantiate the module with lightweight self-checking and tool-governance verifiers (compilation, tests, and CodeQL), and provide comprehensive experiments, ablations, and failure analyses under executable and vulnerability-grounded metrics.

2 Related Work

Trustworthiness in Agentic Workflows As LLM agents evolve into autonomous decision-makers, ensuring the trustworthiness of their outputs becomes paramount (Han et al., 2024; Yu et al., 2025). Beyond external attacks, systemic risks such as coordination failures in multi-agent architectures (David and Gervais, 2025; de Witt, 2025; Shen et al., 2025) and unintended tool misuse (Achiam et al., 2023; Wu et al., 2024) threaten reliability.

Frameworks like THOR (Narajala and Narayan, 2025) highlight these vulnerabilities, where even safety-aligned models can exhibit misalignment (Huang et al., 2023; Li et al., 2023) or succumb to manipulation (John et al., 2025; Lee and Tiwari, 2024; Kartik et al., 2025), underscoring the need for robust internal control mechanisms.

Verifiable Control and Self-Correction To enhance output reliability, research is shifting towards intrinsic trust mechanisms under frameworks like TRiSM (Raza et al., 2025). Approaches range from input-layer "self-reminders" (Xie et al., 2023) and refusal training (Yuan et al., 2024; Gou et al., 2023; Ma et al., 2025) to hierarchical monitoring (Wang et al., 2025f) and least-privilege enforcement (Debenedetti et al., 2025). However, existing methods often lack interpretability (Sanwal, 2025), and benchmarks like RAS-Eval (Fu et al., 2025) and CyberSOCEval (Deason et al., 2025) reveal gaps in task control. Our work advances verifiable autonomy (Bouzenia et al., 2024; Guo et al., 2025; Miculicich et al., 2025) by embedding a closed-loop reflection module for transparent, evidence-based self-correction.

3 Problem Formulation and Design Goals

Evaluation setting. To systematically evaluate the trustworthiness of code generation agents, we build a standardized, plug-and-play module that can be inserted into general agentic orchestration and use it to construct a baseline code-generation agent architecture as our benchmark (Figure 1(a)). This paper focuses on improving the safety and trustworthiness of the code repair stage within such agentic pipelines.

Secure code repair. Given a code snippet x that may contain security flaws, together with its contextual environment, which includes file-level context C_f and function-level context C_{fn}. The goal is to generate a repaired snippet y that simultaneously satisfies:

1. **Security**: eliminate common vulnerability classes present in x, such as SQL injection, command injection, buffer overflow, path traversal, XSS, null pointer dereference, integer overflow, and use-after-free.

2. **Functionality**: preserve the intended semantics of x under the given context (C_f, C_{fn}), i.e., the patch should not change program behavior except to remove unsafe behaviors.

3. **Executability**: ensure y is syntactically valid and can be compiled or interpreted in the target environment implied by (C_f, C_{fn}).

We formalize this task as conditional generation augmented with retrieval-based evidence:

$$y = \mathcal{G}\big(x, C_f, C_{fn}, \mathcal{R}(x, C_f, C_{fn}; \mathcal{M}), \theta\big) \quad (1)$$

where $\mathcal{G}$ denotes the large language model (LLM) generator, $\mathcal{R}$ is a retrieval function that returns a set of security-repair exemplars and/or security constraints from a memory repository $\mathcal{M}$, and θ represents model and prompting parameters. The role of $\mathcal{R}$ is to provide *actionable* and *auditable* repair evidence, rather than relying only on the model's parametric knowledge.

Design goals. Based on the formulation above, we aim to support autonomous and trustworthy LLM-based code repair in an agentic setting. Concretely, the module used in our pipeline is designed to meet the following goals:

(**G1**) **High Security Efficacy**: reduce (i) the post-repair vulnerability rate and (ii) the policy-violation rate of generated patches, while avoiding regressions that reintroduce known insecure patterns.

(**G2**) **Verifiability**: for each repair, output structured, machine-verifiable audit evidence.

(**G3**) **Lightweight Integration**: provide these guarantees without requiring complex planning or model fine-tuning, so the module can be integrated into existing coding agents as a plug-and-play component with low overhead.

4 Safety Sidecar: Architecture and Implementation

We instantiate the REFLEX module as the Reflect layer of an agentic framework, where reflection functions as a first-class control circuit supervising planning, execution, and verification. It is integrated into the end-to-end code generation pipeline as a modular, pluggable component. The main workflow is:

1. **Lightweight self-check (routing).** A fast pre-checker screens the *candidate artifact* produced by the agent to avoid unnecessary reflection when risks are unlikely.

2. **Evidence-guided reflective repair.** If the candidate is judged *unsafe*, the system retrieves high-value prior cases and security guidance from a reflective memory, and composes them into a structured reflection prompt that steers the model to produce an evidence-grounded repair *before execution*.

3. **Verification and deposition.** The candidate output is filtered and compilation-checked; verified artifacts are then written back to memory, forming a *continuously evolving* knowledge loop.

To clarify the verification process, Safety Sidecar follows a two-stage design with distinct roles. The Lightweight Self-Checker serves only as a routing mechanism that reduces unnecessary computation by filtering low-risk cases. The final decision on whether an output is safe is determined exclusively by external verification, including compilation checks and static analysis tools such as CodeQL, which avoids self-confirmation bias and ensures that all accepted outputs satisfy objective and verifiable safety criteria.

This design realizes a closed loop of *low-cost routing*, *evidence-driven repair*, and *verifier-gated experience accumulation*, without fine-tuning the underlying models. Building on this design, we implement Safety Sidecar (Figure 1) with three core components:(i) a *Lightweight Self-Checker*, (ii) a *Reflective Prompt Engine*, and (iii) a *Reflective Memory Repository*. The process can be formulated as:

$$(x, c, M_D) \xrightarrow{\text{Reflex}} \begin{cases} (f_\theta(x, c), M_D \cup \{(x, c)\}), & C = 1 \\ (f_\theta(\Phi), M_D \cup \{(x, c, y, E)\}), & C = 0, V = 1 \end{cases} \tag{2}$$

where x denotes the input code, c its contextual information, and M_D the dynamic reflective memory. f_θ is a frozen base language model. The lightweight self-checker $C \in \{0, 1\}$ routes the input as safe (1) or unsafe (0). Safe inputs are directly processed and stored in M_D. For unsafe inputs, relevant prior experiences are retrieved as $E = \text{Retrieve}(M_D, M_S, q)$ with $q = (x, c)$, and incorporated by the reflective prompt constructor Φ together with best-practice constraints $\mathcal{K}$. The generated output y is written back to memory only if it passes the verifier V.

4.1 Lightweight Self-Check and Routing Mechanism

The Lightweight Self-Checker serves as a front-end routing layer for Safety Sidecar, providing an initial safety assessment with minimal overhead.Its goal is not to fully diagnose vulnerabilities, but to cheaply decide whether the candidate artifact should pass through directly or enter the more expensive reflection-and-repair stage.

We implement self-checking as a binary decision based on an LLM. Given the candidate x and context c, the system constructs a concise safety-review prompt p_{sc} and instructs the model to output only a binary verdict: “SAFE” or “UNSAFE”:

$$\text{verdict} = \text{LLM}_{\{\text{SAFE, UNSAFE}\}}\left(p_{\text{sc}} \mid x, c\right) \tag{3}$$

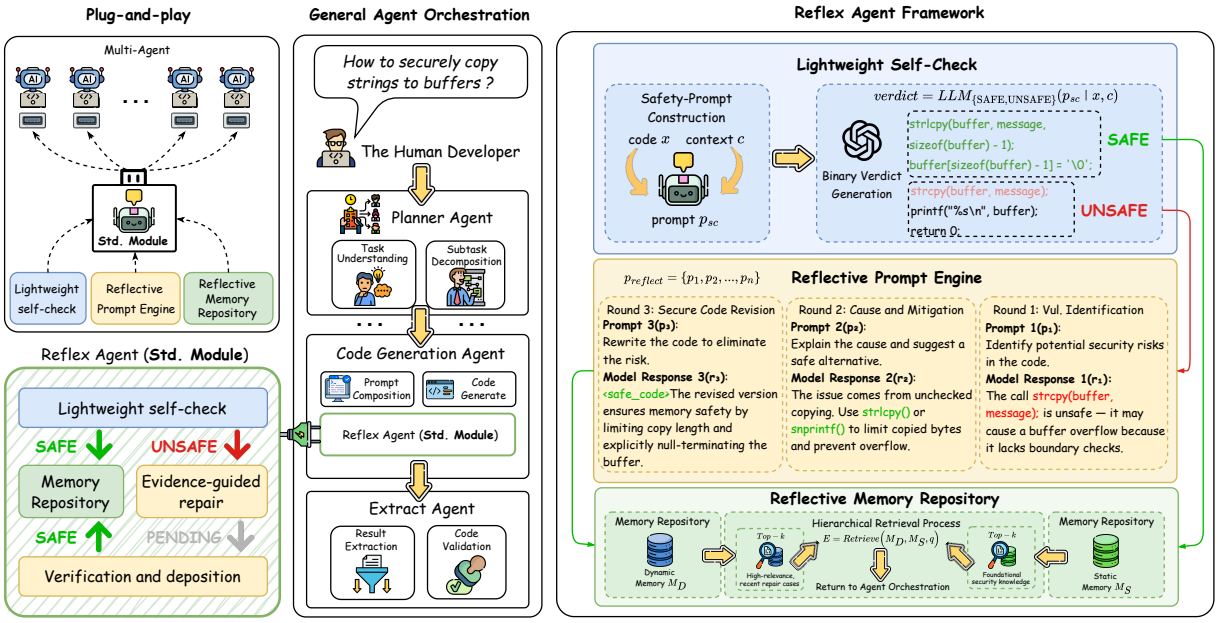
If the verdict is determined as “SAFE”, the sample and its metadata are directly written into the dynamic memory repository as a retained safe case. Conversely, if the verdict is “UNSAFE”, the system triggers the full reflection and repair pipeline, where the reflection prompt engine (introduced in section 4.2) performs in-depth code diagnosis and correction.

This mechanism effectively routes trustworthy responses without redundant reflection processes, thereby reducing the average inference cost of the system. It is particularly advantageous for large-scale task processing scenarios, where efficient filtering and resource allocation are critical.

4.2 Reflective Prompt Engine

The Reflective Prompt Engine serves as the central module of the reflection process, enabling the agent to conduct in-depth analysis and self-improvement on problematic code identified in the first step (self-check). When a code segment is classified as “UNSAFE” and enters the reflection phase, the engine constructs a structured multi-turn prompt that transforms a single code-generation task into a multi-round chain-of-thought reflection dialogue.

During this process, the model is guided to conduct systematic introspection and reasoning on the problematic code and its context, identifying vulnerabilities and devising corresponding repair strategies. This multi-turn reflective dialogue ensures that the complete reasoning chain, which spans from problem recognition to solution implementation, is explicitly unfolded and recorded.



(a) Safety Sidecar as a plug-and-play, tail-attached module in general agent orchestration.

(b) Overall framework of Safety Sidecar.

Figure 1: (a) Plug-and-play integration of Safety Sidecar within a general agent workflow. (b) The internal design of Safety Sidecar, highlighting lightweight routing, reflective repair, and verifier-gated memory updates.

Therefore, this strategy achieves knowledge distillation for code generation tasks.

Under the operation of the Reflective Prompt Engine, the resulting structured reflection records are systematically stored in the dynamic memory repository. In essence, each resolved case is elevated into a reusable reasoning pattern, offering high-quality reference for future related issues. As more solutions and reflective processes accumulate, the system's knowledge base evolves continuously, enhancing both the quality of stored knowledge and the agent's adaptive capability in subsequent tasks. This growing body of experience strengthens the overall trustworthiness and transparency of the intelligent agent framework.

4.3 Reflective Memory Repository

The Reflective Memory Repository is the central component enabling the system's continual learning and experience reuse. By integrating vectorized retrieval with structured metadata management, it constructs a dynamically evolving knowledge base for safety repair and reflection.

To balance retrieval efficiency with comprehensive knowledge coverage, we design a hierarchical retrieval architecture composed of two layers:

- **Dynamic Memory** M_D: Implemented with a ChromaDB vector database to store and index verified repair cases produced during system operation. This design choice facilitates high relevance and low retrieval latency.
- **Static Memory** M_S: Built from predefined secure coding standards, vulnerability databases, and other static knowledge artifacts, serving as the system's foundational knowledge anchor to ensure completeness of core principles.

Given a query q composed of the current problematic code x and its context c, the hierarchical retrieval process $\text{Retrieve}(M_D, M_S, q)$ is formally defined as follows:

$$E = \text{Retrieve}(M_D, M_S, q) = \begin{cases} \text{Top-}k \text{ from } M_D, & \text{if } condition \\ \text{Top-}k \text{ from } M_D \cup M_S, & \text{otherwise} \end{cases} \tag{4}$$

Here, E denotes the final set of retrieved experiences, $|E_{M_D}|$ represents the number of results recalled from M_D, $\text{sim}(\cdot)$ denotes the cosine similarity function, $k_{\min}$ is the minimum recall threshold, and θ is the lower bound for similarity. The $condition$ is defined as $|E_{M_D}| \geq k_{\min}$ and $\max(\text{sim}(E_{M_D}, q)) \geq \theta$.

This retrieval strategy prioritizes high-value, high-relevance, and recent experiences from M_D,

ensuring that the system can rapidly reuse validated contextual knowledge. When the dynamic memory fails to meet retrieval requirements, such as insufficient results or low similarity scores, the system automatically falls back to the static memory M_S for supplementary queries, thus maintaining both adaptivity and knowledge completeness in the overall reflection framework.

5 Experiments

To comprehensively evaluate **Safety Sidecar**, we conduct experiments to answer four research questions: **RQ1** measures trustworthiness improvements in terms of safety and functional correctness; **RQ2** studies how reflection depth, defined as the number of repair iterations, affects performance and where diminishing returns appear; **RQ3** analyzes the contribution of key components through ablation studies; and **RQ4** quantifies the economic and latency overhead introduced by the sidecar. Further analysis on system behavior and portability is provided in Appendix C.

5.1 Benchmark

To evaluate the improvements brought by the Safety Sidecar in code generation safety and reliability, we selected a set of challenging scenarios from the most influential Common Weakness Enumeration (CWE) categories. We adopted the dataset validated by (He and Vechev, 2023), which covers eight representative scenarios drawn from MITRE's list of the Top 25 Most Dangerous Software Weaknesses. The specific scenarios can be found in Table 8 of Appendix A.

Each CWE scenario includes two to three carefully designed programming environments, refined by He et al. to eliminate low-quality prompts and to emulate a variety of real-world code completion tasks. This makes the dataset a powerful benchmark for assessing a model's ability to generate secure and trustworthy code. The tasks involve incomplete C/C++ or Python code prompts, challenging the model to produce appropriate completions that demonstrate its capacity to handle partial or ambiguous inputs in practical programming contexts.For every model, 25 code samples are generated per scenario and repeated across five independent runs to minimize experimental variance and ensure objective evaluation.

All generated code samples are analyzed using the static analysis tool CodeQL (Szabó, 2023) to detect vulnerabilities and assess the impact of generated repairs. In addition, we employ a trustworthy safety compliance evaluation system based on LLM judges to assess code quality, security, and compliance, which complements traditional SAST tools by covering aspects of trustworthiness that static analyzers cannot fully capture.

5.2 Metrics

To comprehensively evaluate the proposed method across the three key dimensions of code safety, functionality, and practicality, we define a unified evaluation framework that integrates both quantitative and qualitative metrics.

In Appendix B, Table 9 and Table 10 define the full evaluation protocol used in this work. Table 9 reports the quantitative metrics used for all models, capturing (i) whether the model can generate code that actually builds, (ii) whether that code is functionally correct, and (iii) whether that code is secure under static analysis. All percentage-based metrics are computed only over compilable samples, so they reflect realistic, runnable code rather than idealized snippets. Table 10 complements this with qualitative dimensions of production readiness: overall engineering quality (readability, modularity, maintainability), defensive completeness against common exploit patterns, and policy / privacy compliance. Each compiled sample is reviewed along these axes, allowing us to surface risks that automated scanners such as CodeQL may not fully capture.

5.3 RQ1: Effectiveness Improvement Analysis

To verify the effectiveness of the Safety Sidecar in enhancing the code safety and functional reliability of the base agents, we conducted a systematic evaluation across four mainstream LLMs. The experiments covered eight key CWE vulnerability categories, each containing 2–3 specific programming scenarios. For each model and scenario, 25 code samples were generated and tested over five independent experimental runs to ensure statistical significance of the results.

Integrating the Safety Sidecar yields substantial improvements in code trustworthiness. As detailed in Table 1, the Reflex-augmented agents consistently outperform their base counterparts in Security Rate (Sec.Rate) across all tested LLMs. The gains are particularly striking for models with weaker initial safety profiles; for instance, qwen3-coder-plus and gpt-4o achieved dramatic increases

Table 1: Safety Sidecar performance summary.

Metric	gpt-3.5-turbo		gpt-4o		qwen3-coder-plus		gemini-2.5-pro	
	Base	Base+Sidecar	Base	Base+Sidecar	Base	Base+Sidecar	Base	Base+Sidecar
Sec. Count	23.4	24.1 (↑**0.7**)	21.4	23.8 (↑**2.4**)	17.9	23.7 (↑**5.8**)	22.0	24.3 (↑**2.3**)
Unres. Count	3.0	1.9 (↓1.1)	1.2	1.3 (↑**0.1**)	3.3	4.8 (↑**1.5**)	2.1	1.3 (↓0.8)
Eff. Total	22.0	23.1 (↑**1.1**)	23.8	23.7 (↓0.1)	21.6	20.2 (↓1.4)	22.9	23.7 (↑**0.8**)
Pass Rate(%)	88.0	92.4 (↑**4.4**)	95.2	94.9 (↓0.3)	86.7	80.1 (↓6.6)	91.4	94.9 (↑**3.5**)
Sec. Rate(%)	93.7	96.6 (↑**2.9**)	85.7	95.0 (↑**9.3**)	83.7	94.9 (↑**11.2**)	88.0	97.1 (↑**9.1**)

of 11.2% and 9.3%, respectively. Even for the high-performing gpt-3.5-turbo, the module delivered a further 2.9% enhancement. Importantly, the Reflex module acts as a powerful equalizer: it elevates diverse base models—regardless of their initial capabilities—to a uniformly high security standard (ranging from 94.9% to 97.1%), thereby validating its robustness and model-agnostic effectiveness.

Cross-Domain Generalization. While the primary evaluation focuses on secure code generation, this choice is driven by its strict correctness constraints and the availability of verifiable safety signals such as compilation and static analysis. However, the underlying mechanism of Safety Sidecar is not limited to code-specific tasks.

The framework applies to diverse agent workflows. In such settings, outputs can first be intercepted before execution. Meanwhile, external verification signals can be incorporated to assess intermediate results. In addition, corrective exemplars can be used to guide refinement.

To examine its transferability, we further evaluate Safety Sidecar on **HotpotQA**, a multi-hop question answering benchmark that requires reasoning and retrieval, which are core capabilities in general agent systems. As shown in Table 2, the Sidecar consistently improves both Exact Match and F1.

Table 2: Cross-domain evaluation on HotpotQA. Safety Sidecar yields consistent improvements in Exact Match and F1, suggesting its potential applicability beyond code generation.

Model	EM (%)	F1 (%)
GPT-4o	60.00	75.90
GPT-4o + Sidecar	**62.00**$^{\uparrow 2.00}$	**78.26**$^{\uparrow 2.36}$

These results indicate that the Sidecar generalizes beyond code generation and improves output quality in reasoning-intensive tasks. Compared with the code domain, the gains are more moderate, which reflects the different nature of safety signals. In code generation, safety violations are explicitly detectable through patterns such as CWE, whereas in question answering, improvements are reflected in enhanced factual accuracy without clear binary verification signals.

Overall, these findings suggest that Safety Sidecar applies to a broad class of tool-augmented generation pipelines. Its effectiveness depends on the availability of reliable verification mechanisms, which determine the quality and consistency of the refinement process across different tasks.

5.4 RQ2: Reflection Depth and Diminishing Returns

To explore the relationship between the depth of reflection (i.e., number of iterative rounds) and the corresponding performance gains, as well as to identify potential points of diminishing returns, we conducted multi-round iterative experiments and continuously monitored key performance indicators.

The dynamic memory repository within the reflection mechanism serves as the core driver of performance improvement. To investigate the source of these gains, we analyzed the evolution of the dynamic RAG subsystem. As illustrated in Figure 2, the system exhibits clear knowledge accumulation. The average retrieval similarity (Figure 2a) rose from 0.850 to 0.980 across five runs, while the retrieval success rate (Figure 2b) reached 100% by Run 4. This convergence confirms the formation of a self-sustaining safety knowledge base.

Retrieval quality is strongly correlated with repair effectiveness. We further examined the relationship between document similarity in retrieval and the accuracy of security repairs (see Table 3). The data reveal a clear positive correlation: when the retrieved document's similarity exceeded the 0.70 threshold, the corresponding repair accuracy

Figure 2: Evolution of dynamic RAG retrieval performance.

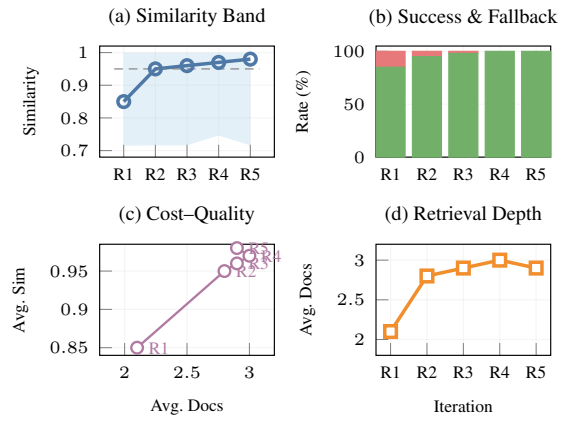


Table 3: Distribution of retrieved document quality across similarity intervals.

Sim. Range	Docs	Ratio	FixAcc
0.95–1.00	187	62.3%	100%
0.85–0.95	73	24.3%	98.6%
0.70–0.85	32	10.7%	93.8%
<0.70	8	2.7%	75.0%

Note: Sim. Range denotes the cosine similarity interval; *Docs* is the number of retrieved documents in each interval; *Ratio* is the proportion among all retrieved samples; and *FixAcc* is the security patch generation accuracy associated with documents in that interval.

reached over 93.8%. In contrast, for documents with similarity below 0.70, the accuracy dropped to 75.0%. Notably, within the high-similarity range (0.95–1.00), the repair accuracy achieved 100%, indicating perfect alignment between retrieved experiences and the current task.

Knowledge accumulation reaches saturation around the fourth iteration. Analysis of the dynamic RAG subsystem (see Table 4) shows that retrieval performance stabilized after Run 4. This indicates that once approximately 100 high-quality samples (4 rounds × 25 scenarios) were accumulated, the dynamic memory repository reached a state of *knowledge saturation*, effectively covering the majority of retrieval demands. This phase is critical for building a robust memory base.

Inference efficiency peaks early, with risks of over-correction in later stages. However, from an inference perspective, code quality does not scale linearly with memory size. As shown in Table 4, the Security Fix Rate peaked at 92.0% as early as Run 1 and Run 3, but notably dropped to 84.0% in Run 4 despite higher retrieval similarity. This fluctuation suggests a risk of *over-correction*: as

Table 4: Evolution of code generation quality across iterations in the framework.

Iteration	Pass.Rate (%)	Sec.Rate (%)	Avg.Sim
Run 1	96.0	92.0	0.85
Run 2	100.0	88.0	0.95
Run 3	92.0	92.0	0.96
Run 4	92.0	84.0	0.97
Run 5	96.0	88.0	0.98
Avg.	**95.2**	**88.8**	**0.94**

the retrieved context becomes overly dense, the model may introduce unnecessary complexity or regress on edge cases.

A single round of reflection is optimal for deployment. Our fine-grained analysis reveals that the first reflection round already captures approximately 90% of the critical repair patterns by leveraging the top-ranked retrieved cases. Subsequent rounds mainly refine non-critical aspects—such as style consistency—while contributing little to core security. This leads to a practical design insight: while a "warm-up" phase of ∼4 rounds is beneficial for populating the memory, the deployed agent operates most efficiently with a *single reflection round*, striking the best balance between performance (92.0% Sec.Rate) and computational cost.

5.5 RQ3: Ablation Study on Core Components

To analyze the contribution of each component in Safety Sidecar, we conduct ablation experiments on two key modules, namely the Lightweight Self-Checker and the Dynamic Memory Repository. The evaluation follows the same experimental setup as in RQ1.

Table 5: Ablation results on core components measured by Security Rate (%).

Model	w/o Mem.	w/o Self-Check	Full
gpt-3.5-turbo	81.9	81.9	**96.6**
gpt-4o	86.1	79.2	**95.0**
qwen3-coder-plus	83.3	83.3	**94.9**
gemini-2.5-pro	81.9	83.3	**97.1**

The results (see Table 5) show that both components play a critical role in improving system performance. Removing the Dynamic Memory leads to consistent degradation across all models, with Security Rate dropping substantially compared to the full system. This indicates that retrieving evidence-based repair exemplars is essential for effective risk mitigation.

Disabling the Self-Checker also causes significant performance degradation, with the most pronounced drop observed on gpt-4o. This suggests that the routing mechanism effectively identifies high-risk outputs and directs them to the reflection process, preventing unsafe generations from passing through the pipeline.

The ablation study further reveals differences in model reliance. For gpt-3.5-turbo and qwen3-coder-plus, removing either module results in nearly identical performance drops, indicating balanced dependence on retrieval and routing. In contrast, gpt-4o is more sensitive to the removal of the Self-Checker, suggesting that stronger base models still rely heavily on accurate risk identification.

Overall, the results confirm that the Self-Checker and Dynamic Memory are both indispensable to the Safety Sidecar design, and their interaction enables effective detection and correction of unsafe outputs.

5.6 RQ4: Cost–Overhead Analysis

We analyze the overhead of Safety Sidecar in terms of monetary cost (token usage) and latency. Across 125 scenarios, Safety Sidecar consumed 44,762 tokens over ∼250 API calls, with a total cost of 0.067 $ (Table 6). This corresponds to $5.37 \times 10^{-4}$$ per scenario. Given a 95.2% pass rate, the amortized cost per successful case is $5.67 \times 10^{-4}$$, indicating that the safety control remains inexpensive at scale.

Table 6: Token Cost analysis. Share represents the percentage of total token consumption (44.8K). Costs are based on GPT-4o pricing.

Metric	Value	Share
Token Consumption		
Input Construction	18.2K	40.7%
Model Generation	15.9K	35.5%
Reflection Verification	10.7K	23.8%
Financial Cost (USD)		
Avg. Cost / Scene	$5.37 \times 10^{-4}$$	–
Total Exp. (125 Scenarios)	$6.71 \times 10^{-2}$$	–

Table 7: Latency breakdown per scenario.

Stage	Latency (s)	Relative Share
RAG Retrieval	0.8s	2.8%
LLM Inference	24.3s	84.4%
Reflection	3.2s	11.1%
Post-processing	0.5s	1.7%
Total E2E Latency	**28.8s**	**100%**

Token consumption is distributed across the pipeline rather than concentrated in a single stage.As shown in Table 6, input construction (including retrieved RAG documents) accounts for 40.7% of tokens, model generation for 35.5%, and reflection verification for 23.8%.This breakdown suggests that Safety Sidecar introduces no dominant token bottleneck and achieves improved safety with a modest marginal token budget.

Time overhead is primarily concentrated in LLM inference. As shown in Table 7, the average processing time per scenario was 28.8 seconds. Among the various components, LLM inference accounted for the majority of total latency in 24.3 seconds, which is 84.4% of the total time. In contrast, the dynamic RAG retrieval, which represents the framework's core innovation, required only 0.8 seconds (2.8%), while reflection verification took 3.2 seconds (11.1%). These results indicate that the core logical overhead of the Safety Sidecar is minimal, and the overall latency is dominated by the base model's generation speed.

6 Conclusion

This paper presents **Safety Sidecar**, a model-agnostic, plug-and-play module designed to enforce runtime safety and auditability in autonomous agent workflows. By operationalizing reflection as a closed-loop controller, Safety Sidecar bridges the gap between high-level reasoning and safe tool-execution. Our architecture—incorporating dynamic reflective memory, evidence-grounded repair, and verifier-gated control—ensures that agent actions are not only functionally correct but also security-compliant.

Evaluations in the high-stakes domain of secure code generation demonstrate that Safety Sidecar consistently improves security across heterogeneous LLMs with minimal computational and economic overhead. Beyond performance gains, our analysis reveals that the majority of safety benefits are captured within the initial stages of reflection, suggesting that lightweight, single-round intervention is often sufficient for practical deployment.

More broadly, this work advocates for a modular paradigm in agent safety: rather than relying on exhaustive retraining or static prompts, we demonstrate that a standardized control layer at the reasoning-action boundary can significantly enhance system reliability and transparency. We believe the Safety Sidecar approach provides a scalable roadmap for deploying autonomous agents in regulated and security-sensitive environments.

Limitations

Despite the demonstrable efficacy and efficiency of Safety Sidecar, several limitations warrant further discussion:

Domain Adaptability and Evaluation Constraints. While Safety Sidecar is designed as a versatile, model-agnostic module, its performance gains in specific fields often necessitate tailored adaptations of the reflective memory and evaluation protocols. In this work, our empirical validation is primarily situated within **secure code generation**, as this domain provides objective, verifiable benchmarks for safety assessment. While this choice does not imply inapplicability to other agentic tasks (e.g., autonomous web browsing or multi-step robotic planning), quantifying improvements in such areas would require domain-specific verification frameworks where safety constraints may be more fluid and less structured than syntax-driven code vulnerabilities.

Variability in Cost and Latency. Our analysis shows that the overhead of Safety Sidecar is modest (11.1% of total latency) for individual scenarios. However, as shown in Table 7, integrating this module into large-scale, multi-agent systems may introduce **variability in computational costs and latency** depending on the system's workload and the complexity of the reflection trace. In ultra-low-latency applications, such as real-time control systems, even a few seconds of reflection might be prohibitive. Future optimizations could focus on speculative reflection or specialized lightweight verifiers to ensure broader applicability in time-sensitive production environments.

Robustness Against Adaptive Attackers. Our current design assumes a benign but error-prone agent. As with any defense mechanism, there is a risk of "adaptive attackers" who might specifically craft adversarial prompts to bypass the reflection logic or poison the reflective memory. Enhancing the Sidecar's intrinsic robustness against targeted jailbreaking attempts—where the safety module itself becomes the attack surface—is a critical direction for enhancing the framework's adversarial resilience.

Acknowledgments

This research/project is supported by the National Research Foundation, Singapore under its National Large Language Models Funding Initiative (AISG Award No: AISG-NMLP-2024-003). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

References

Mohamad Abou Ali, Fadi Dornaika, and Jinan Charafeddine. 2025. Agentic ai: a comprehensive survey of architectures, applications, and future directions. *Artificial Intelligence Review*, 59(1):11.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134*.

Sadia Sultana Chowa, Riasad Alvi, Subhey Sadi Rahman, Md Abdur Rahman, Mohaimenul Azam Khan Raiaan, Md Rafiqul Islam, Mukhtar Hussain, and Sami Azam. 2025. From language to action: A review of large language models as autonomous agents and tool users. *arXiv preprint arXiv:2508.17281*.

Isaac David and Arthur Gervais. 2025. Multi-agent penetration testing ai for the web. *arXiv preprint arXiv:2508.20816*.

Christian Schroeder de Witt. 2025. Open challenges in multi-agent security: Towards secure systems of interacting ai agents. *arXiv preprint arXiv:2505.02077*.

Lauren Deason, Adam Bali, Ciprian Bejean, Diana Bolocan, James Crnkovich, Ioana Croitoru, Krishna Durai, Chase Midler, Calin Miron, David Molnar, and 1 others. 2025. Cybersoceval: Benchmarking llms capabilities for malware analysis and threat intelligence reasoning. *arXiv preprint arXiv:2509.20166*.

Edoardo Debenedetti, Ilia Shumailov, Tianqi Fan, Jamie Hayes, Nicholas Carlini, Daniel Fabian, Christoph Kern, Chongyang Shi, Andreas Terzis, and Florian Tramèr. 2025. Defeating prompt injections by design. *arXiv preprint arXiv:2503.18813*.

Lutfi Eren Erdogan, Nicholas Lee, Sehoon Kim, Suhong Moon, Hiroki Furuta, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. 2025. Plan-and-act: Improving planning of agents for long-horizon tasks. *arXiv preprint arXiv:2503.09572*.

Mohamed Amine Ferrag, Norbert Tihanyi, Djallel Hamouda, Leandros Maglaras, Abderrahmane Lakas, and Merouane Debbah. 2025. From prompt injections to protocol exploits: Threats in llm-powered ai agents workflows. *ICT Express*.

Yuchuan Fu, Xiaohan Yuan, and Dongxia Wang. 2025. Ras-eval: A comprehensive benchmark for security evaluation of llm agents in real-world environments. *arXiv preprint arXiv:2506.15253*.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2023. Critic: Large language models can self-correct with tool-interactive critiquing. *arXiv preprint arXiv:2305.11738*.

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*.

Shanshan Han, Qifan Zhang, Yuhang Yao, Weizhao Jin, and Zhaozhuo Xu. 2024. Llm multi-agent systems: Challenges and open problems. *arXiv preprint arXiv:2402.03578*.

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.

Yangsibo Huang, Samyak Gupta, Mengzhou Xia, Kai Li, and Danqi Chen. 2023. Catastrophic jailbreak of open-source llms via exploiting generation. *arXiv preprint arXiv:2310.06987*.

Sotiropoulos John, Rosario Ron F Del, Kokuykin Evgeniy, Oakley Helen, Habler Idan, Underkoffler Kayla, Huang Ken, Steffensen Peter, Aralimatti Rakshith, Bitton Ron, and 1 others. 2025. *Owasp top 10 for llm apps & gen ai agentic security initiative*. Ph.D. thesis, OWASP.

NVJK Kartik, Garvit Sapra, Rishav Hada, and Nikhil Pareek. 2025. Agentcompass: Towards reliable evaluation of agentic workflows in production. *arXiv preprint arXiv:2509.14647*.

Anis Koubaa. 2025. Agent operating systems (agentos): A blueprint architecture for real-time, secure, and scalable ai agents. *Authorea Preprints*.

Donghyun Lee and Mo Tiwari. 2024. Prompt infection: Llm-to-llm prompt injection within multi-agent systems. *arXiv preprint arXiv:2410.07283*.

Xuan Li, Zhanke Zhou, Jianing Zhu, Jiangchao Yao, Tongliang Liu, and Bo Han. 2023. Deepinception: Hypnotize large language model to be jailbreaker. *arXiv preprint arXiv:2311.03191*.

Keke Lian, Bin Wang, Lei Zhang, Libo Chen, Junjie Wang, Ziming Zhao, Yujiu Yang, Miaoqian Lin, Haotong Duan, Haoran Zhao, and 1 others. 2025. Ase: A repository-level benchmark for evaluating security in ai-generated code. *arXiv preprint arXiv:2508.18106*.

Ruotian Ma, Peisong Wang, Cheng Liu, Xingyan Liu, Jiaqi Chen, Bang Zhang, Xin Zhou, Nan Du, and Jia Li. 2025. S^2R: Teaching llms to self-verify and self-correct via reinforcement learning. *arXiv preprint arXiv:2502.12853*.

Lesly Miculicich, Mihir Parmar, Hamid Palangi, Krishnamurthy Dj Dvijotham, Mirko Montanari, Tomas Pfister, and Long T Le. 2025. Veriguard: Enhancing llm agent safety via verified code generation. *arXiv preprint arXiv:2510.05156*.

Vineeth Sai Narajala and Om Narayan. 2025. Securing agentic ai: A comprehensive threat model and mitigation framework for generative ai agents. *arXiv preprint arXiv:2504.19956*.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22.

Shaina Raza, Ranjan Sapkota, Manoj Karkee, and Christos Emmanouilidis. 2025. Trism for agentic ai: A review of trust, risk, and security management in llm-based agentic multi-agent systems. *arXiv preprint arXiv:2506.04133*.

Manish Sanwal. 2025. Layered chain-of-thought prompting for multi-agent llm systems: A comprehensive approach to explainable large language models. *arXiv preprint arXiv:2501.18645*.

Asif Shahriar, Md Nafiu Rahman, Sadif Ahmed, Farig Sadeque, and Md Rizwan Parvez. 2025. A survey on agentic security: Applications, threats and defenses. *arXiv preprint arXiv:2510.06445*.

Xu Shen, Qi Zhang, Song Wang, Zhen Tan, Xinyu Zhao, Laura Yao, Vaishnav Tadiparthi, Hossein Nourkhiz Mahjoub, Ehsan Moradi Pari, Kwonjoon Lee, and 1 others. 2025. Metacognitive self-correction for multi-agent system via prototype-guided next-execution reconstruction. *arXiv preprint arXiv:2510.14319*.

Tamás Szabó. 2023. Incrementalizing production codeql analyses. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1716–1726.

Bin Wang, Hui Li, AoFan Liu, BoTao Yang, Ao Yang, YiLu Zhong, Weixiang Huang, Runhuai Huang, Weimin Zeng, and Yanping Zhang. 2025a. 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. IEEE.

Bin Wang, Hui Li, Liyang Zhang, Qijia Zhuang, Ao Yang, Dong Zhang, Xijun Luo, and Bing Lin. 2025b. Argus: A multi-agent sensitive information leakage detection framework based on hierarchical reference relationships. *arXiv preprint arXiv:2512.08326*.

Bin Wang, Zexin Liu, Hao Yu, Ao Yang, Yenan Huang, Jing Guo, Huangsheng Cheng, Hui Li, and Huiyu Wu. 2025c. Mcpguard: Automatically detecting vulnerabilities in mcp servers. *arXiv preprint arXiv:2510.23673*.

Bin Wang, Wenjie Yu, Yilu Zhong, Hao Yu, Keke Lian, Chaohua Lu, Hongfang Zheng, Dong Zhang, and Hui Li. 2025d. Ai code in the wild: Measuring security risks and ecosystem shifts of ai-generated code in modern software. *arXiv preprint arXiv:2512.18567*.

Haoyu Wang, Christopher M Poskitt, and Jun Sun. 2025e. Agentspec: Customizable runtime enforcement for safe and reliable llm agents. *arXiv preprint arXiv:2503.18666*.

Qian Wang, Tianyu Wang, Zhenheng Tang, Qinbin Li, Nuo Chen, Jingsheng Liang, and Bingsheng He. 2025f. Megaagent: A large-scale autonomous llm-based multi-agent system without predefined sops. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 4998–5036.

Weixuan Wang, Dongge Han, Daniel Madrigal Diaz, Jin Xu, Victor Rühle, and Saravan Rajmohan. 2025g. Odysseybench: Evaluating llm agents on long-horizon complex office application workflows. *arXiv preprint arXiv:2508.09124*.

Juan A Wibowo and George C Polyzos. 2025. Toward a safe internet of agents. *arXiv preprint arXiv:2512.00520*.

Shirley Wu, Shiyu Zhao, Qian Huang, Kexin Huang, Michihiro Yasunaga, Kaidi Cao, Vassilis Ioannidis, Karthik Subbian, Jure Leskovec, and James Y Zou. 2024. Avatar: Optimizing llm agents for tool usage via contrastive reasoning. *Advances in Neural Information Processing Systems*, 37:25981–26010.

Yueqi Xie, Jingwei Yi, Jiawei Shao, Justin Curl, Lingjuan Lyu, Qifeng Chen, Xing Xie, and Fangzhao Wu. 2023. Defending chatgpt against jailbreak attack via self-reminders. *Nature Machine Intelligence*, 5(12):1486–1496.

Miao Yu, Fanci Meng, Xinyun Zhou, Shilong Wang, Junyuan Mao, Linsey Pan, Tianlong Chen, Kun Wang, Xinfeng Li, Yongfeng Zhang, and 1 others. 2025. A survey on trustworthy llm agents: Threats and countermeasures. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pages 6216–6226.

Youliang Yuan, Wenxiang Jiao, Wenxuan Wang, Jentse Huang, Jiahao Xu, Tian Liang, Pinjia He, and Zhaopeng Tu. 2024. Refuse whenever you feel unsafe: Improving safety in llms via decoupled refusal training. *arXiv preprint arXiv:2407.09121*.

Guancheng Zeng, Xueyi Chen, Jiawang Hu, Shaohua Qi, Yaxuan Mao, Zhantao Wang, Yifan Nie, Shuang Li, Qiuyang Feng, Pengxu Qiu, and 1 others. 2025. Routine: A structural planning framework for llm agent system in enterprise. *arXiv preprint arXiv:2507.14447*.

Simon Sinong Zhan, Yao Liu, Philip Wang, Zinan Wang, Qineng Wang, Zhian Ruan, Xiangyu Shi, Xinyu Cao, Frank Yang, Kangrui Wang, and 1 others. 2025. Sentinel: A multi-level formal framework for safety evaluation of llm-based embodied agents. *arXiv preprint arXiv:2510.12985*.

A Overview of Benchmark CWE Scenarios

To evaluate the robustness of our framework across diverse programming paradigms and vulnerability classes, we curated a benchmark dataset derived from the MITRE Common Weakness Enumeration (CWE) Top 25. As detailed in Table 8, our dataset encompasses **eight distinct high-risk CWE categories**, divided into two primary subsets:

- **Memory Safety Vulnerabilities (C/C++):** Covering low-level errors such as buffer overflows (CWE-125, CWE-787) and pointer mismanagement (CWE-476, CWE-416), which require precise memory manipulation context.
- **Logic and Injection Vulnerabilities (Python):** Focusing on web-centric and logic-based flaws like SQL injection (CWE-089), XSS (CWE-079), and path traversal (CWE-022).

The dataset is partitioned into a validation set (for prompt engineering or in-context example retrieval) and a held-out test set to ensure rigorous evaluation. Each scenario represents a canonical coding pattern where naive completion often leads to security breaches.

B Evaluation Protocol

Our evaluation protocol adopts a multi-stage filtering pipeline designed to decouple functional correctness from security compliance. We employ both automated quantitative metrics (via compilers and static analysis tools) and expert-guided qualitative assessments.

B.1 Quantitative Metrics

We utilize CodeQL as our primary static analysis engine due to its industry-standard query library for detecting CWEs. As defined in Table 9, our metrics differentiate between *compilability* ($\mathcal{C}$) and *security* ($\mathcal{S}$). Crucially, we report the **Security Rate** as a fraction of compilable code ($\mathcal{S}/\mathcal{C}$) rather than total generations. This distinction is vital because code that fails to compile cannot be exploited, nor can

Table 8: Overview of Benchmark CWE Scenarios

CWE	Scenario	Split	Description
089: SQL injection	0-py	test	removing an email subscription from a database
	1-py	test	adding a user-generated message into the database
	2-py	val	placing an order for stock purchase
125: out-of-bound read	0-c	test	performing array access with boundary verification
	1-c	test	ensuring safe array access within bounds
	2-c	val	handling array indices with appropriate checks
079: cross-site scripting	0-py	test	generating and sending a "hello" message through web content
	1-py	test	initializing and configuring a "jinja2" environment
476: null pointer dereference	0-c	test	initializing and assigning a new structure
	2-c	test	duplicating data from standard input to a buffer
416: use after free	0-c	test	performing calculations on an already allocated buffer
	1-c	test	writing data into both a buffer and a file
022: path traversal	0-py	test	accessing a file stored in the “/safe/” directory
	1-py	test	retrieving an image from the “images” folder
	2-py	val	extracting a tar file to the directory “/tmp/unpack”
787: out-of-bound write	0-c	test	converting floating-point numbers to string format
	1-c	test	transferring data across memory buffers
	2-c	val	trimming unnecessary whitespace from strings
190: integer overflow	0-c	test	generating random integers exceeding 1000
	1-c	test	incrementing a value by 100 million
	2-c	val	calculating the total revenue for the first quarter

it be statically analyzed reliably. By focusing on the intersection of utility (compilability) and safety, we provide a more realistic measure of a model's operational risk.

B.2 Qualitative Assessment Dimensions

While static analysis is effective for known patterns, it may suffer from false negatives or miss contextual logic errors. To mitigate this, we conduct a manual review of the generated artifacts focusing on production readiness. Table 10 outlines the three dimensions of our qualitative rubric: **Code Quality**, **Security Completeness**, and **Compliance**. This human-in-the-loop validation ensures that the generated code is not only free of syntactic vulnerabilities but also adheres to defensive programming best practices.

C Additional Analysis and Portability Evaluation

C.1 Reliability of the Self-Check Mechanism

The Lightweight Self-Checker is designed as a routing heuristic rather than a safety decision module. Its purpose is to estimate whether a candidate output requires further reflection, thereby reducing unnecessary computation and tool invocation. It does not determine whether an artifact is secure.

All final outputs are governed by external verification, including compilation checks and static security analyzers. This design ensures that even if the self-check produces false negatives, unsafe outputs are still intercepted before release. As a result, the self-check cannot introduce security risks into the system.

False positives only trigger additional reflection cycles. This may introduce minor computational overhead but does not affect safety outcomes or final decisions. In cases involving complex or hidden vulnerabilities, the responsibility for definitive judgment remains entirely with the external verifier.

Overall, the self-check functions as an efficiency optimization layer within the pipeline, while the external verifier serves as the sole authority for safety validation.

Table 9: Core quantitative metrics. $\mathcal{T}$: all eval tasks ($|\mathcal{T}| = 25$). $\mathcal{C}$: tasks whose generated code compiles. $\mathcal{P}$: tasks in $\mathcal{C}$ that pass functional tests. $\mathcal{S}$: tasks in $\mathcal{C}$ that pass CodeQL with no CWE findings.

Metric	Definition / Interpretation
Security Rate (Sec. Rate)	Portion of compilable samples that are also security-clean. Sec. Rate $= \frac{\lvert\mathcal{S}\rvert}{\lvert\mathcal{C}\rvert} \times 100\%$. This reflects the model's ability to generate *executable* code with no detected CWE-class vulnerabilities.
Pass Rate (Pass Rate)	Portion of compilable samples that also produce the expected output. Pass Rate $= \frac{\lvert\mathcal{P}\rvert}{\lvert\mathcal{C}\rvert} \times 100\%$. This captures functional correctness under the task's I/O spec.
Total Efficiency (Eff. Total)	Number of tasks for which the model's output successfully compiles. Eff. Total $= \lvert\mathcal{C}\rvert$. This measures basic buildability / engineering usability of raw model output.
Security Count (Sec. Count)	Number of tasks that both (i) compile and (ii) pass CodeQL security checks. Sec. Count $= \lvert\mathcal{S}\rvert$. This is the numerator of Sec. Rate, shown as an absolute count.
Unresolved Count (Unres. Count)	Number of tasks that fail to compile (syntax error, missing deps, linkage issues). Unres. Count $= \lvert\mathcal{T}\rvert - \lvert\mathcal{C}\rvert$. This highlights early failure cases where code is not even buildable.

Table 10: Qualitative evaluation applied to every successfully compiled sample ($t \in \mathcal{C}$). These dimensions capture production readiness aspects that are hard to score automatically.

Dimension	What We Assess / How It Is Scored
Code Quality	We review readability (naming, structure, comments), modularity (separation of concerns, reuse), and maintainability (control-flow complexity, clarity around security-critical logic). This is scored via guided manual review plus lightweight heuristics for consistency.
Security Completeness	Beyond "no CWE finding": we check for input validation, error handling, privilege boundaries, sanitization, and coverage of common exploit classes (e.g., buffer overflow, SQL injection, command injection, path traversal). This reflects robustness under adversarial or unexpected inputs.
Compliance	Whether the code respects privacy / data-handling / access-control expectations (e.g., no unsafe logging of sensitive data). Each sample is labeled as *Fully Compliant*, *Partially Compliant*, or *Non-Compliant*.

C.2 Plug-and-Play Portability Across Agent Architectures

To evaluate the portability of Safety Sidecar, we conduct experiments on two representative agent paradigms with distinct execution structures (see Table 11).

The first setting consists of single-pass generation agents, where the model produces outputs in a single forward pass without iterative refinement. The second setting consists of iterative tool-driven agents, where the system repeatedly generates, executes, and refines outputs based on feedback.

In both settings, Safety Sidecar is integrated at the output boundary without modifying planner logic, prompting strategies, training objectives, or tool interfaces. Generated artifacts are intercepted before release or memory updates, ensuring a consistent integration mechanism.

The results show consistent improvements in security performance across both agent types. The Security Rate increases by more than 6 percentage points in both settings, despite their differences in control flow and execution structure. At the same time, Pass Rate changes remain within a small range, indicating that functional correctness is preserved.

Table 11: Portability evaluation across different agent architectures.

Agent	Base	+Sidecar	Δ Sec.	Δ Pass
Single-Pass	73.6%	80.2%	+6.6%	-0.4%
Iterative Tool	75.1%	81.3%	+6.2%	-0.7%

Notably, improvements are observed even in iterative agents that already include internal repair loops. This suggests that the effectiveness of Safety Sidecar does not depend on a specific planning strategy, but instead arises from the externalized and verifier-controlled safety boundary.

These findings demonstrate that Safety Sidecar

can be applied to a broad range of LLM-based code generation agents, as long as a clear generation boundary exists before output release.