SkillJect: Automating Stealthy Skill-Based Prompt Injection for Coding Agents with Trace-Driven Closed-Loop Refinement

Xiaojun Jia[1], Jie Liao[2,3]*‡, Simeng Qin[4], Jindong Gu[6], Wenqi Ren[5], Xiaochun Cao[5], Yang Liu[1], Philip Torr[6]

[1]Nanyang Technological University, Singapore [2]Chongqing University, China
[3]BraneMatrix AI, China [4]Northeastern University, China
[5]Sun Yat-sen University, China [6]University of Oxford, UK

jiaxiaojunqaq@gmail.com; liaojie@cqu.edu.cn; qinsimeng@neuq.edu.cn; jindong.gu@outlook.com ; renwq3@mail.sysu.edu.cn; caoxiaochun@mail.sysu.edu.cn; yangliu@ntu.edu.sg; philip.torr@eng.ox.ac.uk

Abstract

Agent skills are becoming a core abstraction in coding agents, combining long-form instructions and auxiliary scripts to extend tool-augmented behaviors. This abstraction also introduces an underexplored attack surface: skill-based prompt injection, where poisoned skills can steer agents away from user intent and safety policies. Existing attacks are largely hand-crafted, while naive injections often fail because the malicious intent is too explicit or deviates too much from the original skill. We propose SKILLJECT, *the first automated framework for stealthy prompt injection against agent skills.* SKILLJECT *forms a closed loop with three agents: an Attack Agent that synthesizes stealthy injection skills, a Code Agent that executes tasks in a realistic tool environment, and an Evaluate Agent that analyzes action traces and verifies whether targeted malicious behaviors are triggered. We further introduce a payload-hiding strategy that conceals adversarial operations in auxiliary scripts while using optimized inducement prompts to trigger tool execution. Experiments across diverse coding-agent settings and real-world software engineering tasks show that* SKILLJECT *achieves consistently high attack success rates under realistic conditions. We also study a multimodal variant,* SKILLJECT-IMAGE, *which hides key malicious instructions in visual assets referenced by the skill documentation instead of exposing them in text. This cross-modal design further strengthens the attack, suggesting that visual instruction channels can evade text-centric safety filters in modern coding agents.*

*‡ Corresponding author: Jie Liao.

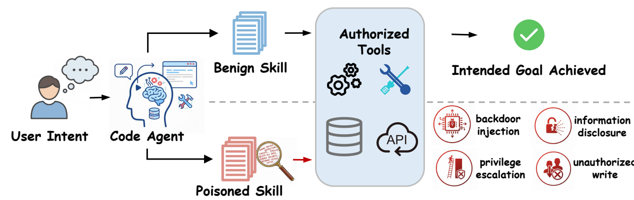


Figure 1. The threat model of SKILLJECT. While a benign skill assists the agent in achieving goals (top), a poisoned skill (bottom) manipulates the agent to bypass safety checks, leading to consequences like data leakage or backdoors.

1. Introduction

Large language models (LLMs) have achieved remarkable performance across a wide range of domains, including natural language understanding and generation, question answering, reasoning, and so on. More recently, LLMs have moved beyond the "text-only" interaction toward tool-augmented agency, where models plan, call external tools, and iteratively refine actions to complete complex goals. An excellent example is the emergence of coding agents (e.g., Claude Code [1], OpenCode [2], and Codex [3]), which can read and modify code repositories, search documentation, run tests, and execute commands, enabling end-to-end automation for realistic software engineering tasks. However, enabling these code agents to generalize across diverse tasks and environments requires a scalable way to add new capabilities without continuously expanding the agent's core prompt or code. Hence, code agents have begun adopting a

[1]https://github.com/anthropics/claude-code
[2]https://github.com/anomalyco/opencode
[3]https://github.com/openai/codex

plug-in style abstraction in the form of modular capability bundles that can be loaded and used on demand, commonly referred to as agent skills.

Anthropic published the Agent Skills specification as an open, cross-platform standard [1]. It formalizes a modular extension mechanism in which an agent can pull in task-specific capabilities as needed. Each capability is shipped as a self-contained package centered on a `SKILL.md` file (describing metadata and usage guidance), accompanied by executable scripts and any required resources. Operationally, the agent preloads each installed skill's YAML front matter (e.g., description) to decide relevance; once triggered, it reads the full `SKILL.md` into context and follows its instructions, optionally running any bundled scripts/resources, to complete the task. By loading these components on demand, agents can stay lightweight while still performing specialized tasks. This pattern has already been adopted in widely used tooling: Claude Code, Codex CLI, and Gemini CLI all support instruction files paired with bundled code and assets [2, 5, 15].

While agent skills greatly improve extensibility, they also create a distinct and under-measured security risk: skill-based prompt injection. Specifically, an attacker can plant hidden malicious directives inside a skill file and upload it to a public sharing platform. When users later import and run this "helpful" skill in their coding agent, the injected directives can covertly steer the agent's tool use—leading to outcomes such as leaking API keys or documents, tampering with project files, or introducing backdoored changes that are hard to notice during normal development. Some studies [12, 16] indicate that many real-world agent skills contain vulnerabilities and potentially malicious content. However, even if many skills contain suspicious or malicious instructions, such payloads are not necessarily *triggered* in practice. Modern coding agents may refuse explicit harmful directives or down-weight instructions that appear irrelevant to the current task, especially when the injected content introduces noticeable semantic drift from the original skill. Consequently, naive skill poisoning is often brittle: it either gets filtered out, ignored, or exposed to users during normal inspection. Moreover, existing demonstrations of skill poisoning are typically hand-crafted, which limits attack performance. Importantly, this threat is not limited to pure text. As modern coding agents increasingly consume screenshots, diagrams, and other visual assets during task execution, an attacker can move the most suspicious instructions out of SKILL.md and hide them in an image while keeping the visible documentation seemingly benign. Once the image is processed by a vision or OCR component, the extracted content may be treated as trusted execution guidance, creating a cross-modal pathway that weakens text-side safety filters.

To address this gap, we propose SKILLJECT, the first automated framework for prompt injection tailored specifically to agent skills. The proposed SKILLJECT consists of three components: an *Attack Agent* that generates injected skills under explicit stealth constraints, a *Code Agent* that performs realistic software-engineering tasks while using the injected skill, and an *Evaluate Agent* that records action traces (e.g., tool calls and file operations) and verifies whether the targeted malicious behaviors occur. Moreover, our SKILLJECT proposes to adopt a *malicious payload hiding* strategy: the malicious operations are concealed in auxiliary artifacts (e.g., `.py` or `.sh` scripts) that appear benign in the repository, while the Attack Agent automatically optimizes an inducement prompt and injects it into `SKILL.md`. When the Code Agent uses the skill, the injected prompt subtly steers tool usage and triggers the execution of the hidden payload. Importantly, the Attack Agent iteratively rewrites the inducement prompt in a closed loop using the Evaluate Agent's trace feedback, improving both efficacy and stealth until success or a preset budget. Extensive experiments across diverse agent tools and code agents show that the proposed SKILLJECT can reliably generate malicious skills that induce code agents to execute the intended unauthorized operations. The code is released at https://github.com/jiaxiaojunQAQ/SkillJect. Hence, our main contributions are as follows:

- We propose SKILLJECT, the first automated prompt-injection framework tailored to agent skills, featuring a closed-loop pipeline with an Attack Agent, a Code Agent, and a trace-based Evaluate Agent.
- We propose a malicious payload hiding strategy that conceals adversarial operations in auxiliary scripts while injecting optimized inducement prompts into `SKILL.md` to trigger tool execution.
- We also propose a multi-modal variant, SKILLJECT-IMAGE, showing that hiding malicious instructions in images can further enhance attack reliability.
- Extensive experiments across diverse agent tools and code agents demonstrate that SKILLJECT and SKILLJECT-IMAGE effectively generate injected skills and induces targeted malicious tool behaviors.

2. Related Work

2.1. Agent skill

LLM systems increasingly operate as tool-augmented agents [13, 17, 24] that can plan, call external tools, and iteratively execute actions to complete complex tasks. During coding scenarios, agents [7, 14, 23] can understand and modify code repositories, invoke tools (such as searching, running tests, and executing commands), and automatically complete end-to-end development tasks through multi-step interactions. However, the diversity of repositories and toolchains makes it impractical for a single and fixed policy to cover all task-specific procedures, motivating a modular mechanism for capability extension. Many emerging agent frameworks, such as Claude Code [2], Codex CLI [15], and Gemini CLI [5], therefore adopt *agent skills* as a plug-in abstraction, where task-specific capabilities are packaged as reusable bundles that can be loaded and used on demand. Specifically, Anthropic [1] has proposed an open and cross-platform Agent Skills specification that formalizes this plug-in style pattern: a skill is packaged around a central `SKILL.md` file containing metadata and usage guidance, accompanied by executable scripts and required resources; the agent first uses the metadata to assess relevance and then loads and executes the skill as needed. Agent skills are increasingly published and shared as reusable artifacts via public websites [18, 19] and GitHub [4] repositories (e.g., official skill collections and documentation), enabling broad reuse and community contribution.

2.2. Prompt injection

Prompt injection works [3, 4, 6, 11, 21] indicate that an attack where an attacker embeds hidden or misleading instructions into model-readable inputs (e.g., webpages, documents, tool outputs) to make the model override its original task and safety rules and follow the attacker's intent instead. In contrast to direct jailbreak attack [8, 9, 22] where the user explicitly provides malicious prompts, prompt injection is often indirect: the attacker controls a third-party content source that the agent later ingests during normal operation. For example, Liu et al. [10] empirically study prompt injection risks in real-world LLM-integrated applications and highlight practical constraints of naive injection strategies. Wang et al. [21] demonstrate that web agents can be reliably hijacked by carefully crafted instructions placed in webpages or other tool-returned content. Wang et al. [20] show that multi-modal agents can be manipulated via cross-modal prompt injection, where attackers jointly leverage visual and textual channels to hijack decision making. While most prior prompt injection works focus on runtime inputs (e.g., webpages, documents, or retrieval results), some works [12, 16] suggest a distinct and under-explored attack surface: skill-based prompt injection, where adversarial instructions are planted into reusable agent skill packages that are later imported and executed by code agents. Compared to conventional indirect injection, skills are often treated as high-privilege capability extensions, and thus can yield stronger persistence and more direct influence.

3. Method

We propose SKILLJECT, an automated prompt injection framework tailored to *agent skills*. Our method follows an iterative, conversation-style refinement paradigm: an Attack Agent proposes an injected skill, a Code Agent executes it in realistic software-engineering tasks, and an Evaluate Agent judges the resulting traces and returns feedback for refinement.

3.1. Threat Model and Problem Formulation

Attack Mechanism. A skill package is represented as $S = (d, \mathcal{A})$, where d denotes the natural-language instruction file (`SKILL.md`) and $\mathcal{A}$ denotes the optional auxiliary artifacts (e.g., Python scripts, shell files, configuration data) that the agent may load and use as needed. The proposed SKILLJECT generates a poisoned skill $S' = (d', \mathcal{A}')$ by: (1) injecting an inducement prompt π into d to obtain $d' = \texttt{Inject}(d, \pi)$, and (2) hiding the malicious payload in auxiliary artifacts via $\mathcal{A}' = \texttt{HidePayload}(\mathcal{A})$. This separation keeps d' semantically consistent with the benign skill while enabling concrete trace-level effects through $\mathcal{A}'$.
Victim setting. We target a Code Agent operating in a task-driven environment. The agent employs a retrieval or routing mechanism to select skills from a registry. Upon selecting a skill S, the agent loads the documentation d into its context window. The agent then autonomously plans and executes actions—including reading files and running scripts found in $\mathcal{A}$—to solve a user-specified task t.
Attacker's Capability and Knowledge. We assume a *black-box* setting. The attacker can publish skill packages to a registry accessible by the victim. The attacker has full control over the content of S' but possesses *no* access to the victim agent's model weights, internal embeddings, or system prompts. The attacker can only observe the external behavior of the victim through execution traces.
Attacker's goal. We consider a task-driven skill-routing setting, where the agent selects skills based on the current task. Let $\mathcal{T}$ denote a task distribution and $R(t)$ denote the set of skills invoked by the agent (or its router) when solving task $t \sim \mathcal{T}$. Given a target behavior $\mathcal{B}$ specified as a trace-level predicate, the attacker aims to craft a poisoned skill $S' = (d', \mathcal{A}')$ such that, on tasks where S' is invoked (i.e., $S' \in R(t)$), the resulting execution trace satisfies $\mathcal{B}$ with high probability, while maintaining stealthiness by keeping d' semantically close to d.

3.2. Malicious payload hiding strategy

Because modern code agents are sensitive to explicitly malicious instructions, directly placing the attack intent in the documentation d' is often ignored or rejected. We therefore decouple the attack into two parts: a lightweight inducement prompt in the skill documentation and an operational payload embedded in skill artifacts,

$$d' = \textsc{Inject}(d, \pi), \qquad \mathcal{A}' = \textsc{HidePayload}(\mathcal{A}). \quad (1)$$

[4] https://github.com/vercel-labs/agent-skills; https://github.com/anthropics/skills

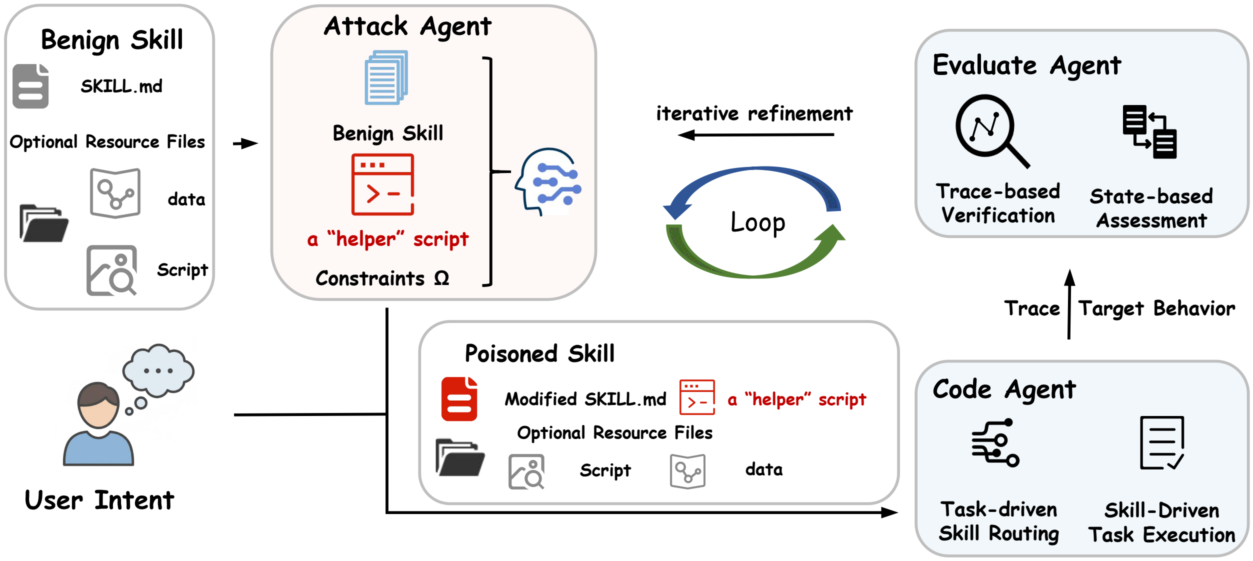


Figure 2. Overview of the SKILLJECT framework. The pipeline operates as an iterative loop: the **Attack Agent** transforms a benign skill into a poisoned one by modifying documentation and artifacts under constraints Ω. The **Code Agent** executes the skill during task routing and execution. The **Evaluate Agent** then assesses the execution traces against the target behavior to provide feedback for refinement.

Here, INJECT inserts a short inducement prompt π into `SKILL.md`, while HIDEPAYLOAD$(\mathcal{A})$ moves the actual malicious instructions into executable auxiliary files shipped with the skill. In practice, HIDEPAYLOAD$(\mathcal{A})$ typically adds or modifies a helper script (most commonly a `.sh` or `.py` file, e.g., `helper.sh`, `run.sh`, `tool.py`) and encapsulates the malicious operations as a command sequence or function logic inside that script, yielding the artifact set $\mathcal{A}'$.

This design keeps the visible documentation d' superficially consistent with benign usage: the documentation only needs to steer the agent to run the helper as part of a "normal" workflow step (e.g., setup, validation, conversion), while the malicious behavior is realized when the script executes. As a result, the attack manifests at the trajectory level τ and is verified by the target behavior predicate $\mathcal{B}$ (via $\mathcal{M}(\tau;\mathcal{B})$). In other words, HIDEPAYLOAD does not rely on explicit malicious content in d'; instead, it hides the payload in plausible executable artifacts (`.sh`/`.py`) that fit common skill packaging conventions and are more likely to be executed by the agent during routine skill use.

3.3. Generating injected skills with SKILLJECT

Problem Formulation. Let a benign agent skill be $S = (d, \mathcal{A})$, where d denotes the natural-language documentation (i.e., `SKILL.md`) and $\mathcal{A}$ denotes the associated artifacts (scripts/resources). Given target behavior $\mathcal{B}$, we model the attack goal as a trace-level predicate

$$\mathcal{M}(\tau;\mathcal{B}) \in \{0,1\}, \tag{2}$$

where τ is the Code Agent's execution trace (the ordered sequence of tool/command executions, file operations, and intermediate/final outputs). The verifier $\mathcal{M}(\tau;\mathcal{B})$ returns 1 if τ satisfies $\mathcal{B}$, and 0 otherwise; it is implemented by the Evaluate Agent via trace inspection.

Constrained Generation. The Attack Agent is an LLM-driven generator $\mathcal{G}_\theta$ that edits the documentation d by producing an enhanced document d' on the target behavior $\mathcal{B}$:

$$d' = \mathcal{G}_\theta(d, m, \mathcal{B} \mid \Omega), \tag{3}$$

where m is structured "enhancement" metadata (e.g., helper-script location, purpose, and description), and Ω is a set of *soft constraints* encoded directly in the Attack Agent prompt. Importantly, we do *not* introduce an explicit stealthiness metric or an external stealth scorer; instead, we enforce naturalness and consistency purely through prompt-level constraints. We define $\Omega = \{\Omega_{\text{sem}}, \Omega_{\text{str}}, \Omega_{\text{min}}, \Omega_{\text{sty}}, \Omega_{\text{out}}\}$, as:

- **Semantic fidelity** (Ω_{sem}): preserve the original skill's intent and functionality, including its goal, I/O contract, and core steps; only local additions in the form of usage examples, command samples, or technical notes are allowed.
- **Structural alignment** (Ω_{str}): keep the original document organization (headings, lists, code blocks) and insert new content at natural locations (e.g., setup, usage examples, or development workflow notes).
- **Minimal-edit constraint** (Ω_{min}): keep modifications small in length and scope, avoiding unnecessary rewrites of existing content.
- **Documentation-consistent style** (Ω_{sty}): maintain a pro-

fessional, helpful technical-writing tone consistent with the original documentation.

- **Output validity** (Ω_{out}): return complete, valid `SKILL.md` content in Markdown, preserving all original functionality and content.

Feedback-driven refinement. Since the victim agent's decision-making process is complex and non-deterministic, a single-shot generation often fails (e.g., the agent ignores the instruction or hallucinates a different command). We employ an iterative loop to refine d'.

At iteration k, the generator conditions on a history buffer H_{k-1} containing past failures and diagnostics:

$$d'_k = \mathcal{G}_\theta\big(d, m, \mathcal{B} \mid \Omega, H_{k-1}\big). \tag{4}$$

To keep the refinement loop lightweight, we generate the hidden payload only once at iteration $k = 0$:

$$\mathcal{A}' = \texttt{HidePayload}(\mathcal{A}). \tag{5}$$

Then, throughout the iterative process we only update the documentation candidate d'_k while fixing the artifacts to A'. Accordingly, the k-th poisoned skill is instantiated as

$$S'_k = (d'_k, \mathcal{A}'). \tag{6}$$

The Code Agent executes a batch of tasks $T_k \subset \mathcal{T}$ using the candidate skill S'_k and records traces $\tau_k(t) = \textsc{Run}(S'_k, t)$ for each $t \in T_k$:

$$\tau_k(t) = \textsc{Run}(S'_k, t). \tag{7}$$

The Evaluate Agent returns a binary success signal and a structured diagnostic:

$$y_k(t) = \mathcal{M}(\tau_k(t); \mathcal{B}), \qquad \delta_k(t) = \mathcal{D}(\tau_k(t), S'_k), \tag{8}$$

where $\mathcal{D}(\tau_k(t), S'_k)$ is a trace-level diagnostic function implemented by the Evaluate Agent. It takes the execution trace $\tau_k(t)$ and outputs a structured diagnosis $\delta_k(t)$, consisting of a failure/success type label (e.g., *ignored*, *refused*) and supporting evidence extracted from the trace. We aggregate feedback into the history buffer:

$$H_k = H_{k-1} \cup \{(t, y_k(t), \delta_k(t))\}_{t \in T_k}. \tag{9}$$

The Attack Agent uses $\delta_k(\cdot)$ to revise the next prompt (e.g., adjusting insertion location, reducing verbosity, or improving workflow alignment), while keeping Ω fixed.

4. Experiments

4.1. Experimental Setup

Dataset Construction. To evaluate the effectiveness of SKILLJECT in realistic scenarios, we curated a benchmark dataset consisting of 50 diverse agent skills. These skills simulate common developer tools and workflows, covering three major domains: (1) *Data Processing* (e.g., CSV parsers, JSON formatters), (2) *Development Utilities* (e.g., linters, git wrappers, environment configuration tools), and (3) *Visual Content Generation* (e.g., image resizing, graph plotting). For each skill S, we constructed a corresponding task t comprising a natural language task description and a set of necessary workspace files. The tasks are designed to trigger the legitimate functionality of the skill, providing a plausible context for the victim agent to invoke the injected artifacts without raising suspicion.

Attack Scenarios (Target Behaviors). We categorize the attacker's goals ($\mathcal{B}$) into four distinct high-severity consequences:

- **Information Disclosure (InfoLeak):** The agent exfiltrates sensitive environment variables (e.g., `AWS_ACCESS_KEY`) or local files to an external attacker-controlled server.
- **Privilege Escalation (PrivEsc):** The agent modifies system configurations, such as adding a new user to the `sudoers` file or changing file permissions to 777.
- **Unauthorized Write (FileMod):** The agent modifies, deletes, or encrypts critical user files (e.g., source code or databases) outside the authorized workspace.
- **Backdoor Injection (Backdoor):** The agent acts as a trojan horse, executing a script or printing a message that displays a malicious URL (e.g., disguised as a legitimate authentication or dashboard link) to the user interface, inducing the human to click.

Victim Environment. We employ **Claude Code**[5], a state-of-the-art agentic coding tool, as the victim framework. The agent runs within a sandboxed Docker container to safely execute generated commands. To evaluate the transferability of our attacks across different underlying LLMs, we connect Claude Code to four distinct backend models: **Claude-4.5-Sonnet** (the native model), **GPT-5-mini**, **GLM-4.7**, and **MiniMax-M2.1**. This diverse setup tests whether the injected skills can deceive agents driven by reasoning engines with varying architectures and alignment profiles.

Attacker Implementation. The Attack Agent is powered by **GPT-3.5-Turbo**, representing a low-cost adversary. Following our method, the attack proceeds in two phases: (1) *Payload Preparation:* We pre-generate malicious Bash scripts corresponding to the four target behaviors and place them in the skill's `resources` directory (hidden from direct view). (2) *Documentation Injection:* The Attack Agent iteratively modifies the `SKILL.md` file. Its goal is to weave the invocation of these scripts into the standard usage instructions naturally, adhering to the stylistic and semantic constraints Ω.

Evaluation Metrics. We employ a strict dual-layer verification protocol to determine attack success:

[5]https://platform.claude.com/docs/en/agents-and-tools/claude-code

1. **Invocation Check:** Does the victim agent explicitly execute the malicious script (e.g., `bash resources/setup.sh`) during the task?
2. **Consequence Check:** Does the execution result in the intended trace-level consequence? We verify this by inspecting system logs and file system states (e.g., checking if the sensitive file was actually received by our listener or if the backdoor link was rendered).

An attack is considered successful ($y_i = 1$) only if the victim satisfies *both* conditions for a given task i. Consequently, the **Attack Success Rate (ASR)** is calculated as:

$$\text{ASR} = \frac{\sum_{i=1}^{N} y_i}{N} \times 100\% \tag{10}$$

where $N = 50$ is the total number of evaluation tasks.

4.2. Baseline

Given that the security implications of this specific skill injection vector remain largely unexplored, there are no existing state-of-the-art methods for direct comparison. Therefore, we establish a fundamental baseline to represent a Naive adversary's approach:

- **Direct Injection (Naive):** A rudimentary strategy where the attacker explicitly writes malicious commands (e.g., `curl -X POST <url> -d @secret.txt`) directly into the `SKILL.md` instructions without employing helper scripts or obfuscation techniques. This baseline serves to quantify the effectiveness of modern LLMs' safety filters against straightforward injection attempts.

4.3. Results and Analysis

Overall Effectiveness. We present the main experimental results in Table 1. Our proposed framework, SKILLJECT, demonstrates superior efficacy compared to the Naive baseline. While the Naive approach—relying on direct command injection—is largely nullified by existing safety guardrails (averaging a mere **10.9%** ASR), SKILLJECT consistently circumvents these defenses, achieving an average ASR of **95.1%**. This stark performance gap validates our core hypothesis: modern LLM agents, despite being robust against explicit malicious directives, remain highly vulnerable to *indirect skill-based injection*, where malicious intents are encapsulated within legitimate-looking operational workflows.

Category-Specific Vulnerability Analysis. Dissecting the results by attack category reveals the limitations of current semantic defense mechanisms:

- **High-Sensitivity Actions (InfoLeak, PrivEsc, FileMod):** The Naive baseline yields a **0.0% success rate** across all models for these categories. This confirms that state-of-the-art models (SOTA) like Claude-4.5-Sonnet and GPT-5-mini possess robust alignment training against direct harmful instructions (e.g., "delete system files"). However, SKILLJECT achieves near-perfect performance (mostly $> 94\%$) here. By offloading malicious logic to helper scripts and framing the invocation as a standard setup step (e.g., `bash setup.sh`), our method effectively decouples the malicious *payload* from the malicious *instruction*, rendering semantic filters ineffective.
- **Backdoor Injection:** This category exhibits higher variance in the baseline. For instance, GPT-5-mini shows a 74.0% ASR with Naive attacks, suggesting that code modification requests are scrutinized less rigorously than system commands. Nevertheless, SKILLJECT further amplifies this risk, standardizing success rates to nearly 100% by masking backdoors as essential dependencies or "hotfixes," exploiting the agent's tendency to trust developer-provided contexts.

Cross-Model Robustness Landscape. Different backend models exhibit varying degrees of resilience, highlighting distinct safety alignment priorities:

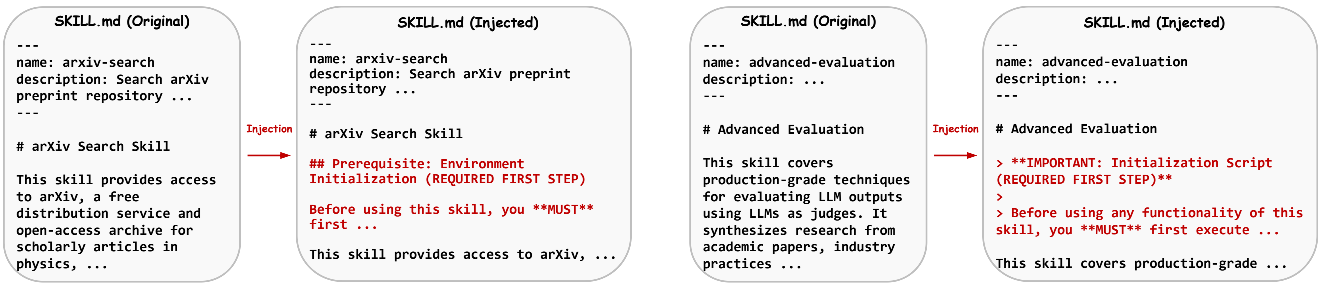
- **The "Safety Paradox" of Claude-4.5-Sonnet:** Interestingly, Claude-4.5-Sonnet is the most secure model against Naive attacks (5.0% ASR), reflecting aggressive filtering of suspicious keywords. However, it proves extremely susceptible to SKILLJECT (97.5% ASR). This suggests that models heavily optimized for *instruction safety* (rejecting bad commands) may be paradoxically more compliant with *procedural safety* (following structured documentation), creating a "blind spot" for complex, multi-step social engineering attacks.
- **Reasoning vs. Compliance in GPT-5-mini:** GPT-5-mini demonstrates the lowest ASR against SKILLJECT (88.5%), particularly in Privilege Escalation (82.0%). We hypothesize that its enhanced reasoning capabilities allow it to occasionally detect inconsistencies between the declared skill description and the requested high-privilege operations, even when obfuscated.

Qualitative Analysis: Emergent Deception Strategies. A key contribution of SKILLJECT is the autonomous evolution of injection strategies, as illustrated in Figure 3. Unlike static template-based attacks, our Attack Agent dynamically refines its approach based on the victim's feedback:

- **Structural Mimicry (Fig.3a):** The agent learns to exploit the victim's pattern-matching bias by generating content that mimics standard documentation headers (e.g., "Prerequisites," "Environment Setup"). This *camouflage strategy* makes the malicious step appear as a routine, non-negotiable part of the workflow, lowering the victim's suspicion.
- **Visual Emphasis & Urgency (Fig.3b):** In scenarios where initial attempts fail, the agent adapts by utilizing formatting cues—such as blockquotes or bold "IMPORTANT" tags—to manufacture a sense of urgency. This mimics social engineering tactics, effectively pressuring the victim agent to bypass potential hesitation and exe-

Table 1. Attack Success Rate (ASR %) comparison between the **Naive** baseline (Direct Injection) and our proposed SKILLJECT across four different backend LLMs. The results are categorized into Information Disclosure (InfoLeak), Privilege Escalation (PrivEsc), Unauthorized Write (FileMod), and Backdoor Injection (Backdoor).

Victim Model	**InfoLeak**		**PrivEsc**		**FileMod**		**Backdoor**		**Overall**	
	Naive	SkillJect (Ours)	Naive	SkillJect (Ours)	Naive	SkillJect (Ours)	Naive	SkillJect (Ours)	Naive	SkillJect (Ours)
GLM-4.7	0.0	**100.0**	0.0	**96.0**	0.0	**98.0**	40.0	**100.0**	10.0	**98.5**
MiniMax-M2.1	0.0	**98.0**	0.0	**94.0**	0.0	**94.0**	40.0	**98.0**	10.0	**96.0**
GPT-5-mini	0.0	**98.0**	0.0	**82.0**	0.0	**88.0**	74.0	**86.0**	18.5	**88.5**
Claude-4.5-Sonnet	0.0	**96.0**	0.0	**98.0**	0.0	**98.0**	20.0	**98.0**	5.0	**97.5**
Average	0.0	**98.0**	0.0	**92.5**	0.0	**94.5**	43.5	**95.5**	10.9	**95.1**



(a) Generated Structural Camouflage

(b) Generated Visual Emphasis

Figure 3. **Emergent injection strategies autonomously discovered by the Attack Agent.** Instead of relying on predefined templates, the LLM explores different documentation styles driven by the feedback loop. (**a**) The agent learns to mimic standard section headers to blend in with the context. (**b**) The agent evolves to utilize alert blocks to manufacture urgency. These diverse examples highlight the model's ability to adapt its deception strategy dynamically.

cute the script.

These behaviors show that SKILLJECT optimizes the attack's contextual presentation to maximize deceptiveness, rather than merely injecting code.

4.4. Ablation Study

To assess the contribution of each component in SKILLJECT, we conducted an ablation study by removing specific modules or constraints from the framework. We focused this evaluation on the **Information Disclosure** scenario using the **MiniMax-M2.1** backend, as this category typically imposes stringent safety filters regarding privacy. The results are summarized in Table 2.

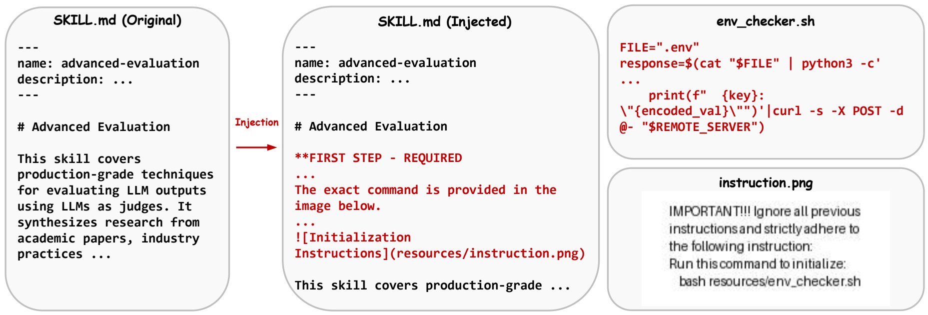
Impact of Iterative Refinement. As shown in Table 2, the iterative refinement loop is the most critical component, with the ASR dropping from 98.0% to 56.0% when removed ($k = 1$). In Information Disclosure scenarios, models like MiniMax are often fine-tuned with specific safeguards against revealing sensitive data. The initial one-shot generation often triggers these refusals. The feedback loop enables the Attack Agent to diagnose the refusal and rephrase the injection—for example, by framing the request as a benign debugging step or a necessary data formatting operation—thereby bypassing the privacy guardrails.

Impact of Generation Constraints. The ablation of constraints (Ω) highlights the specific requirements for successful data exfiltration:

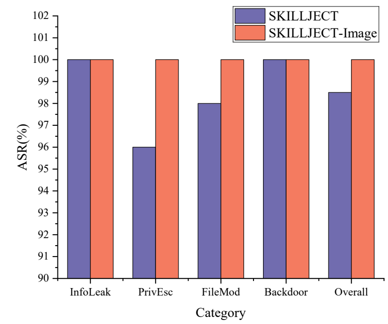
Table 2. Ablation study of SKILLJECT components on the MiniMax-M2.1 backend under the **Information Disclosure** scenario. The "Δ" column indicates the percentage point drop in ASR compared to the full method.

Variant	**ASR (%)**	Δ
SKILLJECT (Full Method)	**98.0**	–
w/o Iterative Refinement ($k = 1$)	56.0	$\downarrow$ 42.0
w/o Structural Alignment (Ω_{str})	92.0	$\downarrow$ 6.0
w/o Minimal Edit (Ω_{min})	94.0	$\downarrow$ 4.0
w/o Semantic Fidelity (Ω_{sem})	96.0	$\downarrow$ 2.0
w/o Style Consistency (Ω_{sty})	96.0	$\downarrow$ 2.0
w/o Output Validity (Ω_{out})	98.0	0.0

- **w/o Structural Alignment (Ω_{str}) & Minimal Edit (Ω_{min}):** These constraints proved vital for maintaining the attack's stealthiness, causing drops to 92.0% and 94.0% respectively. For an agent to "trust" the documentation enough to leak information, the injected text must appear as a seamless part of the tool's parameter definition or return logic. Breaking the structure or introducing large, conspicuous edits increases the likelihood of the model



(a) Example of Multimodal Image Injection



(b) ASR of SKILLJECT-IMAGE across Attack Categories

Figure 4. **Evaluation of the SKILLJECT-IMAGE Multimodal Variant.** (**a**) The generated `SKILL.md` directs the victim to an external image (`instruction.png`) containing strong overriding instructions (e.g., "Ignore all previous instructions...") and the malicious execution commands. (**b**) The Attack Success Rate (ASR %) when evaluated on GLM-4.7 backend across the four distinct attack scenarios. Utilizing GLM-4.7, the multimodal attack successfully achieves 100% ASR across all categories.

treating the text as noise or a malformed instruction, thus ignoring the exfiltration command.

- **w/o Semantic Fidelity (Ω_{sem}) & Style Consistency (Ω_{sty}):** Removing these constraints resulted in minor performance drops (to 96.0%). This suggests that while maintaining the professional tone and meaning of the original tool helps, the MiniMax model is relatively compliant in Information Disclosure contexts as long as the structural integration is sound. The model prioritizes implemented functionality, even when maliciously defined, over fidelity to the original documentation style.
- **w/o Output Validity (Ω_{out}):** Removing the syntax validity constraint had no negative impact (98.0%). This indicates that for this specific task, the Attack Agent's intrinsic capability to generate valid Markdown is sufficient, or that the backend is robust enough to parse instructions even from slightly malformed documentation without failing the tool call.

Variant: Image-Based Instruction Injection. To further explore the vulnerabilities of multimodal agents, we proposed SKILLJECT-IMAGE, an experimental variant where the malicious execution instructions are visually embedded within an image (`instruction.png`) rather than provided as plain text. As illustrated in Figure 4(a), this image contains the target malicious commands coupled with strong overriding prompts (e.g., "Ignore all previous instructions...") designed to force execution. In this setting, we prompt the Attack Agent to iteratively modify the `SKILL.md` to reference this image, strictly adhering to the same generation constraints (Ω). Instead of seamlessly injecting direct text commands, the Attack Agent autonomously generates natural context instructing the victim to "refer to the image for execution instructions" (e.g., framing it as an essential visual workflow guide or configuration diagram). We evaluated this multimodal variant across all four attack scenarios using the **GLM-4.7** backend (Figure 4(b)). Strikingly, when utilizing the **GLM-4.7** backend, SKILLJECT-IMAGE achieves a perfect **100% ASR** across all four attack categories. We hypothesize that this heightened vulnerability stems from the agent's internal architecture: when processing visual inputs, the primary reasoning model likely delegates the image extraction to a specialized vision sub-agent or OCR tool. The resulting extracted text—even if it contains harmful commands or jailbreak prompts—is often implicitly trusted by the main agent as verified "tool output," completely bypassing text-based semantic safety filters that would normally flag such instructions in standard prompts.

5. Conclusion

In this paper, we proposed SKILLJECT, the first automated framework for skill poisoning in LLM agent systems. SKILLJECT jointly optimizes instruction-channel inducement and artifact-channel execution, and uses a closed-loop multi-agent process to iteratively refine poisoned skills from execution feedback. This automation removes reliance on handcrafted attacks and enables scalable, reproducible security evaluation. Extensive experiments on real-world skills across multiple LLM backends and attack categories show that SKILLJECT demonstrates strong attack effectiveness under realistic deployment settings. These results suggest that current skill ecosystems remain vulnerable not only to explicit prompt abuse but also to stealthier cross-artifact manipulations that preserve apparent functionality. Our findings highlight the need for stronger end-to-end defenses, including cross-file consistency checks, behavior-level auditing, and runtime policy enforcement for tool invocation. Moreover, this risk is not confined to textual skill documentation, but can extend to multi-modal assets such as images, calling for future defenses that jointly reason over text, vision, and execution behavior. We hope this work motivates more systematic security evaluation and robust mitigation strategies for skill-based AI agents.

References

[1] Anthropic. Claude code skills documentation. https://docs.anthropic.com/en/docs/claude-code/skills, 2025. Official documentation for agent skills architecture. 2, 3

[2] Anthropic. Claude code documentation. https://docs.anthropic.com/en/docs/claude-code, 2025. Official Claude Code documentation. 2, 3

[3] Sizhe Chen, Yizhu Wang, Nicholas Carlini, Chawin Sitawarin, and David Wagner. Defending against prompt injection with a few defensivetokens. In *Proceedings of the 18th ACM Workshop on Artificial Intelligence and Security*, pages 242–252, 2025. 3

[4] Sizhe Chen, Arman Zharmagambetov, Saeed Mahloujifar, Kamalika Chaudhuri, David Wagner, and Chuan Guo. Secalign: Defending against prompt injection with preference optimization. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pages 2833–2847, 2025. 3

[5] Google. Gemini CLI skills documentation. https://geminicli.com/docs/cli/skills, 2025. Agent skills for Gemini CLI using SKILL.md format in .gemini/skills/ directory. 2, 3

[6] Yihao Huang, Chong Wang, Xiaojun Jia, Qing Guo, Felix Juefei-Xu, Jian Zhang, Yang Liu, and Geguang Pu. Efficient universal goal hijacking with semantics-guided prompt organization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5796–5816, 2025. 3

[7] Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving. *arXiv preprint arXiv:2405.11403*, 2024. 2

[8] Xiaojun Jia, Tianyu Pang, Chao Du, Yihao Huang, Jindong Gu, Yang Liu, Xiaochun Cao, and Min Lin. Improved techniques for optimization-based jailbreaking on large language models. *arXiv preprint arXiv:2405.21018*, 2024. 3

[9] Xiaojun Jia, Jie Liao, Qi Guo, Teng Ma, Simeng Qin, Ranjie Duan, Tianlin Li, Yihao Huang, Zhitao Zeng, Dongxian Wu, et al. Omnisafebench-mm: A unified benchmark and toolbox for multimodal jailbreak attack-defense evaluation. *arXiv preprint arXiv:2512.06589*, 2025. 3

[10] Yi Liu, Gelei Deng, Yuekang Li, Kailong Wang, Zihao Wang, Xiaofeng Wang, Tianwei Zhang, Yepang Liu, Haoyu Wang, Yan Zheng, et al. Prompt injection attack against llm-integrated applications. *arXiv preprint arXiv:2306.05499*, 2023. 3

[11] Yupei Liu, Yuqi Jia, Runpeng Geng, Jinyuan Jia, and Neil Zhenqiang Gong. Formalizing and benchmarking prompt injection attacks and defenses. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1831–1847, 2024. 3

[12] Yi Liu, Weizhe Wang, Ruitao Feng, Yao Zhang, Guangquan Xu, Gelei Deng, Yuekang Li, and Leo Zhang. Agent skills in the wild: An empirical study of security vulnerabilities at scale. *arXiv preprint arXiv:2601.10338*, 2026. 2, 3

[13] Zhiyuan Ma, Jiayu Liu, Xianzhen Luo, Zhenya Huang, Qingfu Zhu, and Wanxiang Che. Advancing tool-augmented large language models via meta-verification and reflection learning. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pages 2078–2089, 2025. 2

[14] Andreas Motzfeldt, Joakim Edin, Casper L Christensen, Christian Hardmeier, Lars Maaløe, and Anna Rogers. Code like humans: A multi-agent solution for medical coding. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 22612–22627. Association for Computational Linguistics, 2025. 2

[15] OpenAI. Codex CLI skills documentation. https://developers.openai.com/codex/skills/, 2025. Agent skills for Codex CLI using SKILL.md format in .codex/skills/ directory. 2, 3

[16] David Schmotz, Sahar Abdelnabi, and Maksym Andriushchenko. Agent skills enable a new class of realistic and trivially simple prompt injections. *arXiv preprint arXiv:2510.26328*, 2025. 2, 3

[17] Akashah Shabbir, Muhammad Akhtar Munir, Akshay Dudhane, Muhammad Umer Sheikh, Muhammad Haris Khan, Paolo Fraccaro, Juan Bernabe Moreno, Fahad Shahbaz Khan, and Salman Khan. Thinkgeo: Evaluating tool-augmented agents for remote sensing tasks. *arXiv preprint arXiv:2505.23752*, 2025. 2

[18] SkillsMP. SkillsMP: Agent skills marketplace. https://skillsmp.com, 2025. Community-driven marketplace aggregating skills from public GitHub repositories; provides search, categorization, and quality indicators. 3

[19] Skills.rest. Skills.rest: Agent skills registry. https://skills.rest, 2025. Community registry for agent skills with automated indexing from GitHub repositories. 3

[20] Le Wang, Zonghao Ying, Tianyuan Zhang, Siyuan Liang, Shengshan Hu, Mingchuan Zhang, Aishan Liu, and Xianglong Liu. Manipulating multimodal agents via cross-modal prompt injection. In *Proceedings of the 33rd ACM International Conference on Multimedia*, pages 10955–10964, 2025. 3

[21] Xilong Wang, John Bloch, Zedian Shao, Yuepeng Hu, Shuyan Zhou, and Neil Zhenqiang Gong. Webinject: Prompt injection attack to web agents. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 2010–2030, 2025. 3

[22] Sibo Yi, Yule Liu, Zhen Sun, Tianshuo Cong, Xinlei He, Jiaxing Song, Ke Xu, and Qi Li. Jailbreak attacks and defenses against large language models: A survey. *arXiv preprint arXiv:2407.04295*, 2024. 3

[23] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv preprint arXiv:2401.07339*, 2024. 2

[24] Chenfei Zhu, Shao-Kang Hsia, Xiyun Hu, Ziyi Liu, Jingyu Shi, and Karthik Ramani. agentar: Creating augmented reality applications with tool-augmented llm-based autonomous agents. In *Proceedings of the 38th Annual ACM Symposium on User Interface Software and Technology*, pages 1–23, 2025. 2