

SkCC: Portable and Secure Skill Compilation for Cross-Framework LLM Agents

Yipeng Ouyang
Sun Yat-sen University
ouyyp5@mail2.sysu.edu.cn

Yuhao Gu
Sun Yat-sen University
guyh9@mail2.sysu.edu.cn

Yi Xiao
Sun Yat-sen University
xiaoy398@mail2.sysu.edu.cn

Xianwei Zhang
Sun Yat-sen University
zhangxw79@mail.sysu.edu.cn

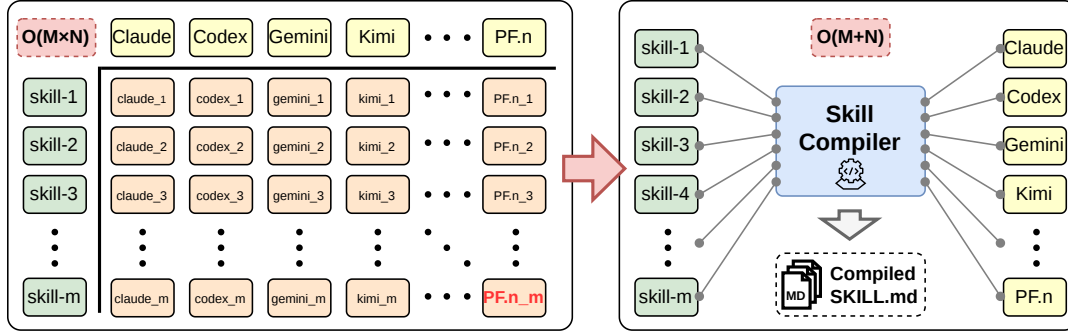


Figure 1. Complexity reduction from $O(m \times n)$ to $O(m + n)$. Traditional per-platform rewriting (left) requires $m \times n$ manual adaptations. SkCC (right) decouples skills and platforms through a shared IR, requiring only m skill sources and n Emitter implementations.

Abstract

LLM-Agents have evolved into autonomous systems for complex task execution, with the SKILL.md specification emerging as a *de facto* standard for encapsulating agent capabilities. However, a critical bottleneck remains: different agent frameworks exhibit starkly different sensitivities to prompt formatting, causing up to 40% performance variation, yet nearly all skills exist as a single, format-agnostic Markdown version. Manual per-platform rewriting creates an unsustainable maintenance burden, while prior audits have found that over one third of community skills contain security vulnerabilities [9]. To address this, we present SkCC, a compilation framework that introduces classical compiler design into agent skill development. At its core, SkIR—a strongly-typed intermediate representation—decouples skill semantics from platform-specific formatting, enabling portable deployment across heterogeneous agent frameworks. Around this IR, a compile-time Analyzer enforces security constraints via Anti-Skill Injection before deployment. Through a four-phase pipeline, SkCC reduces adaptation complexity from $O(m \times n)$ to $O(m + n)$. Experiments on SkillsBench demonstrate that compiled skills consistently outperform their original counterparts, improving pass rates from 21.1% to 33.3% on Claude Code and from 35.1% to 48.7% on Kimi CLI, while achieving sub-10ms compilation latency, a 94.8% proactive security

trigger rate, and 10–46% runtime token savings across platforms.

Keywords: LLM agents, skill compilation, prompt engineering, format adaptation, security hardening, intermediate representation

1 Introduction

The rapid advancement of large language models (LLMs) has catalyzed a new generation of autonomous agent systems [40, 43, 45]. For now, frameworks such as Anthropic Claude Code [8], OpenAI Codex [31], Google Gemini CLI [13], and Kimi CLI [18] provide terminal-based agent environments where LLMs interact with tools, file systems, and external services. Skills, structured prompt artifacts following the SKILL.md specification [3], have emerged as the *de facto* standard for encoding domain-specific knowledge, employing progressive disclosure [44] that loads lightweight metadata at initialization and retrieves full content on demand. EvoSkill [5] further advances the modular skill paradigm by automatically discovering and refining skills through iterative failure analysis.

A growing body of evidence reveals that LLM performance is highly sensitive to the structural format in which skills are presented [15]. Claude performs substantially better when skills use XML semantic layering [7], GPT-series models

benefit from XML-tagged Markdown that avoids the “format tax” of JSON [30], and deeply nested data is parsed most accurately in YAML [16]. Yet the current ecosystem assumes format-agnostic delivery: the same SKILL.md is deployed identically across all platforms. Beyond format compatibility, the skill ecosystem faces an equally pressing security challenge. Snyk’s audit of 3,984 community skills from ClawHub [9] found that 37% contain security vulnerabilities, including 76 confirmed malicious payloads, while nearly one-third degrade agent performance due to formatting errors or missing guardrails. The SKILL.md specification acknowledges the need for negative boundaries [2], yet most existing skills lack such constraints.

We present SkCC, a systematic skill compilation framework that addresses both the portability and security challenges of cross-framework skill deployment. The central insight is that a shared intermediate representation—SkIR—can decouple skill authoring from platform-specific formatting, enabling each skill to be written once and compiled to multiple targets. This architectural decision naturally supports two complementary capabilities: platform-specific emission that aligns skill formatting with each model’s training distribution, improving task completion, and a compile-time Analyzer that enforces security constraints via Anti-Skill Injection before deployment.

Our key contributions are as follows:

- **We identify a structural gap in the agent skill ecosystem:** format sensitivity is a first-class concern in skill deployment, and the growing diversity of agent frameworks makes manual per-platform adaptation infeasible—motivating a compiler-based solution with a shared intermediate representation.
- **We propose SkCC,** a four-phase skill compilation framework that achieves portable deployment through SkIR and secure execution through compile-time Anti-Skill Injection and semantic validation.
- **We implement and evaluate SkCC across four mainstream agent platforms,** demonstrating consistent pass rate improvements (up to +13.5pp, $p < 0.01$), sub-10ms compilation latency, 94.8% Anti-Skill Injection coverage, and 10–46% runtime token savings, with ablation experiments confirming that compilation gains are strictly model-dependent.

2 Background and Related Work

Agent Skills Structure and Retrieval. The concept of agent skills has evolved alongside LLM-based agentic systems, transitioning from monolithic system prompts to modular, retrievable skill artifacts. The Agent Skills open standard [3] introduced SKILL.md as a portable specification with YAML frontmatter and a Markdown body, enabling progressive disclosure [44] where lightweight metadata is loaded at initialization and full content retrieved on demand.

EvoSkill [5] advanced this paradigm through automatic skill discovery and refinement via iterative failure analysis. Recent work has expanded skill retrieval through generation-based [41], augmentation-based [36], graph-based [11], and embedding-based [34] approaches. A recent comprehensive study [25] systematically evaluated how agentic skills perform under realistic retrieval conditions, revealing that skill quality and selection strategy critically impact task success. However, these works all assume format-agnostic delivery, an assumption that breaks down when different underlying LLMs exhibit strong format preferences.

Structured Prompting and Format Sensitivity. The format sensitivity of LLMs provides the empirical foundation for SkCC’s platform-specific emission strategies. He et al. [15] demonstrated up to 40% performance variation from format changes alone, while Liu et al. [26] introduced Content-Format Integrated Prompt Optimization (CFPO) for joint content-format refinement. Platform-specific preferences are well-documented: Anthropic establishes XML tagging as a first-class best practice for Claude [7], reporting up to 23% accuracy improvement [32, 33], with practitioners developing systematic XML-based prompt engineering methodologies [38]; OpenAI’s GPT-series models suffer from a “format tax” when parsing JSON-formatted inputs [19, 30]. For nested data, YAML achieves the highest parsing accuracy (51.9%) compared to JSON (43.1%) and XML (33.8%) [16]. On the security dimension, Snyk’s ToxicSkills study [9] found 37% of 3,984 community skills contain security vulnerabilities, while the SKILL.md specification [2] recommends negative boundaries rarely followed in practice [20]; recent work on secure code generation via reasoning internalization [39] further underscores the urgency of compile-time safety enforcement for agent skills.

Compilation for Agents and Skills. The idea of applying compilation techniques to LLM-based systems has gained significant traction. Mikek et al. [27] demonstrated that compiler-LLM cooperation can effectively optimize agentic code across multiple abstraction levels, while Kim et al. [17] showed how classical compiler orchestration enables efficient parallel function calling. These works illustrate the broader potential of compiler-inspired architectures for agent systems. Most relevant to our work, SkVM [10] pioneered the application of compilation concepts to agent skills, proposing a JVM-like architecture with capability profiling and AOT/JIT optimization. SkVM’s recognition that skills benefit from compilation-like processing is an important contribution, and its multi-platform support through a VM model demonstrates the value of platform abstraction. However, SkVM focuses on semantic capability degradation rather than format-syntax adaptation, and does not incorporate security hardening into its pipeline. Our work builds on these foundations by introducing a classical multi-phase compiler architecture—drawing on principles established by

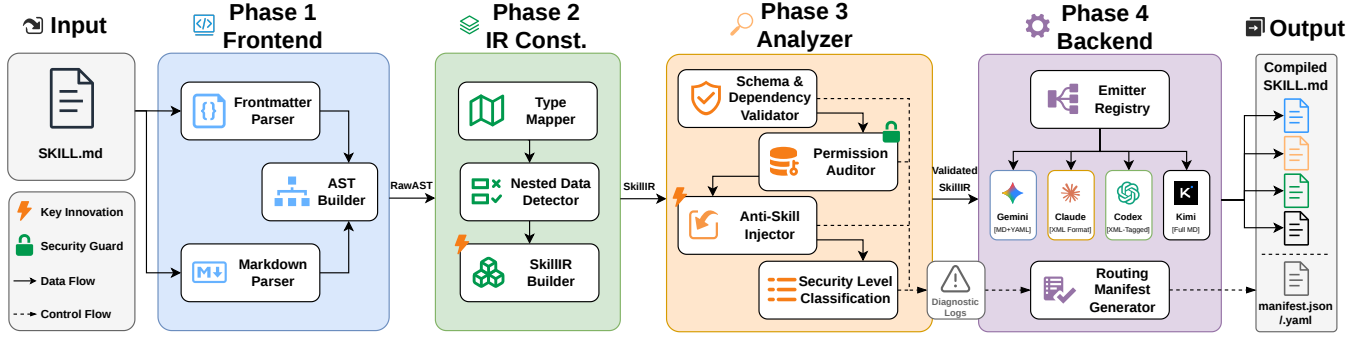


Figure 2. SkCC’s four-phase compilation pipeline. A unified SKILL.md source is parsed into a raw AST, transformed into a strongly-typed SkIR, validated and hardened by the Analyzer, and emitted into platform-native formats through polymorphic Emitters.

Aho, Sethi, and Ullman [4] and advanced by Muchnick [28]—that simultaneously addresses format adaptation through a shared IR and security enforcement through compile-time analysis. The $O(m \times n)$ to $O(m + n)$ reduction principle, first articulated by Strong et al. [35] and exemplified by systems like LLVM [21], provides the architectural motivation: a universal intermediate layer decouples source languages from target machines, and we argue the same principle applies to decoupling skill authoring from platform-specific formatting [22, 23]. SkCC’s Anti-Skill Injection mechanism further parallels compiler-level security hardening such as stack canary insertion and bounds checking [37].

Motivation. The preceding analysis reveals a structural gap in the agent skill ecosystem. Extensive evidence demonstrates that LLM performance is highly sensitive to prompt formatting [15, 26], with specific platforms exhibiting strong, documented preferences [7, 16, 30]. Yet the SKILL.md standard [3] assumes format-agnostic delivery: the same Markdown file is expected to function identically across Claude, GPT, Gemini, and Kimi. Simultaneously, security audits reveal that 37% of community skills contain vulnerabilities [9], with no systematic mechanism for compile-time safety enforcement [20, 39]. These challenges are not merely additive; they share a common architectural root. Supporting m skills across n platforms currently requires $m \times n$ manual adaptations—a pattern that precisely mirrors the classical compiler’s target-platform diversity problem. Just as a shared IR reduces $m \times n$ source-to-target combinations to $m + n$, we argue that agent skills need an analogous compilation layer: a unified intermediate representation that decouples skill authoring from platform-specific formatting, and a compile-time analysis phase that enforces security constraints before deployment. This insight motivates SkCC: a four-phase compilation framework that treats skills as compilable artifacts rather than static text files.

3 SkCC Design

3.1 Architecture Overview

SkCC addresses two interconnected challenges identified in Section 2: (C1) format-agnostic delivery underperforms on format-sensitive platforms, creating an unsustainable $O(m \times n)$ maintenance burden as the number of skills and platforms grows; and (C2) existing skills lack systematic security hardening before deployment. Our solution centers on SkIR, a shared intermediate representation that decouples skill semantics from platform syntax, enabling portable deployment across heterogeneous frameworks (addressing C1). Around this IR, we build a compile-time Analyzer with Anti-Skill Injection that enforces security constraints before skills reach the agent’s context window (addressing C2). The compilation pipeline (Figure 2) implements this architecture in four phases.

3.2 Frontend and IR Construction

The Frontend phase functions as the lexical analyzer and parser for the SKILL.md format. It aggressively decouples metadata from execution logic: the YAML frontmatter is deserialized into a static routing table, while the Markdown body undergoes abstract syntax tree (AST) lowering. Procedure steps, code blocks, and examples are classified into deterministic memory structures, eliminating the ambiguity inherent in raw Markdown text. A SHA-256 content hash is computed to guarantee compilation reproducibility. These parsed components are assembled into a raw AST that serves as the input to IR construction.

The IR Construction phase then transforms the raw AST into SkIR—a strongly-typed, platform-independent representation that serves as the central data structure for all subsequent compilation stages. SkIR organizes skill information into six categories: (1) **Metadata & Routing** (name, version, description for semantic matching), (2) **Interfaces & MCP** (MCP—Model Context Protocol—server dependencies, input/output schemas), (3) **Security & Control** (HITL

Table 1. Agent Frameworks, Models, and Emission Strategies

Framework	Ex. Model	Emitter Format	Key Strategy
Claude Code	claude-opus-4-6	XML Semantic Layering	Tag-wrapped structure, up to 23% gain [7]
Codex CLI	gpt-5.3-codex	XML-Tagged Markdown	Structural markers, avoids format tax [30]
Gemini CLI	gemini-2.5-pro	Markdown + Conditional YAML	YAML at depth ≥ 3 (51.9% vs 43.1%) [16]
Kimi CLI	kimi-k2.5	Full Markdown Preservation	No truncation, ultra-long context

flags, pre/post-conditions, fallbacks, permissions, security level), (4) **Execution Logic** (context gathering steps, procedures, few-shot examples, alternative approaches, execution mode), (5) **Compiler-Injected Constraints** (anti-skill constraints populated during analysis), and (6) **AST Optimization Flags** (YAML optimization flag and nested data depth). SkIR supports four execution modes: Sequential (ordered workflow), Alternative (mode-selector with multiple approaches), Toolkit (reference operations), and Guideline (unstructured recommendations). A concrete SkIR instance is provided in Appendix C.1.

A key optimization performed during IR construction is **nested data detection**. When a skill declares input or output schemas with JSON Schema nesting depth ≥ 3 , the system sets a YAML optimization flag, which downstream Emitters use to decide whether to render nested data in YAML format (which achieves 51.9% parsing accuracy versus JSON’s 43.1% for deeply nested structures [16]).

3.3 Compile-time Semantic and Security Analysis

The Analyzer phase performs semantic validation and security enhancement on the SkIR, producing a validated IR with non-blocking diagnostic warnings. This phase executes a chain of five analyzers.

Structural and Dependency Validation. Schema validation verifies name format (kebab-case, 1–64 characters), description constraints (1–1024 characters, no XML tags), version format (semantic versioning), and consistency between declared schemas and few-shot examples. MCP dependency checking verifies that all declared MCP server dependencies exist in a curated allowlist; unknown servers generate error-level diagnostics that block compilation. Permission auditing validates permission declarations against a security baseline, checking scope formats and flagging dangerous operations. These three checks together ensure structural integrity and dependency safety before the skill proceeds further.

Anti-Skill Injection. This is the core security innovation of SkCC. Rather than relying on skill authors to manually include defensive constraints, the system automatically scans procedure text for dangerous patterns and injects corresponding safety constraints into the SkIR. The injector maintains four anti-pattern rules covering HTTP safety (timeout enforcement, retry limits), HTML parsing safety (fallback to

regex for script tags), destructive database operations (user confirmation gating), and infinite loop prevention (iteration caps). The complete rule table is provided in Appendix C.2. The injection process operates entirely at compile time via AST traversal: when a procedure step matches a trigger pattern, the corresponding constraint is appended to the IR’s constraint array, ensuring the safety instruction is rigidly embedded across all target formats. Across our evaluation corpus of 233 community skills, Anti-Skill Injection triggered in 94.8% of skills (Section 4.4.2).

Security Classification. The final analyzer assigns each skill one of four security levels—Low (basic format validation), Medium (permission declaration check, default), High (mandatory HITL and dangerous keyword scan), or Critical (no auto-execution, requires human approval)—based on declared permissions and HITL requirements. Skills at High or Critical levels automatically enforce human-in-the-loop confirmation, while Critical-level skills block automatic execution entirely.

The compile-time nature of this analysis is a deliberate design choice: by intercepting dangerous patterns before skill deployment, SkCC prevents unsafe skills from ever reaching the agent’s context window. This contrasts with runtime safety mechanisms that rely on the agent’s own judgment, an approach that is inherently unreliable given the well-documented tendency of LLMs to follow instructions literally, even when those instructions are malicious [9].

3.4 Target Emission and Format Hardening

The Backend phase emits platform-native skill artifacts through a polymorphic architecture. For multi-target compilation, Phases 1–3 execute once to produce a single validated SkIR, which is then shared across all emission targets—this is the architectural mechanism that achieves the $O(m + n)$ complexity reduction.

We design four platform-specific emission strategies, each informed by empirical findings about the target model’s format sensitivity (Section 2). Table 1 summarizes the key strategies; detailed output examples for each target are provided in Appendix C.3.

Routing Manifest Generation. In addition to platform-specific skill artifacts, SkCC generates a progressive routing manifest containing only the name, description, security level, and HITL flag for each skill (~50 tokens per skill).

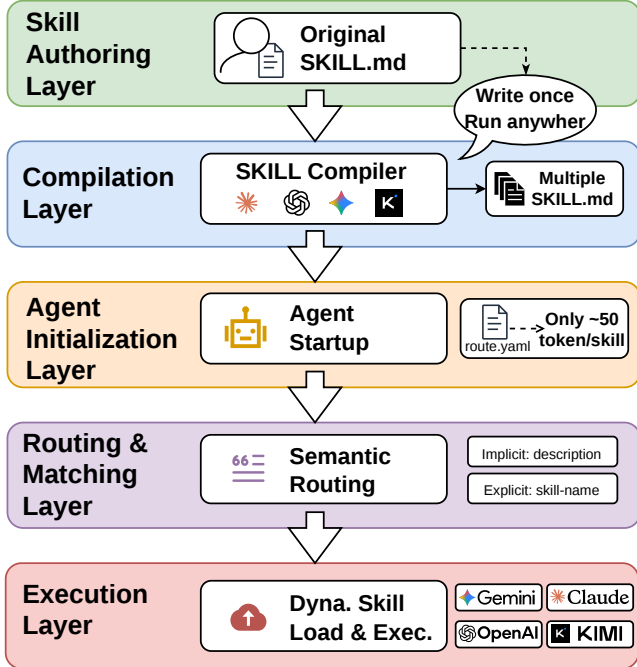


Figure 3. Agent workflow with SkCC integration. Skills are authored once as SKILL.md, compiled to platform-native formats, and loaded via progressive routing manifests at agent initialization.

This manifest enables efficient semantic routing at agent initialization without loading full skill content, implementing the progressive disclosure mechanism defined by the Agent Skills standard [3, 44].

For implementation details including library dependencies, crate structure, and CLI usage, see Appendix A.

4 Evaluation

We evaluate SkCC on three dimensions: (1) whether compiled skills improve over original skills in terms of pass rate, token efficiency, and execution time, and how these gains compare to state-of-the-art alternatives; (2) whether compilation gains are inherently tied to specific model-format pairings—i.e., a format that benefits one model may not benefit another, validating the need for platform-specific emission; and (3) whether the compiler’s engineering properties—compilation latency, Anti-Skill Injection coverage, and compile-time safety interception—meet practical deployment requirements. All experiments use SkillsBench [24] as the benchmark, with 89 real-world programming and data analysis tasks.

4.1 Experiment Setup

Benchmark and Datasets. SkillsBench [24] provides 89 real-world tasks with Docker-based execution and automated pytest verification, classified by difficulty and category.

We use Pass@1 (reward ≥ 0.5) as our primary metric. For compilation performance and token efficiency experiments, we collected 225 skills from four community repositories: Anthropic-skills [6], ecc-skills (everything-claude-code) [1], sentry-skills (Sentry team) [12], and ui-skill [29].

LLM Models and Agent Frameworks. Table 1 in Section 3 summarizes the four mainstream agent frameworks, their corresponding models, and the emission strategies employed by SkCC. All experiments use the Harbor framework [14] for Docker-based task execution, with each agent framework running within Harbor-managed containers. Ablation experiments additionally test glm-5.1 and deepseek-v4-flash via the OpenHands SDK [42] integrated within Harbor.

Baselines. We compare two conditions across all experiments: **Original (O)**—the current de facto standard: a single, format-agnostic SKILL.md deployed identically across all platforms; and **Compiled (C)**—the SkCC-compiled SKILL.md with platform-specific formatting and Anti-Skill constraints. The Original condition represents the state of practice that virtually all community skills follow today.

Metrics. We evaluate across three categories of metrics. For compilation gains (Section 4.2), we measure Pass@1 (task pass rate), Mean Reward, total token consumption, and execution time, comparing Original vs. Compiled conditions on each platform. For ablation experiments (Section 4.3), we use Pass@1 and paired statistical tests (paired t-test, Cohen’s d) to quantify the effect of a fixed compiled format across different models. For engineering properties (Section 4.4), we measure per-skill compilation latency (ms), Anti-Skill Injection trigger rate (%), and compilation interception counts by category.

Data Validity. All experiments ran the full SkillsBench benchmark (89 tasks). Due to regional limitations of Anthropic/OpenAI/Google, network instability in Docker containers, and agent framework crashes, not all trials produced valid results. We present all **valid data** (both conditions successfully executed with valid reward data), guaranteeing up to 74 paired tasks per experiment. Abnormal data (e.g., execution engine crashes, API rate limits) has been excluded from statistical analysis. Importantly, exclusion was strictly based on execution viability, completely blind to the reward outcomes, ensuring a fair and unbiased comparison across all conditions.

4.2 Evaluating Compilation Gains

4.2.1 Four-Model Comparison. Table 2 reports the pass rate and mean reward for Original and Compiled conditions across all four platforms.

Claude Code (claude-opus-4-6). Compiled significantly outperforms Original ($p = 0.0103$, $d = 0.60$) with medium-to-large effect sizes. In 22 paired tasks, Compiled never loses

Table 2. Four-Model Condition Comparison (Original vs. Compiled)

Condition	Tasks	Pass	Pass%	Mean Rwd.
Claude-O	38	8	21.1%	0.245
Claude-C	27	9	33.3%	0.378
Kimi-O	75	26	35.1%	0.341
Kimi-C	76	36	48.7%	0.483
Codex-O	26	10	38.5%	0.433
Codex-C	26	11	42.3%	0.499
Gemini-O	18	4	22.2%	0.250
Gemini-C	18	4	22.2%	0.269

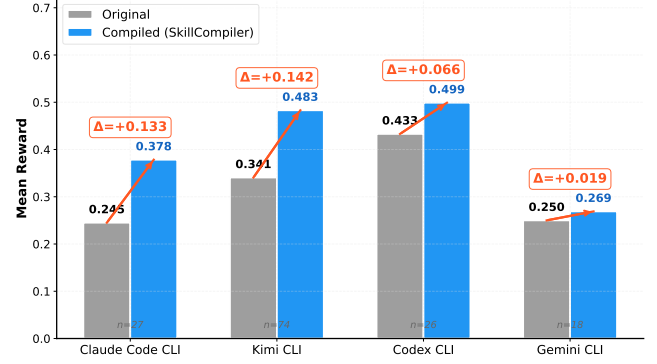
to Original (7 wins, 15 ties, 0 losses). Notably, 6 of the 7 tasks where Compiled outperforms Original flipped from reward=0 to reward=1, demonstrating that the XML Semantic Layering format enables Claude to correctly follow instructions it fails to interpret in plain Markdown. The complete paired statistical test results are provided in Appendix D.2.

Kimi CLI (kimi-k2.5). This experiment achieves $p < 0.01$ (paired t-test, $t = 2.815$, $p = 0.0063$, Cohen’s $d = 0.327$), representing the strongest statistical result in our evaluation. The pass rate improves by 13.5 percentage points (35.1% to 48.7%). Among 16 discriminative tasks, Compiled wins 13 (81.25%) while Original wins 3 (18.75%). Thirteen tasks flipped from reward=0 under Original to reward=1 under Compiled. The complete statistical test results are provided in Appendix D.3.

Codex CLI (gpt-5.3-codex). Among the 26 comparable tasks, Compiled performs better on 3 tasks (11.5%), all of which flipped from complete failure to complete success ($\Delta = +1.0$), while performing worse on 3 tasks (11.5%) with 1 complete flip failure and 2 minor regressions. The remaining 20 tasks (77.0%) show no meaningful difference. Compiled shows a positive reward gain of +0.067.

Gemini CLI (gemini-2.5-pro). Uses **Best Trial strategy** (Best-of-3 Oracle Selection) due to high inter-trial variance (18 comparable paired tasks). Compiled performs better on 3 tasks (15%), worse on 2 tasks (10%), shows no meaningful difference on 13 tasks (65%), and 2 tasks (10%) are not comparable. The modest positive reward gain (+0.019) is expected: Gemini 2.5 Pro is relatively format-tolerant, and YAML optimization only activates when nesting depth ≥ 3 .

Cross-Model Summary. SkCC compilation shows positive gains on all four mainstream agent frameworks, with effect sizes ranging from medium-to-large ($d = 0.60$ on Claude) to small ($d = 0.33$ on Kimi). The core value of compilation lies in flipping originally failed tasks to successful ones, rather than improving already-successful tasks. The complete four-model summary table is provided in Appendix D.1.

**Figure 4.** Cross-model comparison of Original vs. Compiled conditions across four agent frameworks. Compiled consistently outperforms Original, with the largest gains on format-sensitive models (Claude, Kimi).**Table 3.** SOTA Comparison: Compilation vs. Retrieval-Based Refinement

Method	Model	Baseline → Optimized	Δ
SkCC (Ours)	Claude	21.1% → 33.3%	+12.2pp
Liu et al. [25]	Claude	40.1% → 48.2%	+8.1pp
SkCC (Ours)	Kimi	35.1% → 48.7%	+13.5pp
Liu et al. [25]	Kimi	19.8% → 23.1%	+3.3pp

4.2.2 Comparison with State-of-the-Art. We contextualize SkCC’s compilation gains against two recent systems: SkVM [10], a JVM-like skill compilation architecture, and the query-specific skill refinement approach of Liu et al. [25]. For fair comparison, we compare each system’s improvement over its own baseline: our Original → Compiled versus their pre-refinement → post-refinement.

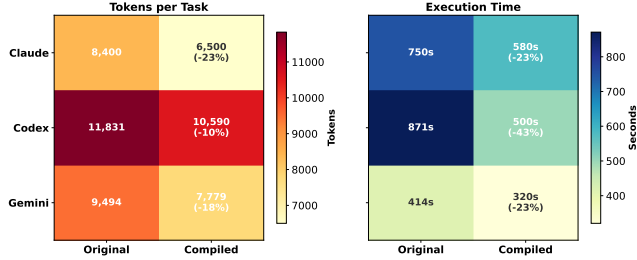
On Claude Code, SkCC achieves a +12.2pp absolute improvement, compared to +8.1pp for retrieval-based refinement. On Kimi CLI, SkCC delivers +13.5pp versus +3.3pp—a 4.1× larger gain. These results highlight a fundamental difference: retrieval-based refinement operates on already-retrieved skills and yields modest improvements, while SkCC’s compilation transforms the format-agnostic original through structural alignment with model-specific training distributions, producing substantially larger gains. SkVM [10] reports a 15.3% average improvement across multiple benchmarks, but focuses on semantic capability degradation rather than format-syntax adaptation and does not include security hardening.

4.2.3 Token and Time Efficiency.

Compile-Time Structural Expansion Overhead. Compilation introduces static structural overhead from XML tags, Anti-Skill constraints, and format hardening. Across 225 skills averaged over four platforms, the expansion overhead

Table 4. Claude Code — Token Consumption Comparison

Cond.	Tasks	Total T.	Task Avg.
Original	40	~33.4M	~0.84M
Compiled	29	~18.7M	~0.65M

**Figure 5.** Cross-platform token and time efficiency heatmap. Compiled skills show consistent reductions in total tokens and execution time across all platforms. Claude token counts are reported in hundreds due to API measurement differences.

is: Claude (XML) +24.8%, Codex (XML-Tagged Markdown) +21.9%, Gemini (MD+YAML) +18.6%, and Kimi (Full MD) +4.2%. For complex skills (>1500t), the Kimi target achieves near-zero overhead (-3.1%) or even reduction for large skills (>5000t: -6.7%). The complete expansion overhead table by complexity is provided in Appendix D.5. While compilation introduces static structural overhead, this overhead translates to dynamic efficiency gains during execution: the clearer structure reduces model trial-and-error and redundant output, resulting in net token savings of 10–46% across platforms.

Real Token and Time Consumption. On Claude Code, the compiled condition achieves lower per-task token consumption (0.65M vs. Original 0.84M) while obtaining higher reward (0.378 vs. 0.245), demonstrating that SkCC compilation improves both task performance and token efficiency simultaneously. The complete token consumption table is provided in Appendix D.6.

Similar efficiency gains are observed on Codex CLI and Gemini CLI. On Codex CLI (26 comparable tasks), compilation reduces total tokens by 10% (11,831 to 10,590) with a median reduction of 21% (7,426 to 5,894), while execution time decreases by 43% (871s to 500s). On Gemini CLI (18 comparable tasks), total tokens decrease by 18% (9,494 to 7,779), input tokens decrease by 21% (530,586 to 418,762), and execution time decreases by 23% (413.9s to 320.4s).

4.3 Ablation Study — Format Specificity

To validate that SkCC’s compiled output format is model-specific rather than universally beneficial, we conduct ablation experiments using the same Kimi-compiled output (Full

Table 5. Ablation Study — Cross-Model Format Specificity

Model	Pass% (O → C)	<i>p</i> -value
kimi-k2.5	35.1% → 48.7%	0.0063
glm-5.1	48.9% → 50.0%	0.857
deepseek-v4-flash	72.7% → 73.9%	0.2561

Table 6. Compilation Latency by Complexity

Complexity	<i>n</i>	Avg (ms)	Min (ms)	Max (ms)
Simple	8	8.54	6.90	11.73
Medium	74	8.58	6.28	17.70
Complex	143	9.13	5.85	22.89
Overall	225	8.93	5.85	22.89

Markdown) on three different models: kimi-k2.5, glm-5.1, and deepseek-v4-flash. All three experiments use the OpenHands SDK as the agent framework, with the Kimi backend format held constant.

The same compiled output produces dramatically different results across models. On Kimi, the Kimi-compiled format yields a significant positive effect ($d = +0.33$, $p = 0.0063$). On GLM-5, the effect is essentially neutral ($d = -0.03$, $p = 0.857$). On DeepSeek-v4-flash, the effect is slightly negative ($d = -0.14$, $p = 0.2561$), though not statistically significant. These results demonstrate that compilation gains are model-dependent with no one-size-fits-all optimal format, providing empirical justification for SkCC’s multi-backend architecture. The complete ablation table with all metrics is provided in Appendix D.4.

SkVM [10] represents the closest prior work in applying compilation concepts to agent skills. While SkVM demonstrates the value of platform abstraction through a VM model, our ablation results reveal a dimension that SkVM does not address: format-syntax adaptation. SkVM focuses on semantic capability profiling and degradation, but as our cross-model results show, even the same semantic content produces divergent outcomes depending on format alignment with model-specific training distributions. This finding underscores why a classical compiler architecture with platform-specific emission—rather than a VM abstraction—is necessary for format-sensitive skill deployment.

4.4 Compiler Engineering Properties

4.4.1 Compilation Performance. We measure compilation latency across 225 skills of varying complexity, compiled to all four target platforms.

All skills compile in under 10ms on average, including the most complex skills. Complexity has minimal impact on compilation time (simple 8.54ms to complex 9.13ms, only +0.59ms), and the maximum compilation time is 22.89ms, well below user perception thresholds.

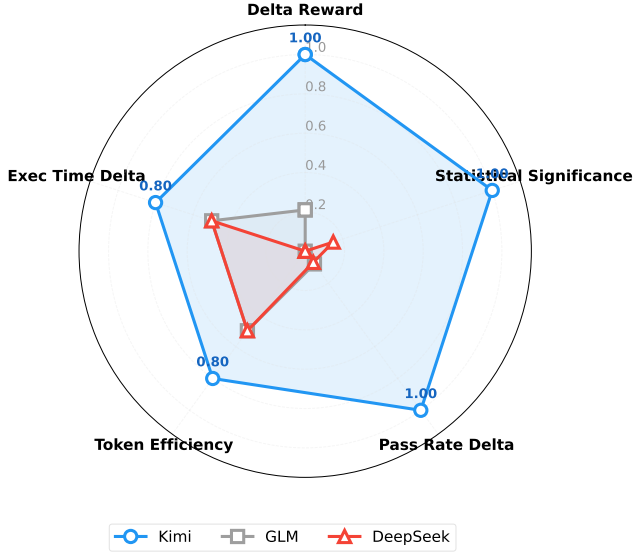


Figure 6. Ablation study radar chart: the same Kimi-compiled format produces divergent effects across three models, confirming model-specificity of compilation gains.

Table 7. Rule Trigger Distribution

Anti-Skill Rule	Triggered Skills
HTTP safety	212 (91.4%)
Loop safety	104 (44.6%)
DB safety	78 (33.5%)
Parse safety	2 (0.9%)

4.4.2 Anti-Skill Injection and Compilation Interception. SkCC’s compile-time safety checking automatically detects dangerous patterns in skill content and injects protective constraints. Across 233 evaluated skills, Anti-Skill Injection triggered in **221 (94.8%)** skills, with only 12 (5.2%) skills not triggering any rule. The complete trigger statistics table is provided in Appendix D.7.

Rule overlap is common: many skills trigger multiple rules simultaneously (HTTP + Loop + DB = most common combination). The complete rule distribution table is provided in Appendix D.8.

We also compiled all 231 SkillsBench skills targeting the Gemini platform. 221 of 231 skills (95.7%) compiled successfully, while 10 skills were intercepted by the compiler’s safety checks across three categories: YAML format violations (5 cases), security check interceptions (4 cases), and schema validation interceptions (1 case). The complete interception type table is provided in Appendix D.9. Rather than a system limitation, these interceptions highlight the efficacy of SkCC’s fail-fast design: by intercepting malformed or dangerous skills at compile time, the system prevents them from polluting the agent’s context window or causing unpredictable

runtime errors. This compile-time safety guarantee distinguishes SkCC from runtime-only safety mechanisms that rely on the agent’s own judgment—an approach that is inherently unreliable given LLMs’ tendency to follow instructions literally [9].

5 Conclusion

We presented SkCC, a skill compilation framework that introduces classical compiler design into agent skill development. Through a four-phase pipeline with a strongly-typed SkIR, Anti-Skill Injection, and polymorphic backend emission, SkCC achieves portable and secure skill deployment across heterogeneous agent frameworks. Our evaluation across four platforms validates this architecture: format adaptation is a functional necessity, not a cosmetic preference. SkCC yielded double-digit pass rate improvements on format-sensitive models and significant runtime token savings on format-tolerant ones, with ablation studies confirming that gains are strictly model-dependent. Compared to retrieval-based skill refinement [25], SkCC’s compilation delivers substantially larger gains on both Claude (+12.2pp vs. +8.1pp) and Kimi (+13.5pp vs. +3.3pp). The compiler compiles skills in under 10ms while automatically hardening 94.8% of evaluated skills against critical vulnerabilities before deployment. SkCC represents a paradigm shift from manual per-platform rewriting to systematic compiler-driven adaptation. As the agent ecosystem diversifies, the portability and security guarantees of this architecture become increasingly valuable. Future work will pursue automated anti-pattern discovery from vulnerability corpora, semantic-level adaptation informed by runtime feedback, and ecosystem integration through WebAssembly bindings for real-time IDE validation. We believe compiler-based skill development, with its emphasis on type safety, semantic validation, and security hardening, will become as essential to agent developers as traditional compilers are to software developers today.

Acknowledgments

This work is a preprint currently under review. In accordance with the conference’s preprint policy, this version is made available on arXiv.

References

- [1] Affan-m. 2026. Everything Claude Code: The agent harness performance optimization system. <https://github.com/affan-m/everything-claude-code>
- [2] Agent Skills. 2026. SKILL.md Explained: How to Structure Your Product for AI Agents — Add Guardrails and Common Pitfalls. <https://www.gitbook.com/blog/skill-md>
- [3] Agent Skills. 2026. SKILL.md Specification and Progressive Disclosure Mechanism. <https://deepwiki.com/agentskills/agentskills/2.2-skill-md-specification>
- [4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools* (1st ed.). Addison-Wesley, Reading, MA.

- [5] Salaheddin Alzubi, Noah Provenzano, et al. 2026. EvoSkill: Automated Skill Discovery for Multi-Agent Systems. arXiv:2603.02766
- [6] Anthropic. 2026. Anthropic Skills: Public repository for Agent Skills. <https://github.com/anthropics/skills>
- [7] Anthropic. 2026. Claude API Docs: Prompting Best Practices — Structure Prompts with XML Tags. <https://platform.claude.com/docs/en/build-with-claude/prompt-engineering/claude-prompting-best-practices>
- [8] Anthropic. 2026. Claude Code Overview. <https://code.claude.com/docs/en/overview>
- [9] Luca Beurer-Kellner, Alexey Kudrinskii, Maciej Milanta, Kasper B. Nielsen, Hila Sarkar, and Liat Tal. 2026. Snyk Finds Prompt Injection in 36%, 1467 Malicious Payloads in a ToxicSkills Study of Agent Skills Supply Chain Compromise. <https://snyk.io/blog/toxicskills-malicious-ai-agent-skills-clawhub/>
- [10] Le Chen, Erhu Feng, Yubin Xia, Haibo Chen, et al. 2026. SkVM: Revisiting Language VM for Skills Across Heterogeneous LLMs and Harnesses. arXiv:2604.03088
- [11] Darren Edge, Ha Trinh, Newman Cheng, et al. 2024. From Local to Global: A Graph RAG Approach to Query-Focused Summarization. arXiv:2404.16130
- [12] getSentry. 2026. Sentry Skills: Agent Skills used by the Sentry team for development. <https://github.com/getsentry/skills>
- [13] Google. 2026. Gemini CLI Documentation. <https://google-gemini.github.io/gemini-cli/docs/>
- [14] Harbor Framework Team. 2026. Harbor: A Framework for Evaluating and Optimizing Agents and Models in Container Environments. <https://github.com/harbor-framework/harbor>
- [15] Jia He, Mukund Rungta, et al. 2024. Does Prompt Formatting Have Any Impact on LLM Performance? arXiv:2411.10541
- [16] Improving Agents. 2025. Which Nested Data Format Do LLMs Understand Best? JSON vs. YAML vs. XML vs. Markdown. <https://www.improvingagents.com/blog/best-nested-data-format/>
- [17] Sehoon Kim, Suhong Moon, et al. 2024. An LLM Compiler for Parallel Function Calling. In *International Conference on Machine Learning (ICML)*. arXiv:2312.04511
- [18] Kimi. 2026. Kimi CLI Documentation. <https://moonshotai.github.io/kimi-cli/en/guides/getting-started.html>
- [19] Steve Kinney. 2026. Prompt Engineering Across the OpenAI, Anthropic, and Gemini APIs. <https://stevekinney.com/writing/prompt-engineering-frontier-llms>
- [20] A. B. V. Kumar. 2026. Deep Dive SKILL.md (Part 1/2): Negative Boundaries and Triggering Accuracy. Medium, March 17, 2026. <https://abvijaykumar.medium.com/deep-dive-skill-md-part-1-2-09fc9a536996>
- [21] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization (CGO)*. 75–86. doi:10.1109/CGO.2004.1281665
- [22] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *International Symposium on Code Generation and Optimization (CGO)*. 2–14. doi:10.1109/CGO51591.2021.9370308
- [23] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2021. The Deep Learning Compiler: A Comprehensive Survey. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2021), 708–727. doi:10.1109/TPDS.2020.3030548
- [24] Xiangyi Li, Wenbo Chen, et al. 2026. SkillsBench: Benchmarking How Well Agent Skills Work Across Diverse Tasks. arXiv:2602.12670
- [25] Yujian Liu, Jiabao Ji, Li An, Tommi Jaakkola, Yang Zhang, and Shiyu Chang. 2026. How Well Do Agentic Skills Work in the Wild: Benchmarking LLM Skill Usage in Realistic Settings. arXiv:2604.04323 [cs.CL] <https://arxiv.org/abs/2604.04323>
- [26] Yuanye Liu, Jiahang Xu, et al. 2025. Beyond Prompt Content: Enhancing LLM Performance via Content-Format Integrated Prompt Optimization. arXiv:2502.04295
- [27] Benjamin Mikek, Danylo Vashchilenko, et al. 2026. Agentic Code Optimization via Compiler-LLM Cooperation. arXiv:2604.04238
- [28] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA.
- [29] NextLevelBuilder. 2026. UI/UX Pro Max Skill: An Agent Skill for UI/UX design tasks. <https://github.com/nextlevelbuilder/ui-ux-pro-max-skill>
- [30] OpenAI. 2025. Structured Outputs and Format Tax Elimination. <https://platform.openai.com/docs/guides/structured-outputs>
- [31] OpenAI. 2026. Codex Documentation. <https://developers.openai.com/codex>
- [32] Roy Philip. 2025. JSON vs. XML: A Data-Driven Analysis of LLM Parsing Efficiency. <https://royphilip.xyz/blog/json-vs-xml-llm-showdown>
- [33] Reddit r/ClaudeAI. 2026. Anthropic's Official Take on XML-Structured Prompting as the Core Strategy. <https://www.reddit.com/r/ClaudeAI/comments/1psxuv7/>
- [34] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Conference on Empirical Methods in Natural Language Processing and International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. doi:10.18653/v1/D19-1410
- [35] John Strong, Joseph Wegstein, Alan Titter, et al. 1958. The Problem of Programming Communication with Changing Machines: A Proposed Solution. *Commun. ACM* 1, 8 (1958), 12–18. doi:10.1145/368892.368915
- [36] Weihang Su, Jianming Long, et al. 2026. Skill Retrieval Augmentation for Agentic AI. arXiv:2604.24594
- [37] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy (S&P)*. 48–62. doi:10.1109/SP.2013.13
- [38] TechforHumans. 2025. Effective Prompt Engineering: Mastering XML Tags for Clarity, Precision, and Security in LLMs. Medium, June 18, 2025. <https://medium.com/@TechforHumans/effective-prompt-engineering-mastering-xml-tags-for-clarity-precision-and-security-in-llms-992cae203fdc>
- [39] Hao Wang, Niels Mündler, Mark Vero, Jingxuan He, Dawn Song, and Martin Vechev. 2026. SecPI: Secure Code Generation with Reasoning Models via Security Reasoning Internalization. arXiv:2604.03587 [cs.CR] <https://arxiv.org/abs/2604.03587>
- [40] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, et al. 2024. A Survey on Large Language Model Based Autonomous Agents. *Frontiers of Computer Science* 18, 6 (2024), 186345. doi:10.1007/s11704-024-40231-1
- [41] Renxi Wang, Xudong Han, et al. 2025. ToolGen: Unified Tool Retrieval and Calling via Generation. In *International Conference on Learning Representations (ICLR)*. arXiv:2410.03439
- [42] Xingyao Wang, Simon Rosenberg, et al. 2026. The OpenHands Software Agent SDK: A Composable and Extensible Foundation for Production Agents. In *Conference on Machine Learning and Systems (MLSys)*. arXiv:2511.03690
- [43] Michael Wooldridge and Nicholas R. Jennings. 1995. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review* 10, 2 (1995), 115–152. doi:10.1017/S0269888900008122
- [44] Renjun Xu, Yang Yan, et al. 2026. Agent Skills for Large Language Models: Architecture, Acquisition, Security, and the Path Forward. arXiv:2602.12430
- [45] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Advances in Neural Information Processing Systems (NeurIPS)*. doi:10.48550/arXiv.2305.10601

A Implementation Details

SkCC is implemented in Rust and organized into four crates:

nexa-skill-cli. CLI entry point using `clap` for argument parsing and `miette` for diagnostic rendering. Provides commands: `build` (compile skills), `check` (validate without emitting), `validate` (strict validation), `init` (scaffold new skill from template), `list` (enumerate skills in directory), `index` (generate routing manifest), and `clean` (remove compiled artifacts).

nexa-skill-core. Core compilation logic organized into six modules: `frontend` (frontmatter parsing, Markdown event-stream parsing, AST construction), `ir` (SkIR definition, IR builder, type mapper, nested data detector), `analyzer` (schema validator, MCP dependency checker, permission auditor, anti-skill injector), `backend` (Emitter trait, EmitterRegistry, four platform-specific Emitters, routing manifest generator), `error` (diagnostic types with source spans), and `security` (security baseline, permission types, security level classification).

nexa-skill-templates. Askama template engine with Jinja2-style compile-time-validated templates: `claude_xml.j2` (XML-tagged SKILL.md for Claude), `codex_md.j2` (XML-tagged Markdown for Codex), `gemini_md_v2.j2` (Markdown with conditional YAML for Gemini), and `kimi_md.j2` (full Markdown for Kimi). Each template is paired with a context struct that maps SkIR fields to template variables.

npm-nexa-skill-compiler. npm wrapper package that downloads the precompiled Rust binary and exposes the `nsc` command globally for Node.js users, enabling integration with JavaScript-based agent toolchains.

Key dependencies and design choices.

- `Arc<str>` for zero-copy string sharing across compilation phases and Emitters.
- `serde` and `serde_json` for SkIR serialization and JSON Schema handling.
- `serde_yaml` for YAML frontmatter parsing and YAML asset generation.
- `pulldown-cmark` for Markdown event-stream parsing.
- `sha2` for source file integrity hashing.
- `chrono` for compilation timestamp recording.
- `askama` for compile-time template validation.

Memory optimization. SkIR uses `Arc<str>` for all string fields shared across Emitters, enabling zero-copy cloning. The `ValidatedSkIR` wrapper adds only a `Vec<Diagnostic>` without duplicating the underlying IR. For batch compilation of large skill corpora (e.g., 233 skills), the compiler processes skills sequentially with per-skill memory deallocation, keeping peak memory usage below 50MB.

Compilation performance. On a standard development machine (Intel i9-13900H, 32GB RAM), single-skill compilation (all four targets) completes in under 10ms, with the Analyzer phase accounting for approximately 40% of total time. Batch compilation of 225 skills completes in approximately 1.8 seconds (8ms average per skill), demonstrating linear scaling with corpus size.

B Design Artifacts

B.1 Qualitative Comparison with Related Systems

Table 8. Qualitative Comparison with Related Systems

Method	Format Adapt.	Security	Multi-Platform	Complexity
SkVM [10]	Semantic only	×	✓	$O(m \times n)$
CFPO [26]	Iterative	×	×	$O(k \times m)$
Wild Retrieval [25]	Query-specific	×	✓	$O(m \times n)$
SkCC	IR-driven	✓	✓	$O(m + n)$

B.2 Key Insights from Evaluation

Our experiments demonstrate consistent compilation gains across four platforms, with gains proven model-specific through ablation studies. Engineering metrics confirm compilation latency under 10ms, Anti-Skill Injection coverage of 94.8%, and runtime token savings of 10–46%. Two system-level insights emerge from these results.

Format Tolerance vs. Format Sensitivity. Compilation gains correlate with the underlying model’s format sensitivity. Claude shows the largest improvement ($d = 0.60$) because its training distribution heavily favors XML-tagged inputs; the compiler aligns structural encoding with parsing expectations. Gemini shows minimal reward improvement ($d \approx 0$) because it is relatively format-tolerant. This validates SkCC’s core premise: different models have different format preferences, and a one-size-fits-all SKILL.md inevitably underperforms on format-sensitive platforms.

Static Overhead vs. Dynamic Efficiency. Compilation increases static skill size by 4–25% yet reduces dynamic token consumption by 10–46% during execution. Structured formats serve as cognitive scaffolding, reducing parsing ambiguity and trial-and-error. The compiler invests tokens upfront in structural clarity, which the model repays through more efficient execution. The true value of skill compilation lies not in compression but in structural investment: spending tokens on clarity to save tokens on execution.

B.3 Platform-Specific Emission Details

The following describes the format hardening strategy for each target platform, as referenced in Section 3.4.

Claude (XML Semantic Layering). Leveraging Anthropic’s documented preference for XML-tagged prompts [7], this target wraps all structural elements in semantic XML tags: procedures in `<execution_steps>/<step>` with order and critical attributes, constraints in `<strict_constraints>/<anti_pattern>`, and examples in `<examples>/<example>` with nested `<input>` and `<output>`. This semantic layering reduces misinterpretation and improves reasoning accuracy by up to 23%.

Codex (XML-Tagged Markdown). This target produces a hybrid XML-tagged Markdown format: instructions in `<skill>/<instructions>`, constraints in `<constraints>/<forbidden>`, and examples in `<examples>/<example>`. This provides structural markers for parsing while avoiding the JSON “format tax” that degrades GPT-series model performance [30]. Structured output enforcement is delegated to the OpenAI API’s Structured Outputs feature, decoupling reasoning from formatting.

Gemini (Markdown + Conditional YAML). Applying the nested data detection flag from the Analyzer phase, this target conditionally renders deeply nested schemas (depth ≥ 3) as YAML code blocks while keeping shallow structures in standard Markdown. When YAML optimization is triggered, separate YAML asset files are generated for complex nested structures. This adaptive strategy leverages YAML’s superior parsing accuracy (51.9% vs JSON’s 43.1%) for nested data while avoiding unnecessary format switching for simple structures.

Kimi (Full Markdown Preservation). This target preserves all skill details in comprehensive Markdown without simplification or format optimization, leveraging Kimi’s ultra-long context window capability. No YAML optimization or content truncation is applied, ensuring maximum information fidelity for platforms that can process full skill content without token budget constraints.

C Design Artifacts

C.1 SkIR Example

Listing 1 shows a simplified SkIR instance for a “github-api-client” skill, illustrating how the raw Markdown source is normalized into a structured, platform-agnostic representation.

Listing 1. Simplified SkIR for a “github-api-client” skill. Note the `anti_skill_constraints` field, which was automatically injected by the Analyzer, and the structured procedures array.

```

1 {
2   "name": "github-api-client",
3   "version": "1.0.0",
4   "description": "Interact with GitHub REST API",
5   "mcp_servers": ["github-mcp"],
6   "input_schema": {
7     "type": "object",
8     "properties": {
9       "repo": { "type": "string" },
10      "action": { "type": "string",
11                "enum": ["create_issue", "list_prs"] }
12    }
13  },

```

```

14  "security_level": "high",
15  "hitl_required": true,
16  "permissions": [
17    { "kind": "network",
18      "scope": "https://api.github.com/*",
19      "read_only": false }
20  ],
21  "procedures": [
22    { "order": 1,
23      "instruction": "Validate GitHub token from env",
24      "is_critical": true },
25    { "order": 2,
26      "instruction": "Construct REST request" },
27    { "order": 3,
28      "instruction": "Execute HTTP POST to GitHub API" }
29  ],
30  "anti_skill_constraints": [
31    {
32      "source": "anti-skill-injector",
33      "content": "Never execute HTTP without timeout...",
34      "level": "warning",
35      "scope": "global"
36    }
37  ],
38  "requires_yaml_optimization": false,
39  "mode": "sequential"
40  }

```

C.2 Anti-Skill Injection Rules

Table 9. Anti-Skill Injection Rules

Anti-Pattern	Trigger Keywords	Injected Constraint
HTTP safety	HTTP, GET, POST, fetch, request	Never execute HTTP without timeout (10s). Max 3 retries on 403.
HTML Parse safety	BeautifulSoup, HTML parse, scrape	Do not parse raw JS variables with HTML parsers. Fallback to Regex.
Destructive DB safety	DROP, DELETE, TRUNCATE	No destructive DB ops without user confirmation. Show affected rows.
Loop safety	while, loop, repeat	All loops must have max iteration limit (1000).

C.3 Four-Platform Format Divergence

Listing 2 illustrates the format divergence across Emitters for a single `SkIR`.

Listing 2. Format divergence across four Emitters for a single `SkIR`. Note the Gemini emitter’s conditional YAML rendering (triggered by nesting depth ≥ 3) and the consistent presence of anti-skill constraints across all formats.

```

1  \SkIR{} (platform-independent)
2  |-- name: "data-migration"
3  |-- procedures: [3 steps]
4  |-- input_schema: { nested depth = 4 }
5  +-- anti_skill_constraints: [1 HTTP safety]
6
7  Compiled Outputs:
8
9  Claude:  <agent_skill>
10           <execution_steps>
11             <step order="1" critical="true">...</step>
12           </execution_steps>

```

```
13         <strict_constraints>
14             <anti_pattern source="anti-skill-injector">
15                 ...
16             </anti_pattern>
17         </strict_constraints>
18     </agent_skill>
19
20     Codex: <skill name="data-migration">
21         <instructions>...</instructions>
22         <constraints>
23             <forbidden>...</forbidden>
24         </constraints>
25     </skill>
26
27     Gemini: # data-migration
28             ## Procedures
29             1. ... **[CRITICAL]**
30             ## Parameter Schema (YAML Optimized)
31             ```yaml
32             type: object
33             properties:
34                 migration_config:
35                     type: object
36                     properties:
37                         source_db:
38                             type: object
39                             properties:
40                                 host: { type: string }
41             ...
42
43     Kimi:    # data-migration
44             ## Description
45             ...
46             ## Procedures
47             1. ... **[CRITICAL]**
48             ## Parameter Schema
49             - `migration_config.source_db.host` (string): ...
```

D Complete Experimental Data

D.1 Four-Model Comparison Summary

Table 10. Four-Model Comparison Summary

Model	Paired	Δ Rwd.	p	d	Verdict
claude-opus-4-6	22–27	+0.26–0.27	0.0096**	0.59–0.60	C $\gg$ O
kimi-k2.5	74	+0.142	0.0063**	0.33	C > O
gpt-5.3-codex	26	+0.067	—	—	C > O
gemini-2.5-pro	18	+0.019	—	—	C > O

D.2 Claude Code — Complete Data

Table 11. Claude Code — Complete Paired Statistical Tests

Cmp.	<i>n</i>	Mean Δ	W/T/L	<i>t</i>	<i>p</i>	<i>d</i>
C vs V	23	+0.265	7/16/0	2.837	0.0096**	0.592
C vs O	22	+0.274	7/15/0	2.820	0.0103*	0.601
O vs V	26	+0.002	3/21/2	0.031	0.9756	0.006

Task classification (22 paired C vs O): Compiled Better: 7 tasks (31.8%) — 6 flipped from reward=0 to reward=1; Compiled Worse: 0 tasks (0%); Tie: 15 tasks (68.2%).

D.3 Kimi CLI — Complete Data

Table 12. Kimi CLI — Complete Statistical Tests

Test	Statistic	<i>p</i>	Sig.
Paired t-test	$t = 2.815$	0.0063	$p < 0.01$
Wilcoxon signed-rank	$W = 22.0$	0.0050	$p < 0.01$
Non-tie only ($n = 17$)	$t = 3.449$	0.0033	$p < 0.01$
Cohen’s d (paired)	0.327	—	Small effect

Task classification (74 paired): Compiled Better: 13 discriminative wins (81.25%); Compiled Worse: 3 (18.75%); Tie: 58 (78.4%); 13 tasks flipped from reward=0 to reward=1.

D.4 Ablation Study — Full Data

Table 13. Ablation Study — Complete Metrics

Model	Framework	Backend	Succ. (O/C)	Paired	Rwd. (O/C)	<i>p</i>	<i>d</i>	Eff.
kimi-k2.5	Kimi CLI	Kimi	26/75 → 36/76	74	0.341 → 0.483	0.0063	+0.33	C > O
glm-5.1	OpenHands	Kimi	43/88 → 44/88	32	—	0.857	−0.03	C ≈ O
deepseek-v4-flash	OpenHands	Kimi	64/88 → 65/88	50	—	0.2561	−0.14	O > C

D.5 Expansion Overhead by Complexity

Table 14. Expansion Overhead by Complexity

Complexity	Claude Ovhd.	Kimi Ovhd.	Claude w/ Reduction	Kimi w/ Reduction
Simple (avg 298t)	+95.0%	+37.4%	0/8 (0%)	0/8 (0%)
Medium (avg 819t)	+43.0%	+14.6%	4/74 (5.4%)	36/74 (48.6%)
Complex (avg 2765t)	+11.4%	−3.1%	31/143 (21.7%)	101/143 (70.1%)

D.6 Claude Code — Full Token Consumption

Table 15. Claude Code — Full Token Consumption Comparison

Condition	Succ. Tasks	Input T.	Output T.	Cache T.	Total	Task Avg.
Vanilla	34	19.5M	421K	17.2M	~19.9M	~0.59M
Original	40	32.9M	574K	30.1M	~33.4M	~0.84M
Compiled	29	18.3M	459K	15.8M	~18.7M	~0.65M

D.7 Anti-Skill Injection — Full Statistics

Table 16. Anti-Skill Trigger Statistics (233 skills)

Metric	Value
Total skills	233
Skills triggering Anti-Skill	221 (94.8%)
Skills not triggering	12 (5.2%)

D.8 Rule Trigger Distribution — Full Data

Table 17. Rule Trigger Distribution (Full)

Anti-Skill Rule	Triggered	Keywords	Example Constraint
HTTP safety	212 (91.4%)	HTTP, GET, POST, fetch, request	Timeout (10s), max 3 retries on 403
Loop safety	104 (44.6%)	while, loop, repeat	Max iteration limit (1000)
DB safety	78 (33.5%)	DROP, DELETE, TRUNCATE	No destructive ops without confirmation
Parse safety	2 (0.9%)	BeautifulSoup, HTML parse, scrape	No parsing raw JS with HTML parsers

D.9 Compilation Interception Types

Table 18. Compilation Interception Types

Interception Type	Cnt.	Description	Example Skills
YAML format violation	5	Frontend rejected non-standard frontmatter	senior-java, senior-data-engineer, threejs (×2), data-reconciliation
Security check interception	4	Dangerous operations or sensitive content	ssh-penetration-testing, restclient-migration, jakarta-namespace, spring-security-6
Schema validation interception	1	IR builder found illegal field types	nlp-research-repo-package-installment