SkillCompiler: A Unified Compilation Framework for Cross-Platform LLM Agent Skills

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

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

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

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

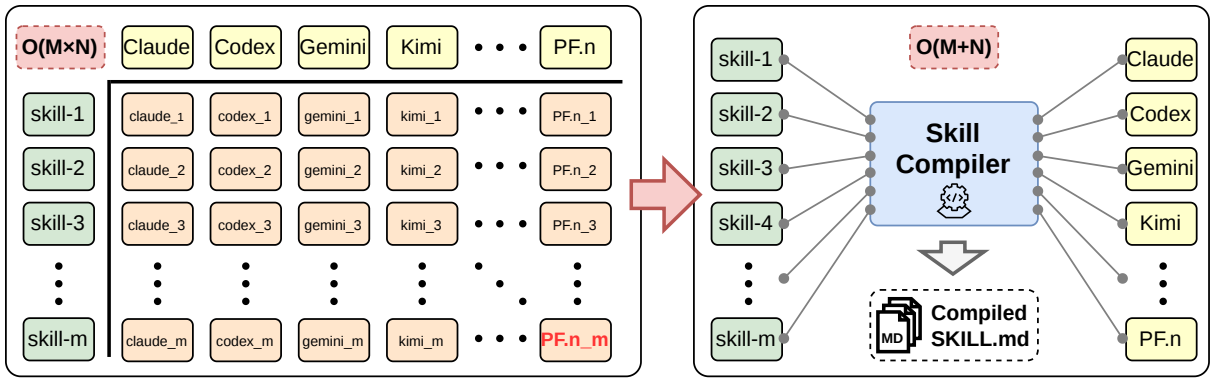

Figure 1. Complexity reduction from $O(m \times n)$ to $O(m + n)$. Traditional per-platform rewriting (left) requires $m \times n$ manual adaptations. SkillCompiler (right) decouples skills and platforms through a shared IR, requiring only m skill sources and n Emitter implementations.

Abstract

LLM-Agents have evolved into autonomous systems for complex task execution, with the SKILL.md specification and progressive disclosure emerging as the *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. Currently, nearly all skills exist as a single, format-agnostic Markdown version. Manual per-platform rewriting creates an unsustainable $O(m \times n)$ maintenance burden, while an audit of 3,984 community skills reveals that 37% contain security vulnerabilities. To address this, we present SkillCompiler, a compilation framework that introduces classical compiler design into agent skill development. Through a four-phase pipeline of frontend format validation, intermediate representation construction, automatic safety constraint enforcement (Anti-Skill Injection), and polymorphic backend emission, SkillCompiler 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. Engineering metrics validate the system's pragmatism: achieving sub-10ms compilation latency per skill, a 94.8% proactive security trigger rate, and a 10–46% improvement in runtime token efficiency across platforms.

CCS Concepts: • **Computing methodologies** → **Natural language processing**; *Intelligent agents*; • **Software and its engineering** → **Compilers**; • **Security and privacy** → *Software security engineering*.

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

https://github.com/Nexa-Language/Skill-Compiler
https://skcc.nexa-lang.com/

1 Introduction

The rapid advancement of large language models (LLMs) has catalyzed a new generation of autonomous agent systems [35, 38, 40]. For now, frameworks such as Anthropic Claude Code [6], OpenAI Codex [26], Google Gemini CLI [10], and Kimi CLI [15] provide terminal-based agent environments where LLMs interact with tools, file systems, and external services. Skills, structured prompt artifacts following the

Agent Skills '26, San Jose, CA, USA
2026. https://doi.org/https://doi.org/10.48550/arXiv.2605.03353

SKILL.md specification [2], have emerged as the *de facto* standard for encoding domain-specific knowledge, employing progressive disclosure [39] that loads lightweight metadata at initialization and retrieves full content on demand. EvoSkill [4] further advances this paradigm by automatically discovering and refining skills through iterative failure analysis.

However, different LLM-backed agent platforms exhibit starkly different sensitivities to prompt formatting, and the current skill ecosystem assumes format-agnostic delivery. Identical content in different structural formats causes dramatic performance variation [12], with specific platforms showing strong preferences: XML Format for Claude [5], XML-tagged Markdown for GPT-series models to avoid the "format tax" [25], and YAML-augmented Markdown for nested data on certain architectures [13]. Beyond format compatibility, the skill ecosystem faces an equally pressing security challenge. Snyk's audit of 3,984 community skills from ClawHub [7] 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 [1], yet most existing skills lack such constraints. Manual per-platform rewriting creates an unsustainable $O(m \times n)$ maintenance burden.

We present SkillCompiler, a systematic skill compilation framework that introduces classical compiler design principles into agent skill development. The key insight is that the skill adaptation problem mirrors the classical compiler's target-platform diversity problem: just as LLVM [18] reduces the $O(m \times n)$ burden of supporting m source languages on n target architectures to $O(m + n)$ through a shared intermediate representation, SkillCompiler applies the same structural transformation pipeline to systematically address both formatting adaptation and security enhancement. SkillCompiler implements a four-phase pipeline (Frontend parsing, IR construction, Analyzer validation, and Backend emission) that transforms a unified SKILL.md source into platform-native skill artifacts, with each phase operating on a strongly-typed intermediate representation (SkillIR) that decouples source handling from target platform handling. This architecture reduces adaptation complexity from $O(m \times n)$ to $O(m + n)$.

Our key contributions are as follows:

- **We propose SkillCompiler**, a four-phase skill compilation framework that automates the adaptation of skills to multiple agent platforms through a strongly-typed intermediate representation, semantic analysis with Anti-Skill Injection, and polymorphic backend emission, reducing the maintenance burden from $O(m \times n)$ to $O(m + n)$.
- **We demonstrate consistent performance improvements across four mainstream agent platforms** through empirical evaluation on SkillsBench: compiled skills improve pass rates from 21.1% to 33.3% on Claude Code ($p = 0.0103$, $d = 0.60$), from 35.1% to 48.7% on Kimi CLI ($p = 0.0063$, $d = 0.33$), with reward improvements of +0.067 on Codex CLI and +0.019 on Gemini CLI.
- **We conduct comprehensive engineering analyses** showing compilation latency under 10ms per skill, 94.8% Anti-Skill Injection trigger rate across 233 skills, token efficiency improvements of 10–46% across platforms, and ablation experiments validating that compilation gains are model-specific and confirm the necessity of platform-tailored backend Emitters.

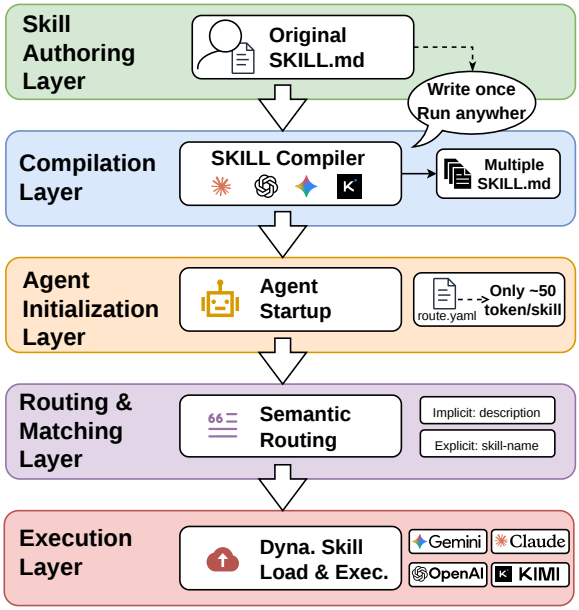

Figure 2. Agent workflow with SkillCompiler integration. Skills are authored once as SKILL.md, compiled to platform-native formats, and loaded via progressive routing manifests at agent initialization.

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 [2] introduced SKILL.md as a portable specification with YAML frontmatter and a Markdown body, enabling progressive disclosure [39] where lightweight metadata is loaded at initialization and full content retrieved on demand. EvoSkill [4] advanced this paradigm through automatic skill discovery and refinement via iterative failure analysis. Recent work has expanded skill retrieval through generation-based [36], augmentation-based [31], graph-based [9], and embedding-based [29] approaches. 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 SkillCompiler's platform-specific emission strategies. He et al. [12] demonstrated up to 40% performance variation from format changes alone, while Liu et al. [22] 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 [5], reporting up to 23% accuracy improvement [27, 28], with practitioners developing systematic XML-based prompt engineering methodologies [33]; OpenAI's GPT-series models suffer from a "format tax" when parsing JSON-formatted inputs [16, 25]. For nested data, YAML achieves the highest parsing accuracy (51.9%) compared to JSON (43.1%) and XML (33.8%) [13]. On the security dimension, Snyk's ToxicSkills study [7] found 37% of 3,984 community skills contain security vulnerabilities, while the SKILL.md specification [1] recommends negative boundaries rarely followed in practice [17]; recent work on secure code generation via reasoning internalization [34] further underscores the urgency of compile-time safety enforcement for agent skills. Collectively, these works establish that format sensitivity is real and platform-dependent, manual per-platform adaptation is an unsustainable $O(m \times n)$ burden, and security vulnerabilities in skills are widespread. No existing system simultaneously addresses all three challenges through a unified compilation framework.

Traditional Compilation Pipeline. SkillCompiler's design draws from classical compiler theory, particularly the multi-phase pipeline architecture established by Aho, Sethi, and Ullman [3] and advanced by Muchnick [24]. Rooted in the $O(m \times n)$ to $O(m + n)$ reduction principle first articulated by Strong et al. [30], it leverages a universal intermediate layer to decouple source languages from target machines. LLVM [18] demonstrated that a well-designed IR enables diverse optimizations independent of source or target specifics [19, 20], while SkillCompiler's Anti-Skill Injection mechanism parallels compiler-level security hardening such as stack canary insertion and bounds checking [32]. Recent work has extended compiler principles to LLM-based systems: Mikek et al. [23] proposed compiler-LLM cooperation for agentic code optimization, while Kim et al. [14] introduced LLMCompiler for parallel function calling orchestration. Earlier work on skill-specific compilation, notably SkVM [8], recognized the need for compilation-like processing of agent skills through a JVM-like architecture, but differs from SkillCompiler in three dimensions: SkVM follows a VM/runtime model rather than a classical compiler architecture, does not address the 37% vulnerability rate through security hardening, and focuses on semantic capability degradation rather than format-syntax adaptation.

Without compilation, supporting m skills across n platforms requires $m \times n$ manual adaptations. SkillCompiler reduces this to $m + n$ by introducing a shared intermediate representation (SkillIR) that decouples skill authoring from platform-specific formatting: each skill is authored once as a SKILL.md source, and each platform is supported by a single Emitter implementation. This architectural insight mirrors classical compiler design, where LLVM's intermediate representation enables supporting multiple source languages and target architectures without combinatorial explosion [18, 30].

3 SkillCompiler Design

3.1 Architecture Overview

SkillCompiler addresses the problem of adapting a unified skill source S (a SKILL.md file) to n heterogeneous agent platforms, each with distinct format preferences and security requirements. Without compilation, supporting m skills across n platforms requires $m \times n$ manual adaptations. SkillCompiler reduces this to $m+n$ by introducing a shared intermediate representation (SkillIR) that decouples skill authoring from platform-specific formatting: each skill is authored once as a SKILL.md source, and each platform is supported by a single Emitter implementation. This architectural insight mirrors classical compiler design, where LLVM's intermediate representation enables supporting multiple source languages and target architectures without combinatorial explosion [18, 30]. The compilation pipeline consists of four phases—Frontend parsing, IR construction, Analyzer validation, and Backend emission—with SkillIR serving as the central abstraction (Figure 3).

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 RawAST that serves as the input to IR construction.

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

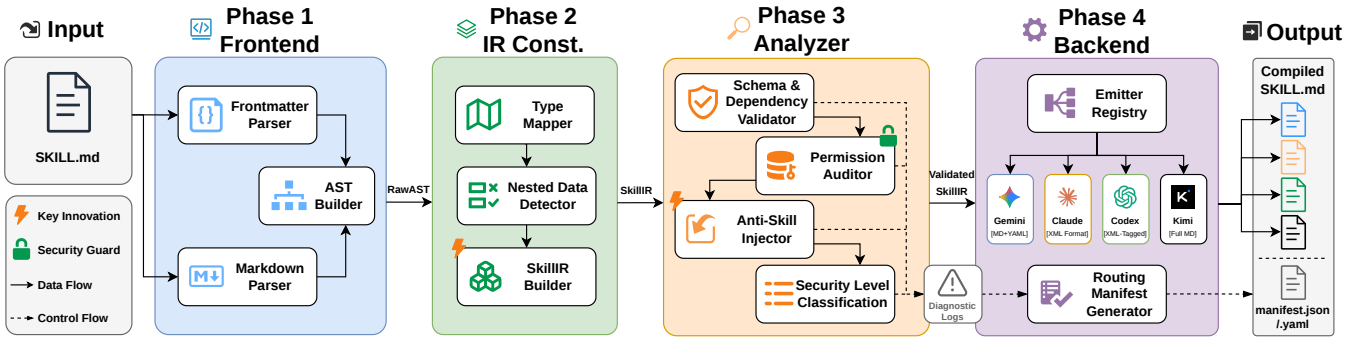

Figure 3. SkillCompiler's four-phase compilation pipeline. A unified SKILL.md source is parsed into a RawAST, transformed into a strongly-typed SkillIR, validated and hardened by the Analyzer, and emitted into platform-native formats through polymorphic Emitters.

Security & Control (HITL 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). SkillIR supports four execution modes: Sequential (ordered workflow), Alternative (mode-selector with multiple approaches), Toolkit (reference operations), and Guideline (unstructured recommendations). A concrete SkillIR instance is provided in Appendix B.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 NestedDataDetector sets the `requires_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 [13]).

3.3 Compile-time Semantic and Security Analysis

The Analyzer phase performs semantic validation and security enhancement on the SkillIR, producing a `ValidatedSkillIR` 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 (URL patterns, filesystem path patterns, operation patterns) 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 SkillCompiler. Rather than relying on skill authors to manually include defensive constraints, the AntiSkillInjector automatically scans procedure text for dangerous patterns and injects corresponding safety constraints into the SkillIR. 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 B.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.anti_skill_constraints` array, ensuring the safety instruction is rigidly embedded across all target formats. Across our evaluation corpus of 233 community skills, at least one anti-pattern rule matched in 94.8% of skills, indicating widespread absence of defensive constraints (Section 4.6).

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, SkillCompiler 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 [7].

3.4 Target Emission and Format Hardening

The Backend phase emits platform-native skill artifacts through a polymorphic Emitter architecture. Each Emitter implements a common interface (`Emitter trait`) that defines the target platform, output format, file extension, and optional asset generation. An EmitterRegistry manages all Emitter instances and retrieves the appropriate one for each target platform. For multi-target compilation, Phases 1–3 execute once to produce a single ValidatedSkillIR, which is then shared across all Emitters—this is the architectural mechanism that achieves the $O(m + n)$ complexity reduction.

Each target applies a distinct format hardening strategy based on empirical research about the underlying model's parsing preferences:

Claude (XML Semantic Layering). Leveraging Anthropic's documented preference for XML-tagged prompts [5], 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 [25]. 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.

Table 1. Experimental Setup: Agent Frameworks and Models

Framework	Underlying Model	Emitter Format
Claude Code	claude-opus-4-6	XML Format
Codex CLI	gpt-5.3-codex	XML-tagged Markdown
Gemini CLI	gemini-2.5-pro	Markdown + YAML
Kimi CLI	kimi-k2.5	Full Markdown

A side-by-side comparison of the four targets' output for a single SkillIR is provided in Appendix B.3.

Routing Manifest Generation. In addition to platform-specific skill artifacts, SkillCompiler generates a progressive routing manifest containing only the name, description, security level, and HITL flag for each skill (~50 tokens per skill). 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 [2, 39].

The system is implemented in Rust to ensure memory safety and zero-cost abstractions across the compilation pipeline (see Appendix A for library dependencies, crate structure, and CLI details).

4 Evaluation

We evaluate SkillCompiler on three dimensions: (1) whether compiled skills improve agent task completion rates compared to original skills across four mainstream agent platforms; (2) whether compilation gains are model-specific, validated through ablation experiments; and (3) compiler engineering metrics including compilation latency, token efficiency, Anti-Skill Injection coverage, and compilation interception analysis. All experiments use SkillsBench [21] as the benchmark, with 89 real-world programming and data analysis tasks.

4.1 Experiment Setup

Benchmark and Datasets. SkillsBench [21] 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, ecc-skills (everything-claude-code), sentry-skills (Sentry team), and ui-skill.

LLM Models and Agent Frameworks. Table 1 summarizes the four mainstream agent frameworks and underlying models used in our evaluation.

All experiments use the Harbor framework [11] for Docker-based task execution, with each agent framework (Claude Code, Kimi CLI, Gemini CLI, Codex CLI) running within

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**

Harbor-managed containers. Ablation experiments additionally test glm-5.1 and deepseek-v4-flash via the OpenHands SDK [37] integrated within Harbor.

Baselines. We compare three conditions where applicable: **Vanilla (V)**—agent operates without any skill, only the task description is provided; **Original (O)**—agent operates with the original, unmodified SKILL.md from SkillsBench; **Compiled (C)**—agent operates with the SkillCompiler-compiled SKILL.md for the target platform.

4.2 EX1: Main Results — Four-Model Compiled vs. Original Comparison

Claude Code (claude-opus-4-6). Compiled significantly outperforms both Vanilla ($p = 0.0096$, $d = 0.59$) and Original ($p = 0.0103$, $d = 0.60$) with medium-to-large effect sizes, while Original does not significantly outperform Vanilla ($p = 0.98$), suggesting that format-agnostic SKILL.md provides negligible improvement on Claude Code. Compiled never loses to Original (7 wins, 15 ties, 0 losses), with 6 of the 7 winning tasks flipping from reward=0 to reward=1, demonstrating that the XML-tagged format enables Claude to correctly follow instructions it fails to interpret in plain Markdown. The complete statistical test results are provided in Appendix C.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 (where one condition passes and the other fails), Compiled wins 13 (81.25%) while Original wins 3 (18.75%). Thirteen tasks flipped from reward=0 under Original to reward=1 under Compiled, demonstrating the transformative impact of format adaptation. The complete statistical test results are provided in Appendix C.3.

Codex CLI (gpt-5.3-codex). Among 26 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; token and time efficiency data are analyzed in Section 4.5.

Gemini CLI (gemini-2.5-pro). Compiled performs better on 3 tasks, worse on 2 tasks, and shows no meaningful difference on 13 tasks. The modest positive reward gain (+0.019) is expected: Gemini 2.5 Pro is relatively format-tolerant, and the GeminiEmitter's YAML optimization only activates when nesting depth ≥ 3, applying to a minority of tasks. Token and time efficiency data are analyzed in Section 4.5. Compilation appears to change which tasks are solvable rather than how many tasks are solvable.

Cross-Model Summary. SkillCompiler 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 with Δ reward, p-values, and Cohen's d is provided in Appendix C.1.

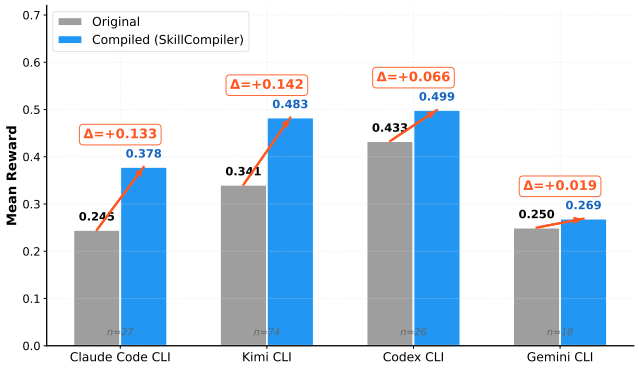

Figure 4. Cross-model comparison of Original vs. Compiled conditions across four agent frameworks. Error bars indicate standard error. Compiled consistently outperforms Original, with the largest gains on format-sensitive models (Claude, Kimi).

4.3 EX2: Ablation Study — Format Specificity

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

Table 3. Ablation Study — Cross-Model Format Specificity

Model	Pass% (O → C)	p-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

The same compiled output produces dramatically different results across models. On Kimi, the Kimi-compiled format yields a significant positive effect ($d = +0.33$, $p = 0.0063$), with the pass rate improving from 35.1% to 48.7%. On GLM-5, the effect is essentially neutral ($d = -0.03$, $p = 0.857$), indicating that GLM-5 is insensitive to the compiled format. On DeepSeek-v4-flash, the effect is slightly negative ($d = -0.14$, $p = 0.2561$), though not statistically significant. These results demonstrate that SkillCompiler's backend format is model-specific rather than a universal improvement, and that compilation gains are model-dependent with no one-size-fits-all optimal format. This provides empirical justification for SkillCompiler's multi-backend architecture: different agent frameworks require different compiled formats. The complete ablation table with all metrics is provided in Appendix C.4.

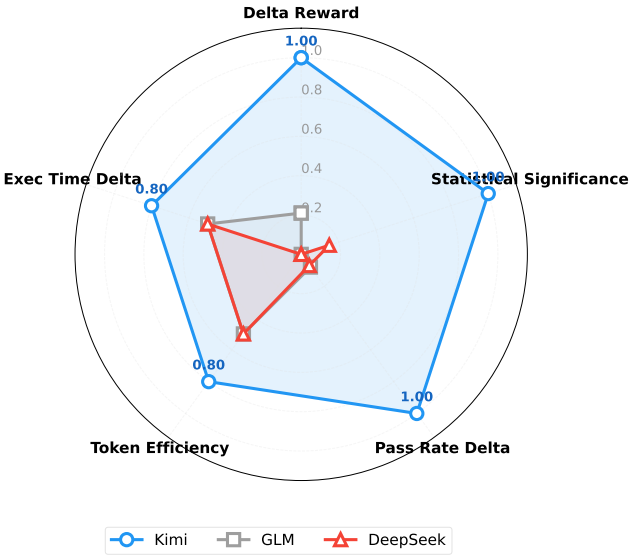

Figure 5. Ablation study radar chart: the same Kimi-compiled format produces divergent effects across three models, confirming model-specificity of compilation gains.

4.4 EX3: Compilation Performance

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

Table 4. Compilation Latency by Complexity

Complexity	n	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**

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. SkillCompiler's compilation speed is in the extremely fast tier for compilers, enabled by three design choices: no code generation, pure text transformation, and a Rust implementation with zero-copy parsing.

4.5 EX4: 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 is: Claude (XML) **+24.8%**, Codex (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 C.5.

While compilation introduces static structural overhead, this overhead translates to dynamic efficiency gains during execution. As shown below, the clearer structure reduces model trial-and-error and redundant output, resulting in net token savings of 10–46% across platforms. SkillCompiler compilation is not a compressor but a structural enhancer—the real token efficiency value comes from progressive disclosure via the routing manifest and reduced runtime ambiguity, not per-skill compression.

Table 5. Claude Code — Token Consumption Comparison

Cond.	Tasks	Total T.	Task Avg.
Vanilla	34	~19.9M	~0.59M
Original	40	~33.4M	~0.84M
Compiled	**29**	**~18.7M**	**~0.65M**

Real Token and Time Consumption in SkillsBench Experiments. On Claude Code, the compiled condition achieves the lowest per-task token consumption (0.65M vs. Original 0.84M) while obtaining higher reward (0.378 vs. 0.245), demonstrating that SkillCompiler compilation improves both task performance and token efficiency simultaneously.

The complete token consumption table with input/output/-cache breakdown is provided in Appendix C.6.

Similar efficiency gains are observed on Codex CLI and Gemini CLI. On Codex CLI, 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, 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). These results indicate that the structural overhead introduced by compilation is more than offset by reduced runtime token consumption, yielding net efficiency gains across all platforms.

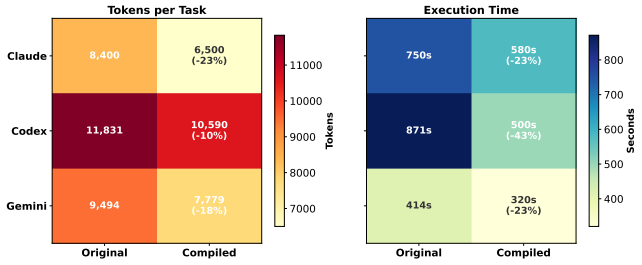

Figure 6. Cross-platform token and time efficiency heatmap. Compiled skills show consistent reductions in total tokens and execution time across all platforms. Claude token counts are reported in hundreds due to API measurement differences.

4.6 EX5: Anti-Skill Injection Trigger Analysis

SkillCompiler's AntiSkillInjector implements compile-time safety checking by automatically detecting dangerous patterns in skill content and injecting 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 C.7.

Table 6. 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%)**

Rule overlap is common: many skills trigger multiple rules simultaneously (HTTP + Loop + DB = most common combination), reflecting the typical characteristics of modern skills involving network operations, iterative processing, and data modification. The complete rule distribution table with trigger keywords and example constraints is provided in Appendix C.8.

SkillCompiler also implements deeper security checking via the SecurityAnalyzer: Baseline Security (detecting sensitive information leakage), Permission Analysis (validating file system, network, and process operation permissions), and Security Level Classification (categorizing skills as safe, warning, critical, or dangerous). This is the core of SkillCompiler's compile-time safety design—proactively identifying and hardening potential risks before skills are deployed to agent environments.

4.7 EX6: Compilation Interception Analysis

We 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 with descriptions and example skills is provided in Appendix C.9.

The Fail-Fast Principle. Rather than a system limitation, these 10 compilation interceptions highlight the efficacy of SkillCompiler's fail-fast design. By intercepting YAML format violations, dangerous operations, and schema mismatches at compile time, the system prevents malformed skills from polluting the agent's context window or causing unpredictable runtime errors. The three interception categories precisely validate SkillCompiler's three defense lines: format strictness (5 cases), security pre-interception (4 cases), and schema type safety (1 case). This compile-time safety guarantee distinguishes SkillCompiler 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 [7].

4.8 Results and Analysis

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%. Beyond these quantitative results, two system-level insights emerge.

Insight 1: 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, with diverse training formats reducing the marginal benefit of any single optimization. This validates SkillCompiler's core premise: different models have different format preferences, and a one-size-fits-all SKILL.md inevitably underperforms on format-sensitive platforms.

Insight 2: The Trade-off Between Static Overhead and 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. On format-sensitive platforms, this yields substantial returns in both task completion and token savings; on format-tolerant platforms, efficiency gains persist through reduced execution time. The true value of skill compilation lies not in compression but in structural investment: spending tokens on clarity to save tokens on execution.

5 Discussion

5.1 Limitations

Our evaluation reveals several limitations that warrant acknowledgment:

Anti-Skill Injection coverage. The current AntiSkillInjector implements four anti-pattern rules (HTTP safety, HTML parsing safety, destructive DB operations, infinite loop prevention). While these cover the most common dangerous patterns in the evaluated skill corpus (matching in 94.8% of 233 skills), they do not address critical security concerns such as prompt injection, data exfiltration, credential theft, and privilege escalation. The rule set is extensible through the AntiPattern struct, but adding new rules requires manual curation rather than automated discovery.

Backend Emitter maintenance. The ablation study confirms that compilation gains are strictly model-dependent: a format optimized for Kimi does not improve GLM-5 and slightly degrades DeepSeek. While SkillCompiler reduces the adaptation problem from $O(m \times n)$ to $O(m + n)$ — each new model requires only one Emitter implementation rather than per-skill rewriting — developers must still research each model's parsing preferences and build corresponding Emitters. Future work could explore automated format preference profiling to further reduce this overhead.

5.2 Future Work

Automated Security Hardening. The current AntiSkillInjector relies on manually curated rules. Future work could leverage LLM-based analysis of vulnerability corpora (e.g., Snyk, CVE databases) to automatically discover dangerous patterns and generate constraint templates, transforming the injector from a static rule engine into a continuously learning security system. Establishing a benchmark of intentionally malicious skills with before/after safety measurements would provide more rigorous validation of compile-time hardening.

Component-level Ablation. While our evaluation demonstrates overall compilation gains, we do not isolate the individual contributions of format adaptation, IR normalization, routing manifest optimization, and Anti-Skill Injection. A controlled ablation systematically enabling and disabling each component would clarify which features drive gains on each platform, guiding future optimization priorities. We also plan to expand evaluation coverage with additional backends and platforms.

Semantic and Runtime Adaptation. SkillCompiler currently adapts skills at the format-syntax level. Future work could extend SkillIR with model capability metadata to support semantic-level adaptation (instruction simplification, procedure decomposition, example difficulty adjustment). Integrating runtime feedback such as task completion rates and token consumption patterns could also enable iterative compilation optimization, bridging the gap between SkillCompiler's AOT approach and JIT-style runtime adaptation.

Ecosystem Integration. Moving beyond the CLI, we plan to implement Emitters for emerging agent frameworks (e.g., OpenHands, Copilot) and develop WebAssembly bindings for real-time skill validation and compilation directly within IDEs, reducing development friction and accelerating adoption of the SKILL.md standard.

6 Conclusion

We presented SkillCompiler, a skill compilation framework that introduces classical compiler design into agent skill development. Through a four-phase pipeline with a strongly-typed SkillIR, Anti-Skill Injection, and polymorphic backend emission, SkillCompiler reduces adaptation complexity from $O(m \times n)$ to $O(m+n)$ while addressing format sensitivity and security challenges.

Our evaluation across four platforms validates this architecture: format adaptation is a functional necessity, not a cosmetic preference. SkillCompiler 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. The compiler compiles skills in under 10ms while automatically hardening 94.8% of evaluated skills against critical vulnerabilities before deployment.

SkillCompiler represents a paradigm shift from manual per-platform rewriting to systematic compiler-driven adaptation. As the agent ecosystem diversifies, the $O(m + n)$ scalability of this architecture becomes increasingly valuable. 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.

References

[1] 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

[2] Agent Skills. 2026. SKILL.md Specification and Progressive Disclosure Mechanism. https://deepwiki.com/agentskills/agentskills/2.2-skill.md-specification

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools* (1st ed.). Addison-Wesley, Reading, MA.

[4] Salaheddin Alzubi, Noah Provenzano, et al. 2026. EvoSkill: Automated Skill Discovery for Multi-Agent Systems. arXiv:2603.02766

[5] 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

[6] Anthropic. 2026. Claude Code Overview. https://code.claude.com/docs/en/overview

[7] 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/

[8] 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

[9] Darren Edge, Ha Trinh, Newman Cheng, et al. 2024. From Local to Global: A Graph RAG Approach to Query-Focused Summarization. arXiv:2404.16130

[10] Google. 2026. Gemini CLI Documentation. https://google-gemini.github.io/gemini-cli/docs/

[11] Harbor Framework Team. 2026. Harbor: A Framework for Evaluating and Optimizing Agents and Models in Container Environments. https://github.com/harbor-framework/harbor

[12] Jia He, Mukund Rungta, et al. 2024. Does Prompt Formatting Have Any Impact on LLM Performance? arXiv:2411.10541

[13] 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/

[14] Sehoon Kim, Suhong Moon, et al. 2024. An LLM Compiler for Parallel Function Calling. In *International Conference on Machine Learning (ICML)*. arXiv:2312.04511

[15] Kimi. 2026. Kimi CLI Documentation. https://moonshotai.github.io/kimi-cli/en/guides/getting-started.html

[16] Steve Kinney. 2026. Prompt Engineering Across the OpenAI, Anthropic, and Gemini APIs. https://stevekinney.com/writing/prompt-engineering-frontier-llms

[17] 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

[18] 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

[19] 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

[20] 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

[21] Xiangyi Li, Wenbo Chen, et al. 2026. SkillsBench: Benchmarking How Well Agent Skills Work Across Diverse Tasks. arXiv:2602.12670

[22] Yuanye Liu, Jiahang Xu, et al. 2025. Beyond Prompt Content: Enhancing LLM Performance via Content-Format Integrated Prompt Optimization. arXiv:2502.04295

[23] Benjamin Mikek, Danylo Vashchilenko, et al. 2026. Agentic Code Optimization via Compiler-LLM Cooperation. arXiv:2604.04238

[24] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA.

[25] OpenAI. 2025. Structured Outputs and Format Tax Elimination. https://platform.openai.com/docs/guides/structured-outputs

[26] OpenAI. 2026. Codex Documentation. https://developers.openai.com/codex

[27] Roy Philip. 2025. JSON vs. XML: A Data-Driven Analysis of LLM Parsing Efficiency. https://royphilip.xyz/blog/json-vs-xml-llm-showdown

[28] Reddit r/ClaudeAI. 2026. Anthropic's Official Take on XML-Structured Prompting as the Core Strategy. https://www.reddit.com/r/ClaudeAI/comments/1psxuv7/

[29] 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

[30] John Strong, Joseph Wegstein, Alan Tritter, 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

[31] Weihang Su, Jianming Long, et al. 2026. Skill Retrieval Augmentation for Agentic AI. arXiv:2604.24594

[32] 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

[33] 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

[34] 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

[35] 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

[36] 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

[37] 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

[38] 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

[39] Renjun Xu, Yang Yan, et al. 2026. Agent Skills for Large Language Models: Architecture, Acquisition, Security, and the Path Forward. arXiv:2602.12430

[40] 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

SkillCompiler 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` (SkillIR 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 SkillIR 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 SkillIR 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. SkillIR uses `Arc<str>` for all string fields shared across Emitters, enabling zero-copy cloning. The `ValidatedSkillIR` 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 SkillIR Example

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

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

```
{
  "name": "github-api-client",
  "version": "1.0.0",
  "description": "Interact with GitHub REST API",
  "mcp_servers": ["github-mcp"],
  "input_schema": {
    "type": "object",
    "properties": {
      "repo": { "type": "string" },
      "action": { "type": "string",
```

```
        "enum": ["create_issue", "list_prs"] }
    }
  },
  "security_level": "high",
  "hitl_required": true,
  "permissions": [
    { "kind": "network",
      "scope": "https://api.github.com/*",
      "read_only": false }
  ],
  "procedures": [
    { "order": 1,
      "instruction": "Validate GitHub token from env",
      "is_critical": true },
    { "order": 2,
      "instruction": "Construct REST request" },
    { "order": 3,
      "instruction": "Execute HTTP POST to GitHub API" }
  ],
  "anti_skill_constraints": [
    {
      "source": "anti-skill-injector",
      "content": "Never execute HTTP without timeout...",
      "level": "warning",
      "scope": "global"
    }
  ],
  "requires_yaml_optimization": false,
  "mode": "sequential"
}
```

B.2 Anti-Skill Injection Rules

Table 7. 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).

B.3 Four-Platform Format Divergence

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

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

```
SkillIR (platform-independent)
  |-- name: "data-migration"
  |-- procedures: [3 steps]
  |-- input_schema: { nested depth = 4 }
  +-- anti_skill_constraints: [1 HTTP safety]

Compiled Outputs:

  Claude:  <agent_skill>
```

```
            <execution_steps>
              <step order="1" critical="true">...</step>
            </execution_steps>
            <strict_constraints>
              <anti_pattern source="anti-skill-injector">
                ...
              </anti_pattern>
            </strict_constraints>
          </agent_skill>

Codex:    <skill name="data-migration">
            <instructions>...</instructions>
            <constraints>
              <forbidden>...</forbidden>
            </constraints>
          </skill>

Gemini:   # data-migration
          ## Procedures
          1. ... **[CRITICAL]**
          ## Parameter Schema (YAML Optimized)
          ```yaml
          type: object
          properties:
            migration_config:
              type: object
              properties:
                source_db:
                  type: object
                  properties:
                    host: { type: string }
          ```

Kimi: # data-migration
          ## Description
          ...
          ## Procedures
          1. ... **[CRITICAL]**
          ## Parameter Schema
          - `migration_config.source_db.host` (string): ...
```

C Complete Experimental Data

C.1 Four-Model Comparison Summary

Table 8. 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** ≫ **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**

C.2 Claude Code — Complete Data

Table 9. Claude Code — Complete Paired Statistical Tests

Cmp.	n	Mean Δ	W/T/L	t	p	d
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 (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%).

C.3 Kimi CLI — Complete Data

Table 10. Kimi CLI — Complete Statistical Tests

Test	Statistic	p	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: 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.

C.4 Ablation Study — Full Data

Table 11. Ablation Study — Complete Metrics

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

C.5 Expansion Overhead by Complexity

Table 12. 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%)

C.6 Claude Code — Full Token Consumption

Table 13. 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**

C.7 Anti-Skill Injection — Full Statistics

Table 14. Anti-Skill Trigger Statistics (233 skills)

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

C.8 Rule Trigger Distribution — Full Data

Table 15. 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

C.9 Compilation Interception Types

Table 16. 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