Article

Not peer-reviewed version

Skills Are the New Apps– Now It's Time for Skill OS

Le Chen , Zichang Wang , Wenxin Zheng , Erhu Feng , Dong Du , Yubin Xia * , Haibo Chen

Posted Date: 13 February 2026

doi: 10.20944/preprints202602.1096.v1

Keywords: LLM agents; operating system

 doi:10.20944/preprints202602.1096.v1

Article

Skills Are the New Apps – Now It's Time for Skill OS

Le Chen, Zichang Wang, Wenxin Zheng, Erhu Feng, Dong Du, Yubin Xia * and Haibo Chen

Shanghai Jiao Tong University, China
* Correspondence: xiayubin@sjtu.edu.cn

Abstract

In a matter of months, skills have taken the agent ecosystem by storm: major LLM agent platforms (e.g., Cursor, Claude Code, Antigravity, Coze) now support skills natively, and skill counts across domains are growing at a breakneck pace. Although skills initially served as on-demand prompts to avoid excessive prompt context length, they differ fundamentally from conventional prompts in three respects: (i) skills embody *locality*: the very presence of a skill implies repeated reuse; (ii) skills encode concrete scenarios, yielding stronger determinism and verifiability than generic prompts; (iii) skills exhibit common requirements on the runtime environment, enabling systems to serve skills more effectively. These properties thus present both new opportunities and new challenges for system design. This paper surveys nearly 100,000 skills from public repositories and analyzes skill characteristics along the dimensions of structure, execution patterns, and system requirements. We argue that skills have become a new form of application and impose new demands on the underlying system; these demands will give rise to a new system abstraction, *Skill OS*, that treats skills as first-class execution artifacts and addresses caching, execution environment construction, global skill management, structured failure handling, and runtime security enforcement.

Keywords: LLM agents; operating system

1. Introduction

Agent skills are gaining rapid adoption as an extension mechanism for LLM-based systems. Mainstream coding agents such as Cursor [12], Claude Code [5], Google Antigravity [17], and Coze [11] support skills natively; open specifications and community repositories [1,2,4,34] are driving rapid growth in the number and variety of skills. In practice, a skill bundles triggering metadata, procedural guidance, and optional resources such as scripts, references, or templates [3], turning general-purpose models into task-specialized agents. Unlike one-off prompts, skills are designed for reuse (*locality*), often encode concrete workflows with verifiable steps (*determinism*), and share recurring needs for tools and runtime support (*environment requirements*); these traits open opportunities for system-level optimization but also impose demands that prompt-only pipelines do not address.

Most deployments, however, still treat skills as *prompt-time text* [3]: the system selects one or more skills, concatenates their bodies into the context window, and relies on the model to re-synthesize an execution procedure while calling tools. This prompt-centric design leads to four recurring problems. *Low reuse under small variations*—even for the same procedure, LLMs often produce plans that are semantically equivalent but differ in minor operational details (e.g., step order, tool arguments), defeating exact matching and preventing reuse of previously validated traces [45]. *Token waste in prompt-specialized blocks*—many skills contain code, scripts, or templates that must be specialized to the current prompt and environment; regenerating these on every run wastes tokens, and once instantiated they are natural candidates for caching. *Fragility from external dependencies*—skills depend on external tools and services (shell, OS interfaces, databases, MCP endpoints) [39]; when dependencies fail, executions often degrade into repeated, token-expensive trial-and-error [22]. *Missing parallel safety*—skills increasingly run in multiple sub-agents or parallel branches; without explicit concurrency

 doi:10.20944/preprints202602.1096.v1

control, concurrent access to the same resource is common, and prompt-level instructions (e.g., "avoid interdependent procedures") are inadequate for strict safety.

This paper takes an empirical approach to understanding these challenges. We analyze nearly 100,000 skills from public repositories [2,4] to characterize recurring patterns in how skills are structured, executed, and reused. Our analysis reveals six key properties: skills are predominantly phased procedures with natural step boundaries; many contain semi-deterministic blocks suitable for caching; LLM-generated procedures exhibit semantic equivalence despite surface-level differences; skills impose safety requirements (e.g., parallel-safety, idempotency) without enforcement mechanisms; execution depends on external tools with high failure rates and substantial token overhead when dependencies are missing; and skills are shared across sessions without global visibility for optimization or management. These properties are largely underexploited by current prompt-centric systems.

The four problems share a common root cause: LLM-based execution is inherently *non-deterministic*. Given the same skill and prompt, models may produce different plans, vary tool-call order, or intermittently ignore safety constraints. This non-determinism undermines both correctness and efficiency: without predictable behavior, the system cannot guarantee security properties or exploit optimization opportunities such as caching.

Traditional operating systems solve an analogous problem: they introduce deterministic abstractions—processes, virtual memory, file permissions—on top of complex, timing-dependent hardware [13,25,40,41]. We argue that a *Skill OS* must play the same role for LLM agents: introducing *deterministic, enforceable boundaries* at the interface between models and tools. Specifically, it must address *locality-aware caching*, *dynamically constructed environments*, *global skill management*, *structured failure handling*, and *runtime security and auditing*. We elaborate on these demands in §4 and position SkillOS relative to format standards and orchestration efforts in §5.

Contributions. We make two contributions: (1) an empirical characterization of skill properties from nearly 100,000 public skills, revealing six recurring patterns underexploited by current systems (§3); and (2) a set of system demands that motivate treating skills as OS-managed execution units (§4).

2. Background

2.1. From Prompts to Skills

The extension model for LLM agents has evolved through distinct phases [7,30]. Early applications relied on prompt engineering; as complexity grew, system prompts provided persistent context [38]. Tool use (function calling) enabled external interactions but required custom integration [35,43,52]. The current phase centers on *skills*, i.e., modular, reusable packages that bundle procedural knowledge, tool conventions, and resources [3]. Unlike ad-hoc prompts, skills are versioned, shareable, and discoverable artifacts.

2.2. Skill Structure

A skill is typically defined in a `SKILL.md` file and comprises three components [1,3]: (1) *metadata*, consisting of a concise name and description that serve as high-level indicators of the skill's scope; (2) *instructions*, the substantive body encoding procedural guidance, operational conventions, and pointers to bundled resources; and (3) *bundled resources*, including scripts, reference documents, and assets stored in separate files.

2.3. Skill Execution Model

Figure 1 shows the standard execution flow: agents *discover* skills at startup by scanning designated directories, *select* relevant skills based on user requests, *load* instructions into context, and *execute* by interpreting steps and invoking tools [3]. To reduce token consumption, agents employ *progressive unfolding*: metadata is loaded at startup, full instructions on selection, and resources only when referenced [4]. This prompt-centric model treats skills as passive text interpreted anew each run, which is simple but inefficient [33].

 doi:10.20944/preprints202602.1096.v1

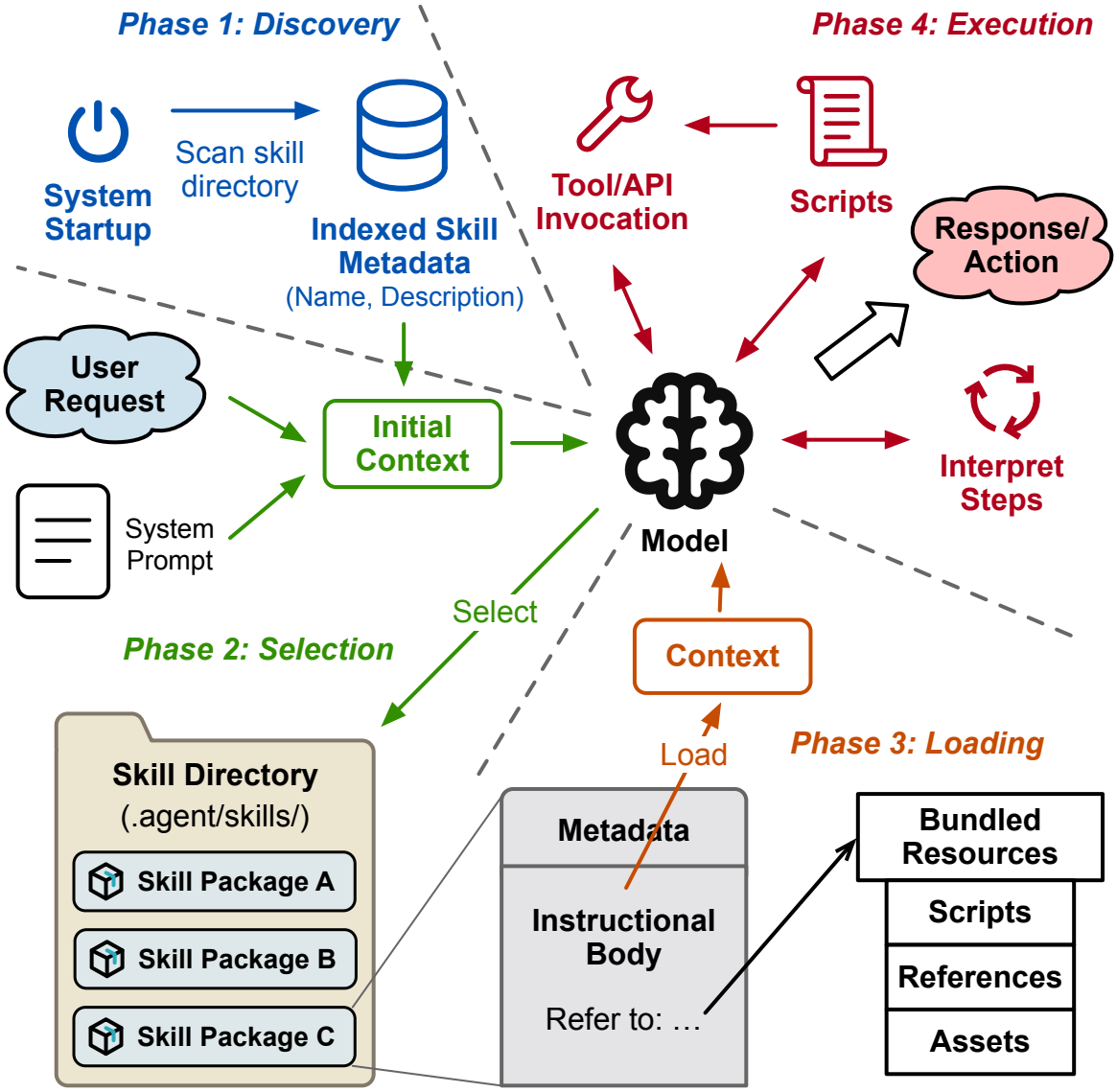

Figure 1. Execution flow: discover, select, load, execute.

2.4. The Skill Ecosystem

Skills have become a de facto standard across major coding agents including Claude Code, Cursor, OpenAI Codex, and Gemini [3,5,12,37]. The open `agentskills` specification enables portability: a skill directory with `SKILL.md` (YAML frontmatter + Markdown instructions) works across these platforms [1,2]. Community repositories host thousands of reusable skills for diverse domains [4].

3. Skills in the Wild

Based on an analysis of nearly 100,000 skills aggregated from public repositories, we identify recurring patterns and commonalities. These properties are not fully exploited by current systems, indicating missed opportunities for optimization.

3.1. Skill as a Procedure

Most skills explicitly specify multiple steps that decompose a complex, domain-specific problem into a sequence of simpler, model-comprehensible tasks [10,16,48,51]. Within such procedures, execution is not restricted to a single linear control flow. It may also involve conditional branching and iterative loops. Figure 2 shows how workflow structure is encoded in a skill.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 13 February 2026 doi:10.20944/preprints202602.1096.v1

Executing Plans

Step 1: Load and Review Plan

1. Read plan file
2. Review critically - identify any questions or concerns
3. If concerns: Raise them with your human partner
4. If no concerns: Create TodoWrite and proceed

Step 2: Execute Batch

...

Figure 2. A procedural skill.

To determine whether a skill can be classified as a procedure, we adopt a hybrid approach that combines heuristic analysis with model-based judgment. We first identify indicative markers and assign different weights to them based on their reliability. For example, markers such as 'Phase N' or 'Step N' are strongly associated with procedural structure and receive high weights. In contrast, weaker indicators such as ordinal expressions ('first', 'second', etc.) are more prone to false positives and are assigned lower weights.

Building on this heuristic filtering, we further employ a local language model to refine the classification. An description of procedural patterns is embedded in the model's system prompt, enabling it to further filter and validate skills identified in the previous stage. As illustrated in Figure 3, more than 50% of skills can be reasonably regarded as procedures.

Insight 1: A majority of skills can be identified as procedures.

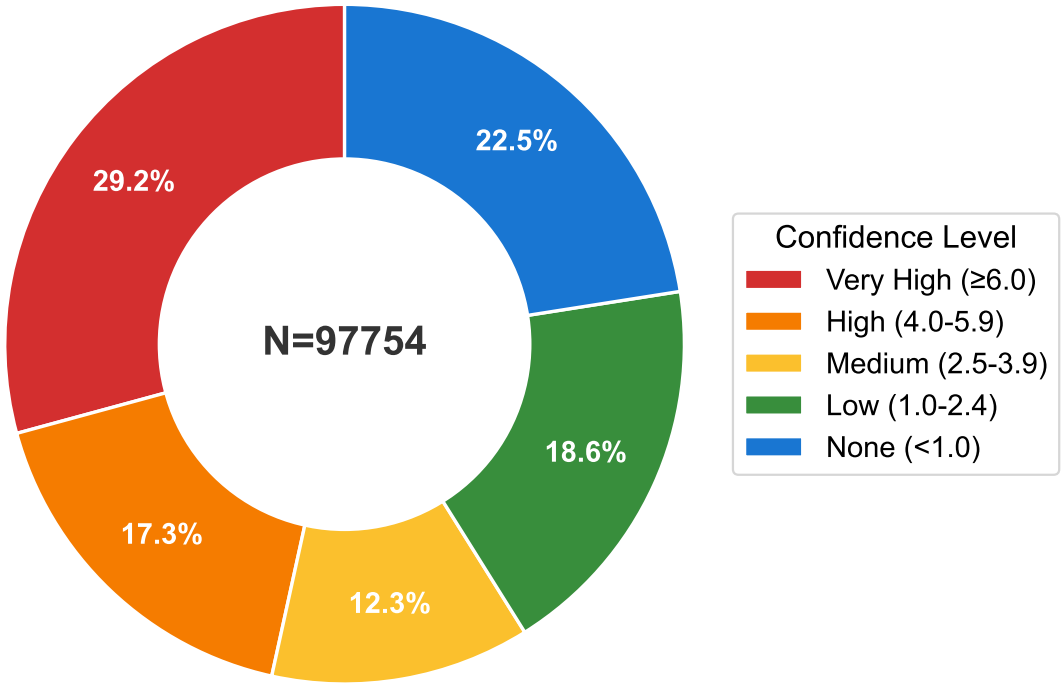

Figure 3. Skill procedure confidence distribution based on weighted pattern matching. A majority of skills exhibit explicit workflow structure with clear procedures and step boundaries (more than half fall into high or very high confidence).

 doi:10.20944/preprints202602.1096.v1

3.2. Semi-Deterministic Blocks in Skills

In the skill design paradigm, the main body of a skill is intended to include only the essential steps, while executable scripts should be stored as bundled resources so that they can be invoked directly without being loaded into the model's context. In practice, however, only fully deterministic scripts can be cleanly separated in this way. Many skills still embed scripts, code fragments, or template snippets that must be adapted to the specific prompt at runtime. We refer to these as semi-deterministic blocks. As shown in Figure 4, the name of files are determined by the user's input.

PYPDF - BASIC OPERATIONS

MERGE PDFS

```
from pypdf import PdfWriter, PdfReader

writer = PdfWriter()
for pdf_file in ["doc1.pdf", "doc2.pdf", "doc3.pdf"]:
    reader = PdfReader(pdf_file)
    …

with open("merged.pdf", "wb") as output:
    writer.write(output)
```

SPLIT PDF

...

Figure 4. A skill with semi-deterministic block.

Figure 5 shows that around 70% of skills contain such semi-deterministic blocks. Although these blocks are partially fixed, they are nevertheless loaded in full into the context on every execution.

Insight 2: Many skills contain semi-deterministic blocks.

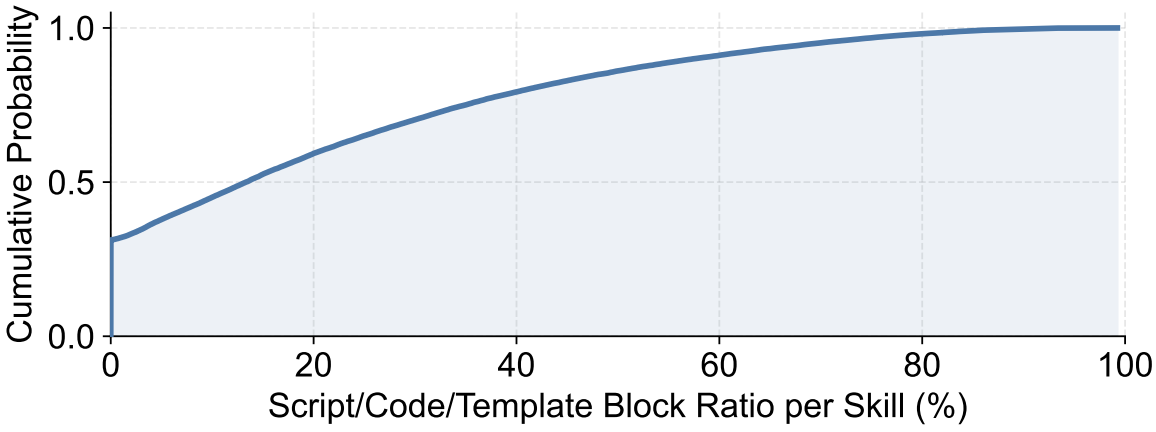

Figure 5. Cumulative distribution of the fraction of tokens occupied by semi-deterministic blocks (scripts, code, and templates) in each skill.

3.3. Execution Drift under Semantic Equivalence

In practice, the same skill may be loaded and invoked multiple times for the same task. For example, an initial attempt to execute a procedure may fail due to missing file write permissions. After enabling the required permissions, the identical task is re-issued. If the model were to generate an

 doi:10.20944/preprints202602.1096.v1

instruction sequence that is identical in both order and content for the two runs, the system could simply replay the previously generated instructions.

However, even when processing identical prompts, large language models may produce outputs that are semantically equivalent yet differ subtly in their operational steps (e.g., ordering, parameterization, or intermediate actions). These minor deviations hinder direct reuse via exact matching, as the system cannot reliably treat the newly generated procedure as a strict replay of the prior execution trace [45,47].

Insight 3: Two procedures with semantic equivalence may differ in operational details.

3.4. Requirements without Strong Guarantees

During execution, skills may be subject to certain requirements. As illustrated in Figure 6, the dispatching-parallel-agents skill requires that agents' tasks be free of mutual interference. Otherwise, concurrency bugs may arise. In the current system, this requirement is enforced solely through the model's internal reasoning. Specifically, the model assesses potential conflicts based on high-level task descriptions rather than concrete operational details. However, in practice, tasks that appear unrelated at the semantic level may still interfere at the execution level [6,9,18,21].

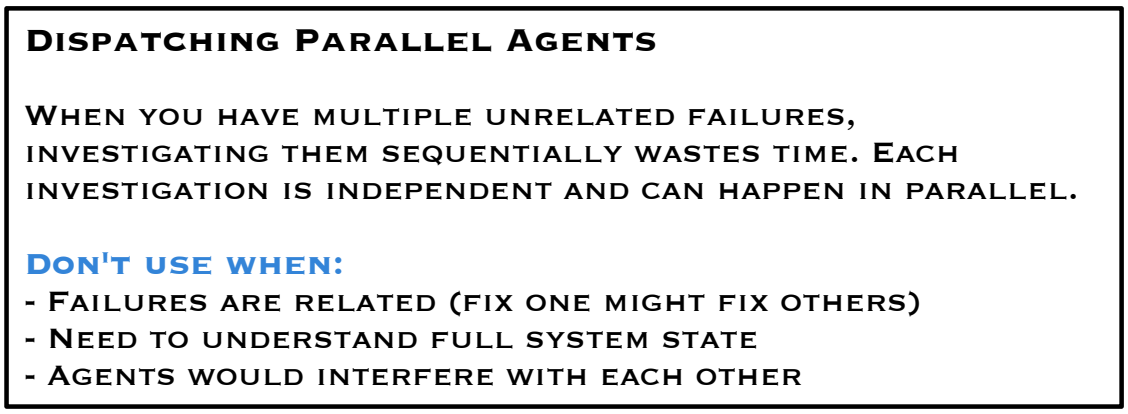

Figure 6. A skill with parallel-safety requirement.

Beyond this example, skills may also exhibit other crucial requirements, such as state consistency requirements, which assume that certain system states exist prior to execution, and idempotency requirements, which require skills to be safely re-runnable without unintended side effects.

Insight 4: Skills often impose critical requirements, yet lack strong enforcement guarantees.

3.5. Skill Dependence on Execution Environment

Some skills depend on external tools and system-specific APIs during execution [39]. As illustrated in Figure 7, about 45% of skills rely on shell utilities such as `grep`, while 31% invoke operating system APIs, including `read` and `exec`. In addition, a subset of skills utilizes database commands and network-related APIs, further reflecting the diversity of external dependencies in skill execution. Although skills may specify the dependencies that must be satisfied prior to execution, they rarely provide detailed configuration guidelines for each dependency across different system environments. As a result, missing dependencies and incompatible environments frequently lead to execution failures [24,29].

 doi:10.20944/preprints202602.1096.v1

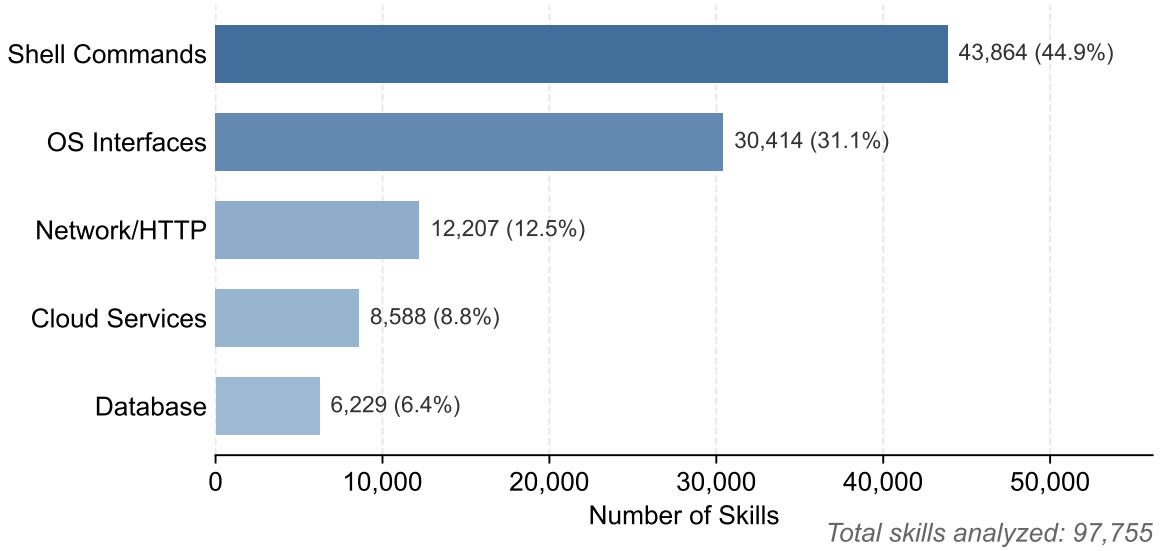

Figure 7. Distribution of external API dependencies. Each bar shows the number (and percentage) of skills referencing at least one dependency in the given category.

In current agent systems, dependency-related issues are primarily handled through the model's reasoning process, which attempts to diagnose and resolve missing dependencies at runtime [50]. However, this approach incurs additional token consumption. Based on our analysis of multiple skills, we find that the absence of a single required dependency during execution leads to an average overhead of tens of thousands of extra tokens, as shown in Figure 8.

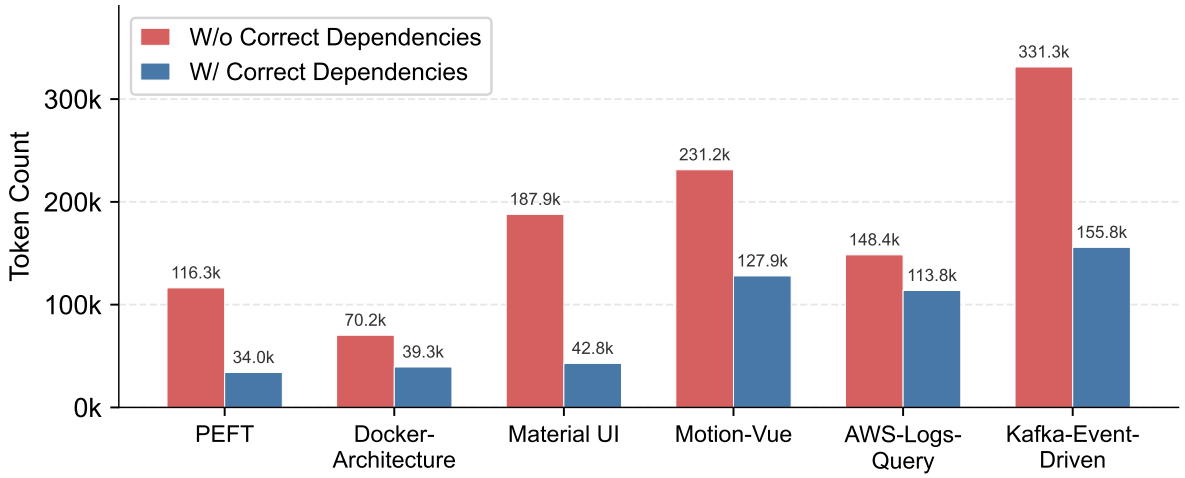

Figure 8. Token consumption with and without correct dependency across skills.

Insight 5: Skills may depend on execution environments, and unresolved dependencies lead to failures and substantial token overhead.

3.6. Shared Skills Between Sessions

Some skills are not consumed in a single, one-off execution. Instead, they are repeatedly invoked across multiple tasks and over extended periods [46]. For example, a software testing skill may be reused during the development of different repositories, or a file organization skill may be applied by a user at different times. However, in current agent systems, the model can only perceive and reason about skills within the scope of current task context, and lacks visibility into their usage and interactions at the system level. This absence of a global view leads to missed opportunities for optimization and undermines effective skill management.

 doi:10.20944/preprints202602.1096.v1

Insight 6: Some skills are shared across sessions, but the model lacks ability to manage them globally.

4. Demands for Skill OS

This section consolidates the core requirements for a Skill OS. Existing community efforts such as the Agent Skills project [1,2] establish a *portable skill format* and discovery mechanism (e.g., SKILL.md, YAML frontmatter, and cross-agent compatibility). Loaders such as OpenSkills [34] enable the *same* skills to be installed and invoked across Claude Code, Cursor, Windsurf, Aider, Codex, and other agents that read `AGENTS.md`, preserving progressive disclosure without tying skills to a single platform. Recent systems such as AgentSkillOS [27,28] target *skill retrieval and orchestration* at scale: organizing large skill pools (e.g., 90,000+ skills) into a capability hierarchy, retrieving task-relevant skills, and composing them into DAG-based workflows.

By contrast, a Skill OS addresses *execution*, *caching*, *concurrency*, and *safety* at the system layer—complementing both format standards and retrieval-orchestration layers with runtime support that current prompt-centric agents do not provide. Analogous to how a traditional operating system manages processes and allocates system resources, a Skill OS must similarly provide a dedicated execution environment for skills, manage the tools and resources required by those skills, optimize their execution efficiency, and ensure the safety and security of skill operations. The following subsections elaborate in detail on these essential capabilities of a Skill OS.

4.1. Leveraging Skill Locality

When the same skill is invoked repeatedly, whether within a single session or across sessions, its executions exhibit strong locality: a large fraction of the content processed by the model remains unchanged between invocations. As noted in *Insight 2*, the semi-deterministic blocks embedded in skills (code fragments, templates, and scripts) are loaded into the context in their entirety on every execution, even though most of their tokens are invariant across runs. Meanwhile, *Insight 3* shows that even semantically equivalent invocations may differ in surface-level operational details, precluding naïve exact-match reuse.

A Skill OS should explicitly exploit this locality. Concretely, the system should identify and cache the semantically stable portions of a skill's execution trace, including semi-deterministic blocks, resolved tool-call sequences, and intermediate artifacts, so that subsequent invocations of the same (or semantically equivalent) skill can reuse these cached fragments rather than regenerating them from scratch. The matching mechanism must operate at the semantic level rather than relying on syntactic identity, thereby tolerating the minor operational drift. Analogous to caching in conventional operating systems, this design avoids redundant model reasoning over content that has already been processed, simultaneously reducing token consumption and improving execution latency for recurring skill invocations.

4.2. Dynamic Environment Construction

As revealed by *Insight 5*, skills depend on execution environments, and unresolved dependencies lead to execution failures as well as substantial token overhead consumed by the model's attempts to diagnose and recover at runtime. A Skill OS must therefore dynamically construct an appropriate execution environment before each skill invocation, rather than leaving dependency resolution to the model's ad-hoc reasoning.

Constructing such an environment requires bridging two layers of information. On one hand, the system must inspect the skill's declared and implicit dependencies. On the other hand, it must account for the underlying system's actual configuration—which tools are installed, what versions are available, and what access policies are in effect. By reconciling these two perspectives, the Skill OS can assemble a per-invocation environment that binds each dependency to a concrete, validated resource, catching incompatibilities and missing components before execution begins rather than surfacing them as mid-run failures.

 doi:10.20944/preprints202602.1096.v1

4.3. Global Management Across Sessions and Agents

Current agent systems can only perceive skills within the scope of single task context and lack visibility into how those skills are used elsewhere in the system. A Skill OS, by virtue of operating at a higher privilege level than any agent, is uniquely positioned to serve as a *global manager* of skill procedures. From this vantage point, the Skill OS maintains a system-wide view of all registered skills. This global perspective allows the OS to determine which resources can be safely shared across tasks, such as stateless utility skills or cached tool outputs, and which must remain isolated to prevent interference, such as session-specific state or task-related data. By mediating access in this way, the Skill OS enables efficient reuse of common skills while enforcing the necessary boundaries between concurrent agents and sessions, thereby reducing redundant computation and preventing conflicts across multiple agents.

4.4. System-Level Fault Management

In a conventional OS, the kernel intercepts hardware exceptions such as page faults and illegal instructions, classifies them, and dispatches structured recovery actions on behalf of the faulting process. Individual applications need not interpret raw hardware signals themselves. In current agent systems, by contrast, when a tool call times out or an API returns an unexpected schema, the failure surfaces as an unstructured error string that the model must diagnose and recover from. A Skill OS should elevate this responsibility to the system level by intercepting and classifying common execution-time faults for each skill and applying appropriate recovery policies.

Furthermore, just as an OS is able to suspend and resume long-running processes, a Skill OS can leverage the procedural structure of skills to maintain logical checkpoints at step boundaries, allowing execution to roll back to the most recent safe point after a mid-run fault [14]. This avoids re-executing completed steps, limits repeated side effects, and reduces the token overhead of full restarts.

4.5. Security, Access Control, and Auditing

In current agent systems, security-critical requirements are expressed only in natural language within the skill text and enforced solely through the model's reasoning. As *Insight 4* shows, such prompt-level enforcement is inherently brittle. Tasks that appear safe at the semantic level may still violate security invariants at the execution level.

A Skill OS must therefore elevate security enforcement to the runtime layer. Concretely, the system should maintain a policy engine that specifies fine-grained access-control rules over tools, data sources, and external systems on a per-skill, per-agent, and per-session basis [15,42]. Before each tool invocation, the Skill OS should verify the calling skill's permissions against the active policy, rejecting unauthorized operations before they reach the underlying tool rather than relying on the model to self-censor. This design supports least-privilege execution: a skill that only needs read access to a repository should be unable to issue write operations, regardless of what its prompt text requests.

Beyond access control, the system should record auditable traces of all sensitive operations and tool invocations, enabling post-hoc analysis, compliance verification, and accountability for how skills interact with critical resources.

5. Discussion

A global OS view over skills. Treating skills as system-managed execution artifacts gives SkillOS a *global* view of what different skills (and different models) are doing. The system can track tool-call patterns, resource access, and execution histories, rather than just raw prompt text.

This global view enables cross-skill optimization. For example, different models may be used to accomplish the same task and often perform overlapping steps: reading the same files, running the same tests, or issuing the same database queries [31]. A system-layer scheduler can deduplicate these operations, share cached artifacts, and enforce a unified set of concurrency and safety policies [20,44,49].

 doi:10.20944/preprints202602.1096.v1

Cross-skill reuse beyond exact matching. Because LLM-generated procedures may differ slightly even under identical prompts, exact matching is insufficient for reuse. A Skill OS can identify reuse opportunities at the level of semantic equivalence and typed tool calls, enabling cross-skill and cross-model reuse even when the surface text differs [26].

One Skill OS, many models. As model ecosystems diversify, deployments will increasingly mix models with different cost/latency/quality tradeoffs. SkillOS decouples system responsibilities (caching, concurrency control, and exception handling) from any one model, allowing different models to share the same cached artifacts and safety guarantees [8,36].

The shifting OS–application boundary. The boundary between operating systems and applications has never been static. Mobile platforms like Android exemplify the trend toward thicker systems and thinner apps: applications delegate storage, networking, authentication, and UI rendering to platform services. Serverless computing pushes this further, reducing applications to stateless functions while the platform handles scaling, scheduling, and fault tolerance. SkillOS continues this trajectory for LLM agents: as skills become system-managed execution artifacts, the "application" layer becomes a thin specification of intent, while the OS handles execution, caching, concurrency, and recovery (Figure 9). In contrast to format-level efforts such as the Agent Skills specification [2], which standardize *what* a skill is (structure, metadata, portability), SkillOS focuses on *how* skills are executed, cached, and secured at the system level—the two layers are complementary. Systems like AgentSkillOS [27,28] take a step toward skill-level management by enabling *retrieval* and *orchestration* over large skill pools (e.g., skill trees and DAG-based workflows); SkillOS further emphasizes execution-layer concerns such as caching, concurrency control, failure handling, and security that remain underexplored in current retrieval-orchestration frameworks.

The shifting system–model boundary. A second boundary is emerging between what systems manage and what models generate. Many real-world tasks are recurring, well-defined, and reusable, such as generating a weekly report, validating a pull request, or deploying a service. These tasks should not be regenerated from scratch on every invocation; instead, they should crystallize into system-managed artifacts that improve over time. Unlike static function calls, SkillOS leverages models to evolve these artifacts: refining procedures based on feedback, adapting to new tool versions, and learning from execution traces. The result is a system that becomes faster and better with use, not by retraining models, but by accumulating validated, reusable execution knowledge [19,23,32,46].

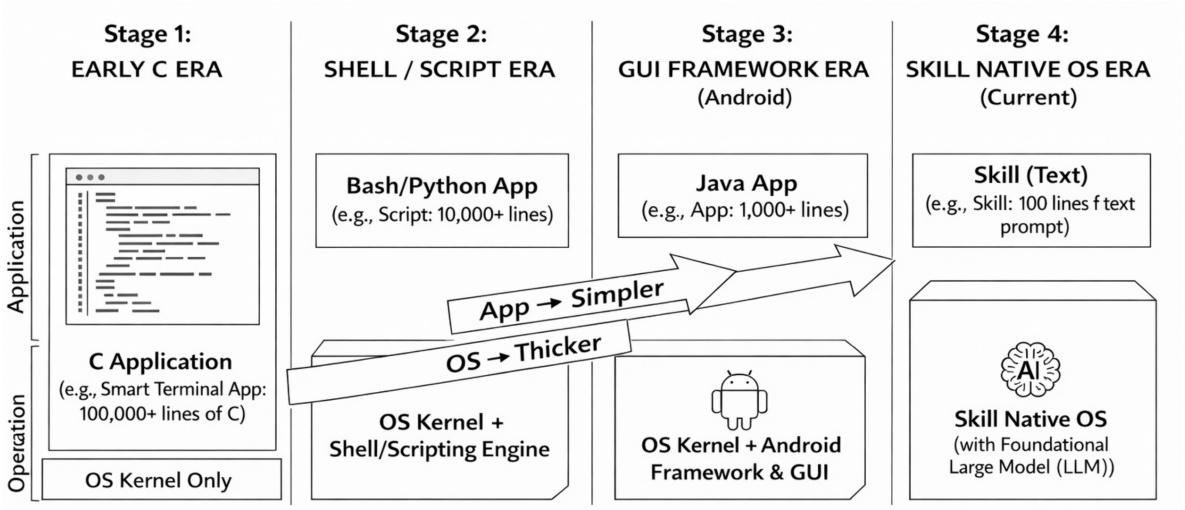

Figure 9. Evolution of operating system technology: OS gets thicker, apps get simpler. SkillOS extends this trajectory to LLM agents.

6. Conclusion

Skill-centric LLM systems are becoming a common way to deliver reliable tool use and domain expertise, but treating skills as prompt-only documentation leads to predictable inefficiencies and

 doi:10.20944/preprints202602.1096.v1

failures: repeated regeneration of prompt-specific blocks, brittle retries under external tool failures, lack of strict parallel safety, and poor reuse when semantically equivalent procedures differ in minor operational details.

Based on an empirical study of nearly 100,000 skills, we identify six key properties of skills in the wild: procedural structure, semi-deterministic blocks, execution drift under semantic equivalence, unenforced safety requirements, environment dependencies, and cross-session sharing. These observations motivate SkillOS, a system perspective that treats skills as first-class execution artifacts requiring OS-level support for locality-aware caching, dynamic environment construction, global management, first-class failure handling, and security enforcement.

The key message is that many properties needed for system support—phased procedures, cacheable blocks, tool dependencies, and resource footprints—already exist implicitly in skills. A Skill OS that makes these properties explicit is a path toward efficient, reliable, and scalable skill ecosystems.

References

1. Agent Skills Initiative. Agent skills specification. https://agentskills.io/specification, 2024. Open SKILL.md format for agent skill portability; accessed: 2026-02-06.
2. agentskills. Agent skills: Specification and documentation. https://github.com/agentskills/agentskills, 2024. Open format for giving agents new capabilities; specification, documentation, and reference SDK; accessed: 2026-02-06.
3. Anthropic. Agent skills. https://platform.claude.com/docs/en/agents-and-tools/agent-skills/overview, 2025. Accessed: 2026-02-02.
4. Anthropic. Equipping agents for the real world with agent skills. https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills, 2025. Accessed: 2026-02-06.
5. Anthropic. Extend claude with skills. https://code.claude.com/docs/en/skills, 2025. Claude Code skill mechanism; accessed: 2026-02-06.
6. Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
7. Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.
8. Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems (NeurIPS)*, 33:1877–1901, 2020.
9. Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–350, 2006.
10. Wenhu Chen, Xueguang Ma, Xuezhi Wang, William Cohen, Wen-tau Yih, Daniel Fried, et al. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions of the Association for Computational Linguistics (TACL)*, 11:547–563, 2023.
11. Coze. Plugin development guide. https://www.coze.cn/open/docs/guides/plugin, 2025. Coze bot plugins and skills; accessed: 2026-02-06.
12. Cursor. Agent skills. https://cursor.com/docs/context/skills, 2025. Cursor IDE skill mechanism; accessed: 2026-02-06.
13. Peter J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, 1970.
14. Elmootazbellah N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.
15. David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed nist standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.
16. Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, et al. Pal: Program-aided language models. *International Conference on Machine Learning (ICML)*, 2023.
17. Google. Agent skills. https://antigravity.google/docs/skills, 2025. Google Antigravity coding agent skills; accessed: 2026-02-06.
18. Jim Gray. The transaction concept: Virtues and limitations. *Proceedings of the 7th International Conference on Very Large Data Bases (VLDB)*, pages 144–154, 1981.

 doi:10.20944/preprints202602.1096.v1

19. Zijian He, Reyna Abhyankar, Vikranth Srivatsa, and Yiying Zhang. Cognify: Supercharging gen-ai workflows with hierarchical autotuning. *arXiv preprint arXiv:2502.08056*, 2025.
20. Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Chaoyun Zhang, et al. Metagpt: Meta programming for a multi-agent collaborative framework. *International Conference on Learning Representations (ICLR)*, 2024.
21. Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference (ATC)*, pages 145–158, 2010.
22. Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *International Conference on Learning Representations (ICLR)*, 2024.
23. Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.
24. Pavneet Singh Kochhar, Ferdian Thung, and David Lo. An empirical study of build failures in open source projects. *Empirical Software Engineering*, 21(4):1467–1503, 2016.
25. Butler W. Lampson. Protection. In *Proceedings of the 5th Princeton Conference on Information Sciences and Systems*, pages 437–443, 1971.
26. Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems (NeurIPS)*, 33:9459–9474, 2020.
27. Hao Li et al. Agentskillos: Build your agent from 90,000+ skills via skill retrieval and orchestration. https://github.com/ynulihao/AgentSkillOS, 2026. Skill tree, retrieval, and DAG-based orchestration over large skill pools; accessed: 2026-02-06.
28. Hao Li, Chunjiang Mu, Jianhao Chen, Siyue Ren, Zhiyao Cui, Yiqun Zhang, Lei Bai, and Shuyue Hu. Leveraging, managing, and scaling the agent skill ecosystem. *Preprint*, 2026. AgentSkillOS: skill retrieval and orchestration at scale.
29. Zhenhao Li, Mingwen Zhang, Yingfei Xiong, et al. A large-scale study of api breaking changes in the wild. *Empirical Software Engineering*, 2023.
30. Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35, 2023. arXiv:2107.13586, 2021.
31. Xiao Liu, Hao Lu, Yuxiang Zhang, Xiao Liu, Zhuohan Qian, Yansong Zhu, et al. Agentbench: Evaluating llms as agents. *International Conference on Learning Representations (ICLR)*, 2024.
32. Zibin Liu, Cheng Zhang, Xi Zhao, Yunfei Feng, Bingyu Bai, Dahu Feng, Erhu Feng, Yubin Xia, and Haibo Chen. Beyond training: Enabling self-evolution of agents with MOBIMEM. *arXiv preprint arXiv:2512.15784*, 2025.
33. Grégoire Mialon, Roberto Dessì, Maria Lomeli, Carsten Eickhoff, Thomas Scialom, Zihan Wang, et al. Augmented language models: A survey. *Transactions on Machine Learning Research (TMLR)*, 2023.
34. numman-ali. Openskills: Universal skills loader for ai coding agents. https://github.com/numman-ali/openskills, 2026. CLI to install and load SKILL.md across Claude Code, Cursor, Windsurf, Aider, Codex; same format as Claude Code; accessed: 2026-02-06.
35. OpenAI. Function calling and other api updates. https://openai.com/blog/function-calling-and-other-api-updates, 2023. Accessed: 2026-02-06.
36. OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.
37. OpenAI. Openai codex cli documentation. https://developers.openai.com/codex, 2025. Accessed: 2026-02-06.
38. Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, pages 27730–27744, 2022.
39. Yujia Qin, Yining Ye, Lei Fang, Haoming Zhang, Wenhui Wang, Bin Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*, 2023.
40. Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Communications of the ACM*, 17(7):365–375, 1974.

 doi:10.20944/preprints202602.1096.v1

41. Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.
42. Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.
43. Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Jonas Hamburger, et al. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems (NeurIPS)*, 36, 2023.
44. Yongliang Shen, Kaitao Song, Xu Tan, Dong Zhang, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems (NeurIPS)*, 36, 2023.
45. Noah Shinn, Federico Cassano, Emmanuel Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems (NeurIPS)*, 37, 2024.
46. Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Angel Fan, Anima Anandkumar, et al. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research (TMLR)*, 2024.
47. Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, et al. Self-consistency improves chain of thought reasoning in language models. *International Conference on Learning Representations (ICLR)*, 2023.
48. Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:24824–24837, 2022.
49. Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.
50. John Yang, Carlos E. Jimenez, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.
51. Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems (NeurIPS)*, 36, 2023.
52. Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *International Conference on Learning Representations (ICLR)*, 2023.