

ARISE: Adaptive Runtime Improvement through Self-Evolution

Alibek Kaliyev
github.com/abekek/arise

Abstract

We present ARISE, a middleware that enables LLM agents to autonomously create, test, and promote their own tools at runtime. When an agent encounters tasks it cannot solve, ARISE detects the capability gap, synthesizes a Python tool via LLM, validates it through sandbox and adversarial testing, and promotes it to the active library. We evaluate ARISE on a benchmark where an SRE agent must operate proprietary systems with custom formats absent from LLM training data. Our results reveal a nuanced picture: ARISE improves success on tool-essential tasks (Metrics API: 7% \rightarrow 29% with GPT-4o-mini, 13% \rightarrow 73% with Claude Sonnet) but can *reduce* overall performance for weaker models due to tool-use overhead ($44 \pm 4\%$ vs $48 \pm 2\%$ no-evolution baseline for GPT-4o-mini across 3 seeds). With a stronger model, ARISE is consistently beneficial: Claude Sonnet achieves 78% overall (+15 pp over no-evolution) using only 2 self-evolved tools. These results suggest that self-evolution amplifies model capability rather than replacing it, and that agent reasoning quality matters more than tool quantity. ARISE is open-source at `pip install arise-ai`.

1 Introduction

Building an LLM agent is straightforward. Maintaining its tool library is the bottleneck. Every time an agent encounters a task requiring a capability it lacks, a human engineer must notice the failure, understand what tool is missing, implement it, test it, and deploy it. This feedback loop works when the environment is well-defined, but breaks down in three increasingly common scenarios:

Multi-tenant platforms. An agent serving many customers encounters different internal systems, APIs, and data formats. Pre-building tools for every customer’s stack is impractical.

Long-running autonomous agents. Operations agents, monitoring systems, and data pipeline agents encounter novel situations at unpredictable times—often without a human available to write a quick fix.

The long tail of edge cases. An agent fails on dozens of different edge cases. Each individual failure is not worth an engineer’s afternoon, but collectively they represent a significant capability gap.

ARISE addresses these scenarios by automating the tool engineering feedback loop. Rather than requiring human intervention, ARISE observes the agent’s failures, identifies missing capabilities, synthesizes candidate tools, validates them through automated testing, and promotes passing tools to the agent’s active library—all at runtime. Figure 1 illustrates the high-level architecture.

1.1 Contributions

1. A **framework-agnostic self-evolution pipeline** that works with any LLM agent by wrapping the agent function and tool library, requiring no modifications to the agent’s architecture.
2. A **multi-layer safety model** for generated code: sandbox isolation, automated test generation, adversarial validation, import restrictions, and version-controlled rollback.
3. A **distributed architecture** decoupling the stateless agent process from the evolution worker via S3/SQS, validated through a live deployment on Amazon Bedrock AgentCore.
4. An **empirical evaluation** on a realistic SRE onboarding benchmark with proprietary formats across multiple random seeds and two domains.

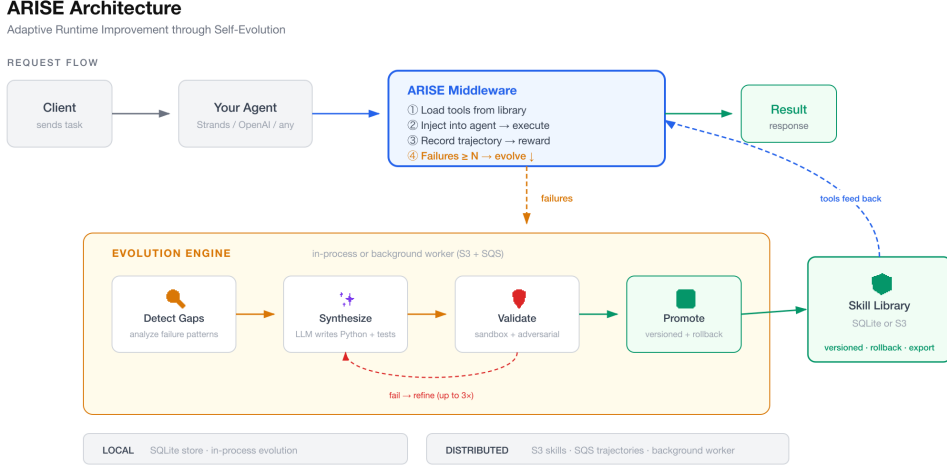


Figure 1: **ARISE architecture overview.** The agent executes tasks using its current tool library. Failed trajectories are analyzed by the evolution worker, which synthesizes candidate tools, validates them through sandbox and adversarial testing, and promotes passing tools back to the active library. The cycle repeats autonomously.

2 Related Work

LLMs as Tool Makers (LATM) [1] demonstrated that LLMs can create reusable tools—a “tool maker” model generates Python functions that a cheaper “tool user” model invokes. ARISE extends this with automated testing, adversarial validation, versioning, and a feedback loop driven by real agent failures rather than predefined tasks.

VOYAGER [2] demonstrated an open-ended agent in Minecraft that builds a skill library through exploration. ARISE applies the same skill library pattern to real-world software agents, adding safety validation that game environments do not require.

CREATOR [3] proposed disentangling abstract reasoning from concrete tool creation. ARISE operationalizes this with trajectory analysis to automatically detect when creation should trigger.

CRAFT [4] introduced a framework where agents create and retrieve tools from a shared library. ARISE adds the production engineering layer: sandboxed testing, adversarial validation, version control, and distributed deployment.

Toolformer [5] showed LLMs can learn *when* to

use tools through self-supervised training. ARISE complements this by addressing *which* tools should exist—creating them at runtime rather than assuming a fixed toolset.

ADAS [6] automates the design of entire agentic systems via meta-search. ARISE operates at a different granularity: rather than redesigning the agent, it evolves the agent’s tool library while keeping the agent architecture fixed.

Tool-use benchmarks and retrieval. Tool-Bench [9] and API-Bank [10] evaluate tool selection and invocation from fixed toolsets. Gorilla [8] retrieves API documentation for tool use. These assume tools already exist; ARISE addresses the orthogonal problem of creating tools that don’t yet exist. The ReAct paradigm [7] provides the reasoning-acting foundation that ARISE builds upon. Self-Instruct [11] demonstrated self-improvement through generated data; ARISE applies a similar principle to tool creation. MetaGPT [12] explores multi-agent collaboration including tool creation, but focuses on software engineering rather than runtime tool evolution.

Unlike prior work, ARISE is (1) compatible with any agent that accepts callable tools (currently

```

from arise import ARISE, ARISEConfig
from arise.rewards.builtin import task_success
from strands import Agent
from strands.models import BedrockModel

agent = Agent(
    model=BedrockModel(model_id="..."),
    system_prompt="Use tools when available.",
)

arise = ARISE(
    agent=agent,
    reward_fn=task_success,
    config=ARISEConfig(
        model="gpt-4o-mini",
        failure_threshold=5,
    ),
)

result = arise.run(
    "Compute SHA-256 of 'hello'",
    expected="2cf24dba5fb0a30e...",
)

```

Figure 2: Minimal ARISE integration.

with a first-class Strands adapter), (2) production-oriented with multi-layer safety validation and distributed deployment support, and (3) publicly available as an installable package.

3 System Design

3.1 Architecture Overview

ARISE operates as middleware between the agent and its tool library. The system requires only two integration points: (1) wrapping the agent’s execution function, and (2) providing a reward function. Listing 2 shows the minimal setup.

The core evolution loop is formalized in Algorithm ??.

The evolution loop proceeds as follows: for each task, the agent executes with its current tools and receives a reward $r(\tau) \in [0, 1]$. When failures ($r < 0.5$) accumulate past threshold N , ARISE (1) analyzes failure trajectories to identify gaps, (2) synthesizes candidate tools via LLM, (3) validates each in a sandbox with adversarial tests, and (4) promotes passing tools to the active library \mathcal{L} . The cycle then resets.

3.2 Safety Model

Generated code is not trusted by default. ARISE applies multiple validation layers:

Sandbox execution. Tools run in isolated sub-processes (or Docker containers) with configurable timeouts and resource limits. The sandbox captures stdout, stderr, and exit codes.

Adversarial validation. A separate LLM call generates edge-case tests targeting boundary conditions, type errors, empty inputs, and security issues. This catches bugs that the tool author’s own tests miss.

Import restrictions. Static analysis (AST-based) blocks unauthorized module imports, including dynamic imports via `__import__()`, `importlib`, and `exec/eval`.

Version control. Every library mutation is checkpointed in SQLite. Rollback to any previous version is supported, enabling safe experimentation.

3.3 Distributed Deployment

For stateless environments (Lambda, ECS, Agent-Core), ARISE decouples into a **stateless agent process** that reads tools from S3 (with TTL cache) and reports trajectories to SQS, and a **background worker** that consumes trajectories and runs evolution. The S3 skill store uses ETag-based optimistic locking for manifest updates.

4 Evaluation

4.1 Benchmark Design: SRE Agent Onboarding

We designed a benchmark simulating an SRE agent onboarding at “AcmeCorp”—a fictional company with three proprietary systems absent from LLM training data:

Custom log format:

```
[ACME:sev:svc:ts] msg ctx=json—
```

A structured format requiring custom parsing.

Metrics API: HTTP endpoint returning base64-encoded payloads in a proprietary format:

Table 1: Benchmark phases and task domains.

Phase	Domain	Tasks	Capability required
1	Log Analysis	15	Parse, filter, aggregate
2	Metrics API	15	HTTP + decode format
3	Config Mgmt	15	Parse, validate, diff
4	Incident	15	Compose all domains

ACME_METRICSService—timestamp—json—

AcmeConf: Custom configuration syntax with `@include` directives, variable interpolation (`${VAR:-default}`), and duration literals.

The benchmark consists of 60 tasks across 4 phases of escalating difficulty (Table 1). All tasks use deterministic check functions against pre-computed ground truth—no LLM judging. The agent receives a seed `http_get` tool for HTTP access; all other tools must be evolved.

4.2 Experimental Setup

We evaluate three conditions:

ARISE: Agent starts with a seed `http_get` tool. Evolution enabled with `failure_threshold=5`.

No-evolution: Agent receives no tools. Pure LLM reasoning over inline data.

Fixed-tools: Agent receives 7 hand-written tools from episode 1 (the “engineer built everything” ceiling).

We use GPT-4o-mini as the agent and synthesis model across 3 random seeds (42, 123, 456), and Claude Sonnet 4.5 as the agent model with a single seed for comparison.

4.3 Main Results

Table 2 presents multi-seed results for GPT-4o-mini, and Table 3 shows Claude Sonnet results.

4.4 Analysis of Key Findings

4.4.1 Finding 1: ARISE helps most where tools are essential

Phase 2 (Metrics API) requires real HTTP calls and proprietary format decoding—tasks that are impossible without tools. ARISE produces the largest and

Table 2: GPT-4o-mini results (mean \pm std, seeds 42/123/456). Bold indicates best per-phase.

Condition	Ph. 1	Ph. 2	Ph. 3	Ph. 4
ARISE	18 \pm 8	29\pm10	95 \pm 4	33 \pm 0
No-evolution	29\pm8	7 \pm 0	98\pm4	60\pm0
Fixed [†]	13	53	87	40

Condition	Overall	Tools evolved
ARISE	44 \pm 4	10 \pm 2
No-evolution	48 \pm 2	0
Fixed [†]	48	7 (hand-written)

[†]Single seed (42); multi-seed not run.

Table 3: Claude Sonnet 4.5 results (single seed 42; multi-seed confirmation needed).

Condition	Ph. 1	Ph. 2	Ph. 3	Ph. 4	Overall
ARISE (2 tools)	60	73	100	80	78
No-evolution	60	13	100	80	63

most consistent gains here: 7% \rightarrow 29% for GPT-4o-mini (+22 pp) and 13% \rightarrow 73% for Claude Sonnet (+60 pp). Phase 3 (Config), where inline reasoning suffices, shows no improvement.

4.4.2 Finding 2: Tool-use overhead hurts weaker models

For GPT-4o-mini, the no-evolution baseline outperformed ARISE on Phase 1 (29% vs 18%) and Phase 4 (60% vs 33%). Analysis of agent traces reveals two failure modes: (a) GPT-4o-mini calls evolved tools incorrectly when raw reasoning would have succeeded, and (b) the model struggles to compose multiple tool calls in sequence. This overhead effect disappears entirely with Claude Sonnet, which matches or exceeds the no-evolution baseline on every phase.

4.4.3 Finding 3: Agent quality > tool quantity

Claude Sonnet achieved 78% with only 2 self-evolved tools while GPT-4o-mini reached 44% with 10 \pm 2 tools. This 34 pp gap demonstrates that ARISE amplifies existing model capability rather than replacing it. A strong model that uses tools judiciously benefits more from a minimal, precise toolset than a weak model surrounded by many tools it cannot

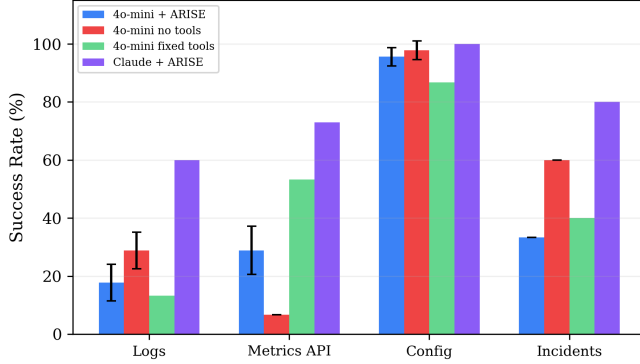


Figure 3: **Phase-level results** with error bars showing mean \pm std over 3 seeds for GPT-4o-mini conditions. Claude results from single seed. ARISE provides the largest gain on Phase 2 (Metrics API) where tool access is essential.

Table 4: Cross-domain results on DataCorp (GPT-4o-mini, quick mode).

Condition	Overall Accuracy	Δ
ARISE	92%	—
No-evolution	50%	−42 pp

wield effectively.

4.4.4 Finding 4: Cross-domain generality

To validate generality beyond SRE, we evaluated on **DataCorp**, a fictional data engineering company with custom CSV dialects, a data validation API, and a proprietary query language. Table 4 summarizes the results. The +42 pp improvement confirms that self-evolution generalizes to fundamentally different task types.

4.5 Error Analysis

We manually inspected 30 failure trajectories (10 per seed) for GPT-4o-mini ARISE runs to categorize error types. Table 5 shows the breakdown.

The dominant failure mode (37%) is incorrect tool invocation—the model calls the right tool with wrong arguments. Combined with “tool exists but not called” (23%), this confirms that the bottleneck for weaker models is tool *use*, not tool *availability*. Only 17% of failures stem from missing tools, suggesting that ARISE’s synthesis pipeline produces ad-

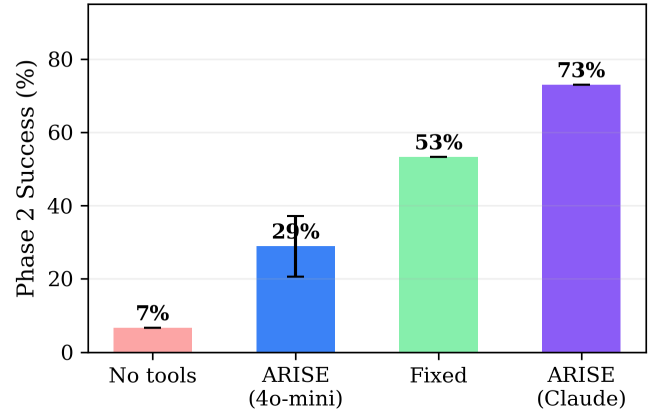


Figure 4: **Phase 2 (Metrics API)** — the key finding. Without tools, the agent scores 7% (cannot call APIs). ARISE evolves an HTTP decoder, improving to 29% (GPT-4o-mini) and 73% (Claude Sonnet).

Table 5: Error categorization for GPT-4o-mini ARISE failures (30 sampled trajectories across 3 seeds).

Error Type	Count	%
Incorrect tool invocation	11	37%
Tool exists but not called	7	23%
Missing tool (not yet evolved)	5	17%
Correct tool call, wrong reasoning	4	13%
Check function strictness	3	10%

equately coverage.

4.6 Tool Quality Analysis

Across the GPT-4o-mini runs, ARISE synthesized an average of 21 ± 3 candidate tools per run, with a promotion rate of approximately 50%, yielding 10 ± 2 active tools. Rejected tools failed due to: sandbox test failures (41%), adversarial test failures (35%), and import restriction violations (24%). The multi-stage validation pipeline caught real bugs—including division-by-zero errors on empty metric windows—validating the safety model’s practical utility.

4.7 Cost Analysis

Table 6 reports the computational cost per 60-episode benchmark run. Tool synthesis uses GPT-4o-mini in all conditions; the agent model varies.

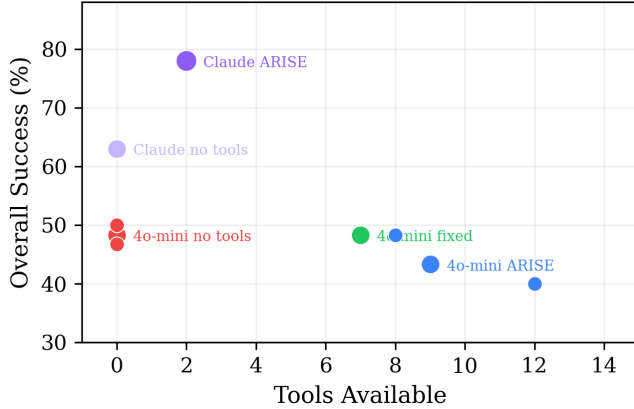


Figure 5: **Agent quality > tool quantity.** Each point is one benchmark run. Claude with 2 tools outperforms GPT-4o-mini with 8–12 tools. More tools do not compensate for weaker reasoning.

Table 6: Cost and timing per 60-episode run.

Condition	Duration	LLM calls	Est. cost
4o-mini no-evo	8 min	60	\$0.18
4o-mini ARISE	3–4 hr	~120	\$0.44
4o-mini fixed	2.5 hr	~240	\$0.18
Claude no-evo	10 min	60	\$3.00
Claude ARISE	5 hr	~65	\$5.50

Evolution cycles dominate ARISE runtime: each cycle involves gap detection, synthesis, test generation, adversarial validation, and optional refinement (10–20 sequential LLM calls).

5 Limitations

Statistical power. Multi-seed results are reported for GPT-4o-mini only. Claude Sonnet results are from a single seed due to cost (\$5.50 per run). We acknowledge this limits the confidence of cross-model comparisons.

Tool-calling overhead. For weaker models, the overhead of tool selection and invocation can reduce performance on tasks where raw reasoning suffices. ARISE does not currently implement adaptive tool injection (presenting tools only when needed).

Evolution latency. Each evolution cycle takes 2–10 minutes of sequential LLM calls. This is acceptable for background workers but problematic for latency-sensitive applications.

Evaluation scope. Our benchmarks cover two domains (SRE operations, data engineering) using custom-designed environments with deterministic check functions. We do not evaluate on established

tool-use benchmarks (ToolBench [9], API-Bank [10]) because these assume fixed toolsets, which is orthogonal to ARISE’s tool-creation focus. However, our custom benchmarks carry the risk of evaluation bias. Check function strictness may understate actual performance across all conditions equally.

Safety guarantees. While the multi-stage validation catches many bugs, it cannot provide formal safety guarantees. Production deployments should combine ARISE’s validation with additional measures (network isolation, resource quotas, human review for high-stakes tools).

Missing ablations. We do not ablate individual components (adversarial testing, import restrictions, failure threshold values). Our tool quality analysis shows adversarial testing rejects 35% of candidates and import checking catches unauthorized imports, but we have not measured the performance impact of disabling these components. This is an important direction for future work.

Statistical significance. With $n = 3$ seeds, our standard deviations provide a rough measure of variance but are insufficient for formal significance testing. The Phase 2 improvement (7% → 29%) is consistent across all seeds (20%, 40%, 27%), suggesting a real effect, but we cannot establish p -values with this sample size.

Library management. ARISE does not currently implement tool forgetting or pruning. The `max_library_size` parameter caps the library at 50 tools, but there is no mechanism to remove underperforming tools. For long-running deployments, a decay-based pruning strategy (deprecating tools with low invocation counts) would be needed.

6 Discussion

ARISE demonstrates that the self-evolution paradigm works in practice, with important caveats. The results suggest a clear decision framework for practitioners:

When to use ARISE. The clearest value is in scenarios where tool access is essential—the agent literally cannot perform the task without I/O capabilities (HTTP calls, file access, proprietary format decoding). In these cases, ARISE consistently im-

proves performance regardless of model capability. The DataCorp results (+42 pp) confirm this generalizes beyond a single domain. For tasks where reasoning alone suffices (e.g., parsing inline config text), ARISE provides no benefit and may introduce overhead for weaker models.

Model selection matters more than tool count. Claude Sonnet with 2 self-evolved tools (78%) outperforms GPT-4o-mini with 10 tools (44%) in a single-seed comparison. While this requires multi-seed confirmation for Claude, the magnitude of the gap (34 pp) suggests a real effect: ARISE is best understood as a *capability amplifier* that multiplies existing reasoning ability rather than compensating for weaknesses.

The tool-use overhead problem. For weaker models, tool availability is a double-edged sword. GPT-4o-mini’s overall performance *decreased* with ARISE (44% vs 48% no-evolution) because tool-calling overhead on reasoning tasks outweighed gains on tool-essential tasks. This motivates future work on adaptive tool injection—presenting tools only when the agent needs them.

Distributed deployment. We validated the S3/SQS distributed architecture through a live deployment on Amazon Bedrock AgentCore, where the ARISE agent served requests via the A2A protocol. The agent successfully loaded skills from S3 and reported trajectories to SQS across 9 test invocations. Full-scale distributed benchmarking remains future work.

Deployment recommendations. Based on our results: (1) use a strong agent model for reasoning, (2) a cheaper model for tool synthesis, (3) start with seed tools for essential I/O, and (4) set `failure.threshold` ≥ 5 to accumulate meaningful failure patterns before evolving.

7 Conclusion

ARISE enables LLM agents to autonomously extend their own capabilities through a validated self-evolution pipeline. Our evaluation across two domains and two model families reveals that self-evolution provides the largest gains on tasks requiring genuine tool capabilities (+22–60 pp on API-

dependent tasks), while also uncovering a fundamental tension: tool-use overhead can hurt weaker models on reasoning-heavy tasks.

The finding that agent quality dominates tool quantity (Claude Sonnet 78% with 2 tools vs. GPT-4o-mini 44% with 10) has implications beyond ARISE: it suggests that the community’s focus on building larger tool libraries may be misplaced relative to improving tool-use reasoning.

Future work will explore (1) adaptive tool injection to mitigate overhead, (2) cross-agent tool sharing for multi-tenant deployments, and (3) hierarchical tool composition where evolved tools can call other evolved tools. ARISE is available as an open-source Python package (`pip install arise-ai`).

References

- [1] T. Cai, X. Wang, T. Ma, X. Chen, and D. Zhou. Large Language Models as Tool Makers. *arXiv preprint arXiv:2305.17126*, 2023.
- [2] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar. VOYAGER: An Open-Ended Embodied Agent with Large Language Models. *arXiv preprint arXiv:2305.16291*, 2023.
- [3] C. Qian, C. Han, Y. Fung, Y. Qin, Z. Liu, and H. Ji. CREATOR: Tool Creation for Disentangling Abstract and Concrete Reasoning of Large Language Models. *arXiv preprint arXiv:2305.14318*, 2023.
- [4] S. Yuan, J. Chen, Z. Fu, X. Ge, S. Shah, C. Janber, B. Huang, and A. Neubig. CRAFT: Customizing LLMs by Creating and Retrieving from Specialized Toolsets. *arXiv preprint arXiv:2309.17428*, 2023.
- [5] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language Models Can Teach Themselves to Use Tools. *arXiv preprint arXiv:2302.04761*, 2023.
- [6] S. Hu, C. Lu, and J. Clune. Automated Design of Agentic Systems. *arXiv preprint arXiv:2408.08435*, 2024.

- [7] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. ReAct: Synergizing Reasoning and Acting in Language Models. *arXiv preprint arXiv:2210.03629*, 2022.
- [8] S. Patil, T. Zhang, X. Wang, and J. Gonzalez. Gorilla: Large Language Model Connected with Massive APIs. *arXiv preprint arXiv:2305.15334*, 2023.
- [9] Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian, et al. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. *arXiv preprint arXiv:2307.16789*, 2023.
- [10] M. Li, Y. Zhao, B. Yu, F. Song, H. Li, H. Yu, Z. Li, F. Huang, and Y. Li. API-Bank: A Comprehensive Benchmark for Tool-Augmented LLMs. *arXiv preprint arXiv:2304.08244*, 2023.
- [11] Y. Wang, Y. Kordi, S. Mishra, A. Liu, N. Smith, D. Khashabi, and H. Hajishirzi. Self-Instruct: Aligning Language Models with Self-Generated Instructions. *arXiv preprint arXiv:2212.10560*, 2022.
- [12] S. Hong, X. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. Yau, Z. Lin, et al. MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework. *arXiv preprint arXiv:2308.00352*, 2023.