ADK Arena: Surfacing Failure Modes in Agent Development Kits via LLM-as-a-Developer

Jintao Huang [1] **Xiaomin Li** [2] **Gaurav Mittal** [2] **Yu Hu** [2]

Abstract

The rapid proliferation of Agent Development Kits (ADKs), SDK-level frameworks for building LLM-powered autonomous agents, has outpaced any empirical understanding of *where the agents built on them break*. We propose **LLM-as-a-Developer**, a methodology that replaces the human developer with an LLM coding agent that learns each framework's API from documentation, writes agent code, and iteratively repairs it through a validate-and-feedback loop until tests pass. By holding the developer constant and varying only the framework, the pipeline turns each framework into a reproducible trigger: generation effort exposes API-usability failures, and the resulting agents expose task-execution failures under a controlled, trace-level protocol. We implement this in **ADK Arena**, a fully automated pipeline with per-framework Docker isolation, a three-level validation pipeline, and benchmark adapters for SWE-bench, τ^2-bench, Terminal-Bench, and MCP-Atlas. Evaluating all 51 popular Python ADK frameworks (204 agent–benchmark pairs), we surface three recurring failure modes: (1) *generation failure*: 43% of runs never yield a validated agent, and generation cost varies 5.6× across frameworks ($0.6 to $3.4 per agent), a quantitative proxy for API complexity, though cost alone does not predict success; (2) *execution collapse*: no single framework dominates, and while the best single-benchmark ADK agents resolve up to 80% of tasks and can even *beat* general-purpose frontier coding agents at a fraction of the cost, the median framework resolves only 32%, with the developer model that *writes* the agent driving twice the resolution gap of the model that *runs* it; (3) *silent thin-wrapping*: across information-source ablations, genuine framework usage stays within a narrow 28–40% band (highest with raw source access and still 33% with no reference material at all), so documentation, source code, and parametric knowledge are largely substitutable and curated docs can anchor the developer toward thin wrappers rather than real framework use[1].

1. Introduction

A developer building an LLM-powered agent in 2025 faces dozens of *Agent Development Kits* (ADKs), SDK-level frameworks that provide reusable abstractions for tool calling (Schick et al., 2024; Patil et al., 2023), agentic loops (Yao et al., 2023; Shinn et al., 2023), multi-agent orchestration (Wu et al., 2024; Li et al., 2023a; Chen et al., 2024), and other capabilities for autonomous agents (Xi et al., 2025; Wang et al., 2024a; Sumers et al., 2024). Fueled by advances in large language models (Brown et al., 2020; OpenAI, 2023; Anthropic, 2024; Touvron et al., 2023), every major AI vendor has shipped an official ADK within 18 months, alongside dozens of open-source alternatives. Our landscape survey identifies 51 popular ADK frameworks distributed as Python packages (§3.1). Yet despite this growth, developers have no empirical basis for choosing among them, and no map of *where the agents these kits produce actually break*: *which framework produces working agents, at what development cost, and how do the failures surface?*

The barrier is methodological. Traditional evaluation requires experts to manually implement benchmark workloads against each framework's API, an $O(N \times M)$ effort that introduces experimenter bias and cannot scale (TechEmpower, 2024; Transaction Processing Performance Council, 2024; MLCommons, 2024). Developer surveys (Wang et al., 2025; Liu et al., 2026; Hasan et al., 2025) reveal what developers *say* about frameworks, but not how the frameworks *perform*. Agent benchmarks (Jimenez et al., 2024; Yao et al., 2024; Liu et al., 2023; Mialon et al., 2023) compare *models*,

Work done during an internship at Microsoft CoreAI. [1]The Ohio State University, Columbus, OH, USA [2]Microsoft, Redmond, WA, USA. Correspondence to: Jintao Huang <huang.5692@osu.edu>, Xiaomin Li <xiaominli@microsoft.com>.

Failure Modes in Agentic AI Workshop (FAGEN) at the 43rd International Conference on Machine Learning, Seoul, South Korea, 2026. Non-archival. Copyright 2026 by the author(s).

[1]We release the full ADK Arena pipeline at https://github.com/jintao-h/ADK-Arena.

holding the framework constant. No prior study has attempted to compare frameworks at ecosystem scale.

We propose **LLM-as-a-Developer** (Figure 1), inspired by LLM-as-a-Judge (Zheng et al., 2023): instead of replacing human *evaluators*, we replace human *developers*. A single LLM-as-a-Developer agent learns each framework from its documentation, writes task-solving agent code, and iteratively repairs it through a validate-and-feedback loop, mirroring the explore–write–test–fix cycle a human developer follows when adopting a new SDK. By holding the developer constant and varying only the framework, each framework becomes a reproducible trigger that surfaces failures along two complementary axes: the *generation process* (tokens consumed, turns needed, failure rate) exposes API-usability failures, while the *execution results* (resolution rate, cost, latency) expose task-solving failures under a controlled, trace-level protocol.

We realize this methodology in **ADK Arena**, a fully automated pipeline that takes a framework's GitHub repository as input and produces benchmark scores as output, with no manually written agent code. The LLM developer operates inside isolated Docker environments with access to framework documentation and source code, iterating through a three-level validation pipeline that catches errors ranging from import failures to runtime crashes on real benchmark tasks. We evaluate across four established benchmarks spanning software engineering (SWE-bench), conversational tool use (τ^2-bench), multi-tool orchestration (MCP-Atlas), and terminal interaction (Terminal-Bench).

Applying ADK Arena to all 51 frameworks (204 agent-benchmark pairs) reveals that generation cost varies 5.6× across frameworks ($0.6 to $3.4 per agent), directly reflecting API complexity: well-designed APIs such as LangGraph and OpenAI Agents are among the cheapest to target, while large or poorly documented ones cost several times more to reach a working agent. No single framework dominates: the best single-benchmark ADK agents resolve up to 80% of tasks and can even beat general-purpose frontier coding agents at a fraction of the cost, yet the median framework resolves only 32%; strikingly, the developer model that *writes* the agent matters more than the one that runs it: Opus-authored agents resolve roughly twice as many tasks as GPT-authored ones on the same backbone. Ablation experiments show that documentation, raw source, and parametric knowledge are largely interchangeable: genuine framework usage stays within a 28 to 40% band across conditions (highest with raw source access), and even with no reference material 33% of agents still pass, so no single information source is a hard bottleneck.

Contributions. We make the following contributions:

1. **LLM-as-a-Developer** (§2), a methodology that uses an LLM-as-a-Developer agent as a controlled proxy developer, yielding generation effort as a quantitative measure of API usability and task performance as a measure of framework effectiveness.
2. **ADK Arena** (§3), a fully automated system with per-framework Docker isolation, a three-level Validate-and-Repair pipeline, token-level telemetry, and adapters for four benchmarks.
3. **An ecosystem-wide failure-mode analysis** (§4) from 51 frameworks (204 agent-benchmark pairs), surfacing three recurring failure modes (generation failure, execution collapse, and silent thin-wrapping), the first ecosystem-wide comparison of ADK frameworks against frontier coding agents, and information-source ablations that isolate the contributions of documentation, source code, and parametric knowledge.

2. Methodology: LLM-as-a-Developer

2.1. Limitations of Traditional Framework Evaluation

Software framework evaluation has a long tradition: TPC benchmarks compare database engines (Transaction Processing Performance Council, 2024), MLPerf compares deep-learning frameworks (MLCommons, 2024), and TechEmpower compares web frameworks (TechEmpower, 2024). These expert-driven approaches work well when frameworks are mature and few, but applying them to the rapidly growing ADK ecosystem (Xi et al., 2025; Wang et al., 2024a; Hong et al., 2024; Wu et al., 2024; Qian et al., 2024) exposes three limitations:

- **Experimenter bias.** Hand-written benchmark code reflects the author's familiarity with each framework, a serious confound when APIs evolve weekly. MAF-Bench (Orogat et al., 2026), the closest prior work, covers only 7 frameworks with hand-written micro-tasks.
- **Scalability.** N frameworks $\times$ M benchmarks requires $O(N \times M)$ expert implementations. We target 51 frameworks across 4 benchmarks (204 pairs).
- **No unified performance metric.** Developer surveys (Wang et al., 2025; Liu et al., 2026; Hasan et al., 2025) capture opinions but not runtime behavior; agent benchmarks (Jimenez et al., 2024; Yao et al., 2024; Liu et al., 2023; Mialon et al., 2023) compare models, not frameworks.

2.2. The Developer Analogy

We treat an LLM as a developer who learns each framework from documentation, writes code, and iteratively debugs until it works, following the same explore–write–test–fix cycle a human follows when adopting a new SDK. This builds on three ideas:

- **From LLM-as-a-Judge to LLM-as-a-Developer.** LLM-

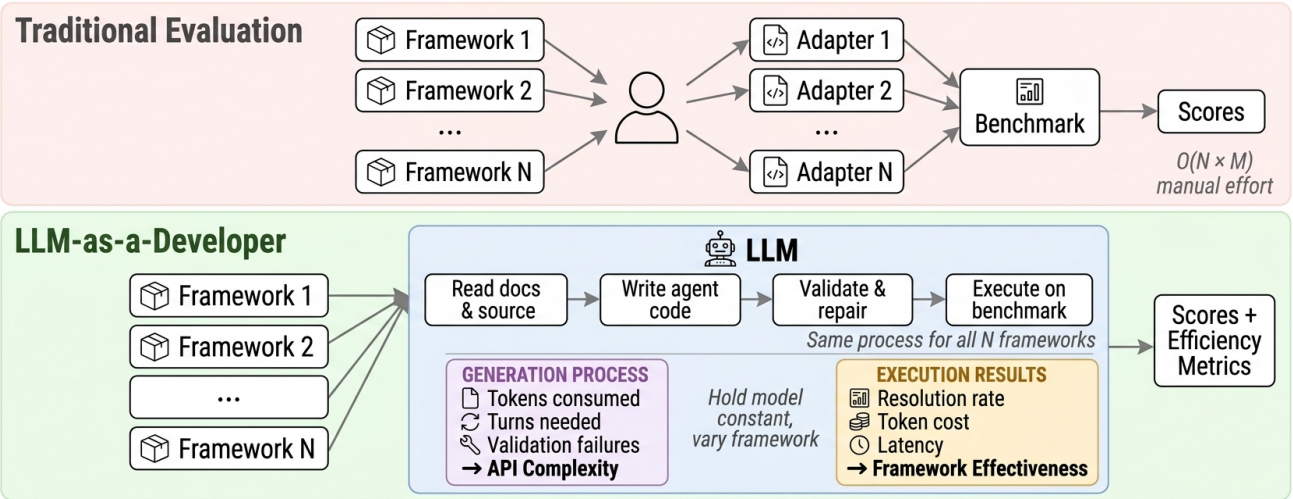

Figure 1. LLM-as-a-Developer.

as-a-Judge (Zheng et al., 2023) replaces human evaluators for scalable assessment. We apply the same principle to a different bottleneck (replacing human *developers*) to achieve scalability, reproducibility, and cost efficiency across dozens of frameworks.

- **Grounded in coding capability.** LLM coding has progressed from code completion (Chen et al., 2021; Li et al., 2023b; Rozière et al., 2024) to tool-augmented agents (Schick et al., 2024; Patil et al., 2023; Qin et al., 2024) that resolve >70% of real GitHub issues (Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2024b). The analogy is strongest for the "first-time developer" experience, making it well-suited for measuring onboarding difficulty.
- **Controlled-variable protocol.** Identical prompts, tools, and token budgets for every framework; information restricted to each framework's own repository (MLCommons, 2024; Transaction Processing Performance Council, 2024). Two signals emerge: *generation effort* (tokens, LLM calls, failures) measures API complexity; *execution performance* (resolution rate, cost) measures framework effectiveness.

2.3. Assumptions and Scope

The methodology rests on three key assumptions that directly address the limitations identified in §2.1:

- **Controlled and scalable.** The LLM starts from the same baseline for every framework, eliminating the experimenter-familiarity confound (Tichy, 1998; Sim et al., 2003). The pipeline requires only a framework's repository as input, scaling to 204 pairs without per-framework engineering, mirroring the scalability gains of LLM-as-a-Judge (Zheng et al., 2023) and LLM-based test generation (Lemieux et al., 2023; Xia et al., 2024).
- **Usability as a measurable signal.** Generation metrics (tokens consumed, LLM calls needed, validation failures) serve as quantitative proxies for API complexity and documentation quality, relating to cognitive dimensions of API usability (Clarke, 2004; Stylos & Myers, 2007; Rama & Kak, 2015). Unlike developer surveys (Wang et al., 2025) or issue mining (Hasan et al., 2025), these signals are objective and reproducible.
- **LLM capability as a constant.** We assume the LLM is a sufficiently capable developer held constant across frameworks, analogous to controlled SE experiments that fix expertise while varying tools (Ko et al., 2004; Murphy et al., 2006). We mitigate the training-data confound through a three-condition ablation (§4.4) that disentangles prior familiarity from documentation quality. Prior work has extensively validated that LLM judgments align with human judgments across diverse evaluation tasks (Zheng et al., 2023; Chiang et al., 2024; Li et al., 2024), supporting the use of LLM-based proxies for developer behavior.

3. ADK Arena

We realize the LLM-as-a-Developer methodology (§2) as a fully automated pipeline (Figure 2).

3.1. Environment Setup

We first curate a comprehensive set of ADKs and prepare isolated execution environments for each.

Collection. We collect 51 Python ADK frameworks from three sources: academic venues (`ICLR`, `NeurIPS`, `ICML`, `ACL`, `EMNLP`, `NAACL`, `AAAI`, `IJCAI`, `AAMAS`, `KDD`, `WWW`, `COLM`), GitHub topic/keyword search (≥1,000 stars), and curated awesome-lists. After filtering out agent applications,

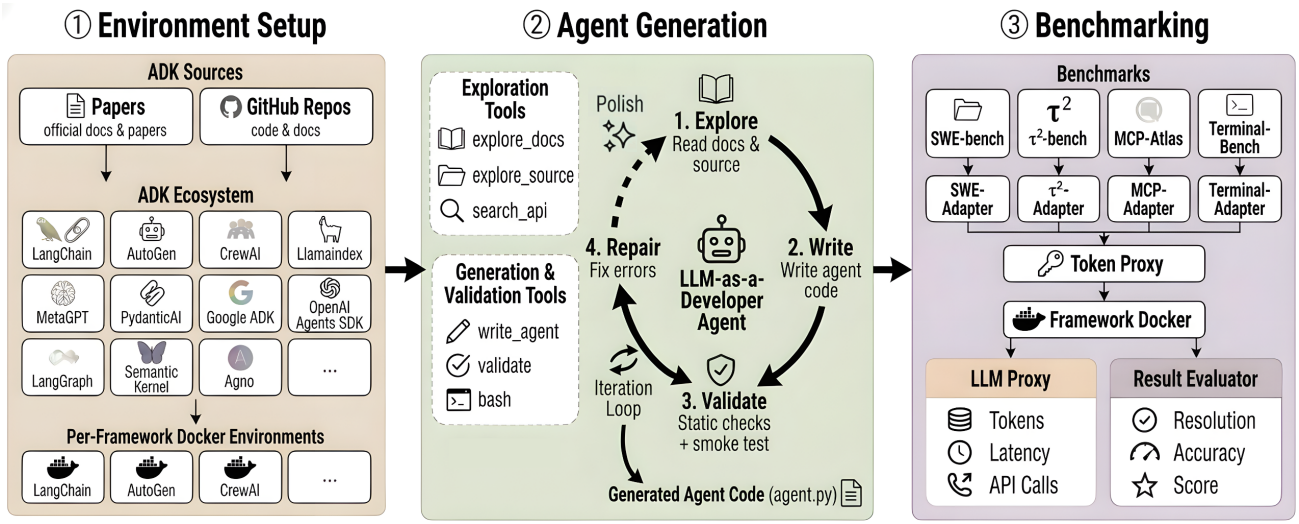

Figure 2. **The ADK Arena pipeline.** ① *Environment Setup*: collect repos, build Docker images. ② *Agent Generation*: explore docs, write code, validate and repair. ③ *Benchmarking*: execute in containers, score output.

no-code platforms, and non-Python projects, the final set spans both research and industry releases (Appendix A).

Per-framework environment. Each framework receives a dedicated Docker image built from a shared base with the framework's packages pre-installed. Benchmark-specific images extend this base with task-specific tooling (e.g., `git` and test runners for SWE-bench). All LLM traffic is routed through a local proxy, ensuring controlled and reproducible environments. Inside each container, the LLM developer has access to two information sources:

- **Curated documentation.** We collect each framework's documentation from its `docs/` directory into a unified format. The LLM developer reads it via the `explore_docs` tool that returns 20K characters per call, allowing incremental exploration of large API references.
- **Source code.** The framework's full repository is mounted read-only, enabling the LLM to inspect implementation details, discover undocumented APIs, and verify function signatures. Accessible via `explore_source` and `search_api` (regex grep) tools (Table 1).

3.2. Agent Generation

We build an *LLM-as-a-Developer* agent using OpenAI Codex (GPT-5.4) that automates the entire learning-and-coding workflow a human developer would follow. Given only a framework's documentation and source code, it explores the API, writes a task-solving agent, and iteratively repairs it until validation passes.

Agent interface. Each generated agent is a single Python file (`agent.py`) exposing one entry point: `solve(prompt: str, workdir: str) -> str`. The function receives a natural-language task description and a working directory, then orchestrates the framework's agents and tools to produce a solution. This uniform interface decouples generation from evaluation. Any benchmark adapter can invoke the agent identically regardless of which framework it wraps.

Generation loop. The LLM-as-a-Developer drives the generation loop with a 5-minute budget. Each iteration follows: *explore* documentation/source, *write* agent code, *validate* against a three-level pipeline, and *repair* based on diagnostic feedback. The loop terminates when all three validation levels pass or the time budget expires.

Iteration loop. Within each generation loop iteration, an inner *iteration loop* repeatedly validates the current code and revises it based on structured diagnostic feedback, enabling targeted single-edit repairs rather than blind regeneration. Only once validation passes all three levels does the agent proceed to the next round of exploration and polishing. The validation pipeline has increasing fidelity:

1. **Step 1: Static analysis**: compile check (`py_compile`), import verification, framework usage check (reject raw API fallbacks), and 40+ AST/regex patterns detecting common anti-patterns (Appendix B).
2. **Step 2: Real LLM smoke test**: execute `solve()` with a real LLM through a token-recording proxy. Verifies that the agent can successfully call the LLM, handle the response format, and return a valid result.
3. **Step 3: Real benchmark task**: run one task from the target benchmark's official task set under realistic conditions. We do not run the task to completion; instead, a

Table 1. Development tools available to the LLM developer during agent generation.

Exploration		Development	
`explore_docs`	Browse curated API documentation	`write_agent`	Write `agent.py` with auto compatibility fixes
`explore_source`	Read framework `.py` source and directory structure	`validate`	Run 3-level Validate-and-Repair pipeline
`search_api`	Regex grep over source for API usage patterns	`bash`	Run shell commands (`python -c`, version checks)

benchmark-specific early-exit check monitors the proxy for expected behavior (e.g., minimum number of LLM calls with tool use for MCP-Atlas). If the check is satisfied, validation passes immediately. Otherwise, after a short timeout the task is marked as failed. Passing Step 3 guarantees the agent can execute on the full benchmark without crashing.

On failure at any level, the validator pattern-matches the error against 70+ known signatures and emits a structured diagnostic hint containing a root-cause diagnosis and a concrete fix. For example, a `ConnectionRefusedError` produces:

> **DIAGNOSIS**: Agent is connecting to a hardcoded URL (e.g., `https://api.openai.com`) instead of the proxy endpoint. The ADK execution environment routes all LLM traffic through a local proxy at `ADK_BASE_URL`.
>
> **FIX**: Replace all hardcoded URLs with `os.environ["ADK_BASE_URL"]`.

Development tools. To support the generation and iteration loops, we provide the LLM developer with six CLI tools inside the per-framework Docker container (Table 1). The generated agent must use the framework's native API (raw OpenAI/httpx fallbacks are rejected by validation).

3.3. Benchmarking

We evaluate on four established benchmarks (50 tasks each): SWE-bench Verified (Jimenez et al., 2024), τ^2-bench (Yao et al., 2024), MCP-Atlas (Bandi et al., 2026), and Terminal-Bench (Merrill et al., 2026). Execution uses GPT-5.4 Nano via the token proxy.

Benchmark adapters. The four benchmarks differ substantially in input format, execution environment, and scoring methodology. Each *adapter* encapsulates these differences behind a uniform interface: given an agent's `solve()` function, a task prompt, and a working directory, the adapter handles environment setup, execution orchestration, and evaluation scoring. This allows a single generated agent to be evaluated identically across all four benchmarks without benchmark-specific code.

Unified LLM proxy. ADK frameworks use diverse HTTP stacks (OpenAI SDK, `httpx`, `requests`) and target different provider APIs (OpenAI, Anthropic, Google). To evaluate all 51 frameworks against the same backbone LLM regardless of which provider API each natively targets, we route all LLM requests through a transparent *token proxy*:

- **Protocol translation**: full bidirectional conversion between Anthropic and OpenAI message formats, so each framework sees its expected wire protocol regardless of the backend LLM.
- **Metric capture**: extracting per-call token counts (input, output, cached) from both streaming and non-streaming responses, and logging latency, request/response content, tool-call sequences, and per-call breakdowns (system prompt, tool schemas, history) for post-hoc analysis.

4. Evaluation

We evaluate 51 frameworks along two dimensions: generation performance (§4.1), measuring how efficiently the LLM developer learns and produces working agent code, and execution performance (§4.2), measuring how well the resulting agents solve real tasks. We then compare against frontier coding agents (§4.3) and stress-test the pipeline by ablating information sources (§4.4).

4.1. Generation Performance

Table 2 reports generation effort for all 51 frameworks across four benchmarks. Of the 408 generated agents (two per framework–benchmark pair, one from each developer model), 232 (57%) pass all three validation levels, showing that fully automated agent generation is viable for over half of the ecosystem. Per-agent generation cost (averaged across benchmarks) ranges from \$0.6 to \$3.4, a 5.6× spread.

Cost tracks API usability rather than raw spend. The most robust frameworks pass all four benchmarks under *both* developer models, *Haystack* (\$2.0 / \$1.3 per agent for GPT / Opus) and *AG2* (\$2.8 / \$1.2), exposing clean, learnable APIs. The cheapest runs are *not* the best: *Langroid* (\$0.2–0.7), *Solace* (\$0.9–1.5), and *Qwen Agent* produce few or no passing agents, because with no productive path into the API the developer gives up early and each attempt stays short. The highest-cost frameworks (*MCP Agent*, *AgentFramework*, *AtomicAgents*, all near \$3.4 on the GPT side) carry the largest API surfaces, yet high spend does not doom them: MCP Agent still passes 3/4. Cost thus signals API complexity, not a clean success/failure split.

> **Finding.** The pipeline produces a validated agent for 57% of runs (232/408). Per-agent generation cost varies 5.6× across frameworks and serves as a quantitative proxy for API complexity, though cost alone does not predict success.

Table 2. Generation effort for 51 ADK frameworks (NORMAL condition), ordered by PyPI monthly downloads. Values averaged across benchmarks. Each cell shows GPT-5.4 / Opus-4.6 results. **Pass**: number of benchmarks (out of 4) for which the generated agent passes all three validation levels, which means it can execute the benchmark but does not necessarily finish the task successfully.

Framework	Pass	In Tok	Out Tok	Cache Tok	Cost ($)	Framework	Pass	In Tok	Out Tok	Cache Tok	Cost ($)
LangChain	3 / 3	1,077K / 1,119K	14K / 12K	851K / 1,057K	1.24 / 1.14	Swarms	2 / 1	1,206K / 1,308K	12K / 9K	696K / 1,260K	1.83 / 1.08
Anthropic SDK	4 / 1	1,704K / 1,572K	14K / 13K	1,073K / 1,512K	2.38 / 1.37	MCP Agent	3 / 3	2,444K / 1,565K	17K / 11K	1,538K / 1,506K	3.39 / 1.32
LangGraph	2 / 4	1,008K / 693K	10K / 15K	556K / 630K	1.58 / 1.00	Guidance	3 / 3	1,904K / 1,847K	12K / 11K	1,104K / 1,774K	2.81 / 1.53
OpenAI Agents	3 / 3	2,013K / 1,085K	12K / 13K	1,128K / 1,022K	3.04 / 1.15	AtomicAgents	3 / 2	2,003K / 1,115K	13K / 11K	1,066K / 1,063K	3.14 / 1.08
PydanticAI	3 / 3	1,469K / 891K	9K / 11K	808K / 846K	2.24 / 0.93	Griptape	3 / 0	1,706K / 1,824K	13K / 11K	939K / 1,767K	2.63 / 1.45
LlamaIndex	1 / 1	1,726K / 1,095K	12K / 11K	1,127K / 1,050K	2.32 / 1.02	Upsonic	2 / 1	1,321K / 1,449K	9K / 7K	849K / 1,397K	1.79 / 1.14
CrewAI	3 / 4	1,433K / 1,155K	10K / 12K	817K / 1,040K	2.16 / 1.39	ControlFlow	0 / 0	1,972K / 1,107K	8K / 8K	1,214K / 1,065K	2.73 / 0.94
Google ADK	2 / 3	1,777K / 1,436K	18K / 11K	1,091K / 1,383K	2.58 / 1.24	AgencySwarm	3 / 3	1,796K / 1,156K	12K / 8K	1,078K / 1,051K	2.59 / 1.26
Strands	2 / 2	1,217K / 1,064K	9K / 8K	815K / 1,019K	1.60 / 0.93	Agently	2 / 2	1,648K / 1,288K	9K / 15K	1,018K / 1,233K	2.30 / 1.27
SemanticKernel	2 / 2	1,397K / 1,180K	8K / 12K	843K / 1,138K	1.99 / 1.07	Lagent	1 / 2	1,456K / 1,345K	9K / 20K	816K / 1,284K	2.20 / 1.44
Agno	4 / 3	1,953K / 1,452K	17K / 14K	1,204K / 1,365K	2.79 / 1.46	AgentUniverse	1 / 4	1,999K / 1,482K	16K / 15K	1,255K / 1,405K	2.81 / 1.47
AG2	4 / 4	1,679K / 1,078K	13K / 14K	839K / 1,016K	2.75 / 1.15	EvoAgentX	1 / 2	1,558K / 1,665K	9K / 14K	929K / 1,597K	2.24 / 1.49
AutoGen	3 / 4	1,812K / 1,008K	12K / 14K	1,123K / 953K	2.55 / 1.10	Nerve	3 / 3	2,271K / 1,484K	12K / 11K	1,517K / 1,409K	2.95 / 1.37
Composio	2 / 4	2,171K / 1,481K	13K / 14K	1,301K / 1,424K	3.12 / 1.34	AgentSquad	3 / 4	2,048K / 1,217K	16K / 14K	1,139K / 1,100K	3.15 / 1.49
Haystack	4 / 4	1,447K / 1,239K	13K / 14K	935K / 1,156K	1.99 / 1.29	MotleyCrew	3 / 2	1,735K / 1,356K	14K / 10K	1,057K / 1,307K	2.49 / 1.14
SmoLAgents	2 / 3	1,014K / 1,057K	9K / 12K	596K / 1,007K	1.51 / 1.04	TaskflowAI	1 / 1	1,781K / 446K	11K / 7K	985K / 399K	2.71 / 0.61
AgentFramework	3 / 4	2,040K / 1,447K	13K / 13K	995K / 1,361K	3.36 / 1.44	CouncilAI	1 / 2	1,814K / 2,618K	14K / 21K	1,050K / 2,366K	2.70 / 2.97
Qwen Agent	2 / 0	1,265K / 920K	10K / 10K	695K / 878K	1.95 / 0.90	AgentFlow	1 / 1	1,567K / 1,926K	10K / 11K	931K / 1,850K	2.27 / 1.59
AgentScope	2 / 2	1,371K / 2,211K	10K / 12K	864K / 2,142K	1.91 / 1.71	AutoAgent	2 / 3	1,807K / 763K	10K / 8K	1,075K / 715K	2.60 / 0.79
PraisonAI	3 / 3	2,138K / 1,447K	14K / 13K	1,314K / 1,387K	3.02 / 1.31	AgentLite	3 / 3	1,565K / 760K	10K / 43K	939K / 706K	2.26 / 1.71
CAMEL	3 / 3	1,550K / 1,083K	14K / 11K	878K / 1,030K	2.36 / 1.06	Octotools	3 / 2	1,783K / 936K	13K / 30K	1,109K / 832K	2.51 / 1.69
FastAgent	0 / 4	2,385K / 1,786K	10K / 12K	1,583K / 1,721K	3.09 / 1.48	GPTSwarm	2 / 2	2,039K / 1,552K	11K / 12K	1,335K / 1,497K	2.71 / 1.33
Solace	1 / 1	979K / 995K	10K / 7K	565K / 946K	1.49 / 0.90	Taskweaver	1 / 1	2,091K / 730K	13K / 8K	1,354K / 682K	2.82 / 0.79
Langroid	0 / 0	512K / 171K	4K / 1K	334K / 142K	0.69 / 0.24	AutoAgents	1 / 2	2,072K / 1,165K	12K / 36K	1,269K / 1,106K	2.92 / 1.75
BeeAI	2 / 2	2,169K / 1,346K	17K / 37K	1,416K / 1,288K	2.94 / 1.85	OpenSage	3 / 1	1,873K / 991K	11K / 9K	1,140K / 938K	2.65 / 0.95
MetaGPT	3 / 1	1,921K / 1,063K	11K / 7K	1,202K / 1,010K	2.66 / 0.95						

4.2. Execution Performance

Table 6 (Appendix C) reports the resolution rate, token usage, and cost for all 51 frameworks across the four benchmarks under the "normal" information condition (framework documentation + repository access). For each framework we evaluate two generated agents, one written by GPT-5.4 (Codex), one by Opus-4.6 (Claude Code), both executing on GPT-5.4 Nano as the backbone LLM. The generated agents are functional (they build, execute, and consume LLM tokens) but most fail to solve their benchmark tasks, with successful resolutions concentrated on a small subset of framework–benchmark pairs.

Resolution varies sharply by benchmark. τ^2-bench is by far the most tractable: among agents that actually run, the median resolves 64% and the best 80%, because its conversational, tool-calling format is comparatively forgiving and leans on the shared backbone rather than on long, self-directed trajectories. The coding and terminal benchmarks are much harder, with medians of just 18% (SWE-bench) and 14% (TerminalBench): both demand long-horizon, iterative editing and command execution, where a single early misstep cascades into failure and the lightweight generated agents rarely sustain the loop. MCP-Atlas sits in between (median 39% among the frameworks with execution data), rewarding agents that correctly discover and route requests through the MCP tool server. The spread within a single benchmark is also wide, from 0% to 80% on SWE-bench, confirming that the framework and its generated agent, not the task suite alone, drive the outcome.

The developer model that *writes* the agent matters more than the model that runs it. Although both variants execute on the same GPT-5.4 Nano backbone, agents authored by Opus-4.6 resolve roughly twice as many tasks as those authored by GPT-5.4 (mean 41% vs. 22%, median 46% vs. 8%). The difference is visible in the code: GPT-5.4 frequently emits a minimal agent that issues a single model call and returns, whereas Opus-4.6 more often wires up the framework's full loop, registering tools and iterating over multiple turns. Agent *code quality*, fixed at generation time, thus largely determines execution success, reinforcing that passing validation is necessary but far from sufficient.

Finding. ADK-generated agents are functional (they execute and consume tokens) but most fail to solve benchmark tasks: generation-time validation ensures syntactic correctness, not task-solving capability. Outcomes diverge sharply even among validated agents, and the developer model is decisive: Opus-authored agents resolve roughly twice as many tasks as GPT-authored ones (41% vs. 22%), all on the same execution backbone.

Table 3. Production coding agents on four benchmarks. All agents use GPT-5.4 Nano. Token, cost, time, and call columns are per-task averages over the 50-task subset.

System	Version	Benchmark	Rate (%)	In Tok	Out Tok	Cache Tok	Cost (USD)	Avg Time (s)	Avg Calls
Claude Code	v2.1.150	SWE-bench	48.0	179.0K	1.5K	164.9K	0.07	104.7	25.6
		τ^2-bench	46.0	15.5K	226	7.1K	0.00	65.1	3.0
		MCP-Atlas	38.0	26.6K	663	25.1K	0.01	38.9	9.2
		TerminalBench	30.0	254.1K	4.2K	243.7K	0.10	207.7	27.8
Codex CLI	v0.95.0	SWE-bench	60.0	673.7K	2.9K	651.6K	0.02	80.5	26.8
		τ^2-bench	46.0	233.4K	1.1K	219.7K	0.01	178.3	3.6
		MCP-Atlas	36.0	167.2K	1.0K	153.1K	0.01	32.5	3.3
		TerminalBench	16.0	372.9K	3.2K	353.4K	0.01	112.9	12.7
Copilot	v1.0.39	SWE-bench	74.0	2,578.3K	41.1K	2,094.4K	0.57	592.5	64.8
		τ^2-bench	46.0	554.2K	40.3K	418.8K	0.16	394.8	52.3
		MCP-Atlas	58.0	327.1K	16.4K	198.9K	0.09	180.0	14.5
		TerminalBench	46.0	425.5K	59.2K	327.9K	0.16	860.8	36.6
OpenHands	v1.6.0	SWE-bench	60.0	2,357.8K	47.6K	1,550.0K	0.29	617.4	44.6
		τ^2-bench	46.0	660.6K	26.1K	572.3K	0.06	806.0	28.5
		MCP-Atlas	52.0	498.7K	12.6K	269.4K	0.07	248.2	16.7
		TerminalBench	36.0	1,190.1K	44.2K	552.4K	0.19	830.5	20.6
OpenCode	v1.15.10	SWE-bench	76.0	885.9K	7.4K	851.1K	0.10	271.9	31.2
		τ^2-bench	48.0	28.6K	1.1K	14.7K	0.01	308.3	2.2
		MCP-Atlas	18.0	92.4K	1.9K	84.4K	0.01	66.2	5.0
		TerminalBench	38.0	655.8K	12.3K	631.9K	0.08	298.0	22.7

4.3. Comparison with Frontier Coding Agents

To contextualize ADK-generated agents, we compare against five frontier coding agents (Table 3): Claude Code (Anthropic, 2025), GitHub Copilot (GitHub, 2025), Codex (OpenAI, 2025), OpenHands (Wang et al., 2024c), and OpenCode (ope, 2025). All use GPT-5.4 Nano as the backbone LLM to control for model effects. Among the frontier agents themselves, Copilot achieves the highest average resolution across the four benchmarks (56%: 74% SWE-bench, 58% MCP-Atlas, 46% TerminalBench, 46% τ^2-bench), followed by OpenHands (48.5%: 60% SWE-bench, 52% MCP-Atlas, 46% τ^2-bench, 36% TerminalBench), OpenCode (45%: 76% SWE-bench, 48% τ^2-bench, 38% TerminalBench, 18% MCP-Atlas), Claude Code (40.5%: 48% SWE-bench, 46% τ^2-bench, 38% MCP-Atlas, 30% TerminalBench), and Codex (39.5%: 60% SWE-bench, 46% τ^2-bench, 36% MCP-Atlas, 16% TerminalBench). OpenCode posts the single highest scores on SWE-bench (76%) and τ^2-bench (48%), while Copilot leads on MCP-Atlas (58%). Crucially, the top of the accuracy ranking is also the least efficient: the two highest-resolution agents (Copilot 56%, OpenHands 48.5%) are by far the most token-hungry, consuming 3.9M and 4.7M input tokens across the four benchmarks. Claude Code and Codex are the standouts on efficiency: Claude Code resolves a competitive 40.5% on just 0.48M tokens (8–10$\times$ fewer than Copilot or OpenHands) at roughly \$0.05 per task, and Codex is cheapest of all at \$0.01–0.02 per task, both reaching near the resolution of the leaders at a small fraction of their cost. The most accurate agent is thus not the most efficient, and a sharp accuracy–efficiency tradeoff separates the field.

Set against the ADK-generated agents, these frontier systems are strong but not dominant. The best ADK-generated agents (Table 6) are competitive on individual benchmarks (resolving up to 80% of tasks), and because each is a lightweight program generated for a single benchmark, the best ones can even exceed the general-purpose frontier agents at a fraction of the cost: on MCP-Atlas, *Agno* resolves 68% of tasks at 38K input tokens per task, beating Copilot's 58% at 327K (roughly 9$\times$ fewer). The generated agents reach comparably low per-task cost in part because they terminate early on unsolved tasks, so the comparison is not purely an apples-to-apples one, but the top performers genuinely match or beat the frontier on their target benchmark. Performance is nonetheless highly uneven across the ecosystem, with the median framework–benchmark pair resolving only 32%, so a well-chosen framework can match hand-built agents on some tasks while most frameworks fall well short.

Finding. Lightweight, single-benchmark ADK agents can beat general-purpose frontier agents far more cheaply. Yet this is a top-of-distribution effect: the median framework resolves only 32%, so framework choice alone does not guarantee strong performance.

4.4. Stress Testing: Ablating Information Sources

We ablate information sources across all 51 frameworks to test how robust generation is under degraded conditions (Table 4). The DOCS ONLY condition provides curated documentation but disables source code exploration. The SOURCE ONLY condition provides raw source code access but removes curated documentation. The NONE condition removes all reference material, forcing the LLM to rely entirely on parametric knowledge from pre-training.

Here **Pass** counts agents that both pass validation and use the framework *natively* (thin-wrapper agents excluded), measuring genuine framework usage under each condition. The striking pattern is that genuine usage does *not* rise with access to more documentation: SOURCE ONLY (40%, raw code but no curated prose) yields the highest rate, above both NONE (33%) and DOCS ONLY (28%), a first sign that more curated information is not always better.

- **Source only** yields the *highest* rate of genuine framework usage (40%, 164/408): seeing the actual API in source code leads the developer to native calls more often than curated prose does.
- **Docs only** is the lowest (28%, 114/408): curated documentation alone leaves the developer more likely to hand-roll a thin wrapper than when it can read the source.
- **None** is surprisingly survivable (33%, 136/408): even with no documentation or source, a third of agents still use the framework natively, drawing on parametric knowledge of popular frameworks (*LangChain*, *CrewAI*, *OpenAI Agents*). Strikingly, NONE even edges out DOCS ONLY (33% vs. 28%): the documentation is complete, but its worked examples (often minimal quickstarts) anchor the developer, which tends to copy or lightly edit those snippets without adapting them to the benchmark, leaving a thin wrapper rather than an agent that exercises the framework's native API. Stripped of examples to imitate, the developer instead falls back on the broader native-API usage it memorized from real code during pre-training. The survivors skew popular (the median framework that still passes ranks 22nd by monthly downloads versus 34th for those that fail) because widely-used APIs are heavily represented in the pre-training corpus, so the developer model has effectively memorized them; niche frameworks, sparse in that corpus, collapse once all reference material is removed.
- Token usage is nearly flat across the three conditions, so the differences reflect what the developer can *discover* about the API, not how much it explores.

Finding. Genuine framework usage stays within a narrow 28–40% band across information conditions and, crucially, does not increase with more curated documentation: DOCS ONLY is the lowest (28%), while withholding documentation in favor of raw source *raises* native usage (SOURCE ONLY 40%, NONE 33%). Documentation's worked examples anchor the developer toward thin wrappers; source code and parametric knowledge are largely substitutable, and no single information source is a hard bottleneck.

5. Limitations

Single execution model. All generated agents run on GPT-5.4 Nano as the execution backbone. This is a controlled-variable design rather than a confound: fixing the execution model isolates framework effects from model–framework interactions, precisely the quantity we set out to measure. It does not bias the comparison, since every framework reaches the model through the same in-container proxy that normalizes their heterogeneous request formats (OpenAI chat-completions, the Azure Responses API, Anthropic Messages) onto one backend, so all 51 frameworks are evaluated against an identical model on identical infrastructure. We chose Nano for cost: the full campaign would cost $\sim$40$\times$ more with GPT-5.4 or Claude Opus. The pipeline is model-agnostic, so sweeping additional model families is straightforward future work.

Training-data bias. The generator LLM has more training data for popular frameworks (LangChain, CrewAI) than newer ones. We argue this reflects a real-world advantage: frameworks with richer community resources are genuinely easier for *any* developer to use. Our ablation (§4.4) shows that even with no documentation or source code, 33% of agents (136/408) still use the framework natively, drawing on parametric knowledge of popular frameworks. Parametric knowledge thus provides a floor without being the dominant factor.

6. Conclusion

We presented LLM-as-a-Developer, an evaluation methodology using an LLM as a proxy developer, and ADK Arena, a fully automated pipeline applying it across 51 frameworks and four benchmarks. Treating each framework as a reproducible trigger surfaces three recurring failure modes: generation cost varies 5.6$\times$ as a proxy for API complexity; the best agents match frontier coding agents (up to 80%) yet the median resolves only 32%; and genuine framework usage stays within a narrow 28–40% band across information conditions, so documentation, source code, and parametric knowledge are largely substitutable, with the developer side of the agent stack, not the runtime alone, proving decisive.

References

Opencode. https://opencode.ai, 2025.

Anthropic. The Claude 3 model family: Opus, Sonnet, Haiku. *Technical Report*, 2024.

Anthropic. Claude code. https://docs.anthropic.com/en/docs/agents-and-tools/claude-code, 2025.

Bandi, C., Hertzberg, B., Boo, G., Polakam, T., Da, J., Hassaan, S., Sharma, M., Park, A., Hernandez, E., Rambado, D., Salazar, I., Cruz, R., Rane, C., Levin, B., Kenstler, B., and Liu, B. MCP-Atlas: A large-scale benchmark for tool-use competency with real MCP servers. *arXiv preprint arXiv:2602.00933*, 2026.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33: 1877–1901, 2020.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Chen, W., Su, Y., Zuo, J., Yang, C., Yuan, C., Chan, C.-M., Yu, H., Lu, Y., Hung, Y.-H., Qian, C., et al. AgentVerse: Facilitating multi-agent collaboration and exploring emergent behaviors. *arXiv preprint arXiv:2308.10848*, 2024.

Chiang, W.-L., Zheng, L., Sheng, Y., Angelopoulos, A. N., Li, T., Li, D., Zhang, H., Zhu, B., Jordan, M., Gonzalez, J. E., and Stoica, I. Chatbot arena: An open platform for evaluating LLMs by human preference. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024.

Clarke, S. Measuring API usability. *Dr. Dobb's Journal*, 2004.

Decan, A., Mens, T., and Grosjean, P. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 24(1):381–416, 2019.

GitHub. Github copilot coding agent. https://github.com/features/copilot, 2025.

Hasan, M. M., Li, H., Fallahzadeh, E., Rajbahadur, G. K., Adams, B., and Hassan, A. E. An empirical study of testing practices in open source AI agent frameworks and agentic applications. *arXiv preprint arXiv:2509.19185*, 2025.

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Zhang, C., Wang, J., Wang, Z., Yau, S. K. S., Lin, Z., et al. MetaGPT: Meta programming for a multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2024.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. SWE-bench: Can language models resolve real-world GitHub issues? In *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024.

Ko, A. J., Myers, B. A., and Aung, H. H. Six learning barriers in end-user programming systems. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 199–206, 2004.

Lemieux, C., Inala, J. P., Qi, S. K., Lahiri, S. K., Sen, S., and Gao, K. CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *International Conference on Software Engineering (ICSE)*, 2023.

Li, G., Hammoud, H. A. A. K., Itani, H., Khizbullin, D., and Ghanem, B. CAMEL: Communicative agents for "mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36, 2023a.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al. StarCoder: May the source be with you! *arXiv preprint arXiv:2305.06161*, 2023b.

Li, T., Chiang, W.-L., Frick, E., Dunlap, L., Wu, T., Zhu, B., Gonzalez, J. E., and Stoica, I. From crowdsourced data to high-quality benchmarks: Arena-hard and BenchBuilder pipeline. *arXiv preprint arXiv:2406.11939*, 2024.

Liu, C., Chen, S., Fan, L., Chen, B., Liu, Y., et al. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2022.

Liu, D., Upadhyay, K., Chhetri, V., Siddique, A., and Farooq, U. A large-scale study on the development and issues of multi-agent AI systems. *arXiv preprint arXiv:2601.07136*, 2026.

Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., Gu, Y., Ding, H., Men, K., Yang, K., et al. AgentBench: Evaluating LLMs as agents. *arXiv preprint arXiv:2308.03688*, 2023.

Merrill, M. A., Carlini, N., Shaw, A. G., Schmidt, L., et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

Mialon, G., Fourrier, C., Swift, C., Wolf, T., LeCun, Y., and Scialom, T. GAIA: A benchmark for general AI assistants. *arXiv preprint arXiv:2311.12983*, 2023.

MLCommons. MLPerf: A benchmark suite for machine learning. https://mlcommons.org/benchmarks/, 2024.

Murphy, G. C., Kersten, M., and Findlater, L. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.

OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

OpenAI. Codex cli. https://github.com/openai/codex, 2025.

Orogat, A., Rostam, A., and Mansour, E. Understanding multi-agent LLM frameworks: A unified benchmark and experimental analysis. *arXiv preprint arXiv:2602.03128*, 2026.

Patil, S. G., Zhang, T., Wang, X., and Gonzalez, J. E. Gorilla: Large language model connected with massive APIs. *arXiv preprint arXiv:2305.15334*, 2023.

Qian, C., Liu, W., Liu, H., Chen, N., Dang, Y., Li, J., Yang, C., Chen, W., Su, Y., Cong, X., et al. ChatDev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2024.

Qin, Y., Liang, S., Ye, Y., Zhu, K., Yan, L., Lu, Y., Lin, Y., Cong, X., Tang, X., Qian, B., et al. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. *arXiv preprint arXiv:2307.16789*, 2024.

Rama, G. M. and Kak, A. Some structural measures of API usability. *Software: Practice and Experience*, 45(1): 75–110, 2015.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., et al. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2024.

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., and Scialom, T. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36, 2024.

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2023.

Sim, S. E., Easterbrook, S., and Holt, R. C. Using benchmarking to advance research: A challenge to software engineering. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pp. 74–83, 2003.

Stylos, J. and Myers, B. Mapping the space of API design decisions. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 50–60, 2007.

Sumers, T. R., Yao, S., Narasimhan, K., and Griffiths, T. L. Cognitive architectures for language agents. *Transactions on Machine Learning Research*, 2024.

TechEmpower. TechEmpower web framework benchmarks. https://www.techempower.com/benchmarks/, 2024.

Tichy, W. F. Should computer scientists experiment more? *IEEE Computer*, 31(5):32–40, 1998.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Transaction Processing Performance Council. TPC-H: Decision support benchmark. https://www.tpc.org/tpch/, 2024.

Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6), 2024a.

Wang, X., Chen, Y., Yuan, L., Zhang, Y., Li, Y., Peng, H., and Ji, H. Executable code actions elicit better LLM agents. *arXiv preprint arXiv:2402.01030*, 2024b.

Wang, X., Li, B., Song, Y., Xu, F. F., Tang, X., et al. OpenHands: An open platform for AI software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024c.

Wang, Y., Xu, X., Chen, J., Bi, T., Gu, W., and Zheng, Z. An empirical study of agent developer practices in AI agent frameworks. *arXiv preprint arXiv:2512.01939*, 2025.

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., et al. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2024.

Xi, Z., Chen, W., Guo, X., He, W., Ding, Y., Hong, B., Zhang, M., Wang, J., Jin, S., Zhou, E., et al. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 2025.

Xia, C. S., Paltenghi, M., Tian, J. L., Pradel, M., and Zhang, L. Fuzz4all: Universal fuzzing with large language models. In *International Conference on Software Engineering (ICSE)*, 2024.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. SWE-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. ReAct: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2023.

Yao, S., Shinn, N., Razavi, P., and Narasimhan, K. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E. P., Zhang, H., Gonzalez, J. E., and Stoica, I. Judging LLM-as-a-judge with MT-Bench and chatbot arena. In *Advances in Neural Information Processing Systems*, 2023.

Zimmermann, M., Staicu, C.-A., Tenny, C., and Pradel, M. Small world with high risks: A study of security threats in the npm ecosystem. In *Proceedings of the 28th USENIX Security Symposium*, 2019.

Table 4. Generation effort under three ablation conditions (D / S / N = Docs only / Source only / No reference). Values averaged across benchmarks (K = thousands). **Pass**: number of benchmarks (of 4) that pass validation *and* use the framework natively (thin-wrapper agents excluded). First row: GPT-5.4 (Codex). Second row: Opus-4.6 (Claude Code).

Framework	Pass	In	Out	Cache	$
LangChain	1/1/1	1719/1651/1342	15/15/13	1084/1072/742	2.4/2.3/2.1
	3/3/2	1921/1331/1247	13/17/16	1798/1240/1172	1.5/1.5/1.3
Anthropic SDK	0/0/0	1962/1824/1505	15/14/14	1366/1188/833	2.5/2.5/2.3
	0/0/0	1287/1751/1591	14/14/16	1236/1694/1514	1.2/1.5/1.5
LangGraph	1/2/3	2178/1738/1223	15/14/15	1458/1097/693	2.9/2.4/1.9
	3/3/3	1353/887/1217	16/15/17	1270/822/1138	1.5/1.1/1.3
OpenAI Agents	2/3/2	1932/1592/1650	14/15/16	1365/992/998	2.4/2.3/2.4
	3/3/4	1411/1462/1364	15/14/14	1325/1391/1257	1.5/1.4/1.5
PydanticAI	2/3/3	2183/1736/1517	14/15/17	1464/1087/977	2.9/2.5/2.1
	2/3/3	1262/1221/1196	12/13/13	1207/1163/1120	1.2/1.2/1.3
LlamaIndex	0/1/0	2583/2144/1654	13/16/13	1911/1400/963	3.0/2.9/2.5
	1/3/3	1307/1244/1092	14/15/15	1256/1172/1032	1.2/1.3/1.2
CrewAI	1/3/2	2550/1993/1302	14/14/15	1738/1127/754	3.3/3.0/2.0
	2/3/3	1229/876/822	12/12/11	1170/817/772	1.2/1.0/0.9
Google ADK	2/3/3	1872/2251/2217	15/14/13	1226/1560/1401	2.5/2.8/3.0
	2/3/1	1860/1410/1332	12/13/12	1797/1356/1286	1.5/1.3/1.2
Strands	3/3/3	2346/1787/1987	14/16/13	1665/1073/1215	2.9/2.6/2.8
	3/3/2	1398/1634/1566	14/14/18	1335/1566/1507	1.3/1.5/1.5
SemanticKernel	3/2/2	1981/1737/1758	15/20/17	1362/1085/1114	2.6/2.5/2.5
	2/3/3	1562/1362/1198	15/14/14	1510/1306/1149	1.4/1.3/1.2
Agno	3/1/2	2317/2143/1587	16/14/15	1545/1469/902	3.1/2.7/2.4
	1/2/3	1602/1590/1283	14/15/16	1539/1527/1218	1.4/1.4/1.2
AG2	1/4/3	479/2091/1799	4/13/17	216/1319/1220	0.8/2.9/2.4
	3/3/3	1332/1197/991	16/18/23	1223/1045/921	1.5/1.2/1.4
AutoGen	1/3/1	1827/1973/1889	14/14/15	1129/1296/1103	2.6/2.6/2.8
	2/3/3	1375/995/952	15/17/16	1313/926/895	1.3/1.2/1.1
Composio	0/0/0	2109/1567/1827	19/14/16	1479/887/1205	2.7/2.4/2.5
	0/0/0	1591/1804/1658	14/14/14	1516/1730/1602	1.5/1.6/1.4
Haystack	2/3/3	2349/1973/1789	16/14/15	1575/1289/1075	3.1/2.7/2.6
	2/3/3	1513/1460/971	13/17/17	1418/1375/906	1.4/1.5/1.2
SmoLAgents†	3/3/3	1769/1907/2112	15/16/19	1202/1284/1385	2.3/2.5/2.9
	1/3/1	1317/1318/1405	15/14/12	1255/1265/1348	1.3/1.3/1.3
AgentFramework	0/2/1	618/2550/2023	4/16/18	218/1870/1312	1.2/3.0/2.8
	3/3/3	1652/1721/1466	16/14/15	1503/1644/1393	1.9/1.6/1.4
Qwen Agent	0/0/0	2374/2567/1599	16/18/15	1651/1751/946	3.0/3.3/2.4
	0/0/0	1462/1737/1622	16/14/13	1400/1676/1574	1.4/1.5/1.4
AgentScope†	0/0/1	2256/1973/1676	15/16/16	1513/1285/1056	3.0/2.7/2.4
	0/1/0	2059/2325/2169	12/14/14	2006/2253/2107	1.6/1.8/1.7
PraisonAI	1/2/1	1832/2341/2298	16/18/15	1163/1690/1537	2.6/2.9/3.0
	0/3/3	1922/1574/1367	14/16/15	1860/1506/1266	1.6/1.5/1.5
CAMEL†	2/1/2	2060/1734/1313	14/14/15	1372/1056/724	2.7/2.5/2.1
	2/3/3	1528/1391/1213	14/14/14	1470/1332/1159	1.4/1.3/1.2
FastAgent	0/1/0	2447/2603/2142	13/11/12	1655/1573/1454	3.1/3.7/2.7
	2/3/2	2287/2282/1574	10/12/11	2225/2213/1526	1.7/1.8/1.3
Solace	0/0/0	3192/2567/2182	12/11/12	2367/1934/1473	3.7/2.9/2.8
	0/1/0	1161/1054/1069	8/9/9	1110/1011/1027	1.0/1.0/0.9
Langroid	0/0/0	812/739/1044	5/4/6	465/437/641	1.2/1.1/1.5
	0/0/0	243/452/82	1/2/0	206/420/61	0.3/0.4/0.1
BeeAI	3/2/2	2388/1688/1545	14/21/27	1769/1064/962	2.8/2.4/2.3
	0/1/0	1586/1814/2282	12/29/23	1530/1754/2227	1.4/1.9/2.0
MetaGPT†	0/0/0	2506/2465/1561	13/16/12	1748/1810/1037	3.1/2.9/2.1
	0/0/0	1219/1554/1101	13/15/13	1163/1485/1044	1.2/1.5/1.1

Framework	Pass	In	Out	Cache	$
Swarms	2/0/1	2722/2092/1957	22/15/17	2113/1389/1371	3.1/2.8/2.5
	2/3/1	1911/2010/1301	15/13/18	1815/1932/1241	1.8/1.7/1.4
MCP Agent	2/1/2	2130/2391/1988	14/18/16	1510/1680/1310	2.6/3.0/2.7
	2/3/3	2107/1753/2220	15/14/14	2032/1665/2138	1.8/1.6/1.8
Guidance	1/1/1	2420/1846/1603	13/15/17	1593/1240/1024	3.2/2.4/2.3
	1/3/0	2442/2140/1555	12/13/13	2388/2071/1513	1.8/1.7/1.3
AtomicAgents	1/2/1	2085/2139/1609	22/14/17	1451/1429/950	2.7/2.8/2.4
	0/0/0	1096/1032/1446	18/21/15	1034/962/1393	1.3/1.3/1.3
Griptape	0/3/0	2522/2849/2010	15/16/16	1666/2057/1278	3.3/3.4/2.8
	1/3/2	1618/1496/1517	15/16/17	1548/1428/1456	1.5/1.5/1.4
Upsonic	4/2/2	1989/2053/1792	14/13/13	1340/1368/1302	2.6/2.7/2.2
	3/3/2	2340/1966/1937	14/14/15	2221/1898/1875	2.1/1.6/1.6
ControlFlow	0/0/0	2273/1783/1663	10/9/8	1538/1168/1020	2.9/2.4/2.3
	0/0/0	1469/1239/976	9/9/9	1417/1201/939	1.2/1.0/0.9
AgencySwarm	0/2/3	534/1737/1677	3/15/12	164/1148/1029	1.1/2.3/2.4
	3/3/3	1232/1144/1137	10/9/8	1146/1095/1080	1.3/1.0/1.0
Agently	0/1/0	2592/2977/1767	12/12/14	1750/2070/1113	3.3/3.7/2.5
	1/2/0	1224/1727/1926	22/19/18	1166/1662/1876	1.4/1.6/1.7
Lagent	0/0/0	2275/2171/1956	17/14/14	1517/1426/1211	3.0/2.9/2.8
	0/0/0	1121/1801/1654	18/17/13	1061/1722/1603	1.3/1.7/1.4
AgentUniverse	0/0/0	2470/2638/2696	15/19/16	1717/1768/1853	3.1/3.5/3.4
	0/0/0	1820/1699/1852	18/15/16	1731/1632/1791	1.8/1.5/1.6
EvoAgentX†	2/0/0	1449/2102/1761	9/14/10	860/1347/1101	2.1/2.9/2.4
	0/1/0	1426/1815/1087	9/14/7	1365/1736/1047	1.2/1.6/0.9
Nerve	1/3/3	2366/2184/3013	14/15/16	1778/1516/2119	2.7/2.8/3.7
	2/3/2	1845/1728/1377	15/16/14	1775/1652/1314	1.6/1.6/1.3
AgentSquad†	0/0/0	698/1993/2315	5/16/18	259/1282/1544	1.3/2.7/3.1
	0/0/0	1675/1361/1210	21/19/18	1574/1275/1122	1.8/1.5/1.4
MotleyCrew	3/4/3	2171/1673/1885	15/15/16	1549/1012/1233	2.7/2.4/2.6
	2/3/1	1875/1447/1892	14/17/13	1797/1363/1844	1.6/1.5/1.5
TaskflowAI	2/3/2	1686/1722/1860	16/14/17	1104/1132/1281	2.3/2.3/2.4
	0/1/0	1122/1028/1202	18/21/17	1060/965/1145	1.3/1.3/1.3
CouncilAI	0/0/1	2274/2271/2500	13/15/15	1517/1533/1876	3.0/3.0/2.9
	0/0/0	1762/6211/6874	21/58/251	1693/5140/6081	1.7/9.4/13.3
AgentFlow†	0/0/1	2394/2641/2226	16/15/20	1573/1725/1537	3.2/3.5/2.9
	0/0/0	1309/1719/1540	12/13/12	1248/1627/1481	1.2/1.6/1.3
AutoAgent†	1/2/2	2120/1975/2376	12/10/19	1386/1412/1451	2.8/2.4/3.4
	2/3/3	877/963/673	8/9/7	831/904/628	0.9/1.0/0.7
AgentLite†	2/2/3	2023/2123/1560	17/15/16	1247/1369/998	2.9/2.9/2.2
	1/2/2	1202/846/1072	35/52/33	1150/791/1017	1.7/2.0/1.6
Octotools	1/1/1	2134/2155/2147	15/17/20	1552/1404/1322	2.6/2.9/3.1
	1/1/2	1141/1048/930	17/23/22	1062/965/845	1.4/1.5/1.4
GPTSwarm†	0/1/0	1242/1584/1601	10/11/23	668/949/981	2.0/2.3/2.4
	0/0/1	1956/1797/1731	16/16/14	1901/1733/1672	1.6/1.6/1.5
Taskweaver†	0/0/0	2301/2308/2604	15/19/17	1675/1731/1915	2.8/2.7/3.1
	0/0/0	1516/1934/1957	17/16/16	1452/1852/1893	1.5/1.7/1.7
AutoAgents†	0/0/0	2570/2444/2243	15/15/14	1691/1772/1494	3.4/2.9/2.9
	0/0/0	1128/1413/1396	16/17/16	1066/1352/1332	1.3/1.4/1.4
OpenSage†	2/2/2	2618/2395/2170	12/13/13	1927/1742/1466	3.1/2.8/2.8
	1/1/0	1588/1675/1419	12/12/10	1529/1621/1374	1.4/1.4/1.2

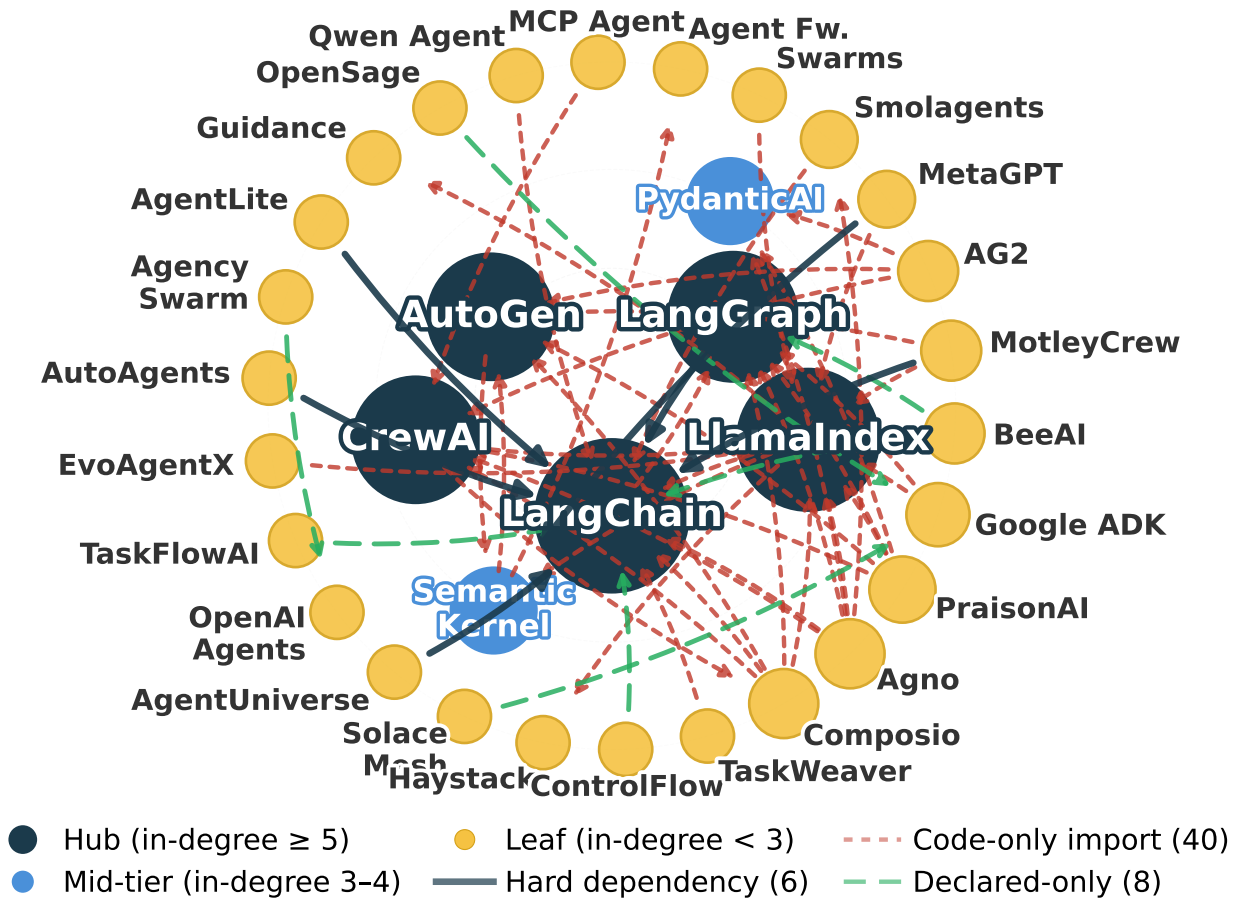

Figure 3. Inter-ADK dependency network.

A. Ecosystem Analysis

Beyond evaluating what ADK frameworks *do* on benchmarks, we examine how they *relate* to each other and to the broader developer community. We analyze inter-framework dependencies (§A.1), which reveal supply-chain risks and hidden coupling, and downstream adoption (§A.2), which quantifies real-world usage and market concentration.

A.1. Inter-ADK Dependencies

Many ADK frameworks reuse components from other ADKs, creating supply-chain concentration risks (Zimmermann et al., 2019; Decan et al., 2019) and hidden coupling invisible to package managers (Liu et al., 2022).

We extract dependency relationships from both package metadata (`pyproject.toml`, `setup.py`, `requirements.txt`) and source code imports across all 51 repositories. An edge from framework A to framework B is classified into three categories (test and example imports are marked *optional* and excluded from the core graph):

- **Hard**. Declared in package metadata *and* imported in source code. The dependency is fully transparent; `pip install` automatically resolves it. We find 6 such edges.
- **Code-only**. Imported in source code but *not* declared in metadata. Users who `pip install` the framework will encounter `ImportError` at runtime when a specific code path triggers the undeclared import. This is the dominant category, accounting for 40 of 54 non-optional edges (74%).
- **Declared-only**. Declared in metadata but *not* imported in source code. The package is installed but unused, likely a historical artifact or planned integration. We find 8 such edges.

Hub-and-spoke topology. The graph exhibits a hub-and-spoke structure. LangChain is the dominant hub (in-degree 14), with nearly half of all participating frameworks depending on it, five via hard dependencies (AgentLite, AgentUniverse, AutoAgents, LangGraph, MotleyCrew). LlamaIndex is the second hub (in-degree 10), but all its dependents use code-only imports, making the coupling less visible but equally fragile. On the consumer side, three frameworks have high out-degree: Agno (6), Composio (5), and PraisonAI (5), acting as integration layers that bridge multiple platforms. A total of 33 frameworks participate in the graph; the remaining 18 (35%) are fully independent.

Shared infrastructure. Beyond inter-ADK edges, frameworks converge on a small set of shared infrastructure packages. `pydantic` is required by 38/51 frameworks (74.5%), `openai` by 24 (47.1%), `requests` by 23 (45.1%), and `pyyaml` by 22 (43.1%). A breaking change in `pydantic` alone propagates to three-quarters of the ecosystem. Across 1,223 dependency specifications, only 35.0% use exact version pinning; 25.5% are completely unbounded. This loose pinning, combined with the high rate of undeclared code-only imports, creates a fragile dependency surface where upstream changes can silently break downstream frameworks.

Mutual dependencies. Three framework pairs have bidirectional edges: LangChain ↔ LangGraph (LangGraph hard-depends on `langchain_core`, while LangChain code-imports LangGraph), AutoGen ↔ Semantic Kernel (both code-import each other), and Composio ↔ CrewAI. These mutual dependencies create tightly coupled clusters where a breaking change in either framework can cascade to the other.

Finding. The ecosystem has fragile supply-chain concentration: 74% of inter-framework dependencies are undeclared, LangChain and LlamaIndex are hubs with in-degree 14 and 10 respectively, and `pydantic` breaking changes propagate to 75% of the ecosystem.

A.2. Downstream Adoption

We measure downstream adoption using four signals: GitHub stars, PyPI monthly downloads, GitHub dependent repositories, and GitHub code-search hits in dependency files (`requirements.txt`/`pyproject.toml`). Table 5 reports all four metrics for all 51 frameworks. Three patterns stand out:

Adoption concentration. Monthly downloads follow a steep power law. The top 5 frameworks by downloads (LangChain, Anthropic SDK, LangGraph, OpenAI Agents, PydanticAI) account for over 93% of all ADK downloads. LangChain alone receives 233M downloads/month, more than the next four combined. The top 14 frameworks each exceed 1M downloads/month; below that threshold, adoption drops sharply. At the tail, 3 frameworks are not published on PyPI at all, and 20 receive fewer than 100K downloads/month. By GitHub dependent repositories, concentration is equally pronounced: LangChain dominates (279.7K repos), followed by Anthropic SDK (50.0K) and LangGraph (38.5K), while the median framework has fewer than 200 dependents. This concentration implies that most developers building agents converge on a small set of frameworks, despite the ecosystem offering 51 alternatives.

Stars and downloads diverge. GitHub stars and PyPI downloads measure different things (community interest vs. production adoption), and the two metrics diverge for several frameworks. MetaGPT ranks 2nd by stars (67.5K) but receives only 42K downloads/month, suggesting research interest without corresponding production use; it is widely forked for academic experiments but rarely `pip install`ed as a dependency. Conversely, Anthropic SDK has just 3.3K stars but ranks 2nd by downloads (103.8M/month), reflecting its role as invisible infrastructure that other packages depend on rather than a project developers star for reference. AutoGen shows a similar pattern: 57.5K stars (3rd overall) but only 1.4M downloads/month (13th), likely because its original Microsoft repo accumulated stars before the AG2 fork split the user base. Newer vendor SDKs (Strands Agents, Google ADK) show the opposite: high download counts from CI pipelines and cloud deployments but few stars, as enterprise users rarely star vendor dependencies.

Code search reveals hidden adoption. GitHub's dependency graph has indexing lag for newer packages, making dependent-repository counts unreliable for frameworks released in the past 12 months. We complement this with GitHub code search over `requirements.txt` and `pyproject.toml` files across all public repositories. This reveals adoption invisible to traditional metrics: Google ADK shows 4.9K code-search hits despite zero indexed dependents, OpenAI Agents shows 4.2K, and Strands Agents shows 1.8K. Smolagents (2.0K hits, zero dependents) is another example: developers

Table 5. Downstream adoption metrics for all 51 ADK frameworks, sorted by monthly downloads. **Ver.**: version evaluated in this study. ★ = GitHub stars; DL/mo = PyPI monthly downloads; Repos = GitHub dependent repositories; CS = GitHub code-search hits in `requirements.txt` and `pyproject.toml`. "NA" indicates the framework is not published on PyPI or not yet indexed by GitHub. Frameworks marked with † originate from academic publications. Data collected April 2026.

Framework	Ver.	★	DL/mo	Repos	CS	Framework	Ver.	★	DL/mo	Repos	CS
Langchain	1.2.17	135.2K	233.1M	279.7K	32.0K	Swarms	11.0.1	6.6K	40.0K	429	1.1K
Anthropic Agent Sdk	0.98.0	3.3K	103.8M	50.0K	22.9K	Mcp Agent	0.2.6	8.3K	31.8K	243	540
Langgraph	1.1.10	30.6K	44.9M	38.5K	15.2K	Guidance	0.3.1	21.4K	28.3K	1.1K	2.3K
Openai Agents	0.15.1	25.3K	24.2M	NA	4.2K	Atomic Agents	2.7.5	5.8K	20.1K	91	159
Pydantic Ai	1.89.1	16.7K	23.4M	3.9K	4.6K	Griptape	1.10.0	2.5K	19.0K	198	148
Llama Index	0.12.52	49.0K	10.1M	24.0K	17.9K	Upsonic	0.76.3	7.8K	9.1K	65	36
Crewai	1.14.4	50.0K	6.7M	18.3K	7.2K	Controlflow	0.12.1	1.4K	7.0K	NA	42
Google Adk	1.32.0	19.3K	6.4M	NA	4.9K	Agency Swarm	1.9.6	4.2K	6.8K	353	212
Strands Agents	1.38.0	5.7K	5.1M	NA	1.8K	Agently	4.1.0.2	1.6K	3.5K	1	36
Semantic Kernel	1.41.3	27.8K	2.7M	2.6K	1.9K	Lagent	0.5.0rc3	2.2K	3.2K	174	2.7K
Agno	2.6.4	39.7K	1.7M	2.5K	3.5K	Agentuniverse	0.0.5	2.2K	2.3K	18	22
Ag2	0.12.2	4.5K	1.5M	774	1.6K	Evoagentx†	0.1.0	2.9K	1.7K	NA	9
Autogen	0.7.5	57.5K	1.4M	4.1K	2.0K	Nerve	1.8.0	1.3K	957	NA	NA
Composio	0.12.0	27.9K	1.0M	321	714	Agent Squad	1.0.2	7.6K	651	10	17
Haystack	2.28.0	25.0K	712.4K	1.4K	1.1K	Motleycrew	0.3.7	0.4K	626	NA	4
Smolagents†	1.24.0	26.9K	559.4K	NA	2.0K	Taskflowai	0.5.13	0.1K	191	6	7
Agent Framework	1.2.2	9.9K	436.7K	NA	1.6K	Council Ai	0.0.29	0.8K	181	12	13
Qwen Agent	0.0.34	16.2K	359.7K	163	270	Agentflow†	0.1.2	1.8K	146	NA	89
Agentscope†	1.0.19	24.4K	217.9K	64	253	Autoagent†	0.1.0	9.2K	123	NA	38
Praisonai	4.6.37	7.0K	183.7K	2	137	Agentlite†	0.1.12	0.6K	86	3	2
Camel Ai†	0.2.90	16.8K	134.9K	380	959	Octotools	0.2.0	1.4K	60	NA	3
Fast Agent	0.6.26	3.8K	105.3K	NA	139	Gptswarm†	0.1.0	1.0K	28	2	5
Solace Agent Mesh	1.23.1	3.3K	83.1K	8	76	Taskweaver†	0.0.12	6.2K	NA	NA	16
Langroid	0.61.1	4.0K	69.5K	68	514	Autoagents†	0.2	1.5K	NA	NA	12
Beeai	0.1.79	3.2K	49.5K	43	157	Opensage†	NA	75	NA	NA	5
Metagpt†	0.8.1	67.5K	42.0K	122	150						

include it in their requirements files, but GitHub has not yet indexed these repositories as dependents. This multi-source approach provides complementary coverage, as no single metric captures all frameworks.

Finding. Adoption follows extreme power-law concentration (top 5 = 93% of downloads). Stars and downloads diverge sharply, and no single metric captures all frameworks.

B. Details of Iteration Loop

This appendix details the three validation levels of the Validate-and-Repair pipeline (§3.2). Each level is progressively more expensive but catches deeper failures; the pipeline short-circuits on the first error and feeds a structured diagnostic hint back to the generator LLM before the next iteration.

B.1. Step 1: Static Analysis

Step 1 catches obvious errors without executing the agent or calling any LLM, making it both fast and free. It proceeds in three phases of increasing semantic depth.

Phase 1a: Compile and import. The validator dynamically loads `agent.py` via `importlib` and verifies that (1) the file parses without syntax errors, (2) a top-level `solve()` function exists, and (3) `solve()` accepts exactly two positional arguments (`problem_statement`, `workdir`). Before loading, environment variables (`ADK_BASE_URL`, `ADK_API_KEY`, `ADK_MODEL`, `ADK_TEMPERATURE`) are injected to mimic the execution environment. Agents declaring `async def solve()` are immediately rejected since the harness calls `solve()` synchronously. This phase catches the most basic failures (syntax errors, missing entry points, wrong signatures) before more expensive analysis begins.

Phase 1b: Framework usage. A design requirement is that each generated agent *genuinely uses* the target framework's orchestration API rather than bypassing it. The validator maintains a per-framework mapping from slug to expected import names (e.g., `crewai`: {`crewai`, `crewai_tools`}; `google-adk`: {`google.adk`, `google.genai`}) and confirms at least one appears in source. It then rejects two common LLM workarounds: *dummy references* (`_ = framework.__name__`) that technically import the framework without calling any API, and *raw API fallbacks* where `solve()` is implemented entirely via `httpx.Client()`, `requests.post()`, or `openai.OpenAI()` without any framework wrapping. These patterns are common in practice: LLMs often import the framework to satisfy the stated constraint while implementing the actual logic through familiar raw HTTP calls.

Phase 1c: Pattern-based static analysis. An AST and regex scanner checks 40+ patterns organized into seven categories:

- *URL double-path*: appending `"/v1"` to env var already containing `/v1` (causes 404).
- *SDK mismatch*: Anthropic SDK pointed at OpenAI-format proxy.
- *Missing timeouts*: HTTP calls without `timeout` (causes indefinite hang).
- *Process-killing calls*: `sys.exit()` terminates container before output.
- *Hardcoded URLs/keys*: secrets that should come from environment variables.
- *Bare `Any` type hints*: tool schemas lacking JSON Schema `type` (causes 400).
- *Broad exception handlers*: `except Exception:` without `raise` swallows diagnostics.

Additionally, a forbidden-dependency deny-list prevents importing other frameworks' packages, and benchmark-specific AST checks verify correct subclassing (e.g., τ^2-bench `HalfDuplexAgent` signatures).

Runtime diagnostics and repair. When any level fails, the validator pattern-matches the traceback against 70+ error signatures across 9 categories:

- *Module/import*: missing packages, renamed APIs between versions.
- *API configuration*: hardcoded URLs, wrong auth sources, SSL mismatches.
- *Async/event loop*: nested `asyncio.run()`, unclosed loops, unawaited coroutines.
- *Type/signature*: wrong argument counts, unexpected kwargs, removed attributes.
- *Container/filesystem*: writes outside `/tmp` or output directory.
- *Output parsing*: framework expects structured format but LLM returns plain text.
- *Response handling*: None values, exhausted retries, Pydantic validation failures.
- *Framework-specific*: renamed params, required base classes, breaking API changes.
- *Abstract method*: missing implementations of required interfaces.

Beyond error matching, the proxy records comprehensive telemetry throughout execution: total LLM calls, tool execution count, whether tools were included in requests and whether tool calls appeared in responses, the names of all tools registered with the LLM, maximum conversation length observed, and whether a `/v1/v1` double-path was detected in any request URL. This telemetry enables precise failure diagnosis without requiring access to the agent's internal state. After `solve()` returns, a final output validation step checks for non-empty results and correct return types, catching common mistakes like returning a coroutine object (forgot `await`), a raw framework object (forgot to extract `.text` or `.content`), an unconsumed async generator, or framework metadata dicts instead of the expected string output.

B.2. Step 2: Real LLM Smoke Test

While Step 1 catches structural problems, many failures only manifest when the agent interacts with a real LLM. Step 2 executes `solve()` with DeepSeek-V4-Flash through a dedicated token-recording proxy, catching model-specific incompatibilities: unsupported parameters (`reasoning_content`, `developer` role), response format mismatches, authentication errors, and framework-internal routing issues that depend on actual model output.

Architecture. The test spawns an HTTP reverse proxy on a random port that transparently forwards requests to the DeepSeek API while recording per-call metrics (input/output/cached tokens, latency). The agent's `ADK_BASE_URL` and `ADK_API_KEY` point to this proxy. Runtime compatibility patches (identical to those applied during actual benchmark execution) are loaded before the agent module, ensuring full environment fidelity. This design means that if an agent passes Step 2, it will encounter the exact same runtime environment during the final benchmarking phase.

Benchmark-specific prompts and environments. Each benchmark receives a minimal test prompt designed to exercise core capabilities without expensive multi-step reasoning: MCP-Atlas forces tool calling ("What is the git status of the repository?"), SWE-bench forces code reasoning ("Fix the TypeError"), Terminal-Bench forces shell execution ("Create hello.txt"), and τ^2-bench runs a 3-round multi-turn conversation with alternating user/assistant/tool messages. The validator also prepares a minimal environment matching what the real benchmark provides: a stub MCP server for MCP-Atlas, a domain configuration file for τ^2-bench, and placeholder source files for SWE-bench.

Pass criteria and failure diagnostics. The test passes if the proxy records $\geq$1 LLM call (confirming the agent routes through the proxy), at least 2 calls for MCP-Atlas (tool discovery + usage), and the agent returns a non-empty string not matching known bad patterns (`<coroutine object`, `object at 0x`). A 45-second timeout with recorded proxy calls is also a pass, since iterative agent loops commonly exceed this budget. On failure, the validator distinguishes four scenarios based on proxy telemetry: 0 calls with timeout (agent bypasses proxy or hangs pre-API), $\geq$1 calls with timeout (agent loop lacks termination condition), 401 errors (wrong authentication source), and 400 errors (malformed request format). Each scenario produces a targeted diagnostic hint that tells the generator LLM exactly what to fix.

B.3. Step 3: Real Benchmark Task

Step 3 provides the strongest validation by running one task from the target benchmark's *official* task set using the full benchmark adapter and execution environment. This catches failures invisible to Steps 1–2 that only emerge under realistic conditions: complex multi-step prompts with large context windows, tool schemas containing $\sim$20 tools with detailed JSON parameter definitions, multi-turn conversation management across dozens of messages, and real environment interactions with Docker containers, git repositories, and file systems.

Task selection and execution. Each benchmark has a designated smoke test task chosen for fast execution while exercising the full agent pipeline: a multi-tool git/file query for MCP-Atlas, `django__django-10097` (focused bug fix) for SWE-bench, a git branch operation for Terminal-Bench, and a customer service interaction for τ^2-bench. The validator launches the benchmark adapter in a daemon thread with the same configuration as the final benchmarking phase: identical runtime patches, proxy, and tool interfaces. Rather than blocking on completion, it polls the proxy's `/status` endpoint every 2 seconds for up to 60 seconds, enabling early termination once the agent demonstrates sufficient capability (e.g., reaching the minimum call count with tool executions).

Pass criteria. Success requires benchmark-specific minimum LLM calls ($\geq$2 for MCP-Atlas, τ^2-bench, and Terminal-Bench; $\geq$1 for SWE-bench), non-empty output where applicable (SWE-bench must produce a patch), and tool execution (MCP-Atlas must dispatch $\geq$1 tool call). When tools are registered but never executed, the validator performs root-cause analysis: tools never sent to the LLM indicates a configuration error, tool calls returned but not dispatched indicates a broken execution loop, and the LLM choosing not to call tools is treated as a prompt issue (warning only). Note that *correctness* of the output is not evaluated at this stage; that responsibility belongs to the full benchmarking phase (§3.3). Step 3 validates only that the agent can survive a complete execution cycle without crashing.

C. Full Execution Performance

Table 6 reports per-framework resolution rate, token usage, and cost for all 51 frameworks across the four benchmarks under the normal information condition, referenced in §4.2.

Table 6. Full execution performance for all 51 ADK frameworks (normal condition). Each cell packs the four benchmarks in order SWE-bench / τ^2-bench / MCP-Atlas / TerminalBench. First row: agent generated by GPT-5.4 (Codex). Second row: by Opus-4.6 (Claude Code). Both execute on GPT-5.4 Nano. "–" = agent failed to build/execute or no data recorded.

Framework	Resolve (%)	In Tok	Out Tok	Cache Tok	Cost ($)
LangChain	44/4/58/40 30/76/–/48	910.5K/1.4M/54.0K/471.8K 436.9K/55.0K/–/374.4K	11.4K/15.1K/2.5K/17.0K 10.0K/9.3K/–/19.9K	580.8K/317.7K/17.4K/376.8K 269.3K/35.6K/–/259.5K	0.13/0.26/0.01/0.07 0.07/0.01/–/0.06
Anthropic SDK	0/66/0/0 –/60/–/–	973/76.7K/67/1.9K –/63.5K/–/–	1.2K/9.2K/252/1.7K –/10.6K/–/–	148/51.8K/0/0 –/40.1K/–/–	0.00/0.02/0.00/0.00 –/0.02/–/–
LangGraph	0/0/–/16 26/66/46/36	4.0K/0/–/104.4K 310.3K/80.1K/45.7K/552.1K	2.4K/0/–/6.0K 8.2K/10.4K/2.6K/19.0K	0/0/–/79.1K 177.1K/48.1K/20.1K/397.8K	0.00/0.00/–/0.02 0.05/0.02/0.01/0.09
OpenAI Agents	6/74/–/10 44/70/–/54	59.2K/69.5K/–/30.5K 1.6M/57.9K/–/1.1M	2.5K/10.2K/–/9.6K 34.6K/9.5K/–/42.4K	20.2K/44.6K/–/12.2K 1.3M/39.2K/–/850.9K	0.01/0.02/–/0.01 0.23/0.02/–/0.17
PydanticAI	0/42/–/2 30/66/–/14	6.2K/41.2K/–/2.8K 920.1K/50.4K/–/143.4K	10.7K/10.0K/–/2.4K 24.5K/8.1K/–/8.4K	2.4K/23.9K/–/594 502.6K/30.5K/–/96.5K	0.01/0.01/–/0.00 0.15/0.01/–/0.03
LlamaIndex	–/0/0/– –/68/–/–	–/1.8K/0/– –/61.1K/–/–	–/822/0/– –/9.4K/–/–	–/0/0/– –/39.7K/–/–	–/0.00/0.00/– –/0.02/–/–
CrewAI	30/44/–/0 46/0/30/56	760.5K/42.4K/–/202.5K 2.7M/1.8K/23.1K/803.4K	20.5K/12.0K/–/17.5K 17.3K/799/2.2K/27.5K	555.7K/22.1K/–/129.1K 2.4M/0/12.2K/577.6K	0.11/0.02/–/0.04 0.32/0.00/0.01/0.12
Google ADK	44/–/–/0 –/80/42/52	1.6M/–/–/723 –/70.9K/12.9K/480.4K	37.6K/–/–/8.1K –/10.9K/1.1K/15.4K	1.1M/–/–/0 –/47.9K/3.2K/367.4K	0.23/–/–/0.01 –/0.02/0.00/0.07
Strands	–/74/–/10 16/34/–/24	–/55.5K/–/13.6K 787.5K/76.0K/–/518.6K	–/9.0K/–/3.0K 9.1K/8.9K/–/19.2K	–/36.4K/–/3.6K 501.7K/52.4K/–/407.0K	–/0.01/–/0.00 0.11/0.02/–/0.08
SemanticKernel	–/40/2/– –/0/–/6	–/64.5K/8.0K/– –/25.3K/–/16.2K	–/9.8K/3.8K/– –/5.5K/–/6.0K	–/33.9K/3.5K/– –/18.6K/–/1.1K	–/0.02/0.00/– –/0.01/–/0.01
Agno	34/0/68/0 52/60/–/54	1.4M/12.3K/38.3K/5.1K 1.3M/48.7K/–/763.2K	13.0K/3.3K/3.1K/1.1K 11.9K/8.3K/–/20.8K	1.1M/7.2K/11.5K/0 960.5K/31.7K/–/558.8K	0.18/0.00/0.01/0.00 0.18/0.01/–/0.11
AG2	0/60/0/6 48/54/62/60	541.3K/48.2K/17.3K/563.3K 1.2M/46.8K/60.2K/0	16.4K/7.3K/572/14.5K 17.1K/7.1K/2.9K/0	508.0K/34.5K/15.0K/496.6K 892.3K/33.5K/20.8K/0	0.07/0.01/0.00/0.07 0.17/0.01/0.01/0.00
AutoGen	34/14/–/0 40/70/46/48	1.7M/18.5K/–/131.0K 762.4K/55.9K/45.7K/456.1K	18.6K/5.4K/–/14.5K 7.4K/7.9K/2.3K/15.7K	1.3M/12.2K/–/88.2K 530.1K/35.8K/18.5K/304.8K	0.22/0.01/–/0.03 0.11/0.01/0.01/0.07
Composio	12/–/–/4 58/70/40/46	59.9K/–/–/305.7K 1.0M/72.3K/22.6K/514.2K	2.6K/–/–/9.4K 12.9K/10.4K/1.7K/17.3K	35.4K/–/–/275.2K 671.3K/42.4K/3.4K/352.7K	0.01/–/–/0.04 0.15/0.02/0.01/0.08
Haystack	0/66/8/0 36/74/44/52	41.5K/43.3K/1.0K/3.4K 702.3K/65.9K/76.1K/641.4K	14.6K/7.2K/728/4.3K 11.9K/8.7K/2.3K/22.1K	14.0K/26.6K/0/26 467.1K/47.4K/22.2K/447.9K	0.02/0.01/0.00/0.00 0.10/0.02/0.01/0.10
SmoLAgents	4/66/–/– 28/0/–/32	32.8K/55.1K/–/– 911.0K/1.8K/–/766.1K	9.7K/7.6K/–/– 13.6K/814/–/35.7K	16.3K/39.2K/–/– 590.3K/0/–/626.7K	0.01/0.01/–/– 0.13/0.00/–/0.12
AgentFramework	–/2/68/14 80/38/32/40	–/5.8K/108.2K/244.5K 2.0M/87.4K/36.4K/356.3K	–/1.8K/3.9K/9.4K 13.7K/5.9K/1.3K/24.5K	–/3.6K/45.3K/0 0/65.2K/0/213.4K	–/0.00/0.02/0.06 0.40/0.02/0.01/0.07
Qwen Agent	0/44/–/– –/–/–/–	591/40.2K/–/– –/–/–/–	85/9.4K/–/– –/–/–/–	0/20.2K/–/– –/–/–/–	0.00/0.01/–/– –/–/–/–
AgentScope	0/–/–/0 56/0/–/–	3.1K/–/–/17.3K 643.0K/14.2K/–/–	3.0K/–/–/10.2K 17.1K/4.1K/–/–	92/–/–/2.2K 537.2K/11.1K/–/–	0.00/–/–/0.01 0.09/0.01/–/–
PraisonAI	0/14/–/32 16/76/–/2	2.3K/58.5K/–/274.4K 98.0K/64.9K/–/11.6K	8.5K/4.7K/–/13.8K 3.5K/9.7K/–/1.7K	328/39.3K/–/202.3K 16.6K/40.2K/–/5.3K	0.01/0.01/–/0.05 0.02/0.02/–/0.00
CAMEL	50/22/–/4 52/64/–/60	1.1M/45.4K/–/144.6K 1.1M/50.0K/–/417.3K	17.0K/5.8K/–/8.2K 15.1K/6.7K/–/20.5K	843.9K/30.8K/–/86.3K 731.1K/29.9K/–/262.1K	0.15/0.01/–/0.03 0.15/0.01/–/0.07
FastAgent	–/0/–/– 0/72/2/40	–/0/–/– 469.8K/61.9K/388/245.3K	–/0/–/– 7.4K/10.6K/63/5.2K	–/0/–/– 335.3K/40.8K/0/223.5K	–/0.00/–/– 0.07/0.02/0.00/0.03
Solace	–/68/–/– –/–/–/32	–/56.3K/–/– –/–/–/1.1M	–/9.2K/–/– –/–/–/39.6K	–/36.9K/–/– –/–/–/874.5K	–/0.01/–/– –/–/–/0.17
Langroid	–/–/–/– –/–/–/–	–/–/–/– –/–/–/–	–/–/–/– –/–/–/–	–/–/–/– –/–/–/–	–/–/–/– –/–/–/–
BeeAI	4/–/–/0 –/0/2/–	5.2K/–/–/4.0K –/1.8K/5.5K/–	17.4K/–/–/23.3K –/811/1.4K/–	26/–/–/1.1K –/0/338/–	0.01/–/–/0.02 –/0.00/0.00/–
MetaGPT	0/68/–/0 –/0/–/–	7.5K/46.0K/–/3.2K –/24.6K/–/–	6.9K/7.2K/–/3.8K –/5.3K/–/–	620/28.1K/–/599 –/18.0K/–/–	0.01/0.01/–/0.00 –/0.01/–/–
Swarms	0/34/–/– 0/64/–/28	22.5K/56.8K/–/– 14.8K/55.8K/–/0	6.9K/8.1K/–/– 635/8.6K/–/0	4.0K/42.2K/–/– 3.7K/38.2K/–/0	0.01/0.01/–/– 0.00/0.01/–/0.00
MCP Agent	10/68/–/0 18/62/–/42	46.0K/64.8K/–/3.8K 742.6K/50.9K/–/575.1K	3.8K/9.0K/–/15.7K 17.1K/7.8K/–/15.3K	11.8K/45.6K/–/788 383.5K/34.0K/–/425.1K	0.01/0.02/–/0.01 0.12/0.01/–/0.08
Guidance	0/72/–/0 38/68/–/52	4.4K/55.1K/–/2.1K 1.2M/55.2K/–/854.2K	385/8.3K/–/616 12.2K/8.7K/–/28.7K	0/33.4K/–/0 818.1K/38.8K/–/623.8K	0.00/0.01/–/0.00 0.17/0.01/–/0.13

continued on next page

Table 6 – continued from previous page

Framework	Resolve (%)	In Tok	Out Tok	Cache Tok	Cost ($)
AtomicAgents	0/76/–/0 64/–/–/48	10.5K/53.8K/–/151.4K 676.7K/–/–/333.6K	3.1K/8.1K/–/14.5K 23.2K/–/–/25.1K	1.6K/34.3K/–/100.8K 586.9K/–/–/221.8K	0.00/0.01/–/0.03 0.10/–/–/0.06
Griptape	–/66/2/0 –/–/–/–	–/62.2K/1.5K/89.8K –/–/–/–	–/10.0K/299/14.0K –/–/–/–	–/41.4K/23/58.8K –/–/–/–	–/0.02/0.00/0.02 –/–/–/–
Upsonic	24/0/–/– 32/–/–/–	1.6M/14.4K/–/– 1.0M/–/–/–	18.2K/3.6K/–/– 14.6K/–/–/–	1.1M/11.0K/–/– 756.3K/–/–/–	0.22/0.00/–/– 0.14/–/–/–
ControlFlow	–/–/–/– –/–/–/–	–/–/–/– –/–/–/–	–/–/–/– –/–/–/–	–/–/–/– –/–/–/–	–/–/–/– –/–/–/–
AgencySwarm	42/46/–/0 52/–/34/16	306.4K/16.1K/–/484.3K 210.8K/–/18.3K/80.9K	3.5K/1.9K/–/12.0K 722/–/268/1.5K	0/1.1K/–/0 165.5K/–/5.4K/64.1K	0.06/0.00/–/0.11 0.03/–/0.00/0.01
Agently	0/–/0/– –/–/0/0	4.3K/–/2.8K/– –/–/4.8K/1.4K	14.8K/–/3.8K/– –/–/2.4K/6.8K	701/–/0/– –/–/0/0	0.01/–/0.00/– –/–/0.00/0.01
Lagent	–/0/–/0 –/0/–/0	–/0/–/1.7K –/27.2K/–/3.0K	–/0/–/1.5K –/6.1K/–/8.1K	–/0/–/72 –/19.0K/–/445	–/0.00/–/0.00 –/0.01/–/0.01
AgentUniverse	–/–/18/– 58/66/38/56	–/–/1.0K/– 1.8M/64.2K/22.1K/962.7K	–/–/967/– 32.0K/10.3K/2.2K/46.6K	–/–/0/– 1.3M/40.5K/7.6K/717.1K	–/–/0.00/– 0.26/0.02/0.01/0.16
EvoAgentX	0/0/–/– –/68/–/14	1.3K/0/–/– –/52.6K/–/75.3K	403/0/–/– –/9.3K/–/9.1K	41/0/–/– –/30.5K/–/62.1K	0.00/0.00/–/– –/0.01/–/0.02
Nerve	26/2/–/0 10/68/–/24	418.3K/6.1K/–/230.5K 933.7K/52.1K/–/300.6K	10.4K/1.5K/–/9.6K 11.2K/7.9K/–/18.6K	271.4K/4.4K/–/177.3K 612.0K/30.4K/–/201.8K	0.06/0.00/–/0.04 0.13/0.01/–/0.05
AgentSquad	0/0/–/4 34/52/48/48	7.6K/1.8K/–/147.6K 383.0K/51.3K/54.0K/686.7K	2.6K/800/–/8.0K 10.2K/6.6K/3.3K/25.8K	2.6K/0/–/106.0K 234.3K/35.0K/19.4K/483.7K	0.00/0.00/–/0.03 0.06/0.01/0.01/0.11
MotleyCrew	38/68/–/0 0/–/–/40	509.8K/85.0K/–/27.2K 242.2K/–/–/190.5K	22.5K/10.0K/–/3.4K 9.0K/–/–/14.8K	338.0K/60.0K/–/9.7K 164.5K/–/–/105.8K	0.09/0.02/–/0.01 0.04/–/–/0.04
TaskflowAI	–/66/–/– –/76/–/–	–/54.3K/–/– –/77.3K/–/–	–/8.6K/–/– –/10.4K/–/–	–/34.3K/–/– –/52.9K/–/–	–/0.01/–/– –/0.02/–/–
CouncilAI	–/–/–/0 –/72/–/46	–/–/–/1.6K –/61.0K/–/822.5K	–/–/–/568 –/10.0K/–/22.5K	–/–/–/0 –/38.7K/–/640.5K	–/–/–/0.00 –/0.02/–/0.12
AgentFlow	0/–/–/– –/68/–/–	0/–/–/– –/52.2K/–/–	0/–/–/– –/9.0K/–/–	0/–/–/– –/32.7K/–/–	0.00/–/–/– –/0.01/–/–
AutoAgent	0/44/–/– –/72/42/44	7.4K/69.4K/–/– –/56.8K/21.8K/671.8K	430/9.5K/–/– –/9.1K/1.5K/34.5K	980/51.7K/–/– –/34.5K/4.7K/449.1K	0.00/0.02/–/– –/0.02/0.01/0.12
AgentLite	0/22/–/0 0/66/14/–	2.2K/72.4K/–/383 43.7K/59.5K/5.5K/–	10.8K/6.6K/–/360 8.8K/9.3K/4.6K/–	0/45.8K/–/0 33.5K/42.9K/1.1K/–	0.01/0.02/–/0.00 0.01/0.02/0.00/–
Octotools	0/60/–/8 0/62/–/–	0/82.4K/–/75.2K 3.9K/53.8K/–/–	0/10.5K/–/16.9K 587/9.0K/–/–	0/58.6K/–/52.7K 2.5K/30.2K/–/–	0.00/0.02/–/0.02 0.00/0.01/–/–
GPTSwarm	0/–/64/– 54/–/58/–	0/–/84.4K/– 770.8K/–/29.8K/–	0/–/3.1K/– 11.8K/–/2.9K/–	0/–/24.1K/– 514.4K/–/9.5K/–	0.00/–/0.02/– 0.11/–/0.01/–
Taskweaver	–/32/–/– –/68/–/–	–/49.9K/–/– –/56.3K/–/–	–/7.4K/–/– –/8.9K/–/–	–/37.9K/–/– –/39.4K/–/–	–/0.01/–/– –/0.01/–/–
AutoAgents	–/0/–/– 0/66/–/–	–/0/–/– 93.3K/53.3K/–/–	–/0/–/– 22.0K/8.5K/–/–	–/0/–/– 11.8K/34.1K/–/–	–/0.00/–/– 0.04/0.01/–/–
OpenSage	0/68/–/40 18/–/–/28	386.3K/70.5K/–/326.7K 1.1M/–/–/275.8K	16.9K/10.0K/–/11.9K 10.8K/–/–/11.4K	216.0K/48.6K/–/249.9K 764.0K/–/–/191.1K	0.07/0.02/–/0.05 0.15/–/–/0.05