AgentProf: Semantic Profiling for AI Agents

Abstract

AI agents increasingly orchestrate long-running, multi-step activities involving thousands to millions of interactions with users, tools, and system resources. To improve quality, safety, and cost efficiency of AI agent, developers need to determine where failures concentrate, which workflows trigger unsafe effects, and which task categories consume the most budget. To answer similar questions in traditional systems software, profiling attributes resource consumption to responsible entities, such as code paths, by aggregation, rather than single-execution debugging or tracing. Yet existing agent observability tools are mainly for debugging and tracing instead of profiling. Agent behavior differs fundamentally from code execution. The responsible entities are semantic categories such as task intent and workflow phase, not code paths, and agent trajectories lack the stable identifiers and structural hierarchies that make attribution possible. We propose a *semantic operation stack model* that adapts profiling to agent trajectories. Uniform *operations* represent all agent activities, and *operation stacks* replace the runtime call stack, enabling hierarchical attribution at different levels of granularity from the same data. AgentProf implements this model as a profiler with pluggable algorithms for *intent attribution* and *stack construction*, compiling local agent trajectories into pprof-compatible profiles. On real trajectories and public datasets, our evaluation shows that semantic profiling effectively attributes cost and locates problems across agent trajectories.

1 Introduction

AI agents [1, 31, 33] orchestrate multi-step activities that span two layers, high-level intent (prompts, LLM calls, tool invocations) and low-level system effects (process spawns, file and network I/O). As agents evolve from short, scripted workflows into complex systems that run for hours and days, with a large number of interactions each, production teams accumulate many such trajectories over weeks or months.

To improve agent quality, safety, and cost efficiency, developers need to analyze operational behavior across trajectories and interactions. Which categories of tasks consume the most token budget? Where do failures concentrate across workflows? Which behavioral patterns trigger unsafe system effects? Answering these questions requires analysis and evaluation across many trajectories [22, 23], a task becoming costly for manual inspection or per-trajectory LLM evaluation [36] at scale. In traditional systems software, profiling answers these questions by attributing resource consumption to responsible entities by aggregation, as distinct from single-execution debugging and tracing.

Yet existing agent tools support debugging and tracing but not profiling (§2). Some [2, 14, 15, 25] organize events into per-execution span trees, effective for diagnosing a single run but unable to aggregate by semantic category. Others [7, 13] cluster inputs or extract structured signals, but characterize *input distributions* ("30% of users ask code questions") rather than *resource attribution* ("review tasks consume 40% of the token budget"), because tags at the request level do not propagate to downstream tool calls and system effects.

Adapting profiling to agent trajectories is challenging (§2). In traditional software, the responsible entities are code paths. Profiling is straightforward because function names are stable and runtime call nesting provides the attribution hierarchy (§2). Agent behavior differs fundamentally. To answer the questions above, the responsible entities need to be semantic categories such as task intent and tool operation, not code paths. Agent trajectories lack the two properties that make traditional profiling possible. Prompts are natural language, so two prompts expressing the same intent share no common string. Agent events have no runtime call-stack hierarchy because prompts, tool calls, and system effects are not nested by execution the way function calls produce stack frames: a sequence of prompts can be a task or multiple tasks, a task can be part of a bigger task.

Despite these differences, the fundamental profiling methodology transfers to agent trajectories: project resources onto responsible entities, assign stable tags, and attribute hierarchically. We propose a *semantic operation stack model* with two components. *Operations* are uniform records with string fields and additive measures that represent all agent activities (prompts, LLM calls, tool invocations, and system effects) without type-specific objects. *Operation stacks* provide hierarchical attribution, replacing the runtime call stack. Choosing different fields changes the attribution granularity: the same operations can be attributed by task category, by execution phase, by session, or by action type (Figure 1).

We implement this model in AgentProf, an offline profiler that compiles local agent trajectories into pprof-compatible profiles (Figure 2). Two pluggable algorithm frameworks enrich the model. *Intent attribution* derives stable, low-cardinality tags from natural-language fields via regex rules, a local LLM tagger, or unsupervised clustering. This gives agents the stable tags that traditional profiling gets for free from function names. *Stack construction* builds the attribution hierarchy, either from a user-specified field list or by automatically detecting boundaries between distinct work phases. On 325 real Codex and Claude trajectories (183,714 system observations), semantic profiling separates over 90% of system-effect cost that per-session views cannot attribute to task categories. On four public datasets (34,539 operations) where the correct annotations are known but hidden from the profiler, ranked groups locate real problems at 9.4% inspection work, with 45% fewer groups than per-session grouping. Derived tags agree with ground truth on 7 of 9 held-out datasets.

This paper makes three contributions:

(1) **Semantic operation stack model.** We identify the missing profiling layer in agent observability and propose a model of uniform operations and query-time operation stacks (§2–§3).
(2) **System: AgentProf.** A profiler that implements this model with pluggable intent attribution and stack construction, producing pprof-compatible output (§4).
(3) **Evaluation.** We evaluate profiler output against hidden ground-truth trajectory annotations, analogous to injecting known bottlenecks in CPU profiler evaluation, on 325 real trajectories and 4 public datasets (§5).

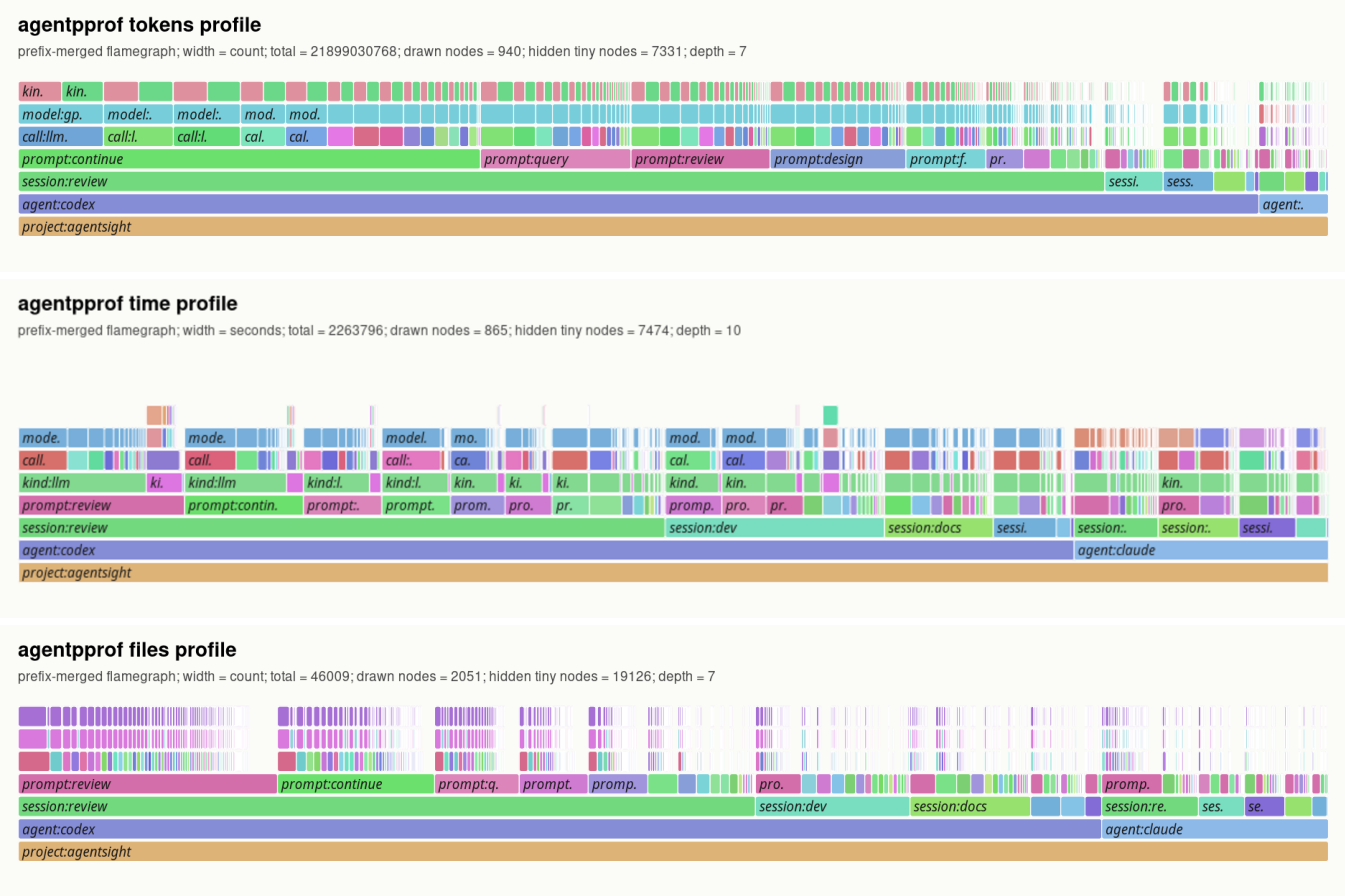


Figure 1: Semantic flame graphs of 325 real agent trajectories under three views. *Top (tokens):* width represents token count. *Middle (time):* width represents wall-clock duration. *Bottom (files):* width represents file-operation count. All three are produced from the same operations by changing only the stack fields and weight function.

2 Background and Motivation

2.1 LLMs and AI Agents

A typical agent trajectory interleaves two layers of activity. At the intent layer, the agent receives a user prompt, issues LLM calls, and invokes tools (code execution, web search, file editing). Each tool invocation triggers system effects at the lower layer: process spawns, file reads and writes, and network requests. A single trajectory may contain hundreds of such intent-effect cycles, and a production team accumulates many trajectories over time.

2.2 System Profiling

Profiling attributes resource consumption to responsible entities by aggregation, as distinct from single-execution debugging and tracing. In traditional software, the profiling pipeline works as follows: a profiler periodically samples execution state, attaches each sample to the current call stack, and merges samples with identical stacks. Flame graphs [12] and pprof [11] call graphs visualize the result, where width or node size represents aggregate cost. Perfetto [10] generalizes this with SQL-based analysis and derived events. These tools support flexible aggregation, but they all assume stable function names and a runtime call stack. Pprof's `tagroot/tagleaf` options can add label-derived pseudo-frames, but only on top of an existing execution stack that agent trajectories do not have.

2.3 Challenges for Agent Profiling

Existing tools [2, 14, 15, 25] record agent events as per-execution span trees, effective for debugging a single run. AgentSight [37] bridges intent and system-effect layers by correlating eBPF-captured system events with intercepted LLM traffic, but the resulting recordings still lack aggregate profiling.

Adapting profiling to agent trajectories faces two core challenges. The first is that the responsible entities differ. In traditional software, profiling attributes cost to code paths, but in agent trajectories the relevant entities are semantic categories such as task intent and workflow phase. Existing tools [3, 25] do not capture these categories because prompts are natural language: two prompts and tool invocations expressing the same intent may be different, so the profiler cannot aggregate them the way it aggregates identical function names. The second challenge is that agent trajectories

have no runtime hierarchy for attribution. In CPU profiling, the call stack determines attribution at every level: `malloc`'s cost belongs to `parse`, to `process`, and to `main` simultaneously. A sequence of prompts may constitute one task or several, and a task may be part of a larger workflow, but no runtime mechanism marks these boundaries or determines at which granularity to attribute cost. We quantify this in RQ1 (§5).

2.4 Design Requirements for Agent Profiling

Traditional profiling relies on three mechanisms that the call stack provides automatically. Agent profiling needs all three but must achieve them without a call stack:

R1: Cross-layer resource projection. In traditional profiling, each sample is taken inside a call context, so resource consumption is automatically attributed to the code path that caused it. Agent activities span two layers: intent (prompts, LLM calls) and system effects (file I/O, process spawns). The profiler must connect system-layer resource consumption to intent-layer semantic categories.

R2: Stable tags for folding. In traditional profiling, function names are stable identifiers that enable folding identical stacks. Agent prompts are natural language, so the profiler must derive short, stable, repeatable tags before folding.

R3: Hierarchical attribution. In traditional profiling, the runtime call stack provides the attribution hierarchy: function calls nest, and folding identical stacks produces the tree that profiles visualize. Agent events are not nested by execution, so the profiler must construct attribution hierarchies from the data.

3 Design

The three requirements (cross-layer resource projection, stable tags, and hierarchical attribution) drive the design. Operations (§3.1) satisfy R1 by representing intent and system-effect activities in a single record with tag inheritance, so intent-layer tags propagate to the system effects they trigger. *Intent attribution* (§3.2) satisfies R2 by deriving stable tags from natural-language fields. *Operation stacks* (§3.1) satisfy R3 by replacing the runtime call stack with query-time field projections that attribute resources at multiple levels of granularity, so the same data supports multiple views without rebuilding the input. The profiling pipeline has four stages: (1) parse raw trajectories into operations, (2) enrich operations via intent attribution or mapping rules, (3) project each operation onto a stack frame sequence chosen at query time, and (4) fold operations with identical stacks, summing their weights.

3.1 Semantic Operation Stack Model

Because agent activities span both the intent and system-effect layers described in §2, a profiler should not distinguish between them in the data representation. AgentProf therefore represents every activity as a single abstraction, the `operation`, a weighted record with string fields and additive measures. Each operation carries string fields (`project`, `agent`, `session`, `prompt_tag`, `kind`, `model`, `path`, `domain`, `status`, …) and additive measures (token count, duration, event count). A prompt, an LLM call, a tool invocation, a file read, a GUI click, and a process event are all operations with the same schema, distinguished only by which fields carry values.

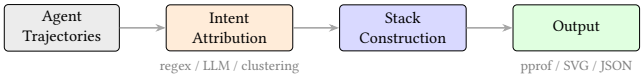


Figure 2: Data-flow and algorithm pipeline. Local agent histories enter through `agent-session` parsing; public datasets enter as operation JSONL. Both paths produce uniform operations. Intent attribution, stack construction, and folding are query-time algorithms applied identically to both sources.

An `operation stack` provides hierarchical attribution for agent trajectories, replacing the runtime call stack. In CPU profiling, the call stack determines which code paths share responsibility for a resource cost. An operation stack serves the same role: an ordered list of fields $[f_1, f_2, \ldots, f_k]$ projects each operation o to a frame sequence $\langle o.f_1, o.f_2, \ldots, o.f_k \rangle$, and operations with identical sequences are merged, summing their weights. Unlike a call stack, the fields are chosen at query time, not determined by execution. The same operations can therefore be attributed as a flat task summary, a per-session tree, a semantic phase/action hierarchy, or a trajectory using the dataset's own annotations, and changing the fields changes the attribution without changing the data. Sessions and spans are optional operation fields, not fixed hierarchy levels, so the same data supports both debugging views (include `session`) and aggregate profiling views (exclude it). A view is a triple (φ, σ, w): a predicate φ selects which operations participate, a stack function $\sigma = [f_1, \ldots, f_k]$ maps each operation to its frame sequence, and a weight function w assigns a nonnegative measure. Four built-in views cover common questions: `tokens` (LLM calls weighted by token count), `time` (all timed operations weighted by duration), `files` (file operations weighted by event count), and `network` (network operations weighted by event count).

3.2 Algorithms

Intent attribution. Intent attribution maps natural-language prompts to short, stable, repeatable tags, giving agent trajectories the stable identifiers that traditional profiling gets for free from function names (§2). The framework supports three pluggable backends. Regex rules are the production default: rules have the form `KIND:TAG=REGEX`, are evaluated in order (first match wins), and an AI Agent can iterates the regex with AgentProf until the unmatched rate drops below 5%, typically in 5–10 rounds. A local LLM tagger (e.g., a quantized 3B-parameter model via llama.cpp) generates a single-word tag per prompt with grammar-constrained decoding (forcing output to match a predefined token grammar) and caching, so AI Agent can help iterate the prompt as part of the configuration. Unsupervised clustering via TF-IDF and K-Means discovers natural prompt groups without predefined rules, feeding rule authoring. For structured trajectories where action types and quality annotations are explicit fields, mapping rules derive new fields from existing ones (e.g., `phase:navigate=type:(click|scroll|key)`).

Stack construction. Stack construction builds attribution hierarchies from available fields. When users supply an ordered list of

fields, the projection is direct. When no field list is supplied, AgentProf induces operation stacks automatically. It scores each pair of adjacent operations for shifts in prompt content (via TF-IDF cosine similarity), changes in structured field values, and consistency of the resulting groups, then places boundaries where the combined score is highest. The induced stacks have variable depth, and a configurable depth cap controls the granularity. This mode produces coarser groups than hand-specified stacks but requires no domain knowledge, making it useful for initial exploration (§5).

4 Implementation

AgentProf is an offline, post-hoc profiler implemented as a Rust CLI (~9.8K LOC, Figure 2). It reads Codex and Claude Code local JSONL history files and AgentSight [37] recordings for system effects, and reconstructs prompts, LLM calls, tool invocations, and their triggered file and network effects. Output formats include pprof protobuf (compatible with `go tool pprof -http`), folded-stack text (compatible with `flamegraph.pl`), standalone SVG flame graphs, and JSON summaries.

5 Evaluation

We use two classes of data. *Real agent trajectories*: 325 readable local trajectories (198 Codex, 50 Claude, 77 Claude subagent) from a multi-month development project, containing 183,714 system observations. *Public annotated trajectories*: 15 families of real agent or human executions, totaling 47,590 operations through mapping rules. These cover web agents (WebLINX [21], Mind2Web [8], AgentTrek [32], WebShop [34]), API agents (API-Bank [16], ToolBench [26], tau-bench [35]), coding agents (SWE-agent [33]), and mobile/GUI agents (AndroidCtrl [17], GUI-Odyssey [20], Scale-CUA [19]). Four families with ground-truth quality annotations (AgentRewardBench [22], SATraj-OS [6], AgentNet [28], OSWorld-Human [30]) provide hidden annotations for RQ2.

5.1 RQ1: Does Semantic Profiling Improve Resource Attribution?

We test whether semantic tags improve resource attribution beyond what traditional tag-free tools provide. We run a semantic-axis ablation on 325 real Codex/Claude trajectories (183,714 system observations). We progressively add tag axes (no tags, session only, prompt only, both) and measure the mixed-weight percentage, the fraction of system-effect cost in groups that mix observations from multiple task categories. A lower mixed-weight percentage means the profiler correctly separates effects by task category. The residual percentage measures weight in groups where the majority category accounts for less than 90% of the group, capturing partial mixing.

Figure 3 shows the result. Without semantic tags, nearly all system-effect cost falls into mixed groups: a flat `cargo test` counter shows 2,903 executions but cannot distinguish whether they came from `review`, `debug`, `refactor`, or `test` prompts. The prompt tag is the decisive axis: it drops both mixed weight and residual by more than half and nearly doubles the number of unique stacks, because it separates observations that share the same system call but originate from different task intents. The session tag alone barely helps because trajectories typically contain multiple intent phases.

A tag-permutation test (1,000 shuffles, $p = 0.001$) confirms the separation is not random. Semantic-tag quality is validated in RQ3. This confirms that traditional tag-free tools cannot separate system effects by task intent; the prompt-level semantic tag is necessary for meaningful resource attribution.

Beyond tag separation, we test whether choosing different field selections for the operation stack produces views that flat grouping cannot. We take the same 13,265 operations from the four families with ground-truth annotations and fold them with five different field selections, counting how many distinct groups each produces. The same operations fold into 9 dataset groups, 57 phase groups, 226 semantic groups, 455 action groups, or 3,757 per-session groups by changing only the stack fields. A flat GROUP BY on a single tag cannot produce these views because different stack depths answer different questions (budget by task vs. duplication by action vs. per-session detail). The same model also covers all 15 public trajectory families without type-specific objects. This shows that operation stacks provide a multi-resolution view that flat grouping cannot: one model answers questions at dataset, phase, action, and session granularity by changing only the field selection.

A profiler that supports only one weight function (e.g., token count) might miss bottlenecks visible under a different metric. We test whether different weight functions produce redundant or distinct rankings on the same trajectories. The `time` and `tokens` views of the same 325 trajectories share only 7/10 top prompt categories (Spearman $\rho = 0.623$). `network`-tagged prompts rank 8th by time (I/O wait) but 93rd by tokens. The rankings are not redundant: a single weight function would miss bottlenecks that are prominent under a different metric, confirming that multi-view profiling is necessary.

Finally, we test whether the profiler can produce useful groups without any user-specified fields. We run the automatic stack-induction mode (§3.1) on the same six tasks used in RQ2 and compare the resulting group count and localization quality against flat summaries and per-session grouping. Induced stacks produce a median of 12 groups (versus 285 for per-session and 1 for flat), with 4/6 tasks generating variable-depth stacks. The induced stacks achieve lower average precision (AP, the area under the precision-recall curve when groups are ranked by problem density) than hand-specified stacks (median 0.276 vs 0.312) because they use only visible fields, but they reduce inspection work to 65.3% versus flat summaries without requiring any user configuration. Automatic induction therefore provides a useful zero-configuration baseline: it loses some precision versus hand-specified stacks but still substantially outperforms flat summaries.

Figure 1 shows the resulting flame graphs for the `tokens` and `files` views, illustrating that changing only the stack fields and weight function produces different aggregates from the same operations.

5.2 RQ2: Does Profiler Output Correspond to Real Problems?

RQ2 tests whether the profiler's top-ranked groups actually contain real failures, safety violations, or wasted effort. We use public benchmark datasets that provide ground-truth quality annotations (failure, safety violation, redundant step, human-task boundary),

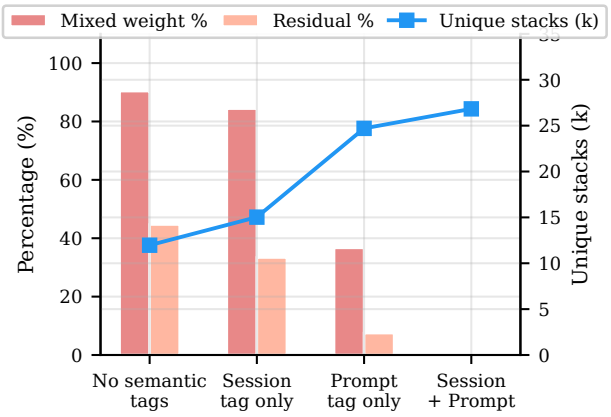


Figure 3: RQ1: Semantic-axis ablation on 183,714 system observations from 325 real trajectories. Bars show mixed weight and residual percentages (left axis); the line shows unique stacks (right axis). Adding prompt tags reduces mixed weight from 90.4% to 36.7% while increasing unique stacks from 12k to 25k. Session tags alone barely help (84.4%).

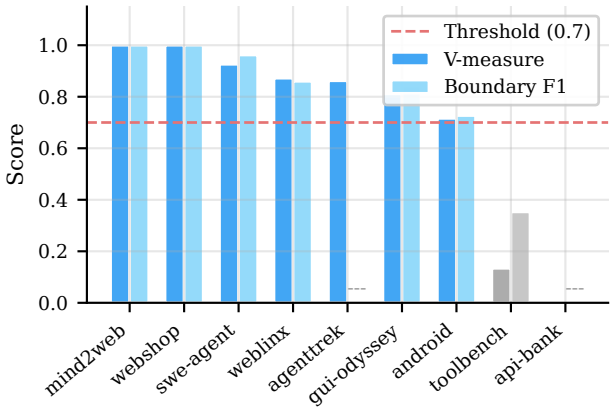


Figure 4: RQ3: Leave-dataset-out mapping quality on 13,265 operations across 9 held-out datasets. Dark bars: V-measure (cluster agreement between mapping-derived `phase` and native `action` annotations). Light bars: boundary F1 (precision/recall of phase transitions). "—": boundary evaluation inapplicable. Dashed line marks 0.7; seven of nine datasets exceed it on both metrics.

hide these annotations from the profiler, and check whether the top-ranked groups recover them, analogous to injecting known bottlenecks in a CPU profiler evaluation.

The benchmark uses six tasks across four datasets (AgentRewardBench looping and side-effect, SATraj-OS unsafe operation, AgentNet incorrect and redundant step, and OSWorld-Human group start), totaling 34,539 operations and 3,699 positives. We compare seven grouping methods and rank groups by the fraction of hidden positives they contain. Flat summary places all operations in one group. Per-session grouping groups by session boundary. Native hierarchy uses the dataset's own structure (e.g., episode and step levels). Raw-action grouping groups by the unprocessed action-type field. Operation-stack profiling applies the model from §3. Two oracle baselines add the hidden annotation fields as upper bounds.

Table 1 summarizes the results. Flat summaries recover all positives but require inspecting everything (Work@5 = 1.0). Per-session grouping finds a first positive quickly (WTFP = 0.004) but fragments the workload into 285 groups with only 2.3% top-five recall. Operation-stack profiling inspects 9.4% of the work in the top five groups, recovers 39.0% of positives within 30% of the inspection budget, and reduces groups to 157.5.

These three methods (which do not use hidden annotations) illustrate a tradeoff between inspection effort and coverage. Flat summaries require full inspection but trivially find everything. Per-session grouping minimizes the work to find a first positive but scatters the remaining positives across hundreds of groups. Operation-stack profiling balances the two: it concentrates positives into fewer, higher-quality groups, trading the fast-first-hit property for broader coverage at lower total effort. The comparison also serves as evidence for the semantic operation stack model (§3). Per-session grouping and native hierarchy are alternative abstractions that fix the hierarchy at session or dataset boundaries. Operation stacks subsume both as special cases (include `session` in the stack to reproduce per-session grouping; include `dataset` to reproduce the native hierarchy). On localization, operation stacks improve top-five recall over per-session on 5/6 tasks, reduce median group count by 45%, and save 90% of inspection work versus flat summaries. Per-session grouping retains lower work-to-first-positive on 4/6 tasks, confirming that profilers and debuggers are complementary.

Statistical validation supports these comparisons. Bootstrap (10,000 resamples) and tag-permutation (2,000 trials) confirm that the AP differences are stable and exceed the 95% confidence threshold on all six tasks.

Stack depth affects localization quality. Sweeping the induced-stack depth cap from 1 to 5 shows that AP peaks at depth 3 (median 0.287), while different tasks achieve their best at different depths (2–5), so depth should be tuned per objective. Modifying stack fields, mapping rules, or ranking criteria and re-running on the same operations improves AP on 5/6 tasks (median improvement 0.038), confirming that the profiler exposes useful tuning axes.

5.3 RQ3: How Accurate Are the Tags?

RQ1 and RQ2 assume that semantic tags (the tags derived by intent attribution or mapping rules) are useful. RQ3 tests whether these semantic tags are accurate by comparing them against ground-truth annotations from the benchmark datasets.

Public benchmark datasets provide native action annotations that serve as ground truth for mapping quality. We run a leave-dataset-out evaluation. For each of 9 datasets, we train mapping rules on the other 8 and measure V-measure (a clustering-agreement score ranging from 0 to 1) between the mapping-derived `phase` field and the dataset's native `action` annotations, plus boundary F1 (precision and recall of phase-transition points) between mapped phase boundaries and native action boundaries.

Table 1: RQ2 hidden-annotation localization (medians across six tasks). AP = average precision. R@5 = recall in top 5 groups. F1@5 = F1 score in top 5 groups. Work@5 = fraction of total work in top 5 groups. R@30% = recall at 30% inspection budget. WTFP = work-to-first-positive (fraction inspected before finding one positive). Hidden = 1 means the method uses hidden annotations (upper bound). "—" = group count varies by task.

Method	Hidden	AP	R@5	F1@5	Work@5	R@30%	WTFP	Groups
Flat (all-in-one)	0	0.168	1.000	0.287	1.000	0.000	1.000	1.0
Per-session grouping	0	0.348	0.023	0.043	0.016	0.356	0.004	285.0
Native hierarchy	0	0.357	0.867	0.282	0.854	0.338	0.058	—
Raw-action grouping	0	0.274	0.244	0.220	0.151	0.333	0.037	—
Operation-stack profiling	0	0.312	0.188	0.198	0.094	0.390	0.038	157.5
Op-stack + oracle fields	1	0.599	0.300	0.403	0.044	0.564	0.012	—
Oracle (hidden annotations)	1	1.000	0.670	0.797	0.115	1.000	0.038	—

Figure 4 shows the results. V-measure quantifies whether the mapping groups operations into the same clusters as the ground-truth annotations; boundary F1 measures whether phase-transition points align. Seven of nine datasets exceed 0.7 on both metrics, meaning rules trained on other families generalize: they assign operations to the correct semantic phase and detect transitions accurately. The two low-scoring datasets (toolbench, api-bank) use atypical action vocabularies absent from the training families, confirming that the mapping is pattern-based and degrades predictably when vocabulary shifts.

As an upper-bound comparison, supervised boundary predictors trained on ground-truth phase transitions beat the rule-based mapping on 4/5 tested datasets (e.g., F1 0.77 vs 0.71 on OSWorld-Human), confirming that mapping rules produce useful tags but leave room for learned intent discovery.

5.4 RQ4: What Is the Profiling Cost?

AgentProf is an offline, post-hoc profiler, so it adds zero overhead to the agent's runtime. The profiling cost is the time and resources spent after the session ends. Each profiling run chooses a stack-field list, a predicate, a weight function, and ranking criteria, then executes the full pipeline (parse, enrich, construct stacks, fold) on the input operations. Across 76 configurations, median end-to-end time is 1.6 s (p95 2.8 s, max 4.3 s). The slowest runs process views with over 600 unique stacks, while runs that vary only ranking criteria or stack depth all complete within 1.6 s.

Intent attribution adds a one-time cost for new prompts. Of 118,021 tag requests on 325 real trajectories, 70% are cache hits (zero cost), and the remaining 35,136 new calls complete at p95 latency of 31 ms per tag using a local 3B-parameter model. Grammar-constrained decoding ensures 100% syntax validity (2,700/2,700 test tags). Because tags are cached, re-profiling the same trajectories with different stack fields or predicates incurs no additional LLM calls.

6 Related Work

Agent observability tools [2–4, 9, 14, 15, 25, 29] focus on single-execution debugging. Datadog LLM Observability [7] and Laminar [13] cluster inputs or extract signals but characterize input distributions, not resource attribution. AgentRx [5], TELBench [27], AgentAtlas [23], TrajAD [18], and AgentFixer [24] perform per-trajectory fault localization. AgentProf complements all of these with category-level aggregate profiling across trajectories.

7 Conclusion

Agent observability needs profiling, not only debugging. AgentProf shows that the semantic operation stack model, with pluggable intent attribution and stack construction, brings hierarchical attribution to agent trajectories. Scoring profiler output against hidden trajectory annotations confirms that semantic profiling separates over 90% of system-effect cost that per-session views leave mixed, localizes annotated problems at 9.4% inspection work versus flat summaries, and produces tags that agree with ground truth on 7/9 held-out datasets. Per-session grouping remains faster to a first positive on 4/6 tasks, confirming that profilers and debuggers are complementary, and AgentProf supports both by including or excluding session fields in the stack query. Multi-project evaluation and tracing-ecosystem integration are the most impactful next steps.

References

[1] Anthropic. 2025. Claude Code: An Agentic Coding Tool. https://code.claude.com/docs/en/overview.

[2] Arize AI. 2024. Arize Phoenix. https://arize.com/docs/phoenix.

[3] Arize AI. 2024. OpenInference Specification. https://arize-ai.github.io/openinference/spec/.

[4] Krishna Chaitanya Balusu. 2026. AgentTelemetry: A Fault Detection Benchmark and Toolkit for LLM Agent Observability. In *Proceedings of the 3rd ACM International Conference on AI-Powered Software (AIware '26).* doi:10.1145/3805760.3814931

[5] Shraddha Barke, Arnav Goyal, Alind Khare, Avaljot Singh, Suman Nath, and Chetan Bansal. 2026. AgentRx: Diagnosing AI Agent Failures from Execution Trajectories. arXiv preprint arXiv:2602.02475. doi:10.48550/arXiv.2602.02475

[6] Xinquan Chen, Zhenyun Yin, Shan He, Bin Huang, Shanzhe Lei, et al. 2026. Safactory: A Scalable Agentic Infrastructure for Training Trustworthy Autonomous Intelligence. arXiv preprint arXiv:2605.06230. doi:10.48550/arXiv.2605.06230

[7] Datadog. 2025. LLM Observability. https://docs.datadoghq.com/llm_observability/.

[8] Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. 2023. Mind2Web: Towards a Generalist Agent for the Web. In *Advances in Neural Information Processing Systems 36 (NeurIPS), Datasets and Benchmarks Track.* doi:10.5555/3666122.3667342

[9] Liming Dong, Qinghua Lu, and Liming Zhu. 2024. AgentOps: Enabling Observability of LLM Agents. arXiv preprint arXiv:2411.05285. doi:10.48550/arXiv.2411.05285

[10] Google. 2024. Perfetto Trace Processor. https://perfetto.dev/docs/analysis/trace-processor.

[11] Google. 2024. pprof. https://github.com/google/pprof.

[12] Brendan Gregg. 2011. Flame Graphs. https://www.brendangregg.com/flamegraphs.html.

[13] Laminar. 2025. Signals: Structured Event Extraction from Traces. https://docs.lmnr.ai/signals/introduction.

[14] LangChain. 2024. LangSmith Observability. https://docs.langchain.com/langsmith/observability.

[15] Langfuse. 2024. Langfuse Observability. https://langfuse.com/docs/observability/overview.

[16] Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. API-Bank: A Comprehensive Benchmark for Tool-Augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 3102–3116. doi:10.18653/v1/2023.emnlp-main.187

[17] Wei Li, William E. Bishop, Alice Li, Chris Rawles, Folawiyo Campbell-Ajala, Divya Tyamagundlu, and Oriana Riva. 2024. On the Effects of Data Scale on UI Control Agents. arXiv preprint arXiv:2406.03679. doi:10.48550/arXiv.2406.03679

[18] Yibing Liu, Chong Zhang, Zhongyi Han, Hansong Liu, Yong Wang, Yang Yu, Xiaoyan Wang, and Yilong Yin. 2026. TrajAD: Trajectory Anomaly Detection for Trustworthy LLM Agents. arXiv preprint arXiv:2602.06443. doi:10.48550/arXiv.2602.06443

[19] Zhaoyang Liu, Jingjing Xie, Zichen Ding, Zehao Li, Bowen Yang, et al. 2025. ScaleCUA: Scaling Open-Source Computer Use Agents with Cross-Platform Data. arXiv preprint arXiv:2509.15221. doi:10.48550/arXiv.2509.15221

[20] Quanfeng Lu, Wenqi Shao, Zitao Liu, Fanqing Meng, Boxuan Li, Botian Chen, Siyuan Huang, Kaipeng Zhang, Yu Qiao, and Ping Luo. 2025. GUIOdyssey: A Comprehensive Dataset for Cross-App GUI Navigation on Mobile Devices. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. https://github.com/OpenGVLab/GUI-Odyssey

[21] Xing Han Lù, Zdeněk Kasner, and Siva Reddy. 2024. WebLINX: Real-World Website Navigation with Multi-Turn Dialogue. arXiv preprint arXiv:2402.05930. doi:10.48550/arXiv.2402.05930

[22] Xing Han Lù, Amirhossein Kazemnejad, Nicholas Meade, Arkil Patel, Dongchan Shin, Alejandra Zambrano, Karolina Stańczak, Peter Shaw, Christopher J. Pal, and Siva Reddy. 2025. AgentRewardBench: Evaluating Automatic Evaluations of Web Agent Trajectories. arXiv preprint arXiv:2504.08942. doi:10.48550/arXiv.2504.08942

[23] Parsa Mazaheri and Kasra Mazaheri. 2026. AgentAtlas: Beyond Outcome Leaderboards for LLM Agents. arXiv preprint arXiv:2605.20530. doi:10.48550/arXiv.2605.20530

[24] Hadar Mulian, Sergey Zeltyn, Ido Levy, Liane Galanti, Avi Yaeli, and Segev Shlomov. 2026. AgentFixer: From Failure Detection to Fix Recommendations in LLM Agentic Systems. In *Proceedings of the 1st International Workshop on Agentic Engineering (AGENT '26, co-located with ICSE)*. doi:10.48550/arXiv.2603.29848

[25] OpenTelemetry. 2024. OpenTelemetry GenAI Semantic Conventions. https://github.com/open-telemetry/semantic-conventions-genai.

[26] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerber, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. In *Proceedings of the 12th International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=dHng2O0Jjr

[27] Jiaming Wang, Ziteng Feng, Jiangtao Wu, Ruihao Li, Qianqian Xie, Yuxiang Ren, He Zhu, Xueming Han, Fanyu Meng, Junlan Feng, and Jiaheng Liu. 2026. Where Do Deep-Research Agents Go Wrong? Span-Level Error Localization in Agent Trajectories. arXiv preprint arXiv:2606.02060. doi:10.48550/arXiv.2606.02060

[28] Xinyuan Wang, Bowen Wang, Dunjie Lu, Junlin Yang, Tianbao Xie, et al. 2025. AgentNet. https://huggingface.co/datasets/xlangai/AgentNet.

[29] Yiqi Wang, Jiaqi Zhang, Taotao Cai, Zirui Liu, Qingqiang Sun, Zequn Sun, Zhangkai Wu, Manqing Dong, Mingkai Zheng, Xuefei Yin, and Yanming Zhu. 2026. From Agent Traces to Trust: A Survey of Evidence Tracing and Execution Provenance in LLM Agents. arXiv preprint arXiv:2606.04990. doi:10.48550/arXiv.2606.04990

[30] WukLab. 2025. OSWorld-Human. https://github.com/WukLab/osworld-human.

[31] Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shi, Joel Tong, Kai-Wei Lu, Vedant Garg, Yitao Wang, Ziniu Dai, Fangyu Li, Yonatan Bisk, and Tao Yu. 2024. OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments. In *Advances in Neural Information Processing Systems 37 (NeurIPS), Datasets and Benchmarks Track*.

[32] Yiheng Xu, Dunjie Lu, Zhennan Shen, Junli Wang, Zekun Wang, Yuchen Mao, Caiming Xiong, and Tao Yu. 2024. AgentTrek: Agent Trajectory Synthesis via Guiding Replay with Web Tutorials. arXiv preprint arXiv:2412.09605. doi:10.48550/arXiv.2412.09605

[33] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems 37 (NeurIPS)*. doi:10.5555/3737916.3739517

[34] Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2022. WebShop: Towards Scalable Real-World Web Interaction with Grounded Language Agents. In *Advances in Neural Information Processing Systems 35 (NeurIPS)*. doi:10.5555/3600270.3601778

[35] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains. arXiv preprint arXiv:2406.12045. doi:10.48550/arXiv.2406.12045

[36] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems 36 (NeurIPS), Datasets and Benchmarks Track*. https://arxiv.org/abs/2306.05685

[37] Yusheng Zheng, Yanpeng Hu, Tong Yu, and Andi Quinn. 2025. AgentSight: System-Level Observability for AI Agents Using eBPF. In *Proceedings of the 4th Workshop on Practical Adoption Challenges of ML for Systems (PACMI '25, co-located with SOSP)*. doi:10.1145/3766882.3767169