AgentPPprof: Semantic Profiler for AI Agents

Yusheng Zheng[1,2] Chaokun Chang[3] Yu Mao[4] Tianyuan Wu[3] Yuxi Huang[2]
Tao Ma[5] Wenan Mao[5] Shuyi Cheng[5] Andi Quinn[1] Wei Wang[3]

[1]UC Santa Cruz [2]Eunomia Labs [3]HKUST [4]Independent Researcher [5]Alibaba Group

Abstract

AI agents increasingly orchestrate long-running activities with users, tools, and system resources for days and weeks. To improve agent quality, safety, and cost efficiency, developers need to determine where failures happen, what triggers unsafe effects, and which tasks consume the most budget, then optimize those tasks. In systems software, profiling answers similar questions by aggregating resource consumption and attributing it to responsible code paths to identify hotspots. Yet existing agent observability tools focus on per-execution debugging and tracing rather than cross-run, long term profiling, making these questions difficult to answer at scale. Agent observability needs profiling, not only debugging, but profiling agents is challenging: the responsible entities are task intent like *diagnose authentication*, *compare branches* rather than code paths, and lack stable identifiers for aggregation. We propose a *semantic operation stack model* that adapts profiling to agent trajectories. Uniform *operations* represent all activities, and *operation stacks* replace the runtime call stack, enabling hierarchical attribution at different granularities. We observe that an agent's task occupies a contiguous span and decomposes into subtasks, so we introduce *recursive operation segmentation*, which recursively splits trajectories at task boundaries. AgentPPprof is a profiler that aggregates agent trajectories into pprof-compatible profiles, enabling flame graph visualization and analysis. AgentPPprof reaches 0.764 B^3 F1 against human annotations on CodeTraceBench. On three problem-localization benchmarks, the profile raises MAP by up to 56%, demonstrating that it effectively attributes resources, locates problems, and helps optimize token cost at practical profiling cost. AgentPPprof is available at https://github.com/eunomia-bpf/agentsight.

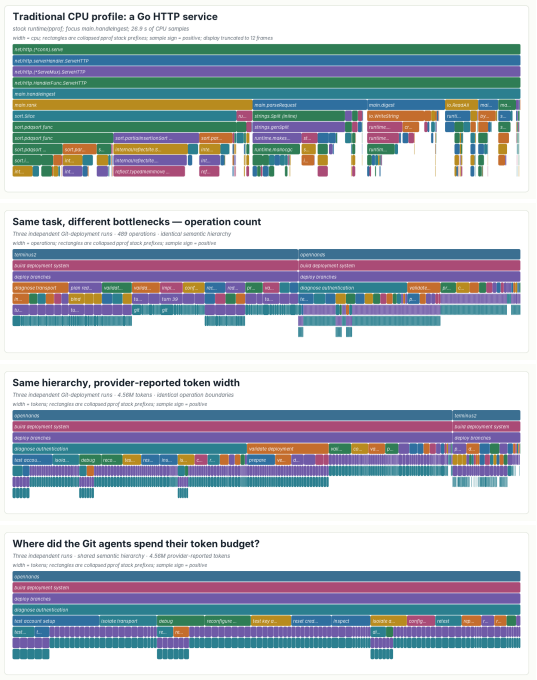

Figure 1: One renderer, four standard pprof profiles. *(a)* CPU time in a Go HTTP service. *(b, c)* Three agent executions under one fixed hierarchy: width is operation count (b) or tokens (c). *(d)* Token profile focused on `diagnose authentication`.

1 Introduction

AI agents [1, 44, 45] orchestrate multi-step activities spanning 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 or days, 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 is the first step toward optimizing high-cost tasks and fixing failure-prone workflows. Answering them requires analysis and evaluation across many trajectories [24, 28], which is increasingly costly for manual inspection or per-trajectory LLM evaluation [46] at scale. In traditional systems software, profiling answers these questions by aggregating resource consumption and attributing it to responsible entities, distinct from per-execution debugging and tracing.

Yet existing agent tools support debugging and tracing but not profiling. Some [2, 17, 19, 32] organize events into per-execution span trees, effective for diagnosing a single run but unable to aggregate by task intent or workflow phase. Others [9, 16] 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 request-level tags do not propagate to downstream tool calls and system effects.

Adapting profiling to agent trajectories is challenging: in traditional software, the responsible entities are code paths, and profiling is straightforward because function names are stable and runtime call nesting provides the attribution hierarchy. Agent behavior differs. To answer the questions above, the responsible entities must be task intent and workflow phase, 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, and a task can be part of a bigger task.

Despite these differences, the fundamental profiling methodology transfers to agent trajectories by projecting resources onto responsible entities, assigning stable identifiers, and attributing hierarchically. We propose a *semantic operation stack model* with two components. *Operations* are uniform records with string fields and additive measures representing all agent activities (prompts, LLM calls, tool invocations, system effects). *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, and one fixed hierarchy replays under any additive measure (Figure 1).

We implement this model in AgentPPProf, an offline profiler that compiles local agent trajectories into pprof-compatible profiles (Figure 2). One algorithm, *recursive operation segmentation*, supplies the missing identifiers. Labeling every prompt individually would be prohibitively expensive, but task structure is recoverable at lower cost: a task occupies a contiguous span of a trajectory and decomposes into contiguous subtasks, so recovering it requires only finding the transitions between them. The algorithm finds where responsibility changes, names the resulting intervals, and yields short names starting with a verb (like `diagnose authentication`) that fold across sessions. The interface is implementation-neutral.

We evaluate whether AgentPPProf effectively attributes resources, locates problems, and recovers human-recognizable task structure at practical cost. Across all 405 CodeTraceBench [21] trajectories, recursive segmentation agrees with human stage annotations at 0.764 B^3 F1 [4], versus 0.663 for a statistical baseline and 0.541 for raw actions. On three complete localization benchmarks, the profile improves ranking over each benchmark's own diagnostic, raising MAP by 0.031, 0.107, and 0.117; as a navigation aid, it reduces content to inspect by 12 percentage points at equal ranking quality. A profile-derived repair reduces agent tokens by 19% without degrading task quality, and annotation cost is reduced by 20%.

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 (Background and Design).
(2) **System: AgentPPProf.** A profiler that implements this model, producing pprof-compatible output.
(3) **Evaluation.** We evaluate profiler output against independent ground-truth annotations that stay hidden from the profiler, on eight public benchmarks and three real trajectory datasets.

2 Background and Motivation

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 cycles between intent and system effects.

System Profiling. In traditional software, a profiler periodically samples execution state, attaches each sample to the current call stack, and merges samples with identical stacks. Flame graphs [14] and pprof [13] call graphs visualize the result, where width or node size represents aggregate cost. Engineers use these profiles to find where CPU or memory budget is spent, then optimize the responsible code paths to reduce cost. 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.

A Motivating Case. Three independent agent attempts at one real Git-deployment task all failed to deliver the requested password-authenticated endpoint. The developer's question is *what went wrong, and is it the same problem across all three runs?* Reading three transcripts totaling 489 operations and 4.6M tokens would take hours. Figure 1 places a conventional CPU profile beside the semantic profile of those three runs. Both are standard pprof profiles drawn by one renderer and read by identical rules, where width is aggregate cost, a parent contains its children, and identical stacks recorded at different moments fold into one frame; what differs is where the frames come from: a call stack supplies `net/http.(*conn).serve` at every sample for free, but nothing in an agent trajectory supplies `diagnose authentication`, because the three runs express that intent through different prompts and shell commands.

The profile answers in one view (Figure 1): all three runs share a `diagnose authentication` responsibility that consumes 46% of their combined token budget but only 21% of operation count. Expanding the view shows all three executions verifying substitute transport paths without ever establishing the target endpoint: the agents kept retrying SSH variants instead of fixing the actual credential issue, and operation count alone hides this because authentication commands are cheap to issue but expensive to verify. The profile points to where to intervene: early credential validation or bounded retry depth in `diagnose authentication` could cut token cost for similar tasks.

Challenges for Agent Profiling. Adapting profiling to agent trajectories faces two core challenges. First, the responsible entities differ: cost must attach to task intent and workflow phase rather than code paths, and existing tools [3, 32] do not capture these entities because prompts are natural language—two prompts expressing the same intent may differ, so the profiler cannot aggregate them the way it aggregates identical function names. Second, agent trajectories have no runtime hierarchy for attribution: in CPU profiling the call stack fixes attribution at every level (`malloc`'s cost belongs to `parse`, to `process`, and to `main` simultaneously), but a sequence of prompts may constitute one task or several, a task may be part of a larger workflow, and no runtime mechanism marks these boundaries or determines the attribution granularity.

3 Design

Agent profiling needs the three mechanisms that call stacks provide automatically, but must achieve them without a call stack: **R1** (cross-layer projection): connect system-layer consumption to intent-level categories; **R2** (stable identifiers): derive short, repeatable identifiers from natural-language prompts before folding; **R3** (hierarchical attribution): construct attribution hierarchies from data, not execution nesting.

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). Operations satisfy R1 by representing intent and system-effect activities in a single record with tag inheritance, so intent tags propagate to the system effects they trigger via AgentSight [47]. *Recursive operation segmentation* satisfies R2 and R3 together, deriving short, stable interval names from trajectory content and nesting them into the attribution hierarchy that replaces the runtime call stack. *Operation stacks* then attribute resources at every level of that hierarchy at query time, so the same data supports multiple views without rebuilding the input. The profiling pipeline has four stages: (1) parse raw trajectories into operations, (2) segment trajectories into named nested intervals (or apply 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

This section defines operations and operation stacks, the data structures that satisfy R1 (cross-layer projection) and enable R3 (hierarchical attribution). R2 (stable identifiers) requires deriving identifiers from trajectory content and is addressed in Section 3.2.

Because agent activities span both intent and system-effect layers, a profiler should not distinguish them in the data representation: AgentPProf 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. Only source content, timestamps, and additive measures are required; explicit tool/event IDs, sessions, and spans are optional. AgentPProf uses recorded identifiers verbatim when present, falls back to the lifetime-overlap rule above when they are missing, and leaves ambiguous or conflicting links unassigned (fail closed) rather than inferring a parent; only the semantic interval names are inferred, derived from content by segmentation.

An `operation stack` provides hierarchical attribution for agent trajectories, replacing the runtime call stack. 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. Changing the fields changes the attribution without changing the data, so the same operations can be attributed by task, by session, or by action type. Sessions and spans are optional fields, so the same data supports both debugging (include `session`) and aggregate profiling (exclude it).

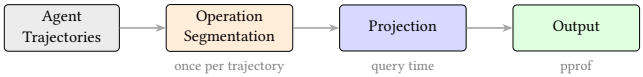

Figure 2: Data-flow pipeline. Local histories and public datasets both produce uniform operations; segmentation runs once, projection and folding at query time.

3.2 Recursive Operation Segmentation

Operation stacks can project any field an operation already carries, but agent trajectories lack the two fields that make traditional profiling work: stable task identifiers and a nesting hierarchy. Recursive operation segmentation derives both fields from trajectory content at lower cost by marking only the transition points where responsibility changes.

The algorithm rests on one structural intuition: a task occupies a contiguous span of the trajectory and decomposes into smaller contiguous subtasks, so we model task structure as a set of nested named intervals and recover it by finding the transition points between them.

The algorithm reads a trajectory, finds the transition points where the agent's responsibility changes, and names the intervals on both sides. It then recurses into each interval, cutting again wherever a finer responsibility change remains, and stops when an interval reads as one coherent piece of work. Formally, a trajectory is an ordered step sequence $T = (t_1, \ldots, t_n)$, and a segmentation is a set S of named intervals over T that is nested (any two intervals are disjoint or one contains the other), covers every step, and contains the session-wide interval as its root. The algorithm computes S by one recursive rule: Segment(I) selects zero or more transition points inside interval I, which partition I into consecutive child intervals. Each child receives a short name starting with a verb and is segmented recursively. Selecting no transition terminates the branch. A step's operation path is the name chain of the intervals containing it, from the session root to its innermost interval. Equal paths fold across sessions. Concretely, in one Git-deployment execution the algorithm cuts the session into `build deployment system` and, inside it, cuts again where the agent stops writing configuration and starts debugging access, yielding `deploy branches` and `diagnose authentication`. A step inside the latter carries the path `build deployment system > diagnose authentication`, and because the same names reappear in the other two executions, their costs fold together in Figure 1. Any implementation able to find transitions and name intervals (an agent, a language model, or a statistical rule) can serve as the segmenter. In evaluation, we use Codex [31] with GPT-5.6-sol-high, with one fixed instruction per trajectory:

> *Input:* one trajectory (each step's prompt, command, and output summary, with no labels or scores).
> *Action:* read it, emit a mark wherever the responsibility changes, and use AgentPProf to check and update the

segmentation, and re-read and update the marks iteratively until you think the segmentation is complete.
Output: sparse path marks, e.g.,
`step 1: [deploy git server]`
`step 4: [deploy git server, diagnose SSH access]`
`step 15: [deploy git server, verify web endpoint]`
—one line only where the path changes; names start with a verb, one to three words.

The sparse marks still produce a complete segmentation because steps between two marks inherit the preceding path. Segmentation is iterative, so the agent can revise marks incrementally without re-annotating the entire trajectory. The model supports both offline evaluation and online annotation during execution.

4 Implementation

AgentPProf is a Rust CLI (~9.8K LOC, Figure 2) that realizes the design of Section 3. It reads Codex and Claude Code local JSONL history files and AgentSight [47] recordings for system effects. These files are written incrementally during agent execution, so AgentPProf can build profiles while the agent is still running. The pipeline has two stages: (1) preprocessing extracts a readable trajectory summary, and (2) the segmentation agent reads this summary and writes interval marks to an annotation file, which it can revise until satisfied. The CLI validates that marks are nested and cover every step, then emits a standard pprof protobuf profile that `go tool pprof` reads directly. Projection is deterministic: each operation contributes its semantic path and LLM/tool evidence, with session identities kept as pprof labels rather than visible frames, so equal semantic prefixes fold across sessions. Switching the additive measure replays the same annotations and changes only stack widths.

Fields already recorded literally (e.g., skill invocations) become stack levels without annotation cost. Multi-agent orchestration works automatically because subagent operations inherit the outer agent's semantic path. A non-LLM statistical segmenter is also available, scoring transitions by NPMI [6] and calibrating an unsupervised cutoff with k-means [27].

5 Evaluation

We evaluate whether AgentPProf effectively attributes resources, locates problems, and recovers human-recognizable task structure at practical cost. The evaluation uses three data classes. *Real inputs*: 41 long-horizon coding-agent sessions (three of which repeat one Git-deployment task and form the RQ1 case), 440 mixed web-agent trajectories [24], and one developer's complete local session history from the authors' workstation (1,394 sessions), whose 42 long-horizon development sessions are the annotated portion. *Annotated trajectories*: 405 CodeTraceBench trajectories with 20,866 operations and 2,948 human-annotated stages [21], 287 OSWorld-Human sessions [42], 1,012 AgentBoard goals [25], and 2,737 published action labels [7]. *Problem-localization benchmarks*: the complete AgentProcessBench, HINTBench, and TraceElephant workloads, with independently annotated faulty operations, over 27,346 operations [8, 11, 40]. All annotations, outcomes, and scores stay hidden from every method until its output is produced. Benchmark trajectories average 8–52 operations (short to medium); the 42-session workstation portion supplies long-horizon coverage.

5.1 RQ1: Does Semantic Profiling Improve Resource Attribution?

Setup. Improved resource attribution should (1) reunite cross-run responsibility scattered by alternative organizations and (2) reveal different bottlenecks across additive measures. We test both on 41 real long-horizon agent trajectories (3,146 user turns, 5,750 operations), including three independent runs of one Git-deployment task whose 735 steps receive 96 recursive annotations to semantic depth five.

The motivating case revisited. The Git-deployment case in Section 2 shows semantic organization reuniting responsibility scattered by alternative organizations; here we add elapsed time as a third measure. The hierarchy replays under all three measures with exact conservation, while `diagnose authentication`'s share rises from 21% (count) through 37% (time) to 46% (tokens). Changing only the measure more than doubles the attributed share. A quantitative control shows why both alternatives fail. *Raw-action grouping* (by literal action-type, e.g., `run`, `edit`) fails: 102 of 105 authentication operations are labeled only `run`, whose bucket includes 97 unrelated operations. Coarser action-kind grouping scatters them across six kinds (execute 39%, version-control 22%, edit 19%, inspect 12%, search 7%, install 1%). *Native call trees* place every operation under its distinct source LLM/tool call, without a shared responsibility node. Only the semantic hierarchy reunites authentication into one focusable cross-run path while preserving all call/tool evidence as leaves. Separately, real AgentSight [47] eBPF recordings likewise fold all 1,520 captured process spawns and file operations under task responsibilities with exact conservation, demonstrating end-to-end system effects.

Use case 2: where did this project's agent budget go? A developer spent weeks building AgentPProf and this paper with coding agents; we annotate the history's long-horizon portion (42 sessions: 18 Codex and 24 Claude Code; 1,252 prompts, 5,620 LLM calls, and 1.38 billion tokens). The question is *what did the agents actually spend tokens doing?* The profile shows a broad distribution without a dominant sink; the largest path (`refine paper > align evaluation`) holds only 1.7% of tokens. The three longest sessions (each tens of hours) have distinct dominant responsibilities: evaluation alignment, evidence inspection, and merge resolution. Switching to operation count shifts the concentration. Token mass stays at prompt depth (70%), whereas operation mass resolves deeper (44% at depths three and four). The history also demonstrates scaled multi-agent composition: 98 subagent delegations across 14 sessions, each containing all downstream subagent operations and composing with annotated semantic levels.

The hierarchy replays under FILE-READ, FILE-WRITE, and NETWORK-target widths, making side effects auditable by responsibility.

Literal skill invocations require no annotation, thus covering the complete history (1,394 sessions, 6.9B tokens). There, `paper-writing-style` leads both measures (29M tokens, 99.47% cache reads), making its repeated-context workflow the first inspection target. The largest session holds only 31% of that mass, so this priority appears only after folding. The 99.47% cache-read ratio

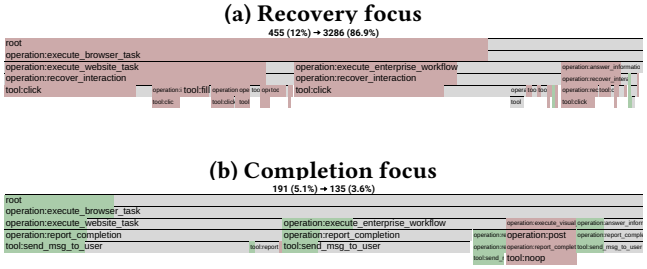

Figure 3: Stock-pprof differential flame graphs (bad minus good). Box width sums both contributions; inset shows net difference (rose: bad excess; green: good excess). Recovery dominates bad runs; completion favors good runs.

suggests repeated rebuilding of similar context. Reusing context across invocations could reduce token cost.

Across 440 AgentRewardBench trajectories (7,229 operations, 51,904,621 tokens, both exactly conserved), count- versus token-based rankings agree strongly (mean tau-b 0.886). Multi-measure profiling adds value precisely for the 13% of tasks below 0.7 agreement.

Takeaway. Both hold: the semantic hierarchy reunites responsibility scattered by raw-action and native organizations (authentication), while switching measures on that hierarchy changes attributed share by over 2× (21% to 46%), exposing bottlenecks hidden by operation count. These optimization targets can be restructured to reduce disproportionate task costs.

5.2 RQ2: Does Profiler Output Correspond to Real Problems?

Setup. Three tests assess correspondence between profiler output and independently annotated real problems: (1) *differential analysis* compares profile failure structure with expert behavioral labels across 440 runs; (2) *localization* tests whether profile scores improve existing benchmark judges' fault rankings; (3) *reading study* tests whether semantic names preserve an LLM agent's ranking with less inspection.

Use case 3: what do failing runs do differently? A team has 440 web-agent runs [24] over 125 tasks, each with successful and failed attempts (202 successful, 238 failed); reading all transcripts (7,229 operations) is infeasible. We pair failed and successful runs per task (338 occurrences), profile each group, and subtract them (failed minus successful) into a signed difference profile (Figure 3). The result is immediate: failed runs spend 44.6% of steps in `recover interaction` (retrying, re-searching, re-navigating) versus 12.0% for successful runs. The hierarchy decomposes recovery into verification challenges, repeated searches, mistaken navigation, and record retries; source labels identify sessions contributing each width. Failing agents thus get stuck in retry loops instead of backing out to try another approach. This structure matches independent expert looping labels on 435 trajectories at AP .634 versus a .398 random baseline (difference interval [.181, .293]), confirming a real behavioral pattern. A fixed-chain repeated/error control reaches

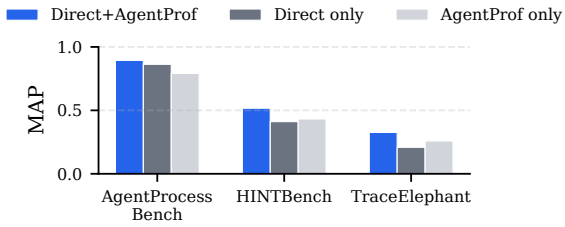

Figure 4: MAP over the complete RQ2 datasets (614, 400, and 220 queries included in MAP). Higher is better.

the same detection quality (AP .656), but only the recursive profile provides this decomposition and source navigation.

Supplementing benchmark's own diagnostics. Setup. Each MAP query has one or more annotated faulty operations. A method ranks its operations, scored by average precision (AP equals reciprocal rank for one faulty operation) and averaged as MAP [34]. We run complete AgentProcessBench, TraceElephant, and HINTBench workloads (the complete released test snapshot, 536 of 629 paper-reported trajectories); all profiler outputs precede fault-annotation disclosure. We average per-query AP over the 614, 400, and 220 MAP queries; 522 trajectories without annotated faulty operations provide dataset coverage but are excluded from MAP. The *direct diagnostic* is each benchmark's per-operation judge output, including LLM-as-judge methods, which reads the full trace and reference answer before naming the responsible agent and decisive step. *Direct-only* uses only that diagnostic; *Direct+AgentPProf* breaks ties by group score (vote mean for AgentProcessBench; otherwise 95% Wilson bound), with HINTBench field order fixed on 80 validation trajectories; *AgentPProf-only* uses only the profiler (Figure 4).

Results. Direct+AgentPProf beats Direct-only by 0.031, 0.107, and 0.117 MAP across three workloads (all statistically significant; Figure 4). Evidence-preserving grouping drives the localization gain. Semantic naming enables cross-run attribution (RQ1) and concentrates attention (measured below).

Profile-guided reading on TraceElephant. On 220 queries, a Grok 4.5 [43] agent-as-judge ranks operations by fault likelihood. Full-trace reading reaches MAP .502 at 12.6K tokens/query. A profile-guided reader selects at most five groups, reaching .455 while opening 53% of source, versus .465 and 65% with raw-action names. Per-query full reading is also context-window bounded. *Takeaway.* Profilers complement debuggers: profiles answer cross-run questions and guide inspection to relevant content. Profiled failure structure matches independent human labels across 435 trajectories. Profiles supplement existing judges' localization signal, while semantic names concentrate agent attention, reducing opened source characters from 65% to 53%.

Use case 4: guiding a real repair. This case tests whether a profile finding can guide effective repair. A standard AgentPProf profile of eight ToolSandbox scenarios isolates recurring call-ID syntax failure (5/21 tool operations). A profile-only analyst—given pprof output but no raw traces—identifies the compatibility-layer boundary as the repair target. The resulting one-line fix removes all syntax failures and, on 23 held-out confirmation scenarios (69

Method	B^3 P	B^3 R	B^3 F1	Bound. F1
Codex segmentation	0.793	0.736	**0.764**	**0.480**
Statistical recurrence	0.782	0.575	0.663	0.266
Causal Qwen2.5-3B task stack	0.557	0.792	0.650	0.257
Raw-action grouping	0.891	0.388	0.541	—
Native source tree	0.975	0.249	0.397	0.259
Source-native step	0.983	0.221	0.361	0.246

Table 1: Agreement with human stages on all 405 CodeTraceBench trajectories.

BEFORE/repair pairs), reduces agent-model tokens by 19.0% while passing a fixed −.05 official-similarity quality threshold .

5.3 RQ3: How Accurate Are the Identifiers?

Setup. We test whether recursive segmentation recovers human-recognizable task structure against 2,948 human-annotated stages from all 405 CodeTraceBench trajectories. F1 [4] asks whether related operations are grouped; exact adjacent-boundary F1 [36] asks whether cuts match human cuts. Baselines receive identical inputs (Table 1). Each output is evaluated at its predicted level: literal names use accuracy and macro-F1 [20], permutation-invariant partitions V-measure [35] or ordinary B^3, and adjacent interval boundaries exact precision, recall, and F1. For legacy field-valued datasets, conversion maps each predicted group into the same interval-annotation format before AgentPProf folds the original additive weights. Codex [31] receives each trajectory (prompts, commands, and output summaries; no visible labels or scores) and emits sparse interval marks over 17,148 steps without stage annotations (loaded after fixing all 405 outputs): 4,496 marks at semantic depths one (3), two (2,873), three (1,588), or four (32), with no requested depth. Name normalization maps 3,895 free-form interval names to 783 stable short names (e.g., `diagnose authentication`) with zero adjacent display-path collisions, rejecting unresolved names. Identical cross-session names merge, folding identical paths in the final profile.

Results. Codex segmentation reaches 0.764 B^3 F1 and 0.480 boundary F1, beating the strongest automatic baseline (multi-resolution recurrence) by +0.101 and +0.214. A stateful per-turn Qwen2.5-3B baseline reaches 0.650 B^3 F1, showing meaningful segmentation by smaller models. The marks exactly conserve 20,866 operations and 494,862,929 tokens. Predicted groups remain largely pure gold-stage subsets (B^3 precision 0.793), with residual error from over-segmentation rather than missed transitions: disagreements subdivide work instead of merging unrelated responsibilities, preserving per-group coherence. The interface generalizes across method families and segmentation scales: on OSWorld-Human [42], supervised Naive Bayes reaches 0.816 B^3 F1, a no-LLM recurrence segmenter 0.786, and the unchanged Codex instruction uses coarser task-level boundaries (0.448 B^3 F1). For closed-label literal tags, Qwen3.6-27B [33] labels 1,012 AgentBoard goals [25] at 0.695 macro-F1 and 2,737 AutoCodeRover, OpenHands, and RepairAgent trajectory actions [7, 37] at 0.498 macro-F1. *Takeaway.* Recursive segmentation recovers human-recognizable task structure: 0.764 B^3 F1 against human stages, +0.101 over the strongest automatic baseline. Non-LLM implementations of the same interface remain viable without a model.

5.4 RQ4: What Is the Profiling Cost?

Setup. We test profiling practicality through segmentation time and tokens plus profile replay speed. Cost comprises one automatic segmentation pass per corpus, then deterministic construction/replay.

Results. Codex (GPT-5.6-sol-high) segments all 405 CodeTraceBench trajectories in 37 minutes with up to four workers, averaging 29,754 input and 573 output tokens per trajectory. With fixed marks, construction scales linearly: the 27,765-operation union completes in 1.16 s at 465 MiB peak RSS, only 5.25 MiB (1.14%) above raw-action grouping. Deterministic construction of all 440 trajectories takes 0.26 s (operations) and 0.25 s (tokens): segmentation dominates cost, while construction remains sub-second.

Reducing annotation cost. Automatic annotation dominates construction. A compact skeleton plus selected full results (SELECTIVE), instead of every turn's complete content (FULL) reduces provider tokens 20.4% on 32 held-out CodeTraceBench task clusters while retaining full operation coverage and meeting a fixed −.03 quality threshold on B^3 and boundary F1 .

Takeaway. Segmentation is one-time (37 min for 405 trajectories); each later view, measure switch, and inspection costs about one second. Selective evidence cuts annotation tokens 20.4% with complete coverage and preserved structural quality.

6 Related Work

Classic profilers fold runtime call stacks over stable code identity, and Pivot Tracing groups causally related measurements [13, 14, 26]. Agent observability platforms such as LangSmith, Langfuse, and Phoenix [2, 17, 19] record spans with metadata for per-execution debugging, and analytics layers such as Datadog Patterns and LangSmith Insights [10, 18] roll up cross-trace hierarchies or input distributions, but none constructs recursive semantic responsibility with conserved additive measures in a standard profile. Cross-run analyses (TraceProbe, Graphectory, Act·onomy, CHIEF, Hodoscope, TraceGraph) [12, 22, 30, 38, 41, 48] and per-run diagnosis and localization [5, 8, 11, 15, 21, 23, 28, 29, 39, 40] answer what an agent did and which step went wrong. AgentPProf instead asks how much of an additive resource a task or workflow phase is responsible for. It recovers variable-depth responsibility from source content (where Act·onomy fixes three levels and TraceProbe one), conserves measures exactly across count, token, and time widths, keeps LLM/tool calls as leaf nodes, and emits standard pprof. No compared system provides all four, and AgentPProf stays complementary to per-run diagnosis, as RQ2 measures directly.

7 Conclusion

Agent observability needs profiling, not only debugging. AgentPProf shows that the semantic operation stack model, driven by recursive operation segmentation, brings hierarchical attribution to agent trajectories, effectively attributes resources, locates problems, and helps optimize token cost at practical profiling cost. Adaptive triggering, intent-level state-transition models, profile-driven runtime control, and multi-project evaluation with tracing-ecosystem integration remain future work.

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] Amit Bagga and Breck Baldwin. 1998. Entity-Based Cross-Document Coreferencing Using the Vector Space Model. In *36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, Volume 1*. 79–85. doi:10.3115/980845.980859

[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] Gerlof Bouma. 2009. Normalized (Pointwise) Mutual Information in Collocation Extraction. In *From Form to Meaning: Processing Texts Automatically, Proceedings of the Biennial GSCL Conference 2009*. 31–40. https://svn.spraakbanken.gu.se/repos/gerlof/pub/www/Docs/npmi-pfd.pdf

[7] Islem Bouzenia and Michael Pradel. 2025. Understanding Software Engineering Agents: A Study of Thought-Action-Result Trajectories. In *2025 IEEE/ACM 40th International Conference on Automated Software Engineering (ASE)*. 2846–2857. doi:10.1109/ASE63991.2025.00234

[8] Mengzhuo Chen, Junjie Wang, Fangwen Mu, Yawen Wang, Zhe Liu, Huanxiang Feng, and Qing Wang. 2026. Seeing the Whole Elephant: A Benchmark for Failure Attribution in LLM-Based Multi-Agent Systems. arXiv preprint arXiv:2604.22708. doi:10.48550/arXiv.2604.22708

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

[10] Datadog. 2026. LLM Observability Patterns. https://docs.datadoghq.com/llm_observability/monitoring/patterns/.

[11] Shengda Fan, Xuyan Ye, Yupeng Huo, Zhi-Yuan Chen, Yiju Guo, Shenzhi Yang, Wenkai Yang, Shuqi Ye, Jingwen Chen, Haotian Chen, Xin Cong, and Yankai Lin. 2026. AgentProcessBench: Diagnosing Step-Level Process Quality in Tool-Using Agents. In *Proceedings of KDD*. doi:10.1145/3770855.3817494

[12] Jie Gao, Kaiser Sun, Jen-tse Huang, Katherine Van Koevering, Sijie Ji, Heyuan Huang, Weiyan Shi, Zhuoran Lu, Ziang Xiao, Daniel Khashabi, and Mark Dredze. 2026. How to Interpret Agent Behavior. arXiv:2605.13625 [cs.AI] https://arxiv.org/abs/2605.13625

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

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

[15] Yeonjun In, Mehrab Tanjim, Jayakumar Subramanian, Sungchul Kim, Uttaran Bhattacharya, Wonjoong Kim, Sangwu Park, Somdeb Sarkhel, and Chanyoung Park. 2026. Rethinking Failure Attribution in Multi-Agent Systems: A Multi-Perspective Benchmark and Evaluation. arXiv:2603.25001 [cs.AI] https://arxiv.org/abs/2603.25001

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

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

[18] LangChain. 2026. LangSmith Insights. https://docs.langchain.com/langsmith/insights.

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

[20] David D. Lewis, Yiming Yang, Tony G. Rose, and Fan Li. 2004. RCV1: A New Benchmark Collection for Text Categorization Research. *Journal of Machine Learning Research* 5 (2004), 361–397. https://www.jmlr.org/papers/volume5/lewis04a/lewis04a.pdf

[21] Han Li, Yifan Yao, Letian Zhu, Rili Feng, Hongyi Ye, Jiaming Wang, Yancheng He, Pengyu Zou, Lehan Zhang, Xinping Lei, Haoyang Huang, Ken Deng, Ming Sun, Zhaoxiang Zhang, He Ye, and Jiaheng Liu. 2026. CodeTracer: Towards Traceable Agent States. arXiv:2604.11641 https://arxiv.org/abs/2604.11641

[22] Shuyang Liu, Yang Chen, Rahul Krishna, Saurabh Sinha, Jatin Ganhotra, and Reyhaneh Jabbarvand. 2026. Process-Centric Analysis of Agentic Software Systems. *Proceedings of the ACM on Programming Languages* 10, OOPSLA1 (2026), 1961–1988. doi:10.1145/3798271

[23] 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

[24] 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

[25] Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. 2024. AgentBoard: An Analytical Evaluation Board of Multi-turn LLM Agents. In *Advances in Neural Information Processing Systems*, Vol. 37. 74325–74362. https://openreview.net/forum?id=suWBHqzaXz

[26] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 378–393. doi:10.1145/2815400.2815415

[27] James B. MacQueen. 1967. Some Methods for Classification and Analysis of Multivariate Observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, Vol. 1. 281–297. https://projecteuclid.org/ebooks/berkeley-symposium-on-mathematical-statistics-and-probability/Proceedings-of-the-Fifth-Berkeley-Symposium-on-Mathematical-Statistics-and/chapter/Some-methods-for-classification-and-analysis-of-multivariate-observations/bsmsp/1200512992

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

[29] 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

[30] Junjie Nian, Kang Chen, Ge Zhang, Yixin Cao, and Yugang Jiang. 2026. TraceGraph: Shared Decision Landscapes for Diagnosing and Improving Agent Trajectories. arXiv:2605.31308 [cs.AI] https://arxiv.org/abs/2605.31308

[31] OpenAI. 2025. Codex CLI: Lightweight Coding Agent That Runs in Your Terminal. https://github.com/openai/codex.

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

[33] Qwen Team. 2026. Qwen3.6-27B: Flagship-Level Coding in a 27B Dense Model. Official model card. https://huggingface.co/Qwen/Qwen3.6-27B

[34] Stephen Robertson. 2008. A New Interpretation of Average Precision. In *Proceedings of the 31st Annual International ACM SIGIR conference on Research and Development in Information Retrieval*. 689–690. doi:10.1145/1390334.1390453

[35] Andrew Rosenberg and Julia Hirschberg. 2007. V-Measure: A Conditional Entropy-Based External Cluster Evaluation Measure. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. 410–420. https://aclanthology.org/D07-1043/

[36] Teemu Ruokolainen, Oskar Kohonen, Kairit Sirts, Stig-Arne Grönroos, Mikko Kurimo, and Sami Virpioja. 2016. A Comparative Study of Minimally Supervised Morphological Segmentation. *Computational Linguistics* 42, 1 (2016), 91–120. doi:10.1162/COLI_a_00243

[37] Amirali Sajadi, Tu Nguyen, Kimmie Huynh, Esteban Parra, and Preetha Chatterjee. 2026. TraceView: Interactive Visualization of Agentic Program Repair Trajectories. arXiv:2606.22110 [cs.SE] doi:10.48550/arXiv.2606.22110

[38] Rui Shu, Chun Yong Chong, Xin Zhou, Yun Peng, Zihan Wu, Xu Han, Zeyang Zhuang, Guowen Yuan, and Yuan Wang. 2026. What Resolve Rate Hides: Trajectory Structure Diagnostics for Coding Agents. arXiv:2607.06184 [cs.SE] https://arxiv.org/abs/2607.06184

[39] 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

[40] Jiacheng Wang, Jinchang Hou, Fabian Wang, Ping Jian, Chenfu Bao, and Zhonghou Lv. 2026. HINTBench: Horizon-Agent Intrinsic Non-Attack Trajectory Benchmark. arXiv preprint arXiv:2604.13954. doi:10.48550/arXiv.2604.13954

[41] Yawen Wang, Wenjie Wu, Junjie Wang, and Qing Wang. 2026. From Flat Logs to Causal Graphs: Hierarchical Failure Attribution for LLM-based Multi-Agent Systems. arXiv:2602.23701 [cs.AI] https://arxiv.org/abs/2602.23701

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

[43] xAI. 2026. Introducing Grok 4.5. https://x.ai/news/grok-4-5.

[44] 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*.

[45] 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

[46] 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

[47] 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

[48] Ziqian Zhong, Shashwat Saxena, and Aditi Raghunathan. 2026. Hodoscope: Unsupervised Monitoring for AI Misbehaviors. arXiv:2604.11072 [cs.AI] https://arxiv.org/abs/2604.11072