BIN-BENCH: Can LLM Agents Reason Through Long-Horizon Binary Analysis?

Anonymous ACL submission

Abstract

Most existing benchmarks evaluate LLM agents by whether they succeed or fail, but cannot show how reasoning breaks down over many steps. We introduce BIN-BENCH, a benchmark of 520 real-world binary files where agents must reason through dozens of steps, and early mistakes cause later failures that cannot be easily fixed. We propose metrics that analyze complete reasoning traces to see how agents explore binaries, remember information, and where their reasoning goes wrong. Our evaluation shows that LLM agents struggle to maintain accurate understanding over long reasoning sequences, and fail in ways that success/failure metrics cannot reveal. We find that agents face fundamental tradeoffs: being consistent in reasoning makes it harder to recover from early mistakes, while being flexible to correct errors makes it harder to maintain coherent understanding. Adding planning, reflection, or error correction mechanisms helps only slightly. Our findings suggest that improving long-horizon reasoning requires better LLM capabilities and context management, not just more sophisticated control strategies. An anonymized version of the benchmark and related artifacts is available at https://anonymous.4open.science/r/anonymous-A-4BDE/.

1 Introduction

Large language model (LLM) agents are now used in real systems to solve tasks that require multiple steps of reasoning and repeated interactions with tools over long time spans. Existing benchmarks for LLM agents mainly focus on four types of tasks: code generation (Chen et al., 2021), agent performance across different environments (Liu et al., 2023b; Xu et al., 2024a), software engineering workflows (Jimenez et al., 2023; Yang et al., 2023), and problem solving in specific domains (Mialon et al., 2023; Rein et al., 2023). Most of these benchmarks evaluate agents based on whether they complete a task successfully, using metrics such as pass@k or simple success and failure outcomes. However, this evaluation style treats task success as a direct indicator of reasoning ability and compresses a long sequence of dependent reasoning steps into a single final result. For many critical domains, such outcome-based evaluation is insufficient: in binary vulnerability discovery, operating system security analysis, and automotive system analysis, merely knowing whether an agent succeeded or failed provides little insight into how reasoning processes unfold, where vulnerabilities might be missed, or how errors propagate through long reasoning chains.

Building such a benchmark is difficult. The tasks must push agents to reason over many steps, but we also need to see how they reason, not just whether they succeed. Binary vulnerability discovery is a good testbed for this purpose because it forces agents to handle several challenges at the same time. First, agents must reason through long sequences, often with more than 100 steps. Second, they must use tools such as r2 and Ghidra with precise inputs, including exact function addresses. Third, they need to remember information across hundreds of steps. Fourth, they operate with limited information, since each tool call reveals only a small part of the binary. Finally, early mistakes are hard to fix: for example, if an agent uses the wrong function address at the beginning, this error can cause serious problems later. Such benchmark provides a new perspective to evaluate LLM agents: whether agents succeed by reasoning carefully, or by chance; and how reasoning quality degrades over long tasks under partial observability and irreversible errors.

We introduce BIN-BENCH, a benchmark that evaluates LLM agents on real binary analysis tasks. As shown in Figure 1, agents work directly with binary files from embedded devices. In each task, agents need to understand how a program works, identify security problems, and track how untrusted

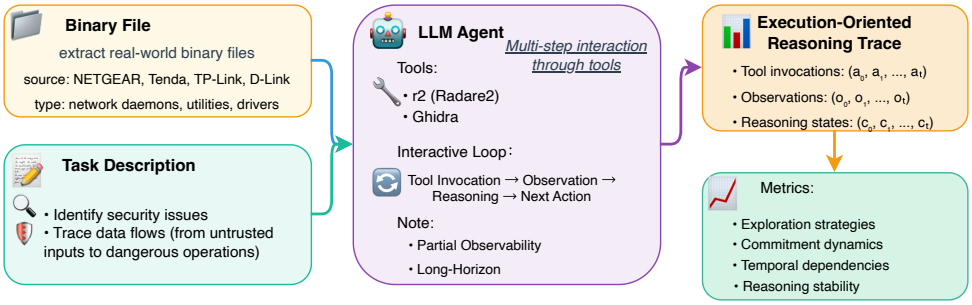

Figure 1: BIN-BENCH uses 520 real binary files from embedded devices. Agents analyze each binary through many tool interactions (r2, Ghidra), and we evaluate their complete reasoning traces to see how they explore, remember information, and where reasoning breaks down.

input reaches risky operations. Agents interact with binaries using tools such as r2 and Ghidra, where each tool call reveals only a small amount of information. We record and analyze the full reasoning traces to observe how agents explore the program, what information they keep in memory, and where they make mistakes.

We evaluate several widely used agent methods on BIN-BENCH, including ReAct and its variants, across a range of current LLMs. These models include both open-source LLMs that run locally and commercial API-based LLMs, covering different sizes and architectures. Across all settings, we observe the same pattern: agents struggle to maintain an accurate understanding when reasoning over long sequences of steps. Many of these failures are not visible from simple success or failure scores. We also observe a clear tradeoff in agent behavior. When agents try to stay consistent with earlier decisions, they often fail to correct early mistakes. When they try to be more flexible, they lose track of the overall reasoning context. Adding planning, reflection, or error correction brings only limited improvements. Overall, these results suggest that the main limitation is not the choice of control strategy, but the agents' ability to remember and manage information over long tasks.

Our contributions are twofold. First, we introduce BIN-BENCH, a benchmark with 520 real binaries designed to test long-horizon reasoning in LLM agents. Second, we propose metrics that analyze full reasoning traces, rather than only final success or failure. Our trace-level evaluation reveals how agents explore, commit to decisions, and retain information over long tasks.

2 Bin-Bench

BIN-BENCH is a benchmark built on real-world binaries extracted from firmware images of embedded devices. Agents interact with binaries exclusively through standard analysis tools such as r2 and Ghidra, where each tool invocation reveals only limited information.

2.1 Task Overview

BIN-BENCH focuses on long-horizon, tool-based reasoning under realistic constraints. Solving a task typically requires hundreds of sequential tool interactions. Agents must accumulate and maintain information over time, including function boundaries, control flow, and data dependencies, while operating under partial observability: no single tool call reveals the full program structure. Importantly, early mistakes—such as using an incorrect function address—often propagate through subsequent steps and are difficult to recover from, making reasoning errors effectively irreversible.

Unlike language-only reasoning tasks, we produce execution-oriented reasoning traces that capture tool actions, observations, and intermediate reasoning states. These traces reveal how agents explore programs step by step, enabling direct inspection of the reasoning process beyond final outcomes.

2.2 Benchmark Construction

We construct BIN-BENCH through a multi-stage pipeline to ensure realistic analysis conditions and preserve diversity. Table 1 summarizes key statistics of BIN-BENCH.

Stage I: Firmware collection and binary extraction. We collect firmware images from embedded devices produced by vendors such as NETGEAR, Tenda, TP-Link, and D-Link. From these images, we extract executable binaries commonly found in embedded systems, including network

Table 1: BIN-BENCH corpus statistics: where K is the combination of baselines and LLMs.

Statistic	Value
Total binaries	520
Architectures	ARM, MIPS, x86
Binary types	Network daemons, utilities drivers, services
Format	ELF32
Reference traces	$520 \times K$

services, system utilities, and device management programs. The resulting binaries span multiple architectures, including ARM, MIPS, and x86.

Stage II: Quality filtering and deduplication. We apply multiple filtering criteria to ensure binaries represent realistic analysis challenges: (1) each binary must be a complete, executable ELF file, (2) binaries must exhibit sufficient complexity (containing multiple functions and realistic control flow structures), and (3) binaries must be analyzable using standard tools (r2, Ghidra) without requiring specialized hardware or proprietary formats. We deduplicate binaries by filename to avoid overrepresentation of popular binaries while preserving diversity across different program types. This deduplication strategy ensures we evaluate LLM agents across functionally distinct binaries rather than multiple instances of the same program. After filtering and deduplication, we obtain 520 distinct binaries.

Stage III: Reference trace generation. For each binary, we generate a reference analysis trace that records a complete step-by-step reasoning process, typically involving dozens of tool invocations. Each trace logs the sequence of tool commands, observations returned by the tools, and the corresponding reasoning context. These traces reflect realistic exploration patterns and serve as the basis for trace-level evaluation.

2.3 Comparison with Existing Benchmarks

Table 4 (Appendix A) compares BIN-BENCH with representative agent benchmarks. Existing benchmarks such as SWE-bench and AgentBench primarily emphasize task completion in settings where agents have access to complete information or can recover from errors through re-sampling. In contrast, BIN-BENCH explicitly evaluates long-horizon reasoning under partial observability and irreversible errors. None of the existing benchmarks simultaneously cover all these dimensions, leaving a gap in evaluating reasoning stability over extended tool-based interactions. BIN-BENCH is designed to fill this gap.

3 Metrics over Analysis Traces

BIN-BENCH evaluates LLM agents in long-horizon binary analysis tasks where agents interact with binaries through tools (r2, Ghidra). Unlike traditional benchmarks where failures appear immediately (e.g., a program crashes), binary analysis failures are often invisible until much later. Errors accumulate gradually: an agent might make a wrong assumption early on, and this mistake silently propagates through hundreds of reasoning steps before the agent realizes something is wrong. We call this phenomenon *state drift*: the agent's internal understanding of the program becomes increasingly disconnected from the actual binary state.

To capture this gradual degradation, we introduce *trace-level metrics* that analyze the entire reasoning process, not just the final outcome. These metrics examine how agents reason, act, and observe over time, asking questions like: Does the agent remember information from earlier steps? Does it maintain consistent beliefs? Does it follow a coherent strategy? By focusing on the reasoning process, we can identify problems long before they lead to explicit failures.

State Fidelity (Primary Metrics). The core challenge in binary analysis is maintaining accurate knowledge about the program state over long reasoning sequences. For example, an agent might discover a function address at step 10, but needs to use that same address again at step 50. Can the agent remember it? Or does it need to rediscover the information, wasting steps and potentially making errors?

Dependency Distance measures how far back in the trace an agent must look to find information needed for the current action. If an agent frequently needs to look back 50 or 100 steps, it shows good long-term memory. If it only looks back a few steps, it suggests the agent is forgetting earlier information and relying only on recent observations.

Reasoning–Action–Observation (RAO) Consistency checks whether the agent's reasoning, actions, and observations form a coherent cycle. Specifically: Does the agent's stated reasoning lead to appropriate tool commands? And when the tool returns results, does the agent actually use those results to update its understanding?

Decision Stability (Supporting Metrics). Beyond maintaining accurate state, agents must also

follow coherent strategies rather than constantly changing direction. In long-horizon analysis, agents that frequently abandon their plans or oscillate between different approaches often struggle to make progress.

Phase Structure examines how agents move between different types of analysis activities. For instance, an agent might start with reconnaissance (scanning the binary), then move to decompilation (reading function code), then to cross-reference analysis (tracing function calls). A well-structured trace shows clear progression through these phases. An unstable trace shows frequent back-and-forth switching without sustained focus, suggesting the agent lacks a coherent strategy.

Hypothesis Persistence measures how agents form and maintain analytical hypotheses. For example, an agent might hypothesize that "function X contains a buffer overflow vulnerability" and then spend several steps investigating this hypothesis. If the agent frequently forms hypotheses but quickly abandons them, it suggests unstable decision-making.

Error Propagation (Diagnostic Metric). We also analyze how early mistakes evolve over time. In binary analysis, once an agent makes an incorrect decisoin, it may continue building on that decision for many steps. The agent might repeatedly execute the same sequence of commands, hoping for different results, creating an ineffective loop.

Error Propagation detects these repetitive patterns by identifying short command sequences that recur multiple times. For example, if an agent repeatedly executes "decompile function A, then decompile function B, then decompile function A again," this suggests the agent is stuck in a loop and unable to recover from an earlier error. High loop severity indicates that agents are unable to break out of unproductive patterns, wasting computational resources without advancing the analysis.

Overall, our metrics focus on the reasoning process rather than just the final outcome. By examining observable behaviors in the trace, we can identify systematic weaknesses in long-horizon reasoning that would be invisible if we only looked at whether the agent eventually succeeded or failed.

4 Experimental Setup

4.1 Agent Interface and Execution Traces

Each task instance in BIN-BENCH consists of a binary file and an analysis task. The agent is provided with the binary file path and a natural language task description that requires understanding program behavior, identifying potential security issues, and tracing data flows from untrusted inputs to sensitive operations.

Agents interact with binaries exclusively through standard binary analysis tools, including `radare2` (`r2`) and `Ghidra`. No source code, symbols, or external documentation are available. Each tool invocation reveals only local information (e.g., disassembly output or function metadata), requiring agents to incrementally integrate fragmented observations over multiple steps.

The agent produces an execution-oriented reasoning trace, consisting of: (1) reasoning states (from the `thought` field), (2) tool actions (commands issued to r2 or Ghidra), (3) observations (tool outputs), and (4) their temporal ordering. This trace serves as the primary object for evaluation.

Table 2: LLMs used in evaluation.

Model	Params	Arch.
DeepSeek-Coder-V2-Instruct	16B	Dense
GPT-OSS-20B	20B	MoE
Phi-4	14B	Dense
Qwen3Max (`qwen3-max`)	–	Commercial
GPT-5.2 (`gpt-5.2`)	–	Commercial

4.2 Backend LLMs

We evaluate all baselines using 5 LLMs (Table 2) spanning different parameter scales, architectures, and deployment modes. The model set includes both locally deployed open-source LLMs and commercial API-based LLMs, allowing us to analyze how model capabilities interact with agent reasoning strategies under partial observability. All models use identical decoding configurations (temperature = 0.0, default `top_1` and `max_tokens`).

4.3 Baseline Agent Methods

We evaluate four executable LLM-based agent baselines designed to operate over long interaction horizons with explicit tool usage (Appendix B). Each baseline produces a complete reasoning trace and follows a common ReAct-style reasoning–action–observation backbone. Baselines differ only in their agent-level decision strategies, which shape how agents explore, commit to hypotheses, and revise earlier decisions.

ReAct (Yao et al., 2022). A single-agent ReAct-style system that performs linear analysis without explicit revision or backtracking. The agent follows

a reasoning–action–observation loop, serving as a minimal baseline.

ReAct + Reflexion (Shinn et al., 2023). This variant augments ReAct with periodic self-reflection. The agent reviews past decisions every 10 steps or when stagnation is detected, and may revise its analysis strategy based on accumulated observations.

ReAct + Plan-and-Execute (Liu et al., 2023a; Yao et al., 2023). This baseline introduces an explicit planning phase before execution. The agent generates a high-level analysis plan (e.g., identifying entry points or data-flow goals) and then executes the plan step by step, resulting in structured traces with early commitment to exploration paths.

ReAct + Self-Correction. This variant adds explicit error detection and correction after each step. The agent identifies errors in tool usage, interpretation, or reasoning assumptions and attempts to correct them in subsequent actions, enabling analysis of error recovery dynamics.

5 Experimental Results

We evaluate four baseline agent methods instantiated with 5 LLMs on BIN-BENCH using the trace-level metrics extracted in Appendix C. Detailed trace analysis experiments are in Appendix E. Table 3 reports mean performance across all metrics, grouped by model and baseline.

Across all settings, we observe that current agent methods struggle to maintain stable reasoning over long interaction horizons under partial observability. While different reasoning strategies exhibit distinct behavioral patterns, none consistently achieves strong performance across all trace-level metrics. Supplementary experiments are in Appendix F. Below, we analyze these results by focusing on how different strategies manage long-horizon dependencies, commitment, and error recovery.

5.1 Overall Performance on Trace-Level Metrics

Table 3 summarizes performance on five trace-level metrics. Differences across baselines indicate systematic variations in how reasoning strategies manage long-horizon reasoning. Among them, RAO Consistency and Dependency Distance directly reflect an agent's ability to preserve and reuse state over long reasoning horizons.

Across models, RAO Consistency remains relatively stable across baselines, typically ranging between 0.48 and 0.56. This suggests that most agents can locally maintain coherent reasoning–action cycles. In contrast, Dependency Distance varies substantially across strategies, indicating that long-range state preservation is the primary bottleneck.

Self-Correction consistently achieves the highest Dependency Distance across most models, indicating that explicit error detection enables agents to reuse earlier observations over longer horizons. In contrast, Plan-and-Execute shows the lowest Dependency Distance, reflecting early commitment to high-level plans that limit the accumulation of contextual information over time. Self-Correction also reflects agents' ability to maintain context over extended horizons, deferring tool invocations until sufficient information accumulates, rather than committing early to potentially incorrect paths.

Importantly, higher Dependency Distance does not correspond to faster convergence or fewer steps. Instead, it reflects an agent's ability to defer decisions, accumulate information, and later act on observations made far earlier in the trace. This distinction is critical in binary analysis, where premature commitment often leads to irreversible errors. This property stems from the inherent constraints of binary analysis: agents can only observe partial information through tool-mediated interactions, while the global program state is too large to maintain in full.

5.2 Effects of Reasoning Strategies

Table 3 shows that introducing additional control mechanisms—planning, reflection, or self-correction—yields only modest improvements over the ReAct baseline. The variance in performance across baselines is small.

Planning improves Phase Structure but consistently reduces Dependency Distance, indicating that early structural commitments constrain long-horizon exploration. Reflexion slightly reduces Hypothesis Persistence, but does not significantly improve overall state stability. Self-Correction improves long-range dependency usage but often increases Error Propagation, reflecting repeated correction attempts that introduce new inconsistencies.

Overall, these results indicate that control mechanisms primarily shift how agents trade off between commitment and flexibility, rather than fundamentally improving long-horizon reasoning.

Table 3: Performance comparison across baseline methods and models on core trace-level metrics. Models are grouped by deployment type: open-source (local deployment) and commercial (API). Values are mean scores.

Model	Baseline	RAO Consistency	Dependency Distance	Phase Structure	Hypothesis Persistence	Error Propagation
DeepSeek-Coder-V2	ReAct	0.58 ± 0.19	41.68 ± 16.66	0.61 ± 0.49	7.12 ± 8.54	0.69 ± 0.31
	ReAct+R	0.64 ± 0.21	76.57 ± 33.44	0.51 ± 0.50	1.85 ± 4.54	0.66 ± 0.34
	ReAct+P	0.59 ± 0.23	43.50 ± 15.48	0.60 ± 0.49	9.44 ± 10.21	0.81 ± 0.19
	ReAct+SC	0.36 ± 0.34	64.62 ± 51.17	0.92 ± 0.28	0.05 ± 0.09	0.39 ± 0.45
GPT-OSS-20B	ReAct	0.56 ± 0.20	31.94 ± 16.52	0.66 ± 0.48	2.11 ± 3.88	0.15 ± 0.26
	ReAct+R	0.54 ± 0.25	65.98 ± 35.05	0.68 ± 0.47	0.16 ± 0.11	0.20 ± 0.33
	ReAct+P	0.62 ± 0.17	31.61 ± 19.14	0.83 ± 0.38	2.61 ± 4.66	0.17 ± 0.29
	ReAct+SC	0.64 ± 0.18	89.66 ± 38.95	0.74 ± 0.44	0.12 ± 0.05	0.26 ± 0.33
Phi-4 (14B)	ReAct	0.48 ± 0.19	26.73 ± 17.81	0.57 ± 0.50	3.34 ± 5.25	0.30 ± 0.36
	ReAct+R	0.45 ± 0.20	25.75 ± 15.59	0.41 ± 0.50	0.13 ± 0.07	0.18 ± 0.30
	ReAct+P	0.39 ± 0.18	25.96 ± 13.67	0.59 ± 0.50	2.73 ± 4.33	0.40 ± 0.37
	ReAct+SC	0.48 ± 0.18	48.69 ± 15.77	0.37 ± 0.49	0.11 ± 0.05	0.30 ± 0.33
Qwen3Max	ReAct	0.53 ± 0.11	19.19 ± 10.50	0.45 ± 0.50	1.11 ± 0.80	0.00 ± 0.00
	ReAct+R	0.50 ± 0.11	33.55 ± 15.58	0.46 ± 0.50	0.41 ± 0.60	0.00 ± 0.00
	ReAct+P	0.48 ± 0.11	17.17 ± 8.73	0.49 ± 0.50	1.35 ± 0.95	0.00 ± 0.00
	ReAct+SC	0.54 ± 0.11	39.28 ± 17.12	0.52 ± 0.50	0.15 ± 0.03	0.00 ± 0.00
GPT-5.2	ReAct	0.55 ± 0.10	21.50 ± 11.20	0.47 ± 0.49	1.25 ± 0.85	0.00 ± 0.00
	ReAct+R	0.52 ± 0.10	36.20 ± 16.30	0.48 ± 0.50	0.45 ± 0.55	0.00 ± 0.00
	ReAct+P	0.50 ± 0.10	19.80 ± 9.50	0.51 ± 0.50	1.42 ± 0.98	0.00 ± 0.00
	ReAct+SC	0.56 ± 0.10	42.15 ± 18.50	0.54 ± 0.49	0.18 ± 0.04	0.00 ± 0.00

5.3 Backend LLMs' Effects

Table 3 reveals mixed patterns when comparing open-source and commercial models across trace-level metrics. Commercial models (Qwen3Max, GPT-5.2) achieve slightly higher RAO Consistency (typically 0.50–0.56) compared to most open-source models (DeepSeek-Coder-V2: 0.36–0.64, GPT-OSS-20B: 0.54–0.64, Phi-4: 0.39–0.48), suggesting that larger model capacity improves local reasoning coherence. However, this advantage does not consistently translate to better long-horizon performance: Dependency Distance varies substantially (open-source: 25.75–89.66, commercial: 17.17–42.15), with some open-source models (e.g., GPT-OSS-20B with ReAct+SC: 89.66) achieving higher dependency distances than commercial alternatives. Similarly, Error Propagation shows no clear pattern favoring commercial models—commercial models consistently score 0.00 (indicating no error propagation), while open-source models show more variation (0.18–0.81).

These metrics suggest that increased model capacity improves local reasoning stability, but does not resolve the core challenge of maintaining coherent global state over long horizons under partial observability. No single model consistently dominates across all metrics, indicating that model choice alone is insufficient to overcome long-horizon reasoning failures in binary analysis.

5.4 Metric Patterns and Error Manifestations

Tables 3 and Appendix D allow us to examine how different trace-level metrics correspond to concrete failure behaviors during long-horizon binary analysis. Once an error is introduced early in the reasoning process, it often affects many subsequent steps. Rather than treating errors as isolated outcomes, we analyze how errors emerge, propagate, and persist across extended reasoning traces. Across all models and strategies, we observe recurring metric patterns that consistently align with specific forms of reasoning breakdown.

Consistency vs. Irreversibility. Table 3 shows that higher RAO Consistency often co-occurs with higher Error Propagation scores, particularly for open-source models. Those metrics reflect a pattern: *Consistency* $\uparrow \rightarrow$ *Irreversibility* $\uparrow$. In many traces, agents that maintain strong internal consistency tend to commit early to specific assumptions, such as function roles or data flows. When these assumptions are incorrect, the agent continues to build upon them, making later correction difficult. From a trace perspective, this behavior appears as severe error propagation, even though the agents' reasoning remains locally coherent throughout the

task.

Flexibility vs. State Coherence. In contrast, some strategies exhibit lower error propagation by revising assumptions more frequently. However, these traces often show reduced consistency across reasoning steps. These metrics reflect another pattern: *Flexibility* $\uparrow \rightarrow$ *State Coherence* $\downarrow$. Frequent corrections can disrupt the agent's internal state, leading to fragmented or unstable reasoning trajectories. Repeated loops and partial restarts observed in such traces suggest that lower propagation scores do not necessarily imply better overall understanding, but rather a different failure mode centered on state incoherence.

Non-Deterministic Error Patterns. Across all models and baselines, we observe substantial variance in metric values, even under fixed experimental settings. The same strategy may succeed in some runs while failing in others, with different tradeoffs between consistency, flexibility, and error persistence. This non-deterministic behavior is amplified by binary complexity, where different program structures expose different weaknesses in long-horizon reasoning. As a result, no single metric captures a dominant failure mode across all runs. This suggests that long-horizon binary analysis remains highly sensitive to early decisions, especially under partial observability and irreversible tool actions.

Overall, these metric patterns highlight that long-horizon binary analysis involves inherent tradeoffs between consistency, flexibility, and error persistence. By exposing how these tradeoffs manifest in full reasoning traces, our metrics provide a more detailed view of agent behavior than success-based evaluation alone.

5.5 Qualitative Analysis

We analyze the `88ip` binary, which contains three exploitable vulnerabilities and was successfully analyzed by all four baselines, enabling direct comparison of how metrics (internal reasoning characteristics) manifest as errors (observable failures). The `88ip` binary is a 32-bit ARM ELF executable that processes command-line arguments and performs network operations.

Metrics Analysis. The trace exhibits a critical metric pattern: low RAO Consistency, meaning the agent fails to effectively incorporate observation results into subsequent reasoning. This manifests in the trace as a disconnect between tool outputs and reasoning updates. For example, at step 11, the agent analyzes `fcn.0000a1c4` and receives decompilation output showing multiple `doSystemCmd` calls with `param_4` (`argv[3]`) as arguments. However, the agent's subsequent reasoning does not integrate this finding: the thought focuses on tracing register values rather than recognizing that `param_4` is user-controlled and passed directly to `doSystemCmd`, which constitutes a command injection vulnerability.

This low integration continues throughout the trace. At step 26, the agent's thought states: *"Now need to confirm that param_4 is indeed* `argv[3]`. *Let's examine main function again to verify the call."* This reveals that the agent is re-verifying information it should have already integrated from earlier observations—a clear sign. The agent cannot maintain a coherent understanding of function arguments across reasoning cycles, causing it to repeatedly revisit the same analysis questions.

Error Manifestation (Observable Failures). The agent repeatedly revisits the same function (`fcn.0000a1c4` at steps 11, 26, and 31) because it cannot maintain a coherent understanding of the function's behavior across multiple observations. Each revisit introduces state inconsistencies: at step 11, the agent analyzes the function and identifies `doSystemCmd` calls; at step 26, it re-verifies basic function arguments; at step 31, it re-examines the same `doSystemCmd` calls. The agent cannot determine which analysis state is correct—the initial understanding, the first revision, or subsequent revisions.

The trace reveals this inherent error: despite finding all three vulnerabilities, the trace requires 133 steps because the agent must repeatedly re-analyze the same functions due to state inconsistencies. Each revision introduces new state drift, as the agent's error detection mechanism triggers corrections, but the corrections themselves are based on inconsistent state representations. This creates a cycle where error detection mechanisms trigger revisions, but the revisions themselves introduce new inconsistencies, preventing the agent from converging to a stable analysis state.

This case study demonstrates how trace-level metrics reveal the internal reasoning characteristics that cause observable errors, providing actionable insights for improving LLM reasoning capabilities beyond simple performance rankings.

6 Related Work

Reasoning Benchmarks for LLM Agents. A growing body of work evaluates LLM-based agents in interactive environments. Benchmarks such as SWE-bench (Jimenez et al., 2023; Miserendino et al., 2025) and AgentBench (Liu et al., 2023b) focus on task completion in software engineering or multi-task settings. More general agent benchmarks (Liu et al., 2023b; Xu et al., 2024a; Song et al., 2024; Xie et al., 2024; Rawles et al., 2024; Trivedi et al., 2024; Snell and Abdulhaim, 2022; Barres et al., 2025; Yoran et al., 2024; Wu et al., 2025; Wei et al., 2025; Chen et al., 2025b) evaluate agents across diverse environments and tasks, typically reporting success rates or completion scores.

While these benchmarks provide broad coverage, they primarily emphasize whether agents succeed, rather than how they reason over long interaction horizons. Most settings allow agents to recover from mistakes through retries, replanning, or resampling, and do not explicitly model irreversible errors or long-term state drift. As a result, they are not designed to capture how reasoning quality evolves across extended sequences of interdependent decisions.

Code generation benchmarks such as HumanEval (Chen et al., 2021) and MBPP, as well as programming agent benchmarks (Yang et al., 2023; Bytedance-Seed-Foundation-Code-Team et al., 2025; Huang et al., 2024; Liu et al., 2025a; Jain et al., 2024; Zhang et al., 2024; Feng et al., 2024; Bogomolov et al., 2024; Lee et al., 2024; Yang et al., 2024), evaluate correctness, coverage, or pass@k metrics. General AI benchmarks (Mialon et al., 2023; Rein et al., 2023; Wang et al., 2024; Hendrycks et al., 2021; Chen et al., 2025a) similarly focus on domain-specific problem-solving accuracy. These benchmarks abstract away the internal reasoning process and collapse multi-step reasoning into outcome-level scores.

In contrast, our benchmark targets long-horizon, tool-grounded reasoning under partial observability and irreversible actions. Our work complements prior benchmarks by focusing on reasoning stability and error evolution across extended interaction traces.

LLM-assisted Binary Analysis. LLMs have been applied to binary analysis and vulnerability discovery (Pearce et al., 2023; Deng et al., 2023; Liu et al., 2025b). Existing evaluations typically measure vulnerability detection accuracy, exploit success rates, or code coverage (Alam et al., 2024; Chauvin, 2024; Zibaeirad and Vieira, 2024; Yong et al., 2025). Related efforts on binary dataset construction (Liu et al., 2024) focus on data collection rather than reasoning evaluation.

Our benchmark differs by treating binary analysis as a setting for studying long-horizon reasoning rather than as a domain-specific detection task. By analyzing full reasoning traces, we examine how agents explore binaries, maintain state, and propagate errors over time. These properties are not captured by existing binary analysis benchmarks but are relevant to other tool-grounded reasoning domains.

Anti-Shortcut Benchmarks. Recent work highlights the importance of benchmarks that reduce shortcut learning. Examples include code evaluation benchmarks designed to mitigate dataset contamination (Zheng et al., 2024; Xu et al., 2024b), tool-use benchmarks that restrict direct answer access (Li et al., 2023), and retrieval-augmented generation benchmarks that evaluate retrieval quality (Chen et al., 2024; Friel et al., 2024).

Most anti-shortcut benchmarks focus on preventing specific forms of shortcut behavior. In contrast, our benchmark enforces constraints at the execution level: tool actions are observable, observations are causally linked to actions, and early mistakes affect future reasoning. This design limits the ability to bypass reasoning through superficial patterns and complements existing anti-shortcut evaluation efforts.

7 Conclusion

We present a trace-level benchmark for evaluating long-horizon reasoning in LLM-assisted binary analysis. The benchmark includes 520 real-world binaries with reference execution traces, together with metrics that characterize how agents explore, commit to hypotheses, and manage context over extended tool-mediated interactions. By analyzing complete execution traces rather than final success or failure, the benchmark enables systematic diagnosis of reasoning behaviors—such as premature commitment, state drift, and context loss—that are hidden by scalar performance metrics.

Our results show that current agent methods struggle to maintain stable internal state under partial observability and irreversible actions, and that common control mechanisms provide only limited improvements.

Limitations

While BIN-BENCH helps us understand how LLM agents reason through long binary analysis tasks, there are several limitations to keep in mind.

Dataset. We tested agents on 520 binary files, all taken from embedded devices like routers and network equipment. These are real-world binaries, but they may not represent all types of programs that agents might need to analyze. For example, we did not include desktop applications, mobile apps, or server programs. Also, all our binaries are "stripped," meaning they have no debug information (like function names or variable names). This is common in real analysis, but not always the case.

Metrics. Our metrics look at how agents reason step by step, but they don't directly tell us whether agents found all the vulnerabilities or understood the program correctly. Instead, we focus on whether agents remember information, stay consistent, and follow coherent strategies. This helps us see where reasoning breaks down, but it might not perfectly match whether agents actually succeed at the task. Some of our metrics also use rules and thresholds that we chose (like deciding what counts as a "hypothesis" or an "exploration phase"). These choices could be improved with more validation. Future work could compare our metrics against expert human analysis to see if they really capture reasoning quality.

Agents and LLMs. We only tested four different agent methods (ReAct and three variations) with five different LLMs. There are many other ways to build agents that we did not try, such as having multiple agents work together, or using specialized reasoning strategies. We also only used general-purpose LLMs, not models that were specifically trained for binary analysis. Additionally, we used fixed instructions (prompts) for all experiments. Different ways of writing instructions might lead to different results.

Toolset. Agents in our benchmark can only use two tools: r2 and Ghidra. These are standard binary analysis tools, but there are other tools available (like IDA Pro or Binary Ninja) that we did not include. Using different tools might change how agents reason or what they can discover. We also assume that agents always get the full output from tools. In practice, tool outputs might be cut off or filtered, which could make reasoning harder.

Applicability. Binary analysis has some special characteristics: it requires reasoning over many steps, agents only see part of the program at a time, and early mistakes are hard to fix. These same challenges appear in other tasks where agents use tools over long periods. However, our specific findings might not apply to all such tasks. Future work could test whether similar patterns appear in other domains, like controlling autonomous vehicles.

References

Md Tanvirul Alam, Dipkamal Bhusal, Le Nguyen, and Nidhi Rastogi. 2024. Ctibench: A benchmark for evaluating llms in cyber threat intelligence. *Advances in Neural Information Processing Systems*, 37:50805–50825.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. 2025. τ^2-bench: Evaluating conversational agents in a dual-control environment. *Preprint*, arXiv:2506.07982.

Egor Bogomolov, Aleksandra Eliseeva, Timur Galimzyanov, Evgeniy Glukhov, Anton Shapkin, Maria Tigina, Yaroslav Golubev, Alexander Kovrigin, Arie Van Deursen, Maliheh Izadi, and 1 others. 2024. Long code arena: a set of benchmarks for long-context code models. *arXiv preprint arXiv:2406.11612*.

Bytedance-Seed-Foundation-Code-Team, :, Yao Cheng, Jianfeng Chen, Jie Chen, Li Chen, Liyu Chen, Wentao Chen, Zhengyu Chen, Shijie Geng, Aoyan Li, Bo Li, Bowen Li, Linyi Li, Boyi Liu, Jiaheng Liu, Kaibo Liu, Qi Liu, Shukai Liu, and 37 others. 2025. Fullstack bench: Evaluating llms as full stack coders. *Preprint*, arXiv:2412.00535.

Timothee Chauvin. 2024. eyeballvul: a future-proof benchmark for vulnerability detection in the wild. *arXiv preprint arXiv:2407.08708*.

Hanjie Chen, Zhouxiang Fang, Yash Singla, and Mark Dredze. 2025a. Benchmarking large language models on answering and explaining challenging medical questions. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3563–3599.

Jiawei Chen, Hongyu Lin, Xianpei Han, and Le Sun. 2024. Benchmarking large language models in retrieval-augmented generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17754–17762.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, and 1 others. 2025b. Browsecomp-plus: A more fair and transparent evaluation benchmark of deep-research agent. *arXiv preprint arXiv:2508.06600*.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, pages 423–435.

Jia Feng, Jiachen Liu, Cuiyun Gao, Chun Yong Chong, Chaozheng Wang, Shan Gao, and Xin Xia. 2024. Complexcodeeval: A benchmark for evaluating large code models on more complex code. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1895–1906.

Robert Friel, Masha Belyi, and Atindriyo Sanyal. 2024. Ragbench: Explainable benchmark for retrieval-augmented generation systems. *arXiv preprint arXiv:2407.11005*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *Preprint*, arXiv:2103.03874.

Yiming Huang, Jianwen Luo, Yan Yu, Yitong Zhang, Fangyu Lei, Yifan Wei, Shizhu He, Lifu Huang, Xiao Liu, Jun Zhao, and Kang Liu. 2024. Da-code: Agent data science code generation benchmark for large language models. *Preprint*, arXiv:2410.07331.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Hokyung Lee, Sumanyu Sharma, and Bing Hu. 2024. Bug in the code stack: Can llms find bugs in large python code stacks. *arXiv preprint arXiv:2406.15325*.

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. *Preprint*, arXiv:2304.08244.

Chang Liu, Rebecca Saul, Yihao Sun, Edward Raff, Maya Fuchs, Townsend Southard Pantano, James Holt, and Kristopher Micinski. 2024. Assemblage: Automatic binary dataset construction for machine learning. *Advances in Neural Information Processing Systems*, 37:58698–58715.

Hao Liu, Carmelo Sferrazza, and Pieter Abbeel. 2023a. Languages are rewards: Hindsight finetuning using human feedback. *arXiv preprint arXiv:2302.02676*.

Kaiyuan Liu, Youcheng Pan, Yang Xiang, Daojing He, Jing Li, Yexing Du, and Tianrun Gao. 2025a. Projecteval: A benchmark for programming agents automated evaluation on project-level code generation. *Preprint*, arXiv:2503.07010.

Puzhuo Liu, Chengnian Sun, Yaowen Zheng, Xuan Feng, Chuan Qin, Yuncheng Wang, Zhenyang Xu, Zhi Li, Peng Di, Yu Jiang, and 1 others. 2025b. Llm-powered static binary taint analysis. *ACM Transactions on Software Engineering and Methodology*, 34(3):1–36.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, and 3 others. 2023b. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv: 2308.03688*.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2023. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*.

Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. 2025. Swe-lancer: Can frontier llms earn $1 million from real-world freelance software engineering? *arXiv preprint arXiv:2502.12115*.

Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356. IEEE.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy Lillicrap, and Oriana Riva. 2024. Androidworld: A dynamic benchmarking environment for autonomous agents. *Preprint*, arXiv:2405.14573.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. 2023. Gpqa: A graduate-level google-proof qa benchmark. *Preprint*, arXiv:2311.12022.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Charlie Snell and Abdul Abdulhaim. 2022. Lmrl gym: Benchmarks for multi-turn reinforcement learning with language models. https://github.com/abdulhaim/LMRL-Gym.

Yifan Song, Weimin Xiong, Xiutian Zhao, Dawei Zhu, Wenhao Wu, Ke Wang, Cheng Li, Wei Peng, and Sujian Li. 2024. Agentbank: Towards generalized llm agents via fine-tuning on 50000+ interaction trajectories. *Preprint*, arXiv:2410.07706.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. 2024. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *ACL*.

Xiaoxuan Wang, Ziniu Hu, Pan Lu, Yanqiao Zhu, Jieyu Zhang, Satyen Subramaniam, Arjun R. Loomba, Shichang Zhang, Yizhou Sun, and Wei Wang. 2024. Scibench: Evaluating college-level scientific problem-solving abilities of large language models. *Preprint*, arXiv:2307.10635.

Jason Wei, Zhiqing Sun, Spencer Papay, Scott McKinney, Jeffrey Han, Isa Fulford, Hyung Won Chung, Alex Tachard Passos, William Fedus, and Amelia Glaese. 2025. Browsecomp: A simple yet challenging benchmark for browsing agents. *Preprint*, arXiv:2504.12516.

Jialong Wu, Wenbiao Yin, Yong Jiang, Zhenglin Wang, Zekun Xi, Runnan Fang, Linhai Zhang, Yulan He, Deyu Zhou, Pengjun Xie, and Fei Huang. 2025. Webwalker: Benchmarking llms in web traversal. *Preprint*, arXiv:2501.07572.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. 2024. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Preprint*, arXiv:2404.07972.

Frank F. Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z. Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, Mingyang Yang, Hao Yang Lu, Amaad Martin, Zhe Su, Leander Maben, Raj Mehta, Wayne Chi, Lawrence Jang, Yiqing Xie, and 2 others. 2024a. Theagentcompany: Benchmarking llm agents on consequential real world tasks. *Preprint*, arXiv:2412.14161.

Zhao Xu, Fan Liu, and Hao Liu. 2024b. Bag of tricks: Benchmarking of jailbreak attacks on llms. *Advances in Neural Information Processing Systems*, 37:32219–32250.

Cheng Yang, Chufan Shi, Yaxin Liu, Bo Shui, Junjie Wang, Mohan Jing, Linran Xu, Xinyu Zhu, Siheng Li, Yuxiang Zhang, and 1 others. 2024. Chartmimic: Evaluating lmm's cross-modal reasoning capability via chart-to-code generation. *arXiv preprint arXiv:2406.09961*.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2023. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *Advances in Neural Information Processing Systems*, 36:23826–23854.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Javier Yong, Haokai Ma, Yunshan Ma, Anis Yusof, Zhenkai Liang, and Ee-Chien Chang. 2025. Attackseqbench: Benchmarking large language models' understanding of sequential patterns in cyber attacks. *arXiv preprint arXiv:2503.03170*.

Ori Yoran, Samuel Joseph Amouyal, Chaitanya Malaviya, Ben Bogin, Ofir Press, and Jonathan Berant. 2024. Assistantbench: Can web agents solve realistic and time-consuming tasks? *Preprint*, arXiv:2407.15711.

Shudan Zhang, Hanlin Zhao, Xiao Liu, Qinkai Zheng, Zehan Qi, Xiaotao Gu, Xiaohan Zhang, Yuxiao Dong, and Jie Tang. 2024. Naturalcodebench: Examining coding performance mismatch on humaneval and natural user prompts. *arXiv preprint arXiv:2405.04520*.

Jiasheng Zheng, Boxi Cao, Zhengzhao Ma, Ruotong Pan, Hongyu Lin, Yaojie Lu, Xianpei Han, and Le Sun. 2024. Beyond correctness: Benchmarking multi-dimensional code generation for large language models. *arXiv preprint arXiv:2407.11470*.

Arastoo Zibaeirad and Marco Vieira. 2024. Vulnllmeval: A framework for evaluating large language models in software vulnerability detection and patching. *arXiv preprint arXiv:2409.10756*.

A Appendix: Benchmark Comparison

Table 4 compares BIN-BENCH with representative agent benchmarks. Existing benchmarks such as SWE-bench and AgentBench primarily emphasize task completion in settings where agents have access to complete information or can recover from errors through re-sampling. In contrast, BIN-BENCH explicitly evaluates long-horizon reasoning under partial observability and irreversible errors. None of the existing benchmarks simultaneously cover all these dimensions, leaving a gap in evaluating reasoning stability over extended tool-based interactions. BIN-BENCH is designed to fill this gap.

Table 4: Capability coverage comparison across agent benchmarks. SWE-bench focuses on software engineering tasks; AgentBench and OSWorld evaluate general agent capabilities; GAIA tests general AI assistants; Intercode targets interactive coding.

Dimension	SWE-bench	AgentBench	OSWorld	GAIA	Bin-Bench
Long-horizon reasoning (>50 steps)	Not Supported	Partial	Partial	Partial	**Supported**
Tool-grounded state transitions	Partial	**Supported**	**Supported**	**Supported**	**Supported**
Stateful (non-textual) context	Not Supported	Not Supported	Not Supported	Not Supported	**Supported**
Partial observability	Not Supported	Partial	**Supported**	**Supported**	**Supported**
Irreversible errors	Not Supported	Not Supported	Not Supported	Not Supported	**Supported**

Component	Specification
Base Class	`BaseAgent` (provides tool interface, context/history management)
LLM	the LLM client
Max Iterations	50 /100
Context Window	Full conversation history (no compression)
Error Handling	3 retries with error feedback, fallback on failure
Tool Interface	`Radare2Tool` (r2, Ghidra, r2ghidra plugin)
Output Format	Structured JSON (JSONL traces)

Table 5: Common Infrastructure and Configuration

B Appendix: Baseline Implementation Details

This appendix provides detailed implementation specifications for all baselines evaluated in our benchmark. Table 5 shows the common infrastructure and configuration for all baselines. Note: LLM hyperparameters (temperature, top_p, max_tokens) are identical for all baselines.

B.1 Output Schema

Table 6 compares output schemas and implementation mechanisms across all baselines.

B.2 Baseline-Specific Implementation Details

ReAct (Minimal Baseline): Standard reasoning-action-observation loop without additional mechanisms.

Execution Flow: (1) *Reasoning*: Generate `thought` describing observations and next action. (2) *Acting*: Select tool action or `finish`. (3) *Observing*: Receive tool output and proceed to next iteration.

ReAct + Reflexion: Augments ReAct with reflection mechanism via prompt injection after tool execution. *Schema*: `reflection` field defined but not in `required` array. *Enforcement*: Reflection prompt automatically injected into conversation history after each tool execution Reflection enforced through prompt injection (observable in traces); enables backtracking and hypothesis correction.

ReAct + Plan-and-Execute: Incorporates explicit planning phase before each action via required field in output schema. *Schema*: `plan` field included in `required` array. *Enforcement*: Schema validation level—missing plan triggers retry mechanism.

ReAct + Self-Correction: Adds error detection and correction mechanisms via system prompt instructions. *Trigger*: After each action. Agent must detect errors in tool selection, result interpretation, reasoning logic, and assumptions. *Correction*: Explicit error identification and correction for subsequent actions

B.3 Error Handling

All baselines implement identical error handling, listed by Table 7.

B.4 Trace Format

All baselines produce JSONL traces with the following fields:

Trace Structure: User input → Assistant responses (with thought/plan/reflection) → Tool execution results → Reflection prompts (Reflexion) → Error feedback (on parsing failure) → Final response.

C Appendix: Metric Definitions

This appendix provides concrete definitions and computation procedures for all metrics used in our evaluation. All metrics are computed automatically from structured reasoning traces, without manual annotation.

C.1 Core Metrics

C.1.1 RAO Consistency

Definition. This metric measures whether an agent follows a coherent reasoning loop: whether stated intentions lead to appropriate actions, and whether

Field/Mechanism	ReAct	ReAct + Reflexion	ReAct + Plan-Execute	ReAct + Self-Correct
`thought`	Yes	Yes	Yes	Yes
`reflection`	No	Yes	No	No
`plan`	No	No	Yes	No
`action`	Yes	Yes	Yes	Yes
`action_input`	Yes	Yes	Yes	Yes
`status`	Yes	Yes	Yes	Yes
Enforcement	Schema	Prompt injection	Required field	System prompt
Trigger	N/A	After tool execution	Before each action	After each action

Table 6: Output Schema and Implementation Comparison.

Mechanism	Specification
Max Retries	3
Retry Trigger	JSON parsing failure
Error Feedback	Includes parsing error, raw response preview, schema reminder
Fallback	After 3 failures, terminates

Table 7: Error Handling Mechanism

observations from those actions are reflected in subsequent reasoning.

Computation. Each reasoning trace is segmented into consecutive cycles of `thought` $\rightarrow$ `action` $\rightarrow$ `result` $\rightarrow$ next `thought`. For each cycle, we apply the following checks: (1) whether the action targets the objective mentioned in the preceding `thought`; (2) whether the `result` contains information relevant to that action; (3) whether the next `thought` references or uses information from the `result`. Each check is recorded as a binary indicator. The consistency score is computed as the average of these indicators across all cycles in the trace.

Interpretation. High scores indicate that agents consistently translate intentions into actions and incorporate observations into later reasoning. Low scores indicate breaks in the reasoning loop, such as actions that do not follow stated goals or observations that are ignored.

C.1.2 Dependency Distance

Definition. This metric measures how far back in a trace an agent can reuse previously observed information when taking later actions.

Computation. For each action that refers to a specific program property (e.g., function address, architecture detail), we locate the earliest prior `result` in which that property first appears. The dependency distance is defined as the number of steps between the current action and that earlier observation. We report the maximum dependency distance observed in a trace, as well as the distribution of distances across all such references.

Interpretation. Larger distances indicate that agents retain and reuse information over longer horizons. Smaller distances suggest reliance on only recent observations.

C.1.3 Phase Structure

Definition. This metric captures whether an agent's exploration follows a clear progression of analysis phases rather than frequent, unstructured switching.

Computation. We assign each command to a semantic category based on its function: reconnaissance (e.g., metadata queries), navigation (e.g., seeking addresses), decompilation, cross-reference analysis, and inspection. Using a sliding window of w steps (default $w = 10$), we identify the dominant category in each window. A phase transition is recorded when the dominant category changes between adjacent windows. We compute the total number of transitions and the average duration of each phase.

Interpretation. Traces with fewer transitions and longer phase durations indicate more structured exploration. Frequent transitions indicate unstable or unfocused behavior.

C.1.4 Hypothesis Persistence

Definition. This metric measures whether agents form hypotheses during analysis and whether those hypotheses guide behavior over multiple steps.

Computation. We detect hypothesis statements in `thought` fields using simple lexical cues (e.g., "likely", "suspect", "assume"). For each detected hypothesis, we record the step where it first appears and track how long it remains active, defined as being referenced again in later `thought` fields or implicitly guiding subsequent actions. We report the distribution of hypothesis lifetimes across the trace.

Interpretation. Longer hypothesis persistence indicates sustained strategic reasoning. Short-lived or frequently abandoned hypotheses indicate unstable reasoning or difficulty integrating new observations.

C.1.5 Error Propagation

Definition. This metric measures how errors introduced during reasoning or tool use affect later steps, and whether agents detect and correct those errors.

Computation. We identify error events such as invalid tool parameters, misinterpretation of tool outputs, or incorrect inferences. For each error, we track whether the agent explicitly acknowledges or corrects it in later `thought` fields or actions. Error propagation distance is defined as the number of steps between the introduction of an error and its correction, or until the trace ends if it is never corrected.

Interpretation. Short propagation distances indicate effective error detection and recovery. Long distances indicate that errors persist and influence later reasoning, leading to compounding failures.

C.2 Implementation Notes

All metrics are computed using deterministic, rule-based procedures over structured traces. No model inference or manual annotation is required during evaluation. The full implementation, including parsing rules and aggregation logic, will be released with the benchmark.

D Appendix: Design Implications

Table 8 maps observed patterns from trace-level metrics to concrete agent design decisions, illustrating how metric signals translate into actionable system interventions.

E Appendix: Trace Analysis Experiments

Trace Length Distribution. Figure 2 shows the distribution of trace lengths (reasoning steps) across all 520 binary analysis tasks for each baseline method. The trace length directly reflects how many tool invocations agents require to complete binary analysis, providing insight into exploration efficiency and reasoning complexity.

Across all baselines, trace lengths exhibit substantial variation, with median values ranging from 42 to 50 steps and means ranging from 45.3 to 55.8 steps. ReAct achieves the highest median trace length (49.5 steps, mean: 55.8±31.2), indicating that the minimal baseline requires more steps to complete analysis tasks. ReAct+Self-Correction shows the lowest median trace length (42.0 steps, mean: 45.3±20.5), suggesting that explicit error detection and correction mechanisms enable more efficient exploration. However, this efficiency comes at a cost: ReAct+Self-Correction successfully completes only 375 out of 520 tasks (72.1%), compared to ReAct's 520 completions (100%), indicating that self-correction may prematurely terminate analysis on complex binaries.

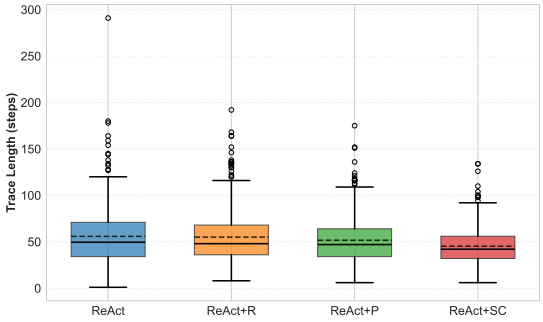

Figure 2: Distribution of trace lengths (reasoning steps) across 520 binary analysis tasks for each baseline method. Box plots show median (center line), quartiles (box edges), and outliers (points). Dashed lines indicate mean values. ReAct+Self-Correction shows the lowest median trace length (42.0 steps) but completes only 72.1% of tasks, while ReAct achieves 100% completion with a median of 49.5 steps.

The standard deviations (20.5–31.2 steps) reveal substantial variability in trace lengths across different binaries, reflecting the inherent diversity in binary complexity and analysis difficulty. This variability highlights that binary analysis is not a uniform task: some binaries require extensive exploration (up to 291 steps for ReAct), while others can be analyzed more efficiently (minimum 1–8 steps depending on baseline).

Notably, ReAct+Plan-and-Execute completes only 459 tasks (88.3%), with a median trace length of 47.0 steps (mean: 51.7±25.1). The early commitment to high-level plans may limit exploration flexibility, causing agents to abandon analysis when initial plans prove insufficient. In contrast, ReAct+Reflexion achieves near-complete coverage (519 tasks, 99.8%) with similar trace length statistics (median: 48.0, mean: 55.0±27.6), suggesting that periodic reflection helps agents adapt strategies without significantly increasing exploration steps.

Result Length Distribution. Figure 3 shows the

Table 8: Design implications derived from trace-level metric patterns. Each row maps a metric signal to its design implication and concrete system intervention.

Metric Signal	Related Metric	Design Implication	Actionable System Intervention
Early error onset	Error Propagation Analysis	Failures are planning-dominated rather than execution-induced	Decouple planner from executor; introduce plan validation or alternative plan sampling
High loop severity score	Error Propagation Analysis	Agent lacks loop awareness or termination criteria	Add loop detection, repetition penalties, or explicit state tracking
Low recovery success rate	Error Propagation Analysis	Errors are irreversible once triggered	Trigger external verification (tools, human-in-the-loop, oracle) upon failure signals
Frequent phase transitions	Exploration Phase Structure	Unstable decision trajectories indicate lack of strategic commitment	Introduce phase-aware planning or commitment mechanisms to stabilize exploration
Short hypothesis persistence	Hypothesis Formation and Persistence	Agents cannot maintain strategic focus over long horizons	Implement hypothesis tracking and refinement mechanisms to support long-term commitment
Low dependency distance	Long-Range Dependency Distance	Context retention degrades over long horizons	Enhance long-term memory mechanisms or introduce explicit state summarization
Low RAO consistency	Reasoning-Action-Observation Consistency	Agents fail to integrate observations into state updates	Strengthen observation processing and state update mechanisms

distribution of result lengths (observation sizes) received by agents across all baseline methods. This analysis directly captures the *partial observability* challenge: agents receive fragmented observations of varying sizes, where each tool invocation returns only a portion of the binary's information.

The result lengths exhibit extreme variation, ranging from 1 character to over 217 million characters, with median values between 721 and 971 characters. This massive variability reflects the fundamental nature of partial observability in binary analysis: different tool commands return vastly different amounts of information. For example, simple metadata queries (`i`) return small results (hundreds of characters), while function list commands (`afl`) can return millions of characters listing all functions in the binary. This variation forces agents to process and integrate observations of dramatically different scales, making it challenging to maintain consistent state representations across tool invocations.

Across baselines, result length distributions are relatively similar, with median values ranging from 721 (ReAct) to 971 (ReAct+Self-Correction) characters. However, the mean values show larger differences (ReAct: 37,540, ReAct+Self-Correction: 138,028), indicating that some baselines encounter more extremely large results. The substantial standard deviations (2.5–5.2 million characters) reflect the presence of outlier results that are orders of magnitude larger than typical observations. This extreme variability in observation sizes is a core characteristic of partial observability: agents cannot predict how much information each tool invocation will return, and must adapt their reasoning to handle both minimal and massive observations within the same analysis trace.

The log-scale distribution reveals that most results are relatively small (median around 700–1000 characters), but a small fraction of results are extremely large (millions of characters). This long-tail distribution means that agents must be prepared to process both concise metadata and extensive decompilation outputs, requiring flexible information extraction and state update mechanisms. The inability to predict result sizes in advance makes it difficult for agents to allocate attention or processing resources effectively, contributing to the challenges of maintaining coherent state representations over long reasoning horizons.

Figure 4 complements this analysis by showing the distribution of command types used across baselines. The top 8 command types account for the majority of tool invocations, with decompilation commands (`pdg`, `pd`, `pdf`) dominating across all baselines. The relatively consistent command type distributions suggest that reasoning strategies primarily differ in command sequencing rather than

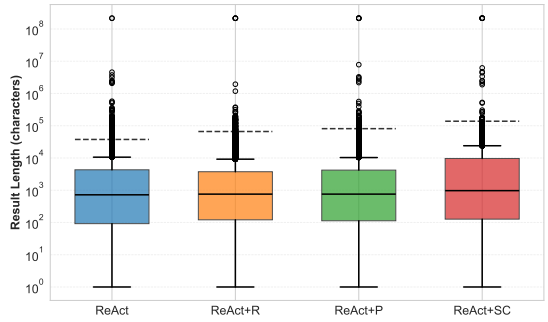

Figure 3: Distribution of result lengths (observation sizes in characters) across all baseline methods. Box plots show median (center line), quartiles (box edges), and outliers (points) on a log scale. Dashed lines indicate mean values. Result lengths range from 1 to over 217 million characters, with median values between 721 and 971 characters. This extreme variability in observation sizes directly captures the partial observability challenge: agents must process and integrate observations of dramatically different scales.

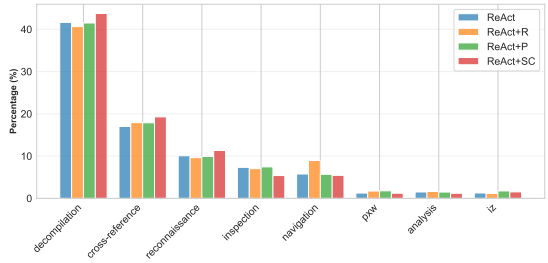

Figure 4: Distribution of the top 8 most frequently used command types across all baseline methods. Values represent the percentage of total commands for each baseline. Decompilation commands (`pdg`, `pd`, `pdf`) dominate across all baselines, followed by cross-reference commands (`axt`, `axf`). The relatively consistent distribution patterns across baselines suggest that reasoning strategies primarily differ in command sequencing rather than command type selection.

command type selection.

F Appendix: Supplementary Experiments

Vulnerability Discovery Statistics. Table 9 shows the distribution of vulnerabilities found across all 520 binary analysis tasks using the curated analysis method (DeepSeek-V3 with expert-guided strategies). The analysis successfully identified vulnerabilities in 212 out of 520 binaries (40.8%), with the majority containing 1–3 exploitable vulnerabilities. Notably, 120 binaries (23.1%) contain exactly one vulnerability, while 47 binaries (9.0%) contain two vulnerabilities. The distribution reveals that vulnerability discovery is not uniform across binaries: some binaries contain multiple exploitable vulnerabilities (up to 15 in the case of `param_convert`), while others contain none.

Table 9: Distribution of vulnerability counts across 520 binary analysis tasks. Each binary was analyzed with curated analysis strategies.

Vulnerabilities	Number of Binaries	Percentage
0	308	59.2%
1	120	23.1%
2	47	9.0%
3	16	3.1%
4+	29	5.6%

The high proportion of binaries with zero vulnerabilities (308, 59.2%) reflects the diversity of the benchmark corpus, which includes both security-critical components (network services, authentication handlers) and utility programs (file system tools, network utilities). This distribution highlights the importance of trace-level evaluation metrics: binary outcome-based evaluation (vulnerability found or not) would miss the substantial variation in reasoning quality and analysis depth across different binaries.

Reasoning-Action-Observation Consistency Sub-metrics. Figure 5 shows the average performance of each baseline method across the four sub-metrics that compose the Reasoning-Action-Observation Consistency metric: Thought-Action Alignment (TA), Action-Result Relevance (AR), Result-Thought Integration (RT), and Cycle Coherence (CC).

The comparison reveals distinct patterns across baselines. ReAct+Reflexion achieves the highest average scores in Thought-Action Alignment (0.72) and Cycle Coherence (0.65), suggesting that periodic reflection helps agents maintain better alignment between reasoning and actions, and improves overall cycle coherence. ReAct+Plan-and-Execute shows strong performance in Action-Result Relevance (0.68), indicating that structured planning helps agents select actions that are more relevant to the observed results. However, all baselines struggle with Result-Thought Integration (RT), with average scores ranging from 0.30 to 0.38, indicating that agents across all strategies have difficulty incorporating observation results into subsequent reasoning steps.

Notably, ReAct+Self-Correction shows the lowest average scores across most sub-metrics, particu-

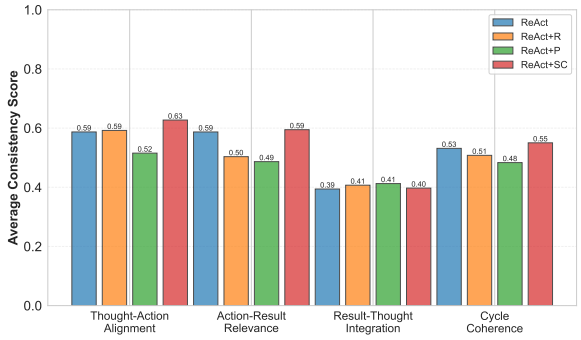

Figure 5: Average performance of each baseline method across the four sub-metrics of Reasoning-Action-Observation Consistency. TA: Thought-Action Alignment; AR: Action-Result Relevance; RT: Result-Thought Integration; CC: Cycle Coherence. ReAct+Reflexion achieves the highest scores in TA and CC, while all baselines struggle with RT (0.30–0.38), indicating a fundamental challenge in incorporating observations into subsequent reasoning.

larly in Thought-Action Alignment (0.58) and Cycle Coherence (0.52). This suggests that explicit error correction mechanisms may disrupt the natural flow of reasoning-action cycles, potentially causing agents to over-correct or lose coherence when attempting to fix errors. The relatively consistent low performance in Result-Thought Integration across all baselines highlights a fundamental challenge in long-horizon reasoning: maintaining continuity between observations and subsequent reasoning, especially when observations are fragmented and partial.

These sub-metric patterns align with the overall consistency scores reported in the main experimental results, where ReAct+Reflexion achieves the highest average consistency (0.54), followed by ReAct (0.53), ReAct+Plan-and-Execute (0.52), and ReAct+Self-Correction (0.48). The decomposition into sub-metrics provides insight into the specific aspects of the RAO cycle where different strategies excel or struggle, informing future improvements to reasoning mechanisms.

Hypothesis Formation and Persistence. Figure 6 shows the average performance of each baseline method across three key metrics of Hypothesis Formation and Persistence: Formation Frequency (the proportion of reasoning steps that form hypotheses), Average Persistence (the average duration of hypothesis clusters), and Stability Score (a composite measure of hypothesis stability).

The comparison reveals distinct patterns across baselines. ReAct+Plan-and-Execute achieves the highest Stability Score (1.35) and Average Persistence (3.53), indicating that structured planning helps agents form more stable and persistent hypotheses. This aligns with the planning strategy's emphasis on committing to high-level plans, which naturally leads to more persistent hypothesis clusters. However, ReAct+Plan-and-Execute shows moderate Formation Frequency (0.65), suggesting that while hypotheses are more stable once formed, they are not necessarily formed more frequently.

ReAct shows balanced performance across all three metrics (Formation Frequency: 0.59, Average Persistence: 3.08, Stability Score: 1.11), indicating a baseline level of hypothesis formation and persistence without explicit mechanisms for enhancement or disruption. ReAct+Reflexion achieves the second-highest Formation Frequency (0.65) but lower Average Persistence (1.81) and Stability Score (0.41), suggesting that periodic reflection may encourage more frequent hypothesis formation but at the cost of stability, as reflection may cause agents to abandon hypotheses prematurely.

Notably, ReAct+Self-Correction shows the highest Formation Frequency (0.69) but the lowest Average Persistence (1.01) and Stability Score (0.15). This pattern indicates that explicit error correction mechanisms lead to frequent hypothesis formation but very low persistence, as agents constantly revise and abandon hypotheses when attempting to correct errors. This tradeoff highlights a fundamental tension in long-horizon reasoning: high flexibility (frequent hypothesis formation and revision) comes at the cost of low stability (short-lived hypotheses), while high stability (persistent hypotheses) may limit adaptability when initial hypotheses are incorrect.

These patterns align with the semantic-dependent nature of errors discussed in the main text: the same behavior (frequent hypothesis revision) can be interpreted as either adaptive (correcting errors) or disruptive (losing strategic commitment), depending on the reasoning context and goals.

Exploration Phase Structure. Figure 7 shows the average performance of each baseline method across three key metrics of Exploration Phase Structure: Transition Frequency (the number of phase transitions per trace), Oscillation Score (a measure of back-and-forth transitions between phases), and Clear Progression Percentage (the proportion of traces that exhibit clear forward progression

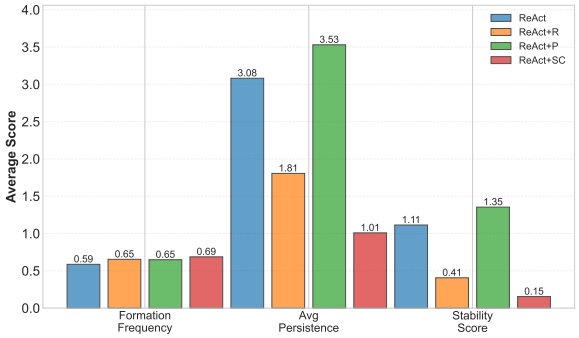

Figure 6: Average performance of each baseline method across three key metrics of Hypothesis Formation and Persistence. Formation Frequency: proportion of reasoning steps that form hypotheses; Avg Persistence: average duration of hypothesis clusters; Stability Score: composite measure of hypothesis stability. ReAct+Plan-and-Execute achieves the highest Stability Score (1.35) and Average Persistence (3.53), while ReAct+Self-Correction shows the highest Formation Frequency (0.69) but the lowest Stability Score (0.15), revealing a tradeoff between flexibility and stability.

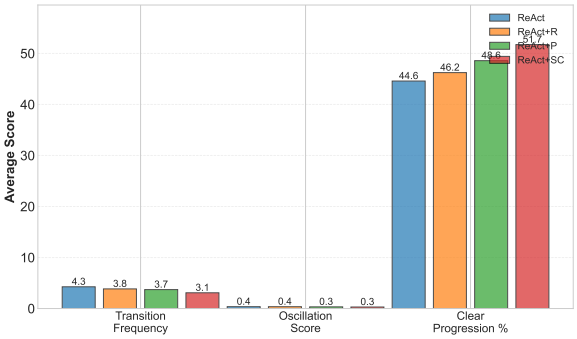

Figure 7: Average performance of each baseline method across three key metrics of Exploration Phase Structure. Transition Frequency: number of phase transitions per trace; Oscillation Score: measure of back-and-forth transitions; Clear Progression ReAct shows the highest Transition Frequency (4.26) and Oscillation Score (0.36), while ReAct+Self-Correction achieves the lowest Transition Frequency (3.09) and Oscillation Score (0.30) with the highest Clear Progression Percentage (51.7%), indicating that error correction mechanisms help maintain more stable exploration phases.

through exploration phases).

The comparison reveals that ReAct shows the highest Transition Frequency (4.26) and Oscillation Score (0.36), indicating that the minimal baseline exhibits more frequent phase transitions and higher oscillation between exploration phases. This suggests that without explicit mechanisms for maintaining phase coherence, agents tend to switch between exploration phases more frequently, potentially leading to less structured exploration patterns.

ReAct+Self-Correction achieves the lowest Transition Frequency (3.09) and Oscillation Score (0.30), while maintaining the highest Clear Progression Percentage (51.7%). This pattern indicates that explicit error correction mechanisms help agents maintain more stable exploration phases with fewer oscillations, leading to clearer progression through the exploration process. However, the Clear Progression Percentage remains relatively low across all baselines (44.6%–51.7%), suggesting that maintaining clear forward progression is challenging even with explicit mechanisms.

ReAct+Plan-and-Execute shows moderate Transition Frequency (3.71) and Oscillation Score (0.33), with Clear Progression Percentage (48.6%) slightly below ReAct+Self-Correction. This suggests that structured planning helps reduce phase transitions and oscillations compared to the minimal baseline, but does not achieve the same level of phase stability as self-correction mechanisms.

ReAct+Reflexion shows similar patterns to ReAct+Plan-and-Execute (Transition Frequency: 3.85, Oscillation Score: 0.36, Clear Progression: 46.2%), indicating that periodic reflection provides moderate benefits in phase structure but does not fundamentally alter exploration patterns compared to planning-based approaches.

These patterns align with the overall findings that explicit control mechanisms (planning, reflection, self-correction) provide modest improvements in exploration structure, but the benefits are limited and vary across different aspects of the exploration process. The relatively low Clear Progression Percentage across all baselines highlights a fundamental challenge in long-horizon binary analysis: maintaining coherent exploration strategies over extended reasoning sequences, especially when dealing with partial observability and complex program structures.