

AgentTAC: Mining Temporal Action-Context Error Patterns from Agent Traces

Chenchen Zhao^{1,2}, Muxi Chen^{1,2}, Jianhai Shu¹, Zhaohua Zhang², Wenyu Jiang²,
Tianwen Jiang², Hexuan Deng², Sa Zhu², Haolin Yang², Geyuan Zhu²,
Cheng Yang², Qiuyong Xiao², Jianhuan Zhuo^{2*}, Jihong Zhang^{2*}, Qiang Xu^{1,3*},

¹The Chinese University of Hong Kong ²Tencent Hunyuan AI Data Department

³Shenzhen Loop Area Institute

Abstract

Interactive agents increasingly fail on long task processes involving user turns, tool calls, and evolving environment contexts. Existing agent debugging methods provide useful trace-level diagnosis, but generally return explanations for individual failed traces rather than recurring, fine-grained action-context patterns across runs. To offer such corpus-level analyses, we present AgentTAC (Temporal Action-Context), an automated framework for mining erroneous and generalizable *temporal sequences* of action and context properties from agent traces. AgentTAC first localizes all distinct errors in each involved agent run. Then, it constructs and maintains a dynamic pool of attributes over the involved tasks, actions, and contexts to summarize their intrinsic properties and contextual relationships. Finally, it extracts temporally contiguous attribute patterns anchored at the localized errors. The resulting error patterns describe failures as interpretable sequential combinations of context and actions, such as making decisions under a missing-constraint context or taking an action after an incompatible prior decision. Extensive evaluations of AgentTAC on multiple benchmarks and agents validate its capabilities in error localization and error pattern mining, and provide preliminary evidence for error-pattern-driven test-time repair.

1 Introduction

Language model agents increasingly operate as long-horizon systems: they converse with users, call tools, update external contexts, and decide when a task is complete (Yao et al., 2024; Zhou et al., 2024; Liu et al., 2024; Lu et al., 2025). As these systems move from single-turn prediction to interactive task execution, developers need debugging methods that explain not only whether an

agent fails, but how its behavior over time leads to the final failure.

Recent works (Barke et al., 2026; Deshpande et al., 2025; Zhu et al., 2025; Zhang et al., 2025; Wang et al., 2026; Liu et al., 2026; Yang et al., 2026) have made substantial progress on agent failure diagnosis. They identify and categorize erroneous or critical steps within a trace; some focus on a root-cause step, while TRAIL can identify multiple errors through trace-level prompting. Their primary output nevertheless remains a diagnosis of an individual trace. This is useful for incident analysis, but does not directly reveal failure patterns that recur across tasks and agent runs. Moreover, methods centered on the earliest critical error have low coverage when a trace contains multiple independent errors, while a vanilla trace-level prompt can conflate newly introduced errors with their downstream propagation. Many failures are also better characterized by temporal combinations of agent actions and environment contexts than by an isolated step.

In this paper, we instead ask which sequences of actions and contexts recur across multiple erroneous agent runs. This shifts the analysis from per-trace diagnosis to scenario- and agent-level debugging that can support monitoring, repair, and agent redesign. To support this goal, we propose AgentTAC (Temporal Action-Context), an automated framework for mining recurring temporal error patterns from multiple agent traces. AgentTAC has three core stages. First, it localizes all distinct errors in each failed trace and uses them as anchors, rather than restricting analysis to the earliest error occurrence. Second, it represents traces with attributes characterizing its task, actions, and contexts, so that both what the agent does and the context in which it acts can be analyzed. Third, it mines temporally contiguous attribute patterns ending at the localized errors to reflect erroneous action-context sequences by the agent in task pro-

*Corresponding authors: tomzhuo@tencent.com, jihongzhang@tencent.com, qxu@cse.cuhk.edu.hk

Agent	Error Pattern Description	Failure Rate	Occurrence
Gemini	When the current state contains a contradiction between the requested booking and the actual passenger details, the agent proceeds to translate the contextual information into formatted parameters for an external tool call.	1.00	12
Qwen	In tasks that require considering multiple nearby airports or flexible geographic regions, the agent deduces viable workarounds or alternative options to present in the final response after concluding the primary request cannot be fulfilled.	1.00	10
Claude	The agent performs a step that calculates an aggregate cost by combining financial components, followed by a step that provides a response without invoking any external tools or systems.	0.50	16
DeepSeek	In tasks involving flexible user preferences such as “prefer morning flights” or “cheapest option”, the agent performs a step where it reasons to translate implicit user context, relative dates, or natural language constraints into specific, formatted parameters required for an external tool call.	1.00	13

Table 1: Representative error patterns extracted by AgentTAC under the `airline` task category in τ^2 -Bench.

cessing. The resulting error patterns are highly interpretable, with examples shown in Table 1. To the best of our knowledge, AgentTAC is the first pipeline for temporal agent error pattern mining.

We evaluate AgentTAC on τ^2 -Bench and VitaBench with multiple agents. Results validate that AgentTAC has the capability of mining high-coverage temporal error patterns in traces, and also provides preliminary evidence that these patterns can support agent repair at test time. The intended generalization unit is a target task domain: patterns are induced from many tasks and traces in that domain and evaluated on held-out traces, rather than assumed to transfer unchanged across unrelated APIs, policies, and environments.

Our contributions are threefold:

- We formulate a frequently neglected problem in agent debugging: a single trace may contain multiple independent errors, making earliest-error-only analyses insufficient for discovering recurring failure mechanisms.
- We introduce temporal action-context analysis as a new perspective for agent debugging, combining task, agent action, and environment context attributes to mine erroneous temporal sequences in traces.
- We present AgentTAC, an automated framework for temporal action-context error pattern mining, and demonstrate its effectiveness through extensive experiments.

2 Related Work

2.1 Agent Trace Diagnosis

Agent failures are increasingly studied through execution trajectories rather than final answers alone.

AgentRx (Barke et al., 2026) validates each step through synthesized constraints before an LLM judge localizes the earliest critical error. Agent-Debug (Zhu et al., 2025) similarly isolates a root-cause failure and generates corrective feedback. TRAIL (Deshpande et al., 2025), in contrast, uses a trace-level LLM prompt and may identify multiple erroneous steps. These systems provide useful per-trace diagnoses, but do not aggregate their outputs into recurrent temporal patterns across traces. In addition, a vanilla trace-level prompt does not explicitly separate newly introduced errors from propagated consequences, a distinction central to our localization stage.

AgentTAC differs primarily in goal. Rather than stopping at the diagnosis of a single failed trajectory, it aggregates traces and mines recurring temporal error patterns. Its localizer explicitly distinguishes new errors from propagated ones and uses all distinct new errors as pattern anchors. In addition, AgentTAC keeps the original agent workflow intact, and does not require agents to follow a prescribed diagnostic schema or expose hidden intermediate results (Zhu et al., 2025; Liu et al., 2026).

2.2 Benchmarks for Agent Evaluation

Interactive agent benchmarks provide the empirical basis for studying long-horizon failures. τ -Bench (Yao et al., 2024) and τ^2 -Bench (Barres et al., 2025) evaluate tool-agent-user interaction in realistic domains. VitaBench (He et al., 2025) emphasizes versatile real-world service tasks with multi-turn interaction and tool use. WebArena (Zhou et al., 2024) and ToolSandbox (Lu et al., 2025) further stress realistic web environments, stateful tool calls, and conversational execution. These benchmarks show that agent reliability cannot be understood

from final answers alone: failures often emerge through partial progress, delayed tool feedback, shifting user constraints, and state updates.

2.3 Error Pattern Discovery

Our pattern mining strategy is related to error slice discovery for model debugging. Domino (Eyuboglu et al., 2022) discovers systematic errors through embedding-space clusters; AdaTest (Ribeiro and Lundberg, 2022) and CheckList (Ribeiro et al., 2020) use human- or model-assisted behavioral tests; HiBug2 (Chen et al., 2025b) and FailureAtlas (Chen et al., 2025a) generate task-specific attributes and efficiently enumerates interpretable error slices.

AgentTAC inherits the attribution spirit of this line but extends it to sequential traces: patterns are ordered groups of task, action, and context attributes, not static sample attributes. The resulting attributes are more comprehensive, process-relevant, and interpretable than those generated by existing methods focusing only on input properties.

2.4 Sequential Pattern Mining

Our work is also connected to software engineering methods for finding failure-inducing program elements and recurring event patterns. AgentTAC shares the goal of extracting signal from executions, but differs in both representation and target. Program fault localization assumes instrumented code entities and test outcomes, whereas agent traces contain natural-language actions, tool outputs, and evolving environment contexts. Based on the fundamental spirit of sequential pattern mining, AgentTAC targets in-trace errors, and conducts agent debugging through interpretable attribute proposal and temporal error pattern enumeration.

3 AgentTAC: Cross-Trace Temporal Action-Context Error Pattern Mining

Given a collection of traces, AgentTAC conducts preprocessing, error localization, and trace component attribution sequentially. Then, it extracts temporal error patterns based on localized errors and generated attributes. Error patterns are contiguous sequences of agent actions and their surrounding environment contexts.

The current instantiation targets errors introduced by the agent’s own decisions or actions. External information - including user messages, tool results, and environment state - is represented as

context, so the same representation can express patterns involving missing or anomalous external signals when those signals are logged. Systematically detecting failures whose primary cause is an external malfunction, however, requires an external-anomaly-aware attribute inventory and is outside the present evaluation.

3.1 Trace Preprocessing

A trace is represented as a chain graph $G = (V, E)$. The task serves as the initial context v_0 ; each later node v_i is the context after an observed user or agent action e_i (as edge). The graph explicitly models context properties that are implicit in original traces, frequently neglected by prior methods, but crucial for comprehensive debugging.

To facilitate the subsequent attribution process, AgentTAC translates all structured content in the traces into natural language summaries. A summarizer rewrites each action in the traces into a natural text containing possible tool-calling actions, corresponding outcomes, and agent feedback. To sufficiently capture the properties of each environment context, AgentTAC summarizes all of its preceding trace segments as preliminary information for the subsequent context attribution. The summaries have two variants: a full version that preserves all available information, and a brief version that keeps only important content for lower token costs.

3.2 Distinct Error Localization

Failed traces often contain error propagation: after an early mistake, many later steps appear wrong as they inherit a corrupted context. An earliest-error-only view collapses it into the first unrecoverable point, which is insufficient for error pattern mining, as a single trace may contain several independent mistakes, as stated in Section 1.

To identify all distinct errors while avoiding this problem, AgentTAC proposes to localize both *newly introduced* errors and *propagated errors*. We first propose formal definitions of them:

- We treat error as a **context** property that may be jointly induced by the preceding sequence of actions and contexts, but is directly triggered by a specific agent action.
- Newly-introduced errors are errors that first emerge at the current context and are not covered by the existing error registry.

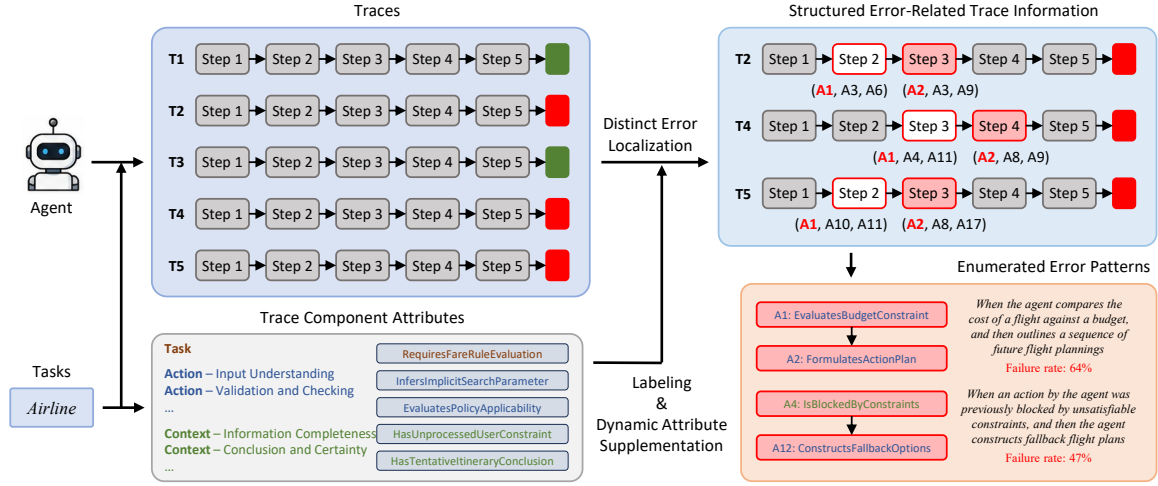


Figure 1: Overview of the AgentTAC workflow. AgentTAC first localizes distinct errors in agent traces (step blocks filled in red). Then, it constructs and maintains an attribute pool for error characterization. Finally, it extracts recurring sequences of attribute combinations as temporal error patterns.

- Propagated errors are errors with semantics causally inherited from earlier errors, and thus are covered by the existing error registry.

For each failed trace, a localizer examines the task and the action-wise summaries, and identifies both types of errors. Specifically, it outputs: 1) all contexts containing newly-introduced errors and 2) the root context of each propagated error. The comprehensive reasoning process conducted by the localizer mitigates the problem of false failure identifications and achieves high failure recognition accuracies. Additional details of the localizer are described in Appendix A.2.

Each error is assigned one or more failure types from a fixed taxonomy shown in Table 2. These types are deliberately broad and not directly applied to error pattern representation. The fine-grained error explanations are provided later by temporal pattern extraction. The detailed usage of the failure type taxonomy is provided in Appendix B.7.

3.3 Trace Component Attribution

AgentTAC first sets a broad attribute taxonomy (shown in Table 3) according to the potential role and state of each component in the trace. Then, an attribute proposer generates a list of unique boolean attributes for each category in the taxonomy.

The generated attribute pool is not considered complete or highly error-relevant. AgentTAC dynamically supplements it while annotating all components in the traces. It first annotates all components with the first-version attributes. Then, in any failed trace, if a component does not match any attribute in its category, the proposer generates

a new attribute that characterizes it. The new attribute may also cover other previously uncovered components of the same type, which is determined by an instant annotation of all traces with the new attribute. The process repeats until all components are covered by at least one attribute. Additional details of the attribute proposal and supplementation process are described in Appendix A.3.

This dynamic expansion serves two purposes:

- Coverage: Each component in each trace is adequately characterized, reducing the chance that observed action-context properties are missed during pattern mining.
- Error relevance: As supplementary attributes are proposed from previously uncovered components in failed traces, they tend to capture more specific error-related properties.

3.4 Error Pattern Extraction

After trace component attribution, each trace is converted into an ordered sequence of attribute blocks. Each block contains the matching attributes of a specific task, action, or context. AgentTAC then searches for sequences of attribute blocks that exhibit high failure rates and repeatedly occur in erroneous traces as error patterns. We define a set of rules that a valid error pattern must satisfy:

- The involved actions must be contiguous in the trace, and the involved contexts must be adjacent to the involved actions, without skipping any intermediate steps. A pattern with broken logical connections fails to reflect consistent agent behavior.

Failure Type	Description
Factual and Content Errors	The agent state contains incorrect, contradictory, or internally inconsistent facts, task content, or contextual information.
Constraint and Requirement Violations	The agent state fails to satisfy task requirements, user requirements, or external constraints that should already be respected.
Incompleteness and Omission	The agent state misses content that should already be present, leaving the subtask incomplete, under-specified, or not closed.
Reasoning and Decision Errors	The agent state reflects an incorrect judgment, inference, decision, or choice, even if it appears superficially complete.
Execution and Grounding Failures	The agent state suggests that a subtask was completed, but the intended action did not actually take effect or reach the correct completed state.

Table 2: The failure type taxonomy of AgentTAC.

Component	Categories
Task	Task characteristic
Agent Action	Input understanding Validation and checking Reasoning and decision-making External interaction Final response
Environment Context	Information completeness Conclusion and certainty Task progress External environment and execution status Context consistency

Table 3: The attribute taxonomy of AgentTAC.

- The pattern must end with an agent action, matching the operational scope in Section 3.2, where errors are anchored to the agent action after which they first appear.
- The ending action must be directly followed by a localized error. This temporal anchoring makes the pattern interpretable and potentially actionable.
- The pattern must preserve the original temporal order of actions and contexts. If the pattern contains task attributes, they must appear at the beginning of the pattern as preliminary.

To ensure the coherence of the extracted patterns with the localized errors, we propose a bounded exhaustive enumeration method named *trace segment backprop*. Specifically, AgentTAC enumerates candidates backward from each localized error context, as illustrated in Figure 2. Enumeration begins with single attributes or attribute combinations from the action block that directly precedes the error. It then expands to earlier adjacent context and action blocks, allowing combinations of multiple attributes within each traversed block. To guarantee the coverage, generalizability, and interpretability of the error patterns while reducing

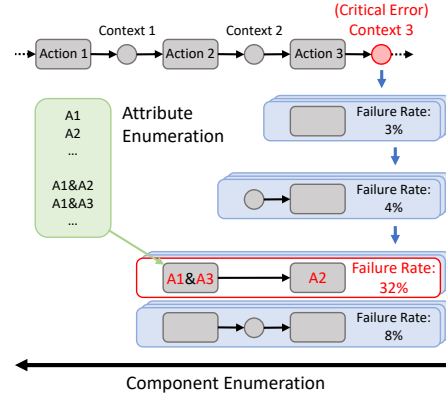


Figure 2: Trace segment backprop from a localized error to longer contiguous temporal patterns.

complexity, each block contains at most m (2 in the paper) attributes and each pattern contains at most k (3 in the paper) non-task blocks. This backprop mechanism significantly reduces the search space and the time complexity of enumeration.

Candidate generation is anchored at erroneous contexts, but candidate scoring scans all traces. The failure rate of a pattern is calculated as the ratio of its occurrences whose ending action introduces an error over all of its occurrences. A pattern will be considered valid and excluded from further enumeration if its failure rate achieves a certain threshold. The pseudocode of the process is presented in Algorithm 1 in Appendix A.4.

3.5 Extension to Reasoning Traces

Specific agents explicitly include reasoning content as part of their outputs, such as chain-of-thoughts and planner messages, which are crucial for understanding agent failures. In contrast, attributes from non-reasoning traces only capture the observable behavioral and environmental properties of agents.

To leverage such reasoning content, we design a plug-and-play attribute expansion mechanism for AgentTAC. Specifically, in trace summarization, the summarizer is required to preserve the whole

reasoning chain. In attribute proposal and supplementation, the proposer generates an extra set of reasoning-specific attributes that describe how the agent conducts reasoning (e.g., uses evidence, interprets constraints, etc). These reasoning attributes operate in the same manner as the behavioral ones, and are only applied to traces with reasoning content. With the hybrid attribute pool, extraction on such traces yields an additional list of reasoning-level error patterns. Note that reasoning attributes do not impact the extraction of non-reasoning patterns. For instance, if two traces share the same behavioral content but differ in reasoning, error pattern extraction on them yields the same behavioral patterns but different reasoning ones.

Through this plug-and-play mechanism, AgentTAC conducts behavioral error analyses on common agents, and further performs in-depth reasoning-level analyses on agents with reasoning traces. It is also worth noting that this mechanism can also be applied to other types of structured trace content beyond reasoning. During processing, AgentTAC can be easily extended with additional property-specific summarization and attribute streams to analyze specific agent behavior, such as tool calls and environment snapshots.

4 Experiments

4.1 Experimental Setup

We evaluate AgentTAC on two multi-turn tool-calling benchmarks: τ^2 -Bench and VitaBench, and four agents: the small-scale Gemini-3.1-Flash-Lite (DeepMind, 2026) and Qwen3-30B-A3B (Yang et al., 2025), and the large-scale Claude-Sonnet-4.5 (Anthropic, 2025) and DeepSeek-V4-Pro (DeepSeek-AI, 2026). Qwen3-30B-A3B and DeepSeek-V4-Pro provide explicit reasoning content. Benchmark users are simulated by DeepSeek-V4-Flash (DeepSeek-AI, 2026). Unless otherwise stated, the LLM-driven modules in AgentTAC use Qwen3-30B-A3B-Instruct.

Since no existing baseline comprehensively matches the goal of AgentTAC, we compare individual components with the closest relevant alternatives. For error localization, we compare against AgentRx, AgentDebug, a root-cause oracle that identifies the earliest error with 100% accuracy, and the vanilla LLM used in AgentTAC without the proposed new-versus-propagated localization procedure. The latter is analogous to trace-level multi-error prompting such as TRAIL, while avoiding

Method	τ^2 -Bench			VitaBench		
	F1	F2	$\mathcal{P}_e$	F1	F2	$\mathcal{P}_e$
AgentRx	0.45	0.37	0.60	0.22	0.18	0.50
AgentDebug	0.26	0.21	0.29	0.21	0.17	0.26
Oracle	0.62	0.51	1.00	0.63	0.52	1.00
Vanilla LLM	0.31	0.49	0.70	0.33	0.49	0.44
AgentTAC	0.71	0.77	0.55	0.54	0.63	0.29

Table 4: Comparisons on error localization. AgentTAC outperforms the baselines in F1 and F2 scores.

dependence on implementation-specific prompts. For attribute proposal, we compare against AdaTest and HiBug2.

4.2 Baseline Comparisons

Error localization. Four human experts annotate every error occurrence in 50 randomly sampled traces from each benchmark. Then, we evaluate the error localization performance of the methods with three metrics:

- F1 and F2 scores, which jointly measure the predicted error locations via precision, recall, and their balance.
- Earliest-error precision ($\mathcal{P}_e$), which measures whether the earliest error is localized.

Table 4 reports localization quality. AgentTAC substantially improves F1 and F2 scores from the baselines, validating its joint reasoning over newly-introduced errors and propagated error states. The vanilla LLM has F2 higher than F1 scores, showing that its recall is higher than precision. This reveals that vanilla processing tends to mark many downstream positions as erroneous and struggles to separate newly introduced errors from their consequences. Note that each failed traces may contain multiple distinct errors, resulting in the low scores of the root-cause oracle. Figure 3 further shows that longer traces tend to have higher failure rates and more errors, highlighting the need for multi-error analyses in long-horizon interactive tasks.

Attribute proposal and labeling quality. The same experts assign boolean scores to proposed attributes along three dimensions. Intrinsic quality (IQ) measures whether an attribute is clear, coherent, and non-redundant. Relevance to the task (RT) measures whether the attribute captures a meaningful aspect of the task distribution. Relevance to concrete components (RC) measures whether the attribute is useful for describing specific agent

Method	τ^2 -Bench			VitaBench		
	IQ	RT	RC	IQ	RT	RC
AdaTest	0.89	0.64	0.40	0.86	0.56	0.27
HiBug2	0.84	0.70	0.28	0.76	0.45	0.31
AgentTAC	0.86	0.47	1.00	0.88	0.33	0.79

Table 5: Comparisons on attribute quality. AgentTAC achieves comparable intrinsic attribute quality to the baselines while significantly outperforming them in component relevance.

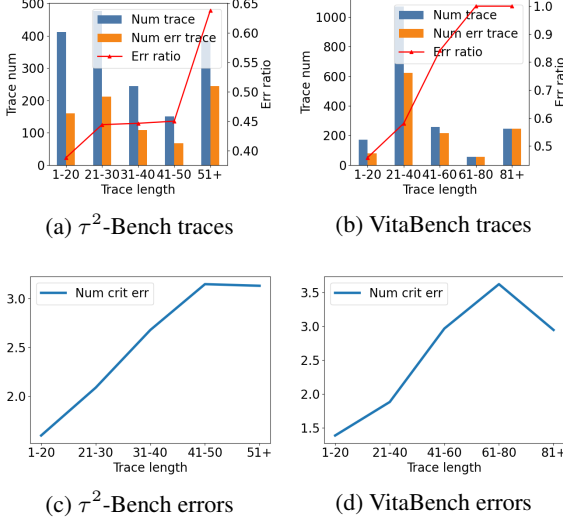


Figure 3: Trace and error statistics on the two tested benchmarks. Results by different agents are averaged.

actions and environment contexts within traces. Additional details are presented in Appendix B.1.

Table 5 reports average scores across expert judgments. AgentTAC achieves comparable intrinsic quality and much higher component relevance than baselines, validating the value of dynamic attribution grounded in concrete trace components. Its component-oriented attributes yield relatively lower task relevance than input-oriented methods such as AdaTest and HiBug2. However, the stronger component relevance better supports our component-level debugging. Experts also score labels assigned by AgentTAC, which reach 0.93 average label accuracy. Attribute examples from different task, action, and context categories are listed in Table 1 in Appendix B.3.

4.3 Extracted Error Patterns

Table 1 shows representative error patterns under the `airline` task category in τ^2 -Bench. Note that since Qwen has high task failure rates, its failure rate threshold is set to 0.05 to avoid exponential complexity explosion in enumeration, while other

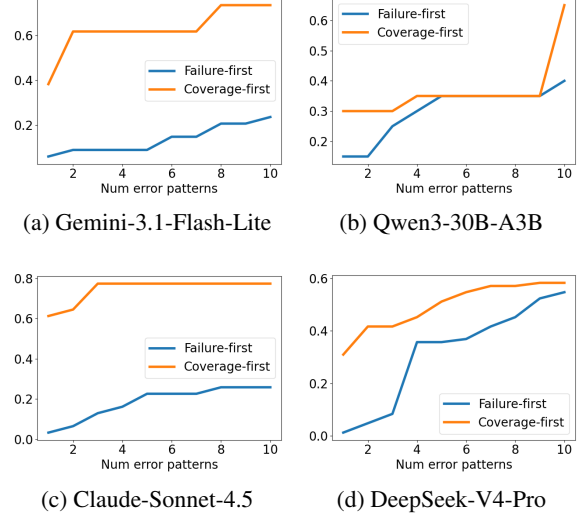


Figure 4: Coverage growth with the increasing number of extracted error patterns.

agents use 0.2. Results show that AgentTAC can extract complex error patterns composed of multiple components and attributes. Additional examples are listed in Table 2 in Appendix B.4.

Figure 4 shows coverage over all errors, defined as the fraction of errors covered by at least one extracted pattern. Coverage increases steadily as more patterns are included, and the top-10 patterns already cover more than 60% of all errors. This indicates that the mined patterns capture recurring failure mechanisms across multiple localized errors in multiple agent runs.

When reasoning traces are available, AgentTAC additionally extracts reasoning-oriented error patterns without replacing the behavior-level analysis. Table 6 shows examples that reveal reasoning-level errors. These patterns are complementary to behavior-level patterns: the latter describe what the agent does in the environment, while the former further explain how the agent’s intermediate reasoning contributes to the final outcome.

In a hold-out study shown in Appendix B.5, error patterns mined on the 85% split retain non-trivial held-out failure rates and coverage across agents. This within-domain transfer suggests that the patterns capture reusable debugging knowledge rather than memorizing training traces; cross-domain transfer is not assumed because task logic, policies, and APIs can differ substantially.

4.4 Ablation Study

In this section, we use the `airline` task category in τ^2 -Bench as a representative setting to investi-

Reasoning Agent	Error Pattern Description	Failure Rate	Occurrence
Qwen	When the agent logically deduces implicit search parameters in an action, such as calculating a required departure window based on the user’s desired arrival time or connecting flight requirements.	0.33	12
DeepSeek	In tasks involving mixed cabin classes, when the agent determines the need for multiple tool invocations, and then provides reasoning for the final conclusion based on those findings.	1.00	8

Table 6: Representative reasoning-oriented error patterns extracted by AgentTAC.

Agent	Setting	$\mathcal{N}$	$\overline{\mathcal{F}}$	$\mathcal{C}_{\max}$
Gemini	Earliest	665	0.47	0.47
	All w/o Expansion	475	0.48	0.32
	All	995	0.45	0.68
Qwen	Earliest	53	0.21	0.68
	All w/o Expansion	38	0.18	0.32
	All	121	0.17	0.68
Claude	Earliest	772	0.46	0.48
	All w/o Expansion	438	0.35	0.61
	All	1068	0.40	0.61
DeepSeek	Earliest	632	0.48	0.50
	All w/o Expansion	485	0.40	0.50
	All	911	0.40	0.95

Table 7: Comparisons among different ablation settings on error pattern extraction. $\mathcal{N}$, $\overline{\mathcal{F}}$, and $\mathcal{C}_{\max}$ are the number, the average failure rate, and the maximum coverage of the extracted error patterns, respectively.

gate the respective impacts of different modules and parameter choices.

All-error vs. earliest-error-only. We compare all-error localization with a baseline that extracts patterns only from the earliest error in each trace. This baseline approximates root-cause-oriented diagnosis, where downstream errors are ignored once the first failure is identified. As shown in Table 7, all-error localization yields more patterns and higher error coverage. Its higher coverage with lower average failure rates suggests better generalizability rather than overfitting to low-frequency errors. The result supports the central assumption of AgentTAC: distinct errors within a trace provide additional anchors for discovering reusable temporal failure mechanisms.

Impact of dynamic attribute expansion. Table 7 also compares AgentTAC with a variant without dynamic attribute expansion. Without expansion, the initial attribute inventory cannot cover all task, action, and context components observed in long traces. Expansion produces more attributes and higher pattern coverage without impacting average failure rates, indicating that it reveals useful analysis perspectives beyond the non-expanded variant without sacrificing generalizability. Over-

all, high average failure rates with low coverage indicate overfitting to narrow trace segments, while high failure rates with high coverage indicate both effectiveness and generalizability.

Additional ablations on failure rate thresholds and the LLM used in AgentTAC are presented in Appendix B.6. The threshold study shows that users can trade off pattern precision and coverage according to downstream needs. The LLM study shows that the resulting labels and localization behavior remain stable, suggesting that AgentTAC is not tied to a single model backend.

4.5 Error-Pattern-Driven Test-Time Repair

To test whether extracted error patterns contain actionable signals for agent improvement, we conduct a lightweight repair experiment. The agent LLM is frozen and also serves as a monitor that observes the current action and context, labels attributes, and checks for error pattern matching. When a match is found, it issues a warning grounded in the current task progress and context, and forces the agent to revise the response; no specialized correction instructions or training are used. We also test two variants: 1) repairing only the earliest matched pattern to align with root-cause-driven baselines, and 2) repairing the same number of randomly selected steps to simulate generic context enrichment and additional revision. Patterns are extracted from 30 traces under the `airline` task category in τ^2 -Bench, and repair is conducted on the remaining 20.

Table 8 reports task failure rates under different settings. Note that there are differences between task failure rates and pattern failure rates, which is stated in Appendix B.2. The lowest average failure rate achieved by all-error-driven repair provides preliminary evidence that extracted error patterns support test-time monitoring and repair, though effectiveness varies by agent. The gap between all-error repair and earliest-error repair also suggests that a later recurring failure mechanism can still be actionable even when it is not the first er-

Agent	$\mathcal{F}_{\text{ori}}$	$\mathcal{F}_{\text{earliest}}$	$\mathcal{F}_{\text{random}}$	$\mathcal{F}_{\text{all}}$	n_{repair}
Gemini	0.44	0.55	0.45	0.40	9.3
Qwen	0.56	0.60	0.50	0.60	7.6
Claude	0.34	0.15	0.25	0.15	7.5
DeepSeek	0.26	0.25	0.20	0.20	6.7
Average	0.40	0.39	0.35	0.34	7.8

Table 8: Comparisons on pattern-driven test-time repair performance. The four $\mathcal{F}$ s are the task failure rates under original, earliest-error-only, random-monitoring, and all-error-driven settings, respectively. n_{repair} denotes the average number of repair operations per trace. The performance decline on Qwen is possibly due to its limited capability in handling complex contexts involving repair-related information.

ror in a trace. Unlike post-hoc repair methods that need complete traces, our strategy operates predictively during execution, indicating that pattern-driven analyses are powerful complements to pure localization-driven ones.

5 Conclusion

We present AgentTAC, a temporal action-context error pattern mining framework for debugging interactive agents across tasks and traces. AgentTAC starts by localizing distinct errors existing in each failed trace. With the detailed error information and a constructed error-related attribute pool, AgentTAC conducts efficient temporal sequence enumeration to extract error-prone attribute patterns. Experiments on multiple agents and benchmarks validate the effectiveness of AgentTAC. The extracted error patterns provide new insights into agent debugging, and connect trace diagnosis to actionable test-time agent monitoring and repair.

Limitations

AgentTAC is designed for detailed component-level debugging, which makes it more computationally expensive than single-trace diagnosis. Trace summarization, error localization, and attribute proposal & labeling all require LLM calls, while error pattern enumeration grows with the number of attributes, trace length, maximum block size, and maximum pattern length. Although dense content summarization, caching, and bounded enumeration reduce cost, very large deployments may require additional sampling or approximate mining. Related computation analyses are provided in Appendix C.

The method is also descriptive rather than fully causal. Newly introduced errors reduce error-propagation noise, but a mined error pattern with

high failure rate should be interpreted as a high-risk association or causal hypothesis, not counterfactual proof that the pattern causes failure. The lightweight repair experiment supplies preliminary interventional evidence of actionability, while controlled interventions and human validation remain important for stronger causal claims and high-stakes use.

Finally, attribute quality depends on the capability of the labeling LLM. We conduct LLM-related ablation studies in Appendix B.6 to validate the stability of AgentTAC from this perspective, and Appendix C shows that a finetuned 4B labeler can reproduce 93.2% of the 30B model’s labels at substantially lower latency and memory. The scope of AgentTAC also depends on the information exposed in traces: if logs omit important environment contexts, tool feedback, or external malfunctions, AgentTAC can only diagnose observable behavior. Extending the context taxonomy to external anomalies is compatible with the framework but remains future work.

Acknowledgments

This work was supported in part by Tencent under Grant No. 3136023.

References

- Anthropic. 2025. Claude sonnet 4.5 system card. <https://www.anthropic.com/claude-sonnet-4-5-system-card>.
- 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*.
- Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. 2025. τ^2 -bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*.
- Muxi Chen, Zhaohua Zhang, Chenchen Zhao, Mingyang Chen, Wenyu Jiang, Tianwen Jiang, Jianhuan Zhuo, Yu Tang, Qiuyong Xiao, Jihong Zhang, and Qiang Xu. 2025a. Failureatlas: Mapping the failure landscape of t2i models via active exploration. *arXiv preprint arXiv:2509.21995*.
- Muxi Chen, Chenchen Zhao, and Qiang Xu. 2025b. Hi-bug2: Efficient and interpretable error slice discovery for comprehensive model debugging. *arXiv preprint arXiv:2501.16751*.
- Google DeepMind. 2026. Gemini 3.1 flash-lite model card. <https://deepmind>.

[google/models/model-cards/
gemini-3-1-flash-lite/](https://huggingface.co/google/models/model-cards/gemini-3-1-flash-lite/).

- DeepSeek-AI. 2026. Deepseek-v4: Towards highly efficient million-token context intelligence. https://huggingface.co/deepseek-ai/DeepSeek-V4-Pro/blob/main/DeepSeek_V4.pdf.
- Darshan Deshpande, Varun Gangal, Hersh Mehta, Jitin Krishnan, Anand Kannappan, and Rebecca Qian. 2025. Trail: Trace reasoning and agentic issue localization. *arXiv preprint arXiv:2505.08638*.
- Sabri Eyuboglu, Maya Varma, Khaled Saab, Jean-Benoit Delbrouck, Christopher Lee-Messer, Jared Dunnmon, James Zou, and Christopher Ré. 2022. Domino: Discovering systematic errors with cross-modal embeddings. *arXiv preprint arXiv:2203.14960*.
- Wei He, Yueqing Sun, Hongyan Hao, Xueyuan Hao, Zhikang Xia, Qi Gu, Chengcheng Han, Dengchang Zhao, Hui Su, Kefeng Zhang, Man Gao, Xi Su, Xiaodong Cai, Xunliang Cai, Yu Yang, and Yunke Zhao. 2025. Vitabench: Benchmarking llm agents with versatile interactive tasks in real-world applications. *arXiv preprint arXiv:2509.26490*.
- 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. 2024. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, volume 2024, pages 52989–53046.
- Yang Liu, Hongjiang Feng, Junsong Pu, and Zhuangbin Chen. 2026. Masprism: Lightweight failure attribution for multi-agent systems using prefill-stage signals. *arXiv preprint arXiv:2605.07509*.
- Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, Zirui Wang, and Ruoming Pang. 2025. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 1160–1183.
- Marco Tulio Ribeiro and Scott Lundberg. 2022. Adaptive testing and debugging of nlp models. In *Proceedings of the 60th annual meeting of the association for computational linguistics (volume 1: long papers)*, pages 3253–3267.
- Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. 2020. Beyond accuracy: Behavioral testing of nlp models with checklist. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pages 4902–4912.
- 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 preprint arXiv:2602.23701*.
- An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.
- Shu-Xun Yang, Cunxiang Wang, Haoke Zhang, Wenbo Yu, Lindong Wu, Jiayi Gui, Dayong Yang, Yukuo Cen, Zhuoer Feng, Bosi Wen, Yidong Wang, Lucen Zhong, Jiamin Ren, Linfeng Zhang, and Jie Tang. 2026. Tracesir: A multi-agent framework for structured analysis and reporting of agentic execution traces. *arXiv preprint arXiv:2603.00623*.
- Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. τ -bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*.
- Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. 2025. Which agent causes task failures and when? on automated failure attribution of llm multi-agent systems. *arXiv preprint arXiv:2505.00212*.
- Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pages 15585–15606.
- Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, Xiaoteng Ma, Xiaodong Yu, Gowtham Ramesh, Jialian Wu, Zicheng Liu, Pan Lu, James Zou, and Jiaxuan You. 2025. Where llm agents fail and how they can learn from failures. *arXiv preprint arXiv:2509.25370*.

A Additional Details of AgentTAC

A.1 Trace Preprocessing

AgentTAC uses trace preprocessing to expose the information needed by later error localization and attribution while keeping the original agent workflow unchanged. Each raw trace is converted into three textual views. The *full* view rewrites every user message, agent message, tool call, tool result, and optional reasoning field into natural language while preserving concrete values such as reservation identifiers, prices, dates, policies, and tool outputs. The *brief* view compresses each step into one or two sentences and is used when later modules only need the core action and observation. The

result view summarizes the final task judgment, including whether the trace satisfies the task and, if applicable, which final requirements are violated.

For context attribution, AgentTAC also constructs *incremental* summaries. For every step index t , the corresponding context summary describes the accumulated execution situation from the beginning of the trace through step t . This makes the context a first-class object: the localizer and attribute labeler can reason about what information is currently available, what constraints remain unresolved, and how previous tool calls have changed the environment.

A.2 Distinct Error Localization

The localizer is applied only to traces whose final result is judged erroneous. It receives the task, final-result summary, and preprocessed trace, and separates two phenomena that are often conflated in long traces.

The implementation returns two aligned registries. The first registry contains newly introduced errors, represented by the triggering step index, one or more error types from the taxonomy in the main paper, and a concise explanation. The second registry contains propagated errors, represented by the later step index, the root step index that initiates the error, and an explanation of why the later context is a continuation rather than a new error. If multiple new errors are proposed for the same step, they are merged into one step-level entry with the union of error types and concatenated evidence. Invalid or non-taxonomy labels are rejected during normalization.

This distinction is important for pattern mining. If every downstream symptom were used as an anchor, the extracted patterns would be dominated by consequences of the same early mistake. If only the first error were used, later independent failures would be invisible. The newly introduced error registry therefore provides a middle ground: it preserves distinct failure mechanisms while filtering propagation noise.

A.3 Attribute Proposal and Supplementation

AgentTAC maintains separate Boolean attribute libraries for tasks, actions, and contexts. Initial attributes are proposed under the taxonomy in the main paper. Task attributes describe intrinsic task properties. Action attributes describe what an agent action does, such as extracting information, checking a constraint, comparing options, calling a tool,

or issuing a final response. Context attributes describe the execution context after an action, including information completeness, certainty, progress, external execution status, and consistency.

The dynamic supplementation procedure enforces coverage over trace components. First, all traces are labeled with the current library. Then the system searches for uncovered components: tasks for which all task attributes are false, actions for which all action attributes are false, and contexts for which all context attributes are false. For task attributes, an uncovered trace is selected and the proposer generates a reusable task attribute that must be true for that task. For action and context attributes, AgentTAC prioritizes traces whose components have the largest mean number of false attributes, selects the latest uncovered component in the trace, proposes a candidate attribute in each applicable category, and keeps the candidate that covers the largest number of previously uncovered components.

Every proposed attribute is immediately validated on the target component. If it is not labeled true for the component that motivated it, it is discarded. Otherwise, the attribute is added to the library and labeled across all traces. The process repeats until no uncovered task, action, or context component remains, except for configured prefix turns that are ignored because they are usually generic greetings or setup states. For agents with explicit reasoning traces, the same procedure is run with reasoning-oriented action and context attributes; these attributes are stored separately and only applied to traces that expose reasoning content.

A.4 Error Pattern Enumeration

After labeling, each trace becomes an ordered sequence of attribute blocks. The implementation first builds a task block, then alternates action and context blocks by step index. Each block contains the attributes that are true for the corresponding component. The enumerator constructs non-empty subsets within each block up to the maximum block size m , and then enumerates contiguous block sequences backward from localized error contexts up to the maximum non-task length k .

Candidate scoring scans all traces, not only failed traces. For each candidate pattern, AgentTAC counts all occurrences and the subset whose ending action is followed by a localized error. The failure rate is the ratio between these counts; cover-

Algorithm 1 Error Pattern Enumeration in AgentTAC

Require: Labeled traces $\mathcal{D}$; critical error anchors $\mathcal{E}$; maximum attributes per block m ; maximum non-task blocks k ; minimum occurrence $n_{\min}$; minimum failure rate τ .

Ensure: Ranked temporal error patterns $\mathcal{P}$.

```

1: Notation:  $x_i$  is a trace;  $\mathcal{E}_i$  is the set of critical error
  anchors in  $x_i$ ;  $T_i$ ,  $A_{i,t}$ , and  $C_{i,t}$  are the task, action, and
  context attribute blocks;  $B_i$  is the ordered block sequence;
   $L_i$  is the trace length;  $e$  is an error anchor;  $a$  is the position
  of the anchored action block;  $w$  is a contiguous block
  window;  $b_j$  is the  $j$ -th block in  $w$ ;  $S_j$  is the set of valid
  attribute subsets for  $b_j$ ;  $\mathcal{C}$  is the candidate set;  $p$  and  $q$  are
  candidate patterns;  $\text{Match}(p, B_i)$  returns all contiguous
  matches of  $p$  in  $B_i$ ;  $\text{end}(o)$  is the ending action index
  of occurrence  $o$ ;  $\text{occ}$ ,  $\text{err}$ ,  $\text{fr}$ , and  $\text{cov}$  denote occurrence
  count, matched error count, failure rate, and coverage;
   $\text{Stat}(p)$  is the tuple of statistics used for pruning;  $q \preceq p$ 
  means that  $q$  is a contiguous subpattern of  $p$ .

2:  $\mathcal{C} \leftarrow \emptyset$ 
3: for  $x_i \in \mathcal{D}$  do
4:    $B_i \leftarrow [T_i, A_{i,1}, C_{i,1}, \dots, A_{i,L_i}, C_{i,L_i}]$ 
5:   for  $e \in \mathcal{E}_i$  do
6:      $a \leftarrow \text{pos}(A_{i,e}, B_i)$ 
7:      $\mathcal{W}_{i,e} \leftarrow \{B_i[s : a] \mid a-s+1 \leq k, B_i[a] = A_{i,e}\}$ 

8:      $\mathcal{W}_{i,e} \leftarrow \mathcal{W}_{i,e} \cup \{[T_i; w] \mid w \in \mathcal{W}_{i,e}\}$ 
9:     for  $w = (b_1, \dots, b_r) \in \mathcal{W}_{i,e}$  do
10:       $S_j \leftarrow \{u \subseteq b_j \mid 1 \leq |u| \leq m\}$  for  $j =$ 
         $1, \dots, r$ 
11:       $\mathcal{C} \leftarrow \mathcal{C} \cup (S_1 \times \dots \times S_r)$ 
12:    end for
13:  end for
14: end for
15: for  $p \in \mathcal{C}$  do
16:    $\text{occ}(p) \leftarrow \sum_{x_i \in \mathcal{D}} |\text{Match}(p, B_i)|$ 
17:    $\text{err}(p) \leftarrow \sum_{x_i \in \mathcal{D}} |\{o \in \text{Match}(p, B_i) : \text{end}(o) \in$ 
      $\mathcal{E}_i\}|$ 
18:    $\text{fr}(p) \leftarrow \text{err}(p)/\text{occ}(p)$ ;  $\text{cov}(p) \leftarrow \text{err}(p)/|\mathcal{E}|$ 
19:   if  $\text{occ}(p) < n_{\min}$  or  $\text{fr}(p) < \tau$  then
20:      $\mathcal{C} \leftarrow \mathcal{C} \setminus \{p\}$ 
21:   end if
22: end for
23:  $\mathcal{C} \leftarrow \text{Canonicalize}(\mathcal{C})$ 
24:  $\mathcal{C} \leftarrow \mathcal{C} \setminus \{p : \exists q \preceq p, \text{Stat}(q) = \text{Stat}(p)\}$ 
25:  $\mathcal{P} \leftarrow \text{Rank}(\mathcal{C}; \text{fr}, \text{cov}, \text{occ}, -|p|)$ 
26: return  $\mathcal{P}$ 

```

age is the fraction of localized errors covered by the candidate. Candidate patterns below the minimum occurrence or failure-rate threshold are removed. Finally, AgentTAC canonicalizes candidates and prunes redundant longer candidates when a shorter subsequence has equivalent metrics. This keeps the final report compact without changing the temporal interpretation of retained patterns.

The pseudocode of the process is presented in Algorithm 1.

τ^2 -Bench			
Agent	$\mathcal{F}_{\text{ori}}$	n	n_e
Gemini	50.0	26.1	2.1
Qwen	86.7	112.4	3.1
Claude	32.7	41.3	1.7
DeepSeek	14.0	34.7	1.5
Qwen (reasoning)	71.0	26.0	2.3
DeepSeek (reasoning)	10.3	30.5	1.4

VitaBench			
Agent	$\mathcal{F}_{\text{ori}}$	n	n_e
Gemini	89.3	30.1	1.5
Qwen	96.0	214.2	3.0
Claude	53.0	34.3	2.0
DeepSeek	40.7	30.8	1.6
Qwen (reasoning)	86.7	42.2	3.2
DeepSeek (reasoning)	40.3	28.9	1.6

Table 9: Benchmark-level statistics of different agents averaged across scenarios. n and n_e are the number of steps and errors per trace, respectively.

B Additional Experimental Results

B.1 User Study Setup

For error localization and attribute labeling, four human experts (Ph.D students working in AI model debugging directions) annotate independently before adjudication. We use the majority consensus as ground truth instead of calculating average scores. Annotators are shown the task, the whole trace, and the final agent execution results, but not the method predictions.

B.2 Agent Performance Statistics

Most of the four tested agents fail to achieve satisfactory performance in the experiments, especially those on VitaBench. Table 9 shows the performance of different agents on the two benchmarks. Note that there are differences between the input-level task failure rates and the component-level pattern failure rates. Since there are only a few error occurrences in a failed trace, the pattern failure rates are significantly lower than the corresponding task failure rates.

B.3 Examples of the Proposed Attributes

Table 10 lists representative attributes from the libraries induced by AgentTAC. The examples illustrate the intended granularity of the representation. Task attributes capture what kind of problem the agent is solving; action attributes capture what the agent does at a particular step; context attributes capture what information and commitments are present after an action. These attributes

Component	Category	Example Attributes
Task	Task characteristic	Whether the task involves modifying or cancelling an existing reservation. Whether the task requires satisfying multiple passenger-specific constraints.
Agent Action	Input understanding	Whether the action extracts a missing user constraint.
	Validation and checking	Whether the action asks for clarification before acting. Whether the action checks policy eligibility. Whether the action verifies that a proposed change is feasible.
	Reasoning and decision-making	Whether the action compares multiple options. Whether the action commits to one option under uncertainty.
	External interaction	Whether the action calls a booking or lookup tool. Whether the action updates an external order or reservation.
	Final response	Whether the action reports a final task outcome. Whether the action asks the user to confirm a proposed external update.
Environment Context	Information completeness	Whether the context has unresolved required information. Whether the context contains all information needed for the next action.
	Conclusion and certainty	Whether the context contains a stable intermediate decision. Whether the context still depends on an unresolved uncertainty.
	Task progress	Whether the context indicates that prerequisite information has been gathered. Whether the context shows that the task is ready for finalization.
	External environment and execution status	Whether the prior tool action has not been confirmed. Whether the environment reports a successful update.
	Context consistency	Whether the context contains conflicting constraints. Whether the context reflects a newly changed user preference.

Table 10: Representative attributes proposed under the constructed taxonomy.

Method	Categories	Items
AgentRx	3	20
AgentDebug	5	17
TRAIL	1	9
AgentTAC	5	116

Table 11: Coverage of analysis taxonomies in the tested scenarios. For AgentTAC, “items” counts induced task, action, and context attributes; the five failure types are a separate surface-level summary.

are not error labels themselves. Instead, they describe reusable execution properties that can later combine into temporal error patterns.

The induced attribute taxonomy is closer in purpose to the diagnostic taxonomies used by prior methods than the five-category failure-type taxonomy. Table 11 therefore compares their representational coverage. In the tested scenarios, AgentTAC attributes cover nearly all applicable items represented by the baseline taxonomies while adding finer task, action, and context distinctions. The five failure types instead summarize the surface characteristics of localized outcomes and are not meant to replace established diagnostic taxonomies.

B.4 Examples of the Extracted Error Patterns

Table 12 shows the top three behavioral error patterns for each agent under the `airline` task category in τ^2 -Bench, after filtering to patterns with more than 10 occurrences. Each row verbalizes a temporal pattern over task, action, and context attributes in the same natural-language style as the main paper. The reported failure rate is computed over all occurrences of the pattern in the analyzed traces.

These examples illustrate the difference between temporal pattern mining and ordinary error categorization. A coarse category such as “constraint violation” says what went wrong after the fact. In contrast, an extracted pattern can state that the agent is in a context lacking explicit user authorization and then synthesizes a final resolution, or that it performs prerequisite-data gathering twice and then responds without a tool invocation. Such patterns are closer to reusable debugging hypotheses because they describe the process that repeatedly precedes critical errors.

For agents that expose reasoning traces, Agent-

Agent	Error Pattern Description	Failure Rate	Occurrence
Gemini	When the current state involves unresolved logistical feasibility uncertainty, the agent proceeds to evaluate policy applicability.	0.53	15
	When there is unresolved uncertainty about the logistical feasibility of a flight itinerary, such as layover durations or transfer viability, the agent evaluates policy applicability without invoking external tools, potentially failing to address the underlying feasibility concerns through system-based verification.	0.50	14
	When the current state contains unresolved logistical feasibility uncertainty, the agent provides reasoning for the final conclusion.	0.47	15
Qwen	In airline customer service tasks, when the agent evaluates and selects a payment method, such as a certificate or gift card, by comparing available balances or applying prioritization rules.	0.31	13
	When the agent invokes an external tool to modify system state or records, such as processing a cancellation, making a booking, or updating a user account, rather than merely retrieving information.	0.27	157
	When the agent executes a state-changing tool while translating contextual information into the required parameters for the tool call.	0.27	15
Claude	In airline customer service tasks, when the agent calculates aggregate costs by combining financial components while also gathering prerequisite data for final resolution, then repeats the data-gathering step before ultimately responding without any tool invocation.	0.67	12
	In tasks requiring financial calculations, when the agent first executes an information retrieval tool to fetch data and gathers prerequisite data for final resolution, followed by providing reasoning for the final conclusion.	0.45	11
	When the agent is in a state with an unresolved pricing inconsistency between previously communicated and newly retrieved flight costs, it first gathers prerequisite data for final resolution via a tool call, but then responds to the user without invoking any necessary external tools to address the discrepancy.	0.43	14
DeepSeek	In tasks involving the application of travel credits, promotional vouchers, or discount codes, when the agent is in a state lacking explicit user authorization for irreversible actions and then presents a final conclusion without articulating the reasoning or justifications for that conclusion.	0.69	13
	In tasks involving travel credits, when the state lacks explicit user authorization, and the agent evaluates policy applicability.	0.60	15
	In tasks involving flexible user preferences, when the agent is in a state lacking explicit user authorization and then proceeds to synthesize findings for a final resolution.	0.57	14

Table 12: Top-3 behavioral error patterns with the highest failure rates among patterns extracted by AgentTAC under the `airline` task category in τ^2 -Bench with more than 10 occurrences.

TAC produces an additional reasoning-oriented pattern set. These patterns share the same enumeration and scoring procedure as behavioral patterns, but they use attributes that describe reasoning properties, such as whether the agent explicitly tests a conclusion against a constraint, carries forward an unverified inference, or derives a tool parameter from an incomplete state. This separation prevents reasoning-specific attributes from changing the behavior-level pattern set, while still allowing developers to inspect deeper logical causes when the trace contains the necessary information.

B.5 Hold-Out Experiments

The main experiments mine patterns on 85% of the traces. We then evaluate the same discovered patterns on the remaining 15% hold-out traces without re-mining or changing their descriptions. Table 13 reports the average held-out failure rates and the

Agent	$\bar{\mathcal{F}}$	$\mathcal{C}_{\max}$
Gemini	0.57	0.43
Qwen	0.17	0.75
Claude	0.33	0.67
DeepSeek	0.39	1.00

Table 13: Average failure rates and maximum coverage statistics of the extracted error patterns on the remaining 15% held-out traces.

highest coverage achieved by the extracted pattern set for each agent under the `airline` task category in τ^2 -Bench. The average failure rate measures whether a matched pattern remains predictive on unseen traces from the same task domain and agent. Coverage measures the fraction of successfully captured held-out errors.

The results show that many mined patterns continue to match errors outside the discovery split.

Threshold	$\mathcal{N}$	$\bar{\mathcal{L}}$	$\bar{\mathcal{A}}$	$\mathcal{F}_{\max}$	$\bar{\mathcal{F}}$	$\mathcal{C}_{\max}$
0.1	526	2.78	2.83	1.00	0.24	0.81
0.2	1068	2.85	2.93	1.00	0.40	0.61
0.5	1085	2.92	3.03	1.00	0.71	0.35

Table 14: Statistics of the error patterns extracted by AgentTAC under different failure rate thresholds. $\bar{\mathcal{L}}$ and $\bar{\mathcal{A}}$ are the average numbers of pattern components and attributes, respectively. $\mathcal{F}_{\max}$ is the maximum failure rate among the patterns.

Agent	Localization Overlap	Labeling Overlap
Gemini	0.56	0.87
Qwen	0.28	0.88
Claude	0.42	0.85
DeepSeek	0.52	0.89

Table 15: Sensitivity of AgentTAC to the LLM used for error localization and attribute labeling.

This is especially important for AgentTAC because the patterns are intended as reusable debugging artifacts rather than explanations of individual traces. The lower held-out average failure rates of some individual patterns are expected because the hold-out split is small; the more relevant signal is that high-precision patterns with non-trivial error coverage remain available across all tested agents. This experiment measures generalization across instances in the same domain. Universal transfer across unrelated domains is neither required nor claimed: different environments may expose different state variables, tools, policies, and task-completion logic.

B.6 Additional Ablation Results

Impact of failure rate thresholds. Table 14 compares different failure rate thresholds for Claude-Sonnet-4.5. Lower thresholds admit more broadly recurring patterns and can increase maximum coverage, while higher thresholds emphasize more error-prone patterns. The average number of components and attributes also increases with the threshold, indicating that stricter thresholds tend to retain more specific temporal configurations. This threshold is therefore a practical knob: a developer who wants a broad debugging report may use a lower threshold, while a monitor that should fire fewer warnings may prefer a higher threshold.

Impact of the LLM used in the pipeline. Several modules in AgentTAC use LLMs, especially critical error localization and attribute labeling. To examine sensitivity to this choice, we compare outputs generated by Gemini-3.1-Pro-Preview and Qwen3-30B-A3B-Instruct. For localization, we re-

port the average Jaccard overlap between the sets of localized error locations. For labeling, we report overlap on attribute labels over the matched trace components.

The overlap in labeling is high across LLMs, showing that component attribution is relatively stable. The overlap in error localization is slightly lower, especially on agents with relatively high task failure rates. However, the localization overlap is still substantial, validating the fundamental stability of AgentTAC. On the other hand, in practical applications, strong LLMs have more positive impacts on error localization instead of attribute labeling.

B.7 Failure Types of Error Patterns

Though failure types are not directly used as components in error patterns, they reflect the failure characteristics of the patterns from another perspective.

Table 16 breaks down the top-1 error pattern of each agent in Table 12 by the failure types of the localized errors covered by that pattern. The distributions show that high-failure-rate temporal patterns are not always tied to a single coarse error category. However, their covered errors are not type-agnostic: They generally exhibit clear concentrations on a small subset of failure types, suggesting that they provide directional evidence about the concrete errors with which they are associated. This supports the use of temporal error patterns as debugging units that can cut across conventional error taxonomies.

B.8 End-to-End Processing Example

Table 17 follows one erroneous step through all three pipeline stages. The trace is produced by Claude on the 38th task of the τ^2 -Bench airline domain. The example illustrates how a verbose action is first normalized, how one newly introduced error is separated from four propagated consequences, and how Boolean attributes become a reusable cross-trace temporal pattern.

C Computation Analysis

Unlike HiBug2, the main computational costs of AgentTAC come from LLM processing instead of error pattern enumeration, as our enumeration starts from error anchors rather than all possible trace segments. Except for agents with high failure rates, complexity of error pattern enumeration is not a bottleneck in most cases.

The main LLM computations are:

Agent	Error Pattern Description	Failure Type	Occurrence
Gemini	When the current state involves unresolved logistical feasibility uncertainty, the agent proceeds to evaluate policy applicability.	Reasoning and Decision Errors	5
		Constraint and Requirement Violations	1
		Factual and Content Errors	1
		Incompleteness and Omission	1
Qwen	In airline customer service tasks, when the agent evaluates and selects a payment method, such as a certificate or gift card, by comparing available balances or applying prioritization rules.	Reasoning and Decision Errors	3
		Factual and Content Errors	1
Claude	In airline customer service tasks, when the agent calculates aggregate costs by combining financial components while also gathering prerequisite data for final resolution, then repeats the data-gathering step before ultimately responding without any tool invocation.	Constraint and Requirement Violations	4
		Factual and Content Errors	4
DeepSeek	In tasks involving the application of travel credits, promotional vouchers, or discount codes, when the agent is in a state lacking explicit user authorization for irreversible actions and then presents a final conclusion without articulating the reasoning or justifications for that conclusion.	Constraint and Requirement Violations	5
		Factual and Content Errors	2
		Reasoning and Decision Errors	2

Table 16: Failure-type distributions of the top-1 error pattern of each agent in Table 12. Occurrences count localized errors of each type covered by the pattern under the `airline` task category in τ^2 -Bench.

Stage	Output for a τ^2 -Bench <code>airline</code> Trace
User Request	The user is frustrated by a delayed flight in the most recent reservation.
Erroneous Step	The agent finds reservation 4OG6T3 and delayed flight HAT018, then offers to change flights or the entire reservation and to provide compensation.
Preprocessing	Step 8: notifies the user of the delayed BOS–LAS flight and offers changes, cancellation, or compensation.
Localization	Step 8 introduces a constraint-and-requirement violation; its consequences propagate to steps 10, 17, 19, and 20.
Attribution	<code>RequiresFareRuleEvaluation</code> , <code>RequiresPolicyEnforcement</code> , and related Boolean attributes.
Pattern Extraction	<code>task:InvolvesTravelCredits + state:LacksRequiredPolicyContext + step:RespondsWithoutToolInvocation</code> .

Table 17: End-to-end example of the outputs produced by the main stages of AgentTAC.

- Step-wise trace preprocessing, scaling linearly with the number of traces and trace length.
- Error localization, scaling linearly with the number of traces. In addition, trace length determines the amount of LLM context to be processed.
- Attribute labeling, scaling with the product of the number of trace components and the number of active attributes.

Table 18 separates the cost of the full pattern-mining pipeline from localization and coarse error typing alone. The latter is comparable to prior trace-diagnosis methods; most additional cost comes from component attribution, which enables the more detailed temporal interpretations returned by the full pipeline. The implementation enables

vLLM automatic prefix caching for repeated trace prefixes, and all measured token counts marked in the table already include this optimization.

Dynamic attribute expansion increases the attribute inventory by 1 task attribute (5%), 15 action attributes (25%), and 2 context attributes (4.3%) in the representative ablation. Because each added attribute induces a labeling query with the same prompt structure, its additional time and token cost is proportional to the number of added attributes. The observed extra time is 21.7%, 19.7%, 21.1%, and 20.6% for Gemini, Qwen, Claude, and DeepSeek traces, respectively; the token overhead has the same percentages. For instance, there are 50 traces under the `airline` task category in τ^2 -Bench. The traces have 37.5 steps on average. The task, the agent action, and the environment context have 20, 60, and 46 attributes respectively. The

Method	Queries (Theoretical)	Tokens (Theoretical)	Queries (Measured)	Tokens (Measured)
AgentRx	1	L	1	4.1K
AgentDebug	$4T$	$4L$	114.8	19.0K [†]
TRAIL	1	L	1	5.4K
AgentTAC	$(1+A)T$	$(1+A)L$	797.4	71.8K [†]
AgentTAC (Localization)	T	L	28.7	4.0K [†]

Table 18: Per-trace LLM cost comparison on the τ^2 -Bench `airline` domain. T , A , and L denote the average trace length, number of active attributes per component, and prompt length. Prompt templates are assumed to have equal length for the theoretical comparison. [†]Automatic prefix caching enabled.

Labeler	Time (s)	Tokens	Memory (GB)
Qwen3-4B	0.348	698.1	8.01
Qwen3-30B	2.965	698.1	57.15

Table 19: Attribute-labeling efficiency per call. The finetuned 4B model obtains 93.2% average label overlap with the 30B model.

three stages of LLM computations therefore involve 1875, 50, and 3995 calls respectively. The whole process with four Qwen3-30B-A3B-Instruct instances deployed with vLLM on a machine with $8 \times \text{H20 GPUs}$ (96GB) costs 2-3 hours for a full single-agent analysis.

Repeated attribute labeling can also be distilled into a smaller model. We finetune a Qwen3-4B (Yang et al., 2025) on approximately 10K labels from the τ^2 -Bench `airline` domain and evaluate it on approximately 10K labels produced by Gemini and Claude traces. It reaches 93.2% average agreement with the Qwen3-30B labeler while reducing per-call latency by $8.5 \times$ and peak memory from 57.15GB to 8.01GB, as shown in Table 19. This suggests a practical deployment split: a larger model proposes the attribute inventory, whereas a small domain labeler handles repeated attribution.

The implementation uses several controls to make the pipeline computationally practical. In LLM processing, full and brief trace summaries reduce token costs by error localization and attribute labeling, while most LLM calls are parallelized across traces, components, and attributes. In error pattern enumeration, trace segment backprop starts only from localized error anchors and only over contiguous previous blocks, while the pattern sizes are restricted to $k = 3$ blocks with $m = 2$ attributes in each. In practice, the same framework can be combined with sampling, approximate candidate generation, or domain-specific limits on attribute categories to further reduce costs in large deployments.

D Prompts Used in AgentTAC

The LLM-driven modules use structured prompts with JSON-only outputs.

The prompt used for full trace rewriting is shown as follows. Prompts for brief trace rewriting, final result summarization, and incremental preceding-trace summarization follow similar instructions.

```
You are converting a chunk of an agent trace into a readable natural-language trace for later error analysis.

You are given:
- A task category.
- A description of a task of the task category.
- A chunk of an agent trace. The chunk is a list of step dicts, each with keys "turn_idx", "role", and "content/tool_calls/reasoning", describing the index, role, and content, tool-calling, and optional reasoning content of each step in the chunk.

You need to rewrite the chunk into a readable natural-language chunk while preserving the information in each step.

Requirements:
- Rewrite each step content into natural language.
- Keep all information of the original steps. Keep all details such as user requests, constraints, tool-calling intents, tool-calling results, numerical values, commitments, policy statements, observations, and optional reasoning content.
- If a step contains reasoning, preserve the full reasoning logic chain faithfully. Keep the assumptions, intermediate inferences, checks, revisions, and conclusions in the same logical order rather than compressing them into a short summary.
- You may reorganize wording to make it easier to read, but the rewritten content must stay faithful to the original steps.
- Do not invent information.
- Return JSON only. The JSON must be a list of rewritten step dicts, each with keys "turn_idx", "role", and "content". "turn_idx" and "role" are the same as the original step, and "content" is the natural-language rewrite of the step.

Example:
Input:
{
  "task_category": "Airline customer service scenarios, including booking, rebooking, cancellations, seat selection, baggage services, and flight information queries",
  "task": "Testing that agent refuses to proceed with a cancellation that is not allowed even if User mentions that she had been told she didn't need insurance. Reason for call: You want to cancel reservation EHGLP3."

  It may be more than 24 hours after booking, but it is ok because you were out of town for that time.",
  "trace_chunk": [
    {
      "turn_idx": 2,
      "role": "assistant",
      "reasoning": "I should first inspect the reservation details and insurance status. If there is no insurance and the booking is outside the free-cancellation window, the user may not be eligible. I need the tool result before I can judge whether the cancellation can proceed.",
      "tool_calls": [
        {
          "name": "get_reservation_details",
          "arguments": {
            "reservation_id": "EHGLP3"
          },
          "result": {
            "reservation_id": "EHGLP3",
            "flight_type": "one_way",
            "cabin": "basic_economy",
            "created_at": "2024-05-04T23:12:06",
            "insurance": "no",
            "flights": [
              {
                "flight_number": "HAT156",
                "origin": "PHX",
                "destination": "SEA",
                "date": "2024-05-17",
                "price": 50
              },
              {
                "flight_number": "HAT021",
                "origin": "SEA",
                "destination": "JFK",
                "date": "2024-05-17",
                "price": 54
              }
            ]
          },
          "passengers": [
            {
              "first_name": "Evelyn",
              "last_name": "Taylor"
            },
            {
              "first_name": "Any",
              "last_name": "Silva"
            }
          ]
        }
      ],
      "turn_idx": 3,
      "role": "user",
      "content": "I was told I didn't need insurance because I was out of town after booking. I still want to cancel it."
    }
  ]
}

Output:
{
  {
```

```

    "turn_idx": 2,
    "role": "assistant",
    "content": "The assistant reasons that it should first inspect the reservation details and insurance status, because if there is no insurance and the booking is outside the free-cancellation window, the user may not be eligible. The assistant also reasons that it needs the tool result before judging whether the cancellation can proceed. The assistant then calls the tool get_reservation_details with argument reservation_id=EHGLP3. The tool result returns reservation_id EHGLP3, flight_type one_way, cabin basic_economy, created_at 2024-05-04T23:12:06, and insurance no. It also returns two flight legs: HAT156 from PHX to SEA on 2024-05-17 priced at 50, and HAT021 from SEA to JFK on 2024-05-17 priced at 54. The returned passengers are Evelyn Taylor and Anya Silva."
  },
  {
    "turn_idx": 3,
    "role": "user",
    "content": "The user says they were told insurance was unnecessary because they were out of town after booking, and they still want to cancel the reservation."
  }
]

```

The prompt used for distinct error localization is shown as follows.

You are identifying error locations in an agent trace.

You are given:

- A task category.
- A description of a task of the task category.
- A flag indicating whether the final result has errors.
- A description of the final execution result of the task.
- An agent trace of the task. The trace is a list of step dicts, each with keys "turn_idx", "role", and "content", describing the index, role, and content of each step in the trace. The trace may include both user and agent steps, but only the agent steps are relevant for error identification.

You need to identify which erroneous agent steps contain distinct errors that are firstly introduced, and which erroneous agent steps carry forward existing errors. For agent steps that contain firstly-introduced errors, you need to assign each error one of the following error types:

- Factual and Content Errors: Whether the state after the agent step contains incorrect, contradictory, or internally inconsistent facts, task content, or contextual information.
- Constraint and Requirement Violations: Whether the state after the agent step fails to satisfy task requirements, user requirements, or external constraints that should already be respected.
- Incompleteness and Omission: Whether the state after the agent step is missing content that should already be present, leaving the subtask incomplete, under-specified, or not closed.
- Reasoning and Decision Errors: Whether the state after the agent step reflects an incorrect judgment, inference, decision, or choice, even if it looks superficially complete.
- Execution and Grounding Failures: Whether the state after the agent step looks as if a subtask was completed, but the intended action did not actually land, take effect, or reach the correct completed state.

For agent steps that only carry forward existing errors, you need to identify the earliest step that introduces the error and provide a concise explanation of why the later step is a continuation of the same error instead of a new error.

Requirements:

- Focus only on agent steps with role "assistant".
- Use the full trace to distinguish newly-introduced errors from later steps that merely carry forward an existing error.
- A trace may contain multiple distinct new errors. Return every agent step that first introduces a distinct error.
- Return JSON only. The JSON must be a dict with keys "error_introduction" and "error_forward". The value for "error_introduction" is a list of step dicts, each with keys "turn_idx", "error_types", and "reason". "turn_idx" is the index of the agent step where one distinct error is firstly introduced. "error_types" is a list of one or more predefined error types listed above. If the same newly introduced error has multiple type characteristics, put all applicable types in this list and explain the combined judgment in one reason string. Use at most one error_introduction item per turn_idx. "reason" is a concise explanation of why this step is judged to introduce a new error. The value for "error_forward" is a list of step dicts, each with keys "turn_idx", "root_idx", and "reason". "turn_idx" is the index of the agent step that carries forward an existing error. "root_idx" is the index of the earliest step that introduces the error. "reason" is a concise explanation of why this step is judged to be a continuation of an existing error instead of a new error.

Example:

Input:

```

{
  "task_category": "Airline customer service scenarios, including booking, rebooking, cancellations, seat selection, baggage services, and flight information queries",
  "task": "The user wants to cancel reservation EHGLP3 and claims it should still be allowed even if it may be outside the normal cancellation window.",
  "overall_error": true,
  "error_summary": "The final result is judged incorrect because the assistant allows a cancellation that should not be allowed under the reservation policy.",
  "trace": [
    {
      "turn_idx": 0,
      "role": "assistant",
      "content": "The assistant greets the user and asks how it can help."
    },
    {
      "turn_idx": 1,
      "role": "user",
      "content": "The user asks to cancel reservation EHGLP3."
    },
    {
      "turn_idx": 2,
      "role": "assistant",
      "content": "The assistant calls a tool to retrieve reservation EHGLP3. The tool result shows that the reservation is a basic economy one-way itinerary from PHX to JFK for two passengers, created on 2024-05-04T23:12:06, with no insurance attached."
    },
    {
      "turn_idx": 3,
      "role": "assistant",
      "content": "The assistant says it understands and will proceed with the cancellation, even though the reservation is basic economy without insurance and no qualifying exception has been established."
    },
    {
      "turn_idx": 4,
      "role": "assistant",

```

```

    "content": "The assistant continues processing the unsupported cancellation.",
  },
  {
    "turn_idx": 5,
    "role": "assistant",
    "content": "The assistant says both passengers used the same payment method even though the reservation shows different payment sources."
  }
]
}
Output:
{
  "error_introduction": [
    {
      "turn_idx": 3,
      "error_types": ["Constraint and Requirement Violations"],
      "reason": "This is the first assistant step that decides to proceed with canceling the reservation despite the known reservation facts showing no cancellation eligibility. The error is newly introduced here because earlier assistant steps only retrieved or summarized the reservation facts and had not yet made the policy-violating cancellation decision."
    },
    {
      "turn_idx": 5,
      "error_types": ["Factual and Content Errors"],
      "reason": "This step introduces a separate factual error by stating that both passengers used the same payment method, which contradicts the retrieved reservation details showing different payment sources. This error would still need correction even if the earlier unsupported-cancellation decision were fixed, so it is a distinct newly introduced error rather than a continuation of turn 3."
    }
  ],
  "error_forward": [
    {
      "turn_idx": 4,
      "root_idx": 3,
      "reason": "This step continues acting on the same unsupported cancellation decision that was first introduced at turn 3. It does not add a new incorrect fact, assumption, policy interpretation, or execution failure; if the turn 3 cancellation decision were corrected, this follow-up processing step would no longer be erroneous in the same way."
    }
  ]
}

```

The prompt used for initial task attribute proposal is shown as follows. Prompts for initial action and context attribute proposal follow similar instructions.

You are helping build a reusable "task" attribute library for agent trace analysis. A "task" is a user request that an agent is asked to complete, such as booking a flight or modifying a hotel reservation. "Task" attributes describe intrinsic properties of the task — what it inherently requires, what domain knowledge it involves, or what structural constraints it imposes — independent of any specific trace.

You are given:

- A task category.
- An integer number 'num_attributes'.
- Existing task attributes that have already been collected.

You need to propose 'num_attributes' task attributes belonging to the given task category.

Requirements:

- Attributes must be binary yes/no attributes.
- Attributes must be broadly reusable across many tasks in the same task category. Avoid task-specific wording or attributes.
 - E.g., in task category "Airline customer service scenarios, including booking, rebooking, cancellations, seat selection, baggage services, and flight information queries", "RequiresExternalSystemAccess" is more preferable than "RequiresAccessToFlightABCDEFSystem".
- Do not propose duplicates or near-duplicates that are already present in the existing attributes.
- Attributes must stay on the cause-side or precondition-side, not symptom-side or final-outcome-side.
 - E.g., in task category "Airline customer service scenarios, including booking, rebooking, cancellations, seat selection, baggage services, and flight information queries", "InvolvesMultiLegItinerary" is more preferable than "ErrorProneDueToMultiLegItinerary".
- Attributes that are more relevant to potential agent errors or failure modes are more preferable, but do not encode specific errors or failure modes themselves.
- Use concise and reusable attribute names.
- Return JSON only. The JSON must be a list of attribute dicts, each with keys "name" and "definition". "name" is the concise attribute name, and "definition" is a one-sentence explanation of the attribute and how to judge it.

Example:

Input:

```

{
  "task_category": "Airline customer service scenarios, including booking, rebooking, cancellations, seat selection, baggage services, and flight information queries",
  "num_attributes": 1,
  "existing_attributes": [
    {
      "name": "InvolvesMultiLegItinerary",
      "definition": "Whether the task category commonly involves itineraries with multiple flight legs."
    }
  ]
}

```

Output:

```

[
  {
    "name": "RequiresExternalSystemAccess",
    "definition": "Whether completing the task category normally requires reading or writing data in external systems."
  }
]

```

The prompt used for dynamic task attribute supplementation is shown as follows. Prompts for dynamic action and context attribute supplementation follow similar instructions.

You are helping expand a reusable "task" attribute library for agent trace analysis. A "task" is a user request that an agent is asked to complete, such as booking a flight or modifying a hotel reservation. "Task" attributes describe intrinsic properties of the task — what it inherently requires, what domain knowledge it involves, or what structural constraints it imposes — independent of any specific trace.

You are given:

- A task category.
- A description of a task of the task category.
- A description of the final execution result of the task.
- An integer number 'num_attributes'.
- Existing task attributes that have already been collected.

You need to inspect the descriptions and propose 'num_attributes' missing task attributes belonging to the given task category that should be added to the library.

Requirements:

- Attributes must be binary yes/no attributes.
- Attributes must be broadly reusable across many tasks in the same task category. Avoid task-specific wording or attributes.
 - E.g., in task category "Airline customer service scenarios, including booking, rebooking, cancellations, seat selection, baggage services, and flight information queries", "RequiresExternalSystemAccess" is more preferable than "RequiresAccessToFlightABCDEFSystem".
- Only add attributes that become salient after inspecting the descriptions relevant to the given task.
- Every proposed attribute must be true for the given task.
- Do not propose duplicates or near-duplicates that are already present in the existing attributes.
- Attributes must stay on the cause-side or precondition-side, not symptom-side or final-outcome-side.
 - E.g., in task category "Airline customer service scenarios, including booking, rebooking, cancellations, seat selection, baggage services, and flight information queries", "InvolvesMultiLegItinerary" is more preferable than "ErrorProneDueToMultiLegItinerary".
- Use concise and reusable attribute names.
- Return JSON only. The JSON must be a list of attribute dicts, each with keys "name" and "definition". "name" is the concise attribute name, and "definition" is a one-sentence explanation of the attribute and how to judge it.

Example:

Input:

```
{
  "task_category": "Airline customer service scenarios, including booking, rebooking, cancellations, seat selection, baggage services, and flight information queries",
  "task": "The user wants to change the date of an already booked flight.",
  "result": "The assistant failed to complete the task because it failed to properly handle the date and schedule information.",
  "num_attributes": 1,
  "existing_attributes": []
}
```

Output:

```
[
  {
    "name": "RequiresExternalSystemAccess",
    "definition": "Whether completing the task category normally requires reading or writing data in external systems."
  }
]
```

The prompt used for task attribute labeling is shown as follows. Prompts for action and context attribute labeling follow similar instructions.

You are labeling a task-level attribute for a task. A "task" is a user request that an agent is asked to complete, such as booking a flight or modifying a hotel reservation.

You are given:

- A task category.
- A description of a task of the task category.
- A description of the final execution result of the task.
- A task-level attribute.
- Definition of the attribute.

You need to decide whether the attribute applies to the task, and return a Yes/No tag with a short reason.

Requirements:

- Return JSON only. The JSON must be a dict with keys "tag" and "reason". "tag" is either "Yes" or "No". "reason" is a short string explaining the rationale for the tag.

Example:

Input:

```
{
  "task_category": "Airline customer service scenarios, including booking, rebooking, cancellations, seat selection, baggage services, and flight information queries",
  "task": "The user wants to cancel an existing reservation and ask about refund eligibility.",
  "result_summary": "The assistant failed to complete the cancellation correctly because it did not handle the eligibility conditions consistently.",
  "attribute_name": "RequiresPolicyInterpretation",
  "attribute_definition": "Whether completing the task requires interpreting policies, eligibility rules, or exceptions."
}
```

Output:

```
{
  "tag": "Yes",
  "reason": "The task requires deciding refund and cancellation eligibility under policy conditions."
}
```