AGENTRX: Diagnosing AI Agent Failures from Execution Trajectories

Shraddha Barke [1] **Arnav Goyal** [2] **Alind Khare** [2] **Avaljot Singh** [3] **Suman Nath** [2] **Chetan Bansal** [2]

Abstract

AI agents often fail in ways that are difficult to localize because executions are probabilistic, long-horizon, multi-agent, and mediated by noisy tool outputs. We address this gap by manually annotating failed agent runs and release a novel benchmark of 170 trajectories across 11 diverse task settings, including structured API workflows, incident management, and open-ended web/file tasks. Each trajectory is annotated with a critical failure step and a category from a grounded-theory derived, cross-domain failure taxonomy. To mitigate the human cost of failure attribution, we present AGENTRX, an *automated domain-agnostic* diagnostic framework that pinpoints the critical failure step in a failed agent trajectory. It synthesizes constraints, evaluates them step-by-step, and produces an auditable validation log of constraint violations with associated evidence; an LLM-based judge uses this log to localize the critical step and category. AGENTRX improves step localization by 75% on average over prior work, while providing failure category attribution.

1. Introduction

Large Language Model (LLM)-based agents are increasingly *acting autonomously* without a human in the loop. Modern agentic systems plan, coordinate, call tools, and execute complex real-world tasks. These agents are no longer confined to sandboxes; they are being deployed in high-stakes environments such as recruiting (LinkedIn, 2024), web browsing (Fourney et al., 2024), and production cloud-service operations (Zhang et al., 2024). This ability to operate with minimal guidance dramatically amplifies human productivity (Brynjolfsson et al., 2023). However, the same autonomy that enables speed also makes these agentic systems difficult to debug: trajectories are long-horizon, tool outputs are noisy, and failures can propagate through side effects before they are observed. Consequently, ensuring the robustness of AI agents is not an option, but a prerequisite for their safe adoption in the real-world.

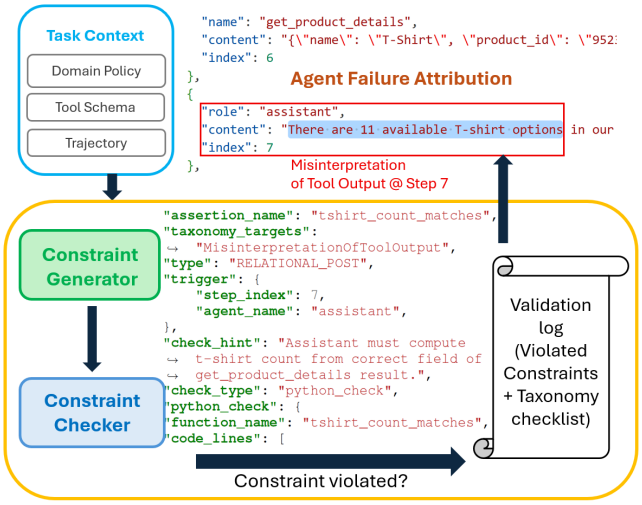

Figure 1. Given a failed trajectory, domain policy, and tool schema, AGENTRX outputs the critical failure step, a root-cause category, and an auditable constraint violation.

In this paper, our focus is on: *root-causing and localizing the critical failure which prevented the Agent from successful task completion*. We define *critical failure* as the first unrecoverable failure by any agent in an execution trajectory. To understand and characterize such failures, we manually analyze failed agent executions and introduce a benchmark of 170 trajectories across 11 diverse task settings, including structured API workflows, incident management, and open-ended web/file tasks. Each trajectory includes the full execution trace (messages, tool calls, tool outputs, and observable environment state) and is annotated with the *critical step* and a *failure category* capturing why the run ultimately fails. Our grounded-theory-based annotation process also yields a cross-domain failure taxonomy for root-cause attribution.

Challenges. However, manual failure attribution is expensive for long-horizon trajectories and hard to scale for agents serving millions of daily users. It requires not only inspecting long trajectories, but also domain expertise to understand the system prompt, available tools, domain policies,

[1]Microsoft Research [2]Microsoft [3]UIUC. Correspondence to: Shraddha Barke <sbarke@microsoft.com>, Chetan Bansal <chetanb@microsoft.com>.

Proceedings of the 43rd International Conference on Machine Learning, Seoul, South Korea. PMLR 306, 2026. Copyright 2026 by the author(s).

user guardrails, and expected state transitions. This creates a need for automated validation of agent executions. In software engineering, contracts make interface misuse and constraint violations explicit; agentic systems need an analogous notion of correctness, but one stated over executions: ambiguous intents, messages, tool calls, and state updates.

AGENTRX. To operationalize this view, we design AGENTRX, a *domain-agnostic* automated framework for AI agents that (1) normalizes heterogeneous multi-agent logs into a common trajectory representation, (2) synthesizes *constraints* from tool schemas, domain policies, and the observed trajectory, and (3) produces a step-indexed validation log of violated obligations with supporting evidence, which a downstream judge uses to localize the critical failure step and assign a category. As shown in Figure 1, after the assistant agent calls GET_PRODUCT_DETAILS and reports "11 available T-shirt options," AGENTRX synthesizes a relational constraint TSHIRT_COUNT_MATCHES that recomputes the count from the tool response and checks agreement with the assistant's interpretation. The resulting violation includes the exact trajectory window that contradicts the assistant's claim, helping the judge localize the decisive failure.

Evaluation. We evaluate AGENTRX on our proposed benchmark and prior failure-analysis datasets (Zhang et al., 2025a; Cemri et al., 2026). AGENTRX beats state of the art and our baselines on both localizing the first unresolved critical failure step and categorizing the root-cause. Our results clearly indicate that trajectory-level constraints are a useful signal for detecting failures. Our contributions include: (i) A manually annotated open-source dataset of **170 failed trajectories across diverse tasks**-including step localization and failure attribution (κ=0.89). (ii) a **novel domain-agnostic failure diagnosis framework** that combines constraint synthesis and LLM-based adjudication to localize agent failures. (iii) A **cross-domain failure taxonomy derived through grounded theory coding**, capturing 9 root-cause categories that generalize across diverse domains, and (iv) Extensive experiments showing 75% average improvement in failure localization over prior work, and 30% improvement in root-cause categorization over our own judge baselines.

2. The AGENTRX Benchmark

We are the first to introduce a benchmark for *attribution of the first unrecoverable critical failure* in AI agents. Our benchmark contains 170 failed trajectories across 11 diverse task settings, spanning structured API workflows, production incident management, software-development agents, mathematical reasoning, and open-ended web/file tasks. These settings vary along three axes: single-agent vs. multi-agent execution, structured APIs vs. open-ended environments, and recoverable vs. unrecoverable failure chains. For each setting, we sample failed trajectories and use them to construct our failure-attribution benchmark.

τ-bench. τ-bench Yao et al. (2024) is a tool-calling benchmark for retail tasks, with an LLM-simulated user and a single agent equipped with domain-specific APIs and policy guidelines. In a typical τ-bench task, the agent can cancel or modify pending orders, return or exchange delivered orders, update user addresses, and answer product queries. We run 115 τ-bench retail tasks using GPT-4O with one trial and analyze the 39 failed trajectories.

Flash. Flash Zhang et al. (2024) is a workflow automation agent for diagnosing recurring incidents in production cloud services. It is a multi-agent system in which specialized agents execute a troubleshooting guide for the incident being investigated. The guide involves steps like querying for clusters in a region that might be affected and also mitigation steps based on whether the incident is a false alarm or not. We analyze 42 Flash trajectories containing at least one agent failure.

Magentic-One. Magentic-One Fourney et al. (2024) is a generalist multi-agent system for open-ended web and file-based tasks. It uses an orchestrator to coordinate specialized agents for web browsing, file navigation, and code execution. We randomly sample 44 failed multi-agent trajectories from the Who&When dataset (Zhang et al., 2025a).

RelWork. We sample traces from MAST (Cemri et al., 2026) and Who&When (Zhang et al., 2025a) by selecting five from each of their nine task domains. Seven domains come from MAST and cover software-development agents, API/tool workflows, math reasoning, and open-ended web/file tasks. The remaining two come from Who&When: algorithm-generated and hand-crafted agent systems over GAIA/AssistantBench-style queries. This yields 45 additional failed trajectories and brings the AGENTRX benchmark to 170 trajectories in total. This broadens the benchmark and lets us evaluate whether the taxonomy and attribution task generalize to agent-failure datasets that include long-horizon trajectories.

2.1. Grounded Theory Annotation of Failures

To systematically analyze the critical agent failures, we manually annotate trajectories using a grounded-theory open coding process (Glaser & Strauss, 2017), allowing failure categories to emerge from the traces rather than imposing a predefined label set. Three annotators independently coded trajectories from the three domains. Annotators read each trajectory step-by-step and wrote a structured *coding memo* whenever a failure was observed. We define a *failure* as any point at which the agent did not make progress toward a correct outcome *e.g.,* violating a policy guideline or executing an invalid tool call. We follow a two-stage coding

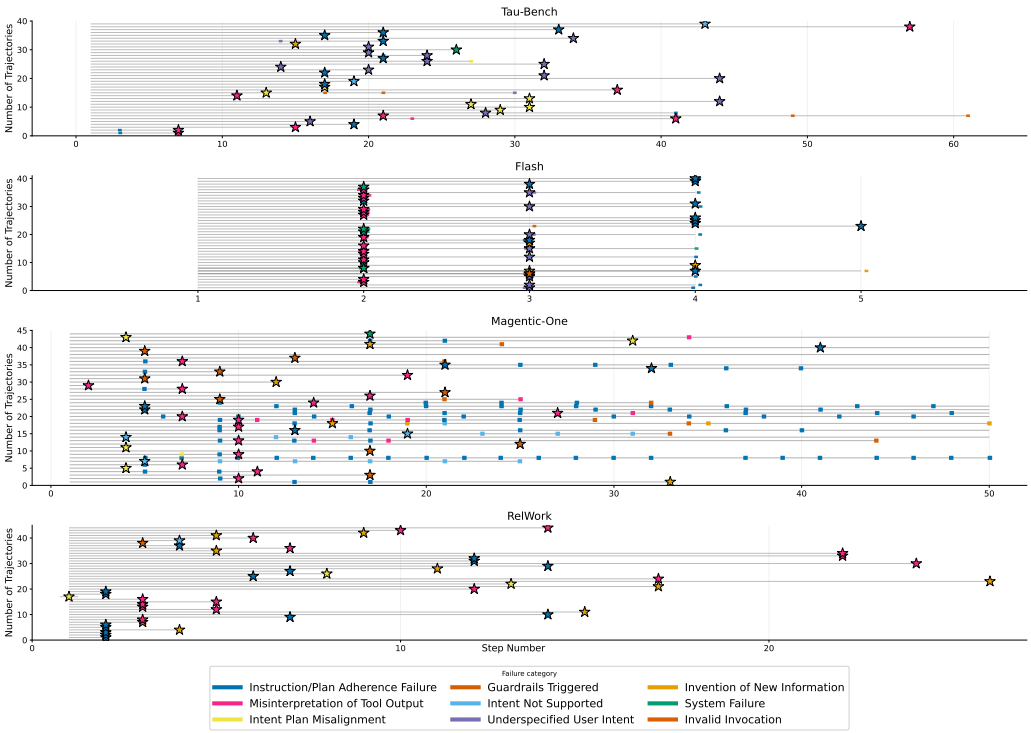

Figure 2. **Failure timelines per trajectory across domains.** Each horizontal line is one trajectory. Colored ticks mark detected failures at the corresponding step number (labeled with the category); the star indicates the root-cause step. Magentic-One has 295 total failures, with 68% of trajectories containing at least two failures; Flash has 55 total failures, and τ-bench has 52.

procedure:

Phase 1: Exhaustive failure marking. Annotators first marked *all* failure steps, even if later steps dominate the final outcome. For each failure event, we recorded: (i) the step index; (ii) an *open code*: a short, concrete description tightly grounded in the trace; and (iii) a reason for that failure. This stage captures early deviations and cascading failures, including ones the agent later recovers from.

Phase 2: Critical failure identification. We then identified the *critical failure step*: the earliest failure in the causal chain from which the agent does not recover. This step determines the root-cause category for the trajectory. Throughout coding, annotators compared new codes against existing ones, reusing codes when possible and creating new definitions when needed. We periodically merged related open codes into higher-level categories and refined their definitions. We iterated until reaching *theoretical saturation*, where new trajectories no longer introduced new failure phenomena. We then froze the category definitions and used closed coding to annotate remaining trajectories and re-code earlier ones, verifying that the taxonomy covers failures beyond the subset that initially shaped it. Further, our labels are drawn from three structurally different domains spanning both single-agent and multi-agent settings, and each domain was annotated by independent annotators.

2.2. A Cross-Domain Failure Taxonomy

Prior work takes a system-level view of multi-agent failures, organizing failure modes by design, coordination, and verification stages (Cemri et al., 2026). We develop a taxonomy that labels the critical failure:

1. **Plan Adherence Failure** The agent fails to follow required directions or the agreed plan by ignoring directives. This covers both under-execution and over-execution (unplanned actions) that violate the plan, domain policy, or orchestrator constraints.
2. **Invention of new information** The agent introduces, removes, or alters information that is not grounded in any available input, context, or tool output. This includes fabricating unsupported facts, hallucinated details, or omitting relevant information without justification.
3. **Invalid Invocation** The agent issues an invalid tool call, e.g., malformed inputs, missing arguments, or values that fail schema validation.
4. **Misinterpretation of Tool Output** The agent incorrectly reasons about its own or another agent's tool output, leading to incorrect assumptions or actions.
5. **Intent–Plan Misalignment** The agent misinterprets the user's goal or constraints and produces an incorrect plan.
6. **Under-specified User Intent** The agent was unable to

Table 1. Distribution of *critical failure* labels across domains. We normalize per domain (percent of failed trajectories)

Root-cause category	τ-bench	Flash	Magentic	RelWork
Instruction Adherence	20.5	23.8	18.2	33.3
Invention of Information	2.6	9.5	9.1	17.8
Invalid Invocation	0	0	0	2.2
Misinterpretation of Output	20.5	33.3	34.1	33.3
Intent-Plan Misalignment	15.4	0	9.1	6.7
Under-specified Intent	33.3	19	0	0
Intent Not Supported	5.1	0	6.8	2.2
Guardrails Triggered	0	0	20.5	2.2
System Failure	2.6	14.3	2.3	0
# failed trajectories	**39**	**42**	**44**	**45**

complete the task due to lack of complete information at that point in the trajectory.

7. **Intent Not Supported** The agent is asked to perform an action for which there is no tool.
8. **Guardrails Triggered** The agent's execution is blocked by safety policies or external access restrictions.
9. **System Failure** The agent faces a system connectivity issue while calling a particular tool like an API endpoint not being reachable.

Inter-annotator Agreement. We assessed inter-annotator agreement on 30 trajectories from τ-bench, Flash, and Magentic-One, each independently annotated by three annotators using the same taxonomy and guidelines. Because Cohen's κ is defined for two annotators, we compute it pairwise across annotator pairs and report the mean. The pairwise κ scores were 0.93, 0.87, and 0.9, resulting in a mean κ of 0.89. This corresponds to near perfect agreement (0.8-1.0), indicating that the taxonomy is well defined and used consistently. We manually reviewed disagreements to refine the taxonomy definitions and annotation checklist; at the critical step, labels are mutually exclusive, with ambiguous cases resolved using the checklist. More details in Appendix A.

Annotation Cost. The manual effort required for these annotations is substantial, especially for long-horizon trajectories. τ-bench trajectories have median length 36 steps (range 20–62), Flash has median length 3 (range 2–6), Magentic-One has median length 33 (range 5–130), and RelWork has median length 10 (range 4–37). Flash additionally averages 8 substeps per step.[1] Annotators spent, on average, 20 minutes per τ-bench trajectory, 22 minutes per Flash and RelWork trajectory, and 24 minutes per Magentic trajectory. Across 170 trajectories, this amounts to 62.5 total human hours and is challenging to scale. This motivates the need for automated failure attribution with AGENTRX.

[1]Flash steps are defined using the Troubleshooting Guide for consistency; each step expands into multiple substeps.

3. AgentRx Framework Formulation

Setup. AGENTRX takes as input (i) a toolset $\mathcal{T}$ with input–output schemas, (ii) optional domain policy Π, and (iii) a failed trajectory $T = \langle s_1, \ldots, s_n \rangle$. Each step s_k contains multiple logged events with fields such as agent name, tool name, step index, tool output or natural language conversational content. Each step s_k contains logged events such as messages, tool calls, and tool outputs. Since log formats differ across domains, we first normalize each trajectory into a common intermediate representation (IR). Let $T_{\leq k}$ denote the trajectory prefix up to and including step k. AGENTRX generates *executable constraints* from the tool schema, domain policy, and trajectory prefixes, then evaluates them step-by-step to produce an auditable validation log V. An LLM-based judge uses V and a taxonomy checklist K to predict the root-cause step $\hat{s}$ and category $\hat{y}$.

Taxonomy checklist. We compile our failure taxonomy into a semantic checklist K: for each category y in the taxonomy, $\mathsf{K}(y)$ specifies a small set of targeted yes/no questions (and brief decision criteria) that operationalize the category definition.

3.1. Global and Dynamic Constraint Synthesis

AGENTRX synthesizes constraints in two stages: *Global constraints* $\mathcal{C}^G$ are synthesized once from the tool schema and domain policy, and stored in a global store G *Dynamic constraints* $\mathcal{C}_k^D$ are synthesized at each step from the task instruction I and the observed prefix $T_{\leq k}$, using G as context, and stored in a local store L_k: Intuitively, global constraints encode domain rules, while dynamic constraints encode trajectory-specific rules and prior observations (e.g., constraints that must hold given earlier tool outputs). We define the constraints available at step k as the union $\mathcal{C}_k := \mathcal{C}^G \cup \mathcal{C}_k^D$. In practice, constraint synthesis emits constraints in a lightweight schema to standardize checking across domains (subsection F.1).

3.2. Guarded Constraints and Evaluation

A constraint C generated by AGENTRX comprises (i) a guard $G_C(T_{\leq k}, s_k) \in \{0, 1\}$ that determines when the constraint applies and (ii) an assertion $\Phi_C(T_{\leq k}, s_k) \in \{\text{SAT}, \text{VIOL}\}$ that returns a binary verdict v when evaluated. Guards capture structural conditions such as whether step k calls a particular tool or invokes a particular agent. Let $g_{C,k} = G_C(T_{\leq k}, s_k)$ and $\phi_{C,k} = \Phi_C(T_{\leq k}, s_k)$. Evaluation at step k is:

$$\text{EVAL}_C(k) = \begin{cases} (\text{SKIP}, \emptyset) & \text{if } g_{C,k} = 0, \\ (\phi_{C,k}, e) & \text{otherwise.} \end{cases}$$

Constraints are checked either programmatically over structured fields (e.g., schema validity, equality, membership), or

semantically as natural-language predicates evaluated using an LLM-based checker. Both return a verdict–evidence pair (v, e); examples appear in Appendix F.

3.3. Validation Log

At step k, AGENTRX evaluates each constraint $C \in \mathcal{C}_k$ whose guard applies, i.e., $G_C(T_{\leq k}, s_k) = 1$. We record each violated constraint and its supporting evidence in a step-indexed validation log:

$$\mathsf{V} := \{(k, C, e) \mid C \in \mathcal{C}_k,\ G_C(T_{\leq k}, s_k), \\ (\text{VIOL}, e) = \text{EVAL}(C, T_{\leq k}, s_k)\}.$$

V is step-indexed, auditable, and directly links each violation to supporting evidence.

3.4. Judge for Root Cause Attribution

Given the task instruction I, trajectory T, checklist K, and validation log V, an LLM-based judge outputs a critical failure step $\hat{s}$ and category $\hat{y}$. The judge selects the first step whose violation evidence is sufficient to explain why the run fails with respect to I. Violations serve as diagnostic evidence rather than hard requirements: the judge may override them when the trajectory context indicates otherwise. The judge sets $\hat{y}$ based on the taxonomy labels associated with the violation(s), and emits a short rationale citing the specific reason for selecting the failure category.

4. Experimental Evaluation

We evaluate AGENTRX as a *root-cause diagnosis* system for agent trajectories on our AGENTRX benchmark. We measure whether it can localize the first unrecoverable critical step and assign the correct root-cause category. Our experiments test accuracy, cross-domain and cross-model generalization, contribution of validation signals, and robustness under relaxed metrics.

4.1. Experimental Settings

Metrics. We evaluate step localization using *exact critical-step accuracy* (the fraction of trajectories where the predicted step exactly matches the annotated step), Acc@$\pm r$ for $r \in \{1, 3, 5\}$, and average step distance, which measures how far predictions are from the annotated step. We evaluate category prediction using *critical, any-failure, earliest, and terminal accuracy*, comparing the predicted category against the annotated critical failure, any annotated failure, the first observed failure, and the terminal failure, respectively. The judge may also output *inconclusive* when no grounded critical failure can be identified.

Experimental settings. We run each configuration n=3 times and report mean $\pm$ standard deviation to account for sampling noise and judge variability (Table 2). Unless otherwise stated, AGENTRX uses both programmatic constraints and semantic constraints. We use GPT-5 as the default model for all experiments. (Results using o3 in Table 7). In Table 2, *Baseline* denotes the judge without AGENTRX-added evidence, while *Baseline+Vio.* and related settings add validation signals incrementally.

4.2. Comparison to Prior Benchmarks

We compare against prior work: Who&When is the closest benchmark for automated failure attribution, while MAST is the closest taxonomy-level analysis of multi-agent failures (Table 6).

Comparison to Who&When. Although W&W annotates failed trajectories with responsible agents and failure steps, its target step is not always the critical failure under our definition. In one trajectory (Appendix C), the Orchestrator recovers from the failure marked by W&W: by switching from WebSurfer to FileSurfer, and only becomes unrecoverable after a later hallucinated file/path error (which our annotation catches). This illustrates that our localization is a stricter debugging target.

Table 3 compares AGENTRX with W&W across domains. W&W was designed for its own trajectory format and failure-attribution setting, so we used the AGENTRX domain adapter to convert trajectories into the format required by W&W. We also modify the W&W prompt to ask for the *first unrecoverable* failure step, matching our task definition. Across domains, AGENTRX improves step localization over W&W, with especially large gains on Flash and RelWork. AGENTRX also improves failed-agent identification in most settings and, unlike W&W, predicts a root-cause category. **Our normalized IR and evidence-grounded validation improve attribution across heterogeneous agent trajectories.**

Comparison to MAST. MAST studies aggregate multi-agent failure patterns, while AGENTRX diagnoses individual failed trajectories by localizing the first unrecoverable failure and grounding it in trace evidence. The taxonomies are not one-to-one: some AGENTRX labels have only partial MAST analogs, while Invalid Invocation, Intent Not Supported, Guardrails Triggered, and System Failure do not appear as MAST labels. Since MAST does not provide gold labels for our critical-step/root-cause task, we use it as a generalization stress test rather than an accuracy benchmark. Running AGENTRX without code changes on all 1,022 MAST trajectories yields a critical step, root-cause category, and supporting violation evidence for every trace. The resulting profiles vary by framework: Plan Adherence is dominant overall, while Invalid Invocation, Misinterpretation of Tool Output, and System Failure appear prominently

Table 2. **Effect of judge inputs and constraint generation.** We compare one-shot vs. step-by-step constraint generation, judge inputs (violations, taxonomy checklist, both) and judging protocols (Step-then-Category or All-at-Once).

	Judge Baselines		AGENTRX Ablations			
Metric (n=3)	**Baseline**	**Step-then-Cat.**	**Baseline+Vio.**	**Step-then-Cat.+Vio.**	**Taxonomy Checklist**	**Checklist+Vio.**
Tau-Bench			**Step-by-Step Constraint Generation**			
Step-index acc. (%) ↑	29.9 ± 1.2	31.6 ± 1.5	**42.7 ± 1.5**	38.5 ± 2.6	30.8 ± 2.1	39.3 ± 1.2
Category acc. (%) ↑	30.8 ± 2.1	32.5 ± 1.5	43.6 ± 1.2	**47 ± 1.5**	29.9 ± 2.4	47 ± 1.2
Avg. step distance ↓	5.4 ± 0.1	5.5 ± 0.8	**3.8 ± 0.6**	4.2 ± 0.1	5.9 ± 0.3	3.9 ± 0.3
Tau-Bench			**One-shot Constraint Generation**			
Step-index acc. (%) ↑	29.9 ± 1.2	31.6 ± 1.5	41 ± 2.1	37.6 ± 1.2	30.8 ± 2.1	**42.7 ± 3.2**
Category acc. (%) ↑	30.8 ± 2.1	32.5 ± 1.5	41.9 ± 2.4	31.6 ± 1.2	29.9 ± 2.4	**43.6**
Avg. step distance ↓	5.4 ± 0.1	5.5 ± 0.8	5.1 ± 0.4	5.3 ± 0.5	5.9 ± 0.3	**5 ± 0.3**
Flash			**Step-by-Step Constraint Generation**			
Step-index acc. (%) ↑	80.2 ± 1.4	76.2 ± 4.1	81.7 ± 1.4	75.4 ± 6.0	**82.5 ± 1.4**	80.2 ± 3.6
Category acc. (%) ↑	58.7 ± 5.0	58.7 ± 1.4	59.5 ± 0.0	**65.9 ± 5.0**	62.7 ± 1.4	59.5 ± 2.4
Avg. step distance ↓	0.33 ± 0.04	0.39 ± 0.05	0.32 ± 0.04	0.42 ± 0.11	**0.29 ± 0.01**	0.34 ± 0.05
Flash			**One-shot Constraint Generation**			
Step-index acc. (%) ↑	80.2 ± 1.4	76.2 ± 4.1	81.0 ± 2.4	75.4 ± 3.6	**82.5 ± 1.4**	80.2 ± 1.4
Category acc. (%) ↑	58.7 ± 5.0	58.7 ± 1.4	53.2 ± 1.4	56.3 ± 7.7	**62.7 ± 1.4**	57.9 ± 1.4
Avg. step distance ↓	0.33 ± 0.04	0.39 ± 0.05	**0.29 ± 0.04**	0.42 ± 0.06	0.29 ± 0.01	0.30 ± 0.03
Magentic			**Step-by-Step Constraint Generation**			
Step-index acc. (%) ↑	32.6 ± 1.3	23.5 ± 1.3	34.1 ± 1.9	34.1	32.6 ± 1.3	**35.6 ± 1.3**
Category acc. (%) ↑	35.6 ± 1.3	38.6	41.7 ± 1.3	**46.2 ± 1.3**	37.1 ± 1.3	42.4 ± 1.3
Avg. step distance ↓	22.5 ± 3	16.9 ± 0.7	14.9 ± 1.9	13.3 ± 0.8	18.9 ± 2.4	**12.4 ± 0.6**
Magentic			**One-Shot Constraint Generation**			
Step-index acc. (%) ↑	32.6 ± 1.3	23.5 ± 1.3	34.8 ± 1.1	28 ± 2.1	32.6 ± 1.3	**36.4**
Category acc. (%) ↑	35.6 ± 1.3	38.6 ± 1.3	38.6 ± 1.9	39.4 ± 1.1	37.1 ± 1.3	**40.9 ± 1.9**
Avg. step distance ↓	22.5 ± 3	16.9 ± 0.7	15.8 ± 1.6	14.2 ± 0.7	18.9 ± 2.4	**14.1 ± 0.3**
RelWork			**Step-by-Step Constraint Generation**			
Step-index acc. (%) ↑	51.9 ± 4.6	52.6 ± 1.3	**61.5 ± 4.2**	54.5 ± 3.2	55.6 ± 4.8	61.5 ± 2.1
Category acc. (%) ↑	46.7 ± 3.1	48.1 ± 2.6	55.6 ± 1.8	49.2 ± 2.8	51.9 ± 5.5	**59.3 ± 2.1**
Avg. step distance ↓	2 ± 0.2	1.9 ± 0.2	**1.2 ± 0.3**	1.7 ± 0.4	1.5 ± 0.1	1.8 ± 0.4
RelWork			**One-shot Constraint Generation**			
Step-index acc. (%) ↑	51.9 ± 4.6	52.6 ± 1.3	47.4 ± 5.8	48.9 ± 3.6	**55.6 ± 4.8**	50.4 ± 2.1
Category acc. (%) ↑	46.7 ± 3.1	48.1 ± 2.6	43.7 ± 2.8	49.6 ± 4.2	51.9 ± 6.8	**54.1 ± 1.0**
Avg. step distance ↓	2 ± 0.2	1.9 ± 0.2	1.8 ± 0.1	1.9 ± 0.1	**1.5 ± 0.1**	1.8 ± 0.03

in specific frameworks (Figure 3); 7.6% of traces receive AGENTRX labels with no MAST counterpart. Step localization also varied: AppWorld failed early, while Magentic-One failed later, consistent with MAST's observation that verification and termination failures are common (Figure 4).

4.3. Impact of Violations and Checklist

Table 2 compares AGENTRX against a judge-only baseline that sees the trajectory and taxonomy prompt, but no AGENTRX-generated validation evidence. We isolate three added signals: violation evidence alone (Baseline+Vio.), checklist alone (Taxonomy Checklist), and checklist+violations (Checklist+Vio.). Violation evidence gives the clearest gains on τ-bench and RelWork. The checklist is most useful on Flash, suggesting that **semantic structure can outweigh sparse or noisy violations evidence.** On Magentic, combining checklist and violations gives the best step accuracy. **Overall, violations and the taxonomy checklist are complementary: violations help localize grounded errors, while combining both signals gives reliable attribution across domains.**

4.4. Impact of Constraint Generation

We evaluate two constraint-generation strategies: *one-shot*, which generates constraints once from the full trajectory, and *step-by-step*, which generates constraints from each prefix $T_{\leq k}$. One-shot is more cost-efficient, while step-by-step better matches online diagnosis by conditioning only on information available up to step k. As shown in Table 2, step-by-step constraint generation consistently outperforms one-shot generation across domains. The advantage is most visible on longer trajectories: on Magentic, step-by-step Checklist+Vio. reduces average step distance from 22.5 to 12.4. By contrast, Flash trajectories are shorter, and the two strategies perform more similarly. This trend aligns with trajectory length statistics in Table 5: longer traces are

Table 3. **Failure attribution accuracy across datasets.** We compare W&W modified (W&W*) and AGENTRX on failed-agent and critical-step accuracy. Mean $\pm$ std over 3 runs.

Dataset	Method	Agent Acc.	Step Acc.	Category Acc.
τ-bench	W&W*	65 ± 1.5	18.8 ± 5.3	–
	AGENTRX	**66.7**	$\mathbf{42.7 \pm 1.5}$	$\mathbf{47 \pm 1.5}$
Magentic	W&W*	61.4 ± 6.8	22.7 ± 3.9	–
	AGENTRX	$\mathbf{71.2 \pm 1.1}$	$\mathbf{35.6 \pm 1.3}$	$\mathbf{46.2 \pm 1.3}$
Flash	W&W*	16.1 ± 1.2	64.3 ± 3.9	–
	AGENTRX	$\mathbf{82.5 \pm 1.4}$	$\mathbf{82.5 \pm 1.4}$	$\mathbf{62.2 \pm 1.8}$
RelWork	W&W*	57 ± 6.8	32.6 ± 1.3	–
	AGENTRX	$\mathbf{65.9 \pm 5}$	$\mathbf{61.5 \pm 4.2}$	$\mathbf{59.3 \pm 2.1}$

more vulnerable to context dilution under one-shot generation, **making localized step-by-step constraint generation more reliable.**

This tradeoff is also reflected in the cost. Under the GPT-5 pricing assumptions used in our experiments, the judge-only baseline costs USD 0.05 per trajectory, while the one-shot AGENTRX configuration costs USD 0.18 per trajectory, about 3.6$\times$ higher. Most of this overhead comes from constraint generation, which accounts for roughly 81% of the total cost; the final judge stage is comparable to the baseline. Thus, **one-shot generation is a cost-efficient option when full-trajectory context is sufficient.**

4.5. LLM-as-a-Judge Evaluation

We compare two LLM-as-judge protocols: *All-at-Once*, which predicts the critical step and category in one call, and *Step-then-Category*, which first predicts the step and then categorizes that step. All-at-Once is the default AGENTRX setting. In Table 2, Step-then-Category often improves category accuracy, including on τ-bench, Flash, and Magentic, but it can compromise step localization because the category decision is conditioned on a potentially noisy first-stage step.

4.6. Robustness Across Relaxed Metrics

Step accuracy increases as we allow more tolerance in the predicted step (Acc@± 1 < Acc@± 3 < Acc@± 5) as shown in Table 4. For category prediction, we report: *critical* accuracy; *any-failure* accuracy, where predicted accuracy matches emphany category among the trajectory's failure steps; *earliest* accuracy, where it matches the first failure; and *terminal* accuracy, where it matches the last failure. Critical accuracy measures root-cause attribution, while any-failure accuracy captures the weaker but practical ability to flag a failure. **AGENTRX remains useful under both strict and relaxed failure-attribution metrics.**

5. Related Work

Benchmarks for Agent Evaluation. Recent work has introduced benchmarks that cover tool execution, web interaction, and assistant behavior (Barres et al., 2025; Yao et al., 2022; Drouin et al., 2024). AgentBench Liu et al. (2024) evaluates LLM agents across eight interactive environments that span code-centric, game-based, and web-based settings, including domains like operating systems and databases. WebArena Zhou et al. (2024) evaluates long-horizon web agents on real websites, testing whether they can convert language instructions into executable browser interactions. GAIA Mialon et al. (2023) targets general assistants with problems that are easy for humans but often require multi-step tool use, and has been used to evaluate generalist agent systems. However, these benchmarks primarily report terminal success and do not include gold failure-attribution annotations. Our dataset annotates the first unrecoverable critical step and assigns a failure category from a cross-domain taxonomy. As discussed above, Who&When is our closest comparison.

Agent Reliability. Koohestani (2025) provides runtime assurance for AI agents by monitoring executions against learned behavioral models. Recent work also translates plans into formal representations and applies model checking to validate alignment between an agent's behavior and its plan (Ramani et al., 2025; Zhang et al., 2025b). These methods primarily target runtime enforcement or plan-conformance checks; our focus is to diagnose trajectories by extracting constraints from tools, policies, and the observed prefix, and use the resulting violations as evidence for failure attribution. A separate line of work improves agent performance through feedback and self-correction. Self-Refine (Madaan et al., 2023) formalizes an iterative loop in which an LLM generates critiques of its own outputs and applies revisions Critic (Gou et al., 2024) uses external tools like search and code execution as feedback to verify outputs.

LLM Evaluation of Agent Trajectories. Gu et al. (2025) systematizes judge design and mitigation strategies for inconsistency and bias. Agent-as-a-Judge Zhuge et al. (2025) uses tool-using agents to evaluate other agents, arguing benefits over single-shot judges. AGENTRX is compatible with these paradigms, but shifts the judge from scoring to diagnosis by providing evidence.

6. Conclusion

Agents should not only be evaluated by whether they fail, but by where recovery becomes impossible. We presented AGENTRX, a domain-agnostic framework that diagnoses failed trajectories by generating constraints, surfacing evidence-based violations, and assigning root-cause cate-

Table 4. **Step and category accuracy under tolerance. Mean ± std over n=3 runs.**

Domain / Method	Step Acc. ↑	Acc@ ±1 ↑	Acc@ ±3 ↑	Acc@ ±5 ↑	Avg. Dist. ↓	Critical Cat. ↑	Any Cat. ↑	Earliest Cat. ↑	Terminal Cat. ↑
τ-Bench (Baseline)	29.9 ± 1.2	41.0 ± 2.1	53.8 ± 2.1	68.4 ± 3.2	5.4 ± 0.14	30.8 ± 2.1	42.1 ± 5.7	36.8 ± 7.4	33.3 ± 3.3
τ-Bench (AGENTRX)	42.7 ± 1.2	48.7 ± 4.2	61.5 ± 3.6	68.4 ± 2.4	3.8 ± 0.6	47.0 ± 1.2	50.9 ± 3.3	43.0 ± 3.3	43.9 ± 3.3
Flash (Baseline)	80.2 ± 1.4	94.4 ± 1.1	100	100	0.2	53.9 ± 3	62.7 ± 4.4	53.2 ± 2.9	62.7 ± 4.4
Flash (AGENTRX)	82.5 ± 1.4	93.7 ± 1.4	97.6	100	0.2	65.9 ± 5.0	65.9 ± 5.0	65.9 ± 5.0	65.9 ± 5.0
Magentic (Baseline)	32.6 ± 1.3	43.2 ± 3.2	43.9 ± 4.3	52.3 ± 4.9	22.5 ± 3	35.6 ± 1.3	50.8 ± 2.1	39.4 ± 2.1	34.1
Magentic (AGENTRX)	36.4	47 ± 5.4	54.6 ± 4.9	59.8 ± 2.8	12.4 ± 0.6	46.2 ± 1.3	59.1 ± 2.1	43.2 ± 2.8	39.4 ± 1.1
RelWork (Baseline)	51.9 ± 4.6	68.1 ± 3.8	84.4 ± 0.0	88.9 ± 1.8	2.0 ± 0.2	46.7 ± 3.1	46.7 ± 3.1	46.7 ± 3.1	46.7 ± 3.1
RelWork (AGENTRX)	61.5 ± 4.2	76.3 ± 2.1	92.6 ± 2.8	92.6 ± 2.8	1.2 ± 0.3	59.3 ± 2.1	59.3 ± 2.1	59.3 ± 2.1	59.3 ± 2.1

Table 5. Mean token and characters per trajectory.

Metric	τ-Bench	Flash	Magentic	RelWork
Avg tokens / step	133	169	330	393
Avg chars / step	480	930	1280	1711
Avg tokens / trajectory	4889	6415	16484	5837

gories. AGENTRX moves agent evaluation toward evidence-grounded debugging: explaining not just that an agent failed, but when and why.

7. Limitations

Our failure taxonomy covers diverse domains and annotations from three independent annotators. However, it may not cover all failure modes in other agentic domains and may require extension. Sometimes AGENTRX can be misled when validation signals are weak or contain false positives. Appendix B analyzes these failure cases in detail, showing that noisy or downstream violation signals can misdirect the judge. This motivates future work on identifying the smallest set of high-quality signals needed to separate true failures from noisy flags. Although AGENTRX relies on LLM calls, it reduces manual effort for failure attribution and is easier to scale. AGENTRX also offers cost-efficient one-shot constraint generation as an alternative. We view both the framework and the dataset as a step toward systematic, evidence-based failure attribution in AI agents.

References

Barres, V., Dong, H., Ray, S., Si, X., and Narasimhan, K. Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025.

Brynjolfsson, E., Li, D., and Raymond, L. R. Generative ai at work. Working Paper 31161, National Bureau of Economic Research, April 2023. URL http://www.nber.org/papers/w31161.

Cemri, M., Pan, M. Z., Yang, S., Agrawal, L. A., Chopra, B., Tiwari, R., Keutzer, K., Parameswaran, A., Klein, D., Ramchandran, K., et al. Why do multi-agent llm systems fail? *Advances in Neural Information Processing Systems*, 38, 2026.

Drouin, A., Gasse, M., Caccia, M., Laradji, I. H., Del Verme, M., Marty, T., Vazquez, D., Chapados, N., and Lacoste, A. Workarena: How capable are web agents at solving common knowledge work tasks? In *International Conference on Machine Learning*, pp. 11642–11662. PMLR, 2024.

Fourney, A., Bansal, G., Mozannar, H., Tan, C., Salinas, E., Zhu, E. E., Niedtner, F., Proebsting, G., Bassman, G., Gerrits, J., Alber, J., Chang, P., Loynd, R., West, R., Dibia, V., Awadallah, A., Kamar, E., Hosn, R., and Amershi, S. Magentic-one: A generalist multi-agent system for solving complex tasks. Technical Report MSR-TR-2024-47, Microsoft, November 2024. URL https://www.microsoft.com/en-us/research/publication/magentic-one-a-generalist-multi-agent-system-for-solving-complex-tasks/.

Glaser, B. and Strauss, A. *Discovery of grounded theory: Strategies for qualitative research*. Routledge, 2017.

Gou, Z., Shao, Z., Gong, Y., Shen, Y., Yang, Y., Duan, N., and Chen, W. Critic: Large language models can self-correct with tool-interactive critiquing, 2024. URL https://arxiv.org/abs/2305.11738.

Gu, J., Jiang, X., Shi, Z., Tan, H., Zhai, X., Xu, C., Li, W., Shen, Y., Ma, S., Liu, H., Wang, S., Zhang, K., Wang, Y., Gao, W., Ni, L., and Guo, J. A survey on llm-as-a-judge, 2025. URL https://arxiv.org/abs/2411.15594.

Koohestani, R. Agentguard: Runtime verification of ai agents, 2025. URL https://arxiv.org/abs/2509.23864.

LinkedIn. Hiring assistant for LinkedIn recruiter & jobs. https://business.linkedin.com/talent-solutions/hiring-assistant, 2024. Accessed: 2025-01-28.

Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., Gu, Y., Ding, H., Men, K., Yang, K., et al. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, volume 2024, pp. 52989–53046, 2024.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Majumder, B. P., Hermann, K., Welleck, S., Yazdanbakhsh, A., and Clark, P. Self-refine: Iterative refinement with self-feedback, 2023. URL https://arxiv.org/abs/2303.17651.

Mialon, G., Fourrier, C., Wolf, T., LeCun, Y., and Scialom, T. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*, 2023.

Ramani, K., Tawosi, V., Alamir, S., and Borrajo, D. Bridging llm planning agents and formal methods: A case study in plan verification, 2025. URL https://arxiv.org/abs/2510.03469.

Yao, S., Chen, H., Yang, J., and Narasimhan, K. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757, 2022.

Yao, S., Shinn, N., Razavi, P., and Narasimhan, K. τ-bench: A benchmark for tool-agent-user interaction in real-world domains, 2024. URL https://arxiv.org/abs/2406.12045.

Zhang, S., Yin, M., Zhang, J., Liu, J., Han, Z., Zhang, J., Li, B., Wang, C., Wang, H., Chen, Y., et al. Which agent causes task failures and when? on automated failure attribution of llm multi-agent systems. In *International Conference on Machine Learning*, pp. 76583–76599. PMLR, 2025a.

Zhang, X., Mittal, T., Bansal, C., Wang, R., Ma, M., Ren, Z., Huang, H., and Rajmohan, S. Flash: A workflow automation agent for diagnosing recurring incidents. October 2024. URL https://www.microsoft.com/en-us/research/publication/flash-a-workflow-automation-agent-for-diagnosing-recurring-incidents/.

Zhang, Y., Cai, Y., Zuo, X., Luan, X., Wang, K., Hou, Z., Zhang, Y., Wei, Z., Sun, M., Sun, J., Sun, J., and Dong, J. S. Position: Trustworthy AI agents require the integration of large language models and formal methods. In Singh, A., Fazel, M., Hsu, D., Lacoste-Julien, S., Berkenkamp, F., Maharaj, T., Wagstaff, K., and Zhu, J. (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 82441–82459. PMLR, 13–19 Jul 2025b. URL https://proceedings.mlr.press/v267/zhang25ds.html.

Zhou, S., Xu, F. F., Zhu, H., Zhou, X., Lo, R., Sridhar, A., Cheng, X., Ou, T., Bisk, Y., Fried, D., et al. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024.

Zhuge, M., Zhao, C., Ashley, D. R., Wang, W., Khizbullin, D., Xiong, Y., Liu, Z., Chang, E., Krishnamoorthi, R., Tian, Y., et al. Agent-as-a-judge: Evaluate agents with agents. In *International Conference on Machine Learning*, pp. 80569–80611. PMLR, 2025.

Impact Statement

This paper is focused on making AI agents easier to debug. As agents increasingly run multi-step workflows: calling tools, changing state, and coordinating sub-agents failures are difficult. They can be wrong tool calls, skipped confirmations, misread outputs, or decisions that push the run onto a path it never recovers from. Our goal is to help developers pinpoint where the run first became unrecoverable and why, using evidence already present in execution traces.

The main upside is practical: faster debugging, clearer accountability, and safer iteration. A system that can surface the root cause and connect it to trace evidence can make it easier to measure reliability. The benchmark and taxonomy also encourage better instrumentation and more careful thinking about what should be logged and what should be checked.

There are also risks. Better diagnosis does not automatically mean agents are safe to deploy, and explanations can be over-trusted if people treat them as guarantees. Trajectory data can also contain sensitive information depending on the application, so anyone using these methods should handle logs carefully and follow appropriate privacy practices. Finally, a taxonomy learned from existing benchmarks can inherit their blind spots, so it should be treated as a starting point rather than a complete map of real-world failures. Overall, we expect this work to be most useful as an Agentic debugging framework: it helps find problems earlier and fix them proactively, but it does not replace careful testing.

A. Benchmark and Taxonomy

Benchmark Format. Each entry corresponds to a trajectory and contains (i) an identifier, (ii) a record of failure events encountered during execution, and (iii) a critical failure. For each failure event, we store its step in the trajectory, a brief natural-language description, and a failure label from the taxonomy with supporting rationale. The list of recorded

Table 6. Comparison with related work. MAST (Cemri et al., 2026) emphasizes broad multi-agent failure characterization, W&W (Zhang et al., 2025a) focuses on responsible-agent and failure-step attribution, while AgentRx combines taxonomy-level analysis with evidence-grounded diagnosis of failed runs.

Criterion	MAST	W&W	AGENTRX
Failed-trace benchmark	✓	✓	✓
Broad multi-agent coverage	✓	✓	✓
Failure taxonomy	✓	–	✓
Cross-run failure analysis	✓	∼	✓
Responsible-agent attribution	–	✓	✓
Failure-step localization	∼	✓	✓
First unrecoverable failure	–	∼	✓
Evidence-grounded validation	–	–	✓

failures gives a causal chain from the first unrecoverable failure to the terminal one. The key design principles for our failure taxonomy are: (i) **Mutual exclusivity at the critical step.** While multiple failure modes may appear across a trajectory, the critical failure step receives exactly one category label. This label captures the primary reason that the step results in an unrecoverable failure; (ii) **Disambiguation criteria.** Categories such as *Plan Adherence Failure*, where the agent deviates from the correct plan, and *Intent–Plan Misalignment*, where the agent forms the wrong plan from the correct intent, are distinguished by whether the plan itself is correct. We operationalize this distinction through the taxonomy checklist, which provides targeted yes/no questions for each category; and (iii) **Extensibility.** Although the taxonomy is currently embedded as a fixed set in the prompts, extending it requires only: (i) defining new category questions in the checklist K, and (ii) adding them to the judge prompt. The grounded-theory methodology can likewise be applied to new domains to discover additional categories.

B. Error Analysis

We manually inspect cases where AGENTRX produces an incorrect or misleading diagnosis. Broadly, the errors fall into two categories. First, surfaced violations can misdirect the judge toward an incorrect error category or toward an inconclusive diagnosis. Second, surfaced violations can capture downstream symptoms or secondary failures rather than the true root cause.

Surfaced violations can misdirect the judge. In some cases, the violation evidence presented to the judge points toward the wrong type of failure. This is especially likely when similar low-level assertions can be triggered by both agent misuse and external system failures.

For example, in the Flash setting, the correct label is *System Failure*, since the underlying problem is infrastructural: the Kusto query failed because of a backend or endpoint issue rather than because the agent chose an invalid action. Concretely, the relevant step is better read as something like

```
KustoAgent → run_kusto_query(args)
           → network/endpoint error.
```

not as an invalidly formed tool call. However, the violation log presented to the judge is dominated by invocation-oriented checks, such as assertions about predefined queries, correct cluster selection, endpoint validity, and repeated retries. This case highlights a limitation of the current setup: when similar low-level assertions can be triggered by both tool misuse and backend failure, the judge may over-attribute the problem to the agent.

Surfaced violations can capture downstream symptoms. A second class of errors occurs when the surfaced violations identify a real problem in the trajectory, but one that is downstream of the true root cause. In these cases, the violation evidence can shift the judge toward a secondary failure rather than the first unrecoverable error.

For example, in Magentic-One task `1f975693-876d-457b-a649-393859e79bf`, the correct label is *Invention of New Information*: the Orchestrator hallucinated that a PDF was successfully downloaded and issued instructions based on that false assumption. AgentRx's checklist focused instead on a late-trajectory guardrail block at step 51, where a `ResponsibleAIPolicyViolation` occurred and the Orchestrator nevertheless emitted a `FINAL ANSWER` that was ungrounded. This shifted the diagnosis to *Guardrails Triggered*, because the violation evidence pointed at a downstream symptom rather than the upstream hallucination.

C. Root-Cause Attribution VS First Failure

The following trajectory demonstrates why root-cause failure attribution is the better objective for evaluating agent reliability. The Who&When dataset labels step 3 as failure citing that "WebSurfer's inability to reliably access the requested documents resulted in the overall task failure, as the necessary time span data could not be extracted or compared. This underscores the need for enhanced fallback mechanisms and more robust search strategies." This diagnosis misses a key fact: the agent recovers from this early access failure, so it is not the cause of the eventual outcome. We annotate the key failure as shown below:

```
{
"trajectory_id": "5f982798-16b9-4051-ab57
↪ -cfc7ebdb2a91",
"failures": [
```

Table 7. o3 Judge Ablation on Flash and RelWork (n=3 runs, 42/45 tasks each).

		Judge Baselines		AGENTRX Ablations			
Domain	Metric (n=3)	Baseline	Step-then-Cat.	Baseline+Vio.	Step-then-Cat.+Vio.	Taxonomy Checklist	Checklist+Vio.
Flash	Step-index acc. (%) ↑	61.9 ± 4.8	60.3 ± 3.6	65.9 ± 5.0	63.5 ± 5.5	60.3 ± 3.6	**69.8 ± 1.4**
	Category acc. (%) ↑	50.0 ± 6.3	48.4 ± 3.6	50.0 ± 4.8	53.2 ± 5.0	54.8 ± 4.1	**55.6 ± 1.4**
	Avg. step distance ↓	0.71 ± 0.27	0.52 ± 0.06	0.52 ± 0.10	0.57 ± 0.13	0.67 ± 0.16	**0.44 ± 0.04**
RelWork	Step-index acc. (%) ↑	52.6 ± 3.8	48.9 ± 1.8	**54.8 ± 4.2**	49.6 ± 1.0	45.9 ± 4.2	**54.8 ± 2.8**
	Category acc. (%) ↑	**52.6 ± 4.2**	45.9 ± 4.2	44.4 ± 5.4	45.9 ± 2.8	43.7 ± 5.8	46.7 ± 1.8
	Avg. step distance ↓	2.6 ± 0.1	**2.1 ± 0.5**	2.7 ± 0.4	2.4 ± 0.1	3.2 ± 0.4	2.5 ± 0.1


```
{
    "failure_id": 1,
    "step_number": 13,
    "step_reason": "Websurfer could not
    ↪  download a PDF file and search
    ↪  throught it which was an
    ↪  instruction given by
    ↪  Orchestrator.",
    "failure_category": "Instruction/Plan
    ↪  Adherence Failure",
    "category_reason": "Instruction not
    ↪  followed, the agent did not
    ↪  download and search through the
    ↪  PDF file as instructed",
    "failed_agent": "Websurfer"
},
{
    "failure_id": 2,
    "step_number": 17,
    "step_reason": "Websurfer could not
    ↪  download a PDF file and search
    ↪  throught it which was an
    ↪  instruction given by
    ↪  Orchestrator",
    "failure_category": "Instruction/Plan
    ↪  Adherence Failure",
    "category_reason": "Websurfer could
    ↪  not  download a PDF file and
    ↪  search throught it which was an
    ↪  instruction given by
    ↪  Orchestrator",
    "failed_agent": "Websurfer"
},
{
    "failure_id": 3,
    "step_number": 33,
    "step_reason": "FileSurfer
    ↪  hallucinated. It downloaded the
    ↪  file at '/workspace/workspace/htt⌋
    ↪  p:/export.arxiv.org/pdf/2007.xx'
    ↪  but later attempted to read a
    ↪  non-existent file
    ↪  'file:///workspace/path_to_july_2⌋
    ↪  020_paper.pdf'.",
    "failure_category": "Invention of new
    ↪  information",
    "category_reason": "FileSurfer
    ↪  hallucinated. It downloaded the
    ↪  file at '/workspace/workspace/htt⌋
    ↪  p:/export.arxiv.org/pdf/2007.xx'
    ↪  but later attempted to read a
    ↪  non-existent file
    ↪  'file:///workspace/path_to_july_2⌋
    ↪  020_paper.pdf'.",
    "failed_agent": "FileSurfer"
}
],
"root_cause": {
"failure_id": 3,
"reason_for_root_cause": "The Orchestrator
↪  was able to recover from earlier
↪  errors but the FileSurfer
↪  hallucination was a critical failure
↪  that prevented further progress."
},
"failure_summary": "The agent could not
↪  download and read the specified PDF
↪  file due to a hallucination by the
↪  FileSurfer agent."
}
```


Failures #1–#2 were recoverable because the orchestrator successfully re-routed execution when WebSurfer stalled. It explicitly acknowledged missing information and switched to FileSurfer, whose file-centric capabilities are better suited for handling PDFs. In other words, the system demonstrated a working fallback: early retrieval failures triggered tool substitution and continued progress rather than terminal collapse. Failure #3 was the key failure because FileSurfer saved file to the wrong path and subsequent operations were based on non-existent data, making recovery impossible.

Figures 3 and 4 show the full breakdown.

D. Failure Annotation for Magentic One

An example annotation for a magentic trajectory in our benchmark. We annotate a total of 22 failures in this trajectory as shown:


```
{
"trajectory_id": "16d825ff-1623-4176-a5b5⌋
↪  -42e0f5c2b0ac",
```

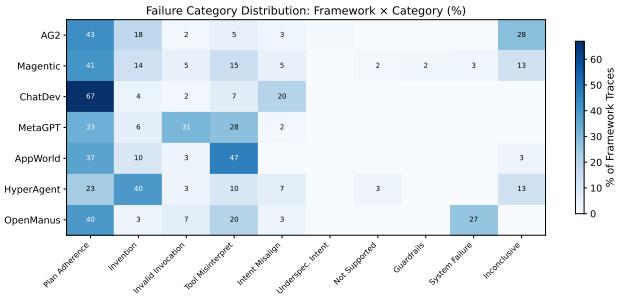

Figure 3. Failure category distribution (%) per framework on 1,022 MAST trajectories. Each row is normalized independently.

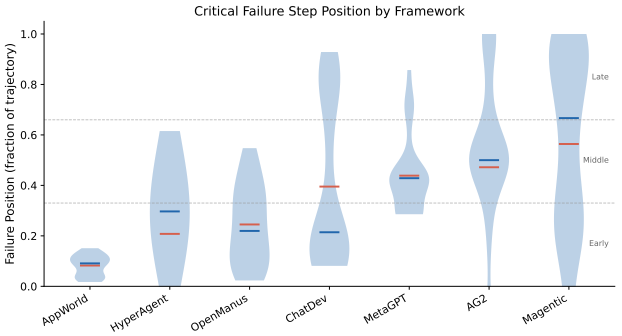

Figure 4. Distribution of critical failure step position (as fraction of trajectory length) by framework in the entire 1022 MAST dataset. Red marks indicate means; blue marks indicate medians.


```
"failures": [
{
    "failure_id": 1,
    "step_number": 5,
    "step_reason": "WebSurfer did not get
    ↪  arrival time information which is
    ↪  important",
    "failure_category": "Instruction/Plan
    ↪  Adherence Failure",
    "category_reason": "WebSurfer did not
    ↪  get arrival time information which
    ↪  is important",
    "failed_agent": "WebSurfer"
},
{
    "failure_id": 2,
    "step_number": 13,
    "step_reason": "WebSurfer did  scroll
    ↪  but did not get information for
    ↪  specified date",
    "failure_category": "Instruction/Plan
    ↪  Adherence Failure",
    "category_reason": "WebSurfer did
    ↪  scroll but did not get information
    ↪  for specified date",
"failed_agent": "WebSurfer"
},
{
    "failure_id": 3,
    "step_number": 17,
    "step_reason": "WebSurfer did not get
    ↪  information for specified date",
    "failure_category": "Instruction/Plan
    ↪  Adherence Failure",
    "category_reason": "WebSurfer did not
    ↪  get information for specified
    ↪  date",
"failed_agent": "WebSurfer"
},
{
"failure_id": 4,
"step_number": 25,
"step_reason": "WebSurfer did not get
↪  information for specified date",
"failure_category": "Instruction/Plan
↪  Adherence Failure",
"category_reason": "WebSurfer did not get
↪  information for specified date",
"failed_agent": "WebSurfer"
},
{
    "failure_id": 5,
    "step_number": 29,
    "step_reason": "WebSurfer did not get
    ↪  information for specified date",
    "failure_category": "Instruction/Plan
    ↪  Adherence Failure",
    "category_reason": "WebSurfer did not
    ↪  get information for specified
    ↪  date",
    "failed_agent": "WebSurfer"
},
{
"failure_id": 6,
"step_number": 33,
"step_reason": "WebSurfer did not get
↪  information for specified date",
"failure_category": "Instruction/Plan
↪  Adherence Failure",
"category_reason": "WebSurfer did not get
↪  information for specified date",
"failed_agent": "WebSurfer"
},
{
    "failure_id": 7,
    "step_number": 37,
    "step_reason": "WebSurfer did not get
    ↪  information for specified date",
    "failure_category": "Instruction/Plan
    ↪  Adherence Failure",
    "category_reason": "WebSurfer did not
    ↪  get information for specified
    ↪  date",
    "failed_agent": "WebSurfer"
},
{
"failure_id": 8,
"step_number": 41,
"step_reason": "WebSurfer did not get
↪  information for specified date",
"failure_category": "Instruction/Plan
↪  Adherence Failure",
"category_reason": "WebSurfer did not get
↪  information for specified date",
"failed_agent": "WebSurfer"
},
```

```
{
    "failure_id": 9,
    "step_number": 56,
    "step_reason": "WebSurfer did not get
    ↪  information for specified date",
    "failure_category": "Instruction/Plan
    ↪  Adherence Failure",
    "category_reason": "WebSurfer did not
    ↪  get information for specified
    ↪  date",
    "failed_agent": "WebSurfer"
},
{
    "failure_id": 10,
    "step_number": 60,
    "step_reason": "WebSurfer did not get
    ↪  information for specified date",
    "failure_category": "Instruction/Plan
    ↪  Adherence Failure",
    "category_reason": "WebSurfer did not
    ↪  get information for specified
    ↪  date",
    "failed_agent": "WebSurfer"
},
{
    "failure_id": 10,
    "step_number": 60,
    "step_reason": "WebSurfer did not get
    ↪  information for specified date",
    "failure_category": "Instruction/Plan
    ↪  Adherence Failure",
    "category_reason": "WebSurfer did not
    ↪  get information for specified
    ↪  date",
    "failed_agent": "WebSurfer"
},
{
    "failure_id": 11,
    "step_number": 66,
    "step_reason": "Orchestrator try to
    ↪  contact through email which might
    ↪  not be good strategy",
    "failure_category": "Intent Plan
    ↪  Misalignment",
    "category_reason": "Orchestrator try
    ↪  to contact through email which
    ↪  might not be good strategy",
    "failed_agent": "Orchestrator"
},
{
    "failure_id": 12,
    "step_number": 70,
    "step_reason": "Orchestrator did not
    ↪  properly interpret user intent
    ↪  again trying to email which is not
    ↪  good strategy",
    "failure_category": "Intent Plan
    ↪  Misalignment",
    "category_reason": "Orchestrator did
    ↪  not properly interpret user intent
    ↪  again trying to email which is not
    ↪  good strategy",
    "failed_agent": "Orchestrator"
},
{
    "failure_id": 13,
    "step_number": 74,
    "step_reason": "Orchestrator did not
    ↪  properly interpret user intent
    ↪  again trying to email which is not
    ↪  good strategy",
    "failure_category": "Intent Plan
    ↪  Misalignment",
    "category_reason": "Orchestrator did
    ↪  not properly interpret user intent
    ↪  again trying to email which is not
    ↪  good strategy",
    "failed_agent": "Orchestrator"
},
{
    "failure_id": 14,
    "step_number": 78,
    "step_reason": "Orchestrator did not
    ↪  properly interpret user intent
    ↪  again trying to email which is not
    ↪  good strategy",
    "failure_category": "Intent Plan
    ↪  Misalignment",
    "category_reason": "Orchestrator did
    ↪  not properly interpret user intent
    ↪  again trying to email which is not
    ↪  good strategy",
    "failed_agent": "Orchestrator"
},
{
    "failure_id": 15,
    "step_number": 82,
    "step_reason": "Orchestrator did not
    ↪  properly interpret user intent
    ↪  again trying to email which is not
    ↪  good strategy",
    "failure_category": "Intent Plan
    ↪  Misalignment",
    "category_reason": "Orchestrator did
    ↪  not properly interpret user intent
    ↪  again trying to email which is not
    ↪  good strategy",
    "failed_agent": "Orchestrator"
},
{
    "failure_id": 16,
    "step_number": 86,
    "step_reason": "Orchestrator did not
    ↪  properly interpret user intent
    ↪  again trying to email which is not
    ↪  good strategy",
    "failure_category": "Intent Plan
    ↪  Misalignment",
    "category_reason": "Orchestrator did
    ↪  not properly interpret user intent
    ↪  again trying to email which is not
    ↪  good strategy",
    "failed_agent": "Orchestrator"
},
{
    "failure_id": 17,
    "step_number": 90,
    "step_reason": "Orchestrator did not
    ↪  properly interpret user intent
    ↪  again trying to email which is not
    ↪  good strategy",
```

```
    "failure_category": "Intent Plan
    ↪  Misalignment",
    "category_reason": "Orchestrator did
    ↪  not properly interpret user intent
    ↪  again trying to email which is not
    ↪  good strategy",
    "failed_agent": "Orchestrator"
},
{
    "failure_id": 18,
    "step_number": 101,
    "step_reason": "Orchestrator did not
    ↪  properly interpret user intent
    ↪  again trying to email which is not
    ↪  good strategy",
    "failure_category": "Intent Plan
    ↪  Misalignment",
    "category_reason": "Orchestrator did
    ↪  not properly interpret user intent
    ↪  again trying to email which is not
    ↪  good strategy",
    "failed_agent": "Orchestrator"
},
{
    "failure_id": 19,
    "step_number": 105,
    "step_reason": "Orchestrator did not
    ↪  properly interpret user intent
    ↪  again trying to email which is not
    ↪  good strategy",
    "failure_category": "Intent Plan
    ↪  Misalignment",
    "category_reason": "Orchestrator did
    ↪  not properly interpret user intent
    ↪  again trying to email which is not
    ↪  good strategy",
    "failed_agent": "Orchestrator"
},
{
    "failure_id": 20,
    "step_number": 109,
    "step_reason": "Orchestrator did not
    ↪  properly interpret user intent
    ↪  again trying to email which is not
    ↪  good strategy",
    "failure_category": "Intent Plan
    ↪  Misalignment",
    "category_reason": "Orchestrator did
    ↪  not properly interpret user intent
    ↪  again trying to email which is not
    ↪  good strategy",
    "failed_agent": "Orchestrator"
},
{
    "failure_id": 21,
    "step_number": 116,
    "step_reason": "Orchestrator did not
    ↪  properly interpret user intent
    ↪  again trying to email which is not
    ↪  good strategy",
    "failure_category": "Intent Plan
    ↪  Misalignment",
    "category_reason": "Orchestrator did
    ↪  not properly interpret user intent
    ↪  again trying to email which is not
    ↪  good strategy",
    "failed_agent": "Orchestrator"
},
{
    "failure_id": 22,
    "step_number": 120,
    "step_reason": "Orchestrator did not
    ↪  properly interpret user intent
    ↪  again trying to email/phone which
    ↪  is not good strategy",
    "failure_category": "Intent Plan
    ↪  Misalignment",
    "category_reason": "Orchestrator did
    ↪  not properly interpret user intent
    ↪  again trying to email/phone which
    ↪  is not good strategy",
    "failed_agent": "Orchestrator"
}
"root_cause": {
"failure_id": 1,
"reason_for_root_cause": "The primary root
↪  cause of the failures in this
↪  trajectory is the WebSurfer's
↪  inability to retrieve specific arrival
↪  time information for the specified
↪  date. This lack of crucial information
↪  led to a series of misinterpretations
↪  by the Orchestrator, which repeatedly
↪  attempted to contact through email a
↪  strategy misaligned with the user's
↪  intent. The cascading effect of these
↪  initial shortcomings resulted in
↪  multiple failures throughout the
↪  trajectory."
}
}
```


E. Agent Trajectories for each category of our failure taxonomy across domains

E.1. System Failure (Flash)


```
{
"trajectory_id":
↪  "9_withouths_tip_session_2_417931231",
"failures": [
{
"failure_id": 1,
"step_number": 3,
"step_reason": "KustoApiError: Error
↪  getting schema for
↪  Cluster='https://azcore1.southeastasi
↪  a.kusto.windows.net/': Failed to
↪  connect to the remote cluster:
↪  InternalServiceError
↪  (520-UnknownError) followed by a
↪  SyntaxError of the Kusto query",
"failure_category": "System Failure",
"category_reason": "Connection failure
↪  error, system error + syntax error",
"failed_agent": "KustoAgent"
}
],
"root_cause": {
    "failure_id": 1,
```

```
    "reason_for_root_cause": "Connection
    ↪  failure error, system error +
    ↪  syntax error"
    },
    "failure_summary": "System failure +
    ↪  Syntax errors"
}
```


E.2. Misinterpretation of Tool Output (τ-bench)


```
"trajectory_id": 2,
"failures": [
{
"failure_id": 1,
"step_number": 3,
"step_reason": "At step 3, the assistant
↪  agent did not authenticate user
↪  information before proceeding to
↪  provide information about available
↪  t-shirts",
"failure_category": "Instruction Adherence
↪  Failure",
"category_reason": "The assistant agent
↪  did not follow the expected policy of
↪  authenticating user information before
↪  providing product details.",
"failed_agent": "Assistant"
},
{
"failure_id": 2,
"step_number": 7,
"step_reason": "At step 7, the agent did
↪  not correctly count the number of
↪  available t-shirts from the tool call
↪  result.",
"failure_category": "Misinterpretation of
↪  Tool Output",
"category_reason": "The assistant
↪  misinterpreted the output from the
↪  tool call, leading to an incorrect
↪  count of available t-shirts.",
"failed_agent": "Assistant"
  }
],
"root_cause": {
  "failure_id": 2,
  "reason_for_root_cause": "The assistant
  ↪  finally did authenticate before
  ↪  providing user specific information.
  ↪  The incorrect count does not
  ↪  correspond with ground truth
  ↪  output."
},
"failure_summary": "The agent did not
↪  correctly count the number of
↪  available t-shirts from the tool call
↪  result."
  }
```


E.3. Instruction Adherence Failure (Flash)


```
{
"trajectory_id":
↪  "7_withhs_tip_session_2_424614956",
"failures": [
{
"failure_id": 1,
"step_number": 4,
"step_reason": "The actual solution is to
↪  \"Delete the VM through the provided
↪  link, or contact the resource owner to
↪  delete it.\" The model's answer does
↪  not explicitly suggest deleting the VM
↪  or contacting the resource owner to do
↪  so. However, it does guide the user to
↪  manually inspect and clean up any
↪  lingering VMs or resources, which
↪  partially aligns with the intent of
↪  deleting the resource. The model fails
↪  to mention using a provided link or
↪  directly contacting the resource
↪  owner, which are key steps in the
↪  actual solution. Thus, while there is
↪  some overlap specifically the
↪  suggestion to clean up resources the
↪  solution is incomplete and misses key
↪  instructions.",
"failure_category": "Instruction/Plan
↪  Adherence Failure",
"category_reason": "incomplete/absent
↪  conclusion/mitigation step and also
↪  did not provide the Azure link.",
"failed_agent": "Orchestrator"
}
],
"root_cause": {
"failure_id": 1,
"reason_for_root_cause":
↪  "incomplete/absent
↪  conclusion/mitigation step and also
↪  did not provide the Azure home link"
},
"failure_summary": "The model's answer
↪  does not follow the plan perfectly"
}
```


E.4. Invention of New Information (Flash)


```
{
"trajectory_id":
↪  "9_withouths_tip_session_1_445308210",
"failures": [
{
"failure_id": 1,
"step_number": 3,
"step_reason": "Step 3 incorrect query,
↪  even though the Kusto query returned
↪  None; the agent tried with a new
↪  invented python script.",
"failure_category": "Invention of New
↪  Information",
"category_reason": "hallucination of
↪  python script",
"failed_agent": "Coder"
},
{
```

```
"failure_id": 2,
"step_number": 5,
"step_reason": "The GeneralAssistant came
↪  up with a link https://portal.azure.c
↪  om/#search/152076538 instead of
↪  providing the home page as per the
↪  plan.",
"failure_category": "Invention of New
↪  Information",
"category_reason": "hallucination of
↪  link",
"failed_agent": "GeneralAssistant"
}
],
"root_cause": {
  "failure_id": 1,
  "reason_for_root_cause": "hallucination
  ↪  of python script + link"
},
"failure_summary": "hallucination, extra
↪  steps executed"
}
```


E.5. Intent Not Supported (Magentic)


```
{
"trajectory_id": "a1e91b78-d3d8-4675-bb8d
↪  -62741b4b68a6",
"failures": [
{
"failure_id": 1,
"step_number": 5,
"step_reason": "Websurfer was asked to
↪  take snapshot in youtube video but it
↪  could not do such actions",
"failure_category": "Intent not
↪  supported",
"category_reason": "Websurfer was asked to
↪  take snapshot in youtube video but it
↪  could not do such actions",
"failed_agent": "Websurfer"
},
{
"failure_id": 2,
"step_number": 9,
"step_reason": "Websurfer was asked to
↪  take snapshot in youtube video but it
↪  could not do such actions",
"failure_category": "Intent not
↪  supported",
"category_reason": "Websurfer was asked to
↪  take snapshot in youtube video but it
↪  could not do such actions",
"failed_agent": "Websurfer"
},
{
"failure_id": 3,
"step_number": 13,
"step_reason": "Websurfer was asked to
↪  take snapshot in youtube video but it
↪  could not do such actions",
"failure_category": "Intent not
↪  supported",
"category_reason": "Websurfer was asked to
↪  take snapshot in youtube video but it
↪  could not do such actions",
"failed_agent": "Websurfer"
},
{
"failure_id": 4,
"step_number": 17,
"step_reason": "Websurfer was asked to
↪  take snapshot in youtube video but it
↪  could not do such actions",
"failure_category": "Intent not
↪  supported",
"category_reason": "Websurfer was asked to
↪  take snapshot in youtube video but it
↪  could not do such actions",
"failed_agent": "Websurfer"
},
{
"failure_id": 5,
"step_number": 21,
"step_reason": "Websurfer was asked to
↪  take snapshot in youtube video but it
↪  could not do such actions",
"failure_category": "Intent not
↪  supported",
"category_reason": "Websurfer was asked to
↪  take snapshot in youtube video but it
↪  could not do such actions",
"failed_agent": "Websurfer"
},
{
"failure_id": 6,
"step_number": 25,
"step_reason": "Websurfer was asked to
↪  take snapshot in youtube video but it
↪  could not do such actions",
"failure_category": "Intent not
↪  supported",
"category_reason": "Websurfer was asked to
↪  take snapshot in youtube video but it
↪  could not do such actions",
"failed_agent": "Websurfer"
}
],
"root_cause": {
"failure_id": 1,
"reason_for_root_cause": "The first error
↪  occurred when the Websurfer agent was
↪  asked to take a snapshot in a YouTube
↪  video, which is beyond its
↪  capabilities. Subsequent failures were
↪  repetitions of this unsupported
↪  intent."
},
"failure_summary": "The Websurfer agent
↪  was repeatedly asked to perform an
↪  action (taking snapshots in a YouTube
↪  video) that it is not capable of
↪  executing, leading to failure."
```


E.6. Underspecified User Intent (τ-bench)


```
{
```

```
"trajectory_id": 71,
"failures": [
{
"failure_id": 26,
"step_number": 24,
"step_reason": "At step 24, the task
↪  instruction did not specify the type
↪  of black lamp and since there are
↪  multiple black lamps available, it
↪  wont be possible to match the ground
↪  truth actions.",
"failure_category": "Underspecified User
↪  Intent",
"category_reason": "The task instruction
↪  is underspecified regarding the type
↪  of black lamp to be added to the
↪  cart.",
"failed_agent": "User"
},
{
"failure_id": 27,
"step_number": 27,
"step_reason": "Assistant prematurely
↪  called modify items tool on a pending
↪  order which locked the order and the
↪  user later on wasn't able to change
↪  the backpack that he wanted.",
"failure_category": "Intent-Plan
↪  Misalignment",
"category_reason": "Here, the plan
↪  generated by the assistant is
↪  incorrect, it should not have
↪  prematurely called modify items tool
↪  on a pending order before finalizing
↪  all the items to be modified in the
↪  user's order.",
"failed_agent": "Assistant"
}
],
"root_cause": {
"failure_id": 26,
"reason_for_root_cause": "The task
↪  instruction did not specify the type
↪  of black lamp and since there are
↪  multiple black lamps available, it led
↪  to incorrect matching of the ground
↪  truth actions. Did not recover from
↪  the error."
},
"failure_summary": "The task instruction
↪  did not specify the type of black lamp
↪  and since there are multiple black
↪  lamps available, it wont be possible
↪  to match the ground truth actions."
  }
```


E.7. Guardrails Triggered (Magentic)


```
{
 "task_instruction": "You are a business
 ↪  analyst evaluating cloud storage
 ↪  solutions. You need to compare the
 ↪  current pricing plans for DataVault's
 ↪  Pro and Enterprise tiers to make an
 ↪  informed purchasing decision for your
 ↪  company. Request the agent to find
 ↪  and summarize the pricing information
 ↪  including features and costs for both
 ↪  tiers.",
 "trajectory_snippet": [
{
    "content": "Can you find the current
    ↪  pricing plans for the Pro and
    ↪  Enterprise tiers on the DataVault
    ↪  cloud storage service? I need to
    ↪  compare them for my business
    ↪  decision.",
    "role": "human",
    "index": 1
},
{
    "content": "I'll help you find that
    ↪  pricing information. Let me
    ↪  navigate to DataVault's website
    ↪  and locate their pricing
    ↪  details.",
    "role": "Orchestrator",
    "index": 2
},
{
    "content": "Please navigate to
    ↪  DataVault's official website, find
    ↪  the pricing page, and extract the
    ↪  pricing and features for both Pro
    ↪  and Enterprise tiers.",
    "role": "Orchestrator (-> WebSurfer)",
    "index": 3
},
{
```

```
    "content": "I navigated to
    ↪  'https://www.datavault.io/pricing␣
    ↪  '.\n\n**Detected blocking
    ↪  mechanism:**\n\nThe page
    ↪  displays:\n\n\"Access
    ↪  Denied\n\nYou do not have
    ↪  permission to access this
    ↪  page.\n\nReason: Our system has
    ↪  detected that your request appears
    ↪  to be coming from an automated bot
    ↪  or script.\n\nTo protect our users
    ↪  and prevent abuse, we require
    ↪  human verification for pricing
    ↪  information.\n\nError Code: 403 -
    ↪  Forbidden (Bot
    ↪  Detection)\nReference ID:
    ↪  CDN-BLK-98472\n\nIf you are a
    ↪  human user, please:\n1. Enable
    ↪  JavaScript in your browser\n2.
    ↪  Complete the CAPTCHA challenge
    ↪  below\n3. Contact
    ↪  support@datavault.io if this
    ↪  problem persists\n\n[CAPTCHA
    ↪  Challenge - Please verify you are
    ↪  human]\n\nThis page is protected
    ↪  by Cloudflare Bot
    ↪  Management.\"\n\n**Navigation
    ↪  blocked:** The website has
    ↪  detected automated access and
    ↪  requires human verification
    ↪  (CAPTCHA) to proceed. I cannot
    ↪  bypass this security measure as it
    ↪  violates the site's terms of
    ↪  service and my operational
    ↪  guidelines.\n\nI am unable to
    ↪  access the pricing information due
    ↪  to the bot detection and CAPTCHA
    ↪  requirement.",
    "role": "WebSurfer",
    "index": 4
},
{
    "content": "I see the direct access is
    ↪  blocked. Let me try searching for
    ↪  the pricing information from
    ↪  alternative sources or cached
    ↪  content.",
    "role": "Orchestrator",
    "index": 5
},
{
    "content": "Please search for
    ↪  'DataVault Pro Enterprise pricing
    ↪  comparison' or try to find cached
    ↪  or third-party sources that list
    ↪  this pricing information.",
    "role": "Orchestrator (-> WebSurfer)",
    "index": 6
},
{
    "content": "I performed a search for
    ↪  'DataVault Pro Enterprise pricing
    ↪  comparison'.\n\nThe search results
    ↪  show several third-party review
    ↪  sites and comparison pages, but
    ↪  when attempting to access detailed
    ↪  pricing information, I encountered
    ↪  similar restrictions:\n\n1.
    ↪  **SoftwareReviews.com** - Pricing
    ↪  details require login\n2.
    ↪  **TechCompare.io** - Shows
    ↪  \"Contact for pricing\" with no
    ↪  specific numbers\n3. **G2.com** -
    ↪  Lists features but pricing marked
    ↪  as \"Available upon request\"\n4.
    ↪  **Cached pages** - Pricing
    ↪  information is outdated (from
    ↪  2022) and may not be
    ↪  current\n\nUnfortunately,
    ↪  DataVault appears to keep their
    ↪  Enterprise tier pricing behind
    ↪  access controls across multiple
    ↪  platforms. Most sources either
    ↪  require account
    ↪  registration/login, display
    ↪  \"Contact sales\" instead of
    ↪  actual prices, or have outdated
    ↪  cached information.\n\nI cannot
    ↪  retrieve the current, accurate
    ↪  pricing information the user needs
    ↪  due to these access
    ↪  restrictions.",
    "role": "WebSurfer",
    "index": 7
},
{
    "content": "I apologize, but I was
    ↪  unable to retrieve the current
    ↪  pricing information for
    ↪  DataVault's Pro and Enterprise
    ↪  tiers. The DataVault website has
    ↪  bot detection and CAPTCHA
    ↪  protection that prevents automated
    ↪  access to their pricing page
    ↪  (Error Code: 403 - Forbidden).
    ↪  Additionally, alternative sources
    ↪  like review sites and comparison
    ↪  platforms either require login
    ↪  credentials, show \"Contact sales
    ↪  for pricing,\" or contain outdated
    ↪  information.\n\nTo get this
    ↪  information, you would need
    ↪  to:\n1. Visit
    ↪  https://www.datavault.io/pricing
    ↪  directly in your browser\n2.
    ↪  Complete the CAPTCHA
    ↪  verification\n3. Or contact
    ↪  DataVault's sales team at
    ↪  support@datavault.io for
    ↪  Enterprise pricing details\n\nThis
    ↪  limitation is due to website
    ↪  access restrictions, not a
    ↪  capability issue with our search
    ↪  approach.",
    "role": "Assistant",
```

```
    "index": 8
}
 ],
    "reason_for_category": "The category
    ↪ is guardrails triggered because at
    ↪ step 4, the WebSurfer agent
    ↪ encountered an explicit access
    ↪ denial when attempting to navigate
    ↪ to DataVault's pricing page. The
    ↪ website returned a 403 Forbidden
    ↪ error with bot detection,
    ↪ requiring CAPTCHA verification to
    ↪ proceed. This is an external site
    ↪ access restriction (Cloudflare Bot
    ↪ Management) that blocks automated
    ↪ agents from accessing the content.
    ↪ The agent's plan was valid and
    ↪ would have worked if this access
    ↪ restriction were removed - it
    ↪ correctly identified the target
    ↪ URL and attempted proper
    ↪ navigation. At step 7, alternative
    ↪ approaches (searching third-party
    ↪ sources, cached pages) also failed
    ↪ due to similar access controls
    ↪ (login requirements, paywalls).
    ↪ This is not a malformed tool
    ↪ invocation (the navigation
    ↪ commands were correct), not an
    ↪ infrastructure failure (the
    ↪ systems are functioning as
    ↪ designed), and not a planning
    ↪ error (the approach was
    ↪ appropriate). The failure is
    ↪ purely due to external guardrails
    ↪ designed to prevent automated
    ↪ access to the pricing
    ↪ information."
}
```


E.8. Intent Plan Misalignment (τ-bench)


```
{
"trajectory_id": 28,
"failures": [
{
"failure_id": 13,
"step_number": 33,
"step_reason": "At step 33,  the assistant
↪ mistakenly believes that it can cancel
↪ a subset of a pending order which is
↪ not allowed as per domain policy, as a
↪ result the entire order got cancelled
↪ instead of just the garden hose.",
"failure_category": "Intent-Plan
↪ Misalignment",
"category_reason": "The assistant came up
↪ with an incorrect plan based on a
↪ wrong assumption that a subset of an
↪ order can be cancelled which violates
↪ the domain policy.",
"failed_agent": "Assistant"
}
],
"root_cause": {
"failure_id": 13,
"reason_for_root_cause": "The assistant
↪ called cancel order on the entire
↪ order which led to an incorrect final
↪ outcome as compared to ground truth
↪ actions."
},
"failure_summary": "Assistant mistakenly
↪ believes that it can cancel a subset
↪ of the pending order which is not
↪ allowed as per domain policy, as a
↪ result the entire order got cancelled
↪ instead of just the hose."
}
```


E.9. Invalid Invocation (τ-bench)


```
{
"trajectory_id": 34,
"failures": [
{
"failure_id": 16,
"step_number": 17,
"step_reason": "At step 17, the assistant
↪ uses modify order to cancel a subset
↪ of orders, however modify orders also
↪ need to have a replacement, which it
↪ did not provide resulting in an
↪ illegal tool call",
"failure_category": "Invalid Invocation",
"category_reason": "The assistant calls
↪ the modify order tool with invalid
↪ arguments.",
"failed_agent": "Assistant"
},
{
"failure_id": 17,
"step_number": 21,
"step_reason": "At step 21, the assistant
↪ tries to bypass the modify tool
↪ argument restriction by trying to
↪ modify the item_id with the same
↪ item_id in order to try to cancel it",
"failure_category": "Invalid Invocation",
"category_reason": "The assistant again
↪ calls modify order tool with invalid
↪ arguments trying to bypass the
↪ previous error.",
"failed_agent": "Assistant"
},
{
"failure_id": 18,
"step_number": 30,
"step_reason": "At step 30, the user does
↪ not ask the assistant to modify the
↪ address of the current pending order
↪ but is clearly a part of the overall
↪ task instruction.",
"failure_category": "Underspecified User
↪ Intent",
"category_reason": "The user does not ask
↪ the assistant to do all the tasks as
↪ mentioned in the task instruction,
↪ hence underspecifying its intent.",
```

```
"failed_agent": "User"
}
],
"root_cause": {
"failure_id": 16,
"reason_for_root_cause": "The assistant
↪  called the modify order tool with
↪  invalid arguments, leading to an
↪  illegal tool call, and the agent did
↪  not recover from this error."
},
"failure_summary": "Assistant used modify
↪  order to cancel a subset of orders,
↪  but modify order requires a
↪  replacement which was not provided -
↪  illegal tool call."
  }
```

F. Constraint Generation

F.1. AGENTRX Constraint Schema Output Format

```
{
  "assertion_name":
  ↪  "string_unique_snake_case",
  "taxonomy_targets": [
    "Instruction/PlanAdherenceFailure",
    "InventionOfNewInformation",
    "InvalidInvocation",
    "MisinterpretationOfToolOutput",
    "IntentPlanMisalignment",
    "UnderspecifiedUserIntent",
    "IntentNotSupported",
    "GuardrailsTriggered",
    "SystemFailure"
  ],
  "constraint_type": "SCHEMA | PROTOCOL |
  ↪  RELATIONAL_POST | PROVENANCE |
  ↪  TEMPORAL | CAPABILITY | ANY",
  "event_trigger": {
    "step_index": "*|int|range",
    "substep_index": "*|int|range",
    "role_name": "AgentName_or_*",
    "content_regex": "regex_or_*",
    "tool_name": "ToolName_or_*"
  },
  "check_hint": "deterministic procedure
  ↪  description in 2-8 sentences",
  "examples": {
    "pass_scenario": "short",
    "fail_scenario": "short"
  },
  "check_type": "python_check|nl_check",
  "python_check": {
    "function_name":
    ↪  "same_as_assertion_name",
    "args": ["trajectory",
    ↪  "current_step_index"],
    "code_lines": [
      "def same_as_assertion_name(traject
      ↪  ory, current_step_index):",
      "    \"\"\"Return True iff invariant
      ↪  holds.\"\"\"",
      "    # parse
      ↪  trajectory[current_step_index],
      ↪  look back if needed",
      "    # MUST include at least one
      ↪  explicit failure path: return
      ↪  False",
      "    return True"
    ]
  },
  "nl_check": {
    "judge_system_prompt_template":
    ↪  "{NL_CHECK_JUDGE_SYSTEM_PROMPT}",
    "judge_user_prompt_template":
    ↪  "template using {POLICY_TEXT}
    ↪  {CURRENT_EVENT_JSON}
    ↪  {WINDOW_EVENTS_JSON}",
    "judge_scope_notes": "what events are
    ↪  in scope and what counts as
    ↪  evidence",
    "focus_steps_instruction": "REQUIRED:
    ↪  Identify 2-4 specific events by
    ↪  relative position and what to
    ↪  check in each.",
    "judge_rubric": ["objective criterion
    ↪  1", "objective criterion 2",
    ↪  "..."],
    "rubric_evaluation_algorithm_template
    ↪  ":
    ↪  "{RUBRIC_EVALUATION_ALGORITHM}",
    "output_format_template":
    ↪  "{OUTPUT_FORMAT}"
  }
}
```

F.2. Semantic Check generated by AGENTRX

```
{
"assertion_name": "explicit_user_confirma
↪  tion_before_write_actions",
"taxonomy_targets": [
  "Instruction/PlanAdherenceFailure",
  "IntentPlanMisalignment"
],
"invariant_type": "TEMPORAL",
"event_trigger": {
  "role_name": "assistant",
  "content_regex": "*",
  "tool_name": "cancel_pending_order|exch
  ↪  ange_delivered_order_items|return_d
  ↪  elivered_order_items|modify_pending
  ↪  _order_.*|modify_user_address"
},
"check_hint": "Before any write-action
↪  tool invocation (cancel, exchange,
↪  return, modify order/address), verify
↪  that the assistant previously
↪  described the intended action and
↪  target ID, and that the user
↪  explicitly confirmed proceeding (e.g.,
↪  'yes', 'confirm', 'please proceed').",
"check_type": "nl_check",
"python_check": {},
"nl_check": {
```

```
"judge_system_prompt_template": "You are a strict compliance judge. Evaluate only with evidence in the provided events and the policy text.\nDo not infer intent beyond explicit statements.\nCRITICAL: If required evidence is missing or ambiguous, mark the criterion as UNCLEAR.\nWhen criteria are marked as UNCLEAR, they are discarded and do NOT contribute to a fail verdict.\nIf all criteria are UNCLEAR or CLEAR_PASS after discarding UNCLEAR ones, the final verdict is PASS.\nOnly mark as CLEAR_FAIL when there is definitive, unambiguous evidence of violation.\nAmbiguity or missing evidence leads to PASS, not FAIL.\n",
"judge_user_prompt_template": "POLICY TEXT:\n{POLICY_TEXT}\n\nTASK INSTRUCTION:\n{TASK_INSTRUCTION}\n\nCURRENT EVENT:\n{CURRENT_EVENT_JSON}\n\nCONTEXT WINDOW EVENTS:\n{WINDOW_EVENTS_JSON}\n\nEvaluate whether, before the current write-action tool call (cancel, modify, exchange, return, address update), the assistant clearly described the action and entity ID, the user explicitly confirmed proceeding with that action, and the IDs match between description and the tool call.",
"judge_scope_notes": "Judge only within the window of events provided. Determine whether this specific write-action tool call is properly preceded by a clear assistant explanation and explicit user confirmation for the same action and same order_id/user_id.",
"focus_steps_instruction": "Focus on: (1) The current assistant tool-call step to identify which write-action tool is being invoked and what identifier (order_id or user_id) is present in its arguments. (2) The immediately prior user message to check for explicit affirmation language indicating consent to proceed with the described action. (3) Assistant messages in the 2-3 steps before the current tool call to see whether the assistant described the intended action (cancel/modify/exchange/return/address update), reminded the user to confirm all relevant items (for exchange/modify-items), and mentioned the same identifier as in the tool call.",
"judge_rubric": [
  "There exists an assistant message earlier in the context that explicitly describes the specific write action type (cancel, modify, exchange, return, or address update) and includes the same identifier (order_id or user_id) that appears in the current tool call arguments.",
  "There exists a user message after the assistant's action description and before the current tool call that contains explicit confirmation language agreeing to proceed with that specific action (e.g., 'yes', 'confirm', 'please proceed', 'go ahead').",
  "When the action is exchange or modify-items, the assistant reminded the customer to confirm they have provided all items to be changed; and the current tool call scope is consistent with the items described and confirmed."
],
"rubric_evaluation_algorithm_template": "Rubric Evaluation Algorithm:\nStep 1: For each criterion in the rubric, evaluate whether it can be CLEARLY judged as PASS or FAIL based solely on the provided events.\n- Mark as CLEAR_PASS if the criterion is demonstrably satisfied by the evidence\n- Mark as CLEAR_FAIL if the criterion is demonstrably violated by the evidence  \n- Mark as UNCLEAR if: insufficient events to judge, criterion is ambiguous in this context, or pass/fail cannot be decisively determined\n\nStep 2: Discard ALL criteria marked as UNCLEAR from consideration.\n\nStep 3: Determine final verdict:\n- If ANY remaining criterion is CLEAR_FAIL == return verdict 'fail'\n- If ALL remaining criteria are CLEAR_PASS (or no criteria remain after discarding UNCLEAR) == return verdict 'pass'\n\nImportant: Only fail when you have CLEAR evidence of failure. When in doubt, mark as UNCLEAR and discard.\n",
```

```
"output_format_template": "Output
↪  Format:\nYou MUST return a JSON
↪  response with this EXACT
↪  structure:\n{\n  \"verdict\": \"pass\"
↪  or \"fail\",\n  \"rubric_results\":
↪  [\n    {\n      \"criterion_index\":
↪  0,\n      \"criterion\": \"full text
↪  of the criterion from the rubric\",\n
↪  \"evaluation\": \"CLEAR_PASS\" or
↪  \"CLEAR_FAIL\" or \"UNCLEAR\",\n
↪  \"reasoning\": \"detailed explanation
↪  of why this evaluation was chosen,
↪  citing specific evidence from
↪  events\"\n    }\n    // ... one entry
↪  per rubric criterion\n  ],\n
↪  \"final_reasoning\": \"explanation of
↪  how the verdict was determined from
↪  rubric_results, stating which criteria
↪  were discarded as UNCLEAR and which
↪  criteria drove the final
↪  decision\"\n}\n"
}
}
```

G. AGENTRX Violation Examples

AGENTRX generates the following programmatic violation because the LLM-extracted "number of t-shirts" from the tool output does not match the true t-shirt count, so the check flags a mismatch.

```
{
    "assertion_name": "tshirt_available_o␣
    ↪  ptions_match_variants_count",
    "taxonomy_targets": [
        "MisinterpretationOfToolOutput",
    ],
    "invariant_type": "RELATIONAL_POST",
    "event_trigger": {
        "step_index": 7,
        "role_name": "assistant",
        "content_regex":
        ↪  "T-?shirt|t-?shirt|T-Shirt",
        "tool_name": "*"
    },
    "check_hint": "When the assistant now
    ↪  reports how many T-shirt options
    ↪  are available, it should compute
    ↪  the count from the variants field
    ↪  of the latest get_product_details
    ↪  result for the T-Shirt product.
    ↪  Specifically, count how many
    ↪  variant entries have available ==
    ↪  true and verify that this equals
    ↪  the numeric count stated by the
    ↪  assistant for 'available' T-shirt
    ↪  options. If the assistant gives an
    ↪  'available options' count that
    ↪  does not match this filtered
    ↪  count, or gives any numeric
    ↪  options count without a
    ↪  corresponding get_product_details
    ↪  result, the invariant fails.",
    "check_type": "python_check",
    "python_check": {
    "function_name": "tshirt_available_op␣
    ↪  tions_match_variants_count",
    "args": [
    "trajectory",
    "current_step_index"
    ],
    "code_lines": [
"import re",
"import json",
"",
"def tshirt_available_options_match_varia␣
↪  nts_count(trajectory,
↪  current_step_index):",
"    '''Verify that the assistant's stated
↪  count of available T-shirt options
↪  matches the tool response.''''",
"    print('Function: tshirt_available_op␣
↪  tions_match_variants_count')",
"    ",
"    # Access the current step from the
↪  trajectory IR format",
"    steps = trajectory.get('steps', [])",
"    if current_step_index >=
↪  len(steps):",
"        raise
↪  IndexError(f'current_step_index
↪  {current_step_index} out of bounds for
↪  {len(steps)} steps')",
"    ",
"    current_step =
↪  steps[current_step_index]",
"    substeps =
↪  current_step.get('substeps', [])",
"    ",
"    # Find assistant substep content",
"    assistant_content = None",
"    for ss in substeps:",
"        if ss.get('role') ==
↪  'assistant':",
"            assistant_content =
↪  ss.get('content')",
"            break",
"    ",
"    if assistant_content is None:",
"        raise KeyError('Assistant content
↪  at current step is missing.')",
"    print(f'Assistant content at step
↪  {current_step_index}:
↪  {assistant_content}')",
"    ",
"    # Try to extract a number near the
↪  word 'available'",
"    match =
↪  re.search(r'(\\d+)\\s+(?:available)',
↪  assistant_content,
↪  flags=re.IGNORECASE)",
"    if match:",
"        stated_count =
↪  int(match.group(1))",
"    else:",
"        # Fallback: extract the first
↪  integer in the content",
"        any_num = re.search(r'(\\d+)',
↪  assistant_content)",
```

```
"        if not any_num:",
"            raise ValueError('No numeric count found in assistant content to verify.')",
"        stated_count = int(any_num.group(1))",
"    ",
"    print(f'Extracted stated_count: {stated_count}')",
"    ",
"    # Find the most recent get_product_details tool response prior to this step",
"    tool_response_content = None",
"    for idx in range(current_step_index - 1, -1, -1):",
"        step = steps[idx]",
"        for ss in step.get('substeps', []):",
"            if ss.get('role') == 'tool':",
"                # Check if previous assistant step called get_product_details",
"                # Look at the assistant step before this tool response",
"                if idx > 0:",
"                    prev_step = steps[idx - 1]",
"                    for prev_ss in prev_step.get('substeps', []):",
"                        if prev_ss.get('role') == 'assistant':",
"                            try:",
"                                calls = json.loads(prev_ss.get('content', ''))",
"                                if isinstance(calls, list):",
"                                    for call in calls:",
"                                        if call.get('function', {}).get('name') == 'get_product_details':",
"                                            tool_response_content = ss.get('content')",
"                                            print(f'Found get_product_details tool response at step {idx}')",
"                                            break",
"                            except json.JSONDecodeError:",
"                                pass",
"                if tool_response_content:",
"                    break",
"        if tool_response_content:",
"            break",
"    ",
"    if tool_response_content is None:",
"        raise KeyError('No prior get_product_details tool response found for verification.')",
"    ",
"    print(f'Raw tool response content: {tool_response_content}')",
"    ",
"    # Parse JSON content",
"    try:",
"        tool_response = json.loads(tool_response_content)",
"    except Exception as e:",
"        raise ValueError(f'Tool response content is not valid JSON: {e}')",
"    ",
"    variants = tool_response.get('variants')",
"    if variants is None:",
"        raise KeyError('Missing variants in get_product_details tool response.')",
"    if not isinstance(variants, dict):",
"        raise TypeError('variants should be a dict.')",
"    ",
"    # Compute available count",
"    available_count = 0",
"    for k, v in variants.items():",
"        if not isinstance(v, dict):",
"            print(f'Skipping variant {k}: not a dict')",
"            continue",
"        avail_flag = v.get('available', None)",
"        print(f'Variant {k} available flag: {avail_flag}')",
"        if avail_flag is True:",
"            available_count += 1",
"    ",
"    print(f'Computed available_count from tool response: {available_count}')",
"    ",
"    result = (stated_count == available_count)",
"    print(f'Assertion result (stated == computed): {result}')",
"    return result"
    ],
"nl_check": {}
}
```

H. Judge Prompts

H.1. Baseline Judge Prompt

You are an Expert Failure-Categorization Judge. You will be provided with a trajectory of an agent's interaction with a user. **Given:** a full trajectory of an agent's conversation with the user (step-indexed)
YOUR TASK: is determine why the agent failed, which failure category applies from the taxonomy below and exactly which step index the failure occurred at. The failure taxonomy has the following categories:

1. **Instruction/Plan Adherence Failure** The agent fails to follow the directions or the agreed plan by ignoring directives and skipping policy steps. This covers both under-execution (missed steps) and over-execution (unplanned or unnecessary actions, e.g., extra tool calls) that deviate from the static plan, domain policy or orchestrator plan.
2. **Invention of New Information:** The agent introduces, removes, or alters information that is not grounded in any available input, context, or tool output. This includes fabricating unsupported facts, hallucinating details, or omitting relevant information without justification.
3. **Invalid Invocation:** The agent encounters errors triggered by inputs that cannot be parsed or validated e.g., Kusto syntax errors or tool calls with bad/missing arguments. Not involving wrong logic; just invalid inputs.
4. **Misinterpretation of Tool Output / Handoff Failure:** The agent incorrectly reasons about its own or another agent's tool output (like computation errors), leading to incorrect assumptions or actions. This also includes cases where the agent considered only partial tool output.
5. **Intent-Plan Misalignment:** The agent misreads the user's goal or constraints and produces the wrong step sequence or structure. Covers both bad ordering/structure and plans aimed at the wrong objective.
6. **Underspecified User Intent:** The agent was unable to complete the task due to lack of complete information at any point in the trajectory/plan execution.
7. **Intent Not Supported:** The agent/user is asking to perform an action for which a tool is not available, like listening to an audio file.
8. **Guardrails Triggered:** The agent is blocked by safety policies or by external site access restrictions, preventing execution despite a valid plan. Examples include policy refusals (e.g., unsafe content, privacy-protected data), CAPTCHA/robot blocks, login/paywall/403/robots.txt denials, or site forbids automation. This is not an agent planning/execution error; it is an external/guardrail block.
9. **System Failure:** The agent faces a system connectivity issue while calling a particular tool like an endpoint not being reachable
10. **Inconclusive (USE SPARINGLY)**: If you are not able to classify the failure into any of the above categories, label it as inconclusive and create your own category.

How to Judge (Decision Procedure):

1. **Step 1 - Locate the first failure:** Scan the trajectory step-by-step from the start. The first step where the agent deviates from the intended plan or emits an error is the first failure. Record the step index and a short failure note.
2. **Step 2 - Check if that failure was resolved**: Look ahead in the trajectory for evidence that the error was resolved. If yes → Resolved; if no such evidence → Not resolved.
3. **Step 3 - Decide and continue:**
   - If Resolved: continue scanning from the next step to find the next new failure, then repeat Step 2 for it.
   - If Not: treat this step as the root-cause failure for the run and assign the taxonomy at this step.

Output format (JSON):

```
{
  "reason_for_failure": "string",
  "failure_case": "int 1-10",
  "reason_for_index": "string",
  "index": "int"
}
```

H.2. Checklist Baseline Judge Prompt

You are an Expert Failure-Categorization Judge. You will be provided with a trajectory of an agent's interaction with a user. **Given:** a full trajectory of an agent's conversation with the user (step-indexed)
YOUR TASK: is determine why the agent failed, which failure category applies from the taxonomy below and exactly which step index the failure occurred at. The failure taxonomy has the following categories: YOUR TASK is determine why the agent failed, which failure category applies from the taxonomy below. and exactly which step index the failure occurred at. The failure taxonomy has the following categories:

1. **Instruction/Plan Adherence Failure**: Goal is correct, but the agent deviates from the required plan by ignoring directives and skipping steps despite having enough information. This covers both

under-execution (missed steps) and over-execution (unplanned or unnecessary actions, e.g., extra tool calls) that deviate from the static plan, domain policy or orchestrator plan. Checklist:

- Can you state the user's goal, and do the agent's intent and end goal match that goal (i.e., the agent is not solving the wrong problem)?

- Was all the required information already available at this step (user intent, required context, prior tool outputs)?

- Is there a step where the ground-truth/policy requires an action (tool call, question, confirmation, ordering) and the agent did something different (skipped it / reordered it / added extra unneeded action)?

2. **Invention of New Information**: The agent introduces, removes, or alters information that is not grounded in any available input, context, or tool output. This includes fabricating unsupported facts, hallucinating details, or omitting relevant information. Checklist:

- Can you pinpoint the exact invented/altered/omitted claim, value, or assumption the agent used?

- Is that claim absent from all evidence available up to that step (user text, provided context, tool outputs)?

- Did the agent rely on that claim to decide an action or produce the failing conclusion (not just harmless wording)?

3. **Invalid Invocation**: Tool call fails because the request is ill-formed (missing args, wrong fields/types, malformed query, schema mismatch). Checklist:

- At the failure step, did the agent attempt a tool call with a concrete invocation payload/arguments?

- Does the tool/runtime explicitly report a parse/validation/schema/syntax error for that call (e.g., missing field, invalid type, cannot parse, malformed query)?

- Is the error NOT a network/timeout/service-unavailable/endpoint-unreachable issue (infra/connectivity)?

- Is the error NOT primarily a CAPTCHA/login/paywall refusal (access/guardrail block)?

4. **Misinterpretation of Tool Output / Handoff Failure**: The agent incorrectly reasons about its own or another agent's tool output, leading to incorrect assumptions or actions. This also includes cases where the agent considered only partial tool output. Checklist:

- Before (or at) the failure step, did the agent receive tool output or handoff output that is relevant to the failing decision?

- Did the agent state or imply a specific reasoning derived from that tool output?

- Does that reasoning contradict the tool output, omit a crucial part, or reflect a clear computation/logic error relative to the output?

5. **Intent-Plan Misalignment**: Agent misunderstands the user's intent/constraints and pursues the wrong objective or violates key constraints due to misunderstanding. Checklist:

- Do the agent's actions/plan optimize for a different goal OR violate a key constraint (not a minor wording/format issue)?

- Is the misalignment due to misunderstanding of intent/constraints (rather than missing required info from the user/context/tool outputs)?

- Is the misalignment not primarily caused by a tool error (invalid invocation, infra failure, or access/guardrail block)?

6. **Underspecified User Intent**: The agent was unable to complete the task due to lack of complete information at any point in the trajectory/plan execution. Checklist:

- Can you identify a specific missing piece of information that is required to proceed correctly (e.g., date, address, account id, item variant)?

- Is that information absent from all evidence available up to that step (user text, provided context, and tool outputs)?

- Did the agent fail because it proceeded without obtaining this information OR because it did not ask for it when needed?

7. **Intent Not Supported**: Requested action cannot be performed with available tools/capabilities. Checklist:

- Is the user requesting an action that requires an external capability/tool (e.g., listen to audio, access a private system, perform a human action)?

- Given the tool set available in this environment, is there no tool that can perform the requested action?

- Is the failure not primarily caused by infrastructure/connectivity issues?

8. **Guardrails Triggered**: The agent is blocked by safety/RAI policies or by external site access restrictions, preventing execution despite a valid plan. Checklist: - Is there an explicit refusal/block signal (policy refusal, CAPTCHA, login required, 403, paywall, robots.txt, automation forbidden)?

   - Would the plan be feasible and correct if this block were removed (i.e., the agent is not pursuing the wrong goal/constraints)?

   - Is the failure not primarily due to malformed tool invocation (schema/syntax/args validation error)?

   - Is the failure not primarily due to infrastructure/-connectivity issues (timeouts, endpoint unreachable)?

9. **System Failure**: The agent faces a system connectivity issue while calling a particular tool like an endpoint not being reachable. Checklist:

   - At the failure step, did the agent attempt a tool call or rely on a tool that should have been callable?

   - Is there an explicit infra/connectivity error signal (timeout, connection refused, DNS failure, endpoint unreachable, service unavailable, premature termination)?

   - Is the failure not primarily a parse/validation/schema/syntax error caused by malformed arguments?

10. **Inconclusive (USE SPARINGLY)**: None of 1-10 clearly apply; must provide a custom category label. Checklist:

    - If labeling as 10, did you provide a non-empty custom category describing the failure type?

How to Judge (Decision Procedure):

1. **Step 1 - Locate the first failure:** Scan the trajectory step-by-step from the start. The first step where the agent deviates from the intended plan or emits an error is the first failure. Record the step index and a short failure note.

2. **Step 2 - Check if that failure was resolved**: Look ahead in the trajectory for evidence that the error was resolved. If yes → Resolved; if no such evidence → Not resolved.

3. **Step 3 - Decide and continue:**

   - If Resolved: continue scanning from the next step to find the next new failure, then repeat Step 2 for it.
   - If Not: treat this step as the root-cause failure for the run and assign the taxonomy at this step.

Output format (JSON):

```
{
  "taxonomy_checklist_reasoning":
  ↪  "string",
  "reason_for_failure": "string",
  "failure_case": "int 1-10",
  "reason_for_index": "string",
  "index": "int"
}
```

H.3. AGENTRX Judge Prompt

You are an Expert Failure-Categorization Judge. You will be provided with a trajectory of an agent's interaction with a user. **Given:** a full trajectory of an agent's conversation with the user (step-indexed)
YOUR TASK: is determine why the agent failed, which failure category applies from the taxonomy below and exactly which step index the failure occurred at. The failure taxonomy has the following categories:

1. **Instruction/Plan Adherence Failure** The agent fails to follow the directions or the agreed plan by ignoring directives and skipping policy steps. This covers both under-execution (missed steps) and over-execution (unplanned or unnecessary actions, e.g., extra tool calls) that deviate from the static plan, domain policy or orchestrator plan.

2. **Invention of New Information:** The agent introduces, removes, or alters information that is not grounded in any available input, context, or tool output. This includes fabricating unsupported facts, hallucinating details, or omitting relevant information without justification.

3. **Invalid Invocation:** The agent encounters errors triggered by inputs that cannot be parsed or validated e.g., Kusto syntax errors or tool calls with bad/missing arguments. Not involving wrong logic; just invalid inputs.

4. **Misinterpretation of Tool Output / Handoff Failure:** The agent incorrectly reasons about its own or another agent's tool output (like computation errors), leading to incorrect assumptions or

actions. This also includes cases where the agent considered only partial tool output.

5. **Intent-Plan Misalignment:** The agent misreads the user's goal or constraints and produces the wrong step sequence or structure. Covers both bad ordering/structure and plans aimed at the wrong objective.

6. **Underspecified User Intent:** The agent was unable to complete the task due to lack of complete information at any point in the trajectory/plan execution.

7. **Intent Not Supported:** The agent/user is asking to perform an action for which a tool is not available, like listening to an audio file.

8. **Guardrails Triggered:** The agent is blocked by safety policies or by external site access restrictions, preventing execution despite a valid plan. Examples include policy refusals (e.g., unsafe content, privacy-protected data), CAPTCHA/robot blocks, login/paywall/403/robots.txt denials, or site forbids automation. This is not an agent planning/execution error; it is an external/guardrail block.

9. **System Failure:** The agent faces a system connectivity issue while calling a particular tool like an endpoint not being reachable

10. **Inconclusive (USE SPARINGLY)**: If you are not able to classify the failure into any of the above categories, label it as inconclusive and create your own category.

You are also provided a list of violations that have been generated through the trajectory through various constraints. Use these to help you identify the root cause category, failure step and agent. Static constraints have been generated through the domain policy and system prompt. Each static constraint is associated with a tool call to ensure it adheres to the domain policy. Dynamic constraints have been generated to cover computation checks, data accuracy, argument validity, and tool output consistency. Each constraints returns a boolean, and if it returns false, it indicates a violation. Note that some violations may be false positives and not all violations may be relevant to the root cause failure. Here are the list of violations:

Executable check violation with grounded evidence

```
================================
VIOLATION #1
================================

Step Index: 2
Assertion Name: kusto_invocation_r
↪  equires_predefined_query_and_c
↪  orrect_cluster
Constraint Type: CAPABILITY
Check Type: python_check
Severity: medium

Check Hint:
----------------------------
When KustoAgent runs a query, it
↪  must be a predefined query
↪  present in the plan or prior
↪  Orchestrator instruction, and
↪  the query must be tailored to
↪  the incident's cluster (no
↪  placeholders like
↪  TODO/TBD/<CLUSTER>). Verify
↪  that a kusto code block exists
↪  earlier and that the current
↪  query's clusterName matches the
↪  cluster parsed from the
↪  incident description.
----------------------------

Evidence:
-----------------------------
Current Event:
  Role: KustoAgent
  Content:
    **Kusto Query:**
    let driftedSettingName =
    ↪  'VncEndpointCandidates';
    ...

     semantic_query_matcher: True

Matched Substeps:
  Sub-index: 5
  Role: KustoAgent
----------------------------

Taxonomy Targets:
  - InvalidInvocation
  - Instruction/PlanAdherenceFailu
  ↪  re
  - IntentPlanMisalignment
```

How to Judge (Decision Procedure):

1. **Step 1 - Locate the first failure:** Scan the trajectory step-by-step from the start. The first step where the agent deviates from the intended plan or emits an error is the first failure. Record the step index and a short failure note.

2. **Step 2 - Check if that failure was resolved**: Look ahead in the trajectory for evidence that the error was resolved. If yes → Resolved; if no such evidence → Not resolved.

3. **Step 3 - Decide and continue:**
   - If Resolved: continue scanning from the next step to find the next new failure, then repeat Step 2 for it.
   - If Not: treat this step as the root-cause failure for the run and assign the taxonomy at this step.

Output format (JSON):

```
{
  "reason_for_failure": "string",
  "failure_case": "int 1-10",
  "reason_for_index": "string",
  "index": "int"
}
```

I. Open Source Licensing

τ-bench, Magentic-One, and Who&When are released under the MIT License; MAST under CC-BY 4.0. Our benchmark is released under MIT License.

Copyright (c) 2026 [Copyright Holder]

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.