PROBEAGENT: PRINCIPLED DIAGNOSIS OF AGENT FAILURES IN VERIFIABLE ENVIRONMENTS

Anonymous authors

Paper under double-blind review

ABSTRACT

Improving agents requires discovering where they fail and understanding why. An agent acts over multiple steps in a changing environment, so diagnosing its failures requires more than inspecting the final response of a large language model (LLM). The same unsuccessful outcome can arise from task difficulty or from an error the agent repeats across tasks. We introduce PROBEAGENT, an automated framework that discovers agent failures, tests their proposed causes, and identifies shared failure modes. Starting from benchmark tasks, PROBEAGENT builds executable tests and validates them using reference solutions, automated checks, and review of the task requirements. For each failed trajectory, it explains the error and constructs new tasks that preserve the suspected cause while changing the surrounding context. Running the agent on these tasks tests whether the same error recurs. PROBEAGENT then groups failures with similar causes and generates descriptions that distinguish their errors. This process connects explanations of individual failures to executable tests and descriptions of shared failure modes. We evaluate PROBEAGENT across 22 agent configurations using tasks from 10 benchmarks, collecting 1,244 failures and producing 536 failure-mode descriptions. These descriptions distinguish specific errors within broad outcomes such as incomplete work, including planning without acting and stopping before delivering a file. In human evaluation, annotators accept 90.0% of assigned descriptions and correctly reject 83.3% of similar alternatives, supporting the accuracy and specificity of the discovered failure modes.

1 INTRODUCTION

Agents are increasingly used to modify code, navigate websites, and complete work in terminal environments (Jimenez et al., 2024; Zhou et al., 2024; Merrill et al., 2026). An agent combines a large language model (LLM) with a *harness*, which manages its context, execution flow, and termination. As agents become part of real-world workflows, they still fail on many tasks. Studying these failures can guide improvement: we need to identify where agents fail and understand why. Benchmark scores summarize task success, but the final outcome alone does not explain the cause of a failure. The same outcome can arise for different reasons. For example, a missing report may result from a difficult computation, or simply because the agent stops before saving its result. These failures require different fixes. Improving agents therefore requires not only measuring failures, but also systematically discovering them and diagnosing their causes.

Existing automated evaluation methods often discover LLM failures by generating new questions or selecting informative tests (Cheng et al., 2024; Li et al., 2025; Afkanpour et al., 2025). ProbeLLM (Huang et al., 2026) goes beyond finding individual errors by explaining incorrect responses and organizing related errors into *failure modes*. Each mode describes a shared error pattern supported by concrete failure examples. This connects the discovery of individual failures with an account of the patterns shared across them.

For agents, existing methods diagnose failed trajectories after execution. For example, Who&When benchmarks failure attribution (Zhang et al., 2025), while AgentRx and AgentDebug analyze *trajectories*, the sequences of actions and observations, to locate errors or guide corrective feedback (Barke et al., 2026; Zhu et al., 2025). These studies help explain observed failures, but leave open how to systematically construct new tasks to discover further failures and organize them into shared modes.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

We therefore ask: *can we build an effective framework that automatically discovers and diagnoses agent failures?*

This requires addressing three challenges. **(1) Building reliable executable tests.** An LLM test can pair a question with a reference answer and a verifier. An agent test also needs a working environment in which actions change the task state, a solvable task, and a checker that evaluates the required outcome. Environment defects or shortcuts accepted by the checker can otherwise give misleading results (Aleithan et al., 2024; Wang et al., 2026). **(2) Distinguishing task-specific difficulties from recurring agent behavior.** A failed trajectory may reflect a difficulty in one task or a behavior that appears in many tasks. Explaining that trajectory alone does not distinguish the two. New tests must change the task context while preserving the condition thought to cause the failure, so that we can check whether the same agent makes the same mistake again. **(3) Identifying shared failures across different tasks.** Omitting a CSV export after data analysis and a JSON export after a database query can reflect the same error: completing the computation without delivering its output. A failed write can also leave a file missing, but requires a different correction. Diagnosis must distinguish the behaviors behind these similar outcomes.

To address these challenges, we propose PROBEAGENT, an automated framework for discovering and diagnosing agent failures. It connects three stages. *Establish failures* constructs executable probes from benchmark tasks, validates their checkers and reference solutions, and explains failed trajectories. *Test failure recurrence* constructs *sibling tasks*: new tasks that preserve the proposed triggering condition while changing the domain, data, or output format. Running the same agent on these tasks tests whether the explained error recurs. *Discover failure modes* groups failures by their causes and generates descriptions that distinguish their errors. By using explanations to guide new executable tests, PROBEAGENT connects diagnosis of observed failures with discovery of further failures and the patterns they share.

We evaluate PROBEAGENT across 22 agent configurations and 10 benchmarks, collecting 1,244 explained failures and producing 536 failure-mode descriptions. These distinguish specific mistakes within incomplete work, such as planning without acting and stopping before delivering a file. Human annotators accept 90.0% of assigned descriptions and correctly reject 83.3% of similar alternatives. Review of generated tasks detects invalid tests that pass execution checks, and enforcing similarity within each group improves description agreement from 78.0% to 91.8% against the model-based reference. These results demonstrate that PROBEAGENT produces usable diagnostic tests and accurate, specific failure descriptions.

Contributions. Our contributions are:

1. **We propose PROBEAGENT, an automated framework for discovering and diagnosing agent failures.** It connects executable testing with failure-mode discovery (Section 3).
2. **PROBEAGENT builds and validates probes to test failure explanations.** It turns explanations into new executable tests and groups failures by their causes to generate descriptions supported by examples (Sections 3.1–3.3).
3. **We evaluate PROBEAGENT across agents and benchmarks.** Our experiments demonstrate diverse failure discovery, accurate descriptions, and effective probe validation (Section 4).

2 PRELIMINARIES

Let A denote an agent. A *rollout* is one execution, producing a trajectory $z = (o_0, a_1, o_1, \ldots, a_T, o_T)$ of actions a_t and observations o_t.

Definition 1 (Probe)**.** A probe is an executable test $s = (E, x, g, \pi^\star)$, comprising an environment E, instruction x, binary task checker g, and reference solution $\pi^\star$. The checker returns $g(z) = 1$ for success and $g(z) = 0$ for failure.

Definition 2 (Failure)**.** A failure is a diagnosed error $r = (s, z, u, e)$ from an execution with $g(z) = 0$ on a validated probe s. It includes the trajectory z, a summary u, and a failure explanation e (Definition 3). One unsuccessful execution may contain several errors, each represented by its own r.

Definition 3 (Failure explanation)**.** A failure explanation $e = (b, \ell, \tau, \delta)$ identifies a proposed cause of one failure: the erroneous action or omitted action b, the point in the task where it occurs ℓ, the task condition τ under which it arises (the *trigger*), and a corrective action δ. Each field is supported by the trajectory z.

Definition 4 (Failure-mode description)**.** A failure-mode description c is a natural-language statement of a shared error. Write $c(r) \in \{0, 1\}$ for the judgment of whether failure r exhibits that error. A *failure mode* is a tuple (c, R, B) with nonempty supporting failures R and disjoint contrasting failures B, for which the description satisfies

$$c(r) = 1 \quad (r \in R), \qquad c(r) = 0 \quad (r \in B).$$

A failure explanation e concerns one trajectory; a description c states the pattern used to group failures. Section 3.3 specifies how these judgments are made.

Example. An agent computes a total but stops without saving the required report. The failure r records this error and its trajectory. Its explanation e identifies the omitted save after computation, the requirement to deliver a file, and the correction of writing that file. A description c, "stops after computing without delivering the required file," can cover this failure and similar errors in database or spreadsheet tasks. A failed write does not match this description.

Definition 5 (Sibling environment)**.** For a failure r from probe s, interpret its trigger $\tau(r)$ as a predicate on probes: $\tau(r)(s') = 1$ means that s' contains the proposed triggering condition. A sibling is a new probe $s' \neq s$ with $\tau(r)(s') = \tau(r)(s) = 1$, constructed by varying domain, data, tools, naming, or output format. Its environment is a *sibling environment*. We validate the new probe before testing the same agent A on it to determine whether the error recurs.

Objective. For agent A, search policy η constructs and tests probes, collecting failures $\mathcal{D}_\eta$ at cost $C(\eta)$. Given budget B_{test} and fixed induction procedure $\mathcal{I}_\theta$, we seek the most distinct, supported failure modes:

$$\max_{\eta} \ \mathbb{E}[|\mathcal{I}_\theta(\mathcal{D}_\eta)|] \quad \text{s.t.} \quad C(\eta) \leq B_{\text{test}}. \tag{1}$$

The expectation accounts for randomness in task construction and agent execution. The budget covers construction, validation, and testing. We count modes supported by validated probes and descriptions satisfying Definition 4, using the same induction settings when comparing search policies.

3 The ProbeAgent Framework

ProbeAgent connects three stages (Figure 1): *Establish failures* builds and validates probes, then explains failed trajectories; *Test failure recurrence* checks whether the explained error appears in new tasks; and *Discover failure modes* groups failures and describes their shared errors. Search alternates between the first two stages, and discovery of failure modes uses the collected failures. Appendix A specifies the LLM configuration for construction, review, explanation, and judging.

3.1 Establish failures

To diagnose an agent, we first need tests whose instructions, environment, and checker agree. We construct these tests from benchmark tasks and validate them before interpreting the agent's errors.

Construct a probe from a seed task. A seed task t is a benchmark task used to start the search. We select tasks with inspectable checkers on which the target agent fails an initial trial. The reference solution succeeds and inaction fails, as specified by Equation (2). The builder receives the task instructions, input files, and checker, and constructs $s \sim \text{Build}(t)$. It changes the setting or operational demands while preserving the question and required outcome. Appendix A gives the construction budget and task-preservation checks.

Validate the probe. We admit a generated probe after three checks. **a) Executed: test reference and null controls.** The reference must succeed and inaction (nop) must fail; g evaluates their executions. Model-judged tasks record their verdict type. **b) Read: inspect checker sources.** We check for inaccessible or external assertions. **c) Reviewed: check instruction–implementation consistency.** A separate read-only agent checks for shortcuts, missing requirements, and solution leakage. It reports evidence but cannot repair candidates; rejected probes return to the builder within budget. This review detects shortcuts that execution controls can miss (Section 4.4). With binary check results λ_{exec}, λ_{read}, and λ_{review}, admission is

$$\begin{aligned} \lambda_{\text{exec}}(s) &= \mathbf{1}\{g(\pi^\star) = 1 \ \wedge \ g(\text{nop}) = 0\}, \\ \Lambda(s) &= \lambda_{\text{exec}}(s)\lambda_{\text{read}}(s)\lambda_{\text{review}}(s). \end{aligned} \tag{2}$$

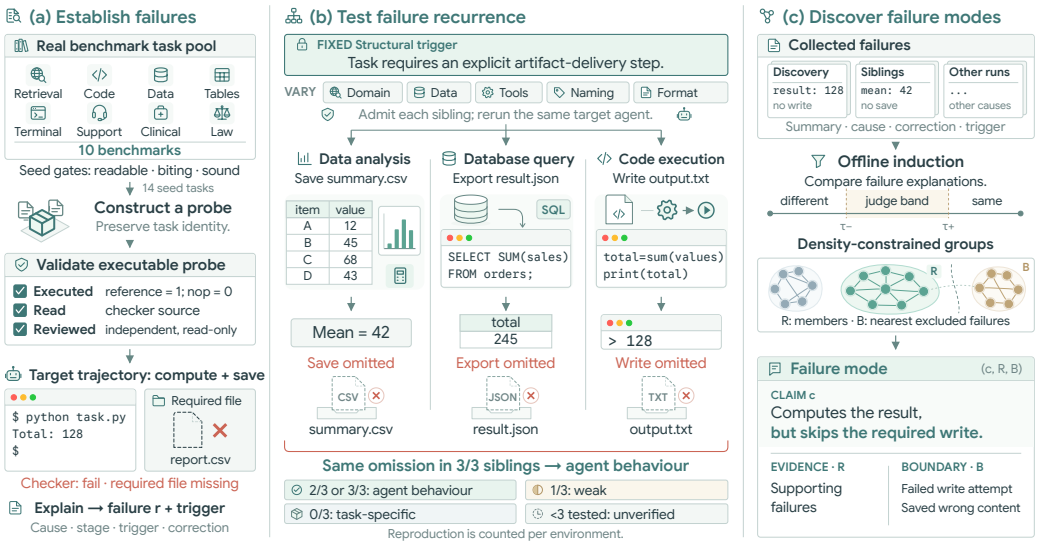


Figure 1: **Overview of PROBEAGENT.** (a) **Establish failures:** construct and validate a probe from a benchmark task, and explain the target's failed trajectory. (b) **Test failure recurrence:** preserve the structural trigger while varying the context; here, each task computes a result but omits delivery. (c) **Discover failure modes:** use cause-based similarity and density-constrained grouping to generate descriptions that cover each group and exclude similar errors outside it.

Only probes with $\Lambda(s) = 1$ proceed to target testing.

Explain the target trajectory. For an admitted probe, let $z \sim \mathrm{Run}(A, s)$ be the target trajectory. If $g(z) = 0$, an LLM analyzes the task, checker feedback, and trajectory to return summary–explanation pairs $(u, e) \in \mathrm{Analyze}(s, z)$, yielding

$$\mathcal{R}(s, z) = \{(s, z, u, e) : (u, e) \in \mathrm{Analyze}(s, z)\}, \qquad g(z) = 0. \tag{3}$$

Each element is a failure r (Definition 2). In Figure 1(a), the explanation identifies an omitted save, proposes writing the report, and supplies the trigger for sibling tests.

3.2 TEST FAILURE RECURRENCE

A failure explanation proposes why one trajectory went wrong. We test that explanation by constructing tasks in which the same error could occur, while changing details that are specific to the original task.

Construct sibling tasks. Given failure r from s, the builder samples $s_j \sim \mathrm{Build}_{\mathrm{sib}}(r, \tau(r))$. We construct and validate each sibling so that $s_j \neq s$, $\tau(r)(s_j) = 1$, and $\Lambda(s_j) = 1$ (Definition 5). For example, CSV analysis can become a JSON database query while retaining the requirement to save the result (Figure 1(b)).

Test the explanation. PROBEAGENT runs the same agent to obtain $z_{j,t} \sim \mathrm{Run}(A, s_j)$ for attempts $t = 1, \ldots, n_j$. An LLM recurrence judge $H(r, z_{j,t}) \in \{0, 1\}$ compares the trajectory with r's failure explanation and corrective action. It returns 1 if the same error recurs under the specified condition, and 0 otherwise. For m tested sibling environments, we count the environments in which the error recurs:

$$y_j = \max_{1 \le t \le n_j} H(r, z_{j,t}), \qquad \rho = \sum_{j=1}^{m} y_j. \tag{4}$$

Thus repeated occurrences in one environment count once. An omitted save matches the example failure; incorrect computation or an unsuccessful write does not. With $m_0 = 3$ and $\theta = 2/3$, the test

outcome is

$$v(r) = \begin{cases} \text{AGENT BEHAVIOR} & m \geq m_0,\ \rho/m \geq \theta, \\ \text{TASK-SPECIFIC} & m \geq m_0,\ \rho = 0, \\ \text{WEAK} & m \geq m_0,\ 0 < \rho/m < \theta, \\ \text{UNVERIFIED} & m < m_0. \end{cases} \tag{5}$$

A mode pools its members' distinct tests and reproduction counts, counting each test once and recording conflicting outcomes among its failures.

Balance new tasks and sibling tests. PROBEAGENT alternates between exploring new tasks and testing sibling tasks. During exploration, PROBEAGENT selects the task category with the fewest admitted probes. Categories are determined by the search goal and seed tasks, and ties are broken at random. A newly observed failure label schedules construction of three sibling tasks. After three consecutive sibling construction attempts, we construct a probe from a seed task to continue exploring for new failures. Appendix A gives the selection rule and budgets.

3.3 DISCOVER FAILURE MODES

Different tasks can expose the same error, while similar unsuccessful outcomes can have different causes. PROBEAGENT groups the collected failures by cause, then generates a description for each group. We call this process *induction*. It uses failures from both seed-task probes and sibling tasks, including new errors found during sibling tests, and outputs modes (c, R, B) (Definition 4; Figure 1(c)).

Cause-based similarity. For each failure $r = (s, z, u, e)$, we form text $\kappa(r)$ by combining its summary u with the cause, corrective action, trigger, and task stage from e, omitting absent fields. We embed this text as $\phi(r) = \mathrm{emb}(\kappa(r))$, subtract the corpus mean $\bar{\phi}$, and normalize:

$$\psi(r) = \frac{\phi(r) - \bar{\phi}}{\|\phi(r) - \bar{\phi}\|_2}, \qquad \mathrm{sim}(r, r') = \langle \psi(r), \psi(r') \rangle. \tag{6}$$

This reduces the shared embedding direction associated with describing agent failures. We use an LLM as a pairwise judge J that compares two failures' explanations, triggers, and corrections: $J(r, r') = 1$ denotes the same mechanism and 0 different mechanisms. It labels a calibration sample spanning the full similarity range. We choose thresholds τ_+, τ_- so that, on this sample, at least $q_+ = 0.90$ of pairs with $\mathrm{sim} \geq \tau_+$ receive $J = 1$, and at least $q_- = 0.98$ of pairs with $\mathrm{sim} \leq \tau_-$ receive $J = 0$. Thus $q_\pm$ specify target agreement rates for accepting or rejecting a pair as sharing the same error. We accept pairs above τ_+ and reject pairs below τ_-. We call the interval between these thresholds the *ambiguous band*. Within this interval, we send pairs among each failure's $k_{\mathrm{nn}} = 15$ nearest neighbors to J and retain those receiving $J = 1$. Other pairs in the interval remain unconnected. Appendix A gives the threshold selection rule.

Density-constrained grouping. Failures form a graph $G = (\mathcal{D}, E)$ whose edges connect pairs accepted by the pair-selection procedure above. Each retained edge has weight $w(r, r') = \mathrm{sim}(r, r')$. Chains of such links can connect otherwise different failures, so PROBEAGENT also checks each group's overall similarity. For a connected group K, let $d_{\mathrm{in}}(K)$ be the mean similarity over all member pairs and $d_{\mathrm{out}}(K)$ the maximum similarity between any member and any outside failure. For connected partitions $\mathcal{P}(G)$ of this graph, our grouping target is

$$\min_{\mathcal{K} \in \mathcal{P}(G)} |\mathcal{K}| \quad \text{s.t.} \quad d_{\mathrm{in}}(K) - d_{\mathrm{out}}(K) \geq \mu \quad \forall K \in \mathcal{K} :\ |K| > 1. \tag{7}$$

We approximate this objective with a greedy procedure. For a group that violates the constraint, we remove edges in ascending weight order until it disconnects, then check each resulting component again. Splitting continues until each group satisfies the constraint or contains one failure. The similarity margin μ controls the required gap between within-group and outside similarity. Unlike $\tau_\pm$, which decide individual pair links, μ checks a whole group. Increasing it can split a broad group into smaller groups with more specific errors.

Generate and check failure-mode descriptions. For each group R, the *boundary* B contains the nearest k_{nn} excluded failures, ranked by their maximum similarity to a failure in R. These are failures that look similar but belong to other groups. An LLM generates a description c, also called a *claim*. We use the induction LLM as a judge to check whether the error and condition stated in c agree with

each failure's explanation and corrective action. We accept c when it covers every failure in R and excludes every failure in B (Definition 4); otherwise the group remains unresolved. For a new failure r, the corresponding matching modes are $\mathcal{M}(r) = \{k : c_k(r) = 1\}$. Thus a failure belongs to a mode when it exhibits the error stated in its description. A failure may match multiple descriptions or remain unassigned. Discovery groups partition the corpus; an execution with several diagnosed errors can contribute to several modes.

We evaluate these descriptions on failures withheld before induction (Section 4.3). Algorithm 1 gives the full procedure; Appendix A specifies the model roles and scoring rules.

4 Experiments

4.1 Experimental setup

Models and harnesses. We evaluate 22 configurations combining 15 models and seven harnesses. The models include GPT-5.6 and GPT-5.4-nano (OpenAI, 2026c;b), Claude (Anthropic, 2026a), DeepSeek-V4-Pro (DeepSeek-AI, 2026), Kimi K3 (Moonshot AI, 2026), GLM-5 and GLM-5.2 (Z.AI, 2026a;b), Qwen3.8-Max (Alibaba Cloud, 2026), Grok 4.5 (SpaceXAI, 2026), and MiniMax-M2.7 (MiniMax, 2026). The harnesses are Terminus 2 (Harbor Team, 2026b), Codex (OpenAI, 2026a), Claude Code (Anthropic, 2026b), Goose (Goose Contributors, 2026), opencode (Anomaly, 2026), OpenHands (Wang et al., 2025), and Aider (Aider Contributors, 2026). Table 4 lists the exact model–harness pairings. Appendix A specifies the internal model configuration for construction, review, analysis, and induction.

Tasks and protocol. We start from 14 seed tasks from 10 benchmarks spanning retrieval, code editing, performance engineering, data pipelines, terminal operation, tabular and spreadsheet work, clinical tasks, customer service, and law (Appendix B). Each diagnostic search constructs and validates new probe environments from these tasks, then collects failures from the agent's trajectories. We group these failures and generate descriptions of their shared errors. For corpus analysis, we use the group-level similarity margin $\mu = 0.02$ defined in Equation (7); ablations examine other values. Probes run as Harbor tasks (Harbor Team, 2026a) in Docker containers (Docker, 2026). Search uses three rollouts per environment. Appendix A.3 gives the hyperparameters and execution details.

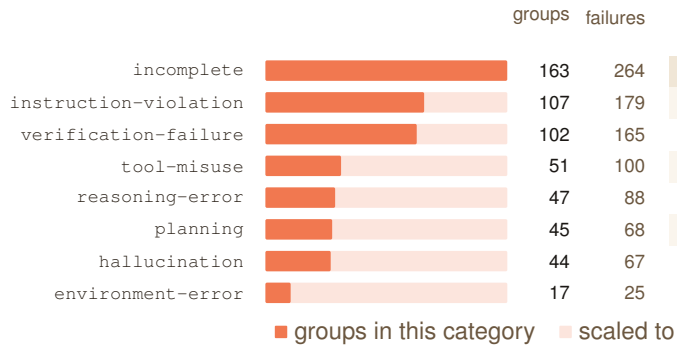
4.2 Failure discovery across agents and tasks

ProbeAgent discovers diverse failure modes across agents and tasks. We run ProbeAgent across 347 generated environments and collect 1,244 explained failures from which we generate 536 failure-mode descriptions. Table 1 provides examples of the resulting descriptions, including treating a human transfer as completion, planning an action without executing it, and continuing research without delivering a file. Each description specifies an error in its task context.

Table 1: Examples of induced claims, abridged for presentation. $|R|$ is the number of supporting failures.

Failure description	$\lvert R\rvert$	Models
The checker rejects the submission when execution leaves a generated artifact or modified file outside the permitted delivery surface.	11	Sonnet 4.5, GPT-5.4-nano
After a successful human transfer, the agent treats the transfer as sufficient termination and omits the required closing actions.	8	GPT-5.6-sol, DeepSeek-V4, Kimi K3
Reaching a usable startup point, the agent remains at *planning* the required call instead of issuing it, and the run ends with nothing created.	5	GPT-5.4-nano, GLM-5, GLM-5.2
When the deliverable is a JSON file, the agent stays in research until the run ends and never creates it.	4	Grok 4.5, Qwen3.8-Max

ProbeAgent distinguishes failures within broad error categories. We compare our groups with a baseline that assigns failures to eight fixed categories, including incomplete work, instruction violations, and verification failures (Figure 2). Both use the same 961 failures. The *incomplete* category contains 264 failures (27.5% of the set), spanning 163 induced groups. The pattern extends to



fixed taxonomy		PROBEAGENT
buckets	8	505
largest, failures	264	11
largest, share	27.5%	1.1%
median, failures	94	1
buckets of one failure	0	381
failures with no bucket	5	0

Figure 2: **Fixed categories and induced descriptions** on 961 failures. Left: failures and induced groups represented in each category. Right: summary statistics for the two representations. The taxonomy provides broad coverage, while induction separates more specific mechanisms.

instruction violation and *verification failure*, which contain 107 and 102 induced groups, respectively. Thus the distinctions found by PROBEAGENT occur across several common error categories, rather than only within one type of unsuccessful outcome.

The largest induced group contains 11 failures, compared with 264 in the largest fixed category (Figure 2, right). A category such as *incomplete* identifies an outcome; our descriptions distinguish the actions or omissions behind it.

4.3 ACCURACY OF FAILURE-MODE INDUCTION

PROBEAGENT accurately characterizes failure modes. We evaluate description accuracy by testing whether each description correctly identifies matching failures. The induction judge assigns held-out failures using the descriptions, and the pairwise judge confirms each assignment against the selected group's first member in corpus order. The classifier may return no match; agreement is measured among assigned failures. With 25% of failures held out before induction, PROBEAGENT achieves 88.5% agreement on the 1,244-failure corpus at $\mu = 0.02$. These failures are excluded from centering, calibration, and description construction. The result supports the accuracy of the induced failure descriptions (Appendix C).

Human evaluation supports description accuracy and specificity. We ask two annotators to evaluate 30 failures, each paired with its assigned group's description and a similar alternative from another group. Each annotator evaluates all 60 pairs. They accept 28 and 26 assigned descriptions, respectively, and each correctly rejects 25 alternatives, giving mean acceptance and rejection rates of 90.0% and 83.3%. These judgments support accurate descriptions that distinguish similar errors (Appendix C.5).

4.4 EFFECTIVENESS OF PROBE VALIDATION

Review of task requirements detects defects that execution checks miss. Across 1,451 build rounds, the execution gate rejects 349 candidates, and the reviewer rejects a further 638 after executable controls pass. The most common finding is a rewarded shortcut that bypasses the intended task, observed in 565 rounds. Neither the reference nor null run attempts these shortcuts. In 323 rounds, the checker does not evaluate the requested outcome; in 255, it imposes an unstated requirement. Findings can overlap within a round. These findings show why both checks contribute to probe validation: execution tests whether the reference succeeds and inaction fails, while semantic review examines whether the checker rewards the work specified in the instruction. Appendix E.1 gives the complete gate and finding counts.

Classification of 940 failures by whether they concern task completion, the harness, or formatting identifies 95.0% as substantive failures, 3.1% as harness-related problems, and 1.9% as format-related problems. Reference solutions pass in all nine audited environments where every tested

agent failed. Appendix E reports these checks together with the manual environment audit, including expected-answer errors and format ambiguities.

4.5 Sibling testing separates recurring and task-specific failures

Sibling tests distinguish recurring behavior from context-specific failures. Figure 3 summarizes 239 proposed failure explanations, of which 119 reached sibling testing within the search budget. Among those tested, 48 (40.3%) reproduced in no sibling, 20 (16.8%) met the agent-behavior criterion, and 26 (21.8%) reproduced weakly. The remaining 25 (21.0%) had fewer than three tested siblings. The tests give each explanation an explicit recurrence outcome under changed task contexts.

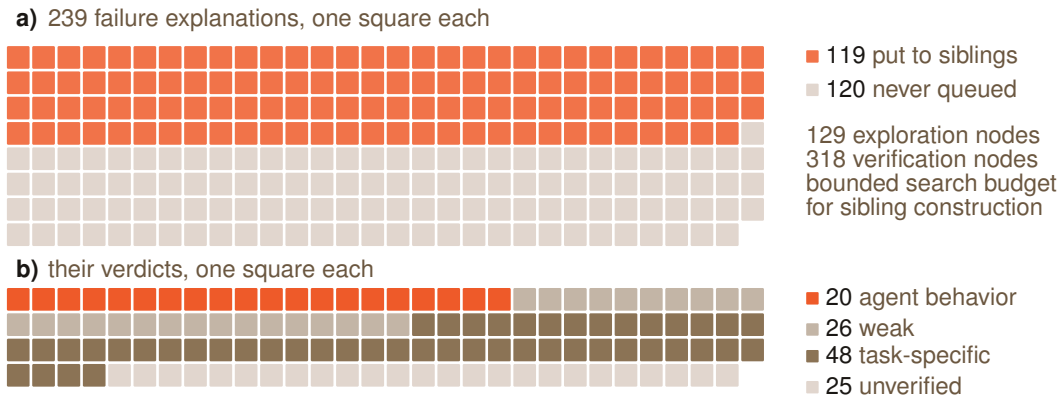


Figure 3: **Failure recurrence across sibling tasks.** Of 239 proposed explanations, 119 reached sibling testing. Verdicts follow Equation (5); reproduction is counted per environment.

The tests distinguish explanations that recur across new tasks from those that do not recur in the tested contexts. This gives each explanation evidence beyond its original trajectory and identifies recurring errors for further investigation. Appendix B.3 links these outcomes to the discovered failure modes.

4.6 Grouping components and efficiency

We test which grouping choices improve description accuracy and reduce LLM calls.

The group-level constraint prevents excessive merging. Table 2 compares induction variants on the same failure set. Removing the constraint reduces 505 groups to 23, including a component containing 650 of 961 failures, and lowers held-out agreement from 91.8% to 78.0%. Pairwise links alone therefore merge failures that the group-level constraint separates.

Table 2: **Induction ablations** on 961 failures with similarity margin $\mu = 0.05$. Shared counts groups spanning agent configurations; judged counts pairs sent to J. Agreement measures held-out assignments confirmed by J.

Variant	Groups	Shared	Judged	Agree.
Full method	505	11	7,449	91.8%
No density constraint	23	4	7,449	78.0%
Fixed thresholds	500	11	6,789	90.8%
Neighbors $k_{\text{nn}} = 5$	506	10	2,614	93.3%
Neighbors $k_{\text{nn}} = 30$	505	11	14,297	92.4%
Single threshold	500	11	0	92.1%

The similarity margin controls how broadly failures are grouped. Reducing μ from 0.05 to 0.02 increases shared groups from 13 to 27 on the 1,244-failure corpus, while held-out agreement changes from 89.3% to 88.5%. We use $\mu = 0.02$ for the main corpus analysis because it identifies

more groups shared across agent configurations with similar agreement. The component ablations in Table 2 use $\mu = 0.05$; Appendix C reports the full margin sweep.

Threshold selection and pair reuse reduce judging costs. Selecting thresholds from judge-labeled pairs increases the fraction of pairs judged to match within the ambiguous band from 9% to 41%, compared with fixed thresholds. Nearest-neighbor selection reduces judged pairs by 75.5% while preserving all 53 groups in a 182-failure experiment. Reusing cached pair judgments answers 7,150 of 8,655 ambiguous-band pairs in the 1,244-failure run. Appendix C reports the similarity analysis, thresholds, and additional cost measurements.

5 Related Work

Agent evaluation. Agent benchmarks cover repository repair, web interaction, desktop operation, and domain-specific tasks (e.g., Jimenez et al., 2024; Zhou et al., 2024; Xie et al., 2024; Yao et al., 2024; Jiang et al., 2025; Liu et al., 2024). Recent work also examines how agents progress through a task. For example, AgentBoard measures intermediate progress, τ^2-Bench studies coordination when users and agents share control, and Gaia2 evaluates dynamic, asynchronous environments (Ma et al., 2024; Barres et al., 2025; Froger et al., 2026). Agent-as-a-Judge assesses intermediate work against task requirements, while AgentRewardBench compares automatic trajectory evaluations with expert assessments (Zhuge et al., 2025; Lù et al., 2025). ProbeAgent uses failed tasks to construct further tests, checking whether the same error recurs under changed task conditions.

Task generation and benchmark validity. Generated environments expand the experience available for agent training. For example, AgentGen synthesizes environments and planning tasks, while SWE-smith and R2E-Gym construct executable software-engineering tasks through repository modifications, test generation, and commit-based synthesis (Hu et al., 2025; Yang et al., 2025; Jain et al., 2025). Reliable verification is central to using such tasks. A different stream focuses on human validation and benchmark audits to identify ambiguous specifications, weak tests, and exploitable evaluation logic (OpenAI, 2024; Aleithan et al., 2024; Wang et al., 2026). ProbeAgent uses task generation for diagnosis: it constructs validated tests around a proposed cause of failure and checks whether the error recurs.

Automated probing and failure analysis. Dynamic evaluation moves beyond fixed test sets by adapting what is tested. For instance, Meta Probing Agents transform evaluation problems along cognitive dimensions (Zhu et al., 2024); AutoDetect coordinates examiners, questioners, and assessors to expose model weaknesses (Cheng et al., 2024); AutoBencher optimizes dataset descriptions for criteria such as difficulty and novelty (Li et al., 2025); ACE selects informative capabilities through active learning (Afkanpour et al., 2025); and FACT-AUDIT adapts fact-checking assessments to model responses and evaluates both verdicts and justifications (Lin et al., 2025). For agents, MAST organizes multi-agent failures into a taxonomy, and Who&When provides annotations of the responsible agent and decisive step (Cemri et al., 2025; Zhang et al., 2025). AgentDebug combines critical-error detection with feedback for later rollouts, while AgentRx uses stepwise constraints to ground failure localization (Zhu et al., 2025; Barke et al., 2026). These methods organize or locate errors in observed trajectories. ProbeLLM (Huang et al., 2026) discovers failure modes of LLM responses through active probing and verifiable test cases. ProbeAgent extends this approach to agents: it constructs and validates environments, uses trajectory explanations to design new tasks, tests whether the same error recurs, and describes shared failure modes.

6 Conclusion

ProbeAgent provides an automated framework for discovering and diagnosing agent failure modes. Across 22 agent configurations and 10 benchmarks, it collects 1,244 failures and produces 536 failure-mode descriptions. Its experiments demonstrate diverse failure discovery, accurate descriptions, and effective probe validation and grouping. By combining executable tests with tests of recurrence, ProbeAgent helps identify specific agent errors and the task conditions in which they arise.

AI USE STATEMENT

Generative models are components of the proposed methodology: they construct probe and sibling environments, review candidate tasks, explain failed trajectories, and judge failure pairs and failure claims, as described in Section 3. Generative AI tools also assisted with manuscript restructuring and language revision, consistency checks, LaTeX formatting, and the design and revision of the overview figure and its description.

REFERENCES

Arash Afkanpour, Omkar Dige, Fatemeh Tavakoli, Negin Baghbanzadeh, Farnaz Kohankhaki, and Elham Dolatabadi. Automated capability evaluation of foundation models. *arXiv preprint arXiv:2505.17228*, 2025. URL https://arxiv.org/abs/2505.17228.

Aider Contributors. Aider: AI pair programming in your terminal, 2026. URL https://github.com/Aider-AI/aider. Accessed September 20, 2026.

Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. SWE-Bench+: Enhanced coding benchmark for LLMs. *arXiv preprint arXiv:2410.06992*, 2024. URL https://arxiv.org/abs/2410.06992. 32.67% of passing patches attributed to solution leakage, 31.08% to weak tests.

Alibaba Cloud. Alibaba unveils Qwen3.8-Max: Its largest and most capable flagship model to date, 2026. URL https://www.alibabacloud.com/en/press-room/alibaba-unveils-qwen3-8-max.

Anomaly. OpenCode, 2026. URL https://opencode.ai. Accessed September 20, 2026.

Anthropic. Anthropic's transparency hub, 2026a. URL https://www.anthropic.com/transparency. Claude model documentation and system cards; accessed September 20, 2026.

Anthropic. Claude Code overview, 2026b. URL https://code.claude.com/docs/en/overview. Accessed September 20, 2026.

Shraddha Barke, Arnav Goyal, Alind Khare, Avaljot Singh, Suman Nath, and Chetan Bansal. AgentRx: Diagnosing AI agent failures from execution trajectories. *arXiv preprint arXiv:2602.02475*, 2026. URL https://arxiv.org/abs/2602.02475. Accepted to Findings of EMNLP 2026.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-Bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025. URL https://arxiv.org/abs/2506.07982.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? *arXiv preprint arXiv:2503.13657*, 2025. URL https://arxiv.org/abs/2503.13657.

Jiale Cheng, Yida Lu, Xiaotao Gu, Pei Ke, Xiao Liu, Yuxiao Dong, Hongning Wang, Jie Tang, and Minlie Huang. AutoDetect: Towards a unified framework for automated weakness detection in large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 6786–6803, 2024. doi: 10.18653/v1/2024.findings-emnlp.397. URL https://aclanthology.org/2024.findings-emnlp.397/.

DeepSeek-AI. DeepSeek-V4 model card, 2026. URL https://fe-static.deepseek.com/chat/transparency/deepseek-V4-model-card-EN.pdf.

Docker. What is Docker?, 2026. URL https://docs.docker.com/get-started/docker-overview/. Docker documentation; accessed September 20, 2026.

Romain Froger, Pierre Andrews, Matteo Bettini, Amar Budhiraja, Ricardo Silveira Cabral, Virginie Do, Emilien Garreau, Jean-Baptiste Gaya, Hugo Laurençon, Maxime Lecanu, Kunal Malkan, Dheeraj Mekala, Pierre Ménard, Gerard Moreno-Torres Bertran, Ulyana Piterbarg, Mikhail Plekhanov, Mathieu Rita, Andrey Rusakov, Vladislav Vorotilov, Mengjue Wang, Ian Yu, Amine Benhalloum, Grégoire Mialon, and Thomas Scialom. Gaia2: Benchmarking LLM agents on dynamic and asynchronous environments. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://arxiv.org/abs/2602.11964.

Goose Contributors. Goose, 2026. URL https://github.com/aaif-goose/goose. Accessed September 20, 2026.

Harbor Team. Harbor: Evaluate agents in sandboxed environments, 2026a. URL https://www.harborframework.com/. Accessed September 20, 2026.

Harbor Team. Terminus-2, 2026b. URL https://github.com/harbor-framework/harbor/blob/main/docs/content/docs/agents/terminus-2.mdx. Agent documentation; accessed September 20, 2026.

Mengkang Hu, Pu Zhao, Can Xu, Qingfeng Sun, Jianguang Lou, Qingwei Lin, Ping Luo, and Saravan Rajmohan. AgentGen: Enhancing planning abilities for large language model based agent via environment and task generation. In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2025. URL https://arxiv.org/abs/2408.00764.

Yue Huang, Zhengzhe Jiang, Yuchen Ma, Yu Jiang, Xiangqi Wang, Yujun Zhou, Yuexing Hao, Kehan Guo, Pin-Yu Chen, Stefan Feuerriegel, and Xiangliang Zhang. ProbeLLM: Automating principled diagnosis of LLM failures. In *International Conference on Machine Learning (ICML)*, 2026. URL https://icml.cc/virtual/2026/poster/66530.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents. *arXiv preprint arXiv:2504.07164*, 2025. URL https://arxiv.org/abs/2504.07164.

Yixing Jiang, Kameron C. Black, Gloria Geng, Danny Park, James Zou, Andrew Y. Ng, and Jonathan H. Chen. MedAgentBench: A realistic virtual EHR environment to benchmark medical LLM agents. *arXiv preprint arXiv:2501.14654*, 2025. URL https://arxiv.org/abs/2501.14654.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2310.06770. arXiv:2310.06770.

Xiang Lisa Li, Farzaan Kaiyom, Evan Zheran Liu, Yifan Mai, Percy Liang, and Tatsunori Hashimoto. AutoBencher: Towards declarative benchmark construction. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/eb216114f3eaad22506fd1bc7bbff0ca-Abstract-Conference.html.

Hongzhan Lin, Yang Deng, Yuxuan Gu, Wenxuan Zhang, Jing Ma, See-Kiong Ng, and Tat-Seng Chua. FACT-AUDIT: An adaptive multi-agent framework for dynamic fact-checking evaluation of large language models. *arXiv preprint arXiv:2502.17924*, 2025. URL https://arxiv.org/abs/2502.17924.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as agents. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2308.03688. arXiv:2308.03688; eight environments.

Xing Han Lù, Amirhossein Kazemnejad, Nicholas Meade, Arkil Patel, Dongchan Shin, Alejandra Zambrano, Karolina Stańczak, Peter Shaw, Christopher J. Pal, and Siva Reddy. AgentRewardBench: Evaluating automatic evaluations of web agent trajectories. *arXiv preprint arXiv:2504.08942*, 2025. URL https://arxiv.org/abs/2504.08942.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. AgentBoard: An analytical evaluation board of multi-turn LLM agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/877b40688e330a0e2a3fc24084208dfa-Abstract-Datasets_and_Benchmarks_Track.html.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026. URL https://arxiv.org/abs/2601.11868.

MiniMax. MiniMax M2.7, 2026. URL https://www.minimax.io/models/text/m27.

Moonshot AI. Kimi K3: Open frontier intelligence, 2026. URL https://github.com/MoonshotAI/Kimi-K3.

OpenAI. Introducing SWE-bench verified. https://openai.com/index/introducing-swe-bench-verified/, 2024. A 500-instance subset validated by human annotators.

OpenAI. Codex CLI, 2026a. URL https://github.com/openai/codex. Accessed September 20, 2026.

OpenAI. Introducing GPT-5.4 mini and nano, 2026b. URL https://openai.com/index/introducing-gpt-5-4-mini-and-nano/.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition, 2026c. URL https://openai.com/index/gpt-5-6/.

SpaceXAI. Introducing Grok 4.5, 2026. URL https://x.ai/news/grok-4-5.

Hao Wang, Hanchen Li, Qiuyang Mang, Alvin Cheung, Koushik Sen, and Dawn Song. Do androids dream of breaking the game? systematically auditing AI agent benchmarks with BenchJack. *arXiv preprint arXiv:2605.12673*, 2026. URL https://arxiv.org/abs/2605.12673.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2407.16741.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5d413e48f84dc61244b6be550f1cd8f5-Abstract-Datasets_and_Benchmarks_Track.html. arXiv:2404.07972; 369 tasks in a real operating system.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/8b86cf5ace600c48fd188efbb8dedec8-Abstract-Datasets_and_Benchmarks_Track.html.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024. URL https://arxiv.org/abs/2406.12045. Scores by comparing the final database state against an annotated goal state.

Z.AI. GLM-5, 2026a. URL https://docs.z.ai/guides/llm/glm-5. Model documentation; accessed September 20, 2026.

Z.AI. GLM-5.2, 2026b. URL https://docs.z.ai/guides/llm/glm-5.2. Model documentation; accessed September 20, 2026.

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. Which agent causes task failures and when? On automated failure attribution of LLM multi-agent systems. In *International Conference on Machine Learning (ICML)*, pp. 76583–76599, 2025. URL https://proceedings.mlr.press/v267/zhang25cq.html.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2307.13854. arXiv:2307.13854; 812 tasks graded by functional correctness.

Kaijie Zhu, Jindong Wang, Qinlin Zhao, Ruochen Xu, and Xing Xie. Dynamic evaluation of large language models by meta probing agents. In *International Conference on Machine Learning (ICML)*, 2024. URL https://proceedings.mlr.press/v235/zhu24m.html.

Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, Xiaoteng Ma, Xiaodong Yu, Gowtham Ramesh, Jialian Wu, Zicheng Liu, Pan Lu, James Zou, and Jiaxuan You. Where LLM agents fail and how they can learn from failures. *arXiv preprint arXiv:2509.25370*, 2025. URL https://arxiv.org/abs/2509.25370.

Mingchen Zhuge, Changsheng Zhao, Dylan R. Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, Yangyang Shi, Vikas Chandra, and Jürgen Schmidhuber. Agent-as-a-Judge: Evaluate agents with agents. In *International Conference on Machine Learning (ICML)*, 2025. URL https://proceedings.mlr.press/v267/zhuge25a.html.

A Implementation and Detailed Protocols

Internal model configuration. The provided configuration assigns GPT-5.6-sol to probe construction, semantic review, trajectory analysis, pairwise judging, and claim generation. The reviewer uses the builder's model settings in a separate read-only session. The induction judge also classifies held-out failures and checks their assigned groups. The embedding configuration uses `google/gemini-embedding-2`. Builder and reviewer calls use the identifier `openai/gpt-5.6-sol`; analysis and induction use `gpt-5.6-sol`. Generative roles default to temperature zero. These internal roles are configured separately from the target agents listed in Table 4.

A.1 Probe construction and search

The implementation supports discovery, corpus consolidation, and controlled comparison. The builder runs on the host; target probes use Harbor task directories and run in Docker. Settings are specified through configuration files and per-run overrides.

For a benchmark task t, seed selection requires

$$\Gamma(t) = \gamma_{\mathrm{read}}(t) \wedge \gamma_{\mathrm{bite}}(t) \wedge \gamma_{\mathrm{sound}}(t). \tag{8}$$

The gates cost zero, one, and two rollouts in order; rejection stops further evaluation. The reviewability gate checks the available verification sources, including whether assertions are external or hidden behind an apparent exact-match check. The failure gate runs the target once and requires $g_t(A(t)) < 1$. The soundness gate requires a passing reference and a failing null run. Tasks whose evaluation requires a model judge are identified explicitly in the probe metadata. In the benchmark survey, 88.1% of 45,811 tasks had executable verdicts; judged tasks were concentrated in suites with simulated users.

The builder receives the complete seed and must preserve its question and required outcome. The identity check addresses semantic drift that executable controls alone can miss. For example, a GAIA question about which scientist first predicted thinking machines was transformed into an unrelated code-editing task. A matching checker and reference could make that candidate executable without preserving the source task.

Admission is the conjunction $\Lambda(s) = \lambda_{\mathrm{exec}} \wedge \lambda_{\mathrm{read}} \wedge \lambda_{\mathrm{review}}$. The reviewer has read-only access and no shell; its verdict requires a rationale grounded in the instruction and checker. Candidate construction stops at either five rounds or 30 minutes. The time limit controls the cost of individual rounds, which the round limit alone does not bound.

Let $\mathcal{C} = \mathcal{C}_1 \times \cdots \times \mathcal{C}_d$ be the coverage space derived from the campaign intent and seeds, and let $n(c)$ count admitted probes in cell c. Exploration selects

$$c^\star \in \arg\min_{c \in \mathcal{C}} n(c), \tag{9}$$

with uniform tie-breaking. Each new failure label queues three sibling builds, with exploration forced after three consecutive verification builds. The coverage policy is suited to campaigns with tens of expensive environment builds. For comparison, an exploration term of the form $\sqrt{\ln N / n}$ is approximately 1.3 at $N = 30, n = 2$, comparable to a reward bounded by one. This motivates the simpler allocation rule at the scale studied here.

Sibling reproduction is based on the explained mechanism. Labels such as `inconsistent-timing-artifact` and `invalid-timing-artifact` can describe the same failure, so exact label equality is not used as a reproduction test. A sibling that produces the mechanism in several rollouts still contributes one reproducing environment.

A.2 Similarity calibration and induction

Write $\mathcal{D}$ for the failures available to an induction run. For held-out evaluation, select every fourth failure in corpus order to form the 25% hold-out set $\mathcal{H}$ before the pipeline and set $\mathcal{D} = \mathcal{R} \setminus \mathcal{H}$; for corpus description, use the available corpus. The mean in Equation (6) is $\bar{\phi} = |\mathcal{D}|^{-1} \sum_{r \in \mathcal{D}} \phi(r)$. Centering, calibration, grouping, and claim construction operate on $\mathcal{D}$.

Given calibrated thresholds $\tau_- < \tau_+$, pair decisions are

$$\mathrm{same}(r, r') = \begin{cases} 1 & \mathrm{sim}(r, r') \geq \tau_+, \\ J(r, r') & \tau_- < \mathrm{sim}(r, r') < \tau_+, \\ 0 & \mathrm{sim}(r, r') \leq \tau_-. \end{cases} \tag{10}$$

In the ambiguous band, only pairs retained by the nearest-neighbor trim are judged; unjudged band pairs introduce no graph edge. Calibration uses empirical judge agreement on a stratified sample from the full similarity range:

$$\begin{aligned} \tau_+ &= \min\{\tau : \widehat{\Pr}(J = \text{same} \mid \mathrm{sim} \geq \tau) \geq q_+\}, \\ \tau_- &= \max\{\tau : \widehat{\Pr}(J = \text{different} \mid \mathrm{sim} \leq \tau) \geq q_-\}. \end{aligned} \tag{11}$$

Using the full range ensures that threshold estimation includes pairs outside the current working band. The targets are $q_+ = 0.90$ and $q_- = 0.98$.

For a component K with at least two failures, the density statistics are

$$d_{\mathrm{in}}(K) = \frac{1}{\binom{|K|}{2}} \sum_{\{r,r'\} \subseteq K} \mathrm{sim}(r, r'), \qquad d_{\mathrm{out}}(K) = \max_{r \in K,\, r' \in \mathcal{D} \setminus K} \mathrm{sim}(r, r'). \tag{12}$$

Edge weights are the centered cosine similarities, $w(r, r') = \mathrm{sim}(r, r')$, on the retained edges. To split a violating component, sort its edges by increasing weight and remove them one at a time until it disconnects. Recursively apply the same test to each resulting component, using the remaining edges. This greedy procedure approximates the grouping objective in Equation (7). When $K = \mathcal{D}$, no outside failure exists; the implementation instead requires $d_{\mathrm{in}}(K) \geq \tau_+$. Singletons are retained without a density test.

The boundary comprises the k_{nn} non-members maximizing $\max_{r \in K} \mathrm{sim}(r, r')$. A generated claim is tested on every member and boundary failure. The output includes the supporting failures, exclusions, and any failure to generate a distinguishing claim.

Algorithm 1: Induction with held-out evaluation. For full-corpus description, omit the hold-out split and membership evaluation.

Input: failures $\mathcal{R}$, judge J, margin μ, precision targets $q_\pm$, neighbor count k_{nn}, hold-out fraction h

Output: groups, distinguishing claims, boundaries, held-out agreement

Select every fourth failure for $\mathcal{H}$ ($h = 0.25$); set $\mathcal{D} = \mathcal{R} \setminus \mathcal{H}$
Embed causes in $\mathcal{D}$, subtract their mean, and normalize
Calibrate (τ_-, τ_+) on a stratified sample of pairs from $\mathcal{D}$
$E \leftarrow \{(r, r') \in \mathcal{D}^2 : \mathrm{sim}(r, r') \geq \tau_+\}$
foreach *retained nearest-neighbor pair in the ambiguous band* **do**
  if $J(r, r') = \text{same}$ **then** add (r, r') to E
end
$w(r, r') \leftarrow \mathrm{sim}(r, r')$ for each edge in E
$\mathcal{K} \leftarrow$ connected components of $(\mathcal{D}, E)$
while *some non-singleton $K \in \mathcal{K}$ fails its density test* **do**
  remove edges of K in ascending weight order until K disconnects
  replace K with the resulting connected components
end
foreach $K \in \mathcal{K}$ **do**
  $B_K \leftarrow$ nearest k_{nn} non-members in $\mathcal{D}$
  $c_K \leftarrow$ claim covering K and excluding B_K, or UNRESOLVED
end
Assign held-out failures using the claims; confirm assignments with J against each selected group's first member
return $\{(c_K, K, B_K)\}$ *and held-out agreement*

Held-out scoring. Let $a(r)$ be the group selected by the classification model after reading the induced descriptions and the held-out failure, or $\perp$ if no description matches. The classifier sees descriptions

rather than the groups' member failures. For each selected group K, let r_K be its first member in corpus order. The reference check is $J(r, r_K)$, using the same pairwise prompt as grouping, without supplying the generated description. For $\mathcal{H}_K = \{r \in \mathcal{H} : a(r) = K\}$, the implementation logs

$$n_K = |\mathcal{H}_K|, \qquad u_K = \sum_{r \in \mathcal{H}_K} J(r, r_K).$$

These are the number of assignments and the number confirmed for each group; held-out agreement counts confirmations among assignments. No-match outputs and failed classification calls do not enter these counts. Both classification and pairwise confirmation use the induction judge model. The reference labels are therefore supplied by model-based pair comparisons.

Embedding text. The serialization $\kappa(r)$ joins nonempty fields with spaces in this order: summary, explanation, corrected action, trigger, and stage. Trigger and stage are prefixed by "Triggered by:" and "Stage:" respectively. If all fields are empty, the implementation falls back to the failure label.

A.3 Hyperparameters

Table 3 lists the exposed settings. The corpus analysis uses $\mu = 0.02$; the component study uses $\mu = 0.05$, and both settings are compared on the full corpus. The thresholds $\tau_\pm$ are calibration outputs. The neighbor count controls the number of ambiguous pairs judged and the number of boundary failures considered.

Table 3: Hyperparameters used in the experiments. The density-margin sweep is reported in Figure 4; fixed thresholds are used only in the corresponding ablation.

Symbol	Stage	What it controls	Value
$r_{\max}$	admission	rounds of build/verify/review before a candidate is discarded	5
T_{build}	admission	wall-clock budget for one candidate, independent of rounds	30 min
e	search	verification builds before an exploration build is forced	3
m	search	siblings queued for a newly seen failure	3
k	measurement	rollouts per environment	3 / 6
m_0	reproduction	siblings required before a recurrence verdict is assigned	3
θ	reproduction	fraction of siblings that must reproduce for AGENT BEHAVIOUR	$2/3$
q_+	induction	precision distance must reach before it may join a pair automatically	0.90
q_-	induction	precision distance must reach before it may separate a pair automatically	0.98
μ	induction	density margin a component must clear to survive uncut	0.02
k_{nn}	induction	neighbours kept per failure, for the judged band and for B_m	15
h	induction	failures withheld from the whole pipeline	0.25
τ_+, τ_-	induction	derived from $q_\pm$; fixed fallbacks 0.15 and -0.10	derived

B Corpus and Failure Evidence

B.1 Agent configurations

The panel contains 22 model–harness configurations, with one discovery campaign per configuration. A 16-configuration campaign set includes 13 models under Terminus 2, Claude Code with Sonnet 4.5, and Codex with GPT-5.6-sol and GPT-5.4-nano. Six additional configurations support the matched design: GPT-5.6-luna and GPT-5.6-terra under Terminus 2, and Sonnet 5 under Goose, opencode, OpenHands, and Aider. Table 4 reports the 1,244 retained failures.

Campaign counts describe the observations collected for each configuration. For example, Terminus 2 with Sonnet 4.5 contributes 143 failures in 52 groups, while Claude Code with the same model contributes 37 failures in 16 groups. Such counts reflect the tasks and depth reached during discovery; matched failure rates are reported separately in Appendix D.

Table 4: Corpus contributions by configuration. Groups includes all induced groups; shared counts those reached by more than one configuration. Largest is the total size of the largest group containing a failure from the configuration. Group counts are non-additive across rows.

Configuration	Records	Groups	Shared	Largest
Terminus 2 / DeepSeek-V4-Pro	132	**68**	7	8
Terminus 2 / GPT-5.6-sol	132	58	4	8
Terminus 2 / Claude Sonnet 5	105	56	5	4
Terminus 2 / Claude Sonnet 4.5	143	52	7	**15**
Terminus 2 / Kimi K3	87	52	6	8
Terminus 2 / Grok 4.5	84	47	3	5
Terminus 2 / GLM-5	72	35	0	4
Terminus 2 / GPT-5.6-luna	64	33	4	8
Terminus 2 / GPT-5.4-nano	73	30	3	12
Terminus 2 / Claude Opus 5	55	24	1	4
Terminus 2 / GPT-5.6-terra	50	21	2	6
OpenHands / Claude Sonnet 5	40	20	2	5
Terminus 2 / Qwen3.8-Max	39	20	5	5
Goose / Claude Sonnet 5	35	16	3	6
Claude Code / Claude Sonnet 4.5	37	16	0	3
Terminus 2 / MiniMax-M2.7	35	15	2	**15**
opencode / Claude Sonnet 5	30	14	2	6
Terminus 2 / Claude Opus 4.8	11	8	2	**15**
Codex / GPT-5.6-sol	9	7	1	4
Terminus 2 / GLM-5.2	11	6	1	4
Codex / GPT-5.4-nano	0	0	0	0
Aider / Claude Sonnet 5	0	0	0	0
Corpus	**1,244**	**565**	**27**	**15**

B.2 SEED COVERAGE

The seed tasks span retrieval, performance engineering, data pipelines, code editing, terminal operation, tabular analysis, spreadsheets, clinical tasks, customer service, and law. Table 5 provides the recorded seed-level breakdown for the 16-configuration campaign set. Five entries account for approximately 94% of the reported group counts. Banking-support dialogue and data-pipeline repair contribute many observations, consistent with their extended workflows and multiple opportunities for operational failure.

Seed eligibility also depends on executability and whether the target still fails the task. Among twelve candidate tasks drawn from SWE-bench Verified and SWE-bench Pro, six were solved by the target and six could not be built. Additional candidate suites failed to produce usable seeds because their tasks were saturated, their runs produced no rewards, or their reference solutions failed. These outcomes motivate task-level seed gates rather than assuming every benchmark supplies suitable probes.

Table 5: Seed-level breakdown of the 16-configuration campaign set. Groups are counted within each seed and need not sum to a corpus-wide grouping. Configurations counts those contributing at least one failure under the seed.

Suite	Seed task	Groups	Records	Configs.
tau3-bench	banking_knowledge task-081	**162**	234	7
ade-bench	airbnb009	116	151	10
gaia	00d579ea	73	102	9
gaia	2a649bb1	72	96	9
ade-bench	asana005-hard	57	80	**12**
gso	pandas-9097263	7	14	4
tau3-bench	retail-32	6	10	5
gso	numpy-893db31	4	14	2
lawbench	2-4-23-zero-shot	4	8	3
terminal-bench	polyglot-c-py	2	5	2
dabstep	1459	2	3	2
medagentbench	task10_1	2	2	2
dabstep	17	1	1	1

B.3 ATTRIBUTION AND BREADTH

Table 6 reports breadth and sibling evidence separately. A singleton describes one observed failure; its attribution depends on the sibling tests linked to the source failure. Cross-configuration membership describes breadth across agents; sibling tests measure recurrence across task contexts.

Each test is keyed to the environment whose failure motivated its siblings. Across 146 tests, 311 failures and 161 groups inherit test evidence. Seven groups pool more than one distinct test, and four contain tests with different verdict classes. Tests can support several groups when trajectories from one tested environment yield failures that induction separates. Within a group, each test contributes its reproduction count only once. Of the 27 tests outside the 119-candidate experiment in Figure 3, 14 have insufficient tested siblings. The remaining 404 groups have no linked sibling test.

B.4 EXAMPLE CLAIMS

Table 1 presents four distinguishing claims. The 565 induced groups comprise 361 single-failure groups, 177 multi-failure groups from one configuration, and 27 groups spanning configurations. The largest group contains 15 failures. Overall, 29 of 565 groups are unresolved, including two of the 27 shared groups. The other 536 groups have distinguishing claims.

Table 6: Evidence supporting the 565 induced groups. Breadth counts observed failures and configurations. Attribution pools distinct sibling tests associated with member failures; the two parts describe different dimensions of support.

Breadth of supporting failures	**Groups**	**Share**
One failure	361	63.9%
Two or more failures, one configuration	177	31.3%
Two or more failures, spanning two configurations	22	3.9%
Two or more failures, spanning three or more	**5**	0.9%
Total	**565**	**100%**
Attribution from sibling tests	**Groups**	**Share**
Groups linked to sibling tests	**161**	28.5%
agent behavior	33	5.8%
weak	32	5.7%
task-specific	53	9.4%
unverified	43	7.6%
Groups without sibling tests	404	71.5%

C Additional Induction Results

C.1 Granularity

Repeated evaluations retain the benefit of the density constraint. On the 961-failure corpus, two evaluations of the full method yield held-out agreement of 91.8% and 91.4%, compared with 78.0% and 70.7% without the constraint. The full method retains higher agreement in both evaluations, supporting the role of the density constraint in separating distinct errors. The additional 940-failure audit set yields 91.1% agreement (Appendix C).

With the same pair cache, grouping of the 961-failure corpus yields 505 groups: 381 singletons and 124 multi-failure groups. On the 940-failure audit set, induction yields 491 groups and 91.1% held-out agreement.

Figure 4 reports the density-margin sweep on 961 failures. Without the constraint, one component contains 650 failures. At the other extreme, $\mu = 0.20$ produces 665 groups, of which only 38 contain multiple failures and none spans multiple configurations. Intermediate settings retain more multi-failure groups while achieving higher held-out agreement than unconstrained grouping.

On 1,244 failures, $\mu = 0.05$ yields 659 groups, 75% singletons, 13 shared groups, a largest group of 10 failures, and 89.3% held-out agreement. At $\mu = 0.02$, the corresponding values are 565, 64%, 27, 15, and 88.5%. We select $\mu = 0.02$ for its greater multi-failure and cross-configuration support. Both settings are reported to show the resolution tradeoff.

C.2 Comparison with a fixed taxonomy

Figure 2 compares induction with eight failure categories on 961 failures. The taxonomy assigns 956 failures and leaves five unassigned. Its median category contains 94 failures, compared with a median group size of one under induction. The category `incomplete` contains 264 failures and includes 163 induced groups. The category-level group counts are non-additive. The median pairwise Jensen–Shannon divergence between configurations is 0.21 in the taxonomy representation.

C.3 Centering and calibration

On 12 failures from three hand-labeled modes, centering increases the gap between mean within-mode and between-mode cosine similarity from 0.048 to 0.162. The respective means are 0.704 and 0.656 before centering, and 0.027 and -0.135 after centering. This gap measures how well similarity distinguishes the labeled modes.

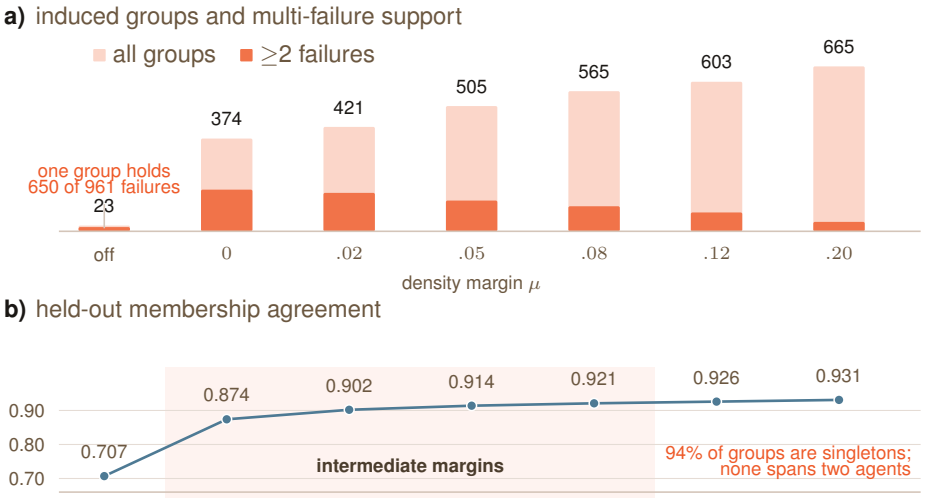


Figure 4: **Effect of the density margin** on 961 failures. Top: all groups and groups with at least two failures. Bottom: held-out membership agreement. Larger margins progressively split groups and increase the singleton share.

Figure 5 visualizes the fixed-band and calibrated-band comparison reported in Section 4.6. Across corpora of 664, 940, and 1,244 failures, the calibrated join thresholds are 0.630, 0.592, and 0.568, respectively.

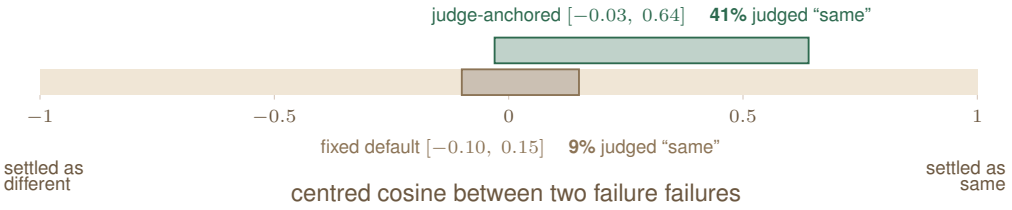


Figure 5: **Similarity-band calibration.** The calibrated band concentrates pairwise judging on more candidate matches than the fixed thresholds. The displayed band is the calibration comparison; thresholds are derived separately for each induction run.

C.4 Pair selection and computational cost

Section 4.6 reports the neighbor-trimming and cache savings. Of the 7,267 judged pairs on the 940-failure audit set, 2,711 are marked as matching. The join threshold changes by 0.024 between these two runs. Cache reuse depends on which pairs fall in the calibrated band, as well as which failure pairs are already stored.

C.5 Human evaluation of failure descriptions

We randomly sample 30 held-out failures assigned to groups with generated descriptions, keeping the descriptions fixed before annotation. Each failure is paired with its assigned description and a similar alternative from another group, selected by the maximum centered cosine similarity between the failure and any member of that group. Two annotators independently evaluate all 60 pairs, yielding 120 judgments. Each item provides the task requirements, the specified failure, relevant trajectory excerpts, checker feedback, and one candidate description; the full trajectory is also available. Items are shuffled and presented separately, with model and group identities, automatic judgments, and description type hidden. Annotators decide whether the description accurately characterizes the failure, answering yes, no, or insufficient information and giving a brief evidence-based reason. A

match requires the described error and key task conditions to agree with the trajectory; different errors, incompatible conditions, or contradictory evidence receive a no judgment. Unresolved evidence receives insufficient information. Equivalent wording is accepted, while a shared broad outcome alone does not establish a match. Both description types use the same criteria. For each annotator, we divide yes judgments on assigned descriptions and no judgments on alternatives by 30, then average the respective rates across annotators. They accept 28/30 and 26/30 assigned descriptions and each reject 25/30 alternatives, giving mean acceptance and rejection rates of 90.0% and 83.3%, respectively.

D Agent Comparison and Resource Use

D.1 Controlled replay

Replay fixes the environment pool and execution settings across configurations. The pool contains exploration probes sampled evenly across source campaigns. Sibling probes are excluded because they are constructed specifically to reproduce a target's failure. Each probe retains its validated execution settings. The replay pilot includes 20 environments, ten of which all tested agents fail. Eight of the remaining environments form the discriminating pool.

Each configuration is scheduled for six rollouts in each of the eight environments. Comparison uses five commonly measured environments and truncates each to the shallowest available depth: $(6, 6, 4, 1, 1)$, totaling 18 matched trials. Environments that cannot start under a harness, including network-isolated environments requiring a network installation, are excluded from that common set. The Codex/GPT-5.4-nano configuration has no valid replay executions and is excluded from failure-rate estimates.

Figure 6 reports point estimates and 95% Wilson intervals. Kimi K3 under Terminus 2 has a failure rate of 22.2%, and GPT-5.4-nano under Terminus 2 has a failure rate of 94.4%. The Opus 4.8/Opus 5 paired comparison has discordant counts of six and one, with exact $p = 0.13$. The Sonnet and GLM version pairs yield $p = 0.38$ and $p = 0.45$. For GPT-5.6-sol under Codex versus Terminus 2, $p = 0.73$.

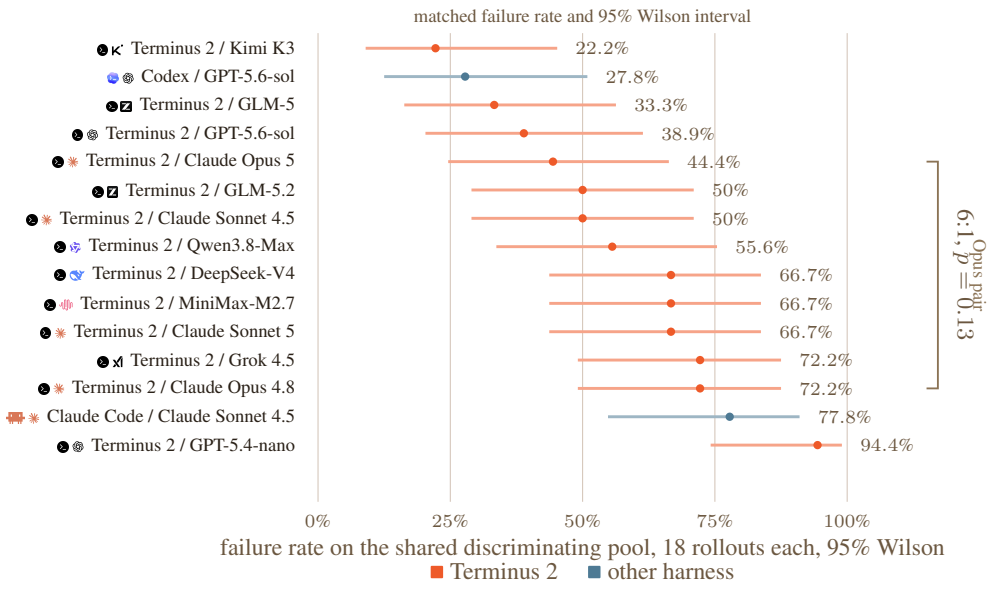


Figure 6: **Matched replay results.** Rates use 18 matched trials per valid configuration across five environments; intervals are 95% Wilson intervals. The pool was selected for discriminating failures. Infrastructure-only executions are excluded.

D.2 CONFIGURATION CONTRASTS

Figure 7 displays selected model and harness contrasts using the group counts in Table 4. The model comparisons hold Terminus 2 fixed. The harness comparisons hold Sonnet 4.5 or GPT-5.6-sol fixed. These counts summarize the failure failures collected during discovery.

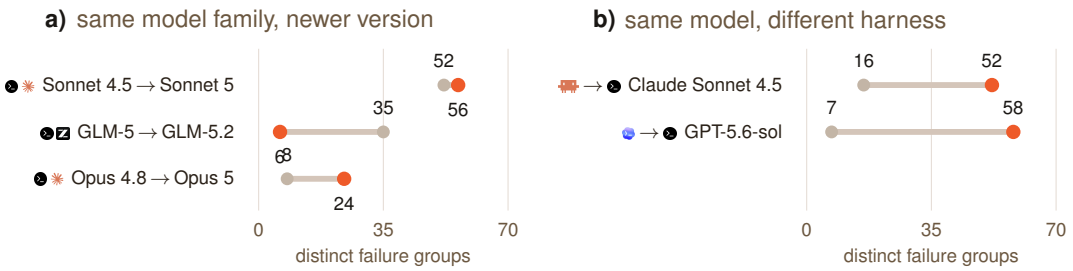


Figure 7: **Group counts for selected configurations.** (a) Model-version pairs under Terminus 2. (b) Harness pairs with the same model. Counts match Table 4. The pale and solid markers denote the first and second configurations, respectively.

The model and harness comparisons hold four seed tasks, builder, engine, intent, and rollout budget fixed. GPT-5.6-luna, GPT-5.6-sol, and GPT-5.6-terra under Terminus 2 define three model pairs. Sonnet 5 under Terminus 2, Goose, opencode, and OpenHands defines six harness pairs. Because environment generation remains adaptive, these campaigns study shared discoveries; replay supplies the fixed-task performance comparison.

Among the 190 pairs of failure-producing configurations, 30 share at least one group. Four of the nine designed pairs share a group, compared with 26 of the other 181 pairs (Figure 8). The three model pairs share 1, 2, and 0 groups; the six harness pairs share 2, 1, and four zeros. Mean overlap is 0.67 versus 0.19 groups per pair at $\mu = 0.02$, and 0.44 versus 0.07 at $\mu = 0.05$. Pairs reuse configurations, so these are descriptive comparisons rather than independent replicates.

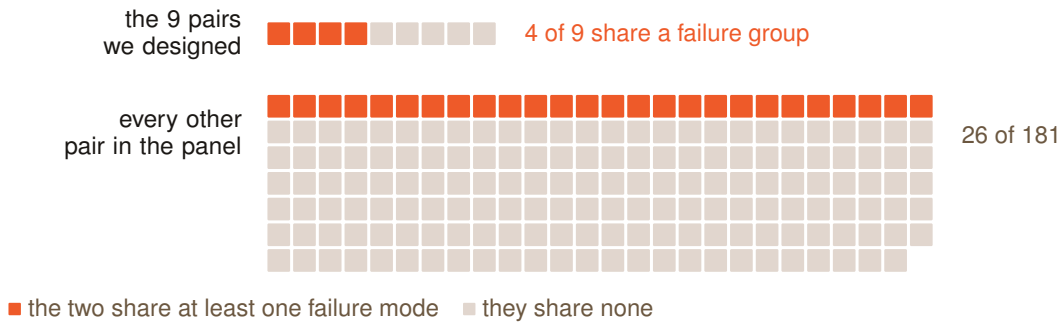


Figure 8: **Failure sharing across configuration pairs.** Each square represents a pair and is filled when the configurations share at least one group. The designed pairs share at a rate of $4/9$; other pairs share at $26/181$.

The harness comparison also differs in observed failure count and timeout rate. Terminus 2 with Sonnet 5 contributes 105 failures, versus 30–40 for each of the other three harnesses, and times out on 46% of rollouts versus 12–13%. It consumes 19.8M input tokens per rollout versus 3.2–8.6M. Restricting shared support to completed rollouts changes the two nonzero harness-pair overlaps from 2 and 1 to 1 and 0; the other overlaps remain zero.

D.3 TOKEN USE

Replay accounting is available for 15 configurations, each scheduled for 48 rollouts. Mean input tokens per rollout range from 112k for Sonnet 5 under Terminus 2 to 3.49M for Sonnet 4.5 under

Claude Code, a 31-fold spread. Among the 13 configurations using Terminus 2, the spread is approximately tenfold. Figure 9 relates token use over the replay pool to the matched failure rates. For the Terminus 2 configurations, Spearman correlation is $\rho = 0.50$ ($p = 0.08$). This association does not identify whether higher token use contributes to failure or results from extended attempts on difficult tasks.

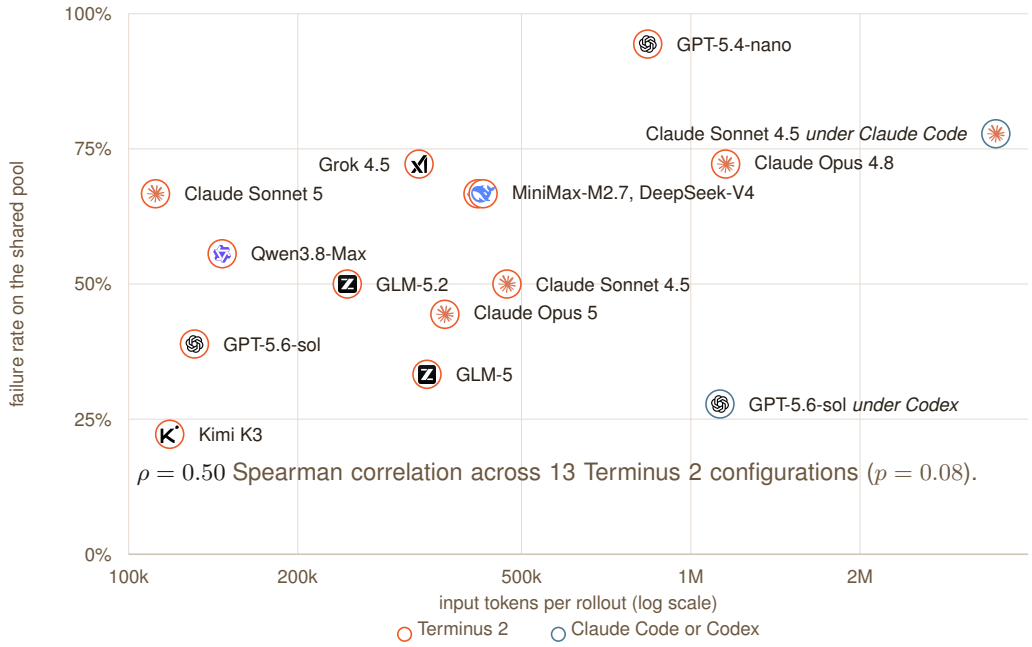


Figure 9: **Replay token use and matched failure rate.** Input tokens per rollout use the replay accounting; failure rates use the 18 matched trials. The horizontal axis is logarithmic. Model marks and harness rings identify configurations; coincident MiniMax-M2.7 and DeepSeek-V4 values share a point.

Across the 16-configuration discovery set, target execution consumes 365M input and 7.3M output tokens over 936 rollouts with recorded usage. These counts include campaign and seed-gate calls reaching the model; the execution audit separately examines all trial outcomes.

E Admission Analysis and Corpus Audit

E.1 Admission-gate findings

The gates run sequentially and stop at rejection. Table 7 summarizes 1,451 build rounds. Execution rejects 349 candidates; semantic review rejects 638 candidates that passed the executable controls. The static source gate has no separately recorded rejection. Inaccessible assertions are instead identified by the reviewer on 59 rounds. These are conditional rejection counts from the sequential pipeline.

Table 7: Admission rejections over 1,451 build rounds.

Gate	Rounds rejected	Evaluation cost
Executed: reference and null controls	349	two rollouts
Read: static checker inspection	0	no rollout
Reviewed: separate read-only agent	638	one review pass

Section 4.4 discusses the most common findings and explains how shortcuts can pass the executable controls. Table 8 gives the full breakdown, including missing instructions, inaccessible assertions, and leaked solutions. A round may receive several findings.

Table 8: Semantic-review findings. Categories may overlap within a build round.

Finding	Rounds
A shortcut receives reward without completing the intended work	565
The checker does not evaluate the requested outcome	323
The checker imposes an unstated requirement	255
The instruction omits a rule needed to solve the task	220
The instruction cannot be followed as written	92
A task claim lacks an implementing mechanism	81
Checker assertions are inaccessible to the reviewer	59
The environment leaks its solution	51

E.2 Task and failure quality

Table 9 summarizes the quality measurements. The manual audit uniformly samples 20 environments and finds one hard error in an expected value. Six of the 20 have either a wrong expected value or an ambiguity in the required output format. Format defects include byte-exact checks on separator spacing or key order that the instruction does not specify. The per-failure classification uses the 940-failure audit set.

The answer-consistency audit finds that 2.2% of 222 eligible variants change their seed's expected answer. For judged environments, 225 reference replays across 72 environments have zero observed reward spread. All nine audited environments failed by every tested agent accept their reference solutions. These controls assess answer consistency, judge repeatability, and reference feasibility, respectively.

Table 9: Corpus quality audit. The manual sample contains 20 environments; failure classification covers 940 failures. Judge consistency measures repeatability of the reference verdict.

Check	Method	Result
Hard ground-truth errors	unbiased sample of 20 environments, manual	5%
Wrong *or* format-ambiguous	same sample	30%
Variant overrules its seed's answer	automatic, all 222 eligible	2.2%
Records that are substantive	per-failure classification, all 940	**95.0%**
Records that are harness artifacts	same	3.1%
Records that are format artifacts	same	1.9%
Judge self-consistency	225 oracle replays over 72 judged environments	spread 0.000
All-fail environments still solvable	reference solution re-run	**9/9**