AGENTLENS: Revealing The Lucky Pass Problem in SWE-Agent Evaluation

Priyam Sahoo[1,2,*], **Gaurav Mittal**[2,†], **Xiaomin Li**[2,†], **Shengjie Ma**[2], **Benjamin Steenhoek**[2], **Pingping Lin**[2], **Yu Hu**[2]

[1]University of Illinois, Urbana-Champaign, IL, USA [2]Microsoft, Redmond, WA, USA

priyams3@illinois.edu, Gaurav.Mittal@microsoft.com

Evaluation of software engineering (SWE) agents is dominated by a binary signal: whether the final patch passes the tests. This outcome-only view treats a principled solution and a chaotic trial-and-error process as equivalent. We show that this equivalence is empirically false. We evaluate 2,614 OpenHands trajectories from eight model backends on SWE-bench Verified. Of the 60 tasks in this corpus, 47 have enough passing trajectories to construct task-level process references, yielding a 1,815-trajectory evaluation subset. Among passing trajectories in this subset, 10.7% exhibit behavior we call a *Lucky Pass*: regression cycles, blind retries, missing verification, or temporally disordered exploration, implementation, and verification. We introduce **AGENTLENS**, a framework for process-level assessment of SWE-agent trajectories, and define **AGENTLENS-Bench**, a dataset of 1,815 trajectories annotated with quality scores, waste signals, divergence points, and 47 task-level Prefix Tree Acceptor (PTA) references. AGENTLENS combines two components. First, it merges multiple passing solutions for the same task into a PTA reference space of correct behaviors. Second, it uses a context-sensitive intent-stage labeler that assigns actions to *Exploration*, *Implementation*, *Verification*, or *Orchestration* using trajectory history rather than tool identity alone. On AGENTLENS-Bench, the composite score separates passing trajectories into Lucky, Solid, and Ideal tiers; decomposes Lucky Passes into five recurring mechanisms; and changes how the eight evaluated model backends are ranked compared with pass rate alone. Across these models, AGENTLENS classifies between 0.5% and 23.2% of successful trajectories as Lucky, and some models move by as many as five rank positions when ranked by quality score instead of pass rate. We plan to release the project repository soon, including AGENTLENS-Bench, the AGENTLENS SDK, and the analysis tooling.

1 Introduction

Software engineering agents have moved quickly from prototypes to systems that resolve real GitHub issues end-to-end. SWE-agent (Yang et al., 2024), OpenHands (Wang et al., 2024b), AutoCodeRover (Zhang et al., 2024), Agentless (Xia et al., 2024), and Devin (Sana Ansari, 2024) all read codebases, edit files, and run test suites without human input. The benchmark anchoring this progress, SWE-bench (Jimenez et al., 2024), evaluates these systems with a binary signal: does the final patch pass the tests? That signal is useful for measuring capability, but insufficient for evaluating behavior. Consider two agents resolving the same issue. One explores the repository in a few targeted steps, identifies the root cause, applies a minimal fix, and verifies it. The other repeatedly attempts similar edits, loops through failed checks, and eventually reaches a working patch through trial and error. Both receive the same SWE-bench label of "resolved." The behavioral difference is real, important for downstream uses of trajectories, and invisible to outcome-only evaluation.

We show that this conflation occurs in practice. Across 1,136 passing agent trajectories from eight model backends on SWE-bench Verified, 10.7% are reached through behavior we call a *Lucky Pass*: regression cycles, blind retries, missing verification, or temporally disordered exploration, implementation, and verification. A

[*]Work done as a student researcher while at Microsoft.
[†]Co-second authors.

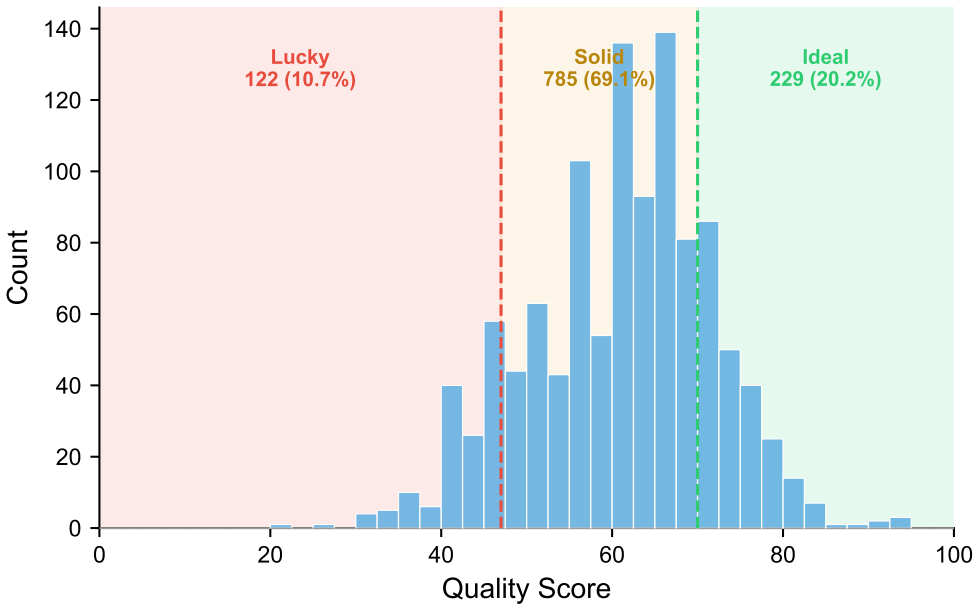

Figure 1: **Passing trajectories are not behaviorally homogeneous.** Among 1,136 passing trajectories in AGENTLENS-Bench, AGENTLENS classifies 229 as Ideal (20.2%), 785 as Solid (69.1%), and 122 as Lucky (10.7%). Binary evaluation treats all of these trajectories as equally successful, while process-aware scoring separates direct, coherent solutions from weak processes that happen to pass.

further 69.1% are Solid but imperfect, and only 20.2% are Ideal: principled, low-waste, and well-ordered. Pass-rate rankings disagree with process-quality rankings on all eight configurations, and Lucky rates range from 0.5% to 23.2% across models.

This matters for three reasons. First, trajectory datasets such as SWE-Gym (Pan et al., 2025), R2E-Gym (Jain et al., 2025), and SWE-smith (Yang et al., 2025) commonly filter on pass rate, treating every successful trajectory as equally valuable supervision. This makes pass-rate filtering a coarse proxy for demonstration quality: a trajectory that reaches the correct outcome through brittle exploration or excessive retry is selected in the same way as a direct, coherent solution. Second, as models converge on pass-rate benchmarks, process quality becomes a useful axis for model comparison. In Section 5.2 and Table 2, we show that ranking models by AGENTLENS quality score changes their ordering relative to pass rate, with some models moving by as many as five rank positions. Third, deployment risk depends on process. Agents that succeed by repeated trial and error may behave unpredictably when repositories are large, tests are expensive, or actions are irreversible.

We introduce AGENTLENS, a framework for process-level assessment of SWE-agent trajectories (Figure 2). AGENTLENS has two technical components. The first is a PTA-based quality reference: instead of comparing a candidate trajectory to a single reference trace, we build a Prefix Tree Acceptor (Oncina et al., 1992) from multiple passing solutions for the same task. The resulting directed acyclic graph encodes a space of known-good strategies. This lets AGENTLENS recognize valid alternative solution paths while flagging redundant retries, irrelevant exploration, and divergence from known successful processes. The second is context-sensitive intent-stage labeling. Each action is assigned to one of four cognitive phases: *Exploration*, *Implementation*, *Verification*, or *Orchestration*, using trajectory history rather than tool identity alone. This resolves common ambiguities such as terminal commands: `grep` remains exploratory even after an edit, whereas `pytest` is verification.

Together, these steps give AGENTLENS a task-specific reference for judging not only whether an agent solved a task, but how it moved through the solution process. Each raw trajectory is first converted into an intent-labeled state sequence. Passing trajectories for the same task are then merged into a task-level PTA, and new trajectories are scored against that PTA to compute a composite quality score, tier label, divergence point,

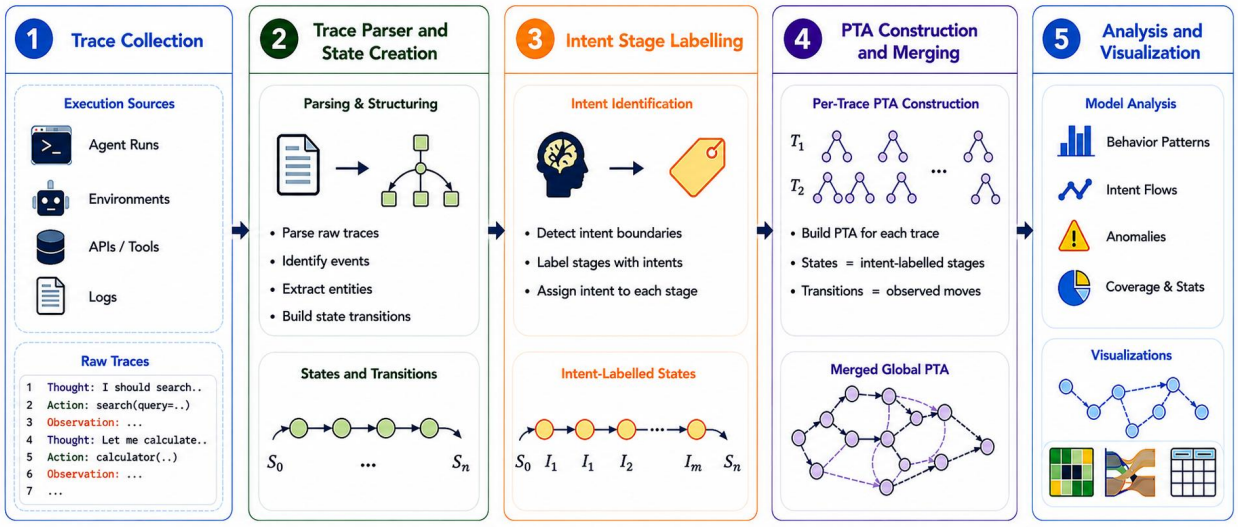

Figure 2: **AgentLens workflow.** AgentLens starts from raw execution traces and converts them into structured state sequences. Intent-stage labeling assigns each state a process role, such as exploration, implementation, testing, or cleanup. These intent-labeled states become the states used for per-trace PTA construction. Passing trajectories for the same task are then merged into a task-level PTA, which represents the known-good solution strategies for that task. New trajectories are scored against this task-level reference to produce quality tiers, divergence points, waste reports, and visual analyses.

and structured inefficiency report. We validate the intent-stage labels with a seven-annotator agreement study, obtaining Fleiss' $\kappa = 0.933$, and evaluate the scoring pipeline on 2,614 trajectories from 60 SWE-bench Verified tasks. Of these 60 tasks, 47 have enough passing trajectories to build a task-level PTA. These 47 tasks form **AgentLens-Bench**, which contains 1,815 process-annotated trajectories with quality scores, waste annotations, divergence metadata, and one ground-truth PTA per task. Because each PTA is tied to a distinct SWE-bench task, this collection provides task-diverse references for scoring trajectories, filtering training pools, and analyzing failures across different solution spaces.

Below are our main contributions:

- **AgentLens-Bench.** To our knowledge, AgentLens-Bench is the first process-annotated SWE-agent trajectory dataset. It contains 1,815 trajectories from 47 PTA-eligible SWE-bench Verified tasks, with 40-column feature vectors, one task-level ground-truth PTA per task, waste annotations, divergence metadata, and tier labels. Because each PTA represents the known-good solution space for a distinct task, the collection provides task-diverse references for trajectory scoring, filtering, failure analysis, and future process-aware training studies.
- **The Lucky Pass finding and taxonomy.** We show that 10.7% of passing trajectories reach correct patches through weak processes. These Lucky Passes decompose into five behavioral categories with significant model associations ($\chi^2(28) = 102.47$, $p < 0.0001$). Across the eight evaluated model backends, the share of successful trajectories classified as Lucky ranges from 0.5% to 23.2%.
- **Context-sensitive intent labeling.** We introduce a trajectory-history-aware labeler that resolves exploration-vs-verification ambiguity in terminal commands, validated at $\kappa = 0.933$ on 200 states with 96.0% raw agreement across seven annotators.
- **PTA-based process references and quality scoring.** We introduce a PTA-based representation that merges passing trajectories for the same task into a task-level reference of known-good solution strategies. AgentLens then scores new trajectories by combining structural alignment with coverage, coherence, and temporal-profile signals. On a pilot validation set, this combined score significantly separates passing from failing trajectories ($p = 0.0017$).
- **Planned open-source tooling.** We plan to release an SDK and web interface for process-aware trajectory

analysis through the project repository. The tooling will support ATIF trajectory logs (Harbor Framework, 2026) and OpenHands traces (Wang et al., 2024b). The scoring pipeline is deterministic and does not require LLM calls or external API access. Because ATIF provides a standardized JSON format for agent trajectories, agents that export or are converted to ATIF can be analyzed by AGENTLENS without changing the core pipeline. For agents that do not yet support ATIF, adding a lightweight trace adapter that maps their logs into the same intent-labeled state representation is sufficient for AGENTLENS to analyze them.

2 Related Work

Outcome-based SWE-agent benchmarks. SWE-bench (Jimenez et al., 2024) established binary pass/fail as the standard for coding-agent evaluation, and subsequent benchmarks refine this outcome signal through human validation, decontamination, live issue streams, multi-language coverage, or realistic task pricing (Chowdhury et al., 2024; Badertdinov et al., 2025; Zhang et al., 2025; Zan et al., 2025; Miserendino et al., 2025). Adjacent code benchmarks such as LiveCodeBench (Jain et al., 2024), BigCodeBench (Zhuo et al., 2024), and TerminalBench (Merrill et al., 2026) similarly evaluate final correctness. ABC (Zhu et al., 2025) documents measurement errors in this benchmark family, including insufficient test coverage. These works improve outcome evaluation; AGENTLENS instead measures the process that produced the outcome.

Process-level trajectory evaluation. Graphectory (Liu et al., 2026) is the closest prior work: it encodes execution traces as graphs and computes process-centric metrics independently of task success. Other studies characterize successful and failing SWE-agent trajectories through thought-action-result patterns, length, variance, or patch quality (Bouzenia and Pradel, 2025; Majgaonkar et al., 2025), while TRAIL (Deshpande et al., 2025), Agent-as-a-Judge (Zhuge et al., 2024), AgentBoard (Ma et al., 2024), and AgentBench (Liu et al., 2023) study broader agent evaluation beyond final success. Process reward models and step-level supervision make a related argument that intermediate reasoning signals differ from outcome labels (Uesato et al., 2022; Lightman et al., 2023; Wang et al., 2024a; Zheng et al., 2025; Chae et al., 2025; Shum et al., 2025). AGENTLENS differs by providing deterministic, decomposable process scores for SWE trajectories, using context-sensitive intent labels, PTA references built from multiple passing solutions, and structured inefficiency attribution with divergence localization.

Trajectory datasets for SWE agents. SWE-Gym (Pan et al., 2025), R2E-Gym (Jain et al., 2025), SWE-smith (Yang et al., 2025), and OpenHands logs (Wang et al., 2024b) provide execution traces or training instances, but they filter or organize trajectories primarily by outcome. To our knowledge, no existing coding-agent dataset provides per-trajectory quality scores, ground-truth reference graphs, divergence localization, and waste annotations together. AGENTLENS-Bench fills this gap. Appendix A.4 provides compact dataset and framework comparison tables.

3 How AGENTLENS Works

AGENTLENS evaluates a candidate trajectory in four stages: it parses raw logs into labeled states, constructs a task-specific reference graph from passing solutions, scores the candidate against that reference, and returns a structured quality report.

3.1 From raw logs to labeled states

An agent log is a sequence of tool calls paired with environment responses. AGENTLENS parses each step into a state containing the tool, target file, affected line range, content hash, trajectory position, and an intent-stage label. We use four cognitive phases: **Exploration** (E; reading files, searching, listing directories), **Implementation** (I; editing or creating source files), **Verification** (V; running tests, checking errors, re-reading edited files), and **Orchestration** (O; bookkeeping and reasoning steps). These phases follow empirical studies of developer cognition (Ko et al., 2006; Alaboudi and LaToza, 2021) and prior trajectory analysis (Liu et al., 2026).

A key challenge is that tool identity alone is insufficient. For example, `read_file(test_api.py)` may be exploratory before any patch is written but verifying after an implementation step. We therefore use a deterministic, rule-based, context-sensitive labeler that tracks whether implementation has occurred and which files have been edited. Search and file-inspection commands such as `grep`, `cat`, and `ls` are labeled E, test-running commands such as `pytest` are labeled V, source edits are labeled I, and reads of previously edited files are labeled V. The full registry and rule decision flow are provided in Appendices B.1 and B.3. Section 5.4 reports the reliability study that validates these labels before they are used for scoring.

3.2 Building a PTA reference

A single reference trajectory cannot represent the diversity of correct strategies: two agents may solve the same task through different but valid sequences. AGENTLENS instead constructs a Prefix Tree Acceptor (PTA) (Oncina et al., 1992) from $k \geq 2$ passing trajectories for the same task. Shared prefixes are merged into common nodes, while divergent but successful strategies form branches. Each root-to-terminal path therefore represents one known-good solution, and the resulting directed acyclic graph encodes a space of correct behaviors rather than a single exemplar (Figure 3).

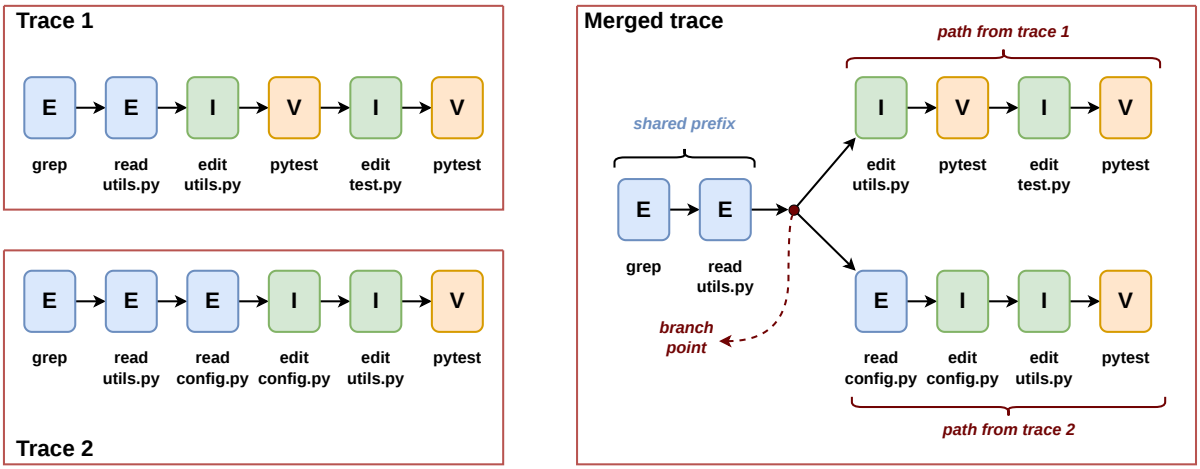

Figure 3: **PTA construction.** Two individual passing traces sharing an early exploration prefix but diverging at implementation are merged into a single DAG. Shared nodes reflect equivalent actions across agents; branches reflect genuine strategic divergence.

During construction, states from different trajectories are merged when they represent equivalent actions. The equivalence engine handles surface variation such as different tool names, overlapping file regions, and equivalent terminal commands. For example, `grep` and `rg` calls with the same search intent can match the same PTA state rather than being treated as different actions. Appendix B.2 gives the full equivalence cascade and thresholds.

3.3 Scoring a candidate trajectory

Given a candidate trajectory τ_c and task PTA $\mathcal{G}$, AGENTLENS computes four complementary signals. **Structural alignment** (Φ_{struct}) measures whether the candidate visits PTA states in roughly the right order, combining ordered recall with unordered precision. **Set coverage** (Φ_{cov}) measures the fraction of PTA states matched by the candidate regardless of order. **Trajectory coherence** (Φ_{coh}) summarizes the intent-stage sequence, rewarding forward progress such as E→I→V and penalizing backtracks and blind retries. **Temporal profile similarity** (Φ_{temp}) compares the candidate's stage distribution over early, middle, and late trajectory segments against the PTA using Jensen-Shannon divergence (Lin, 1991). Intuitively, the first two signals ask whether the agent touched the right parts of the solution space, while the latter two ask whether it moved through them in a plausible problem-solving order. Appendix B.4 provides formal definitions and a worked scoring example.

The four signals are combined into a 0–100 quality score:

$$f(\tau_c, \mathcal{G}) = 0.20 \cdot \Phi_{\text{struct}} + 0.15 \cdot \Phi_{\text{cov}} + 0.30 \cdot (100 \cdot \Phi_{\text{coh}}) + 0.35 \cdot (100 \cdot \Phi_{\text{temp}}). \quad (1)$$

Φ_{struct} and Φ_{cov} are percentage-based, while Φ_{coh} and Φ_{temp} are rescaled from $[0, 1]$. The weights were selected by grid search on a disjoint pilot calibration set and then held fixed for all scaled experiments. Behavioral signals receive 65% of the total weight, reflecting that structural coverage and process quality fail on different trajectories.

3.4 Reports, tiers, and waste signals

The final report includes the composite score, per-stage coverage, divergence-point localization, and structured inefficiency analysis. Waste is detected in five categories: regression loops, blind retries, redundant steps, unnecessary exploration, and cyclic patterns. Each instance is localized to trajectory steps and attributed to tools where possible. For downstream analysis, we use fixed quality tiers. Passing trajectories with score ≥ 70 are **Ideal**, those with $47 \leq$ score < 70 are **Solid**, and those with score < 47 are **Lucky**. Failing trajectories are labeled **Partial-fail** when score ≥ 47 and **Off-track** otherwise. These thresholds were set on the pilot calibration set and held fixed for the scaled evaluation. Additional report fields and the five-level verdict are described in Appendix B.4.

4 Experimental Setup

Compute environment. All AGENTLENS scoring, PTA construction, stratification, and waste analysis experiments were run locally on a machine with 11 CPU cores and 18GB memory. The pipeline is CPU-only and does not require GPU workers; trajectory generation uses external model API calls and is separate from the post-hoc AGENTLENS analysis reported here.

Dataset. We evaluate AGENTLENS on trajectories generated by the OpenHands coding agent (Wang et al., 2024b) on SWE-bench Verified (Chowdhury et al., 2024). The corpus contains 2,614 trajectories across 60 tasks and eight model backends: GPT-4.1, GPT-4o, GPT-5.2-Codex, GPT-5.3-Codex, Claude Sonnet 4.5, Claude Opus 4.5, Claude Opus 4.6, and Gemini 2.5 Pro (Comanici et al., 2025). Across these trajectories, 1,389 pass, 1,217 fail, and 8 have unrecorded outcomes. PTA construction is task-specific and requires at least two passing trajectories for a task, since a merged PTA is built from multiple known-good solutions. This requirement is satisfied by 47 of the 60 tasks, spanning 1,815 trajectories: 1,136 passing and 679 failing. All scoring, stratification, and waste analysis is performed on this 1,815-trajectory subset, which constitutes AGENTLENS-Bench.

Calibration and holdout. Signal weights were calibrated on a separate pilot set of 278 trajectories across 10 tasks. The pilot was used only for grid-search weight optimization (step 0.05, unit-sum constraint, AUROC-maximizing), yielding $w = (0.20, 0.15, 0.30, 0.35)$ with pilot AUROC $= 0.755$ and pilot F1 $= 0.791$. No pilot trajectories appear in the scaled evaluation set, and the weights are frozen for all subsequent experiments. In the scaled set, PTA construction is task-specific: for each scored trajectory, the reference PTA is built from other passing trajectories for the same task, excluding the trajectory being scored. This lets us assign quality scores to all 1,136 passing trajectories without scoring any trajectory against a PTA that contains itself. Pass/fail discrimination is computed entirely on the scaled set with no pilot data.

Baselines. We compare against three reference strategies: individual trajectory matching, which scores each test trajectory against every passing training trajectory and reports the best match; TF-IDF alignment in the BERTScore style (Zhang et al., 2020); and dense embedding alignment with `text-embedding-3-large`.

Metrics. We report micro-averaged AUROC (Fawcett, 2006) as the primary discrimination metric because task-level class balance varies across evaluation slices. Decision thresholds are selected by Youden's J (Youden, 1950). For significance testing, we report the Kolmogorov–Smirnov test p-value comparing passing and failing score distributions.

5 Results

We organize the results around the main empirical findings first, followed by validation checks. Section 5.1 shows that passing trajectories are not behaviorally homogeneous: 10.7% are Lucky Passes despite producing correct patches. Section 5.2 analyzes these Lucky Passes, decomposes them into five weak-success mechanisms, and shows how process quality changes model comparison. Section 5.3 tests whether the resulting quality score also separates passing and failing trajectories. Section 5.4 validates the intent-stage labels used to construct trajectory states.

5.1 Passing trajectories are not behaviorally homogeneous

The central question for AGENTLENS is whether all successful trajectories should be treated as equally good demonstrations. Across 1,136 passing trajectories eligible for assessment, the answer is no. Applying the fixed tier thresholds from Section 3.4 yields 229 Ideal trajectories (20.2%), 785 Solid trajectories (69.1%), and 122 Lucky trajectories (10.7%). Figure 1 shows this distribution. Binary evaluation assigns all 1,136 trajectories the same label, while AGENTLENS separates direct, coherent solutions from weak processes that happen to pass.

Not all non-Ideal trajectories are weak in the same way. Some have low structural overlap with the merged PTA but remain coherent and temporally well organized. We call this profile *efficient-but-atypical*: the agent follows an unconventional but valid path rather than the dominant known-good branches. This distinction matters because low structural overlap alone would make these trajectories look weak, while the full AGENTLENS score separates them from Lucky Passes. Appendix C.1 reports the full four-profile breakdown.

5.2 Lucky Passes reveal recurring weak-success mechanisms

We now analyze the 122 Lucky Passes: trajectories that produce correct patches but receive low process-quality scores. Using automatically computed AGENTLENS signals, including trajectory length, verification coverage, waste patterns, implementation coverage, and coherence, we assign each Lucky Pass to one of five mutually exclusive categories (Table 1). The two largest categories are C2, Brute-Force Convergence, where the agent reaches a correct patch through repeated low-coherence attempts, and C3, Incomplete Implementation, where the agent makes a partial fix that passes because visible tests are insufficient. Together, C2 and C3 account for 68.0% of Lucky Passes.

Table 1: **Lucky Pass taxonomy.** Five categories with discriminating signals, prevalence, and causal mechanisms. C2 and C3 together account for 68.0% of all Lucky Passes.

Category	n	%	Key Signal	Primary Cause
C1: Minimal & Unverified	19	15.6	Short + no V stage	Agent overconfidence
C2: Brute-Force Convergence	42	34.4	High waste, low coh.	Lack of planning
C3: Incomplete Implementation	41	33.6	Partial fix, low cov.	Test-suite gaps
C4: Excessive Exploration	5	4.1	Very long, unfocused	Missing termination
C5: Divergent-but-Valid	15	12.3	Alternative approach	Multiple valid solutions

The waste signals explain why these trajectories are weak despite passing. We track regression loops, blind retries, redundant steps, unnecessary exploration, and cyclic patterns as defined in Section 3.4. These signals are computed relative to the task-level PTA, so behavior already present in a known-good solution path is not counted as waste. Across pass/fail groups, unnecessary exploration and cyclic patterns are the strongest discriminators: failing trajectories are 58% more likely to inspect files outside the known-good solution space (F/P $= 1.58$), and cyclic patterns are 32% more prevalent in failing trajectories (F/P $= 1.32$). The full pass/fail waste breakdown is in Appendix C.2.

For Lucky Passes, the clearest waste pattern is not how often blind retries appear, but how much work they waste when they do appear. Lucky trajectories with blind retries waste 11.4 steps per instance, compared

with 2.7 in Ideal trajectories, a 4.2× increase. This captures a common weak-success pattern: instead of systematically debugging a failure, the agent repeats similar actions until one attempt succeeds. Appendices C.3 and C.4 provide the full tier-wise waste table and representative timelines.

The Lucky categories are not evenly distributed across models. A chi-square test over model and category assignments shows a significant association ($\chi^2(28) = 102.47$, $p < 0.0001$). Opus 4.6 disproportionately produces Minimal-and-Unverified passes, GPT-4.1 produces most Brute-Force and Excessive-Exploration cases, and the Codex variants concentrate in Incomplete Implementation. Lucky Passes are also task-concentrated: the top 10 tasks account for 63.1% of all Lucky Passes. Appendix D gives the decision tree, model and task cross-tabs, category-level waste analysis, verification-gap analysis, and extended case studies.

Model comparison. Process quality changes how models are ranked on AGENTLENS-Bench. Table 2 reports model-level aggregates over the 1,815-trajectory PTA-eligible subset, with quality scores (QS) computed for each model's successful trajectories and compared against pass-rate (PR) rankings on the same subset. The rankings disagree for every model. GPT-4o rises from 8th by pass rate to 3rd by quality score, but this should be interpreted conditionally: among the tasks it solves, its successful trajectories tend to receive higher process-quality scores. Since GPT-4o succeeds less often, this shift may partly reflect which tasks it solves rather than a model-wide advantage. Opus 4.6 shows the opposite pattern: a 77.3% pass rate hides an 18.7% Lucky rate. Across models, the share of successful trajectories classified as Lucky ranges from 0.5% to 23.2%.

Table 2: **Frontier model comparison.** Quality-score rankings (QS Rank) disagree with pass-rate rankings (PR Rank) on all eight models.

Model	Pass%	PR Rank	Quality	QS Rank	Lucky%
sonnet-4.5	86.8%	2	**67.4**	1	1.0%
opus-4.5	87.9%	1	66.2	2	0.5%
gpt-4o	34.9%	8	63.4	3	4.1%
gemini-2.5-pro	42.9%	7	59.2	4	7.6%
gpt-5.3-codex	45.9%	6	58.3	5	15.3%
opus-4.6	77.3%	3	56.7	6	18.7%
gpt-5.2-codex	64.6%	4	56.1	7	19.4%
gpt-4.1	59.9%	5	54.7	8	**23.2%**

Additional model-comparison visualizations and token-cost analysis appear in Appendices C.8 and F.

5.3 The combined score separates passing and failing trajectories

After analyzing the main process-quality findings, we validate whether the composite score also captures outcome-relevant behavioral differences beyond the pilot set used for calibration. On the scaled 47-task evaluation set, the combined score reaches AUROC (Bradley, 1997) = 0.766, accuracy = 72.0%, F1 = 0.723, and KS (Massey, 1951) $p = 0.0017$ for separating passing and failing trajectories. This experiment is a sanity check, not a replacement for pass/fail evaluation: if process quality were unrelated to outcome, the score would be difficult to interpret. The value of AGENTLENS comes from the mismatch cases. Passing trajectories with low quality scores expose Lucky Passes, while failing trajectories that remain close to known-good processes expose Partial-fail cases that may be recoverable.

The combined score is the only signal with statistically significant pass/fail separation. No individual signal reaches $p < 0.05$ (Table 3). This partial agreement with outcome is useful, but incomplete by design: structural alignment and set coverage measure how much of the known-good solution space the agent follows, while coherence and temporal profile measure whether the agent moves through the task in an orderly way. These process signals explain why trajectories with the same binary outcome can receive different quality scores.

We also compare AGENTLENS with simpler trajectory-similarity baselines. AGENTLENS outperforms TF-IDF and dense embedding alignment by 0.065–0.094 AUROC. Individual trajectory matching slightly outperforms AGENTLENS in AUROC (0.805), but it requires $O(N_{\text{train}})$ comparisons at inference and returns only a scalar similarity to one best-match trace. The merged-PTA representation is more useful for analysis because it localizes divergence, measures branch-level coverage, and attributes waste relative to known-good solution paths. Appendix C.7 reports the full baseline table, and Appendix C.6 shows the score distributions.

Table 3: **Per-signal AUROC.** No individual signal achieves $p < 0.05$; only the combined score significantly separates passing and failing trajectories.

Signal	AUROC	Weight	KS p
Structural alignment (Φ_{struct})	0.710	0.20	0.196
Set coverage (Φ_{cov})	0.718	0.15	0.603
Trajectory coherence (Φ_{coh})	0.728	0.30	0.603
Temporal profile (Φ_{temp})	0.653	0.35	0.365
Combined	**0.766**	1.00	**0.0017**

Among failing trajectories, 54.9% are Partial-fail and 45.1% are Off-track. Partial-fail trajectories remain close enough to known-good processes to exceed the failure-tier threshold, while Off-track trajectories diverge earlier or more completely. This split suggests that roughly half of failures are structurally recoverable: the agent follows a reasonable strategy but makes a localized error.

5.4 Intent-stage labels are reliable

Before using intent-stage labels for scoring, we validate them with a seven-annotator agreement study. Following the MAST protocol (Cemri et al., 2025), five human annotators, all software engineers who use coding agents in their daily work, and two LLM annotators labeled 200 deduplicated state-actions sampled across tools and phases. The labels reach Fleiss' $\kappa = 0.933$ with 96.0% raw agreement. Of the 200 states, 192 reached consensus; evaluated against these consensus labels, the AGENTLENS heuristic achieves 93.8% accuracy and macro-F1 $= 0.933$. The remaining disagreements concentrate on post-implementation `read_file` calls, the boundary case targeted by the context-sensitive labeler. Appendix C.5 gives the per-stage breakdown.

6 Robustness and Ablations

We use the disjoint 278-trajectory pilot set for controlled ablations. The goal is not to re-tune the score, but to test whether the main design choices are necessary: combining structural and behavioral signals, merging multiple passing trajectories into a PTA, and fixing a merge count for scaled evaluation. Full tables are reported in Appendix E.

Signal contribution. Removing any one signal reduces AUROC relative to the full score. The largest drops come from removing temporal profile divergence (-0.037) and trajectory coherence (-0.031), followed by set coverage (-0.024) and structural alignment (-0.016). This supports the design choice that structural signals and behavioral signals should be fused rather than treated as substitutes. Structural alignment and coverage measure whether the agent visits relevant states; coherence and temporal profile measure whether it moves through them in a plausible problem-solving order. Weight sensitivity is mild: perturbing any single weight by ± 0.05 reduces AUROC by at most 0.006.

Merge-count sensitivity. The number of passing trajectories merged into the PTA controls a precision–coverage trade-off. With small k (here, k represents the number of passing trajectories used to construct the merged PTA), the PTA is compact and precise but may penalize valid strategies absent from the reference. With larger k, the PTA covers more successful strategies but becomes more permissive and can exceed scoring limits on complex tasks. On the pilot set, $k = 2$ achieves AUROC $= 0.749$ with full task coverage, while $k = 5$ achieves AUROC $= 0.777$ and covers a broader solution space. At $k \geq 6$, AUROC rises further but only a small subset of easier-to-score task resamples remains, indicating survivorship bias rather than genuine improvement. We therefore use $k = 5$ for scaled evaluation.

Merge-order robustness. Because PTA construction is incremental, we also test whether merge order materially changes downstream scores. On one of the tasks, we evaluate 10 trajectory combinations and all 6 merge permutations for each combination. Trajectory selection explains 64.1% of the variance, while

merge ordering explains 35.9%; 8 of 10 combinations are fully order-invariant. The remaining order effects occur when an ambiguous exploration prefix can be either merged or branched, but the observed range is bounded and smaller than the effect of which trajectories are selected. Thus, the main source of PTA variation is reference-set choice, not merge ordering.

7 Conclusion

In this work, we introduced AGENTLENS, a process-aware framework for evaluating SWE-agent trajectories beyond binary pass/fail outcomes. AGENTLENS converts raw trajectories into intent-labeled state sequences and merges passing trajectories into task-level PTA references. This lets us score not only whether an agent reached a correct patch, but whether it followed a coherent and low-waste solution process. With this view, AGENTLENS distinguishes direct solutions, valid alternative paths, weak successful trajectories that pass only after brittle behavior, and failed trajectories that remain close to known-good processes.

Evaluated on 2,614 OpenHands trajectories from 60 SWE-bench Verified tasks, AGENTLENS reveals substantial variation among successful runs. On the 47 PTA-eligible tasks that form AGENTLENS-Bench, 10.7% of passing trajectories are Lucky Passes: correct outcomes reached through weak processes. These weak successes follow recurring patterns rather than isolated accidents, with Brute-Force Convergence and Incomplete Implementation accounting for 68.0% of Lucky Passes. Process-aware scoring also changes model comparison: across all eight evaluated model backends, quality-score rankings differ from pass-rate rankings, and Lucky rates range from 0.5% to 23.2%. We plan to release AGENTLENS-Bench, the AGENTLENS SDK, and the web interface soon, with the goal of making process-aware analysis a standard layer in coding-agent evaluation.

References

A. Alaboudi and T. D. LaToza. An exploratory study of debugging episodes, 2021.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *arXiv preprint arXiv:2505.20411*, 2025.

Islem Bouzenia and Michael Pradel. Understanding software engineering agents: A study of thought-action-result trajectories. *arXiv preprint arXiv:2506.18824*, 2025.

Andrew P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30(7):1145–1159, 1997. ISSN 0031-3203. doi: https://doi.org/10.1016/S0031-3203(96)00142-2. URL https://www.sciencedirect.com/science/article/pii/S0031320396001422.

M. Brunsfeld et al. tree-sitter/tree-sitter: v0.23.0, 2024.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. Why do multi-agent llm systems fail? *arXiv preprint arXiv:2503.13657*, 2025.

Hyungjoo Chae, Sunghwan Kim, Junhee Cho, Seungone Kim, Seungjun Moon, Gyeom Hwangbo, Dongha Lim, Minjin Kim, Yeonjun Hwang, Minju Gwak, et al. Web-shepherd: Advancing prms for reinforcing web agents. *arXiv preprint arXiv:2505.15277*, 2025.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, et al. Introducing swe-bench verified. *arXiv preprint arXiv:2407.01489*, 2024.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Darshan Deshpande, Varun Gangal, Hersh Mehta, Jitin Krishnan, Anand Kannappan, and Rebecca Qian. Trail: Trace reasoning and agentic issue localization. *arXiv preprint arXiv:2505.08638*, 2025.

Tom Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

Harbor Framework. Agent trajectory format (atif). https://www.harborframework.com/docs/agents/trajectory-format, 2026. Accessed 2026-05-05.

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

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.

C. E. Jimenez et al. SWE-bench: Can language models resolve real-world GitHub issues?, 2024.

Amy J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering*, 32(12):971–987, 2006.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The twelfth international conference on learning representations*, 2023.

J. Lin. Divergence measures based on the Shannon entropy. *IEEE Transactions on Information Theory*, 37(1): 145–151, 1991.

Shuyang Liu, Yang Chen, Rahul Krishna, Saurabh Sinha, Jatin Ganhotra, and Reyhaneh Jabbarvand. Process-centric analysis of agentic software systems. *Proceedings of the ACM on Programming Languages*, 10 (OOPSLA1):1961–1988, 2026.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.

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. *Advances in neural information processing systems*, 37:74325–74362, 2024.

Oorja Majgaonkar, Zhiwei Fei, Xiang Li, Federica Sarro, and He Ye. Understanding code agent behaviour: An empirical study of success and failure trajectories. *arXiv preprint arXiv:2511.00197*, 2025.

Frank J. Massey. The kolmogorov-smirnov test for goodness of fit. *Journal of the American Statistical Association*, 46(253):68–78, 1951. ISSN 01621459, 1537274X. URL http://www.jstor.org/stable/2280095.

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

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

José Oncina, Pedro Garcia, et al. Inferring regular languages in polynomial update time. *Pattern recognition and image analysis*, 1(49-61):10–1142, 1992.

J. Pan et al. Training software engineering agents and verifiers with SWE-Gym, 2025.

Sakshi Kini Sana Ansari. The world's first ai software engineer, devin ai. *International Journal of Innovative Research in Technology*, 11(1):1813–1816, June 2024. ISSN 2349-6002. URL https://ijirt.org/article?manuscript=165794.

KaShun Shum, Binyuan Hui, Jiawei Chen, Lei Zhang, Jiaxi Yang, Yuzhen Huang, Junyang Lin, Junxian He, et al. Swe-rm: Execution-free feedback for software engineering agents. *arXiv preprint arXiv:2512.21919*, 2025.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process-and outcome-based feedback. *arXiv preprint arXiv:2211.14275*, 2022.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9426–9439, 2024a.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024b.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*, 2025.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

T. Xie et al. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments, 2024.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.

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, 2025. URL https://arxiv.org/abs/2504.21798.

W. J. Youden. Index for rating diagnostic tests. *Cancer*, 3(1):32–35, 1950. doi: https://doi.org/10.1002/1097-0142(1950)3:1<32::AID-CNCR2820030106>3.0.CO;2-3. URL https://acsjournals.onlinelibrary.wiley.com/doi/abs/10.1002/1097-0142%281950%293%3A1%3C32%3A%3AAID-CNCR2820030106%3E3.0.CO%3B2-3.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, et al. Multi-swe-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605*, 2025.

Linghao Zhang et al. SWE-bench Goes Live!, 2025. URL https://arxiv.org/abs/2505.23419.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert, 2020. URL https://arxiv.org/abs/1904.09675.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, pages 1592–1604, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706127. doi: 10.1145/3650212.3680384. URL https://doi.org/10.1145/3650212.3680384.

Chujie Zheng, Zhenru Zhang, Beichen Zhang, Runji Lin, Keming Lu, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. ProcessBench: Identifying process errors in mathematical reasoning. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1009–1024, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.50. URL https://aclanthology.org/2025.acl-long.50/.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023. URL https://webarena.dev.

Yuxuan Zhu, Tengjun Jin, Yada Pruksachatkun, Andy Zhang, Shu Liu, Sasha Cui, Sayash Kapoor, Shayne Longpre, Kevin Meng, Rebecca Weiss, Fazl Barez, Rahul Gupta, Jwala Dhamala, Jacob Merizian, Mario Giulianelli, Harry Coppock, Cozmin Ududec, Jasjeet Sekhon, Jacob Steinhardt, Antony Kellermann, Sarah Schwettmann, Matei Zaharia, Ion Stoica, Percy Liang, and Daniel Kang. Establishing best practices for building rigorous agentic benchmarks, 2025. URL https://arxiv.org/abs/2507.02825.

Mingchen Zhuge, Changsheng Zhao, Dylan 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, 2024. URL https://arxiv.org/abs/2410.10934.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

Appendix

Table of Contents

A Limitations, Future Directions, and Positioning . . . 15
A.1 Limitations . . . 15
A.2 Future Directions . . . 15
A.3 Broader Impacts . . . 15
A.4 Extended Related Work and Positioning . . . 15
B Additional Method Details . . . 15
B.1 Tool Registry . . . 16
B.2 State Equivalence Engine . . . 16
B.3 Intent-stage Labeling Flow . . . 17
B.4 Scoring Details . . . 17
B.5 Scoring Signal Examples . . . 19
C Additional Experimental Results . . . 19
C.1 Behavioral Profiles . . . 20
C.2 Pass/Fail Waste Breakdown . . . 21
C.3 Ideal-versus-Lucky Waste . . . 21
C.4 Failure-mode Gallery . . . 21
C.5 Heuristic Labeler Validation . . . 22
C.6 Score Distributions . . . 22
C.7 Baseline Results . . . 23
C.8 Model-comparison Visualizations . . . 24
D Extended Lucky Pass Analysis . . . 24
D.1 Signal Landscape and Taxonomy Design . . . 24
D.2 Category Distribution . . . 26
D.3 Waste as a Category Differentiator . . . 26
D.4 Model-specific Lucky Pass Signatures . . . 27
D.5 Task-level Concentration . . . 27
D.6 Case Study: One Task, Five Behavioral Profiles . . . 28
D.7 Additional Lucky Pass Cases . . . 28
D.8 Verification Gap . . . 30
E Ablation Details . . . 30
E.1 Signal Contribution . . . 31
E.2 Merge-count Sensitivity . . . 31
E.3 Merge-order Robustness . . . 32
F Token Cost and Statistical Tests . . . 32
F.1 Statistical Tests for Lucky Pass Taxonomy . . . 33

A Limitations, Future Directions, and Positioning

A.1 Limitations

AGENTLENS is designed as a post-hoc analysis framework for SWE-agent trajectories, and our evaluation focuses on OpenHands-style coding-agent traces on SWE-bench Verified tasks. This scope lets us study process quality in a controlled setting with task-level PTA references and reproducible scoring. Extending the same analysis to other agent scaffolds primarily requires trace-format adapters or ATIF conversion, rather than changes to the core scoring pipeline. Similarly, the fixed score weights used in this paper are chosen to keep the benchmark deterministic and comparable across models; future deployments may tune these weights for domain-specific priorities such as verification discipline, exploration cost, or repair efficiency.

A.2 Future Directions

AGENTLENS opens several directions for process-aware agent research. First, quality scores can be used as dense reward signals for reinforcement learning, encouraging agents not only to reach correct patches but to follow coherent, low-waste solution processes (Wei et al., 2025). Second, PTA references can support curriculum construction: training pools can be organized by process quality, divergence type, or recoverability rather than by final outcome alone. Third, longitudinal process-quality tracking can compare model versions over time, revealing whether higher pass rates come from cleaner reasoning, broader exploration, or increased retry behavior. Finally, the same trajectory-analysis view can extend beyond software repair to web navigation (Zhou et al., 2023) and computer-use agents (Xie et al., 2024), where success alone often hides large differences in action efficiency and recoverability.

A.3 Broader Impacts

AGENTLENS is intended to improve the evaluation and analysis of software engineering agents. By exposing brittle successful trajectories, recoverable failures, and wasteful solution processes, it can help practitioners make safer deployment decisions, select cleaner training demonstrations, and diagnose weaknesses that are hidden by pass/fail benchmarks. This can improve benchmark incentives by rewarding agents that solve tasks coherently rather than through excessive retry or accidental success.

At the same time, process scores should be treated as complementary diagnostics rather than replacements for functional correctness, security review, or human judgment. A trajectory that receives a high AGENTLENS score may still contain an incorrect or insecure patch if the underlying tests or references are incomplete. We therefore position AGENTLENS as an additional layer for responsible evaluation, not as an automatic approval mechanism for deploying code changes.

A.4 Extended Related Work and Positioning

Table 4 compares AGENTLENS-Bench against related SWE-agent trajectory collections. Table 5 compares AGENTLENS against related evaluation frameworks.

B Additional Method Details

Section 3 describes the AGENTLENS pipeline at the level needed to follow the main argument. This appendix provides the reproducibility-level detail: the full tool registry that drives first-pass label assignment, the state equivalence engine that allows PTA construction to work across heterogeneous agent tool sets, the intent-stage classification decision tree, the formal definitions and worked examples for all four scoring signals, and the planned web interface through which practitioners will be able to inspect the annotations.

The material is organized to mirror the pipeline order. We begin with the tool registry (B.1), which provides the raw mapping from agent tool calls to default intent-stage hints. The equivalence engine (B.2) explains how states from different agents are recognized as covering the same ground-truth action during PTA merging.

Table 4: AGENTLENS-Bench versus related trajectory collections. Dashes indicate the feature is absent; "Partial" indicates a related but incomplete feature.

Feature	SWE-Gym	R2E-Gym	OpenHands	Graphectory	AGENTLENS-Bench
Trajectories	∼2.4K	∼10K	∼300	∼100	**1,815**
Models covered	1	1	5	1	**8**
Pass/fail labels	✓	✓	✓	✓	✓
Quality score (0–100)	—	—	—	—	✓
Multi-dim. metrics	—	—	—	Partial	✓
Ground-truth PTAs	—	—	—	—	✓
Waste annotations	—	—	—	—	✓
Divergence localization	—	—	—	—	✓
Quality tier labels	—	—	—	—	✓
Curation support	Outcome	Outcome	—	—	✓
No LLM calls needed	—	—	—	✓	✓

Table 5: Framework comparison. AGENTLENS is the only system that combines SWE-specific trajectory analysis, multi-dimensional scoring, validated phase labels, and structured inefficiency attribution.

	Domain	Granularity	Multi-dim.	Inefficiency	Label κ
AgentBoard	Multi (9)	Sub-goal	✓	—	—
Graphectory	SWE	Trace	✓	—	—
MAST	Multi-agent	Trace	✓	—	0.88
TRAIL	Multi (3)	Step	✓	Partial	—
Web-Shepherd	Web	Step	scalar	—	—
SWE-RM	SWE	Trajectory	scalar	—	—
ABC	SWE	Task	checklist	—	—
AGENTLENS	**SWE**	**Trajectory**	✓	✓	0.933

The intent-stage labeling flow (B.3) gives the full seven-rule priority cascade, including the context-sensitive rules that resolve terminal-command ambiguity. The scoring details (B.4) present the formal definitions, equations, and design rationale for all four signals and the combined score. Finally, worked examples of all four scoring signals are provided in Appendix B.5.

B.1 Tool Registry

The tool registry provides the first-pass mapping from raw tool calls to intent-stage hints and comparison strategies. These hints are intentionally treated as defaults rather than final labels: the context-sensitive labeler can override them using trajectory history, edited-file state, and command semantics. Table 6 shows the abbreviated registry used in the experiments.

Table 6: Tool registry (abbreviated). Each tool maps to a category, a default stage hint, and a comparison strategy.

Category	Examples	Default Stage	Comparison
read	read_file, view_file	E / V (context)	file path
edit	replace_string_in_file	I / V (context)	file path
search	grep_search, semantic_search	E	query
execute	run_in_terminal	E / V (context)	command
validation	get_errors, test_failure	V	identity

B.2 State Equivalence Engine

The equivalence engine determines whether two states from different agents represent the same action. Different agents use different tool names (`grep` versus `rg`), read overlapping but not identical file regions, and format terminal arguments differently. The engine applies a confidence-weighted cascade in priority order:

1. **Exact content hash** (confidence 1.0): states with identical MD5 hashes are equivalent.

2. **File-scope matching** via tree-sitter (Brunsfeld et al., 2024) (confidence 0.90): states targeting the same AST-level scope (function, class, module) are equivalent.
3. **Line-range overlap** of at least 30% (confidence 0.80–0.95): states reading or editing overlapping regions of the same file are equivalent, with confidence scaled by overlap fraction.
4. **Semantic terminal grouping** (confidence 0.70–0.85): terminal commands in the same functional group (e.g., `grep`, `rg`, `ag` are all "search" commands) with Jaccard token similarity above 0.5 are treated as equivalent.

An optional LLM fallback, which queries a language model for ambiguous cases, was disabled throughout for reproducibility. The cascade is evaluated in order; the first match determines the equivalence decision and its confidence.

B.3 Intent-stage Labeling Flow

The intent-stage labeler is the component that most directly affects the behavioral signals. If labels are wrong, coherence and temporal profile become unreliable. The main challenge is terminal commands: `grep`, `cat`, and `ls` can serve exploratory or verificatory purposes depending on where they appear in the trajectory. A naive tool-identity approach labels all terminal commands as a single stage, which causes 70–80% of PTA states to collapse into one category and renders the temporal signal uninformative.

Figure 4 shows the full seven-rule priority cascade. Rules 1 through 4 assign fixed stages based on tool type (search tools receive E, validation tools receive V, source edits receive I, orchestration tools receive O). Rules 5 through 7 are context-sensitive: the same tool can map to different stages depending on whether a source-file edit has already occurred, whether the target file is a test file, or whether it was previously modified. The terminal-command sub-tree (Rule 7) disambiguates five command categories by pattern matching. The critical design decision is Rule 5: file-inspection commands (`grep`, `cat`, `ls`, `git log`, `find`) receive Exploration regardless of when they appear, because these commands read information from the codebase and do not verify the correctness of any prior edit.

B.4 Scoring Details

Given a candidate trace τ_c and a ground-truth PTA $\mathcal{G}$, AGENTLENS computes four signals.

Structural alignment (Φ_{struct}). The candidate's state sequence is aligned against the best-matching PTA path via greedy forward scan (ordered recall) and maximum bipartite matching (unordered precision). Their harmonic mean is the structural F1. This signal measures whether the agent visited the right states in roughly the right order.

Trajectory coherence (Φ_{coh}). The intent-stage sequence is compressed into a workflow fingerprint. Consecutive stage pairs are classified as pivots (forward progress, e.g. E→I), backtracks (regression, e.g. V→E), deepenings (same phase continues), or confirmations (transition to O). The coherence score combines a forward-progress ratio with a blind-retry penalty:

$$\Phi_{\text{coh}}(\tau) = \frac{|\text{pivots}| + |\text{confirms}|}{|\text{pivots}| + |\text{confirms}| + |\text{backtracks}| + \epsilon} \times \left(1 - \frac{r}{|T|}\right), \tag{2}$$

where r is the blind-retry count and $|T|$ is the total transition count. A clean E→E→I→V trajectory scores 1.0; a trajectory with three regression cycles and a blind-retry cluster scores about 0.51. A worked example of the coherence calculation is provided in Appendix B.5.

Temporal profile divergence (Φ_{temp}). The trajectory is divided into three equal segments. In each segment the stage distribution is computed with Laplace smoothing ($\alpha = 0.01$) and compared against the PTA

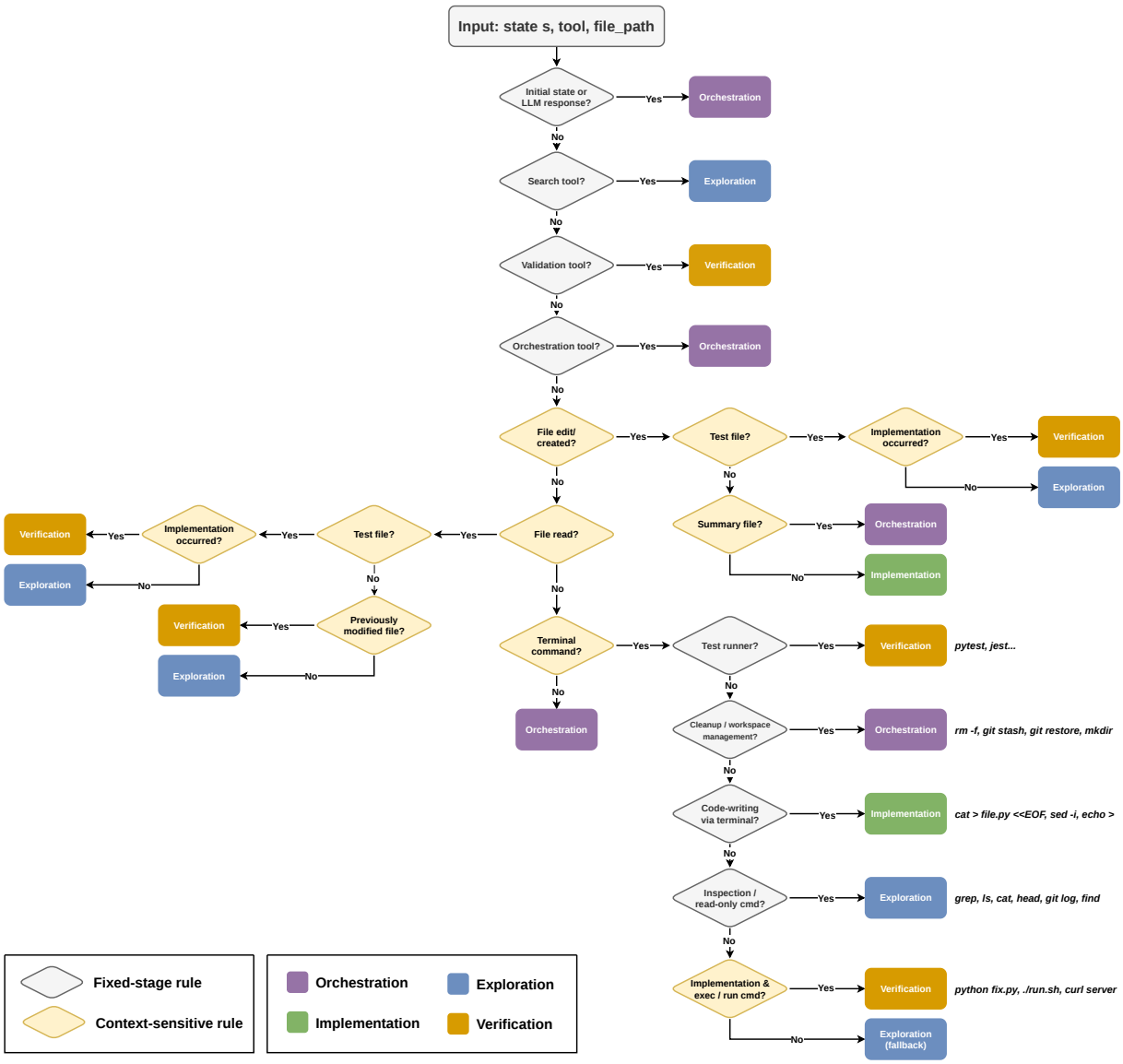

Figure 4: **Intent-stage classification decision tree.** Each agent action is classified into one of four intent stages (Exploration, Implementation, Verification, Orchestration) via a priority cascade of seven rules. Rules 1 to 4 (gray diamonds) assign fixed stages based on tool type. Rules 5 to 7 (yellow diamonds) are context-sensitive: the same tool can map to different stages depending on whether a source-file edit has already occurred, whether the target file is a test file, or whether it was previously modified. The terminal-command sub-tree (Rule 7) further disambiguates five command categories by pattern matching.

distribution via Jensen-Shannon divergence (Lin, 1991):

$$\Phi_{\text{temp}}(\tau_c, \mathcal{G}) = 1 - \frac{1}{3}\sum_{k=1}^{3} \text{JSD}\left(P_{\tau_c}^{(k)} \parallel P_{\mathcal{G}}^{(k)}\right). \tag{3}$$

This measures whether cognitive phases occurred in the expected temporal order.

Set coverage (Φ_{cov}). The fraction of PTA states across all paths matched by any state in the candidate trajectory, computed via maximum bipartite matching without ordering constraints. This complements the ordered structural F1.

Structured inefficiency analysis. Five categories of behavioral waste are detected with step-level localization: regression loops, blind retries, redundant steps, unnecessary exploration, and cyclic patterns. Each instance receives per-tool attribution, token-waste estimation, and a severity score. Their definitions and discriminative power are reported in Appendix C.2.

Combined score. The four signals are combined with weights optimized via grid search (step 0.05, unit-sum constraint, AUROC-maximizing on the pilot calibration set):

$$f(\tau_c, \mathcal{G}) = 0.20 \cdot \Phi_{\text{struct}} + 0.15 \cdot \Phi_{\text{cov}} + 0.30 \cdot (100 \cdot \Phi_{\text{coh}}) + 0.35 \cdot (100 \cdot \Phi_{\text{temp}}). \quad (4)$$

Φ_{struct} and Φ_{cov} are natively on a 0–100 scale; Φ_{coh} and Φ_{temp} are on $[0, 1]$ and are rescaled by 100 so that all four terms contribute on a common scale. Behavioral signals carry 65% of the weight. This is an empirical result of the grid search rather than a design choice. It reflects the fact that structural coverage and behavioral process quality fail on different trajectories: a failing agent that touches 70% of ground-truth states through chaotic trial-and-error can be structurally similar to a passing agent with comparable coverage but more principled reasoning. The behavioral signals catch the difference.

B.5 Scoring Signal Examples

Figure 5 illustrates all four scoring signals on paired examples, contrasting a principled trajectory against a chaotic one for each dimension.

Coherence (panel a). The principled trajectory follows a clean E→E→I→V sequence with 2 pivots, 2 deepenings, 0 backtracks, and 0 retries, yielding $\Phi_{\text{coh}} = 1.00$. The chaotic trajectory contains backtracks (V→E, I→E) and a cluster of 4 identical edits flagged as blind retries, producing 6 pivots, 4 deepenings, 2 backtracks, and 4 retries. The retry penalty and the reduced forward-progress ratio together bring the score to $\Phi_{\text{coh}} = 0.51$.

Structural alignment (panel b). The high-alignment candidate matches 6 of 6 PTA states in order (ordered recall = 1.00) with 6 of 8 candidate states matching (precision = 0.75), giving F1 = 0.86. The low-alignment candidate matches only 2 of 6 PTA states in order and 2 of 9 total, giving F1 = 0.27. The gap reflects the difference between a candidate that follows the PTA path and one that takes a substantially different route through the solution space.

Set coverage (panel c). Set coverage ignores ordering and asks how many PTA states the candidate touches at all. The high-coverage candidate matches 12 of 15 PTA states ($\Phi_{\text{cov}} = 0.80$). The low-coverage candidate matches only 3 of 15 ($\Phi_{\text{cov}} = 0.20$). A candidate can have high coverage but low structural alignment if it visits the right states in a scrambled order, which is why both signals are needed.

Temporal profile divergence (panel d). Each trajectory is divided into three equal segments (early, middle, late), and the stage distribution in each segment is compared against the PTA via Jensen-Shannon divergence. The principled trajectory has low JSD in all three segments (0.02, 0.03, 0.01), yielding $\Phi_{\text{temp}} = 0.92$. The chaotic trajectory has high JSD because verification dominates the early segment (where exploration should dominate) and exploration dominates the late segment (where verification should dominate), yielding $\Phi_{\text{temp}} = 0.38$. This signal catches temporal disorder that coherence alone would miss: an agent can have reasonable forward progress but do everything in the wrong order.

C Additional Experimental Results

The main text reports the central experimental findings: quality stratification among passing trajectories (Section 5.1), the Lucky Pass taxonomy and model comparison (Section 5.2), scoring validity (Section 5.3), and intent-label reliability (Section 5.4). This appendix provides the supporting tables, figures, and breakdowns that underpin those findings. We begin with the four behavioral profiles that explain why the combined score is more informative than any single signal (C.1). We then report the full pass/fail waste breakdown (C.2) and

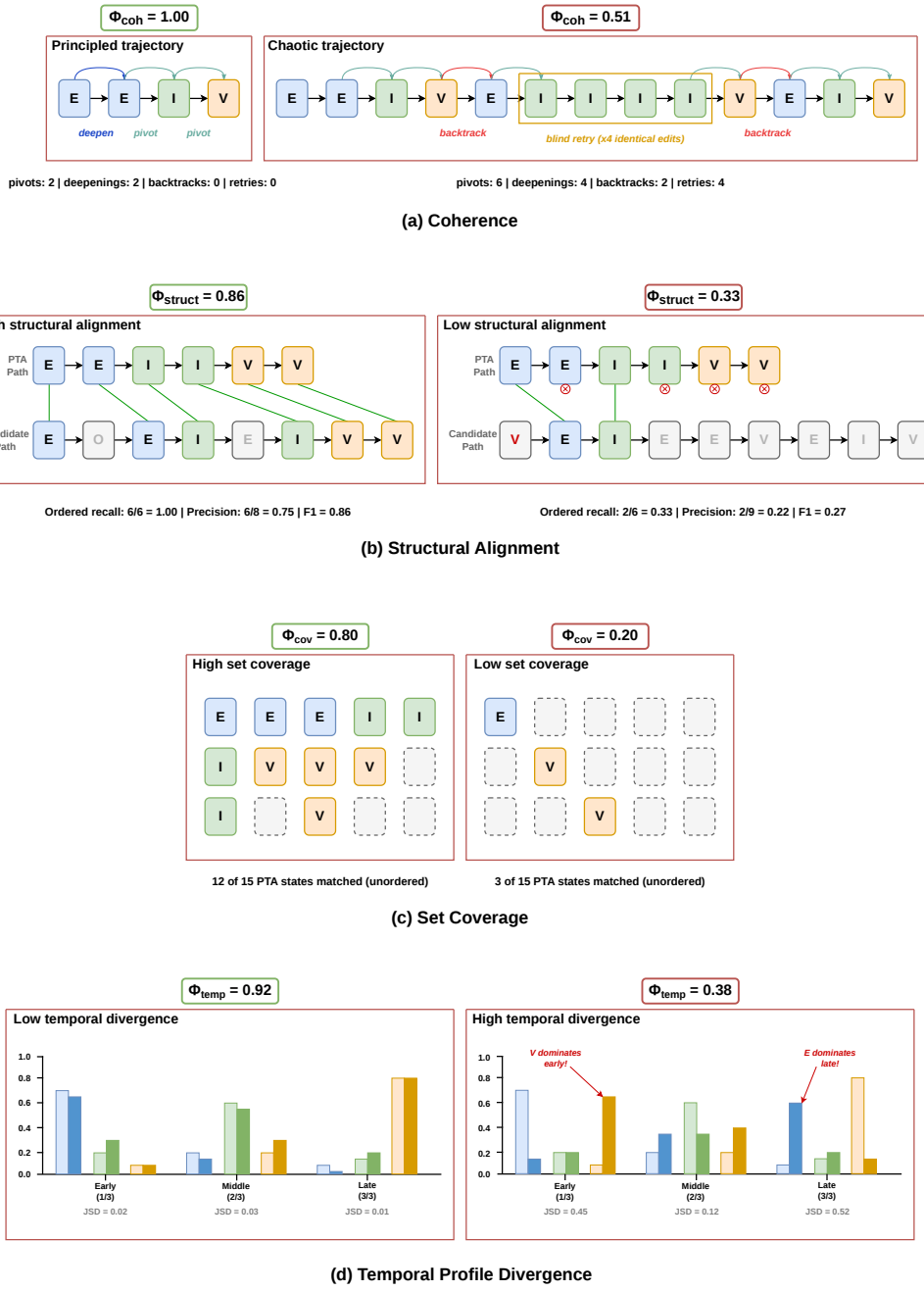

Figure 5: **Scoring signal examples.** Each panel contrasts a principled trajectory (left) against a chaotic trajectory (right) on one of the four scoring dimensions: (a) coherence, (b) structural alignment, (c) set coverage, and (d) temporal profile divergence. Together, the four signals capture complementary aspects of process quality.

the Ideal-versus-Lucky waste comparison that identifies the blind-retry severity fingerprint (C.3). Representative failure-mode timelines (C.4) and the per-stage labeler validation (C.5) follow. Score distributions (C.6), baseline comparison (C.7), and model-comparison visualizations (C.8) complete the experimental detail.

C.1 Behavioral Profiles

The three quality tiers (Ideal, Solid, Lucky) describe aggregate outcome categories. Within these tiers, trajectories cluster into four behavioral profiles that reveal why the combined score is more useful than any

single signal. Table 7 reports the signal ranges and prevalence for each profile.

The most instructive profile is *efficient-but-atypical*, which accounts for roughly 53% of passing trajectories. These agents follow an unconventional but internally coherent solution path: high coherence and good temporal alignment combined with low structural F1 relative to the PTA. A single-signal evaluation based on structural alignment alone would flag these as weak. The combined score correctly identifies them as Solid rather than Lucky, because the behavioral signals are high even though the structural signals are low. This is the practical payoff of multi-signal fusion: it prevents principled-but-unconventional trajectories from being confused with chaotic ones.

Table 7: Behavioral profiles among passing trajectories.

Profile	Struct	Coh.	Temp.	Score	Prev.
Principled solver	.80–.95	.85–1.0	.80–.95	75–95	∼18%
Efficient, atypical	.30–.55	.80–1.0	.75–.90	55–72	∼53%
Exploratory, correct	.55–.75	.35–.55	.55–.75	45–62	∼19%
Lucky pass	.25–.45	.30–.50	.35–.55	28–48	∼10%

C.2 Pass/Fail Waste Breakdown

Table 8 provides the full pass/fail waste breakdown. Waste figures are mean steps wasted per trajectory that contains at least one instance of the category. All waste detections are ground-truth-aware: patterns already present in the merged PTA are excluded, so the numbers reflect genuinely unnecessary behavior rather than valid exploration strategies that happen to differ from one reference path.

The prevalence figures deserve careful reading. Regression loops, blind retries, and redundant steps are all slightly more common in passing trajectories than failing ones ($F/P < 1.0$). This is a length effect rather than a detection failure: Ideal-tier passing trajectories are longer and more thorough than most failing ones, exploring more of the solution space and therefore generating more GT-excluded detections in absolute terms. The discriminating signal lies in two categories: unnecessary exploration ($F/P = 1.58$), where failing trajectories are 58% more likely to inspect files outside the known-good solution space, and cyclic patterns ($F/P = 1.32$), which cost 7.8 steps per instance in failing runs versus 4.6 in passing runs.

Table 8: Waste prevalence and per-instance step cost: passing (P) versus failing (F). F/P is the ratio of failing to passing prevalence; values below 1 indicate the category is more prevalent among passing trajectories.

Type	Prev. P	Prev. F	F/P	Waste P	Waste F
Regression loops	38.7%	39.5%	1.02	15.5	12.2
Blind retries	46.0%	44.9%	0.98	5.6	7.7
Redundant steps	50.7%	47.6%	0.94	3.7	5.3
Unnecessary exploration	6.1%	9.6%	1.58	2.0	1.9
Cyclic patterns	33.6%	44.5%	1.32	4.6	7.8

C.3 Ideal-versus-Lucky Waste

Table 9 reports waste by quality tier among passing trajectories. The key Lucky-pass signal is blind-retry severity: Lucky trajectories have slightly lower blind-retry prevalence, but when retries occur, they waste substantially more steps.

C.4 Failure-mode Gallery

Figure 6 gives representative timelines for the waste patterns detected by the pipeline. These examples are illustrative; the quantitative claims in the main text come from the full 1,815-trajectory evaluation set.

Table 9: Waste by quality tier among passing trajectories (Ideal $n = 229$, Lucky $n = 122$). L/I below 1 indicates lower prevalence in Lucky.

Type	Prev. Ideal	Prev. Lucky	L/I	Waste Ideal	Waste Lucky
Regression loops	47.6%	18.0%	0.38	18.6	7.1
Blind retries	44.5%	39.3%	0.88	2.7	**11.4**
Redundant steps	57.2%	24.6%	0.43	2.9	4.3
Unnecessary exploration	6.6%	0.8%	0.13	2.0	1.0
Cyclic patterns	41.5%	15.6%	0.38	3.9	3.4

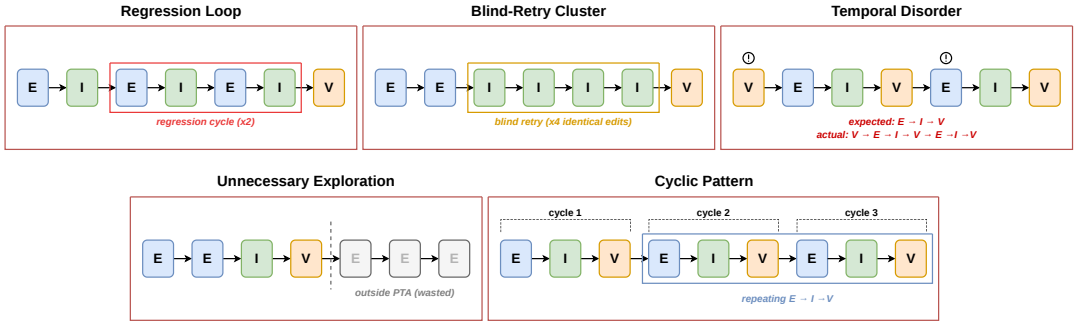

Figure 6: **Failure-mode gallery.** Six annotated stage-colored timelines: regression loop, blind-retry cluster, temporal disorder, E/V confusion before and after the context-sensitive fix, unnecessary exploration, and a cyclic pattern.

C.5 Heuristic Labeler Validation

The main text reports aggregate labeler reliability (Section 5.4): Fleiss' $\kappa = 0.933$ with 96.0% raw agreement across seven annotators. This subsection provides the per-stage breakdown that reveals where the labeler succeeds and where it struggles.

Table 10 gives inter-annotator agreement by category. The highest agreement is on Orchestration ($\kappa = 1.000$), which is expected since orchestration actions (thinking steps, bookkeeping) are unambiguous. The E-versus-V distinction on terminal commands, the hardest boundary, still reaches $\kappa = 0.939$. Implementation has the lowest κ (0.713) but the highest raw agreement (>99%), a statistical artifact of the small support (only 8 implementation states in the 200-state sample).

Table 11 evaluates the AGENTLENS deterministic heuristic against annotator consensus. The heuristic achieves 93.8% accuracy and macro-F1 = 0.933. The eight E/V disagreements concentrate on post-implementation `read_file` calls, the exact boundary case that the context-sensitive labeler targets. In these cases, the annotators labeled the read as Verification (the agent is checking its edit), while the heuristic labeled it as Exploration (the agent is reading a file). The distinction is genuinely ambiguous, and the heuristic's conservative choice (Exploration) prevents over-counting verification coverage.

Table 10: Inter-annotator agreement on intent-stage labels (200 states, 7 annotators).

Category	κ	Agreement
Overall (E/I/V/O)	**0.933**	96.0% (192/200)
E vs. V (terminal commands)	0.939	97.6%
Implementation (I)	0.713	>99%
Orchestration (O)	1.000	100%

C.6 Score Distributions

Figure 7 shows the pass/fail score distributions on the full 1,815-trajectory evaluation set. The Youden-J threshold (46.4) was computed on the pilot set and held fixed for all scaled experiments. The passing distribution concentrates in the 50–75 range with a tail extending to 95, while the failing distribution peaks around 30–40 and drops sharply above the threshold. The overlap region (roughly 35–55) contains both Solid passing trajectories and Partial-fail trajectories, which is expected: these are cases where the agent followed a

Table 11: Heuristic classifier versus annotator consensus (192/200 states with $\geq$ 67% agreement across 7 annotators: 2 LLMs, 5 humans).

Stage	Precision	Recall	F1	Support
Exploration (E)	0.980	0.926	0.952	108
Implementation (I)	0.889	1.000	0.941	8
Verification (V)	0.857	0.982	0.915	55
Orchestration (O)	1.000	0.857	0.923	21
Overall		93.8% acc	**0.933** macro-F1	192

reasonable strategy but either succeeded or failed on a localized implementation step. The KS test confirms that the two distributions are distinct ($p = 0.0017$).

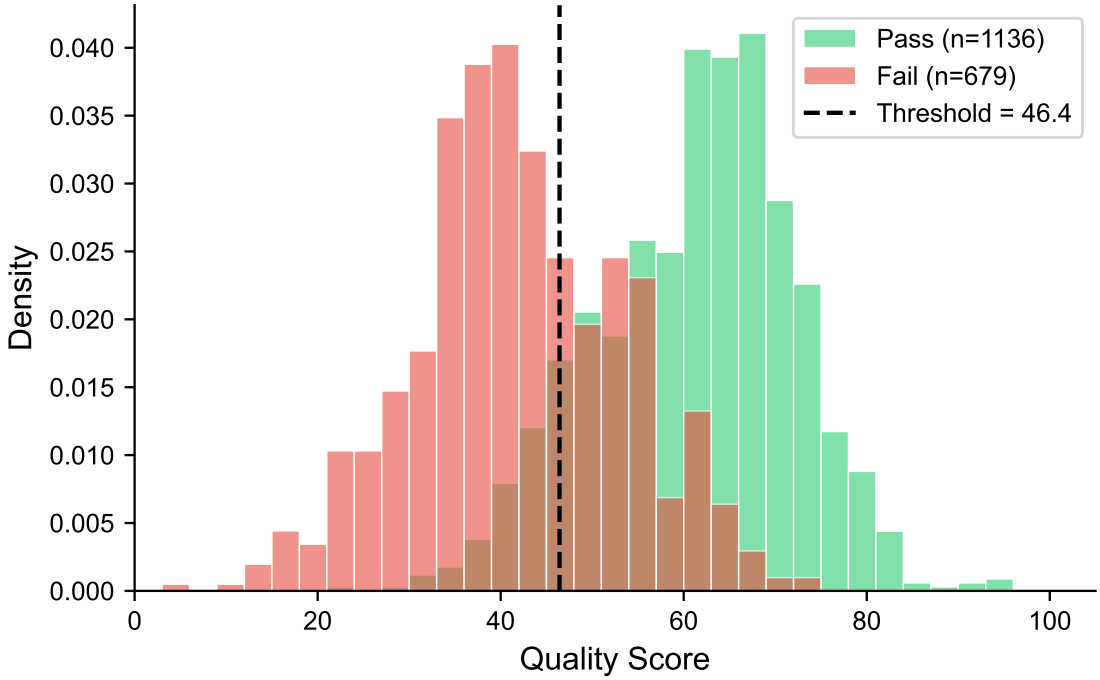

Figure 7: **Score Density by Outcome.** Overlapping density histograms of quality scores for Pass (n=1,136) and Fail (n=679) instances. The vertical dashed line marks the empirically chosen threshold at 46.4. Pass instances concentrate in the upper range while fail instances skew left. This confirms the score's ability to separate outcomes.

C.7 Baseline Results

Table 12 compares AGENTLENS against three baselines. Individual matching achieves the highest AUROC, but it does not produce structured diagnostics. The PTA is designed for interpretable, actionable output rather than maximum AUROC alone.

Table 12: Baseline comparison on the full 1,815-trajectory evaluation set.

Method	AUROC	Acc.	F1	KS p	API?
AGENTLENS (PTA)	**0.766**	0.720	0.723	0.0017	No
Individual match.	0.805	**0.744**	**0.751**	0.0031	No
TF-IDF align.	0.672	0.651	0.639	0.182	No
Dense embed.	0.701	0.667	0.658	0.094	Yes

Branch-localization example. On `astropy__astropy-13236`, the merged PTA branches at state 8 where two valid solution families fork. A candidate that edits a third file outside both branches receives only a scalar

similarity score from individual matching. AGENTLENS instead localizes the divergence to step 8, identifies the edit site as outside both PTA branches, and flags wasted exploration. This branch-aware diagnostic output is what makes the curation, model comparison, and waste reports possible.

C.8 Model-comparison Visualizations

The main text reports model-level aggregates in Table 2 (Section 5.2). The two figures below visualize the same data to make the rank disagreements and Lucky-rate spread easier to see at a glance.

Figure 8 plots pass rate against mean quality score for each of the eight model configurations. If pass rate and quality were perfectly correlated, all points would lie on a monotonic curve. Instead, the scatter shows substantial disagreement: GPT-4o sits in the lower-left quadrant by pass rate but the upper-left by quality, while Opus 4.6 sits in the upper-right by pass rate but mid-right by quality. The color coding highlights the magnitude of rank divergence.

Figure 9 shows the Lucky rate per model, which is the percentage of each model's passing trajectories that fall into the Lucky tier. The 46× range from Opus 4.5 (0.5%) to GPT-4.1 (23.2%) is the single largest behavioral gap between models and is completely invisible to pass-rate evaluation.

D Extended Lucky Pass Analysis

Section 5.2 in the main text presents the Lucky Pass taxonomy, model-specific signatures, and the three-models-one-task case study. This appendix provides the full supporting analysis: the signal landscape that guided taxonomy design (D.1), the category distribution with extended descriptions (D.2), waste as a category differentiator (D.3), the complete model cross-tabulation (D.4), task-level concentration patterns (D.5), a case study showing five behavioral profiles on a single task (D.6), detailed case studies for each category (D.7), and the verification-gap analysis (D.8).

All categories are assigned automatically by the AGENTLENS pipeline using the structural quality signals described in Appendix B.4. No manual category labels are used. The decision tree is fully deterministic and reproducible: given the same trajectory and PTA, the same category assignment will always result.

D.1 Signal Landscape and Taxonomy Design

Before defining categories, we profiled the signal distribution across all 122 Lucky-tier trajectories. Two signals are near-universal: early divergence from the ground-truth path (99.2%) and missing verification stages (94.3%). Because these are baseline conditions shared by almost every Lucky Pass, they cannot differentiate sub-categories. The discriminating signals are trajectory length, waste patterns, implementation completeness, and coherence.

We use a priority-ordered decision tree that assigns each trajectory to exactly one of five mutually exclusive categories. The tree checks, in order: whether the trajectory is short with zero waste and no verification (C1), whether it contains blind retries, cyclic patterns, regression loops, or thrashing (C2), whether it is long with excessive exploration (C4), whether it carries an incomplete-implementation failure reason (C3), and assigns the remainder to C5. All criteria are defined over structural quality signals computed by AGENTLENS.

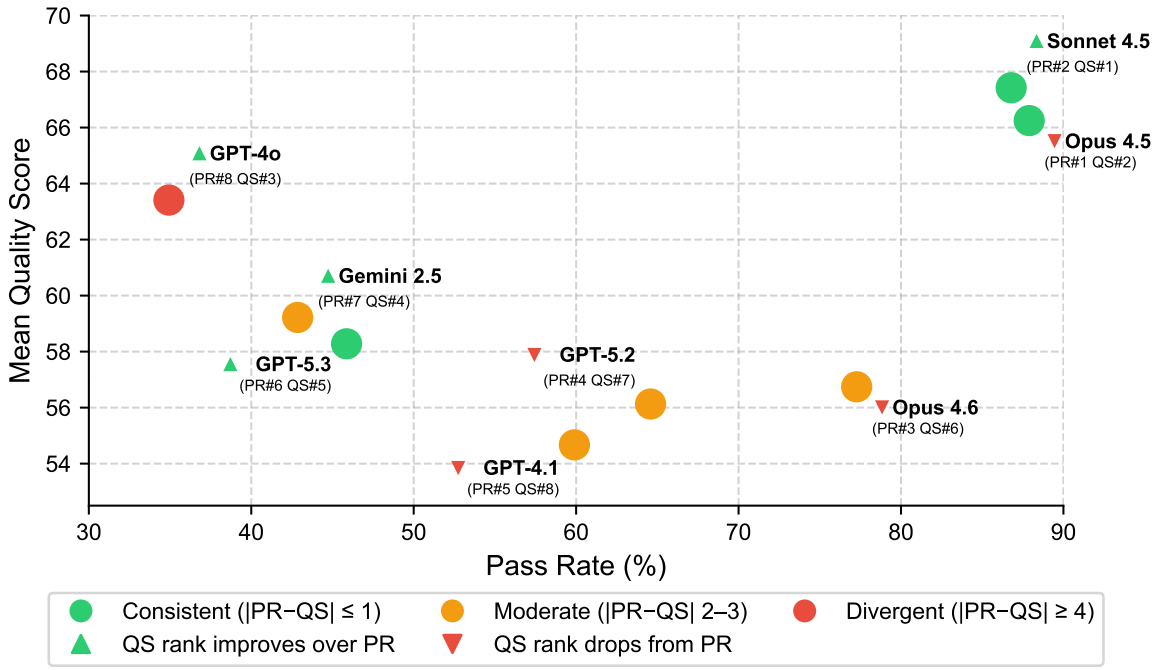

Figure 8: **Pass Rate vs. Mean Quality Score.** Each point represents one of eight LLM coding agents evaluated on AGENTLENS-BENCH. Dot color encodes rank divergence, defined as $|\text{PR} - \text{QS}|$ where PR is the pass-rate rank and QS is the quality-score rank: • consistent (≤ 1), • moderate (2–3), • divergent (≥ 4). Arrows indicate whether a model's quality rank improves (▲) or drops (▼) relative to its pass-rate rank.

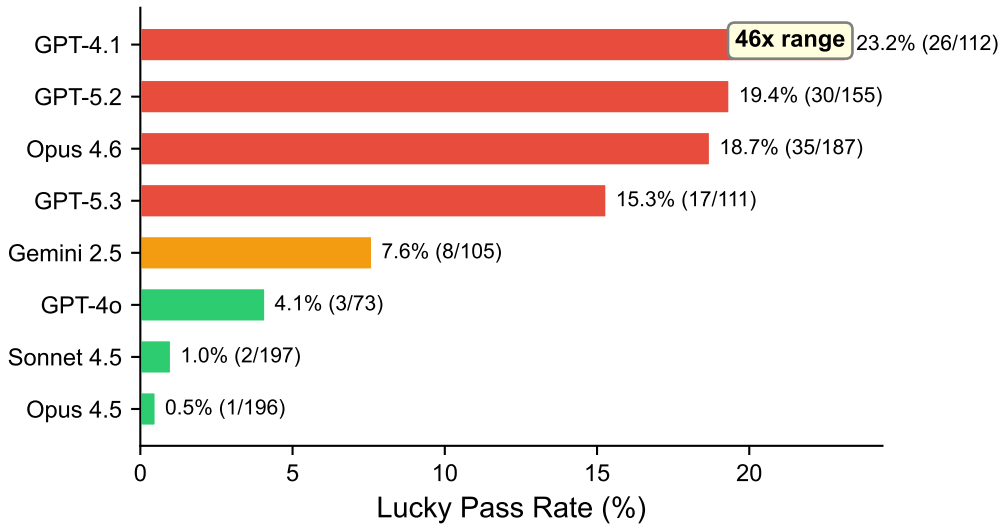

Figure 9: **Lucky rate by model.** Percentage of passing trajectories classified as Lucky tier for each model configuration. The range spans a 46× factor from Opus 4.5 (0.5%) to GPT-4.1 (23.2%).

D.2 Category Distribution

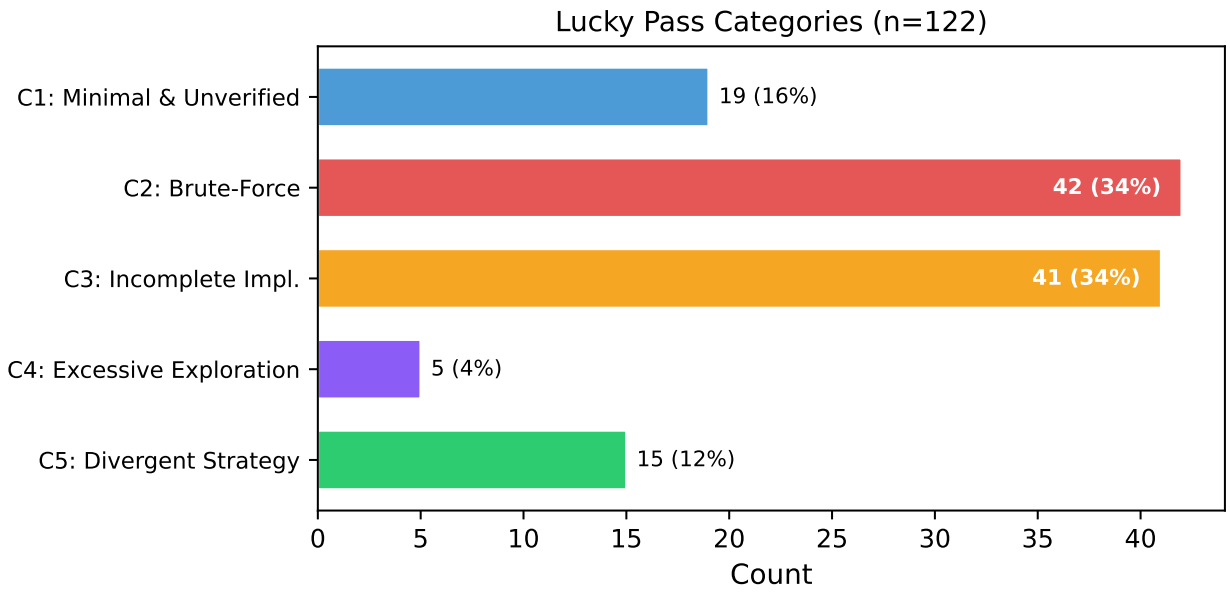

Figure 10: **Lucky Pass category distribution.** The 122 Lucky Passes decompose into five categories: C1 Minimal & Unverified (19, 15.6%), C2 Brute-Force Convergence (42, 34.4%), C3 Incomplete Implementation (41, 33.6%), C4 Excessive Exploration (5, 4.1%), and C5 Divergent-but-Valid (15, 12.3%). Categories C2 and C3 together account for 68% of all Lucky Passes.

C1: Minimal & Unverified ($n = 19$, 15.6%). The agent finds a fix in ≤ 8 steps with zero waste but skips verification entirely. These trajectories look efficient under outcome evaluation, but they lack the testing and regression checks expected from reliable SWE behavior.

C2: Brute-Force Convergence ($n = 42$, 34.4%). The agent tries multiple approaches through blind retries, cyclic patterns, or regression loops until one attempt works. This is the largest category, with mean trajectory length 35.6 states and 19.6 wasted steps per trajectory.

C3: Incomplete Implementation ($n = 41$, 33.6%). The agent implements a partial fix that addresses a surface symptom. It passes because the test suite does not cover the missing aspects of the full solution. Mean ground-truth coverage is 16.7%, indicating that the agent overlaps with less than a fifth of the reference solution space.

C4: Excessive Exploration ($n = 5$, 4.1%). The agent explores extensively in a prolonged, unfocused manner. All five cases come from GPT-4.1, making this a model-specific termination problem.

C5: Divergent-but-Valid Strategy ($n = 15$, 12.3%). The agent takes a legitimately different approach that does not align well with the current PTA but still produces a valid fix. This is the least concerning category and reflects a known limitation of structural comparison under incomplete multi-reference coverage.

D.3 Waste as a Category Differentiator

C1 and C5 both have near-zero waste, but for different reasons: C1 is too short to accumulate waste, while C5 often follows a coherent alternative strategy. C2 sits at the opposite extreme, with nearly half the trajectory wasted.

Table 13: Waste metrics by Lucky Pass category. C1 and C5 both show near-zero waste but for different reasons: C1 is too short, while C5 is coherent but structurally divergent.

Category	Mean Waste	Severity	Zero-Waste %	Mean Length
C1: Minimal	0.0	0.00	100%	3.2
C5: Divergent-valid	0.7	0.04	73%	15.5
C3: Incomplete	1.2	0.05	56%	12.1
C4: Excessive	3.2	0.09	20%	50.4
C2: Brute-Force	19.6	0.47	0%	35.6

D.4 Model-specific Lucky Pass Signatures

The main text reports that Lucky Pass categories are not uniformly distributed across models (Section 5.2). Table 14 provides the full cross-tabulation. A chi-square test on the category×model contingency table yields $\chi^2(28) = 102.47$, $p < 0.0001$, with Cramér's $V = 0.458$, indicating a large association. This means that knowing which model produced a Lucky Pass significantly predicts which category it falls into.

Three model-specific patterns are visible in the table. First, Opus 4.6 accounts for 89.5% (17/19) of all C1 trajectories. It finds fixes quickly but systematically skips verification, a confidence-calibration issue where the model is skilled enough to identify correct fixes but too confident to check them. Second, GPT-4.1 dominates C2 (18/42, 43%) and is the only model producing C4 instances. It compensates for lower initial accuracy with persistence, generating trajectories up to 100 steps long. Third, GPT-5.2-Codex and GPT-5.3-Codex cluster in C3, producing partial fixes that pass because existing test suites do not cover the full fix space. Sonnet 4.5 and Opus 4.5 produce only 2 and 1 Lucky Passes respectively, indicating that these models rarely produce low-quality passing solutions.

Table 14: Lucky Pass categories by model. Bold values mark each model's dominant category.

Model	C1	C2	C3	C4	C5	Total
opus-4.6	**17**	5	8	0	5	35
gpt-5.2-codex	0	12	**14**	0	4	30
gpt-4.1	0	**18**	1	**5**	2	26
gpt-5.3-codex	2	1	**12**	0	2	17
gemini-2.5-pro	0	2	**5**	0	1	8
gpt-4o	0	2	1	0	0	3
sonnet-4.5	0	2	0	0	0	2
opus-4.5	0	0	0	0	1	1

Opus 4.6 accounts for 89.5% of all C1 trajectories, indicating a tendency to find fixes quickly but skip verification. GPT-4.1 produces most C2 and all C4 trajectories, suggesting persistence without effective termination. GPT-5.2-Codex and GPT-5.3-Codex cluster in C3, producing incomplete implementations that pass visible tests.

D.5 Task-level Concentration

Of the 47 PTA-eligible tasks, 30 produce at least one Lucky Pass. The top 10 tasks account for 77 of the 122 Lucky Passes (63.1%), and a single task, `psf__requests-1724`, produces 16 Lucky Passes. A chi-square test on the category×task contingency table yields $\chi^2(116) = 248.45$, $p < 0.0001$, with Cramér's $V = 0.714$.

Table 15: Tasks with highest Lucky Pass concentration. The top 6 tasks account for 53/122 (43.4%) of all Lucky Passes.

Task	Lucky	Categories	Dominant Models	Pattern
`psf__requests-1724`	16	C1:3, C2:7, C3:6	opus-4.6, gpt-5.2-codex	All failure modes
`sphinx-doc__sphinx-10323`	10	C1:1, C2:5, C5:4	gpt-5.2-codex	Brute-force + divergent
`pylint-dev__pylint-6903`	8	C1:5, C2:2, C3:1	opus-4.6 (5/8)	Overconfidence
`pallets__flask-5014`	7	C2:1, C3:6	gpt-5.2-codex	Incomplete impl.
`scikit-learn__10844`	7	C2:4, C3:2, C5:1	gpt-4.1	Brute-force
`scikit-learn__12585`	6	C1:6	opus-4.6 (6/6)	Systematic overconfidence

Lucky Pass type is a property of the model-task interaction. For example, `psf__requests-1724` produces C1 from Opus 4.6, C2 from GPT-4.1, and C3 from GPT-5.3-Codex, exposing different agent shortcomings on the same task.

D.6 Case Study: One Task, Five Behavioral Profiles

The task `psf__requests-1724` provides five passing trajectories with quality scores ranging from 22 to 88. Binary evaluation treats all rows as equivalent, while AGENTLENS separates an Ideal repair, a Solid repair, and three Lucky mechanisms.

Table 16: Five passing trajectories for the same SWE-bench task.

Profile	Model	Score	States	Coherence	Coverage	Category
Ideal	GPT-4o	88	11	1.00	80.0%	–
Solid	Opus 4.5	67	28	0.53	64.0%	–
Lucky C1	Opus 4.6	33	4	0.50	12.0%	Minimal and unverified
Lucky C3	GPT-5.3-Codex	32	6	0.40	12.0%	Incomplete implementation
Lucky C2	GPT-4.1	22	34	0.03	16.0%	Brute-force convergence

GPT-4o produces the Ideal trajectory: it locates the relevant `method.upper()` call, implements the fix, adds a regression test, runs relevant tests, and reviews the diff. Opus 4.5 solves the task but explores more broadly than necessary. The three Lucky trajectories fail differently: Opus 4.6 stops after an unverified edit, GPT-5.3-Codex produces an incomplete implementation, and GPT-4.1 spends 34 states repeatedly reading and grepping without clear implementation progress.

D.7 Additional Lucky Pass Cases

This section provides additional case studies for Lucky Pass categories. Each case study reports the task, agent, trajectory length, token cost, stage sequence, and ground-truth comparison.

D.7.1 C2: GPT-5.2-Codex on `matplotlib__matplotlib-22719`

Task. Fix matplotlib's empty category converter deprecation warning (9 GT states, 359 GT states in merged PTA).

Trajectory. 18 states, 235,346 tokens. Steps 1–4 search the wrong directory (`src/matplotlib` instead of `lib/matplotlib`). Steps 6–11 recover by finding the correct path and reading `category.py`. Step 12 creates a reproducer script. Step 14 applies a single `replace_string_in_file` edit to the `_check_unit` method. Steps 15–18 run the reproducer and edge-case tests.

Why C2. Two blind retries detected, trajectory thrashing (coherence 0.20), 12 wasted steps (67% waste severity). Only 2 of 19 alignment steps match the ground truth. The wrong-directory detour and redundant file reads account for the bulk of the waste. The final edit is correct, but the path to it was not.

Cost context. 235K tokens for a fix that the ground-truth solution accomplishes in 9 steps.

D.7.2 C3: GPT-5.3-Codex on `psf__requests-1724`

Task. The same URL encoding task described in Appendix D.6.

Trajectory. 6 states, 64,314 tokens. The agent lists the workspace, navigates to the testbed, reads file contents, greps for `method.upper()`, and then creates a reproducer script via `cat > repro_unicode_method.py`. The reproducer is labeled as "implementation" by the intent labeler, but it is not a source-code edit.

Stage sequence. E→E→E→E→E→I (no V, no O).

Why C3 and not C1. Despite being short (6 states), this trajectory carries the `incomplete_implementation` failure reason at high severity. The agent's implementation covers 0% of ground-truth implementation steps because it wrote a reproducer rather than editing the actual source file. Coverage: 12.0%, coherence: 0.40.

Cross-category note. This task (`psf__requests-1724`) appears as C1 (Opus 4.6: 4 states, targeted edit, no verification), C2 (GPT-4.1: 34 states, chaotic exploration, no edit), and C3 (GPT-5.3-Codex: 6 states, incomplete implementation). The same task produces three different Lucky Pass categories depending on the model, confirming that Lucky Pass type is a property of the model-task interaction rather than the task alone.

D.7.3 C4: GPT-4.1 on `django__django-11066`

Task. Fix Django content types management to pass `using=db` to `content_type.save()` (15 GT states, 621 GT states in merged PTA).

Trajectory. 100 states, 2,620,266 tokens. The agent begins by listing locale directories (wrong area). Step 2 finds the target file via `grep`. Step 3 reads `contenttypes/management/__init__.py`. Step 4 attempts a `replace_string_in_file` edit that fails (`old_str` not found). Step 6 succeeds with a corrected edit (adds `using=db`). Step 43 runs the contenttypes test suite and tests pass. Steps 44–100 are 57 additional exploration steps: the agent reads the same directories repeatedly, producing identical `ls` output on steps 96–100.

Stage sequence. E→E→E→I→V→I→O→E×35→V→E×57.

Cost. 2.62M tokens for a one-line fix. This is 77× the C1 average (34K). Of the 100 states, 95 use `run_in_terminal`, 2 use `read_file`, 2 use `replace_string_in_file`, and 1 uses `think`. Only 4 alignment steps match.

Model-specific pattern. All five C4 instances come from GPT-4.1. The model lacks a termination heuristic: it continues exploring after the fix has been verified. The 57 post-verification exploration steps provide no value but cost over 1.5M tokens.

D.7.4 C5: Opus 4.6 on `pylint-dev__pylint-4970`

Task. Fix pylint's `similar` checker to handle `min_similarity_lines=0` (21 GT states, 276 GT states in merged PTA, 24 files).

Trajectory. 8 states, 88,207 tokens. Six exploration steps search for the target file, read the `run()`, `process_module`, and `_compute_sims` methods. Two implementation steps edit `pylint/checkers/similar.py`, modifying `process_module` and `run()` to add early-return guards for `min_similarity_lines=0`.

Stage sequence. E→E→E→E→E→E→I→I.

Why C5. The agent's approach is genuinely alternative: it modifies `process_module` and `run()` with early-return guards, while the ground truth approaches the same problem through different code paths. Coverage is 14.3%, but the unmatched 85.7% is mostly verification and orchestration that the agent skipped, not implementation it missed. The agent's two matched alignment steps hit ground-truth states for implementation, confirming functional overlap.

D.7.5 C5: GPT-5.2-Codex on `sphinx-doc__sphinx-10323`

Task. Fix Sphinx `literalinclude` directive's `dedent` interaction with `prepend/append` (53 GT states, 384 GT states in merged PTA, 60+ files).

Trajectory. 14 states, 182,820 tokens. The agent finds the target file, reads the `append_filter` method and filter application order, creates a reproducer, verifies the bug, implements a fix by reordering filter application in `sphinx/directives/code.py`, re-verifies, and runs a parser verification test.

Stage sequence. E→E→E→E→E→E→V→I→V→I→V→V→O.

Why C5 is the least concerning category. Unlike the other Lucky Pass categories, this trajectory demonstrates a complete E→I→V→O lifecycle. The agent created a reproducer, verified the bug, implemented a fix, and verified the fix works. Its low quality score (37) reflects low ground-truth coverage (11.3%), but the coverage gap is structural (different approach) rather than qualitative (bad approach). This category argues for multi-reference ground truths and is why the PTA merges $k \geq 2$ traces.

D.8 Verification Gap

Lucky Passes arise from the intersection of three factors: an agent shortcoming (skipping verification, lacking planning, producing partial fixes, over-exploring, or using an alternative approach), a test-suite gap (insufficient coverage to catch the shortcoming), and task characteristics (some tasks admit multiple valid solutions). The verification gap is the most actionable of these factors because it is directly addressable through agent training or scaffolding changes.

Of the 122 Lucky Passes, 94.3% have missing verification as a failure reason. Table 17 shows that this gap is not uniform across categories. C1 trajectories have zero verification by definition (the agent stops before running any test). C5 trajectories have the highest verification rate (87%) because agents in this category often follow a complete E→I→V→O lifecycle and are classified as Lucky only because their solution path diverges from the PTA. The gradient from C1 to C5 maps directly onto a gradient from most concerning to least concerning Lucky Passes.

This gradient has practical implications. Models that cluster in C1 (Opus 4.6) need verification training: the model should learn to run tests after making edits. Models that cluster in C2 (GPT-4.1) need planning and termination heuristics: the model should learn to stop exploring after a fix has been verified. Models that cluster in C3 (Codex variants) produce incomplete implementations that require better test suites to catch. The taxonomy identifies which intervention each model needs, and the verification gap is the clearest signal for prioritizing those interventions.

Table 17: Verification stage coverage by Lucky Pass category.

Category	No verification (V=0)	Some verification (V>0)
C1: Minimal & Unverified	19 (100%)	0 (0%)
C2: Brute-Force Convergence	17 (40%)	25 (60%)
C3: Incomplete Implementation	18 (44%)	23 (56%)
C4: Excessive Exploration	1 (20%)	4 (80%)
C5: Divergent-but-Valid	2 (13%)	13 (87%)

E Ablation Details

Section 6 in the main text summarizes the ablation findings in compact form. This appendix provides the full tables and extended discussion. All ablation experiments use the pilot calibration set introduced in Section 4. The pilot contains 278 trajectories across 10 SWE-bench tasks from five agent configurations run under the OpenHands scaffold, and is entirely disjoint from the 2,614-trajectory scaled evaluation corpus. The pilot was used for two purposes: grid-search weight optimization (producing the weight vector $w = (0.20, 0.15, 0.30, 0.35)$ with pilot AUROC $= 0.755$ and F1 $= 0.791$) and the controlled ablation experiments below. All weights and thresholds were frozen before any scaled-set experiment was run.

We test three design decisions. First, whether all four scoring signals are necessary or whether a subset would suffice (E.1). Second, how many passing trajectories should be merged into each task-level PTA (E.2). Third, whether the order in which trajectories are merged affects the resulting PTA and downstream scores (E.3).

E.1 Signal Contribution

The combined score fuses four signals. To test whether each signal is genuinely necessary, we ablate one at a time and measure the AUROC drop on the pilot holdout. Table 18 reports the results.

The two behavioral signals (temporal profile and trajectory coherence) produce the largest drops when removed (-0.037 and -0.031 respectively), confirming that the behavioral dimension is not redundant with structural matching. The two structural signals produce smaller but still meaningful drops (-0.024 for set coverage, -0.016 for structural alignment). No single signal is dispensable: even the smallest drop (structural alignment) represents a statistically meaningful reduction in discrimination.

The weight vector is robust to perturbation. Shifting any single weight by ± 0.05 (while re-normalizing to maintain the unit-sum constraint) reduces combined AUROC by at most 0.006. This stability means that the exact weight values are not fragile design choices, and practitioners who want to emphasize a specific dimension (e.g., weighting verification discipline more heavily for safety-critical applications) can do so without breaking the overall scoring system.

Table 18: Signal ablation on the pilot calibration set ($n = 278$). Behavioral signals are more impactful than structural ones in isolation; all four are necessary for maximum discrimination.

Configuration	AUROC	Δ from full
Full combined (all 4 signals)	0.755	—
Remove temporal profile (Φ_{temp})	0.718	-0.037
Remove trajectory coherence (Φ_{coh})	0.724	-0.031
Remove set coverage (Φ_{cov})	0.731	-0.024
Remove structural alignment (Φ_{struct})	0.739	-0.016

E.2 Merge-count Sensitivity

The number of passing trajectories k merged into each task-level PTA controls a precision-coverage trade-off. Table 19 reports combined AUROC as a function of k, evaluated on the pilot set with 3 random resamples per k per eligible task.

At low k, the PTA is compact and precise: it encodes only a few solution strategies and penalizes any valid alternative not represented in the reference. At higher k, the PTA admits more valid paths and better represents the diversity of correct solutions, but the graph becomes more permissive and branches that correspond to genuine strategic choices become harder to distinguish from noise. At $k \geq 6$, a secondary effect dominates: PTA size causes some tasks to exceed scoring limits, and only easier-to-score task resamples survive. The rising AUROC at these values reflects this survivorship bias rather than genuine improvement in scoring quality.

$k = 2$ achieves the highest AUROC (0.749) at full task coverage, but it encodes at most two solution strategies per task. For tasks with multiple valid approaches, this is too narrow. $k = 5$ provides a better balance: it covers a substantially larger solution space, its AUROC (0.777) exceeds that of $k = 2$, and the reduction in task coverage from 41 to 31 resamples reflects the PTA-size scoring limit rather than a methodological failure.

Table 19: Merge-count sensitivity on the pilot set (3 resamples per k per eligible task). $k = 5$ balances solution-space coverage against scoring precision.

k	AUROC	Acc.	Task-resamples scored
2	0.749 ± 0.232	81.2%	41 / 41
3	0.697 ± 0.273	79.6%	40 / 41
4	0.670 ± 0.293	77.2%	36 / 41
5	$\mathbf{0.777 \pm 0.220}$	**83.1%**	31 / 41
6	0.863 ± 0.186	89.7%	13 / 41
7	0.822 ± 0.133	81.3%	10 / 41

We use $k = 5$ for the scaled experiments because it covers a broader solution space than $k = 2$ while remaining below the severe survivorship regime observed at larger k.

E.3 Merge-order Robustness

Because PTA construction is incremental (trajectories are merged one at a time into the growing graph), a natural concern is whether the order in which trajectories are merged affects the resulting PTA structure and downstream scores. If it did, the scoring pipeline would be fragile: the same set of passing trajectories could produce different quality scores depending on an arbitrary processing order.

To test this, we selected one pilot task (`astropy__astropy-12907`) with $k = 4$, enumerated 10 random trajectory combinations, and ran all 6 permutations of each combination, yielding 60 total scoring runs. Table 20 reports the per-combination results. Trajectory selection accounts for 64.1% of total variance, while merge ordering accounts for 35.9%. Eight of ten combinations produce zero within-combination variance and are fully order-invariant. The three combinations with nonzero ordering variance (1, 5, 6) share a common property: they include at least one trajectory whose exploration prefix is ambiguous under the equivalence engine, so that small ordering differences determine whether a prefix state is merged or branched. Even in these cases, the AUROC range is bounded and the effect is substantially smaller than the effect of which trajectories are selected. The main source of PTA variation is reference-set choice, not merge ordering.

Table 20: Merge-order study: per-combination AUROC across all 6 permutations ($k = 4$, `astropy__astropy-12907`, 60 total runs).

Combination	AUROC (mean $\pm$ std)	Perfect?	Order-invariant?
0	0.667 ± 0.000	No	✓
1	0.778 ± 0.172	No	×
2	1.000 ± 0.000	✓	✓
3	1.000 ± 0.000	✓	✓
4	1.000 ± 0.000	✓	✓
5	0.944 ± 0.136	No	×
6	0.833 ± 0.183	No	×
7	1.000 ± 0.000	✓	✓
8	1.000 ± 0.000	✓	✓
9	1.000 ± 0.000	✓	✓
Overall	$\mathbf{0.922 \pm 0.141}$	6/10	8/10

F Token Cost and Statistical Tests

The Lucky Pass taxonomy reveals not only qualitative differences in agent behavior but also quantitative differences in computational cost. Binary evaluation treats all correct patches as equivalent, but the token expenditure behind those patches varies by a factor of 40 across Lucky Pass categories. Table 21 reports the breakdown.

The cost gradient follows a clear pattern. C1 (Minimal & Unverified) trajectories are the cheapest because they are the shortest: mean 34K tokens per trajectory. C3 (Incomplete Implementation) and C5 (Divergent-but-Valid) are moderately expensive. C2 (Brute-Force Convergence) is 25× more expensive than C1 because the agent spends many steps retrying failed approaches. C4 (Excessive Exploration) is the most expensive at 40× C1, with one trajectory consuming 2.62M tokens for a one-line fix.

Table 21: Token cost by Lucky Pass category. C4 costs 40× more than C1 on average for the same binary evaluation outcome.

Category	n	Mean Tokens	Range	vs. C1
C1: Minimal	19	34,032	22K–75K	1.0×
C3: Incomplete	41	162,024	55K–520K	4.8×
C5: Divergent-valid	15	232,854	59K–882K	6.8×
C2: Brute-Force	42	856,512	120K–3.17M	25.2×
C4: Excessive	5	1,377,226	569K–2.62M	40.5×

Aggregate token expenditure across all 122 Lucky Passes is approximately 58.3M tokens. If all were solved at C1 efficiency (34K each), the total would be 4.1M tokens. The excess 54M tokens (93% of the aggregate) is

attributable to C2 and C4 patterns. At current API pricing, the 122 Lucky Passes cost roughly 14× what they would cost if every model solved tasks at C1 efficiency. A model that produces C2 (brute-force) Lucky Passes is 25× more expensive per trajectory than one that produces C1 (minimal) Lucky Passes, yet both receive identical binary scores. Token efficiency is invisible to pass/fail evaluation but directly affects deployment economics.

F.1 Statistical Tests for Lucky Pass Taxonomy

Table 22: Association tests for the Lucky Pass taxonomy.

Test	χ^2	**dof**	p	**Cramér's** V
Category × Model	102.47	28	< 0.0001	0.458 (large)
Category × Task	248.45	116	< 0.0001	0.714 (very large)

Both associations are highly significant with large to very large effect sizes, confirming that (1) model choice strongly predicts the type of Lucky Pass produced, not just whether one occurs, and (2) task characteristics strongly predict which Lucky Pass category dominates. The per-category verification coverage further supports the causal structure described in Section D.8: C1 trajectories have zero verification coverage by definition, C2 and C3 have 40–60% verification rates, and C5 has 87% verification.