Fantastic Adaptive Taxonomies and How to Use Them

Mert Cemri [*1] **Andrei Cojocaru** [*1] **Melissa Pan** [1] **Shu Liu** [1] **Shubham Agarwal** [1] **Alexander Krentsel** [1] **Jay Tang** [1] **Kannan Ramchandran** [1] **Joseph E. Gonzalez** [1] **Matei Zaharia** [1] **Alex Dimakis** [1 2] **Ion Stoica** [1]

Abstract

Many LLM-based methods improve agent systems without changing model weights: a verifier selects the best trajectory from N attempts, a prompt or workflow optimizer rewrites the agent after failed runs, and a runtime reflection module asks the agent to reconsider before acting. These methods typically operate by asking an LLM to read execution traces directly. We show that they can be improved by first constructing a failure taxonomy for the target domain and agent system, and then passing this taxonomy to the LLM-based improvement procedure.

We present ATLAS (Automatic Taxonomy Learning for Agent Systems), a framework that creates *Adaptive Taxonomies*: 15–30 failure codes induced from execution traces, with no hand-authored codes and no per-trace annotation. We evaluate ATLAS's adaptive taxonomies in three downstream procedures. (1) As best-of-N judges on Terminal-Bench 2.0, ATLAS based LLM-Judge agents set a new high score with 89.9% accuracy and outperform judges with fixed taxonomies or no taxonomies up to 15%. (2) As mutation feedback for evolutionary agent-system optimization, taxonomy-coded diagnoses outperform free-form LLM reflection across competitive programming, math, STEM QA, and discrete reasoning. (3) As runtime feedback for SWE-agent on SWE-bench-Verified-Mini, adaptive taxonomies improve over free-form reflection in both in-prompt and external-judge integrations. On TRAIL, induced taxonomies align with expert annotations at Cohen's κ=0.725.

[1]University of California, Berkeley [2]Bespoke Labs. Correspondence to: Mert Cemri <cemri@berkeley.edu>.

Published at the ICML 2026 Workshop on Failure Modes in Agentic AI (FAgEn). Copyright 2026 by the author(s).

1. Introduction

Procedures that improve LLM agent systems consume execution traces and act on the failures those traces expose: a best-of-N selector chooses among completed trajectories, a program-search loop rewrites prompts and roles based on prior failures, and a runtime monitor inspects partial traces before an action commits. Although two failed runs of the same agent on the same task may fail for unrelated reasons, trace summaries often are unable to preserve the relevant distinctions. Scalar outcomes aggregate cleanly but discard why a trajectory failed. Free-text critiques (Madaan et al., 2023; Shinn et al., 2023) can describe concrete mistakes, but they are unstructured, instance-specific, and must be reinterpreted by every downstream consumer. Hand-authored rubrics such as MAST (Cemri et al., 2025) impose structure but fix the failure vocabulary before observing the agent, its roles, or the target domain, so they capture cross-cutting failures such as premature termination or repeated tool use while missing the role- and domain-specific errors that determine how a particular agent should be improved. We show that these procedures improve when we insert a step between the trace and the LLM: induce a failure taxonomy from the traces, then pass the taxonomy together with each trace to the procedure's LLM.

Concrete failures from the domains we study illustrate these gaps. A terminal agent on Terminal-Bench 2.0 may respond to a tool error by abandoning its objective and retreating into read-only commands (`Defensive_Pivoting`). A competitive-programming agent on Frontier-CS may commit to a greedy strategy on a problem whose structure requires dynamic programming (`Algorithm_Mismatch`). A STEM agent on MMLU-Pro may produce a numerically coherent derivation that violates a conservation law (`Physical_Law_Violation`). Each demands a different intervention (changing termination and tool-use policy, swapping the solver's algorithmic prior, or adding domain-specific checks), and a fixed catalogue cannot anticipate this long tail. MAST (Cemri et al., 2025) demonstrated that even a hand-curated 14-mode multi-agent taxonomy substantially improves debugging effectiveness, and we build directly on that line of work. However, the limitation is that a fixed

vocabulary cannot expose failures specific to a particular domain, architecture, or set of agent roles if those failures are not included in the vocabulary. The diagnostic vocabulary needed to improve a specific agent system, therefore, must be specific to its own traces.

We present ATLAS (Automatic Taxonomy Learning for Agent Systems), a framework that induces an *adaptive taxonomy* directly from agent execution traces, with no hand-authored failure codes and no manual annotation of any individual trace. Given traces from a target system, ATLAS produces 15–30 categorical *failure codes* organized along three fixed axes: *system-level* failures that arise in any agent system, such as context exhaustion or premature termination; *role-specific* failures tied to responsibilities discovered in the component's trace, such as a checker that rubber-stamps a solver's output; and *domain-specific* failures that require task knowledge, such as algorithm mismatch or physical-law violation. The axes are fixed and are motivated by the empirical clustering of multi-agent failure modes introduced by MAST (Cemri et al., 2025); the concrete codes, role labels, descriptions, and evidence patterns are generated from the traces themselves (Section 3.1).

We evaluate ATLAS's adaptive taxonomies as feedback in three downstream procedures (results in Section 4). (1) As best-of-N judges on Terminal-Bench 2.0, ATLAS-Judge sets a new high score (89.9% accuracy, $+15$ pp over Pass@1) and outperforms judges with fixed or no taxonomies. (2) As mutation feedback for evolutionary agent-system optimization, taxonomy-coded diagnoses outperform free-form LLM reflection across competitive programming, math, STEM QA, and discrete reasoning. (3) As runtime feedback for SWE-agent on SWE-bench Verified Mini, adaptive taxonomies improve over free-form reflection in both in-prompt and external-judge integrations.

How faithful are these induced codes to human judgment? On TRAIL (Deshpande et al., 2025), a benchmark of 117 expert-annotated GAIA traces, a four-LLM panel using ATLAS-induced codes reaches Cohen's κ=0.725 against the human gold and recovers expert-labeled failures more faithfully than TRAIL's hand-crafted vocabulary (Appendix F.1), indicating that the induced codes align with human-recognizable failure modes rather than reflecting an LLM-internal labeling regime.

Our contributions are:

- **Automated Taxonomy Generation.** We introduce a fully automated method for inducing compact, evidence-grounded failure taxonomies from execution traces of a target agent system, without hand-authored failure codes or manual trace annotation.
- **Taxonomies as reusable feedback interfaces.** We show that the same induced failure vocabulary can be consumed by multiple downstream improvement procedures: best-of-N trajectory selection, evolutionary agent-system optimization, and runtime agent feedback.
- **Empirical validation across agentic benchmarks.** ATLAS's adaptive taxonomies improve over free-form reflection and fixed hand-authored taxonomies in three settings: best-of-N selection on Terminal-Bench 2.0, evolutionary agentic system optimization on Frontier-CS, OlympiadBench, MMLU-Pro, TheoremQA, and DROP, and runtime feedback for SWE-agent on SWE-bench Verified Mini. We also validate ATLAS-generated taxonomies' alignment to expert human annotations on TRAIL dataset, showing they achieve Cohen's $\kappa = 0.725$, demonstrating substantial agreement.

2. Related Work

Multi-agent failure analysis. MAST (Multi-Agent System failure Taxonomy) (Cemri et al., 2025) introduced the first empirically grounded taxonomy of 14 failure modes across 3 categories for multi-agent LLM systems, validated on more than 1,600 traces with $\kappa = 0.88$. Subsequent work extends this line: Jia et al. (2025) propose a Goal-Plan-Action framework that uses LLM judges to assess agent alignment; TRAIL (Deshpande et al., 2025) releases 148 expert-annotated traces with an error taxonomy for agent debugging, which we use as a human-anchored validation set for INTER-ANNOTATOR AGREEMENT in §F.1; Pan et al. (2026) documented reliability challenges in production agent deployments. Instance-level self-reflection methods such as Reflexion (Shinn et al., 2023) and Self-Refine (Madaan et al., 2023) produce per-problem critiques rather than categorical codes that aggregate across problems and persist across iterations. These methods are diagnostic: they identify failure patterns but do not use them to verify or improve systems. ATLAS closes this loop by inducing domain-specific taxonomies from execution traces and applying them to verification and architecture search.

LLM-as-a-judge and agent verification. LLM-as-a-Judge (Zheng et al., 2023) uses generic criteria like helpfulness and correctness that do not adapt to task domain. LLM-based trace verifiers (Kwok et al., 2026) keep the rubric fixed and vary the verifier, and Pan et al. (2026) notes that standard metrics miss domain-specific failures. Process-supervision verifiers (Lightman et al., 2024) score reasoning steps from human-labeled correctness signals, leaving the criteria implicit in the labels. ATLAS-Judge instead induces scoring criteria from execution traces and names them as categorical failure codes, so one vocabulary drives selection (§4.2), mutation feedback (§4.3), and runtime checkpointing

(§4.4).

Search-based optimization of agent systems. A recent line of work optimizes LLM-driven agent systems by searching over their components. Program-level evolution methods (FunSearch (Romera-Paredes et al., 2024), AlphaEvolve (Novikov et al., 2025), AdaEvolve (Cemri et al., 2026)) evolve programs with scalar feedback; ATLAS runs inside AdaEvolve as a fixed substrate. Multi-agent design methods explore agent architecture (ADAS (Hu et al., 2025), EvoAgent (Yuan et al., 2024), GPTSwarm (Zhuge et al., 2024)) and workflow structure (AFlow (Zhang et al., 2025)); prompt-level methods (Promptbreeder (Fernando et al., 2023), EvoPrompt (Guo et al., 2024), GEPA (Agrawal et al., 2025), DSPy (Khattab et al., 2024)) treat prompts as the search variable; tree-search methods like LATS (Zhou et al., 2024) prune trajectories at inference time. None of these incorporate structured failure diagnosis into the search signal: they use outcome-style rewards. The closest prior art is the Trace/OPTO line (Cheng et al., 2024) and TextGrad (Yuksekgonul et al., 2024), which formulate LLM-agent improvement as backpropagation over execution graphs and ingest structured feedback at intermediate nodes; both supply that signal per-instance. ATLAS instead auto-generates the categorical signal from the taxonomy pipeline and accumulates it across problems and iterations.

3. Method

We first define the following terms. A **failure mode** is a recurring way an agent execution deviates from its intended outcome (e.g., `Premature_Reasoning_Truncation`: "agent stops a multi-step chain before reaching a final answer") characterized by a name, description, and evidence pattern. A **failure taxonomy** is a structured catalog of such modes, and a **judge** is an LLM-driven mechanism that applies a taxonomy to execution traces and emits which codes fired, on which tasks, with what evidence.

ATLAS's core component is a taxonomy generator (§3.1) that receives execution traces and returns a domain-specific taxonomy of 15–30 failure codes, organized into three fixed top-level categories, system-level, role-specific, and domain-specific. The empirical case for categorical failure analysis over scalar scores or free-text critique has already been made by Cemri et al. (2025); we take it as given. The question we address is the next one: whether the codes themselves should be fixed across systems, as in a hand-authored catalog, or induced from the traces of the system being analyzed. We argue that adaptation matters because failure modes fire in a given system depending on what agents it has, what tasks they solve, and how they interact; a generic catalog will mismatch the failure surface of any specific system, while codes induced from that system's own traces are shaped by it. Empirically: role-specific codes appear only in multi-agent architectures (13 codes for Frontier-CS, 0 for flat solver-verifier pipelines like TheoremQA), and domain-specific codes show low overlap across our six evolution domains (Appendix A.7). The rest of this section describes how the induced taxonomy is generated (§3.1) and how it supports three downstream usage patterns, *trajectory verification* (§3.2), *evolutionary architecture search* (§3.3), and *runtime failure-mode feedback for agentic system* (§3.4), each consuming the taxonomy through a different judge mechanism.

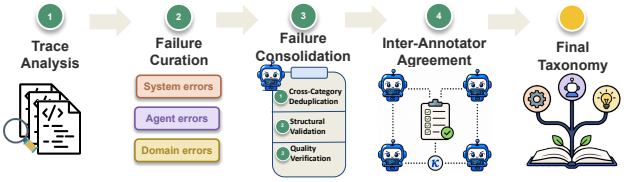

Figure 1: ATLAS overview. The taxonomy generator runs four LLM-driven phases: ANALYSIS, GENERATION, CONSOLIDATION, and INTER-ANNOTATOR AGREEMENT. Each arrow denotes the output of one phase becoming the input to the next; more details in §3.1 and Appendices C.1, D.5.

3.1. Generating the taxonomy

The generator runs in four LLM-driven steps (Figure 1); each step is a separate LLM call. The split is deliberate: separating analysis from generation prevents premature commitment to obvious codes, parallel category generators prevent collapse between categories, consolidation must come last to resolve the overlap that parallel generation inevitably produces, and inter-annotator agreement provides the consistency check that none of the upstream construction steps perform. The INTER-ANNOTATOR AGREEMENT step is the gate: four LLM annotators (two Claude Opus, two GPT-4o) independently apply the codes to 50 traces with one round of peer deliberation, and we accept taxonomies reaching $\kappa \geq 0.70$.

The ANALYSIS step extracts domain context, agent roles, and behavioral signals (errors, loops, stalls) from the traces. The GENERATION step produces failure codes along the three category axes in parallel, each grounded in the analysis. The CONSOLIDATION step deduplicates codes across categories under role-boundary rules, validates each code against its category's structural requirements, and rejects codes with placeholder text, missing evidence, or near-duplicate overlap. The full checklist appears in Appendix C.

Failure categories. The three top-level categories, system-level, role-specific, and domain-specific, partition failures by intervention point: a failure is assigned to the category

SYSTEM-LEVEL fix at substrate

A.2 Premature pipeline termination

Output delivered before a final solution is produced; final answer field empty or truncated.
EVIDENCE
`Checker_3 emits empty return tag; no answer field; pipeline halts before solver finalizes.`

ROLE-SPECIFIC fix at role

B.1 Checker rubber-stamp verification

Checker endorses solver output without genuine re-derivation or cross-check.
EVIDENCE
`Checker accepts WA solver output with "looks correct" verdict; no proper validation sequence.`

DOMAIN-SPECIFIC fix in domain

C.4 Degenerate case omission

Boundary or degenerate configurations neglected (zero-length segments, $n{=}1$ inputs).
EVIDENCE
`Solver passes random tests; fails n=1 with index error, no boundary check in submitted code.`

Figure 2: Failure categories. One representative generated code per category, with evidence excerpts from real evolution traces.

whose remedy applies. Figure 2 shows one representative code per category, with evidence excerpts from real evolution traces; a full generated taxonomy for Frontier-CS, with firing frequencies, is in Appendix A. In addition, the categories also differ in how their codes are sourced: system-level and domain-specific codes combine priors (architectural-risk patterns; common error modes) with trace-grounded discovery, while role-specific codes are entirely trace-induced, since the roles themselves are discovered from the traces and admit no general catalog.

3.2. Usage 1: Designing better LLM-as-a-Judge for test-time-scaling

As the first use case, we ask the question: Can we use ATLAS-generated adaptive taxonomies to obtain better LLM-Judges? In particular, we test the usefulness of LLM-Judge agents for test-time scaling, where, given N independent agent trajectories on the same task, we expect the model to be able to differentiate successful traces from failed ones and pick the best trace out of N as the final answer (Snell et al., 2024). For this, following prior work of (Gu et al., 2024; Kwok et al., 2026) we ask an LLM agent to do pairwise comparisons with position debiasing to correct for systematic ordering effects. In particular, we first build an adaptive taxonomy as discussed in Section 3.1 from traces particular to a selected domain and an agentic architecture, then put this taxonomy inside an LLM-Judge to do these comparisons for new traces at test-time, thereby allowing the judge to use the taxonomy that has encoded information from previous rounds. The results from Section 4.2 across different harnesses on Terminal-Bench (Merrill et al., 2026) show that using ATLAS for LLM-Judge during such test-time scaling tasks clearly outperforms judges not using any taxonomies or fixed taxonomies.

3.3. Usage 2: Evolutionary agentic system optimization

The second use case of the adaptive taxonomy is designing better agentic systems. Here, we target optimizations such as modifying the multi-agent topology (adding or removing agents from a graph), optimizing agent prompts, or writing new tools to improve the harness around the agents. We use AdaEvolve (Cemri et al., 2026) as the substrate: given a seed agentic system and a task evaluator, AdaEvolve iteratively mutates the system's source code to improve evaluator scores, maintaining a population of candidate variants and selecting which to mutate next based on prior performance. ATLAS plugs into this loop at the mutation step: instead of mutating from score-only feedback, the mutator receives a taxonomy-based diagnosis of the parent's execution traces, recurring failure codes, quoted evidence, and cross-problem patterns that explain why the parent failed (representative diagnoses are shown in Appendix G.4). To keep the taxonomy in step with the evolving system, we run an online refinement round (Appendix D.5) when search stagnates: the round proposes targeted edits, merging overlapping codes, adding codes for newly observed failure patterns, or relabeling codes whose boundaries have drifted, over the post-warmup trace pool. Warmup parameters and refinement triggers are in Appendix C.

3.4. Usage 3: Runtime integration for agentic systems

The third use case is exposing the taxonomy as a runtime *skill*: a drop-in capability that any harness supporting tool calls can attach to its agent without modifying the loop. Systems such as SWE-agent, Claude Code, and ReAct-based harnesses run action-observation loops without structured failure diagnosis, so failure modes that recur across the agent's history are visible only through free-form introspection. Attaching the taxonomy as a skill changes this: at declared checkpoints (sub-task completions and a mandatory pre-submission check) the agent consults a structured failure vocabulary rather than generating whatever explanation sounds plausible.

Keeping the taxonomy outside the agentic system entirely and consulting it through a separate LLM call at each checkpoint yields a critical advantage: the judge sees only the trajectory, not the agent's plans or reasoning state, eliminating the self-evaluation bias that lets an agent rationalize its own mistakes. At each checkpoint, the agent sends a structured summary of its recent trajectory; the judge applies the taxonomy and returns the most relevant failure codes with concrete next-step guidance, delivered as the

agent's next observation. The agent's narrative of what it did contaminates its evaluation of whether what it did was correct; the external judge has no such narrative to suppress. We evaluate both integration patterns empirically in §4.4.

4. Experiments

We evaluate ATLAS as feedback for three downstream procedures, plus an external faithfulness check. The three pillars are best-of-N *trajectory selection* on Terminal-Bench 2.0 (Section 4.2), iterative *agent-system optimization* across four reasoning benchmarks with an Olympiad-Bench deep-dive (Section 4.3), and *runtime integration* with SWE-agent on SWE-bench Verified Mini (Section 4.4). The TRAIL validation step (Appendix F.1) tests whether the induced codes recover the same failures human experts identify.

4.1. Setup

Models. Evolution experiments (mutation operator, solver agents within evolved architectures, and feedback for the LLM-reflection baseline) and the taxonomy generation pipeline use GPT-5.4-mini at every stage. ATLAS-Judge on Terminal-Bench 2.0 uses GPT-5.4 as the verifier across all three configurations, providing a cross-family design against Opus-4.6 agents. Runtime integration on SWE-bench uses GPT-5 as the SWE-agent solver. The TRAIL panel uses Claude Opus 4.7, GPT-5.4, Gemini-2.5-Pro, and GPT-5.4-mini.

Benchmarks. Terminal-Bench 2.0 (89 terminal tasks, 5 pre-recorded trajectories per task) supports the test-time scaling study in Section 4.2. Frontier-CS (Mang et al., 2025) (172 competitive-programming problems, 40 train / 132 held-out), TheoremQA (30 graduate math problems), MMLU-Pro (Wang et al., 2024) (40 STEM problems), OlympiadBench (20 dev / 655 held-out) and DROP (Dua et al., 2019) (30 discrete-reasoning problems) support the multi-benchmark agentic system optimization comparison. SWE-bench Verified Mini (50 instances) supports the runtime study, and TRAIL (Deshpande et al., 2025) (117 GAIA traces, four-expert gold) supports the validation.

4.2. Best-of-N trajectory selection on Terminal-Bench 2.0

We evaluate the ATLAS based LLM-Judge on Terminal-Bench 2.0 (Merrill et al., 2026) using three different agentic harnesses: *terminus-2* (Merrill et al., 2026), *claude-code* (Anthropic, 2025) and ForgeCode (TailcallHQ, 2026) which currently tops the Terminal-Bench 2.0 leaderboard at Pass@1 81.8%. All three configurations use Opus-4.6 as the trace generator and GPT-5.4 as the LLM-Judge model. The trajectory pool is fixed across methods, so any difference cannot be attributed to stochastic differences among the agents being judged. The implementation details are in Appendix C.

Headline result. On ForgeCode, ATLASbased LLM-Judge (that we call ATLAS-Judge for simplicity) reaches 89.9% and saturating the Best-of-5 oracle without any change to the agent. Across the three configurations, full-benchmark accuracy improves over Pass@1 by +11.2%, +14.9%, and +8.1%, closing 71%, 65%, and 100% of the gap to the oracle (Table 1).

Where the gain comes from. A plain LLM-as-a-Verifier scores trajectories on generic criteria such as helpfulness or output match. Substituting MAST's 14 cross-cutting codes into ATLAS-Judge's pipeline already lifts accuracy because the verifier is now grounded in named failure patterns (premature termination, repetitive looping, context exhaustion) and the forward selector has discriminative features to choose from rather than free-form judgments. The remaining +3.4 to +4.5% gap on the 30- and 35-task swing sets isolates the value of *terminal-specific* codes MAST's universal vocabulary does not contain: `Defensive_Pivoting`, `Service_State_Verification`, and `Training_Stall_Detection` name precisely the failures that decide passing from failing trajectories on terminal tasks (full code list in Appendix A.3). On Forge the comparison is uninformative because both methods saturate the oracle within a single-task margin (17/17 vs. 16/17 on a 17-task swing set).

Held-out validation and selector behavior. A 5-fold split stratified by task category regenerates the taxonomy on each train fold and evaluates only on the held-out fold. Held-out accuracy meets or exceeds same-pool on the two discriminative configurations (terminus-2 +1.2%, claude-code +5.8%), ruling out taxonomy overfitting to the evaluation pool; Forge sits at the oracle ceiling under both protocols. The forward selector adapts per configuration: terminus-2 is dominated by heuristic features (failing trajectories run 67% longer on average), while claude-code and Forge weight LLM-based semantic scores more heavily.

4.3. Iterative agentic system optimization

We compare taxonomy-guided mutation against an LLM-reflection baseline. Both methods share the AdaEvolve evolutionary backbone, the same seed architecture per benchmark, and matched compute budgets; the only independent variable is whether taxonomy-coded failure feedback is injected into the mutation prompt. We retain the term *LLM Reflection* for the baseline because AdaEvolve's default mutation prompt already contains a free-form "what went

Table 1: Terminal-Bench 2.0 verification accuracy on all 89 tasks. Best-of-5 selection only changes outcomes on the swing-task subset of 30/35/17 tasks (terminus-2/claude-code/Forge); all-pass and all-fail tasks contribute identically across methods. The *LLM-as-a-Verifier* row is the baseline of Kwok et al. (2026). The *MAST* row substitutes the published 14-code MAST taxonomy (Cemri et al., 2025) into ATLAS-Judge's pipeline (forward feature selection, leave-one-task-out CV, identical heuristic substrate); only the taxonomy vocabulary changes. All three configurations use GPT-5.4 as verifier.

Method	terminus-2	claude-code	ForgeCode
Pass@1 (no selection)	61.8%	57.5%	81.8%
LLM-as-a-Verifier (Kwok et al., 2026)	61.8%	61.2%	86.4%
MAST (Cemri et al., 2025)	68.5%	69.0%	88.8%
ATLAS-Judge	**73.0%**	**72.4%**	**89.9%**
Best-of-5 oracle	77.5%	80.5%	89.9%

wrong" critique; the comparison is between adding categorical taxonomy feedback on top of this versus running the baseline alone.

Aggregate improvements across benchmarks. Taxonomy-guided mutation outperforms LLM Reflection on all five benchmarks: +6.7 % generalization on Frontier-CS, +4.0 % on OlympiadBench, +7.5 % on MMLU-Pro, +3.3 % on TheoremQA at n=2 seeds, and +3.5 % F1 on DROP. A controlled comparison against an ADAS reimplementation (Hu et al., 2025) on TheoremQA, an injection-location ablation, and an iterations-to-ceiling observation on Magentic Marketplace are reported in Appendix D.

OlympiadBench: detailed analysis. OlympiadBench is the benchmark with the longest training trace and the largest held-out test set, so we use it to characterize what the +4.0 % gap looks like during search. Figure 3 (b) reports four conditions on the 655-problem test set: the unmodified three-agent seed architecture (*Seed*, 84.6%), the best-scoring architecture from the no-taxonomy evolution run (*No-tax*, 87.9%), the best from a MAST-guided evolution run (*MAST*, 89.5%), and the best from ATLAS (*ATLAS*, 91.9%, registered at iteration 92). All three evolution runs share the AdaEvolve substrate, the same seed, and matched compute; the only independent variable is what failure feedback shapes the mutation prompt.

Figure 3 (a) anchors the test-set numbers in the training trace. The no-taxonomy run reaches 0.40 at iteration 19 and plateaus there: the substrate alone can find architectures of that quality, but cannot identify what to change next. ATLAS hits the same 0.40 ceiling at iteration 9 and continues past it twice more, both breakthroughs traceable to specific judge-fired codes on each parent iteration. First, at iteration 67 (parent: iteration 43), the judge surfaced A.3 *Instruction Compliance Violation*, B.3 *Refiner Fails to Improve*, and C.2 *Edge Case Omission*; the child introduces a VERIFIER agent enforcing final-answer format (A.3), replaces a flat 4-variant pool with a 3-branch tournament of paired candidates adding redundancy against refiner stagnation (B.3), and rewrites the solver prompt to require "edge-case check and invariant sanity checks before producing the final answer" (C.2). Second, iteration 92 (parent: iteration 60), the judge surfaced A.6 *Ground-Truth/Verifier Mismatch*, B.6 *Coordinator Aggregation Mismatch*, and C.3 *Overgeneralization in Domain Prereqs*; the child repositions the planner upstream so each solver expands a distinct named seed strategy (A.6), promotes the verifier's pass from a binary gate to a numerical score boost giving the convergence signal quantitative weight (B.6), and adds a planner-prompt requirement to articulate "core assumptions about the problem class" before solving (C.3). Full architecture differences and verbatim judge outputs are in Appendix G.

4.4. Runtime integration with SWE-agent on SWE-bench Verified Mini

We evaluate ATLAS as runtime feedback inside SWE-agent on SWE-bench Verified Mini[1](50 instances, GPT-5 solver). Five arms vary only the metacognitive scaffolding around the same solver. *Base* runs vanilla SWE-agent. *Reflexion* (Shinn et al., 2023) adds free-text self-reflection at fixed checkpoints. The remaining three arms share the same checkpoint protocol with forced reflection at each checkpoint, differing only in which taxonomy is used and how it is applied: *MAST in-prompt* embeds the published 14-code MAST taxonomy (Cemri et al., 2025) in the system prompt, *ATLAS pattern A* replaces it with ATLAS's induced SWE-bench taxonomy in the system prompt, and *ATLAS pattern B* keeps the same induced taxonomy outside the agent and consults an external judge at each checkpoint.

All three taxonomy arms exceed Reflexion by at least 8 pp, supporting the claim that categorical failure-mode reasoning provides an anchor that free-text self-criticism lacks.

Why the external judge helps. Pattern B reaches 78% versus Pattern A's 70%, an 8 % gap on identical taxonomy

[1]A 50-task random subset of SWE-bench Verified maintained on the HAL leaderboard: HAL SWE-bench Verified Mini.

Table 2: Post-search accuracy per benchmark. ATLAS adds taxonomy-coded failure feedback to the mutation prompt; LLM Reflection is the default AdaEvolve mutation loop. *Pre-Evol* is the seed architecture's score before evolution. Per-benchmark experimental details (seed counts, iteration budgets, train/test splits) are in Appendix C.

Benchmark	Problems	Pre-Evol	LLM Reflection (post)	ATLAS (post)
Frontier-CS (132 unseen)	132	∼20%	26.0%	**32.7%**
OlympiadBench (655 held-out)	655	84.6%	87.9%	**91.9%**
MMLU-Pro (STEM QA)	40	21.3%	35.0%	**42.5%**
TheoremQA (math)	30	39.0%	60.0% (±6.7)	**63.3%** (±3.3)
DROP (discrete reasoning)	30	80.3%	88.2%	**91.7%**

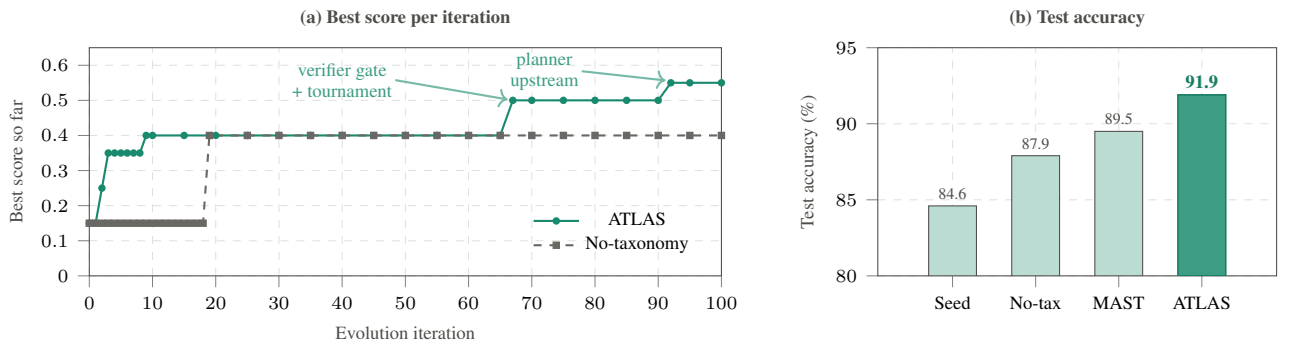

Figure 3: OlympiadBench evolutionary search. **(a)** Best score per iteration on the 20-problem dev set across the 100-iteration run. Both runs start from the same seed architecture (score 0.15) and share the AdaEvolve substrate; only the mutation prompt differs. **(b)** Post-search test accuracy on the held-out 655-problem set. "Seed" denotes the unmodified starting architecture before evolution.

Table 3: Runtime integration on SWE-bench Verified Mini (50 instances, GPT-5 solver in SWE-agent). ATLAS taxonomies (Arms 4 and 5) additionally incur a one-time generation cost of 8 LLM calls per domain, amortized across all subsequent runs.

Arm	**Resolved**	**Rate**
Arm 1: Base	25/50	50%
Arm 2: Reflexion (free-text)	30/50	60%
Arm 3: MAST in-prompt	34/50	68%
Arm 4: ATLAS pattern A	35/50	70%
Arm 5: ATLAS pattern B	**39/50**	**78%**

and identical checkpoint protocol, with only the locus of classification differing. The aggregate gap masks a sparse pattern of disagreement: the two patterns reach the same outcome on 44 of 50 instances, and on the six where they differ, Pattern B wins five (three on instances no other arm resolved) and Pattern A one. A representative case: Arm 4 produced an overcomplicated patch and its internal reflection failed to flag the misdiagnosis, since the agent's own narrative of what it had done felt consistent. Pattern B's judge raised the concern at three consecutive checkpoints, "the real bug is order-dependent collection . . . not simply filtering by field type", and required a deterministic failing unit test before any further edits, redirecting the agent to a 2-line fix that passed. The mechanism: when classification happens inside the agent's own context, the agent's narrative of its trajectory contaminates its evaluation of that trajectory; the external judge sees only the trajectory, not the reasoning that produced it, and so can name failures the agent's self-account would smooth over; this is the self-evaluation discussed in §3.4. A full case-by-case breakdown is in Appendix I.

5. Discussion

Why a categorical structure? A natural objection to ATLAS is that the taxonomy is just dressing free-text critiques in structured form. We argue the value of the categorical form lies elsewhere. Structured codes are a stable vocabulary that supports inter-annotator agreement (INTER-ANNOTATOR AGREEMENT reaches $\kappa = 0.725$ on TRAIL, where free-text annotation cannot even be measured), and they enable the two distinct deployment patterns of §3.4, in-prompt checklist and external judge, that free-text reflection cannot support. The categorical form is best understood as a substrate that makes downstream uses possible, not as a stylistic improvement to the mutation-prompt content itself.

he OlympiadBench mechanism analysis (§4.3) shows that ATLAS-driven mutation addresses A-code (system structure) and B-code (role behaviour) failures across both breakthroughs, introducing verifiers, planners, and aggregation rules in response to specific judge-fired codes. The C-code domain-reasoning failures fire on most judge calls but are not addressed by either breakthrough's architectural changes. This is unsurprising: an architectural search rewires the system around the underlying model, but does not improve the model's mathematics. The 8 pp residual gap on OlympiadBench, and analogous ceilings we expect on other reasoning-heavy domains, plausibly reflect this architectural-vs-parametric distinction. The natural next step is to investigate whether ATLAS codes can serve as a structured reward signal for parametric methods like RL or DPO: each trajectory carries multiple categorical labels rather than a single number, and because the codes are induced from traces rather than hand-crafted, the signal scales to new domains without manual annotation. Whether this denser, more transferable signal actually translates into better reasoning under realistic compute is a separate research question that we leave to future work.

6. Conclusion

ATLAS treats domain-specific failure taxonomies as infrastructure, substrates other procedures consume, rather than as descriptive end products. The same induced taxonomy supports trajectory verification, drives architectural evolution, and serves as a runtime reflection skill, with measurable gains across all three usages on benchmarks spanning competitive programming, mathematics, software engineering, and agentic computer use. The categorical form does not, by itself, produce better critique than free-text; what it provides is a stable vocabulary that downstream procedures can act on consistently. We see induced taxonomies as a missing intermediate representation in agent-system research, sitting between raw traces and the procedures that improve them.

References

Agrawal, L. A., Tan, S., Soylu, D., Ziems, N., Khare, R., Opsahl-Ong, K., Singhvi, A., Shandilya, H., Ryan, M. J., Jiang, M., et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

Anthropic. Claude code. https://docs.anthropic.com/en/docs/claude-code/overview, 2025. Accessed: 2026-05-07.

Bansal, G., Hua, W., Huang, Z., Fourney, A., Swearngin, A., Epperson, W., Payne, T., Hofman, J. M., Lucier, B., Singh, C., Mobius, M., Nambi, A., Yadav, A., Gao, K., Rothschild, D. M., Slivkins, A., Goldstein, D. G., Mozannar, H., Immorlica, N., Murad, M., Vogel, M., Kambhampati, S., Horvitz, E., and Amershi, S. Magentic marketplace: An open-source environment for studying agentic markets. *arXiv preprint arXiv:2510.25779*, 2025.

Cemri, M., Pan, M. Z., Yang, S., Agrawal, L. A., Chopra, B., Tiwari, R., Keutzer, K., Parameswaran, A., Klein, D., Ramchandran, K., Zaharia, M., Gonzalez, J. E., and Stoica, I. Why do multi-agent llm systems fail? In *Advances in Neural Information Processing Systems*, 2025.

Cemri, M., Agrawal, S., Gupta, A., Liu, S., Cheng, A., Mang, Q., Naren, A., Erdogan, L. E., Sen, K., Zaharia, M., et al. Adaevolve: Adaptive llm driven zeroth-order optimization. *arXiv preprint arXiv:2602.20133*, 2026.

Cheng, C.-A., Nie, A., and Swaminathan, A. Trace is the next AutoDiff: Generative optimization with rich feedback, execution traces, and LLMs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Deshpande, D., Gangal, V., Mehta, H., Krishnan, J., Kannappan, A., and Qian, R. TRAIL: Trace reasoning and agentic issue localization. *arXiv preprint arXiv:2505.08638*, 2025.

Dua, D., Wang, Y., Dasigi, P., Stanovsky, G., Singh, S., and Gardner, M. DROP: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In *Proceedings of NAACL-HLT*, pp. 2368–2378, 2019.

Fernando, C., Banarse, D., Michalewski, H., Osindero, S., and Rocktäschel, T. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*, 2023.

Gu, J., Jiang, X., Shi, Z., Tan, H., Zhai, X., Xu, C., Li, W., Shen, Y., Ma, S., Liu, H., et al. A survey on llm-as-a-judge. *The Innovation*, 2024.

Guo, Q., Wang, R., Guo, J., et al. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *International Conference on Learning Representations (ICLR)*, 2024.

Hu, S., Lu, C., and Clune, J. Automated design of agentic systems. In *International Conference on Learning Representations (ICLR)*, 2025.

Jia, A. S., Huang, D., Vytla, N., Yoo, S. W. W., Choudhury, N., Sen, S., Mitchell, J. C., and Datta, A. What is your agent's GPA? a framework for evaluating agent goal-plan-action alignment. *arXiv preprint arXiv:2510.08847*, 2025.

Khattab, O. et al. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *International Conference on Learning Representations (ICLR)*, 2024.

Kwok, J., Li, S., Atreya, P., Liu, Y., Pavone, M., Stoica, I., and Mirhoseini, A. Llm-as-a-verifier: A general-purpose verification framework, 2026. Notion Blog.

Landis, J. R. and Koch, G. G. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.

Lightman, H., Kosaraju, V., Burda, Y., et al. Let's verify step by step. In *International Conference on Learning Representations (ICLR)*, 2024.

Madaan, A. et al. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Mang, Q., Chai, W., Li, Z., Mao, H., Zhou, S., Du, A., Li, H., Liu, S., Chen, E., Wang, Y., et al. Frontier-CS: Evolving challenges for evolving intelligence, 2025. URL https://arxiv.org/abs/2512.15699.

Merrill, M. A., Shaw, A. G., Carlini, N., Li, B., Raj, H., Bercovich, I., Shi, L., Shin, J. Y., Walshe, T., Buchanan, E. K., et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

Novikov, A., Balog, M., Chaudhuri, S., et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Pan, M. Z., Arabzadeh, N., Cogo, R., et al. Measuring agents in production. *arXiv preprint arXiv:2512.04123*, 2026.

Romera-Paredes, B., Barekatain, M., Novikov, A., et al. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024.

Shinn, N., Cassano, F., Gopinath, A., et al. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Snell, C., Lee, J., Xu, K., and Kumar, A. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

TailcallHQ. ForgeCode: An open-source multi-agent coding harness. https://github.com/tailcallhq/forgecode; https://forgecode.dev, 2026. Top-ranked open-source agent on Terminal-Bench 2.0 (Pass@1 81.8% with the Forge harness).

Wang, Y., Ma, X., Zhang, G., Ni, Y., Chandra, A., Guo, S., Ren, W., Arulraj, A., He, X., Jiang, Z., Li, T., Ku, M., Wang, K., Zhuang, A., Fan, R., Yue, X., and Chen, W. MMLU-Pro: A more robust and challenging multi-task language understanding benchmark. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2024.

Yuan, S. et al. EvoAgent: Towards automatic multi-agent generation via evolutionary algorithms. *arXiv preprint arXiv:2406.14228*, 2024.

Yuksekgonul, M., Bianchi, F., Boen, J., et al. TextGrad: Automatic "differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

Zhang, J., Xiang, J., Yu, Z., et al. AFlow: Automating agentic workflow generation. In *International Conference on Learning Representations (ICLR)*, 2025.

Zheng, L., Chiang, W.-L., Sheng, Y., et al. Judging LLM-as-a-judge with MT-Bench and chatbot arena. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Zhou, A. et al. Language agent tree search unifies reasoning, acting, and planning in language models. In *International Conference on Machine Learning (ICML)*, 2024.

Zhuge, M. et al. GPTSwarm: Language agents as optimizable graphs. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2024.

A. Generated Taxonomy Examples

A.1. Frontier-CS Final Taxonomy — Illustrative Codes

The Frontier-CS run's final taxonomy contains 25 codes (5A/13B/7C). The tables below show a curated, illustrative subset chosen to span the failure-mode space; representative names and frequencies are included to convey the texture of the discovered failure modes. The complete machine-generated catalogue (with auto-generated long-form names and per-iteration provenance) is released alongside the code, in the canonical taxonomy JSON format.

Table 4: Illustrative system-level failure codes (5 in final taxonomy).

Code	Name	Description	Freq.
A.1	Token_Exhaustion	Agent exceeds context window mid-reasoning	8.2%
A.2	Payload_Corruption	Generated code is malformed or incomplete	12.1%
A.3	Tool_Failure	Judge/compiler service error	3.4%
A.4	Budget_Exhaustion	All submissions used without progress	15.7%
A.5	Stagnation_Loop	Same approach retried $\geq$3 times	9.3%

Table 5: Illustrative role-specific failure codes (13 in final taxonomy).

Code	Name	Description	Freq.
B.1	Solver_Bias	Defaults to familiar algorithm ignoring analysis	11.4%
B.2	Logic_Inconsistency	Reasoning contradicts implemented code	5.3%
B.3	Syntax_Generation	Produces invalid C++	7.8%
B.5	Feedback_Ignorance	Ignores judge feedback on subsequent attempts	6.2%
B.7	Superficial_Review	Reviews without checking edge cases	4.1%
B.9	Classification_Error	Routes problem to wrong specialist	3.7%

Table 6: Illustrative domain-reasoning failure codes (7 in final taxonomy).

Code	Name	Description	Freq.
C.1	Algorithm_Mismatch	Wrong algorithm family for problem	18.3%
C.3	State_Tracking	DP state misses dimension	9.7%
C.5	Constraint_Violation	Solution violates problem bounds	6.1%
C.7	Graph_Traversal	Incorrect BFS/DFS order	5.4%
C.11	Integer_Overflow	Intermediate overflow	4.8%
C.14	Complexity_Violation	$O(n^2)$ when $O(n \log n)$ needed	7.2%

A.2. TheoremQA Taxonomy (13 codes)

The TheoremQA evolution pipeline evaluates 30 graduate-level math problems with a flat solver-verifier architecture (no role specialization, hence no role-specific B-codes; see Cross-domain comparison in Section A.7). The five most-frequent codes from the run are listed below; the dominant failure (`Premature_Reasoning_Truncation`, fired on 50% of failures) names the pattern where the solver halts a multi-step derivation mid-calculation and prints an intermediate quantity as the final answer. The remaining failures distribute across context exhaustion, unit handling, numerical precision, and instruction compliance.

A.3. Terminal-Bench Verification Criteria

The Terminal-Bench taxonomy yields scoring criteria that target failure modes specific to the benchmark's task categories (security, machine learning, system administration, data science). The table below shows representative examples of how an induced code becomes an actionable verification criterion for ATLAS-Judge: each entry takes the form "check whether the agent did X," where X is grounded in a category-specific failure mode the taxonomy generator surfaced from traces. These criteria, together with the universal A-code criteria (premature termination, context exhaustion, repetitive looping), constitute the LLM-scoring portion of the Best-of-5 selection pipeline reported in Table 1.

Table 7: Top five TheoremQA failure codes by firing frequency. Generated by the pipeline from the evolution-run traces with no human authoring.

Code	Name	Freq.	Impact
C.3	Premature_Reasoning_Truncation	50%	Critical, causes 7/14 failures
A.1	Token_Exhaustion	14%	High
C.1	Unit_Mismatch	12%	Medium
C.5	Precision_Loss	8%	Medium
A.4	Instructional_Compliance	7%	Low

Table 8: Representative ATLAS-derived verification criteria on Terminal-Bench 2.0, grouped by task category. Each criterion targets a domain-specific failure pattern that a generic "did the output match?" rubric would miss.

Category	Criterion	What It Catches
Security	Exploit Completion Verification	Agent abandons exploit and suggests defenses
Security	Defensive Pivoting Detection	Subtle shift from offensive to defensive posture
ML	Training Stall Detection	No error message, just timeout
Sys-Admin	Service State Verification	systemctl fails in Docker, agent unaware
Data Science	Output Format Alignment	Column name case sensitivity

A.4. OlympiadBench taxonomy

The taxonomy used in the OlympiadBench evolution (Section 4.3, Appendix G) is generated from training-iteration traces and contains 16 codes total (6 system-level, 5 role-specific, 5 domain-reasoning). The taxonomy was refined nine times over the 97-iteration run; the final version was active from approximately iteration 96. Tables below show the top codes by firing frequency from each category. Frequencies are measured across the 36 judge-firing iterations; a code is counted as fired on an iteration if it appears in that iteration's judge output.

Table 9: Top system-level (A) codes in the OlympiadBench taxonomy.

Code	Name	Description	Freq.
A.3	Instruction Compliance Violation	Outputs violate required final-answer format or task constraints	100.0%
A.1	Behavioral Anomaly: Stagnation	Non-convergent iteration; recurrent non-progressive patterns	91.7%
A.4	Evaluation Feedback Misinterpretation	Agents fail to incorporate evaluator critique; refiners regress after feedback	80.6%
A.6	Ground-Truth / Verifier Mismatch	Divergent final answers across checkers for the same problem	75.0%
A.2	Execution Error	Timeouts, crashes, or resource exhaustion	38.9%

A.5. SWE-bench Verified taxonomy

The taxonomy used in Arms 4 and 5 of the runtime-integration experiment (§4.4, Appendix H) is generated from SWE-bench Verified trajectories and contains 30 codes total (8 A, 11 B, 11 C). Tables below show the top codes by firing frequency from each category. Frequencies are measured across 50 Mini instances; a code is counted as fired on an instance if it appears in any judge call on that instance's trajectory.

A.6. Per-pair Jaccard Matrix Across Six Domains

Table 15 reports per-pair Jaccard similarity over the universal-category set for all six evolution domains. Each domain's category-A (system-level) failure codes are mapped to a shared catalogue of universal failure tokens (context exhaustion, premature termination, looping, format violation, instruction non-compliance, handoff loss, tool/runtime error, logical inconsistency, unsupported claim, numerical precision); domain-specific A-codes without a universal counterpart are kept as singletons. Jaccard is then computed on these canonical sets.

Table 10: Top role-specific (B) codes in the OlympiadBench taxonomy.

Code	Name	Description	Freq.
B.2	Solver Provides Low-Quality Reasoning	Solver outputs lack derivation, justification, or domain-specific reasoning	75.0%
B.3	Refiner Fails to Improve	Refiners regress after critique; persist with placeholders	75.0%
B.6	Coordinator Aggregation Mismatch	Aggregated results conflict due to divergent final-answer lines	66.7%
B.1	Coordinator Drops Feedback	Coordinator fails to propagate feedback to solvers in next round	63.9%
B.4	Refiner Overcorrects/Hijacks	Refiner shifts approach, ignoring established protocol	50.0%

Table 11: Top domain-specific (C) codes in the OlympiadBench taxonomy.

Code	Name	Description	Freq.
C.2	Edge Case Omission	Edge cases enumerated but not validated within the same turn	94.4%
C.4	Algebraic Manipulation Gap	Incomplete or flawed algebraic steps; missing justifications	91.7%
C.5	Combinatorics Counting Misreasoning	Counting arguments rely on independent choices or naive decompositions	88.9%
C.1	Sign/Difference Misreasoning	Errors in sign-handling or difference-based derivations	80.6%
C.3	Overgeneralization in Domain Prereqs	Rules or bounds asserted without full justification	80.6%

A.7. Cross-domain taxonomy comparison

How much of the failure structure of agent systems is universal across domains, and how much is domain-specific? Table 16 compares the taxonomies the pipeline generates for the six evolution domains.

Three patterns emerge. *C-codes are entirely domain-specific.* Algorithm_Mismatch (competitive programming), Physical_Law_Violation (STEM QA), and Negotiation_Leverage_Failure (economic simulation) are failure modes no fixed taxonomy could anticipate; they require the trace-driven generation the pipeline performs. *B-codes track architectural structure.* They appear only in multi-agent systems with specialized roles (Frontier-CS: 13 B-codes, Terminal-Bench: 16) and are absent in flat pipelines (TheoremQA, MMLU-Pro, DROP). The taxonomy thus reflects the architecture being evaluated rather than imposing a fixed schema. *A-codes form a partial universal backbone.* Context exhaustion and premature termination appear in every domain. Quantitatively, restricted to category-A codes, mean pairwise Jaccard similarity is $J = 0.47$ across all $\binom{6}{2} = 15$ domain pairs (Table 15); the QA-style domains cluster tightly (mean 0.65), while Frontier-CS is the most distant (mean 0.16 against the others) because its A-codes are dominated by tool-and-dependency failures absent from QA-style pipelines.

B. Terminal-Bench Configurations

Table 1 in the main text reports ATLAS-Judge results for three agent/model configurations on Terminal-Bench 2.0. Table 17 summarizes the per-configuration setup and aggregate annotation statistics. The three configurations span distinct verification regimes (heuristic-dominated vs. LLM-dominated) under a single GPT-5.4 verifier (genuine cross-model design with all three agents based on Claude Opus 4.6). Taxonomy annotations are consistent across configurations: A.7 (Premature_Termination) is the most frequent failure code in every one.

A.7 (Premature_Termination) is the dominant failure code in every configuration. B.11 (Superficial_Solution) and B.13 (Validation_Not_End_to_End) are second and third most frequent. This consistency across agent frameworks and underlying models indicates that the taxonomy captures genuine structural failure patterns rather than artifacts of any single system.

Table 12: System-level (A) codes that fired on the SWE-bench Verified Mini benchmark. The remaining five A codes (A.1, A.2, A.3, A.7, A.8) did not fire on any of the 50 instances.

Code	Name	Description	Freq.
A.5	Misread root cause	Misdiagnosed mechanism; implemented superficial or irrelevant change	82.0%
A.4	Wrong abstraction layer	Changes in a related but incorrect layer (manager vs queryset, etc.)	18.0%
A.6	Invalid reproduction	Reproduction sidesteps the real failing scenario	12.0%

Table 13: Top role-specific (B) codes in the SWE-bench Verified taxonomy.

Code	Name	Description	Freq.
B.4	Over-broad behavior change	Wide-ranging logic changes beyond the minimal fix	42.0%
B.3	Incompatible API change	Altered public signatures or behaviors that break callers	16.0%
B.5	Ineffective layer fix	Fix placed in a code path not used by the failing behavior	14.0%
B.11	Over-narrow checks	Type/condition checks too specific, missing valid variants	14.0%
B.2	Partial path coverage	Changed only some required code paths (e.g., `__eq__` not `__ne__`)	2.0%

C. Implementation Details

C.1. Taxonomy generation pipeline

The taxonomy generator (§3.1) runs as a sequence of eight LLM calls plus regex-only pre-processing within Step 2. Each step's outputs feed the next; the three category generators (Steps 3–5) run in parallel from shared structural inputs. The total per-generation cost is $C_{\text{gen}} = 8$ LLM calls (Appendix E).

Step 1 — Domain analysis. A stratified sample of 8 traces (capped at 3,000 characters each, balanced across source MAS) is shown to the LLM together with metadata hints (benchmark name, model name, sample task descriptions). The prompt asks for the domain characterization: domain name and content type, subdomains, key terminology paired with the errors associated with each term, common error patterns with detection hints, and correctness criteria. Output JSON seeds two later steps: the *common error patterns* and *terminology error associations* feed Step 5 (domain-reasoning generation), and the *subdomain* list drives the C-code coverage check in Step 8.

Step 2 — Role and topology discovery. A regex pass over each trace's raw trajectory extracts agent markers (`[AgentName]`, `### Agent`, `**AgentName:**`) to produce the set of discovered agents, plus tool-call patterns (`[TOOL CALL]`, `Action:`) yielding the interaction style (single-shot / multi-turn / tool-using). A regex aggregation over *every* trace in the dataset (not just the sample) computes behavioral signals: truncation rate (trajectories ending with operators, opening delimiters, ellipses, or trailing commas), empty output rate, error incidence with error-type breakdown (`traceback`, `exception`, `timeout`, `rate limit`, `token limit`), refusal incidence, within-trace repetition, abrupt-ending detection, tool-error rates, and trajectory-length statistics with outlier IDs. A single LLM call then takes the discovered agents (with behavior excerpts), the role canonical definitions, the architecture-relevant trace sample (8 longer traces at 4,000 characters), and the regex signals, and produces: agent-to-role classification (`solver` / `checker` / `refiner` / `coordinator`); the trace format markers; the architecture topology (sequential, parallel, hierarchical, debate, hybrid); the verification pattern (self-verify, peer-verify, dedicated-checker, consensus, none); the termination owner; and the critical handoffs.

Step 3 — System-level (A) code generation. Inputs: the architecture block from Step 2 (topology, handoffs, verification pattern, termination owner), the role list, the behavioral-signal aggregate, and a stratified sample of up to 50 traces. The LLM is asked to generate codes covering both architectural risk (what could go wrong at the system level given this topology,

Table 14: Top domain-reasoning (C) codes in the SWE-bench Verified taxonomy.

Code	Name	Description	Freq.
C.1	Ad-hoc only	Used a custom script instead of the project's full test suite	90.0%
C.5	No proper reproduction	Proceeded without successfully reproducing the failure	36.0%
C.3	No syntax/build check	Submitted without at least importing or building the patched module	20.0%
C.2	Didn't revalidate final diff	Failed to run tests after the last edit	12.0%
C.4	Irrelevant subset tests	Ran only a narrow or unrelated subset of tests	6.0%

Table 15: Per-pair Jaccard similarity over A-code structural backbones for all 15 domain pairs. Mean $= 0.47$. The QA-style domains (MMLU-Pro, TheoremQA, DROP, Marketplace) cluster tightly (pairwise Jaccard 0.50–1.00, mean 0.65). Frontier-CS is consistently the most distant domain (mean 0.16 against the others) because its A-codes are dominated by tool-and-dependency failures (Context_Window_Overflow, Recursive_Computation_Exhaustion, Dependency_Injection_Failure, Tool_Interaction_Error) absent from QA-style pipelines. Terminal-Bench sits between the two groups (mean 0.55 against the QA cluster, 0.25 against Frontier-CS), consistent with its mixed agentic-coding nature.

	Frontier-CS	Terminal-B.	MMLU-Pro	TheoremQA	DROP	Marketplace
Frontier-CS	1.00	–	–	–	–	–
Terminal-Bench	0.25	1.00	–	–	–	–
MMLU-Pro	0.17	0.63	1.00	–	–	–
TheoremQA	0.20	0.50	0.50	1.00	–	–
DROP	0.00	0.63	0.67	0.50	1.00	–
Marketplace	0.17	0.63	1.00	0.50	0.67	1.00

regardless of whether observed) and empirically grounded failures (codes anchored in the regex signals and trace sample). After the call, a non-LLM filter drops any code whose name contains a role identifier (those should have been B-codes; Step 7 catches any that slip through).

Step 4 — Role-specific (B) code generation. Active roles (those with at least one discovered agent) are filtered in first; if none survive, a fallback regex re-extraction runs over metadata and trajectories. Inputs to the LLM call: active roles with their definitions and architecture context, plus a role-grouped trace sample (50 traces per role). The prompt asks role-by-role: "what are the ways an agent doing this role could do its job incorrectly?" Each role gets generated guidance text with example phrasings (e.g., "what ways can an agent whose job is to validate solutions do that job incorrectly?"). Every B-code is required to carry an explicit `applies_to_role` field referencing one of the discovered roles.

Step 5 — Domain-reasoning (C) code generation. Inputs: the *common error patterns* and *terminology error associations* from Step 1, plus a trace sample. The prompt extends each seed error pattern into one or more codes (e.g., for math: `unproven_lemma_use`, `casework_omission`; for SWE-bench: `partial_path_coverage`, `wrong_abstraction_layer`) and additionally surfaces reasoning flaws visible in the traces but not present in the seed patterns. No role-name sanitization is needed since role context is not included.

Step 6 — Cross-category deduplication. Looks for codes that describe the same concept across different categories ($A \leftrightarrow B$, $A \leftrightarrow C$, $B \leftrightarrow C$). The prompt enforces strict boundary rules: a B-code for one role is never a duplicate of a B-code for a different role; a system-level A-code and a role-specific B-code describing the same event are not duplicates because they target different intervention points. Each detected duplicate produces a `keep_in` / `remove` decision, after which the affected lists are filtered.

Step 7 — Per-category structural validation. Each code is checked against its category's structural requirements: A-code names must not contain role identifiers; B-code names must contain a role identifier and the code must carry an `applies_to_role` field; C-code names must not contain role identifiers. Violations can be repaired in three ways: rename the code, change the `applies_to_role` value, or move the code to the correct category.

Table 16: Generated taxonomy size per domain. The A column counts system-level codes (architectural failures), B counts role-specific codes (only meaningful for multi-agent architectures), and C counts domain-reasoning codes (task-specific logic errors). The representative C-code illustrates the kind of failure that would be invisible to a generic evaluation rubric.

Domain	A	B	C	Representative C-code
Frontier-CS	5	13	7	Complexity_Class_Constraint_Mismatch
OlympiadBench	6	5	5	Combinatorics_Counting_Misreasoning
Terminal-Bench	10	16	18	Defensive_Pivoting
MMLU-Pro	6	0	10	Physical_Law_Violation
TheoremQA	4	0	9	Premature_Reasoning_Truncation (50%)
DROP	4	0	10	Arithmetic_Operation_Error
Marketplace	6	2	6	Negotiation_Leverage_Failure

Table 17: Terminal-Bench 2.0 agent/model configurations evaluated with ATLAS-Judge. Avg Pass Rate is the fraction of trials that pass across all 89 tasks. Swing Tasks are tasks with mixed pass/fail across 5 trials (the only tasks where verification adds value). Fired Codes is the number of taxonomy code firings across all failing trajectories.

Agent	Model	Trials	Avg Pass Rate	Swing Tasks	Fired Codes
terminus-2	Claude Opus 4.6	445	62.9%	30	1,110
claude-code	Claude Opus 4.6	445	58.0%	35	1,282
Forge	Claude Opus 4.6	445	81.8%	17	–

Step 8 — Quality and coverage gate. A non-LLM heuristic pass runs first and rejects codes with placeholder strings (e.g., `field_N`, `TBD`), missing `detection_heuristics` or `when_to_use` fields, role-name violations the prior step missed, or definitions shorter than a minimum length. The checker also verifies category coverage: A-codes must collectively span seven failure-pattern groups (output issues, context/memory, communication, behavioral, execution, instruction, tool/API); every active role must have at least one B-code; every subdomain identified in Step 1 must have at least one C-code. A final embedding-based overlap check flags pairs of codes whose descriptions are too similar. A single LLM repair call then addresses any flagged issues, rewriting placeholder text, repairing overlap, filling missing fields.

C.2. Hyperparameters

The taxonomy generator and the surrounding evolutionary loop expose a small set of hyperparameters; values used in our experiments are listed below with the rationale for each. The warmup rounds, stagnation window, and stagnation threshold control when the loop generates and refines the taxonomy; the population, island, and migration settings inherit standard island-model genetic-algorithm defaults; and the feedback-injection parameter K controls how many of the most-frequent codes are surfaced into each mutation prompt.

Table 18: Hyperparameters and the rationale for each value.

Parameter	Value	Rationale
Warmup rounds (N_{warm})	5	Balances trace pool quality vs. compute; 3 produced thin taxonomies
Stagnation window (w)	10 iter.	One full population evaluation cycle; w=3 fired during exploration
Stagnation threshold (ϵ)	0.10	Set above per-iteration noise floor (0.02–0.05)
Min refine interval ($r_{\min}$)	10 iter.	Prevents back-to-back refinement on the same trace pool
Population size	20	Standard for island-model GA
Number of islands	3	Diversity without fragmentation
Migration interval	8 iter.	Allow island divergence before sharing
Feedback codes injected (K)	5–7	Top codes by frequency
TheoremQA problems	30	Graduate-level math
Random seeds	42, 123, 456	Standard reproducible seeds

C.3. Seed Architecture

Every evolution run begins from the same minimal seed architecture: a flat three-agent pipeline that the search is free to elaborate on. Starting from the same seed across the LLM-reflection and ATLAS runs ensures the comparison isolates the contribution of taxonomy-coded feedback and not an architectural prior. The seed is intentionally simple so that the verifier insertion and planner addition observed in the OlympiadBench deep-dive (Section 4.3) are mutations the search itself discovers.

All experiments start from the same simple seed:

- **Agents**: analyzer, solver, verifier (3 agents)
- **Topology**: analyzer $\rightarrow$ solver $\rightarrow$ verifier $\rightarrow$ evaluate $\rightarrow$ refine (loop)
- **Strategy**: 1 analysis pass + iterative solve-verify-refine loop

D. Extended Ablation Details

D.1. Injection-Location Ablation on TheoremQA

We ask a design question on TheoremQA: where in the mutation prompt should failure feedback appear? We fix the feedback format (full ATLAS taxonomy) and vary three positions: appended to evaluation results, embedded as inline comments in the evolved code, or placed in the system message.

Table 19: Injection-location ablation on TheoremQA (n=30 problems, single seed, seed 42). The position of failure feedback in the mutation prompt produces a 16.6% gap between best and worst injection.

Injection position	Accuracy
Appended to evaluation results	**63.3%**
Inline comments in code	53.3%
System message	46.7%

The 16.6% gap between best and worst injection (63.3% vs. 46.7%) makes injection placement a meaningful design lever for taxonomy-guided evolution.

D.2. ADAS Comparison on TheoremQA

We compare ATLAS against our reimplementation of ADAS (Hu et al., 2025), the closest prior method for automated agent design, on the 30-problem TheoremQA evaluation. The original ADAS targets single-agent systems; we adapted it for multi-agent evolution while preserving its core archive-and-meta-agent design. Both methods received compute budgets of 40 generations on the same model. ADAS's best evolved design ("Forensic Multi-Expert Triangulation") solved 17/30 problems getting 56.7% whereas ATLAS multi-seed mean reported in Table 2 is 63.3%, giving a +6.6% gap in favor of ATLAS.

D.3. Frontier-CS Architecture Discovery

On the Frontier-CS evolution run reported in Table 2, the ATLAS-guided search discovered a multi-agent architecture with seven specialized roles and regex-based routing between them, alongside algorithm templates emitted by the planner. The matched LLM-reflection baseline run discovered a 3-function pipeline. The ATLAS run also produced fewer failed mutations (7 of 54 attempted) than the baseline (22 of 38).

D.4. Magentic Marketplace: Iterations-to-Ceiling

On Magentic Marketplace (Bansal et al., 2025) (10 customers, 30 businesses; single-seed simulation), both methods converge to the same ceiling ($\sim$100% on the simulation objective), with ATLAS reaching it at iteration 1 versus iteration 6 for the LLM-reflection baseline.

D.5. Inter-annotator agreement protocol

After the 8-step pipeline produces a draft taxonomy (Appendix C.1), the codes are validated against held-out traces through a multi-phase deliberative annotation protocol. A panel of four LLM annotators (identical model, distinct seed identities) labels the same set of held-out trajectories; agreement metrics determine whether the draft taxonomy is sufficiently well-defined for downstream use, and disagreement patterns surface codes that need refinement.

Five-phase protocol. Each annotation round runs five sequential phases. *Phase 1 (independent error discovery):* each annotator independently identifies potential failure points in the trace. *Phase 2 (error reconciliation):* annotators see what others flagged and deliberate; an error advances if at least two annotators proposed it and a majority confirms after deliberation. *Phase 3 (high-level failure typing):* confirmed errors are categorized as Type A (infrastructure failure), Type B (role-level failure, at most one per agent), or Type C (reasoning flaw, multiple allowed). *Phase 4 (code assignment):* typed errors receive specific taxonomy codes with format validation. *Phase 5 (code deliberation):* disagreements at the code level trigger a multi-turn discussion bounded at two rounds.

Supporting infrastructure. A shared knowledge base accumulates across rounds: decision rules from past disagreements, anchor examples from cases where annotators converged, confusion-pair tracking (frequently disagreed-upon code pairs), and an embedding-based check for semantically overlapping codes. When a disagreement involves a known confusion pair, annotators receive disambiguation guidance based on observable trace evidence rather than arbitrary tie-breaking.

Targets and termination. The protocol targets Cohen's $\kappa \geq 0.75$ at the area level (substantial agreement under Landis-Koch (Landis & Koch, 1977)) with a coverage floor of 0.70. Each round samples five stratified traces; the protocol runs up to five rounds, terminating early when both targets are met. Empirical results validating the Inter-Annotation Agreement Results against the TRAIL human-annotated benchmark are in Appendix F.

When κ targets are not met, ATLAS proposes targeted edits to the taxonomy: merging codes flagged by the embedding-based overlap check, adding codes for failure patterns the panel identified that fall outside the current taxonomy, or relabeling codes flagged by the confusion-pair tracker as having ambiguous boundaries. Confirmed edits are applied and the next round runs against the revised taxonomy.

E. Cost Analysis

ATLAS adds LLM-call overhead to each of its three downstream usages. The cost shape differs by usage, verification is per-task, evolutionary search is per-iteration, runtime integration is per-checkpoint, and a single per-domain taxonomy generation cost is amortized across all three. We analyze each usage in turn.

Shared cost: taxonomy generation. The taxonomy generator runs once per domain on accumulated traces, issuing $C_{\text{gen}} = 8$ LLM calls (one per pipeline step in Appendix C.1). This cost is paid once and the resulting taxonomy is reused indefinitely across all three usages on that domain.

E.1. Usage 1 — Trajectory verification

Notation. Let N be the number of candidate trajectories to score for a single task, and $S \in [5, 10]$ be the number of features the per-agent forward selector retains from the 27 candidates (6 LLM prompts + 20 heuristic + 1 pairwise).

Cost breakdown. The 20 heuristic features are regex-based and have negligible LLM cost. The dominant per-task cost comes from the LLM scoring prompts and pairwise comparisons that survive feature selection: at most $6N + \binom{N}{2}$ LLM calls per task in the worst case (all six LLM prompts and the pairwise signal selected), typically substantially less because the selector retains only S features and most agents end up with a heuristic-heavy or LLM-heavy, not both, selected feature set.

One-time setup cost. The forward feature selector is trained once per agent under leave-one-task-out cross-validation on a small calibration set; this is a one-time cost per agent type, not per deployment.

Deployment-time cost. At deployment, only the S selected features are evaluated per trajectory. For agents whose selector picks a heuristic-heavy subset, deployment cost is near-zero LLM-wise; for agents whose selector picks an LLM-heavy subset, deployment cost is $O(SN)$ calls per task.

E.2. Usage 2 — Evolutionary architecture search

Notation. Let N be the number of evolution iterations in a run, T the number of tasks in the evaluation set per iteration, and K the number of LLM calls a candidate makes per task during evaluation. Total evaluation cost over a run is $C_{\text{eval}} = N \cdot T \cdot K$ calls. Let ρ be the fraction of iterations whose parent is novel (not previously judged); empirically $\rho \approx 0.4$ in our runs because parents repeat across iterations under island-model resampling. Let r be the number of refinement events fired during the run.

Cost breakdown. Two overhead components on top of the shared C_{gen}: (i) *Two-pass judge:* fires only when a newly-evaluated parent is sampled (judge results are cached per parent); Pass 1 issues one call per task in parallel and Pass 2 is one synthesis call, so $C_{\text{judge}} = \rho \cdot N \cdot (T + 1)$. (ii) *Refinement:* a small number of LLM calls per refinement event proposes merge/add/relabel edits over the current taxonomy, $C_{\text{refine}} = O(r)$.

Total overhead. The full evolution-time ATLAS overhead is $C^{\text{evo}}_{\text{atlas}} = C_{\text{gen}} + C_{\text{judge}} + C_{\text{refine}}$. The dominant term is C_{judge}, which scales linearly with N but is suppressed by $\rho < 1$ (only novel parents trigger the judge) and by parallelism (Pass 1 calls run concurrently, so wall-clock cost is much smaller than total call count). The ratio of overhead to evaluation cost in the limit of large N is dominated by ρ/K, which is small because K is much larger than one in any non-trivial multi-agent architecture.

No deployment-time overhead. The costs above are paid once during evolution. The output of the loop is a Python program representing a multi-agent system; that program is what gets shipped and re-evaluated at test time. The taxonomy generator, the judge, and the refinement machinery are not invoked when the evolved program runs on new tasks: they exist only to guide mutation during search. The test-set numbers we report in Section 4.3 are produced by executing the evolved program with no ATLAS machinery in the loop. The comparison against the AdaEvolve baseline at test time is therefore between two programs of identical inference-time cost.

E.3. Usage 3 — Runtime integration

Notation. Let M be the number of checkpoints the harness fires per task (typically two to four: one per declared sub-task completion plus the mandatory pre-submission checkpoint), and let L be the number of tasks the agent solves over its deployment lifetime.

Cost breakdown. The two integration patterns have distinct cost profiles.

Pattern A (in-prompt taxonomy). Zero additional LLM calls beyond what the agent's loop already does. Cost is paid in prompt tokens (the taxonomy is appended to the system prompt) and in agent reasoning overhead at each checkpoint, neither of which adds an LLM call.

Pattern B (external judge). One additional LLM call per checkpoint, so $C_{\text{B}} = M$ calls per task. As a fraction of the agent's own LLM expenditure, this is small (under 5% in the SWE-agent runs reported in Section 4.4, where each task involves dozens of solver and tool-use calls).

Lifetime cost. Over L deployed tasks, runtime overhead is 0 for Pattern A and $M \cdot L$ for Pattern B. The shared taxonomy generation cost $C_{\text{gen}} = 8$ amortizes to zero per task as $L \to \infty$, so the long-run per-task overhead reduces to the per-checkpoint cost in Pattern B and to nothing in Pattern A.

F. TRAIL Validation Details

F.1. Taxonomy validation on TRAIL

TRAIL (Deshpande et al., 2025) provides 117 GAIA-derived agent traces hand-annotated by four software-engineering experts under a 20-category canonical taxonomy. We run a four-LLM panel (Claude Opus 4.7, GPT-5.4, Gemini-2.5-Pro,

Table 20: TRAIL panel-vs-human-gold area-κ (4-LLM majority vote, n=117 GAIA traces) at majority threshold (2-of-4). Each row is the same panel under a different methodology variant; 95% CIs from a 2000-sample trace-level bootstrap. The bottom row matches the ATLAS protocol and gives the panel-validity result.

Methodology	area-κ	95% CI	Landis & Koch
Baseline (free-text prompts)	0.500	$[0.476, 0.611]$	moderate
Same-model 4× panel (4× GPT-5.4 with seed variation)	0.630	$[0.548, 0.710]$	substantial
Span-grounded prompts (cross-family panel)	0.563	$[0.492, 0.635]$	moderate
Span-grounded + 1-round deliberation (ATLAS protocol)	**0.725**	$[\mathbf{0.654}, \mathbf{0.795}]$	**substantial**

GPT-5.4-mini) under the ATLAS protocol of Section 3.1 (span-grounded prompts, one round of peer deliberation, majority vote) and report Cohen's κ at the area level (Reasoning / Planning / System) against the four-expert gold; bootstrap 95% CIs use n_{boot}=2000 trace-level resamples.

Auto-generated codes recover expert failures more faithfully than the hand-crafted vocabulary. Holding the panel and prompts fixed and varying only the vocabulary the annotators consume, TRAIL's hand-crafted 20-category taxonomy yields area-$\kappa = 0.516$, while ATLAS's 29 auto-generated codes (mapped back to TRAIL via a one-time judge step) yield area-$\kappa = 0.682$, a +0.166 absolute gain (+32% relative). The mapping is high-precision: 28 of 29 ATLAS codes (96.6%) admit a clean TRAIL counterpart, with several broad TRAIL categories receiving 6-to-1 fan-in from finer-grained ATLAS codes (e.g., R5 "Incorrect Problem Identification" receives six). Annotators agree more readily on the finer-grained, trace-discovered codes than on the broader hand-curated categories, and they do so without any additional human authoring effort.

Panel-validity check. Under the same ATLAS protocol, the panel reaches area-$\kappa = 0.725$ [95% CI $0.654, 0.795$] against the four-expert gold (Table 20), substantial agreement on the Landis & Koch scale, with the lower CI bound exceeding the conventional 0.61 threshold. This anchors the auto-versus-hand-crafted comparison: the panel is a reliable judge against humans, so the residual difference between 0.682 and 0.516 is attributable to the vocabulary, not to panel noise. At the 20-leaf-category level, panel-vs-gold κ remains in the fair range ($\kappa \approx 0.34$), consistent with the difficulty of fine-grained failure annotation reported in the published TRAIL paper (Deshpande et al., 2025).

F.2. Bootstrap CIs (n=2000) for TRAIL panel-vs-gold area-κ

Each row is the same 117-trace evaluation with a different methodology variant; thresholds are the number of panel members (out of 4) that must agree on a category for it to enter the panel prediction. CIs are 95% percentile intervals from 2000 trace-level bootstrap resamples (seed 42).

Methodology	thr=1	95% CI	thr=2	95% CI	thr=3	95% CI
Cross-family 4-panel (no grounding)	0.682	$[0.608, 0.754]$	0.500	$[0.476, 0.611]$	0.307	$[0.236, 0.380]$
+1-round deliberation	0.692	$[0.620, 0.764]$	0.590	$[0.517, 0.665]$	0.439	$[0.354, 0.524]$
Same-model 4× GPT-5.4	0.625	$[0.544, 0.697]$	0.630	$[0.548, 0.710]$	0.563	$[0.485, 0.640]$
Span-grounded 4-panel	0.725	$[0.651, 0.796]$	0.563	$[0.492, 0.635]$	0.404	$[0.335, 0.473]$
Span-grounded + deliberation	**0.761**	$[\mathbf{0.690}, \mathbf{0.824}]$	**0.725**	$[\mathbf{0.654}, \mathbf{0.795}]$	0.645	$[0.573, 0.717]$

Table 21: Bootstrap 95% CIs on TRAIL panel-vs-gold area-κ. The bottom row is the ATLAS protocol; thr=2 is the headline result reported in the main text.

F.3. Methodological journey: from naive κ=0.027 to κ=0.725

A naive cross-family 4-panel with strict (cat, span) majority on TRAIL initially produced area-$\kappa = 0.027$, an apparent failure of ATLAS. Five methodology choices recover the substantial-agreement result: (1) score at the 3-area level rather than 20 leaf categories (κ=0.500); (2) drop the strict span-id match, since 29% of LLM-emitted span IDs do not actually appear in the trace (κ=0.516); (3) replace the cross-family panel with a same-architecture 4× panel (κ=0.630); (4) add span-grounded prompts that eliminate hallucinated span IDs at the source (κ=0.563 alone, but enables (5)); (5) add one round of peer deliberation, the ATLAS protocol's intended aggregation step (κ=0.725). Each step is a defensible methodology choice

motivated by a specific gap in the data rather than a metric-fishing maneuver. Each row of the table below adds one such choice to the previous row.

#	Step	thr=2 area-κ
0	Cross-family 4-panel, leaf level, strict (cat,span) majority (naive)	0.027
1	+ Score at 3-area level (R / P / S) instead of 20 leaves	0.500
2	+ Drop strict span match (29% LLM-emitted span IDs are hallucinated)	0.516
3	+ Replace cross-family panel with same-architecture 4× panel	0.630
4	+ Span-grounded prompts (eliminate hallucination at the source)	0.563
5	+ 1-round peer deliberation (the ATLAS protocol's intended aggregation)	**0.725**

Table 22: Stepwise progression. Each step is a defensible methodology choice motivated by a specific gap in the data, not a metric-fishing maneuver.

F.4. Per-annotator post-deliberation κ vs human gold

The four-LLM panel result is the majority vote across annotators, but it is informative to ask how each individual annotator scores against the four-expert TRAIL gold before voting. The table below reports per-annotator leaf-level and area-level κ under the same span-grounded + 1-round-deliberation protocol the panel uses. Three of four annotators reach substantial agreement at the area level on their own, and Claude Opus 4.7 alone reaches $\kappa = 0.744$, comparable to the panel's 0.725. This indicates the panel result is not driven by a single dominant annotator: every annotator contributes signal, and majority voting consolidates rather than rescues individual judgments.

Annotator	leaf-κ	area-κ	area Jaccard	Landis & Koch (area)
Claude Opus 4.7	0.334	0.744	0.821	substantial
GPT-5.4-mini	0.308	0.676	0.783	substantial
GPT-5.4	0.317	0.625	0.729	substantial
Gemini 2.5 Pro	0.301	0.542	0.650	moderate

Table 23: Individual annotator metrics vs TRAIL gold after grounded prompts and 1 round of deliberation. The worst single annotator already reaches moderate agreement; Opus 4.7 alone reaches κ=0.744, comparable to the panel's 0.725.

F.5. Inter-annotator κ matrix (post-deliberation, span-grounded panel)

A complementary check is whether the four panel members agree with one another, not just with the human gold. The table below reports pairwise κ between every pair of annotators (leaf-level / area-level) after grounding and deliberation. All six pairs reach $\kappa \geq 0.587$ at the leaf level, and the median pair reaches $\kappa \approx 0.71$ leaf and $\kappa \approx 0.74$ area, replicating the inter-LLM $\kappa = 0.78$ that ATLAS originally reports on a different benchmark within methodology-matched conditions. High inter-annotator agreement on the same vocabulary is a precondition for the panel-vs-gold result: if the panel disagreed internally, the majority vote against the gold could not be trusted as a vocabulary signal.

	Opus 4.7	GPT-5.4	GPT-5.4-mini	Gemini 2.5 Pro
Opus 4.7	—	0.742 / 0.791	0.794 / 0.814	0.604 / 0.688
GPT-5.4		—	0.710 / 0.808	0.651 / 0.687
GPT-5.4-mini			—	0.587 / 0.671
Gemini 2.5 Pro				—

Table 24: Pairwise inter-annotator κ (leaf-level / area-level) on TRAIL after grounded + deliberation. Six of six pairs reach $\kappa \geq 0.587$ at leaf level; the median pair reaches $\kappa \approx 0.71$ leaf and ≈ 0.74 area, replicating ATLAS's claimed $\kappa = 0.78$ inter-LLM agreement on a different benchmark.

F.6. Auto-vocabulary mapping examples (ATLAS codes → TRAIL canonical IDs)

The judge step that maps each ATLAS-generated code (29 codes) to its closest TRAIL canonical ID (20 codes) finds a 6-to-1 fan-in to several broad TRAIL categories, evidence that the auto-taxonomy is consistently finer-grained than the

hand-curated one.

TRAIL category (gold)	ATLAS auto-codes that map to it
R5 Incorrect Problem Identification	B.1 Tracker_Incorrect_Tag_Configuration; B.2 Tracker_Incomplete_Instrumentation_Coverage; C.1 Entity_Identity_Collapse; C.2 Temporal_Anchor_Mismatch; C.6 Tabular_Filter_Or_Aggregation_Misapplication; C.9 Spatial_Or_Ordering_Frame_Misinterpretation
R4 Tool Output Misinterpretation	B.9 Orchestrator_Evidence_Integration_Failure; C.5 Retrieved_Fact_To_Answer_Transformation_Error; C.7 Exact_Span_Extraction_Error; C.8 Multimedia_Perceptual_Misread; C.10 Cross_Source_Schema_Mismatch; C.13 Cross_Source_Synthesis_Reconciliation
P4 Task Orchestration Errors	A.3 Strategy_Initialization_Or_Routing_Failure; B.4 Unknown_Role_Misalignment; B.8 Orchestrator_Incorrect_Tool_Selection; B.10 Orchestrator_Insufficient_Self_Verification; C.3 Incomplete_Multihop_Chain
R8 Instruction Non-compliance	A.1 Final_output_format_noncompliance; B.3 Tracker_Unsafe_Or_Noncompliant_Tracking_Method; C.12 Required_Source_Type_Not_Actually_Used
P1 Context Handling Failures	A.2 Self_inconsistency_across_pipeline_states; A.6 Inter-stage_Handoff_Loss_Or_Drift

Table 25: Auto-to-hand-crafted code mapping. ATLAS's 29 codes map at 96.6% precision (28/29 codes have a clean TRAIL counterpart); 0 codes map to "no equivalent in TRAIL". Each ATLAS code expresses a more specific failure mode than the broader TRAIL category, e.g., 6 different fine-grained codes capture distinct sub-types of TRAIL's R5 "Incorrect Problem Identification".

G. Best architectures from the OlympiadBench evolution

We provide the full source for the three best architectures discussed in the OlympiadBench analysis (Section 4.3): the ATLAS-guided best (registered at iteration 92, 91.91% on the held-out 655-problem test set), the no-taxonomy best (87.94%), and the MAST-guided best (89.47%). All three runs share the same AdaEvolve substrate, the same three-agent seed architecture, and matched compute; only the failure-feedback signal in the mutation prompt differs. Below each architecture, we also reproduce the verbatim taxonomy-judge output that drove the iteration-67 and iteration-92 breakthroughs in the ATLAS run.

G.1. ATLAS best architecture (iter 92, test 91.91%)

The architecture runs four diversified solver variants (A–D) under a 25-call LLM budget per problem, then applies a lightweight VERIFIER that boosts the score of structurally-valid candidates, selects up to four diverse candidates for refinement via a length-and-variant-keyed frontier, and runs up to two refinement rounds before final aggregation. The verifier replaces the binary gate of earlier iterations with a numerical score boost (`c["score"] += 0.1` on `VALID`), giving the convergence signal a quantitative weight that breaks ties between divergent candidates, the iteration-92 response to the B.6 *Coordinator Aggregation Mismatch* code raised by the iteration-60 judge.

```
SOLVER_PROMPT_BASE = (
    "You are a problem-solving agent. Your task is to produce the final answer "
    "for the given problem. Output must be a single line in the exact format: "
    "FINAL ANSWER: <answer> and nothing else. Do not include any additional "
    "text, steps, or explanations in the output. You may internally consider "
    "multiple approaches, but the final output must present only the final "
    "answer line. If the problem requires a numeric, symbolic, or logical "
    "result, provide the correct value consistent with the problem's constraints."
)

SOLVER_VARIANT_PROMPTS = [
    "Variant A (Balanced exploration): Solve the problem with a robust, "
    "general approach that emphasizes correctness and coverage of typical "
    "edge cases. Prioritize producing a valid final answer line only.",
```

```
    "Variant B (Alternative reasoning path): Solve using a distinct "
    "reasoning path that explores a different method or perspective from "
    "Variant A, still ensuring correctness. Final answer line only.",

    "Variant C (Alternate derivation): Solve by deriving via an alternate "
    "method or decomposition to validate results and reduce risk of "
    "method-specific blind spots. Final answer line only.",

    "Variant D (Near-frontier ideas): Solve by exploring ideas that are "
    "plausible but less conventional, aiming to diversify potential "
    "solution profiles while ensuring correctness. Final answer line only.",
]

VERIFIER_PROMPT_BASE = (
    "You are a lightweight verifier. Given a problem and a candidate "
    "solution, assess whether the solution is internally consistent and "
    "likely correct. Respond with a succinct verdict and notes. Format: "
    "VERDICT: VALID or VERDICT: INVALID\nNOTES: <text>"
)

REFINER_PROMPT_BASE = (
    "You are a concise refiner. Given a problem, a candidate solution, "
    "and any verifier notes, produce a corrected final answer. Output "
    "exactly one line in the format: FINAL ANSWER: <answer> and nothing else."
)

def solve(problem, call_llm, evaluate):
    MAX_LLMS, MAX_EVALS = 25, 8
    llm_used, eval_used = {"count": 0}, {"count": 0}

    def safe_call(prompt, user_message):
        if llm_used["count"] >= MAX_LLMS:
            raise BudgetError()
        res = call_llm(prompt, user_message)
        llm_used["count"] += 1
        return res

    def safe_evaluate(sol):
        if eval_used["count"] >= MAX_EVALS:
            raise BudgetError()
        res = evaluate(sol)
        eval_used["count"] += 1
        return res

    # Phase 1: four diversified solver variants
    candidates = []
    for vid in range(4):
        solution = safe_call(SOLVER_PROMPT_BASE,
                             f"{SOLVER_VARIANT_PROMPTS[vid]}\nProblem:\n{problem}\n")
        candidates.append({"variant": vid, "solution": solution,
                           "score": 0.0, "correct": False, "verifier_notes": ""})

    # Phase 2: evaluate; early exit on correct
    for c in candidates:
        res = safe_evaluate(c["solution"])
        c["score"] = float(res.get("score", 0.0))
        c["correct"] = bool(res.get("correct", False))
    correct = [c for c in candidates if c["correct"]]
    if correct: return max(correct, key=lambda c: c["score"])["solution"]

    # Phase 3: verifier ,  score boost, not gate
    for c in candidates:
        v = safe_call(VERIFIER_PROMPT_BASE,
                      f"Problem:\n{problem}\nCandidate:\n{c['solution']}\n")
```

```
        c["verifier_notes"] = v
        if "valid" in v.lower():
            c["score"] += 0.1   # numerical convergence weight (B.6 response)

    # Phase 4: diversity-frontier selection (top 4 by score, varied
    # variants and lengths)
    frontier = select_diverse_candidates(candidates, max_keep=4)

    # Phase 5: up to two refinement rounds on frontier
    refined = []
    for idx, base in enumerate(frontier[:2]):
        r = safe_call(REFINER_PROMPT_BASE,
                      f"Problem:\n{problem}\n\nPrevious Attempt:\n{base['solution']}\n\n"
                      f"Verifier Feedback:\n{base['verifier_notes']}\n")
        res = safe_evaluate(r)
        if res.get("correct"): return r
        refined.append({"solution": r, "score": float(res.get("score", 0.0)),
                        "correct": bool(res.get("correct", False))})

    # Phase 6: aggregate; best correct, else best score
    all_c = candidates + refined
    correct = [c for c in all_c if c.get("correct")]
    if correct: return max(correct, key=lambda c: c["score"])["solution"]
    return max(all_c, key=lambda c: c["score"])["solution"]
```

Per-call budget. Up to 25 LLM calls per problem: 4 solver variants + 4 verifier calls + 2 refiner calls = 10 calls in the typical path, leaving headroom for stochastic retries. A length-and-variant-keyed cache eliminates duplicate `evaluate()` calls when the LLM returns the same output across stages.

G.2. No-taxonomy best architecture (test 87.94%)

The architecture is an adversarial-debate pipeline: a primary solver produces an initial solution, two critics independently identify failure modes, an improver consolidates the critiques into a revised solution, two more critics review the revision, a second improver incorporates that feedback, and a final synthesis step combines all candidates. Total: eight LLM calls per problem, four `evaluate` calls. There is no verifier and no explicit diversification mechanism.

```
PRIMARY_SYSTEM = (
    "You are an expert problem-solving agent. Given a problem statement, "
    "produce a complete solution including clear reasoning. End with a single "
    "line that starts with 'FINAL ANSWER:' followed by the solution text."
)

CRITIQUE_A_SYSTEM = (
    "You are Critic A. Read the problem and the initial solution. Identify "
    "the most important failure modes, ambiguities, and missing cases. "
    "Provide a concise bullet list of issues and concrete fixes. Do not "
    "output a final answer here; just critique."
)

CRITIQUE_B_SYSTEM = (
    "You are Critic B. Focus on alternative failure modes, edge cases, and "
    "opportunities to improve robustness. Provide a compact, prioritized "
    "list of problems and actionable remedies. Do not output a final answer here."
)

IMPROVER1_SYSTEM = (
    "You are Improver 1. You have the problem, the initial solution, and "
    "critiques A and B. Produce a revised solution that addresses the "
    "failures and strengthens the reasoning. End with: FINAL ANSWER: <solution>."
)

# Critiques C, D and Improver 2 follow the same structure as A, B, Improver 1
# but operate on the first revision.
```

```
SYNTHESIS_SYSTEM = (
    "You are Synthesis Agent. You have the original problem, three candidate "
    "solutions (initial, first revision, second revision), and four critiques "
    "(A--D). Synthesize a single final answer that combines the strongest "
    "elements from all sources while resolving remaining gaps."
)

def solve(problem, call_llm, evaluate):
    # Stage 0: Primary solver, evaluate, early-exit
    primary = call_llm(PRIMARY_SYSTEM, "Problem:\n" + problem)
    if evaluate(primary).get("correct"): return primary

    # Stage 1: Two critics on initial solution
    critique_A = call_llm(CRITIQUE_A_SYSTEM, ...)
    critique_B = call_llm(CRITIQUE_B_SYSTEM, ...)

    # Stage 2: Improver 1 consolidates critiques into revision
    revision1 = call_llm(IMPROVER1_SYSTEM, ...)
    if evaluate(revision1).get("correct"): return revision1

    # Stage 3: Two more critics on revision
    critique_C = call_llm(CRITIQUE_C_SYSTEM, ...)
    critique_D = call_llm(CRITIQUE_D_SYSTEM, ...)

    # Stage 4: Improver 2 produces second revision
    revision2 = call_llm(IMPROVER2_SYSTEM, ...)
    if evaluate(revision2).get("correct"): return revision2

    # Stage 5: Synthesis combines all sources
    synthesis = call_llm(SYNTHESIS_SYSTEM, ...)
    if evaluate(synthesis).get("correct"): return synthesis

    return best_by_score([primary, revision1, revision2, synthesis])
```

Comparison with the ATLAS architecture. The no-taxonomy run discovered *post-hoc* consolidation (multiple critics + a synthesis step), but never introduced a structural verification stage operating on each candidate before refinement. The mutator was capable of producing such a structure, ATLAS discovered a verifier-gated four-variant architecture starting from the same seed, but the no-taxonomy mutation prompts contained only free-text reflection on the previous candidate's score, not the categorical failure-mode codes that prompted ATLAS's mutator to target verification weaknesses specifically.

G.3. MAST-guided best architecture (test 89.47%)

The MAST run uses the published 14-code MAST taxonomy (Cemri et al., 2025) in the mutation prompt instead of a domain-induced taxonomy. The best architecture is a strict five-stage pipeline: a primary solver emitting a JSON-wrapped solution, a refiner using evaluator feedback, a voter producing a consensus from primary and refined candidates, a finalizer normalizing the consensus, and a terminal content generator producing the boxed answer for the judge.

```
SOLVER_PRIMARY_PROMPT_STATIC = (
    "You are an expert olympiad-level mathematics solver. Solve the given "
    "math olympiad problem completely and clearly, showing the necessary "
    "steps and reasoning. Your final line must contain a LaTeX-encoded "
    "boxed answer of the form \\boxed{<answer>}. Output your response as a "
    "single JSON object with the key 'proposed_solution'."
)

SOLVER_REFINE_PROMPT_STATIC = (
    "You are an expert at refining olympiad solutions. You will receive a "
    "problem statement, an initial proposed_solution, and evaluator feedback. "
    "Carefully review the reasoning for errors, check arithmetic and logical "
    "steps, preserve correct parts, fix errors. Output a JSON object with "
    "the key 'refined_solution'."
)
```

```
VOTER_PROMPT_STATIC = (
    "You are an ensemble solver. You will receive two candidate solutions "
    "and evaluator feedback. Produce a consensus candidate that combines "
    "the best parts of both attempts. Output a JSON object with the key "
    "'consensus_candidate'."
)

FINALIZER_PROMPT_STATIC = (
    "You are a sanity-check agent that normalizes the consensus solution. "
    "Ensure the content is coherent, self-contained, and that the final "
    "answer is present as a LaTeX-encoded boxed expression at the end. "
    "Output a JSON object with the key 'final_for_output'."
)

TERMINAL_PROMPT_STATIC = (
    "You are the terminal content generator. Your output will be the final "
    "content presented to the judge. Output a JSON object with key "
    "'final_output' containing the string with the boxed final answer."
)

def solve_inner(problem, call_llm, evaluate, judge=None):
    # Five sequential stages, each gated by an early-exit-on-correct check
    primary = call_llm(SOLVER_PRIMARY_PROMPT_STATIC, ...)
    if evaluate(primary).get("correct"): return primary
    refined = call_llm(SOLVER_REFINE_PROMPT_STATIC, ...)
    if evaluate(refined).get("correct"): return refined
    consensus = call_llm(VOTER_PROMPT_STATIC, ...)
    finalized = call_llm(FINALIZER_PROMPT_STATIC, ...)
    return call_llm(TERMINAL_PROMPT_STATIC, ...)
```

Comparison with ATLAS. The MAST-guided run discovered a structurally similar pipeline to ATLAS's (multi-stage, with consensus and finalization), but it does not include a parallel diversification phase: the primary solver runs once rather than across four variants, and the consensus stage operates on two candidates rather than across a frontier. The induced ATLAS taxonomy contained codes specific to the math-domain solver behaviour (e.g., `C.5 Combinatorics Counting Misreasoning`, `B.4 Refiner Overcorrects`) that surfaced the need for parallel exploration; the published MAST codes (generic specification/inter-agent/verification failures) did not.

G.4. Verbatim judge outputs that drove the two breakthroughs

Iteration-43 judge output (drove the iter-67 breakthrough, score 0.15 → 0.50).

```
A-codes:
  [A.1] Behavioral Anomaly: Non-convergent iteration / stagnation
    Evidence: "Output: FINAL ANSWER: Please provide the problem statement.";
              "Output: FINAL ANSWER: 0";
              "Output: FINAL ANSWER: Cannot determine without the problem statement."
  [A.3] Instruction Compliance Violation
    Evidence: "Output: FINAL ANSWER: Please provide the problem statement.";
              "OUTPUT: FINAL ANSWER: Problem statement missing."
  [A.6] Ground-Truth / Verifier Mismatch
    Evidence: Coordinator Aggregation Mismatch notes;
              inconsistent final answers across branches.
  [A.4] Evaluation Feedback Misinterpretation
    Evidence: "Evaluator Feedback: Wrong Axis Routing Diagnosis"
              followed by non-addressing outputs.

B-codes (by role):
  checker:
    [B.1] Non-compliant Output by Checker ,  checkers default to problem-
          statement prompts instead of solving.
    [B.2] Low-Quality or Inconsistent Final Outputs from Checker.
  refiner:
```

```
    [B.2/B.3] Refiner Fails to Improve or Regresses After Critique
              ,  refiners produce final-answer-style outputs or
              non-informative AXIS_VECTOR lines (all-zero vectors).
    [B.4] Refiner Overcorrects or Hijacks Methodology ,  shifts from
          AXIS_VECTOR to final numeric answers, ignoring axis-vector protocol.
  coordinator:
    [B.6] Coordinator Aggregation Mismatch: aggregated results conflict
          due to divergent FINAL ANSWER lines; no single coherent vector.

C-codes:
  [C.1] Sign/Difference Misreasoning
  [C.2] Edge Case Omission / Degeneration ,  edge cases enumerated in
        plans but not validated within the same turn.
  [C.3] Overgeneralization in Domain Prereqs
  [C.4] Algebraic Manipulation Gap
  [C.5] Combinatorics Counting Misreasoning

Cross-Turn Patterns:
  - Output format drift and axis-vector discipline violations are pervasive
  - Refiner critique misalignment: after evaluator feedback, refiners often
    persist with axis-vector placeholders or switch to final answers
  - Domain reasoning gaps persist: AXIS_VECTOR lines lack problem-specific
    interpretation or derivation
```

Iteration-60 judge output (drove the iter-92 breakthrough, score 0.30 → 0.55, the registered best).

```
A-codes:
  [A.3] Instruction Compliance Violation: Output formats and content
        frequently violate task prompts (multiple final-answer lines,
        interleaved reasoning, or axis-vector only outputs instead of
        a single required line).
    Evidence: "Output: AXIS_VECTOR: [0.15, 0.25, ...]" followed by notes
              that final-answer formatting violated the rule;
              "Turn lacks a concrete corrected solution or final
              'FINAL ANSWER:' line."
  [A.6] Ground-Truth / Verifier Mismatch: Divergent final answers across
        checkers for the same problem.
    Evidence: "FINAL ANSWER: 807" (checker) vs "FINAL ANSWER: 2017"
              (checker_4) and other inconsistent finals.
  [A.4] Evaluation Feedback Misinterpretation: Critique echoed but not
        corrected.
    Evidence: "Evaluator Feedback: Wrong" with subsequent turns not
              delivering a corrected, fully reasoned fix.

B-codes:
  refiner:
    [B.3] Refiner Fails to Improve or Regresses After Critique
          ,  refiner outputs AXIS_VECTOR without addressing critique.
    [B.4] Refiner Overcorrects or Hijacks Methodology
          ,  axis-vector outputs without justification.

C-codes:
  [C.2] Edge Case Omission / Degeneration ,  explicit edge cases ignored.
  [C.3] Overgeneralization in Domain Prereqs ,  rules or bounds asserted
        without full justification.
  [C.5] Combinatorics Counting Misreasoning ,  counting arguments rely
        on independent choices or naive decompositions.
  [C.4] Algebraic Manipulation Gap ,  incomplete or flawed algebraic steps.

Cross-Turn Patterns:
  - Pervasive Instruction Compliance Violations: failures to deliver only
    the required final-answer line; mixing reasoning, axis-vectors, or
    guidelines with the final answer.
  - Checker consensus instability: multiple checkers on the same turn
    produce divergent finals or inconsistent validations.
```

```
  - Refiners stagnate or derail critiques: emit axis-vectors with little
    or no justification.
```

H. SWE-bench Verified Mini: experimental setup

This appendix gives the full setup for the runtime-integration experiment reported in §4.4: harness configuration, solver model, the prompt templates and submit-gate that define each of the five arms, the Pattern B judge service, and per-arm wall-clock and cost.

Harness and solver. All five arms use SWE-agent v1.1.0 (the upstream Princeton harness) with the same tool bundle (`tools/registry`, `tools/edit_anthropic`, `tools/review_on_submit_m`) and the same per-instance limits: 75 LLM calls, $4 cost ceiling per task, $200 cost ceiling per run. The solver is GPT-5 (the OpenAI `gpt-5` model) at temperature 0, greedy decoding, with the same model handling reasoning and tool-call generation through OpenAI function calling. Solver and judge share the same underlying model; only the prompt differs. Per-arm wall-clock runtime on the 50-instance Mini benchmark is 60–90 minutes (most of which is the per-instance Docker build phase) at 25 parallel workers, with a per-arm cost of $24–28 measured from the saved trajectory `model_stats`.

Five arms, shared scaffolding. Arms 1 and 2 are baselines. Arms 3, 4, and 5 share the same three-checkpoint protocol with forced reflection at each checkpoint, differing only in (a) which taxonomy the agent reasons about and (b) whether classification happens inside the agent's context (Arms 3 and 4) or in an external judge (Arm 5).

Checkpoint protocol (Arms 2–5). The harness does not expose a native checkpoint hook; the protocol is prompt-induced. The agent is told it must emit a structured reflection at three points:

- **Checkpoint 1**: after initial exploration, before the first edit (after a few `view`/`grep`/`find` commands and before the first `str_replace` or `create_file`).
- **Checkpoint 2**: after each edit, before running tests (after at least one editor action, before `pytest`).
- **Checkpoint 3**: after tests pass, before submitting (after the agent's final test run produces a green result, immediately before the `submit` command).

The submit-gate enforces compliance: a wrapper inside `tools/review_on_submit_m` scans the trajectory at submit time for valid checkpoint markers and refuses submission if fewer than three are found.

Arm 1 (Base). Vanilla SWE-agent. The system prompt is the default "*You are a helpful assistant that can interact with a computer to solve tasks.*" No taxonomy, no reflection protocol, no submit-gate.

Arm 2 (Reflexion). A Reflexion-style protocol implemented from scratch (inspired by Shinn et al. (2023) but not a port of their reference implementation, since the original targets a different loop structure). The protocol fires at the same three checkpoints as Arms 3–5; at each checkpoint the agent must emit:

```
REFLECTION (checkpoint N):
  What I just did: <one sentence summary>
  What could go wrong: <one specific risk in MY recent actions>
  Adjustment: <concrete next action OR "ruled out because: ...">
  Second potential issue: <another specific risk>
  Adjustment: <concrete next action OR "ruled out because: ...">
```

The submit-gate requires three such markers but does not check their content. Reflection is purely free-text; no taxonomy codes are required.

Arm 3 (MAST in-prompt). Same checkpoint protocol as Arm 2 but the reflection is structured against the published 14-code MAST taxonomy (Cemri et al., 2025) (FC1.1 through FC3.3). The MAST taxonomy is included verbatim in the agent's system prompt as a checklist with code IDs, names, and one-line descriptions per code. At each checkpoint the agent must emit:

```
REFLECTION (checkpoint N):
  Candidate failure 1: <code, e.g. FM-2.6>
    Why it might apply: <one specific sentence about MY recent actions>
    Adjustment: <concrete next action OR "ruled out because: ...">
  Candidate failure 2: <code, e.g. FM-3.2>
    Why it might apply: <one specific sentence about MY recent actions>
    Adjustment: <concrete next action OR "ruled out because: ...">
```

The submit-gate parses each reflection block, validates that two candidate codes are listed and that each cites a code matching the MAST regex, and rejects submission otherwise.

Arm 4 (ATLAS Pattern A). Identical to Arm 3 except the taxonomy is the 30-code SWE-bench-induced ATLAS taxonomy (Appendix A.5) rather than MAST. The reflection format is the same; the submit-gate validates code references against the regex `[ABC]\d+`.

Arm 5 (ATLAS Pattern B). The taxonomy lives outside the agent. At each checkpoint, the agent calls a judge service (`172.17.0.1:8770/judge/atlas`) by sending a structured summary of its recent trajectory: free-text summary, the current git diff, and any recent pytest output. The service applies the taxonomy to that summary via a single LLM call and returns the top-five most likely failure codes ranked by evidence support, each with a concrete next-step guidance. The response is delivered to the agent as its next observation.

The judge is stateless across checkpoints: each call is a fresh chat completion with the prompt rebuilt from whatever the agent chooses to include in the current summary. The submit-gate validates that three judge responses (parseable JSON matching the schema) are present in the trajectory before allowing submission.

Table 26: Per-arm cost on the 50-instance SWE-bench Verified Mini. Cost is computed from `info.model_stats` in saved trajectories. The five arms are within $4 of each other; the two ATLAS arms incur an additional one-time taxonomy-generation cost of 8 LLM calls (Appendix E), amortized across all subsequent runs on this domain.

Arm	Total cost	Mean per instance
Arm 1: Base	$23.72	$0.50
Arm 2: Reflexion (free-text)	$25.00	$0.50
Arm 3: MAST in-prompt	$26.00	$0.52
Arm 4: ATLAS Pattern A	$26.80	$0.54
Arm 5: ATLAS Pattern B	$27.92	$0.56

Cost summary.

I. SWE-bench Verified Mini: Case Studies

This appendix backs the per-instance breakdown referenced from §4.4. We provide three artifacts: (i) the pass/fail outcome of each of the five arms on every Mini instance where at least one arm succeeded; (ii) the set of instances where Pattern A and Pattern B disagreed; and (iii) three case studies showing what the in-prompt taxonomy versus the external judge actually surfaced on the same trajectory.

I.1. Per-instance arm outcomes

Of the 50 SWE-bench Verified Mini instances, 46 were resolved by at least one of the five arms. Table 27 lists those 46 grouped by repository (4 instances on which all five arms failed are omitted; the totals at the bottom of the table match the aggregate scores reported in Table 3).

I.2. Pattern A vs Pattern B disagreements

Pattern A and Pattern B disagreed on six instances (Table 28); five resolved in Pattern B's favor and one in Pattern A's. The net 4-instance gap matches the aggregate $39 - 35 = 4$ reported in Table 3. Notably, three of the five Pattern-B-only wins

(django-12193, django-12308, django-12325) are instances where *no other arm* succeeded, not just where Pattern A failed, suggesting the external judge captures failure modes the in-prompt taxonomy and MAST both miss.

I.3. Case study: `django-12325` (parent_link order bug)

The bug is an order-dependent collection of parent links in multi-table inheritance: when two `OneToOneFields` point to the same parent, the field discovered first is treated as the canonical parent link, raising `ImproperlyConfigured` for the other declaration order. Only Pattern B resolved this instance.

Pattern A's in-prompt reflection at the pre-submission checkpoint surfaced *six* codes from the active taxonomy: A5 (Misread root cause), A1 (Wrong file targeted), B4 (Over-broad behavior change), B11 (Over-narrow checks), C3 (No syntax/build check), B5 (Ineffective layer fix). Critically, A5 (Misread root cause), which corresponds exactly to what went wrong, is on the list. The agent enumerated it, wrote a brief justification, and proceeded to submit a 12-line patch with multi-condition precedence logic over `parent_link`.

Pattern B's external judge raised A5 as the *primary* code at three consecutive checkpoints, with reasoning that named the actual mechanism: "the real bug is order-dependent collection of parent links or how parent_link is determined/propagated, not simply filtering by field type." The judge's guidance required a deterministic failing unit test before further edits. The agent followed that sequence and produced a 2-line filter that passed.

The diagnostic difference is not whether the right code was *available*, both arms had A5, but whether the agent acted on it. Pattern A's reflection emitted A5 alongside five other plausible codes; Pattern B's external classifier ranked A5 first with concrete guidance the agent could not silently absorb.

I.4. Case study: `django-12193` (CheckboxInput attrs mutation)

The bug is a state leak in `SplitArrayWidget` where shared `attrs` dictionaries get mutated by `CheckboxInput`. The correct fix lives in `CheckboxInput` (the mutating component); patching `SplitArrayWidget.get_context` (the symptom site) leaves the root cause intact. No arm except Pattern B resolved this instance.

Pattern A's reflection cited A5 (Misread root cause), A4 (Wrong abstraction layer), B5 (Ineffective layer fix), C3 (No syntax/build check), B4 (Over-broad behavior change), C2 (Didn't revalidate final diff). Again A5 was on the list. The agent submitted a fix at the symptom site and validated with an ad-hoc script.

Pattern B's judge raised A5 as primary with direct guidance: "the real mutation happens inside `CheckboxInput`", followed by concrete reproduction steps. A second judge call raised C1 (Ad-hoc only): "relying on an ad-hoc script risks missing regressions or integration breakage." The judge then named the wrong-layer issue explicitly via B5 (Ineffective layer fix), with a concrete remedy: "ensure the copy logic handles `None` and non-dict gracefully (e.g., `attrs_copy = dict(attrs or {})`)." The agent ran the full `tests/forms/widgets` suite, applied the named fix, and passed.

I.5. Case study: `sphinx-8548` (autodoc inherited attributes)

This is the one instance in the disagreement set where Pattern A succeeded and Pattern B did not. The bug concerns `autodoc`'s handling of attribute docstrings on inherited members. Pattern A resolved it; Pattern B did not converge on a viable patch within the step budget.

Pattern B's first judge call ranked C5 (No proper reproduction) as the primary code, directing the agent to first reproduce with the project test runner ("run `pytest -q tests/test_ext_autodoc.py::test_autodoc_typed_inherited_instance_variables`") before any code changes. The judge's secondary codes (A5 Misread root cause, A4 Wrong abstraction layer) correctly identified that the fix likely belonged in `ClassDocumenter`/`AttributeDocumenter` rather than `importer.get_class_members`, but the C5-first ordering sent the agent into an extended reproduction-and-instrumentation loop that exhausted its budget before a coherent patch emerged.

The case demonstrates a real failure mode of the external judge: when its prioritization defers to a procedural prerequisite (here, test reproduction) over a substantive diagnosis the agent could have acted on directly, the verification overhead can outweigh the diagnostic benefit. This is one of six disagreement instances overall, and the only Pattern-A-only win.

Table 27: Per-instance pass (✓) / fail (–) outcomes for the 46 Mini instances with at least one passing arm. *Base* is vanilla SWE-agent, *Reflex* is Reflexion (free-text reflection at fixed checkpoints), *MAST* is the published 14-code MAST taxonomy in-prompt, *A* is ATLAS Pattern A (induced taxonomy in-prompt), *B* is ATLAS Pattern B (induced taxonomy consulted via external judge).

Instance	Base	Reflex	MAST	A	B
django (23 instances)					
django-9296	✓	✓	✓	✓	✓
django-11815	✓	–	✓	✓	✓
django-11848	✓	✓	✓	✓	✓
django-11880	✓	✓	✓	✓	✓
django-11885	–	✓	✓	✓	✓
django-11951	✓	✓	✓	✓	✓
django-11964	–	–	✓	–	✓
django-11999	✓	✓	✓	✓	✓
django-12039	✓	✓	✓	✓	✓
django-12050	✓	✓	✓	✓	✓
django-12143	✓	✓	✓	✓	✓
django-12155	✓	✓	✓	✓	✓
django-12193	–	–	–	–	✓
django-12209	✓	✓	✓	✓	✓
django-12262	✓	✓	✓	✓	✓
django-12273	–	–	✓	✓	–
django-12276	✓	✓	✓	✓	✓
django-12304	✓	✓	✓	✓	✓
django-12308	–	–	–	–	✓
django-12325	–	–	–	–	✓
django-12708	✓	✓	✓	✓	✓
django-12713	–	✓	✓	✓	✓
django-12774	–	✓	✓	✓	✓
sphinx-doc (23 instances)					
sphinx-7454	✓	–	–	–	–
sphinx-7757	✓	✓	✓	✓	✓
sphinx-7889	✓	–	–	–	–
sphinx-7910	✓	–	–	–	–
sphinx-7985	–	–	–	–	✓
sphinx-8035	–	✓	✓	✓	✓
sphinx-8120	✓	–	–	–	–
sphinx-8269	✓	✓	✓	✓	✓
sphinx-8459	✓	–	–	–	–
sphinx-8475	–	✓	✓	✓	✓
sphinx-8548	–	–	–	✓	✓
sphinx-8551	–	✓	✓	✓	✓
sphinx-8638	–	✓	✓	✓	✓
sphinx-8721	–	–	✓	✓	✓
sphinx-9230	–	✓	✓	✓	✓
sphinx-9281	–	✓	✓	✓	✓
sphinx-9320	–	✓	✓	✓	✓
sphinx-9367	–	✓	✓	✓	✓
sphinx-9698	–	✓	✓	✓	✓
sphinx-10323	–	–	–	✓	✓
sphinx-10449	✓	–	–	–	–
sphinx-10466	✓	✓	✓	✓	✓
sphinx-10673	✓	✓	✓	✓	✓
Total (out of 50)	**25**	**30**	**34**	**35**	**39**

Table 28: The six instances where Pattern A and Pattern B reached different outcomes. Three of the five Pattern-B-only wins are instances no other arm resolved.

Instance	Winner	Resolved by other arms?	Notes
django-11964	B	MAST also resolved	A failed where B and MAST succeeded
django-12193	B	No other arm resolved	B-only win
django-12273	A	MAST also resolved	B's only loss in disagreements
django-12308	B	No other arm resolved	B-only win
django-12325	B	No other arm resolved	B-only win
sphinx-7985	B	No other arm resolved	B-only win