WHEN SUCCESSFUL STRATEGIES FAIL: ADAPTATION TO ENVIRONMENTAL NOVELTY IN TERMINAL AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents increasingly solve long-horizon tasks by autonomously interacting with their environment. In doing so, their strategies rely on assumptions about that environment: which resources and tools exist, where they are located, and how they behave. When these assumptions no longer hold, reliable agents must detect the change and adapt while pursuing the same goal. We study this *adaptation capability* through *environmental novelty*: a change that keeps the task objective fixed while invalidating an assumption underlying an otherwise successful trajectory. We introduce AGNI, an automated pipeline that extracts trajectory-relevant assumptions, injects targeted environmental changes, and validates that the resulting novel tasks remain solvable. Across three terminal benchmarks, AGNI produces diverse novelties spanning resources, interfaces, constraints, and execution semantics. Evaluating multiple LLM agents reveals a substantial adaptation gap between base and novel tasks. Trajectory analysis suggests that agents often encounter evidence of the change but fail to diagnose its cause and revise their strategy. Finally, post-training for environmental novelty improves adaptation to held-out novel tasks while also improving performance on base tasks. Our results highlight a gap between task competence and adaptive capability and motivate environmental variation as a core dimension of agent training and evaluation.

1 INTRODUCTION

LLM agents increasingly pursue long-horizon tasks from high-level goals rather than procedural instructions (Yang et al., 2024; Xie et al., 2024). This increased autonomy requires agents to construct and execute task-solving strategies. These strategies often rely on agents' implicit assumptions about their environment, such as available resources, interface behavior, and other agents' behavior. Since these assumptions are often unspecified by the task, whether they hold may only become clear through interaction. Reliable agents must, therefore, go beyond executing an initial strategy and adapt when their assumptions fail. They must identify the relevant environmental changes, revise their strategy, and continue pursuing the original goal (Langley, 2020; Kejriwal et al., 2024).

We study this adaptation capability through environmental novelty: a change in the environment that invalidates an assumption used by a successful trajectory, while preserving the task objective. We particularly focus on terminal agents for two reasons. First, terminal agents span heterogeneous tasks, from simple file processing to advanced machine learning, and can interact with open-ended tools and resources (Merrill et al., 2026). This diversity makes them a suitable testbed for studying whether LLMs can reason effectively to adapt across diverse strategies and environments. Second, adaptation in software environments is practically important, since software continually evolves through changes in interfaces, dependencies, configurations, and workflows (Dig & Johnson, 2006).

We formalize environmental novelty by separating the task objective from the software environment in which it is achieved. Given a base terminal task and a verified successful trajectory, the trajectory may rely on assumptions not specified by the task instruction. These assumptions can include a writable directory, an installed dependency version, or a particular tool interface. An environmental novelty modifies the environment so that such an assumption no longer holds, while preserving the instruction, success criterion, and solvability. Since the novelty is undisclosed to the agent, it must infer the changed condition through interaction, revise its strategy, and complete the original task. The performance gap between base and novel tasks reflects a model's ability to adapt.

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

Terminal tasks are well suited for studying adaptation, but their heterogeneity and open-endedness also make base–novel pairs difficult to construct. A task can often be solved using a diverse set of command sequences and resources. As a result, the relevant environmental assumptions are often implicit and cannot be inferred from the task instructions alone. Arbitrary changes in the environment may leave a successful strategy unaffected, admit trivial workarounds, or break task solvability. We therefore introduce AGNI (Assumption-Guided Novelty Injection), an LLM-based pipeline that starts from verified successful trajectories, identifies the environmental assumptions supporting their strategies, and invalidates one while preserving the task objective and solvability. AGNI retains variants where the original strategy no longer suffices but an adapted trajectory can succeed, enabling controlled evaluation, trajectory-level analysis, and post-training under environmental novelty.

We study adaptation by evaluating seven diverse LLMs on AGNI-generated task pairs. We generate task pairs from three terminal datasets of varying difficulty: Endless Terminals, TB-Lite, and TB-2. We find that models consistently degrade on novel tasks, and higher base-task performance does not reliably predict adaptation capability. We further test whether increased test-time reasoning improves adaptation and find that it often helps, but not consistently across models. We also analyze whether disclosing the environmental change in the task instruction recovers the adaptation gap, and find that it recovers a substantial fraction across models. However, this recovery cannot be attributed simply to whether agents observe the novelty during interaction. Our trajectory-level analysis shows that stronger adapters are better at turning observed novelty into a diagnosis and a revised strategy, while weaker adapters more often repeat failed approaches and terminate early.

We next ask whether post-training can improve adaptation to novelty. Standard agentic post-training (Shao et al., 2024; Shrivastava et al., 2026) improves performance on both base and novel terminal tasks, but leaves a significant adaptation gap. We develop two complementary approaches to narrow this gap. First, we augment training with novel environments that require agents to solve familiar tasks under changed conditions (Devries & Taylor, 2017; Park et al., 2019; Liu et al., 2026c). Second, we introduce explicit *meta-action* reasoning, where agents reason about the strategic intent of each action, such as inspecting, diagnosing, revising, or verifying. Both novelty-augmented training and meta-action reasoning provide improvements on held-out novel tasks, with gains transferring to an additional terminal benchmark. We also study process rewards over meta-actions, finding that they reshape agent behavior but provide limited improvement in adaptation.

Overall, our work highlights adaptation as a key capability for reliable autonomous agents. We make three contributions. First, we formalize environmental novelty as the controlled invalidation of strategy-relevant assumptions and introduce AGNI, a scalable pipeline for injecting such novelties into terminal tasks. Second, we use AGNI-generated environments to study adaptation and how it unfolds during agentic interaction. Third, we develop post-training approach based on data augmentation and meta-action reasoning to improve adaptation. Together, our results motivate environmental variation as a core axis for training and evaluating terminal agents.

2 Related Work

Open-World Learning and Adaptation. Open-world learning studies how agents adapt to environmental novelty (Langley, 2020; Kejriwal et al., 2024). Prior work decomposes adaptation into novelty detection, diagnosis, and behavioral updates (Muhammad et al., 2021; Loyall et al., 2025). Crucially, novelty is relative to the agent and task: an environmental change may invalidate one strategy while being irrelevant to another (Boult et al., 2021; Goel et al., 2024). Prior work largely focuses on embodied, simulated, or game environments with structured world models and adaptation components. In contrast, we study general-purpose LLM agents operating through open-ended programmatic interfaces, where strategies and assumptions are implicit in interaction trajectories.

Robustness Evaluation in LLM Agents. Recent work evaluates LLM-agent robustness under perturbations to tool availability (Liu et al., 2026a;b), tool interfaces (Tian et al., 2026; Wu et al., 2026), workspace contents (Li et al., 2026a; Mahmud et al., 2026), and runtime conditions (Jha et al., 2026). These works largely predefine perturbation families. AGNI instead derives perturbations from assumptions revealed by successful trajectories, making novelty strategy-dependent. This is important for terminal agents, where heterogeneous tasks and open-ended tools and resources make relevant environmental dependencies difficult to enumerate in advance. Furthermore, since AGNI targets a known assumption, it enables a finer-grained analysis of adaptation within the interaction trajectory.

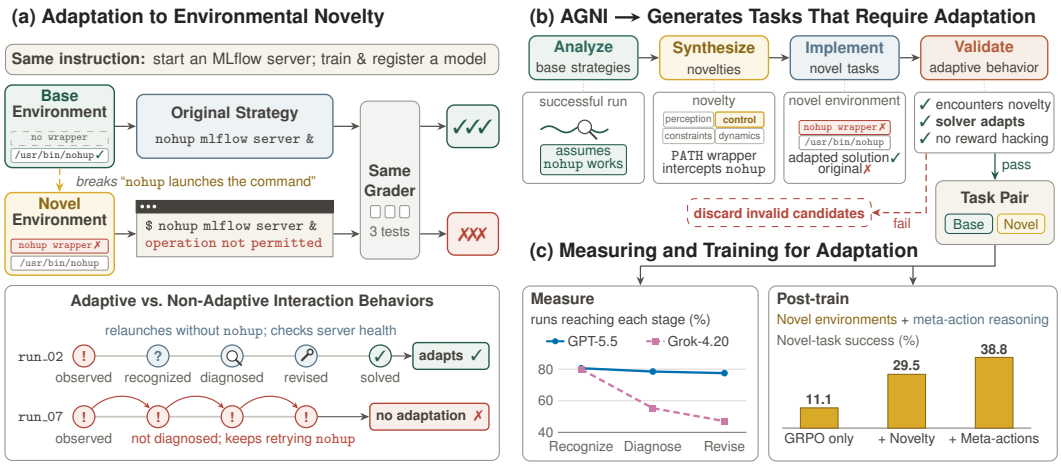


Figure 1: **Adaptation to Environmental Novelty in Terminal Agents.** **(a)** Novelty can disrupt an agent's original strategy and recovery depends on adaptive behaviors during interaction (example from Appendix C.7). **(b)** AGNI injects novelties into base environments such that task success requires adaptation, while preserving task objective and solvability. **(c)** We use AGNI-generated tasks to measure adaptation, analyze adaptive behaviors, and post-train agents to improve adaptation.

Post-training for Adaptive LLM Agents. Recent work post-trains LLM agents to recover from perturbations in tool-use (Vuddanti et al., 2025; Wu et al., 2026), text-game (Ye et al., 2026), and GUI environments (Sun et al., 2026). These works typically train on predefined perturbations and supervise recovery using task outcomes, recovery demonstrations, supplied diagnoses, and incentives for reflection and exploration. In contrast, we train on trajectory-grounded perturbations and provide process-level adaptation guidance through self-reported and rewarded meta-actions, including inspecting, diagnosing, revising, and verifying.

Software Evolution and Resilience Testing. Software routinely evolves in ways that break previously valid workflows. Interfaces and dependencies change (Dig & Johnson, 2006; Kula et al., 2017; Thomas et al., 2021), configuration errors arise (Yin et al., 2011; Xu & Zhou, 2015), and system state changes can also expose failures (Yuan et al., 2014). These observations motivate our study of adaptation to environmental novelty. Analogous to resilience testing in software systems (Basiri et al., 2016), we study whether terminal agents remain reliable as their software environment changes.

A more detailed discussion of related work and additional references is provided in Appendix B.

3 Environmental Novelty and Adaptation in Terminal Tasks

We now formalize environmental novelty and adaptation in terminal tasks.

Terminal Task. We represent a terminal task as $\mathcal{T} = (I, E, G)$, where I is the instruction, E is the execution environment, typically a Docker container, and G is a task-specific outcome grader implemented with deterministic evaluation scripts. Given $\mathcal{T}$, an agent executes a task-solving strategy through interaction with E, producing a trajectory $\tau = (o_0, r_1, a_1, o_1, \ldots, r_T, a_T, o_T)$, where o_0 contains I and the initial observation, and r_t, a_t, and o_t denote reasoning, action, and resulting observation at turn t. The task succeeds if $G(\tau, E) = 1$ and fails otherwise. This formulation separates what the agent must accomplish, specified by I and G, from the environmental conditions under which it must do so, specified by E.

Environmental Novelty. A successful trajectory τ may depend on environmental assumptions that are not specified by I, reflecting the agent's learned priors about how such environments typically behave. Let $\mathcal{A}(\tau, E)$ denote the set of trajectory-relative environmental assumptions on which the success of τ depends, and let $\alpha \in \mathcal{A}(\tau, E)$ be one such assumption. An *environmental novelty* transforms E into E' such that α no longer holds, while preserving the task objective and success criterion. For example, τ may write an artifact to `/workspace` while implicitly assuming that the

directory is writable. A novelty may make this directory read-only. Thus, the strategy underlying τ succeeds in E but is no longer sufficient in E'.

Adaptation. Once an environmental assumption is invalidated, an adaptable agent must revise its strategy while still achieving the task objective. This may require recognizing evidence of novelty, diagnosing which assumption has been invalidated, and revising the strategy appropriately. We focus on the autonomous setting, where the agent must discover the environmental change through interaction and recover without explicit human guidance about the novelty or how to respond to it. Accordingly, we define adaptation as *task-consistent recovery*: revising the affected strategy in response to the changed environmental condition and successfully completing the original task, without requiring any particular recovery strategy. In the preceding example, the agent could adapt by restoring write access to `/workspace` and then completing the original task.

To measure adaptation, we compare an agent's performance across three task variants:

$$\mathcal{T}_{\text{base}} = (I, E, G), \qquad \mathcal{T}_{\text{novel}} = (I, E', G), \qquad \mathcal{T}_{\text{disc}} = (I_{\text{disc}}, E', G).$$

A *controlled task pair* $(\mathcal{T}_{\text{base}}, \mathcal{T}_{\text{novel}})$ consists of the original environment E and a solvable modified environment E' that invalidates an assumption underlying a verified successful trajectory in E. The pair preserves the instruction and success criterion while changing an assumption relied upon by τ. In the disclosed variant, I_{disc} explicitly states the novelty but provides no guidance on how to adapt.

Let P_{base}, P_{novel}, and P_{disc} denote the agent's pass@1 rates on the three variants. We define the *adaptation gap* Δ_{adapt}, *disclosure gain* Δ_{disc}, and *recovery fraction* ρ_{rec} as

$$\Delta_{\text{adapt}} = P_{\text{base}} - P_{\text{novel}}, \qquad \Delta_{\text{disc}} = P_{\text{disc}} - P_{\text{novel}}, \qquad \rho_{\text{rec}} = \Delta_{\text{disc}} / \Delta_{\text{adapt}}.$$

A larger adaptation gap indicates greater degradation under environmental novelty, while a smaller gap suggests better adaptation. We interpret this gap alongside base and novel success rates to assess a model's adaptive capability, since the gap also depends on base-task competence. By contrast, disclosure gain measures the performance recovered when the novelty is made explicit. This gain helps distinguish the difficulty of identifying the relevant environmental change through interaction from the difficulty of adapting once it is known. For a positive adaptation gap, ρ_{rec} measures the fraction of lost performance recovered through disclosure.

4 AGNI: Assumption-Guided Novelty Injection

Terminal tasks are heterogeneous across domains and open-ended in the tools, resources, and strategies available to agents. This diversity makes enumerating novelties from the task instruction alone difficult; and an arbitrary environmental change may be irrelevant to the strategy or make the task unsolvable. We therefore introduce AGNI, a scalable LLM-based pipeline that injects strategy-relevant novelties while preserving task solvability. Figure 1 illustrates AGNI, Algorithm 1 presents the procedure, and Appendix C provides full details. We briefly describe the key components below.

Generating Trajectory-Relative Novelties. For each base task, AGNI samples multiple trajectories from a capable *solver* model and retains tasks with at least one successful trajectory. Then, an *analyzer* model examines the base task (I, E, G) and sampled trajectories to characterize solver strategies and underlying environmental assumptions. Using these assumptions, AGNI identifies applicable novelty dimensions and generates candidate novelty descriptions that target the corresponding environmental conditions. We consider four novelty dimensions: what the agent observes (*perception*), how it can act (*control*), what actions are permitted (*constraints*), and how the environment responds (*dynamics*). AGNI then de-duplicates and filters candidates for viability, as assessed by the analyzer. It then implements and validates the corresponding environments, as described next.

Implementing and Validating Novel Tasks. Building on prior work on terminal task generation Gandhi et al. (2026); Ivison et al. (2026), AGNI implements each novelty by modifying the base environment E to obtain E' while keeping the instruction I and grader G unchanged. It also generates an adapted oracle solution for the novel variant. AGNI retains a novelty implementation only when the adapted oracle solution succeeds and the original oracle solution fails. This ensures that the task remains solvable while the novelty invalidates the base strategy. AGNI then samples solver trajectories in E' and retains tasks with at least one successful trajectory. Finally, it analyzes successful trajectories turn by turn to verify that the solver encounters the novelty and adapts to it. We also reject cases where the solver exhibits reward hacking. Together, these checks verify solvability and provide evidence of legitimate adaptation in sampled successful trajectories.

Algorithm 1 AGNI: **A**ssumption-**G**uided **N**ovelty **I**njection. AGNI generates validated base–novel task pairs in four stages by: (a) analyzing base-task strategies; (b) synthesizing novelty descriptions; (c) implementing and validating novel tasks; and (d) validating adaptive behavior.

Require: Base task $\mathcal{T}_{\text{base}} = (I, E, G)$; oracle solution S; solver model M_{S}; analyzer model M_{A}; novelty injector model M_{N}; discarded-novelty history $\mathcal{H}$; trajectory sampling budget K; novelty specification budget N_{spec}; retention budget N_{keep}

Ensure: Validated task pairs $\mathcal{V} = \{(\mathcal{T}_{\text{base}}, \mathcal{T}_{\text{novel}})\}$

$\mathcal{R}_{\text{base}} \leftarrow \{(\tau^{(k)}_{\text{base}}, G(\tau^{(k)}_{\text{base}}, E)) : \tau^{(k)}_{\text{base}} \sim M_{\text{S}}(I, E)\}_{k=1}^{K}$ ▷ sample and grade base trajectories
if $\nexists\,(\tau, r) \in \mathcal{R}_{\text{base}}$ such that $r = 1$, **return** $\emptyset$ ▷ ensure base-task solvability
$(\mathcal{B}, \mathcal{A}) \leftarrow M_{\text{A}}(\mathcal{T}_{\text{base}}, S, \mathcal{R}_{\text{base}})$ ▷ extract resource-use behavior and assumptions

$\mathcal{D} \leftarrow M^{\text{dim}}_{\text{N}}(\mathcal{T}_{\text{base}}, \mathcal{B}, \mathcal{A})$ ▷ identify applicable novelty dimensions
for $d \in \mathcal{D}$ **do**
  $\mathcal{C}_d \leftarrow M^{\text{spec}}_{\text{N}}(\mathcal{T}_{\text{base}}, \mathcal{B}, \mathcal{A}, d, \mathcal{H}_d, N_{\text{spec}})$ ▷ generate novelty descriptions
end for
$\mathcal{C} \leftarrow \text{Select}\big(\text{Filter}\big(\bigcup_{d\in\mathcal{D}} \mathcal{C}_d\big)\big)$ ▷ de-duplicate and filter by viability

$\mathcal{V} \leftarrow \emptyset$
for $c \in \mathcal{C}$ **do**
  $(E'_c, S'_c) \leftarrow M^{\text{impl}}_{\text{N}}(\mathcal{T}_{\text{base}}, S, c)$ ▷ implement novelty and adapted solution
  if $G(S'_c, E'_c) = 0$, **continue** ▷ ensure novel-task solvability
  if $G(S, E'_c) = 1$, **continue** ▷ ensure novelty invalidates the base strategy
  $\mathcal{R}_c \leftarrow \{(\tau^{(k)}_c, G(\tau^{(k)}_c, E'_c)) : \tau^{(k)}_c \sim M_{\text{S}}(I, E'_c)\}_{k=1}^{K}$ ▷ sample and grade novel trajectories
  if $\nexists\,(\tau, r) \in \mathcal{R}_c$ such that $r = 1$, **continue** ▷ ensure solver can succeed on novel task

  $v_c \leftarrow M^{\text{obs}}_{\text{A}}(c, \mathcal{R}_{\text{base}}, \mathcal{R}_c)$ ▷ evaluate expected trajectory change
  if $v_c = 0$, **continue** ▷ verify the expected trajectory-level change
  $\mathcal{L}_c \leftarrow \text{AnalyzeTrajectories}(c, \mathcal{R}_c)$ ▷ label novelty handling stages per trajectory
  if $\neg\forall\,\ell \in \mathcal{L}_c : \text{Solved}(\ell) \Rightarrow \text{Adapt}(\ell)$, **continue** ▷ ensure successful trajectories adapt
  if $\exists\,\ell \in \mathcal{L}_c : \text{RewardHack}(\ell)$, **continue** ▷ reject observed reward hacking
  $\mathcal{T}'_c \leftarrow (I, E'_c, G)$
  $\mathcal{V} \leftarrow \mathcal{V} \cup \{(\mathcal{T}_{\text{base}}, \mathcal{T}'_c)\}$ ▷ retain validated task pair
  if $|\mathcal{V}| = N_{\text{keep}}$, **break**
end for
return $\mathcal{V}$

Curating the Final Datasets. We run AGNI on three base-task sources: Endless Terminals (ET) (Gandhi et al., 2026), TB-Lite (OpenThoughts-Agent, 2026), and Terminal-Bench 2 (TB-2) (Merrill et al., 2026). We generate base–novel task pairs for evaluation from all three sources and use ET to construct the training set at scale, as TB-Lite and TB-2 are evaluation-only datasets. For evaluation, we construct evaluation sets through joint stratification across dimensions capturing difficulty and real-world plausibility. We stratify by base-task performance P_{base}, relative novelty-induced difficulty $\delta_{\text{adapt}} = (P_{\text{base}} - P_{\text{novel}})/P_{\text{base}}$, and real-world plausibility. We estimate P_{base} and P_{novel} using a weak solver model and use an LLM judge to score whether each novelty could plausibly occur in real-world deployments. This yields 4,000 training pairs from ET and evaluation sets of 140 ET-eval, 118 TB-Lite, and 87 TB-2 pairs.

Implementation Details. All AGNI model components use GPT-5.5, with high reasoning effort for the solver and medium effort otherwise. We allocate higher effort to the solver because its capability bounds the difficulty of retained tasks. More details in Appendix C.

5 Evaluating Adaptation under Environmental Novelty

We now study adaptation in terminal agents using the three evaluation sets curated in Section 4.

Setup. We evaluate a diverse set of LLMs, including GPT-5.5, GPT-5.4-mini, Kimi-2.6, DeepSeek-V4-Flash, Grok-4.20, GPT-OSS-120B, and Mistral-Large-3. Using a fixed Terminus-2 harness, we sample $K = 5$ trajectories per model-task pair. More details in Appendix E.1.

Model	ET-eval Base	ET-eval Novel	ET-eval Δ_{adapt}	TB-Lite Base	TB-Lite Novel	TB-Lite Δ_{adapt}	TB-2 Base	TB-2 Novel	TB-2 Δ_{adapt}
GPT-5.5	97.0	74.5	22.5	93.2	71.7	21.5	88.8	79.7	9.1
GPT-5.4-mini	94.5	61.9	32.6	65.0	39.5	25.5	75.9	46.6	29.3
Kimi-2.6	92.7	74.7	18.0	83.6	58.3	25.3	73.8	61.4	12.4
DeepSeek-V4-Flash	89.4	74.2	15.1	70.2	49.3	20.8	66.0	54.0	12.0
GPT-OSS-120B	78.6	29.4	49.2	52.7	17.6	35.1	28.0	13.3	14.7
Grok-4.20	71.9	35.5	36.4	48.3	16.6	31.7	40.0	16.6	23.4
Mistral-Large-3	64.5	23.6	40.9	32.4	10.3	22.1	21.1	6.9	14.3
Average	84.1	53.4	30.7	63.6	37.6	26.0	56.2	39.8	16.5

Table 1: Environmental novelty consistently degrades pass@1 (%) across models and datasets. Moreover, models with stronger base performance do not always exhibit smaller adaptation gaps. Here, Δ_{adapt} denotes the Base-Novel pass@1 gap in percentage points.

Q5.1. Are Curated Novelties Relevant Across Models? Our base-novel pairs are constructed from assumptions derived from GPT-5.5 trajectories. This raises the question of whether these novelties also affect the strategies used by other models. We measure interact@1, the fraction of novel-task trajectories that interact with the changed component. Across the three evaluation sets, interact@1 remains high (90.7–99.0%), while successful trajectories that bypass the novelty are rare ($\leq 2.7\%$). Thus, the curated novelties generally affect the strategies of other models rather than being specific to the trajectories used for construction. More details in Appendix E.2.

Q5.2. How Does Novelty Affect Performance? Table 1 reports pass@1 (%) on the base and novel versions of all three datasets. First, environmental novelty substantially reduces performance. Average pass@1 drops from 84.1% to 53.4% on ET-eval, from 63.6% to 37.6% on TB-Lite, and from 56.2% to 39.8% on TB-2. These correspond to adaptation gaps of 30.7, 26.0, and 16.5 percentage points, respectively. Second, this degradation occurs across models and datasets, suggesting that it is not specific to any particular model or benchmark. Third, strong base performance does not necessarily translate to better adaptation. For instance, on ET-eval, GPT-5.4-mini achieves 94.5% on base tasks, compared with 92.7% for Kimi-2.6 and 89.4% for DeepSeek-V4-Flash. Under novelty, however, GPT-5.4-mini drops to 61.9%, while Kimi-2.6 and DeepSeek-V4-Flash achieve 74.7% and 74.2%, respectively. Likewise, GPT-OSS-120B outperforms Grok-4.20 on base tasks (78.6% vs. 71.9%) but falls below it on novel tasks (29.4% vs. 35.5%). These ranking reversals persist even after normalizing performance degradation by base performance. Overall, performance in the base environment can overestimate an agent's ability to adapt to environmental novelty.

Q5.3. How Does Reasoning Effort Affect Adaptation? Adaptation requires recognizing environmental changes, diagnosing invalid assumptions, and revising strategy. These adaptation stages may benefit from more test-time reasoning between interactions. Table 2 tests this hypothesis by reporting pass@1 (%) on ET-eval across varying reasoning efforts. We observe that moving from low to high reasoning effort increases novel pass@1 by 20.4 percentage points for GPT-5.4-mini and 13.1 points for GPT-OSS-120B. Their adaptation gaps also decrease, suggesting that greater reasoning effort improves adaptation for these models. By contrast, for Kimi-2.6, medium effort yields the highest novel-environment performance, while low effort yields the smallest adaptation gap. Thus, Kimi-2.6 shows a weaker, non-monotonic relationship between reasoning effort and adaptation. This is consistent with prior findings that increased reasoning effort may not always improve performance and can reduce accuracy due to overthinking (Zhou et al., 2026). Overall, reasoning effort often improves adaptation, but the gains can be model-dependent.

Model	Effort	Base	Novel	Δ_{adapt}
GPT-5.4-mini	Low	85.4	41.5	44.0
	Medium	91.4	53.8	37.6
	High	94.5	61.9	32.6
GPT-OSS-120B	Low	75.0	22.2	52.8
	Medium	78.6	29.4	49.2
	High	84.2	35.3	48.9
Kimi-2.6	None	83.0	65.5	17.6
	Low	90.1	72.7	17.4
	Medium	92.7	74.7	18.0
	High	91.8	70.8	21.0

Table 2: Pass@1 (%) on ET-eval across reasoning efforts. Higher effort reduces Δ_{adapt} for GPT-5.4-mini and GPT-OSS-120B but has a non-monotonic effect for Kimi-2.6.

Q5.4. How Much of the Adaptation Gap Does Novelty Disclosure Recover? To measure how much of the adaptation gap can be recovered when the novelty is known in advance, we disclose the exact environmental change in the task instruction. This allows the agent to plan for the changed environment directly, rather than encountering the change, diagnosing it, and revising a strategy built on its assumption. Importantly, this setting does not measure adaptation itself, but how much of the adaptation gap is recoverable when the changed condition is known. As shown in Figure 2, disclosure improves performance across all evaluated models, recovering 71.7% of the adaptation gap on average, with recovery fractions ranging from 59.2% to 82.1%. However, degradation under undisclosed novelty does not necessarily mean that agents fail to observe the change. As the next question shows, agents often observe the novelty but differ in how well they diagnose and respond to it. Overall, explicit disclosure recovers much of the adaptation gap, and the remaining gap reflects the difficulty of acting successfully even when the novelty is known.

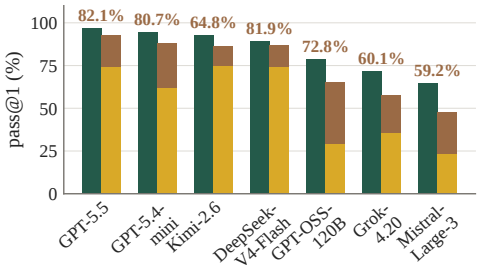
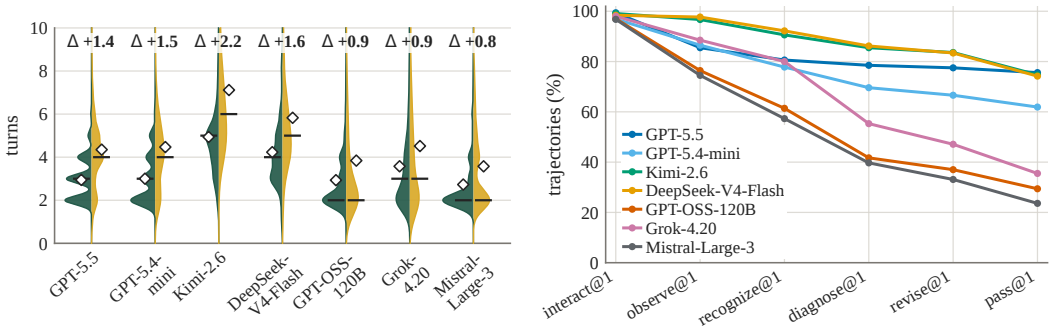


Figure 2: Novelty disclosure substantially recovers the adaptation gap on ET-eval. Bars show pass@1 for **base** and **novel** tasks, with **disclosure gain** Δ_{disc}; labels report corresponding recovery fraction ρ_{rec}.



(a) Interaction length (# turns)

(b) Adaptation stages

Figure 3: Interaction behaviors associated with adaptation on ET-eval. (a) Better adapting models show larger increases in interaction length when going from **base** to **novel** tasks; lines mark medians, diamonds means, and labels report mean novel-base difference. (b) Better adapting models show more reliable progress from recognizing novelty to diagnosing it and revising their strategy.

Q5.5. Which Interaction Behaviors Are Associated with Better Adaptation? Novelty disclosure reveals how much performance is recoverable when the novelty is known, but not where adaptation fails when it is undisclosed. We therefore compare the interaction behaviors of strongly and weakly adapting models. To answer this, we first analyze interaction length, measured by the number of turns, across ET-eval base-novel pairs. Figure 3(a) shows that all models interact longer on novel tasks. Stronger adapters (GPT-5.5, Kimi-2.6, and DeepSeek-V4-Flash) use 1.4–2.2 additional turns on average, compared with less than one additional turn for weaker adapters (GPT-OSS-120B, Grok-4.20, and Mistral-Large-3). Thus, better adaptation is associated with sustaining longer interactions under novelty. Second, we perform a turn-level analysis of ET-eval novel-task trajectories using a GPT-5.5-based judge. At each turn, we annotate the agent's novelty-handling stage as *interacted*, *observed*, *recognized*, *diagnosed*, or *revised*. We also assign the agent's action a strategic intent, such as *inspect*, *diagnose*, *retry*, or *revise*. Appendix F provides details on the annotation process. Figure 3(b) reports the fraction of trajectories that reach each adaptation stage. GPT-5.5 and Grok-4.20 recognize the novelty at similar rates (80.6% vs. 80.0%), but diverge sharply afterward: GPT-5.5 reaches diagnosis and strategy revision in 78.5% and 77.5% of trajectories, compared with 55.3% and 47.1% for Grok-4.20. This suggests that the adaptation gap is not explained simply by whether agents recognize the novelty. Successful adaptation also requires turning recognized novelty into a diagnosis and a revised strategy. Appendix Tables 20 and 22 further show that weakly adapting models more often retry failed approaches, whereas stronger adapters more often transition from recognition to diagnosis and strategy revision.

6 Improving Adaptation with Agentic Post-Training

We now ask how adaptation changes during agentic post-training and whether simple changes to the training data and objective can improve it.

Setup. We use Qwen3-8B (Qwen-8B) and Qwen3-14B (Qwen-14B) as our base models. We curate training data from Endless Terminals by splitting 8,000 tasks into two disjoint 4,000-task sets, T_1 and T_2. AGNI generates novel variants T_1' of T_1. This yields three equal-sized datasets:

$$D_{\text{clean}} = T_1 \cup T_2, \qquad D_{\text{novel-unpaired}} = T_2 \cup T_1', \qquad D_{\text{novel-paired}} = T_1 \cup T_1'.$$

The unpaired regime exposes the model to novelty across disjoint underlying tasks, whereas the paired regime presents the same tasks in both base and novel environments. The paired construction, therefore, provides a contrast in the operating environment while keeping the underlying task fixed. We evaluate on the ET-eval and TB-Lite base–novel pairs curated in Section 4; all evaluation tasks are disjoint from the training tasks. We select checkpoints using a separate set of 100 ET base tasks that are disjoint from both training and ET-eval. We study GRPO (Shao et al., 2024) and ECHO (Shrivastava et al., 2026) as post-training approaches, with a primary focus on GRPO. More details in Appendix G.

Q6.1. Does Generic Agentic Post-Training Improve Adaptation? We train Qwen-8B and Qwen-14B on D_{clean} using GRPO and ECHO and compare both post-trained models with the base model. Figure 4 shows that agentic post-training improves both base and novel performance, but does not narrow the adaptation gap. On ET-eval, Qwen-8B Novel pass@1 increases from 7.1 to 11.1 with GRPO and 26.3 with ECHO, while Base pass@1 increases more sharply from 36.3 to 56.9 and 68.7, respectively. Consequently, the adaptation gap remains large, increasing from 29.2 points to 45.8 with GRPO and 42.4 with ECHO. We observe a similar trend in TB-Lite and with Qwen-14B (Appendix G).

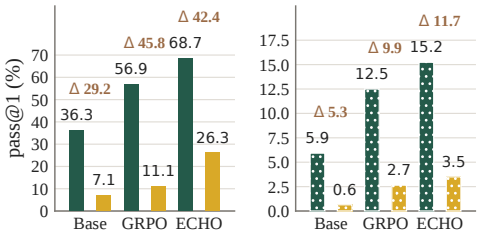


Figure 4: Generic post-training raises absolute performance but does not close the adaptation gap. Bars show **base** and **novel** pass@1; ET-eval is solid and TB-Lite dotted. Δ reports the adaptation gap.

Q6.2. Does Novelty-Augmented Training Improve Adaptation? We train Qwen-8B with GRPO on D_{clean}, $D_{\text{novel-unpaired}}$, and $D_{\text{novel-paired}}$. Figure 5 shows that novelty-augmented training substantially improves held-out ET-eval novel performance while largely preserving base performance. $D_{\text{novel-unpaired}}$ increases novel pass@1 from 11.1 to 27.6 with similar base performance (56.9 vs. 56.2), reducing the adaptation gap from 45.8 to 28.6 points. $D_{\text{novel-paired}}$ further increases novel pass@1 to 29.5 and base pass@1 to 59.8, though its adaptation gap is slightly larger than unpaired training's (30.4 points). The gain in novel pass@1 suggests that the paired contrast may help the model learn when environmental changes require a different strategy. The gains also transfer to TB-Lite, where novel pass@1 increases from 2.7 under clean training to 4.1 and 5.0 with novel-unpaired and novel-paired training, respectively. We observe a similar qualitative trend in Qwen-14B (Appendix G).

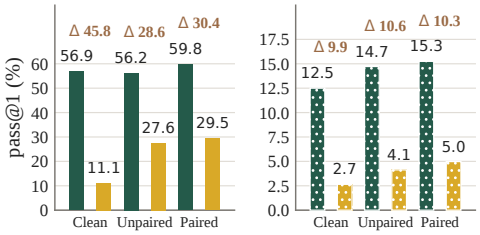


Figure 5: Novelty-augmented training narrows the ET-eval adaptation gap and transfers to TB-Lite. Bars show **base** and **novel** pass@1; ET-eval is solid and TB-Lite dotted. Δ reports the adaptation gap.

Q6.3. Does Meta-Action Reasoning Improve Adaptation? Q5.5 (Section 5) shows that stronger adapters more reliably turn recognized novelties into diagnoses and strategy revisions. Motivated by this, we ask whether explicitly reasoning about the intent behind each action, which we call the *meta-action*, improves adaptation. We modify the system prompt to generate one of five meta-actions alongside each action: Execute, Inspect, Diagnose, Revise, or Verify. We refer to this as the Meta prompt; exact prompt changes are provided in Appendix G.4. We restrict this experiment to paired training data, which performs relatively better on novel tasks.

Method	Base	Novel	Δ_{adapt}
GRPO on $D_{\text{novelty-paired}}$	59.82	29.46	30.36
+ META Prompt	61.25	32.59	28.66
+ META Prompt Training	64.29	37.86	26.43
+ Recovery & Verify	69.55	38.00	31.55
+ Diagnosed Exploration	53.93	25.14	28.79
+ Exploration & Completion	66.96	38.80	28.16

Table 3: Meta-action reasoning improves adaptation; process rewards provide limited additional gains. Table reports ET-eval pass@1 (%) and adaptation gap.

We first apply the META prompt at inference time to the paired-data GRPO model, which raises held-out ET-eval novel pass@1 from 29.46% to 32.59%. We then train a separate model from the same base checkpoint on paired data with the META prompt, keeping the reward purely outcome-based. This further improves novel pass@1 to 37.86% (Table 3). Base pass@1 also increases from 59.82% to 61.25% and 64.29%, respectively, though by a smaller magnitude. We further explore adding simple process rewards over these meta-actions. We consider three process level rewards: *Recovery & Verify*, *Diagnosed Exploration*, and *Exploration & Completion* while holding all other training settings fixed. Details of the design of these rewards are provided in Appendix G. *Recovery & Verify* and *Exploration & Completion* yield only small improvements in novel pass@1 over META training alone (38.00% and 38.80% vs. 37.86%), while *Diagnosed Exploration* reduces it to 25.14%. These rewards change the distribution of meta-actions (Appendix G.3), but these changes translate into only limited improvements on novel tasks.

7 Discussion and Limitations

As agents become more autonomous, users increasingly delegate not only task execution but also how those tasks are carried out. This makes an agent's strategy and behavior as important as task success. We study one aspect of this broader problem: whether an agent can reliably revise its strategy when environmental assumptions fail during execution. Other aspects include resource efficiency (Garikaparthi, 2026; Foster et al., 2026) and safety (Vijayvargiya et al., 2025; Li et al., 2026b): a successful strategy may waste substantial time or compute, and task completion may involve harmful actions or violations of operational constraints. Agent evaluation should therefore assess both task outcomes and behavior throughout execution.

AGNI curates task pairs by invalidating trajectory-relative assumptions while preserving task objectives and solvability. A broader direction is to extend this pipeline to construct controlled settings for studying a wider range of agent behaviors. For example, environments could test whether agents respect access boundaries around planted honeypots, consider alternative strategies under uncertainty, or abandon an unpromising strategy as contrary evidence accumulates. This could enable controlled evaluation of how agents reason during execution, rather than evaluating only whether they succeed.

Our evaluation captures a bounded form of adaptation: recovery from curated environmental changes in terminal tasks. It does not establish reliable adaptation under multiple changes, in very long-horizon tasks, or in open-ended deployment; measured failure rates only reflect performance on these simple curated tests. LLM-assisted environment curation and evaluation may also introduce biases that our checks cannot fully eliminate. Finally, computational costs restrict our post-training experiments to a limited set of models and training settings. Despite these limitations, our results motivate environmental variation as a core dimension of agent training and evaluation.

8 Conclusion

We study adaptation in terminal agents through environmental novelty. To evaluate and train adaptation in controlled environments, we introduce AGNI, a scalable LLM-based pipeline to inject novelty. AGNI invalidates assumptions underlying successful trajectories while preserving task objectives and solvability. Across models and datasets, novelty substantially reduces performance, and strong base-task competence does not reliably predict adaptation. Trajectory analysis shows that successful adaptation depends on progressing from recognizing a novelty to diagnosing it and revising the strategy. Novelty-augmented post-training and explicit meta-action reasoning improve performance on novel tasks, while process rewards reshape the agent's behavior with only limited additional gains. Our findings motivate treating environmental variation as a core dimension of agent training and evaluation, as well as evaluating agent strategy and behavior alongside task success.

REFERENCES

Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016. doi: 10.1109/MS.2016.60.

Ali Basiri, Lorin Hochstein, Nora Jones, and Haley Tucker. Automating chaos experiments in production. *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 31–40, 2019. URL https://api.semanticscholar.org/CorpusID:152282467.

Terrance E. Boult, Przemyslaw A. Grabowicz, Derek S. Prijatelj, Roni Stern, Lawrence B. Holder, Joshua Alspector, Mohsen Jafarzadeh, Touqeer Ahmad, Akshay Raj Dhamija, Chunchun Li, Steve Cruz, Abhinav Shrivastava, Carl Vondrick, and Walter J. Scheirer. Towards a unifying framework for formal theories of novelty. In *AAAI Conference on Artificial Intelligence*, 2021. URL https://api.semanticscholar.org/CorpusID:235363448.

BusyBox Project. *BusyBox: The Swiss Army Knife of Embedded Linux*, n.d. URL https://busybox.net/downloads/BusyBox.html. Online command reference; the page is undated and unversioned.

Yuxin Chen, Xiao-Dong Cai, Junfeng Fang, Zhuowen Han, Yu Wang, Yaorui Shi, Yi Evie Zhang, Qi Gu, Xunliang Cai, Xiang Wang, An Zhang, and Tat-Seng Chua. Learning to act under noise: Enhancing agent robustness via noisy environments. *ArXiv*, abs/2605.27209, 2026. URL https://api.semanticscholar.org/CorpusID:288672327.

Terrance Devries and Graham W. Taylor. Improved regularization of convolutional neural networks with cutout. *ArXiv*, abs/1708.04552, 2017. URL https://api.semanticscholar.org/CorpusID:23714201.

Danny Dig and Ralph Johnson. How do apis evolve? a story of refactoring: Research articles. *J. Softw. Maint. Evol.*, 18(2):83–107, March 2006. ISSN 1532-060X.

Leon Arne Engländer, Sophia Althammer, A. Ustun, Matthias Gall'e, and Tom Sherborne. Agents explore but agents ignore: Llms lack environmental curiosity. *ArXiv*, abs/2604.17609, 2026. URL https://api.semanticscholar.org/CorpusID:287635649.

Thomas Foster, Bassel Al Omari, Tingchen Fu, Thomas Mann, Carl Domond, Lucia Cipolina-Kun, Bhavul Gauri, Muna Aghamelu, Alexander David Goldie, Eryk Helenowski, Jean-Christophe Gagnon-Audet, Alberto Pepe, Saba Nazir, Daniel Izcovich, Noam Levi, Rishi Hazra, Karen Hambardzumyan, Nicolas Mario Baldwin, Xian Li, Martin Josifoski, Paris Giampouras, Masoud Jalili Sabet, Anya Sims, Hela Momand, Tatiana Shavrina, Despoina Magka, Jason Weston, Yu-Lin Wang, Anirudh Goyal, João F. Henriques, Yoram Bachrach, Emily McMilin, and Jakob Nicolaus Foerster. Ai research preference models. 2026. URL https://api.semanticscholar.org/CorpusID:291167819.

Free Software Foundation. *GNU Autoconf 2.72: Creating Automatic Configuration Scripts*, December 2023. URL https://www.gnu.org/software/autoconf/manual/autoconf-2.72/. Manual dated 4 December 2023; see Chapters 6 and 11 on system and shell portability.

Free Software Foundation. *GNU Coreutils 9.11 Manual*, 2026. URL https://www.gnu.org/software/coreutils/manual/coreutils.html. See "Common options" and "Standards conformance" for environment- and standards-dependent behavior.

Kanishk Gandhi, Shivam Garg, Noah D. Goodman, and Dimitris Papailiopoulos. Endless terminals: Scaling rl environments for terminal agents. *ArXiv*, abs/2601.16443, 2026. URL https://api.semanticscholar.org/CorpusID:284982004.

Aniketh Garikaparthi. Can llms perceive time? an empirical investigation. *ArXiv*, abs/2604.00010, 2026. URL https://api.semanticscholar.org/CorpusID:286494320.

Shivam Goel, Panagiotis Lymperopoulos, Ravenna Thielstrom, Evan A. Krause, Patrick Feeney, Pierrick Lorang, Sarah Schneider, Yichen Wei, Eric Kildebeck, Stephen Goss, Michael C. Hughes, Liping Liu, Jivko Sinapov, and matthias. scheutz. A neurosymbolic cognitive architecture framework for handling novelties in open worlds. *Artificial Intelligence*, 331:104111, 2024. URL https://api.semanticscholar.org/CorpusID:268473117.

Yihao Hu, Zhihao Wen, Xiujin Liu, Pan Wang, Xin Zhang, and Wei Wu. Seal: Synergistic coevolution of agents and learning environments. *ArXiv*, abs/2605.24426, 2026. URL https://api.semanticscholar.org/CorpusID:288670387.

Naoki Inada. PEP 686: Make UTF-8 mode default. Python Enhancement Proposals, March 2022. URL https://peps.python.org/pep-0686/. Standards Track; status: Final; targets Python 3.15.

Hamish Ivison, Junjie Oscar Yin, Rulin Shao, Teng Xiao, Nathan Lambert, and Hanna Hajishirzi. Tmax: A simple recipe for terminal agents. *ArXiv*, abs/2606.23321, 2026. URL https://api.semanticscholar.org/CorpusID:289621599.

Rishi Jha, Hal Triedman, Arkaprabha Bhattacharya, and Vitaly Shmatikov. Agent meltdowns: The road to hell is paved with helpful agents. *ArXiv*, abs/2605.19149, 2026. URL https://api.semanticscholar.org/CorpusID:288651635.

Mayank Kejriwal, Eric Kildebeck, Robert J. Steininger, and Abhinav Shrivastava. Challenges, evaluation and opportunities for open-world learning. *Nature Machine Intelligence*, 6:580 – 588, 2024. URL https://api.semanticscholar.org/CorpusID:270717953.

Jiayi Kuang, Yinghui Li, Xin Zhang, Yangning Li, di yin, Xing Sun, Ying Shen, and Philip S. Yu. Process-level trajectory evaluation for environment configuration in software engineering agents. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=Q8qgloDKUO.

Raula Gaikovina Kula, Daniel M. Germán, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23:384 – 417, 2017. URL https://api.semanticscholar.org/CorpusID:21305181.

Ashish Kumar, Zipeng Fu, Deepak Pathak, and Jitendra Malik. Rma: Rapid motor adaptation for legged robots. *ArXiv*, abs/2107.04034, 2021. URL https://api.semanticscholar.org/CorpusID:235650916.

Pat Langley. Open-world learning for radically autonomous agents. In *AAAI Conference on Artificial Intelligence*, 2020. URL https://api.semanticscholar.org/CorpusID:208285052.

Hanyu Li, Yichi Zhang, Speed Zhu, Hang Su, Jun Zhu, and Yinpeng Dong. Repomirage: Probing repository context reasoning in code agents with perturbations. *ArXiv*, abs/2605.26177, 2026a. URL https://api.semanticscholar.org/CorpusID:288672764.

Yu Li, Haoyu Luo, Yuejin Xie, Yuqian Fu, Zhonghao Yang, Shuai Shao, Qihan Ren, Wanying Qu, Yanwei Fu, Yujiu Yang, Jing Shao, and Dongrui Liu. ATBench: A diverse and realistic agent trajectory benchmark for safety evaluation and diagnosis. In *Third Conference on Language Modeling*, 2026b. URL https://openreview.net/forum?id=QdmJ4NlpHG.

Jiayu Liu, Qihan Lin, Cheng Qian, Rui Wang, Emre Can Acikgoz, Xiaocheng Yang, Jiateng Liu, Zhenhailong Wang, Xiusi Chen, Heng Ji, and Dilek Hakkani-Tur. Planbench-xl: Evaluating long-horizon planning of llm tool-use agents in large-scale tool ecosystems. *ArXiv*, abs/2606.22388, 2026a. URL https://api.semanticscholar.org/CorpusID:289625815.

Jiayu Liu, Cheng Qian, Zhaochen Su, Qing Zong, Shijue Huang, Bingxiang He, and Yi R. Fung. CostBench: Evaluating multi-turn cost-optimal planning and adaptation in dynamic environments for LLM tool-use agents. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12826–12858, San Diego, California, United States, July 2026b. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.584. URL https://aclanthology.org/2026.acl-long.584/.

Zhihan Liu, L. Guan, Yixin Nie, Kai Zhang, Zhuoqun Hao, Lin Chen, Asli Celikyilmaz, Zhaoran Wang, and Natalia Zhang. Paying less generalization tax: A cross-domain generalization study of rl training for llm agents. *ArXiv*, abs/2601.18217, 2026c. URL https://api.semanticscholar.org/CorpusID:285051441.

Bryan Loyall, Avi Pfeffer, James Niehaus, Michael Harradon, Paola Rizzo, Alex Gee, Joe Campolongo, Tyler Mayer, and John Steigerwald. Coltrane: A domain-independent system for characterizing and planning in novel situations. *Artificial Intelligence*, 345:104336, 2025. URL https://api.semanticscholar.org/CorpusID:278107092.

Hasan Mahmud, Shreya Gupta, Isha Chaudhary, Nathaniel Enis, Ravi Mangal, Gagandeep Singh, Corina Pasareanu Colorado State University, Microsoft, U. Illinois Urbana-Champaign, and Carnegie Mellon University. A jagged frontier: Evaluating robustness of code agents to semantics-preserving transformations. 2026. URL https://api.semanticscholar.org/CorpusID:291217367.

Mike A Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Kumar Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Kwesi Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Jenia Jitsev, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

Shiwali Mohan, Wiktor Mateusz Piotrowski, Roni Stern, Sachin Grover, Sookyung Kim, Jacob Le, and Johan de Kleer. A domain-independent agent architecture for adaptive operation in evolving open worlds. In *Artificial Intelligence*, 2023. URL https://api.semanticscholar.org/CorpusID:259138770.

Faizan Muhammad, Vasanth Sarathy, Gyan Tatiya, Shivam Goel, Saurav Gyawali, Mateo Guaman, Jivko Sinapov, and matthias. scheutz. A novelty-centric agent architecture for changing worlds. In *Adaptive Agents and Multi-Agent Systems*, 2021. URL https://api.semanticscholar.org/CorpusID:232128080.

Zechun Niu, Yukun Zhao, Jia-Xin Zhang, Xuexin Shen, Jin-Hua Si, Hantao Tian, Can Xu, Yunze Song, Jiale Mao, Yanmei Gao, Yu-Chen Li, Jian-Min Wu, Lingyong Yan, Shuaiqiang Wang, and Datong Yin. Dumatebench: Evaluating autonomous agents in complex real-world workflows. 2026. URL https://api.semanticscholar.org/CorpusID:291383324.

Bespoke Labs OpenThoughts-Agent, Snorkel AI. OpenThoughts-TBLite: A High-Signal Benchmark for Iterating on Terminal Agents. https://www.openthoughts.ai/blog/openthoughts-tblite, February 2026.

Daniel S. Park, William Chan, Yu Zhang, Chung-Cheng Chiu, Barret Zoph, Ekin Dogus Cubuk, and Quoc V. Le. Specaugment: A simple data augmentation method for automatic speech recognition. *ArXiv*, abs/1904.08779, 2019. URL https://api.semanticscholar.org/CorpusID:121321299.

Marc H. Raibert, Kevin Blankespoor, Gabriel M. Nelson, and Robert Playter. Bigdog, the rough-terrain quadruped robot. *IFAC Proceedings Volumes*, 41:10822–10825, 2008. URL https://api.semanticscholar.org/CorpusID:8414062.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Jun-Mei Song, Mingchuan Zhang, Y. K. Li, Yu Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *ArXiv*, abs/2402.03300, 2024. URL https://api.semanticscholar.org/CorpusID:267412607.

Vaishnavi Shrivastava, Piero Kauffmann, Ahmed Awadallah, and Dimitris Papailiopoulos. Echo: Terminal agents learn world models for free. *ArXiv*, abs/2605.24517, 2026. URL https://api.semanticscholar.org/CorpusID:288670944.

Junhao Su, Yuanliang Wan, Junwei Yang, Hengyu Shi, Tianyang Han, Yurui Qiu, and Junfeng Luo. Failure makes the agent stronger: Enhancing accuracy through structured reflection for reliable tool interactions. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 12712–12734, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.618. URL https://aclanthology.org/2026.findings-acl.618/.

Jingwei Sun, Jianing Zhu, Yuanyi Li, Tongliang Liu, Xia HU, and Bo Han. Agenthijack: Benchmarking computer use agent robustness to common environment corruptions. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=0H5Im3Xvuf.

Geoffrey Thomas, Matthias Klose, Filipe Laíns, Donald Stufft, Tzu ping Chung, Stefano Rivera, Elana Hashman, and Pradyun Gedam. PEP 668: Marking python base environments as "externally managed". Python Enhancement Proposals, May 2021. URL https://peps.python.org/pep-0668/. Standards Track; status: Final.

Yang Tian, Zheng Shi, Yu Zhou, and Bo Zhao. Beyond function calling: Benchmarking tool-using agents under tool-environment unreliability. *ArXiv*, abs/2606.25819, 2026. URL https://api.semanticscholar.org/CorpusID:289622099.

Vernon Toh, Navonil Majumder, Zhengyuan Liu, Nancy F. Chen, and Soujanya Poria. Scramble-toolbench: Agents search exhaustively even when their own map points to the next step. 2026. URL https://api.semanticscholar.org/CorpusID:290830667.

Eduardo Treviño, Hugo Contant, James Ngai, Graham Neubig, and Zora Zhiruo Wang. Benchmarking failures in tool-augmented language models. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 2916–2934, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.149. URL https://aclanthology.org/2025.naacl-long.149/.

Sanidhya Vijayvargiya, Aditya Bharat Soni, Xuhui Zhou, Zora Zhiruo Wang, Nouha Dziri, Graham Neubig, and Maarten Sap. Openagentsafety: A comprehensive framework for evaluating real-world ai agent safety. *ArXiv*, abs/2507.06134, 2025. URL https://api.semanticscholar.org/CorpusID:280149606.

Sri Vatsa Vuddanti, Aarav Shah, Satwik Kumar Chittiprolu, Tony Song, Sunishchal Dev, Kevin Zhu, and Maheep Chaudhary. Paladin: Self-correcting language model agents to cure tool-failure cases. *ArXiv*, abs/2509.25238, 2025. URL https://api.semanticscholar.org/CorpusID:281682345.

Andrew Wang, Sophia Hager, Adi Asija, Daniel Khashabi, and Nicholas Andrews. Hell or high water: Evaluating agentic recovery from external failures. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=Zk224WPT42.

Weiming Wu, Song-Lin Lv, Rui Zhu, Zi-Jian Cheng, and Lan-Zhe Guo. Can agents generalize to the open world? unveiling the fragility of static training in tool use. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=G3hX5Fkrg7.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *ArXiv*, abs/2404.07972, 2024. URL https://api.semanticscholar.org/CorpusID:269042918.

Tianyin Xu and Yuanyuan Zhou. Systems approaches to tackling configuration errors: A survey. *ACM Comput. Surv.*, 47(4), July 2015. ISSN 0360-0300. doi: 10.1145/2791577. URL https://doi.org/10.1145/2791577.

An Yang, An-Feng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bo-Wen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxin Yang, Jingren Zhou, Jingren Zhou, Junyan Lin, Kai Dang, Keqin Bao, Ke-Pei Yang, Le Yu, Li-Chun Deng, Mei Li, Min Xue, Ming-Ze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tian-Yi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yi-Chao Zhang, Ying-Er Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhen-Ru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. 2025. URL https://api.semanticscholar.org/CorpusID:278602855.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Adriano Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *ArXiv*, abs/2405.15793, 2024. URL https://api.semanticscholar.org/CorpusID:270063685.

Ziang Ye, Wentao Shi, Yuxin Liu, Yu Wang, Zhengzhou Cai, Yaorui Shi, Qibin Gu, Xunliang Cai, and Fuli Feng. Look before you leap: Autonomous exploration for llm agents. *ArXiv*, abs/2605.16143, 2026. URL https://api.semanticscholar.org/CorpusID:288392402.

Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011. URL https://api.semanticscholar.org/CorpusID:974223.

Ding Yuan, Yu Luo, Zhuang Xin, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *USENIX Symposium on Operating Systems Design and Implementation*, 2014. URL https://api.semanticscholar.org/CorpusID:8930101.

Zhiwei Zhang, Fei Zhao, Rui Wang, Zezhong Wang, Bin Liang, Jiakang Wang, Yao Hu, Shaosheng Cao, and Kam-Fai Wong. Robust tool use via fission-GRPO: Learning to recover from execution errors. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 40477–40491, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.1880. URL https://aclanthology.org/2026.acl-long.1880/.

Xian-Hai Zhao, Han Li, Shuai Li, Tianyi Zhao, Earl T. Barr, Federica Sarro, and He Ye. Failure as a process: An anatomy of cli coding agent trajectories. *ArXiv*, abs/2607.09510, 2026. URL https://api.semanticscholar.org/CorpusID:290120985.

Shu Zhou, Rui Ling, Junan Chen, Xin Wang, Tao Fan, and Hao Wang. When more thinking hurts: Overthinking in LLM test-time compute scaling. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 23967–23977, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.1199. URL https://aclanthology.org/2026.findings-acl.1199/.

Dongsheng Zhu, Xucheng Ma, Yucheng Shen, Xiang Li, Yukun Zhao, Shuaiqiang Wang, Lingyong Yan, and Dawei Yin. When tools fail: Benchmarking dynamic replanning and anomaly recovery in llm agents. *ArXiv*, abs/2606.05806, 2026. URL https://api.semanticscholar.org/CorpusID:288977070.

APPENDIX CONTENTS

A AI Use, Ethics, and Reproducibility Statements **17**
A.1 AI Use Statement . . . 17
A.2 Ethics Statement . . . 17
A.3 Reproducibility Statement . . . 17

B Detailed Related Work **18**

C Details of the AGNI Pipeline **19**
C.1 LLM Roles in AGNI . . . 19
C.2 Analyzing Base-Task Strategies . . . 19
C.3 Synthesizing Novelty Descriptions . . . 20
C.4 Implementing and Validating Novel Tasks . . . 20
C.5 Validating Adaptive Behavior . . . 21
C.6 Evaluation-Set Sampling . . . 21
C.7 Examples from AGNI-Generated Base–Novel Task Pairs . . . 21

D Realism Evaluation Details **28**

E Detailed Evaluation Results **29**
E.1 Evaluation Models and Inference Configuration . . . 29
E.2 Cross-Model Relevance of AGNI Novelties . . . 29
E.3 Retry Prevalence and Persistence . . . 30
E.4 Adaptation Stage Definitions . . . 30
E.5 Behavioral Transitions and Conditional Success . . . 30

F Analyzing Trajectories for Adaptive Behavior **31**

G Improving Adaptation with Agentic Post-Training **32**
G.1 Starting Checkpoints . . . 32
G.2 Controlled Training Comparisons . . . 32
G.3 META Interface and Process-Reward Comparisons . . . 32
G.4 Normal and META Prompts . . . 34
G.5 Qwen3-14B Generic Post-Training Results . . . 36

H Model Prompts Used in AGNI **37**

A AI USE, ETHICS, AND REPRODUCIBILITY STATEMENTS

A.1 AI USE STATEMENT

Generative AI tools were used both as components of our experimental methodology and as auxiliary research tools. GPT-5.5 was used within the AGNI pipeline for strategy analysis, novelty generation and implementation, behavioral validation, trajectory analysis, and realism evaluation, as described in Sections 4 and the appendices. Claude was used as an auxiliary tool for locating and summarizing potentially relevant literature, supporting code implementation and debugging, and polishing the grammar, clarity, and concision of author-written text. Claude was not used as a source of research ideas, methodological decisions, interpretations, or claims. All experimental designs, literature judgments, analyses, claims, and conclusions were reviewed and determined by the authors, who take full responsibility for the final manuscript.

A.2 ETHICS STATEMENT

This work evaluates agents on software benchmarks. LLM-generated outputs may contain harmful content, and improved adaptation may also enable misuse. Our evaluation measures task adaptation and does not establish the safety of agent behavior or generated content. Generated environments and agent commands should be executed with appropriate isolation and access controls.

A.3 REPRODUCIBILITY STATEMENT

We provide implementation and experimental details for environment generation, agent evaluation, trajectory analysis, and post-training. Algorithm 1 and Appendix C describe the AGNI pipeline, including model roles, generation budgets, environment validation, and evaluation-set sampling. Appendix H provides the generation and analysis prompts and output schemas. Appendices E.1, F, and D specify the evaluation models and inference settings, evidence-based annotation criteria, and realism-scoring protocol, respectively.

For post-training, Section 6 and Appendix G document the training-data constructions, starting checkpoints, optimization hyperparameters, interaction budgets, checkpoint-selection criterion, full agent prompts, and process-reward definitions. We report model identifiers, dated snapshots where available, and sampling settings to support replication of the experimental setup. Exact numerical reproduction may nevertheless vary because generation is stochastic and hosted model endpoints can change.

B Detailed Related Work

Open-World Learning and Adaptation. Open-world learning studies how agents adapt when environmental novelty invalidates their assumptions (Langley, 2020; Boult et al., 2021; Kejriwal et al., 2024). Prior work decomposes adaptation into detecting novelty, diagnosing its cause, and updating knowledge or behavior accordingly (Muhammad et al., 2021; Mohan et al., 2023; Goel et al., 2024; Loyall et al., 2025). Crucially, novelty is relative to the agent and its task: an environmental change may invalidate one strategy yet be irrelevant to another (Boult et al., 2021; Goel et al., 2024). Much prior work studies embodied, simulated, or game-based environments, often using structured world models and dedicated components for novelty detection, diagnosis, and adaptations (Raibert et al., 2008; Kumar et al., 2021). In contrast, we study adaptation in general-purpose LLM agents operating through open-ended programmatic interfaces, where strategies and assumptions are largely implicit in interaction trajectories.

Robustness Evaluation in LLM Agents. Recent work studies the robustness of LLM agents by perturbing tool availability (Treviño et al., 2025; Wang et al., 2025; Liu et al., 2026a;b), tool interfaces (Zhu et al., 2026; Tian et al., 2026; Wu et al., 2026), workspace contents (Li et al., 2026a; Mahmud et al., 2026; Engländer et al., 2026), and runtime conditions (Jha et al., 2026; Niu et al., 2026). AGNI shares several design principles with some of these works, including paired base/-modified tasks, undisclosed changes, preserved solvability, and perturbations to agent dependencies. However, prior works largely predefine perturbation families, which are less suited for open-ended terminal tasks using heterogeneous resources and conditions (Ivison et al., 2026; Merrill et al., 2026). In terminal tasks, the relevant dependencies may depend on the agent's specific strategy rather than on the task alone. Thus, AGNI starts from successful trajectories, identifies environmental assumptions that the agent relies on but are not guaranteed by the task instruction, and invalidates them while preserving solvability. Moreover, since generated novelties are tied to a task-specific violated assumption, this enables a more controlled analysis of novelty adaptation than prior trajectory-level analyses of agent failure and recovery (Toh et al., 2026; Kuang et al., 2026; Zhao et al., 2026).

Post-Training for Adaptive LLM Agents. Recent work post-trains agents for robustness to perturbations in tool-use (Vuddanti et al., 2025; Zhang et al., 2026; Wu et al., 2026; Su et al., 2026; Chen et al., 2026; Hu et al., 2026), text-game (Ye et al., 2026), and GUI environments (Sun et al., 2026). Our work differs along two axes. First, what agents train on: prior work samples perturbations from predefined families, which are not suited for the heterogeneous action space of a terminal. AGNI instead constructs executable novel environments from assumptions revealed by the agent's successful trajectories. Second, how agents are supervised: prior methods use task outcomes (Chen et al., 2026; Sun et al., 2026), teacher-written recovery demonstrations (Vuddanti et al., 2025; Wu et al., 2026), externally supplied diagnoses (Zhang et al., 2026; Hu et al., 2026), or a single rewarded behavior such as reflection or exploration (Su et al., 2026; Ye et al., 2026). We instead reward the adaptation process: agents report actions with meta-actions such as inspecting, diagnosing, revising, and verifying, enabling process-level rewards for adaptive behaviors expressed through meta-actions.

Software Evolution and Resilience Testing. Prior work shows that software environments routinely change in ways that invalidate previously valid workflows. Common examples include interface and dependency evolution, configuration errors, and changes in system state. Interface and dependency evolution can break existing workflows: APIs evolve and break downstream clients (Dig & Johnson, 2006), applications retain older dependency versions (Kula et al., 2017), and system policies and runtime defaults change over time (Thomas et al., 2021; Free Software Foundation, 2026; BusyBox Project, n.d.; Free Software Foundation, 2023; Inada, 2022). Configuration errors are prevalent across commercial and open-source systems and can cause difficult-to-diagnose failures (Yin et al., 2011; Xu & Zhou, 2015). Changes in system state can expose failures that depend on particular combinations and orderings of otherwise ordinary events (Yuan et al., 2014). These kinds of changes motivate resilience testing and chaos engineering, which evaluate whether software systems continue to function reliably under perturbed conditions (Basiri et al., 2016; 2019). These works motivate both our study and our design of environmental novelty in terminal environments. Analogous to resilience testing for software systems, we evaluate whether terminal agents remain reliable when the surrounding software environment changes.

C Details of the AGNI Pipeline

C.1 LLM Roles in AGNI

Table 4 lists the roles in Algorithm 1; Table 5 gives their settings. Appendix H provides their prompts and output schemas.

Role	Symbol	Purpose
Solver	M_{S}	Generates interaction trajectories in the base and novel environments.
Strategy analyzer	M_{A}	Summarizes observed task-solving behavior and extracts trajectory-grounded environmental assumptions.
Novelty generator	$M_{\mathrm{N}}^{\mathrm{dim}}$, $M_{\mathrm{N}}^{\mathrm{spec}}$	Proposes environmental changes that target extracted assumptions while preserving the task objective.
Implementer	$M_{\mathrm{N}}^{\mathrm{impl}}$	Instantiates a novelty specification as a modified environment E' and produces an adapted reference solution S'.
Behavioral observer	$M_{\mathrm{A}}^{\mathrm{obs}}$	Checks whether the injected novelty produces the expected change in behavior between the base and novel environments.
Trajectory analyzer	Analyze Trajectories	Analyzes novel-task trajectories to characterize adaptation and identify unintended shortcuts or reward hacking.

Table 4: LLM roles in AGNI.

Setting	Value
Model	GPT-5.5 (2026-04-24), all roles.
Reasoning effort	High for the solver; medium for other roles.
Sampling	Temperature and top-p unset (API defaults); no fixed seed.
Output-token limits	8,192 per solver turn; 32,000 per non-solver call.
Solver harness	Terminus-2 (Harbor 0.6.6); at most 32 turns.
Construction rollouts	$K = 8$ per base and novel task, across all three sources.
Candidate budgets	$N_{\mathrm{spec}} = 5$ per applicable dimension; up to $N_{\mathrm{keep}} = 8$ variants per base task.
Implementation repair	At most one round.

Table 5: AGNI model settings and construction budgets.

C.2 Analyzing Base-Task Strategies

Base trajectories. For each base task $\mathcal{T}_{\mathrm{base}} = (I, E, G)$, we sample K independent trajectories from M_{S} and retain the task only if at least one passes the deterministic grader G. This ensures nonzero observed base performance: if the solver already scores zero, added difficulty from novelty may leave the score unchanged, obscuring its effect. The reference solution S provides a valid base strategy.

Strategy analysis. For each retained task, M_{A} receives (I, E, G), S, and all K trajectories with their pass/fail labels. It returns the task objective, summaries of reference and solver behavior, and a comparison of successful and failed trajectories. The summary records commands executed, tools and resources used, files and paths accessed, and where successful and failed trajectories diverge.

Environmental assumptions. The analyzer extracts assumptions underlying the observed strategy. Both assumptions and the novelties targeting them span four dimensions:

- **Perception:** expected observations, including resource presence, absence, identity, and location.
- **Control:** how the solver can act, including available tools, commands, and interfaces.
- **Constraints:** permitted or feasible actions, including permissions, resource limits, and writable locations.

- **Dynamics:** action and environment behavior, including runtime behavior, dependency semantics, and the effects of commands or system operations.

Assumption schema. For each assumption, M_{A} returns: (i) its statement; (ii) the sources relying on it (the reference solution, successful or failed solver trajectories, or both); (iii) supporting evidence; (iv) why the assumption may be brittle; and (v) what the solver would need to notice or do if it were invalidated. These outputs ground candidate novelties in observed base-task behavior. A subsequent dimension-identification call assigns the applicable categories.

The exact strategy-analysis prompt and output schema are provided in Appendix H.

C.3 Synthesizing Novelty Descriptions

Applicable novelty dimensions. Given the base task and strategy analysis from Appendix C.2, $M_{\mathrm{N}}^{\mathrm{dim}}$ selects dimensions in which minimal, task-preserving changes can invalidate an extracted assumption and require different solver behavior.

Candidate generation. For each selected dimension d, $M_{\mathrm{N}}^{\mathrm{spec}}$ receives (I, E, G), the behavior summary, the associated assumptions, and $\mathcal{H}_d$: short descriptions of previously rejected candidates and their failure reasons. This history is maintained separately for each base task and updated after each rejection. The model generates $N_{\mathrm{spec}} = 5$ candidate descriptions per dimension, each specifying a novelty subtype, proposed environmental edits, expected trajectory change, why adaptation is needed, why the task objective is preserved, risks, and the two scores below.

Filtering and selection. We discard malformed descriptions and filter candidates using two scores assigned by $M_{\mathrm{N}}^{\mathrm{spec}}$, each on a 1–5 scale: *implementation plausibility* (1: impossible or too invasive; 3: requires moderate care; 5: trivial to implement) and *expected behavioral change* (1: identical solutions; 3: some visible change; 5: the original strategy fails). An LLM compares short candidate summaries within each base task and removes proposals for the same or very similar environmental changes. We interleave the remaining descriptions across dimensions to give each dimension equal opportunity in candidate selection.

Appendix H provides the dimension-selection and description-generation prompts and output schemas.

C.4 Implementing and Validating Novel Tasks

Implementation. For each candidate c, $M_{\mathrm{N}}^{\mathrm{impl}}$ receives (I, E, G), the reference solution S, and the novelty description. It produces a modified environment E'_c and an adapted reference solution S'_c, which provides one valid strategy for the modified task.

Task preservation. Edits are limited to those needed for the novelty, such as changes to environment state, permissions, tools, dependencies, or runtime behavior. The task objective and grader assertions must remain unchanged. G still denotes the same success criterion. The agent must encounter the novelty through interaction, without the instruction revealing it.

Implementation repair. Each candidate receives one implementation attempt. If building the environment or executing S'_c fails, we return execution and grader feedback to $M_{\mathrm{N}}^{\mathrm{impl}}$ and allow one repair; candidates that still fail are rejected. This repair step is omitted from Algorithm 1.

Validation. Candidates must satisfy the task-preservation constraints and pass three checks before behavioral validation (Appendix C.5):

1. **Task solvability:** the adapted reference succeeds, $G(S'_c, E'_c) = 1$.
2. **Base-strategy invalidation:** the original reference fails, $G(S, E'_c) = 0$.
3. **Solver recoverability:** among K fresh trajectories $\mathcal{R}_c$ sampled from the construction solver M_{S} in E'_c, at least one passes G.

C.5 Validating Adaptive Behavior

After the executable checks in Appendix C.4, we verify that successful solver behavior responds to the injected novelty.

Behavioral-effect validation. The observer $M_{\mathrm{A}}^{\mathrm{obs}}$ receives the novelty description and successful trajectories from the base and modified environments. It checks for visible novelty and expected execution changes consistent with the targeted assumption being invalidated. Candidates without evidence of the intended behavioral effect are rejected.

Trajectory analysis. The trajectory analyzer examines all fresh solver trajectories $\mathcal{R}_c$ in E'_c step by step for adaptive recovery and reward hacking. A valid recovery addresses the novelty and completes the original task; it may differ from S'_c. Reward hacking includes attempts to satisfy checks by violating explicit task requirements, manipulating grader-relevant artifacts, or replacing required operations without completing the task.

Acceptance. A candidate is retained only when the injected novelty produces the intended behavioral effect, every grader-successful trajectory contains evidence of a task-preserving adaptation, and no analyzed trajectory exhibits reward hacking. An observed reward-hacking attempt causes rejection even if that trajectory fails the grader.

Appendix F describes the trajectory annotations; Appendix H provides the prompts and output schemas.

C.6 Evaluation-Set Sampling

We sample the final 140 ET-eval, 118 TB-Lite, and 87 TB-2 base–novel pairs using the same joint-stratification procedure for each source. The strata capture task difficulty, novelty-induced difficulty, and realism.

Joint stratification. Each pair receives five tags (Table 6). We assess difficulty with a relatively strong sampler, GPT-5.4-mini, and a weaker one, Ministral-14B. GPT-5.4-mini alone scores many ET-eval base tasks at 1.0, leaving too little spread for difficulty stratification. The two samplers estimate base-task difficulty, P_{base}, and relative novelty-induced difficulty, $\delta = (P_{\mathrm{base}} - P_{\mathrm{novel}})/P_{\mathrm{base}}$. The realism judge supplies occurrence plausibility, realization fidelity, and mechanism family (Appendix D). We sample from the joint cells formed by these tags, rather than balancing each tag separately. The scores define strata; they are not exclusion thresholds.

Table 6: Strata used for joint sampling across ET-eval, TB-Lite, and TB-2.

Axis	Strata
Base-task difficulty	Easy, medium, hard
Novelty-induced difficulty	Low, medium, high
Occurrence plausibility	Low, medium, high
Realization fidelity	Low, medium, high
Mechanism family	Categorical labels from the frozen taxonomy

C.7 Examples from AGNI-Generated Base–Novel Task Pairs

This section presents examples of novelties introduced by AGNI that make agents' initial strategies insufficient while preserving the task objective and grader. We identify the task source, describe the task and novelty, and summarize the agents' responses in a short narrative and table. Comparisons between adaptive and non-adaptive runs, which may come from the same model or different models, illustrate adaptive behaviors.

	Adaptive behavior	Non-adaptive behavior
Agent	GPT-5.4-mini, `run_02`.	GPT-5.4-mini, `run_07`.
Explanation	Reads the launch log and identifies `nohup` as the obstacle.	Leaves the cause unresolved and continues to rely on `nohup`.
Action	Relaunches without `nohup`, using `python3 -m mlflow server ... &`.	Retries launch commands that still use `nohup`.
Check	Checks the server's health endpoint and confirms that it responds.	Launch attempts continue to fail; no working server is established.
Result	Trains and registers the model; all three tests pass.	Does not complete the task; all three tests fail.

Table 7: Adaptive and non-adaptive responses to a blocked launch command.

Example 1: A blocked command for starting a server.

Task Source: TB-Lite [`mlflow-register`].

Task. Start an MLflow tracking server in the background on port 8080, train a linear regression model on random data with three input features, and register the model as `gpt-5`.

Novelty. Base runs start the server using `nohup`. The modified environment adds a script at `/usr/local/bin/nohup` that is found before the original command and rejects attempts to launch the server. The original utility remains at `/usr/bin/nohup`, and starting the server directly in the background still works. The task instructions and grader remain unchanged.

Adaptive vs Non-Adaptive Run. Both GPT-5.4-mini runs encounter `Operation not permitted` when starting the server (Table 7). The adaptive run reads the launch log, identifies `nohup` as the obstacle, and starts the server without it. It checks that the server responds, then trains and registers the model. The non-adaptive run repeatedly uses `nohup` despite the error and never establishes a working server. This comparison suggests that adaptive behaviors such as inspecting errors, changing the failed approach, and checking recovery can lead to success, whereas non-adaptive behaviors such as repeating the failed approach can lead to failure.

	Adaptive behavior	Non-adaptive behavior
Agent	GPT-5.4-mini, `run_00`.	GPT-5.5, `run_02`.
Explanation	Identifies the changed encoding as the reason direct searches miss the text.	Concludes that the logs contain none of the target words.
Action	Uses Python to decode the logs as UTF-16 before counting matching lines.	Repeats `grep` searches without accounting for the encoding.
Check	Confirms the UTF-16 encoding marker in the file bytes.	Accepts repeated zeros as evidence that the target words are absent.
Result	Reports counts of 4, 3, and 8; all three tests pass.	Reports zero counts; the grader rejects the summary.

Table 8: Adaptive and non-adaptive responses to changed log encoding.

Example 2: A changed text encoding in log files.

Task Source: TB-Lite [`log-summary`].

Task. Read the log files in `/app/logs`, count lines containing `ERROR`, `WARNING`, and `INFO`, and write the counts in a CSV summary.

Novelty. The modified environment stores the logs in UTF-16LE instead of UTF-8, changing how the same text is stored as bytes. Extra zero bytes now separate the letters in the target words, so

the original searches return zero counts. The filenames, log records, task instructions, and grader remain unchanged.

Adaptive vs Non-Adaptive Run. Both runs initially report zero counts (Table 8). The adaptive GPT-5.4-mini run questions the result, inspects the file bytes with Python, and identifies the UTF-16 encoding. It decodes the logs before counting and produces the correct summary. The non-adaptive GPT-5.5 run also notices suspicious zeros but repeats `grep` searches and concludes that the target words are absent. This comparison suggests that adaptive behaviors such as inspecting how data is stored and changing how it is read can lead to success, whereas non-adaptive behaviors such as repeating an unsuitable search and accepting unexplained zeros can lead to failure.

	Adaptive behavior
Agent	GPT-5.4-mini, `run_00`.
Explanation	Suspects an incorrect program path in the first line or Windows line endings; does not directly inspect the extra character.
Action	Reads the script and runs `bash /app/generate_zk_protocols.sh`, avoiding the broken first line.
Check	Confirms that the script exists and has execution permission before changing how it is run.
Result	Runs the generator and writes both reports; all 15 tests pass.

Table 9: An adaptive response to a broken script launch.

Example 3: A script that fails to start despite being present.

Task Source: TB-Lite [`cryptographic-protocol-verifier`].

Task. Run the provided script to generate cryptographic protocol transcripts, analyze them, and write `verification_results.json` and `cryptographic_analysis.txt`.

Novelty. The script originally runs directly at `/app/generate_zk_protocols.sh`. The modified environment adds a carriage-return character to its first line, which specifies the program used to run it. The extra character becomes part of that program's path, so direct execution fails even though the script exists and has execution permission. The script body, task instructions, and grader remain unchanged.

Adaptive Run. This example follows one successful GPT-5.4-mini run (Table 9). The agent confirms that the script exists and is executable but receives `cannot execute: required file not found` when it tries to run it. It suspects a problem with the first line or Windows line endings, reads the script, and runs it explicitly through `bash`. The generator works, and the agent completes both reports. This illustrates how adaptive behaviors such as investigating an unexpected error and changing how a tool is called can lead to success without repairing the tool itself.

	Adaptive behavior	Non-adaptive behavior
Agent	GPT-OSS-120B, `run_00`.	GPT-OSS-120B, `run_01`.
Explanation	Finds that a configuration setting prevents pip from searching for packages.	Attributes the remaining failure to Python-version compatibility without confirming that explanation.
Action	Inspects pip configuration and removes `no-index` from `/etc/pip.conf`.	Restores pip but leaves its global configuration unchecked.
Check	Successfully installs `requests` after removing the setting.	Confirms pip's version but accepts `No matching distribution found` when testing installation.
Result	Restores package installation; both tests pass.	Leaves installation broken and declares completion. Verifier setup also fails, so the task tests do not run.

Table 10: Adaptive and non-adaptive responses to a package-installation restriction.

Example 4: A setting that prevents pip from finding packages.

Task Source: TB-Lite [`broken-python`].

Task. Repair the system-wide Python package installer, `pip`, so that it can install packages.

Novelty. The original task requires restoring missing packaging tools. The modified environment also adds `no-index = true` to `/etc/pip.conf`, preventing pip from searching the package index for available packages. Restoring pip alone therefore leaves package installation broken. The task instructions and grader remain unchanged.

Adaptive vs Non-Adaptive Run. Both GPT-OSS-120B runs restore pip but still encounter installation failures (Table 10). The adaptive run inspects pip's configuration, finds the restriction, removes it, and successfully installs `requests`. The non-adaptive run confirms pip's version but cannot install `requests`. It blames Python-version compatibility without checking the configuration and declares the repair complete. This comparison suggests that adaptive behaviors such as investigating a remaining error and testing the required operation can lead to success, whereas non-adaptive behaviors such as accepting an unchecked explanation and stopping before the operation works can lead to failure.

	Adaptive behavior	Non-adaptive behavior
Agent	GPT-5.4-mini (high), `run_01`.	GPT-5.4-mini (high), `run_00`.
Explanation	Suspects that an inherited `TAR_OPTIONS` setting changes the stored names.	Recognizes the wrong names but does not identify the setting that changes them.
Action	Unsets `TAR_OPTIONS` when recreating and extracting the archive.	Repeats the archive command without changing the setting.
Check	Lists the rebuilt archive and verifies the names `app.log` and `error.log`.	Sees the `raw/` prefix again but stops after extraction fails.
Result	Extracts the log and completes the outputs; all four tests pass.	Leaves the archive incorrect and the extracted log missing; three tests fail.

Table 11: Adaptive and non-adaptive responses to changed archive filenames.

Example 5: A setting that changes filenames inside an archive.

Task Source: ET-eval [`task_54fe38b3`].

Task. Create `INC123_logs.tar.gz` containing exactly `app.log` and `error.log`, with no directory names before them. Also list the archive contents, extract `error.log`, and write its SHA-256 checksum.

Novelty. The original `tar` command stores the filenames as given. The modified environment sets `TAR_OPTIONS` to add `raw/` before each name. The archive therefore contains `raw/app.log` and `raw/error.log`, so extracting `error.log` by its expected name fails. The task instructions and grader remain unchanged.

Adaptive vs Non-Adaptive Run. Both GPT-5.4-mini runs encounter `error.log: Not found in archive` and see the unexpected `raw/` prefix when listing the contents (Table 11). The adaptive run suspects an inherited setting, recreates the archive with `TAR_OPTIONS` unset, checks the corrected names, and extracts the log. The non-adaptive run recognizes the wrong names but repeats the same command and stops while extraction still fails. This comparison suggests that adaptive behaviors such as investigating tool settings and verifying a correction can lead to success, whereas non-adaptive behaviors such as repeating an unchanged command and stopping with an unresolved error can lead to failure.

	Adaptive behavior	**Non-adaptive behavior**
Agent	GPT-OSS-120B, `run_00`.	GPT-OSS-120B, `run_02`.
Explanation	Identifies that the script treated a repeated header as a release.	Blames a blank line at the end of the CSV for creating an extra file.
Action	Reads the CSV, removes the generated files, and rebuilds the outputs while skipping header lines.	Does not read the CSV. Attempts to delete `release-.json`, although the extra file is named `release-version.json`.
Check	Lists the regenerated files and confirms that exactly three remain.	Sees that the count is still four but declares the task complete.
Result	Completes the task; all seven tests pass.	Leaves the extra file and log entry; two tests fail and five pass.

Table 12: Adaptive and non-adaptive responses to a repeated CSV header.

Example 6: A repeated header in the input CSV.

Task Source: ET-eval [`task_x1uj21`].

Task. Read `releases.csv` and create exactly three JSON configuration files (manifests), one for each software release, plus a three-line deployment log in input order. Each file must be named `release-<version>.json`, using the version from the input, and contain an integer port number.

Novelty. The original CSV has one header listing the column names, followed by three releases. The modified CSV repeats the header, `version,app_name,port`, before the third release. A script that skips only the first line now treats the repeated header as a fourth release. The task instructions and grader remain unchanged.

Adaptive vs Non-Adaptive Run. Both GPT-OSS-120B runs finish their initial commands without an error but report four files instead of three, including an extra `release-version.json` (Table 12). The adaptive run reads the CSV, identifies the repeated header, and regenerates the outputs while skipping header lines. It checks that exactly three files remain and passes all tests. The non-adaptive run blames a blank line, attempts to remove the wrong file, and stops with the count still at four. This comparison suggests that adaptive behaviors such as investigating the cause and verifying a correction can lead to success, whereas non-adaptive behaviors such as acting on an unchecked explanation and stopping with an unresolved error can lead to failure.

	Adaptive behavior	Non-adaptive behavior
Agent	GPT-5.5, `run_01`.	GPT-5.5, `run_03`.
Explanation	Reads the startup error and identifies an environment setting that requests a missing database script.	Treats a successful build and the expected JAR path as sufficient to declare completion.
Action	Sets the database connection explicitly in the application and rebuilds the JAR.	Configures the database through the properties file, which the environment setting overrides.
Check	Launches the rebuilt JAR, confirms startup, and tests login.	Checks that the JAR exists, but never launches it or tests an endpoint.
Result	Runs the service successfully; all ten tests pass.	The grader cannot start the service; all ten tests report setup errors.

Table 13: Adaptive and non-adaptive responses to an overridden database setting.

Example 7: An environment setting overrides the application's database configuration.

Task Source: TB-Lite [`token-auth-websocket`].

Task. Build a Spring Boot authentication service supporting login, token renewal, one-time tokens, logout, and cleanup. Package it as an executable Java archive (JAR) at `/app/build/libs/app.jar`. The grader launches this file and tests the service on port 8080.

Novelty. The base runs configure the database through `application.properties`. The modified environment sets `SPRING_DATASOURCE_URL` to request initialization from a missing file, `/tmp/missing-init.sql`. This setting overrides the packaged properties: the application builds successfully but fails when started. The task instructions and grader remain unchanged.

Adaptive vs Non-Adaptive Run. Both GPT-5.5 runs build the JAR, but only the adaptive run launches it (Table 13). It reads the missing-file error, inspects the environment, and identifies the database override. It sets the database connection explicitly in the application, rebuilds, and checks startup and login. The non-adaptive run confirms that the JAR exists and stops without launching it; the startup error appears only during grading. This comparison suggests that adaptive behaviors such as testing the application, diagnosing configuration conflicts, and verifying recovery can lead to success, whereas non-adaptive behaviors such as stopping after a successful build without testing runtime behavior can lead to failure.

	Adaptive behavior	Non-adaptive behavior
Agent	Kimi-2.6, `run_03`.	Kimi-2.6, `run_04`.
Explanation	Recognizes that Git's saved decision has left the requested biography update missing.	Recognizes that Git kept the old biography, but treats the conflict as resolved.
Action	Retrieves the updated biography from the project history and saves it.	Saves the result with the old biography still in place.
Check	Reads the biography and checks whether it includes the requested Stanford update.	Reads the old biography but accepts Git's resolved status as sufficient.
Result	Restores the updated biography and page layout; both tests pass.	Restores the layout but leaves the biography unchanged; one test fails and one passes.

Table 14: Adaptive and non-adaptive responses to Git reusing an old decision.

Example 8: A saved Git decision keeps the old biography.

Task Source: TB-2 [`fix-git`].

Task. Recover missing website edits: an updated biography mentioning Stanford and a changed page layout. The edits remain in the project's Git history and must be restored to its current version (`master`).

Novelty. Normally, combining these versions makes Git flag conflicting edits and wait for the agent to choose the intended text. The modified environment enables Git's `rerere` feature to reuse an earlier decision: keep the old biography. It also marks the file as ready to save. Git still reports a merge error, but allows the agent to finalize the result without restoring the updated biography. The instructions and grader remain unchanged. The modified task also reconstructs the project's history, preserving the required final contents and conflicting edits but simplifying earlier versions and adding a named shortcut to the lost version.

Adaptive vs Non-Adaptive Run. Both Kimi-2.6 runs find the missing edits, ask Git to combine them with the current website, and see a notice that Git reused a previous decision (Table 14). Both read the biography and recognize that Git kept the old text. The adaptive run notices that the Stanford update is missing, retrieves the updated biography, and saves the corrected result. The non-adaptive run treats Git's resolved status as sufficient, saves the old biography, and declares the changes restored. This comparison suggests that adaptive behaviors such as checking a tool's automatic decision against the user's goal and correcting the result can lead to success, whereas non-adaptive behaviors such as accepting the tool's status without ensuring the requested changes are present can lead to failure.

D REALISM EVALUATION DETAILS

We score two aspects of each validated novelty: *occurrence plausibility*, whether the environmental condition could arise in practice, and *realization fidelity*, how faithfully its implementation reproduces that condition (Table 15). These scores do not affect acceptance in Algorithm 1; they are used afterward for characterization and stratification into low (1–2), medium (3), and high (4–5) bins (Appendix C.6).

Score	Occurrence plausibility	Realization fidelity
1	Implausible or specific to the evaluator	Unfaithful or incoherent implementation
2	Technically possible but contrived	Crude proxy with major artificial artifacts
3	Plausible in some deployments	Controlled abstraction preserving the central effect
4	Recognizable real deployment or configuration change	Faithful simulation with minor simplifications
5	Documented, observed, or widely recognized failure mode supported by the artifacts	Direct or near-exact reproduction of the real mechanism

Table 15: The two realism scores are assigned separately using this rubric.

Judging protocol. We use GPT-5.5 (2026-04-24) with medium reasoning effort, a 32,000-token initial completion budget, and temperature unset. We retain one valid verdict per novelty, with no score aggregation. Truncated or unparseable responses are retried with a doubled token budget; missing required fields trigger one corrective retry.

Inputs and evidence. The judge receives the agent-facing instruction, novelty specification, environment diff, and changed files; reference solutions, tests, and grader content are excluded. It identifies the underlying condition from the artifacts and provides concrete evidence and rationale before each score. The prompt and output schema are provided in Appendix H.

Results. Table 16 reports the final evaluation sets. Before stratification, mean plausibility/fidelity scores were 3.75/4.41 for ET (283 candidate pairs), 3.81/4.51 for TB-Lite (370), and 3.74/4.35 for TB-2 (239).

		Occurrence plausibility			Realization fidelity		
Dataset	**Pairs**	**Counts (1–5)**	**Mean**	≥ 4	**Counts (1–5)**	**Mean**	≥ 4
ET-eval	140	0, 2, 43, 86, 9	3.73	68%	0, 0, 20, 35, 85	4.46	86%
TB-Lite	118	0, 6, 25, 76, 11	3.78	74%	0, 1, 11, 41, 65	4.44	90%
TB-2	87	0, 2, 18, 60, 7	3.83	77%	0, 0, 12, 29, 46	4.39	86%
All	345	0, 10, 86, 222, 27	3.77	72%	0, 1, 43, 105, 196	4.44	87%

Table 16: LLM realism ratings for the final evaluation sets. Counts list scores 1–5; ≥ 4 is the percentage rated 4 or 5, rounded to the nearest integer. Median plausibility and fidelity are 4 and 5, respectively, in every set.

E Detailed Evaluation Results

The base, novel, and disclosed results in Section 5 use the same weighting over base–novel pairs.

E.1 Evaluation Models and Inference Configuration

Table 17 lists the evaluated models and reasoning settings; Table 18 gives the shared inference configuration.

Model	**Provider**	**Model / snapshot**	**Main effort**	**Other efforts**
GPT-5.5	OpenAI	gpt-5.5 (2026-04-24)	High	–
GPT-5.4-mini	OpenAI	gpt-5.4-mini (2026-03-17)	High	Medium, Low
GPT-OSS-120B	OpenAI	gpt-oss-120B	Default	High, Low
Kimi-2.6	Moonshot	Kimi-K2.6 (2026-04-20)	Default	High, Low, None
DeepSeek-V4-Flash	DeepSeek	DeepSeek-V4-Flash-0731 (2026-07-31)	None	–
Grok-4.20	xAI	grok-4.20-beta-0309	Non-reasoning	–
Mistral-Large-3	Mistral	Mistral-Large-3	N/A	–

Table 17: Evaluation models. Default means `reasoning_effort` was omitted (approximately medium for GPT-OSS-120B and Kimi-2.6). Other efforts are used in Table 2.

Setting	**Value**
Agent harness	Terminus-2, at most 32 interaction turns.
Rollouts	$K = 5$ per model–task pair, unless otherwise specified.
Completion budget	8,192 tokens per model response.
Sampling	Temperature and top-p unset (deployment defaults); no explicit seed.
Verifier timeout	$10\times$ the configured verifier timeout.

Table 18: Shared settings for the main evaluation.

E.2 Cross-Model Relevance of AGNI Novelties

For each novelty, GPT-5.5 identifies affected tools, paths, and command patterns, which we match against executed commands. Table 19 reports interact@1 for base and novel rollouts: the fraction interacting with the affected component, averaged as for pass@1. For novel rollouts, bypass@1 is the fraction of successful rollouts without interaction; Fail w/o Int. is the fraction of failed rollouts without interaction.

	ET-eval				TB-Lite				TB-2			
	Interact@1				Interact@1				Interact@1			
Model	Base	Novel	Bypass@1	Fail w/o Int.	Base	Novel	Bypass@1	Fail w/o Int.	Base	Novel	Bypass@1	Fail w/o Int.
GPT-5.5	99.1	99.0	0.7	1.2	96.9	96.3	1.7	7.2	96.6	98.3	1.4	1.4
GPT-5.4-mini	95.3	95.9	2.7	3.8	92.7	94.9	1.5	8.6	91.2	95.0	0.7	8.1
Kimi-2.6	97.3	98.4	1.6	0.0	94.2	97.8	0.8	3.3	92.6	96.1	2.3	4.2
DeepSeek-V4-Flash	96.1	97.8	1.3	3.4	95.4	96.6	0.7	5.4	94.6	95.5	0.8	4.2
GPT-OSS-120B	95.3	95.3	1.7	4.3	91.0	93.2	0.3	7.8	93.8	94.5	0.0	6.4
Grok-4.20	95.4	97.0	1.7	2.0	91.3	90.7	1.0	10.0	93.1	95.2	0.0	5.8
Mistral-Large-3	94.5	96.0	1.3	3.6	91.7	91.4	1.0	8.5	92.4	93.3	0.0	7.2

Table 19: Cross-model relevance of curated novelties (%).

E.3 Retry Prevalence and Persistence

Retry prevalence measures trajectories with at least one Retry step; *retry persistence* measures $P(\text{Retry}_{t+1} \mid \text{Retry}_t)$ over transitions after the novelty first becomes observable. Retry means repeating a failed strategy or a superficial variant that preserves its assumption (Appendix F).

Model	Retry prevalence Rollouts with ≥ 1 Retry (%)	Retry persistence $P(\text{Retry}_{t+1} \mid \text{Retry}_t)$ (%)
GPT-5.5	9.8	9
GPT-5.4-mini	15.2	50
Kimi-2.6	18.1	6
DeepSeek-V4-Flash	17.4	10
GPT-OSS-120B	27.8	40
Grok-4.20	40.7	45
Mistral-Large-3	28.1	35

Table 20: Retry prevalence and persistence on ET-eval novel rollouts (%).

E.4 Adaptation Stage Definitions

Stage	Definition
Interacted	An executed command touches an affected tool, path, or file. Detected deterministically; no visible effect is required.
Observed	Terminal output exposes novelty-related evidence, such as an error, unexpected result, or changed file state. Recognition is not required.
Recognized	The agent explicitly identifies the observation as unexpected or inconsistent with its assumptions, without necessarily identifying the cause.
Diagnosed	The agent identifies the changed condition sufficiently to explain the problem and guide a response. Restating a symptom is insufficient.
Revised	A changed strategy successfully removes or bypasses the obstacle while preserving the task. Failed attempts and exploits do not count.
Solved	The agent completes the original task under novelty using a task-preserving strategy.

Table 21: Stages of novelty handling used in trajectory analysis.

E.5 Behavioral Transitions and Conditional Success

Model	Diag.→Rev.	Diag.→Retry	Premature Done	Pass | Diag.	Pass | Rev.+Diag.	Pass | Rev. w/o Diag.
GPT-5.5	71	5	14	96	96	68
Kimi-2.6	41	4	11	86	87	50
DeepSeek-V4-Flash	47	5	12	86	86	70
GPT-5.4-mini	57	11	20	87	86	59
GPT-OSS-120B	30	10	21	65	64	22
Grok-4.20	31	14	16	61	64	20
Mistral-Large-3	27	17	31	57	57	16

Table 22: Detailed behavioral transitions and conditional success rates on ET-eval. **Diag.→Rev.** and **Diag.→Retry** measure revision or retry immediately after diagnosis. **Premature Done** denotes trajectories that finalize immediately after a failed or partial step. The final columns report success conditioned on diagnosis and on whether strategy revision occurs with or without a preceding diagnosis. All values are percentages.

For every model, revision preceded by diagnosis is associated with higher success than revision without diagnosis. Table 22 also shows how often models retry or stop after a failed step, and how often diagnosis leads to task success.

F Analyzing Trajectories for Adaptive Behavior

We use a GPT-5.5-based analyzer to label each step of a novel-task trajectory. It receives the task instruction, novelty specification, grader outcome, and full interaction. These labels support the final checks in Algorithm 1: every grader-successful trajectory must show task-preserving adaptation, and no trajectory may show reward hacking. They also locate where unsuccessful trajectories stop progressing toward recovery.

Strategic intents. A step can have multiple substantive intents. The analyzer assigns:

- **Inspect:** examining task inputs, files, or requirements before attempting a solution.
- **Execute:** making the first attempt that produces task output.
- **Diagnose:** investigating the cause of a visible failure or anomaly.
- **Explore:** finding an alternative method without applying it yet.
- **Retry:** repeating a failed strategy or a superficial variant that retains the failed assumption.
- **Revise:** applying a task-preserving fix for the novelty.
- **Repair:** finishing required artifacts after the novelty is handled.
- **Verify:** checking outputs against the task requirements.
- **Finalize:** requesting or confirming completion without further substantive changes.
- **Exploit:** passing visible checks while violating task requirements.
- **Other:** a substantive intent not captured by the labels above.

Adaptation stages. Separately, the analyzer marks when a step establishes each stage:

- **Observed:** an observation exposes evidence specific to the change.
- **Recognized:** the agent treats that evidence as unexpected or inconsistent with its strategy.
- **Diagnosed:** the agent identifies the changed condition well enough to fix it.
- **Revised:** a changed method clears the novelty obstacle; the task may still be unfinished.
- **Solved:** the agent completes or verifies the original task under novelty with a task-preserving method.

We count a stage as reached only when the trajectory supports it. Seeing an error does not establish recognition or diagnosis; trying a different command does not establish a successful revision. The grader determines task success. *Solved* records evidence of how the task was completed and does not replace the grader result.

Evidence grounding. Labels must be supported by the agent's visible reasoning, commands, observed outputs, or deterministic command effects. The novelty specification and grader outcome provide context but cannot establish an intermediate stage: a passing grader result alone does not show that the agent recognized or diagnosed the change. A generic error also does not count as *Observed* unless it specifically points to the changed condition. We count *Recognized* or *Diagnosed* only when the agent engages with the evidence, rather than merely receiving it in terminal output.

Retention predicates. For retention, adaptation must change the method that relied on the invalidated assumption and still satisfy the original task. Repeating a failed method or changing only an unrelated detail does not qualify. An *Exploit* label flags a substitute evaluator-facing artifact or a violation of the required task operation. A different method that genuinely completes the task remains valid even if it differs from S'_c.

Verification-dependent novelties. Some changes yield a clean command result but incorrect task artifacts. We mark these as requiring an explicit check of the resulting state to expose the novelty, distinguishing failure to uncover evidence from failure to act on evidence already seen. The trajectory analyses count progress through these stages, including observation, diagnosis, revision, and task completion. The exact prompt and output schema appear in Appendix H.

G IMPROVING ADAPTATION WITH AGENTIC POST-TRAINING

We use the training hyperparameters of ECHO (Shrivastava et al., 2026) in our post-training experiments (Table 23). The data, prompts, and rewards vary as described below. We train 8B models for at most 500 steps and 14B models for at most 300 steps.

Setting	Value used in our experiments
Optimizer	AdamW; $\beta_1 = 0.9$, $\beta_2 = 0.95$
Weight decay	0.01
Learning rate	10^{-6}, constant; no warmup
Gradient clipping	0.2
Batch size	16
Rollouts per prompt	16
GRPO clipping	$\epsilon_{\text{lo}} = 0.2$, $\epsilon_{\text{hi}} = 0.28$
KL penalty	None
Advantage normalization	Within each prompt group
Loss aggregation	Per sequence
Training temperature	0.8
Training interaction limit	16 turns
Training context limit	16k tokens
Generation limit	2,048 tokens per turn
Training timeouts	Agent: 600 s; verifier: 120 s
ECHO auxiliary weight	$\lambda = 0.05$ (ECHO runs only)
ECHO prediction targets	Terminal output, excluding harness warnings; normalized by total observation length

Table 23: Training hyperparameters used in our experiments, adopted from Shrivastava et al. (2026). ECHO-specific settings apply only to ECHO runs; run lengths are stated above.

G.1 STARTING CHECKPOINTS

All runs start from `Qwen/Qwen3-8B` or `Qwen/Qwen3-14B`, respectively (Yang et al., 2025). Within each model size, the starting checkpoint is fixed across comparisons.

Checkpoint selection. We select checkpoints using performance on 100 base tasks derived from ET that are disjoint from both ET-eval and all training splits. ET-eval therefore remains disjoint from both training and checkpoint selection. TB-Lite provides evaluation on a separate task source.

G.2 CONTROLLED TRAINING COMPARISONS

For Qwen3-8B, we compare the clean, novelty-unpaired, and novelty-paired training sets defined in Section 6. Table 24 reports the corresponding clean and novelty-paired GRPO comparison for Qwen3-14B.

	ET-eval		TB-Lite	
Qwen3-14B GRPO	**Base**	**Novel**	**Base**	**Novel**
Clean	65.3	24.6	17.2	2.9
Novel-paired	67.1	35.5	18.9	5.5

Table 24: Qwen3-14B GRPO pass@1 (%) on ET-eval and TB-Lite.

G.3 META INTERFACE AND PROCESS-REWARD COMPARISONS

Prompt and action interface. The META prompt adds a required `meta_action` field to every bash call, with five possible values: EXECUTE, INSPECT, DIAGNOSE, REVISE, and VERIFY. It also tells the agent when to use each value based on its interaction history. The complete prompts appear in Appendix G.4.

Controlled comparisons. All Q6.3 runs use Qwen3-8B and $D_{\text{novel-paired}} = T_1 \cup T_1'$. We evaluate the paired-data GRPO checkpoint with and without the META prompt, keeping its weights fixed. We train the META model from the base checkpoint with the META prompt and binary task reward. The three process-reward arms also start from the base checkpoint and use the same data, prompt, and otherwise matched training settings; they differ in reward and are not successive training stages.

Process-reward arms. Each arm adds a small shaping term to the binary task reward: $R = r + S$, where $r = 1$ only when the verifier passes. Shaping uses the recorded command tags, commands, exit codes, and `done` call. Missing or invalid tags incur one 0.1 penalty per rollout. Unless noted, event credit is paid in full on a pass and at 25% on a failure. A call fails when its exit code is nonzero; a mutating call changes the environment.

Recovery & Verify gives 0.1 credit when the same command succeeds after an earlier failure and a DIAGNOSE or REVISE tag, and 0.1 for a read-only VERIFY call after the last environment change. Each credit is awarded at most once.

Diagnosed Exploration gives 0.1 for each of up to two distinct new mutating fixes after the first failure followed by DIAGNOSE or REVISE; repeating the failed command earns no credit. It subtracts 0.05 if any command fails and 0.05 for each repeat of the same command without an intervening environment change, up to three. Thus, a failed rollout cannot profit from creating failures to earn fix credit.

Exploration & Completion uses the same exploration reward and penalties, then adds 0.2 for read-only verification after the last change and 0.1 for calling `done`. These last two credits require task success. The rewards use observable command patterns, not a judgment that the agent's diagnosis was correct.

Held-out evaluation. We evaluate on 140 held-out ET-eval base–novel pairs with eight novel rollouts per pair. Base scores use the corresponding paired-task weighting (Table 3). Figure 6 uses all 1,120 novel rollouts per arm to measure diagnosis or revision within two calls after a failure, verification after the last change, explicit `done` calls, and the fraction of calls that fail. These measures record actions, not whether the diagnosis or final answer was correct.

Behavioral effects of process rewards. Relative to binary-reward META training, *Recovery & Verify* raises verification after the last change from 49.4% to 87.3%, but only 0.3% of rollouts call `done`. *Diagnosed Exploration* raises diagnosis or revision within two calls after failure from 31.5% to 88.2%, while failed calls rise from 17.0% to 41.2%. *Exploration & Completion* reaches 84.7% on diagnosis or revision and 72.2% on both verification and completion; novel pass@1 rises only from 37.86% to 38.80%.

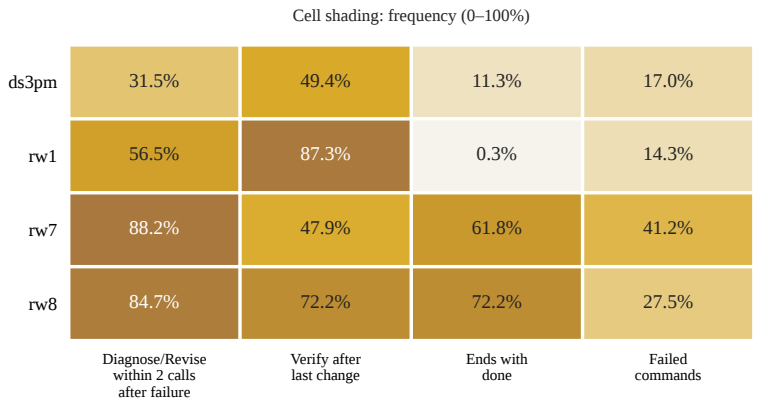


Figure 6: **Process rewards change behavior but yield limited gains on novel tasks.** Held-out ET-eval, 1,120 novel rollouts per arm. Failed commands are a percentage of calls; Table 3 reports task success.

G.4 NORMAL AND META PROMPTS

The listings reproduce both full prompts. Teal text marks additions in the META prompt.

Prompt G.1: Normal Prompt (non-META)

```
# Tools

You have access to the following functions:

<tools>
{"type": "function", "function": {"name": "bash", "description":
"Execute a bash command in the terminal.", "parameters": {"type":
"object", "properties": {"command": {"type": "string", "description":
"The bash command to execute"}, "timeout": {"type": "integer",
"description": "Timeout in seconds (default 30)"}}, "required":
["command"]}}}
{"type": "function", "function": {"name": "done", "description":
"Mark the task as complete. Call this when you have verified the task
is solved.", "parameters": {"type": "object", "properties": {}}}}
</tools>

If you choose to call a function ONLY reply in the following format
with NO suffix:

<tool_call>
<function=example_function_name>
<parameter=example_parameter_1>
value_1
</parameter>
<parameter=example_parameter_2>
This is the value for the second parameter
that can span
multiple lines
</parameter>
</function>
</tool_call>

<IMPORTANT>
Reminder:
- Function calls MUST follow the specified format: an inner
<function=...></function> block must be nested within
<tool_call></tool_call> XML tags
- Required parameters MUST be specified
- You may provide optional reasoning for your function call in
natural language BEFORE the function call, but NOT after
- If there is no function call available, answer the question like
normal with your current knowledge and do not tell the user about
function calls
</IMPORTANT>

You are a highly capable Linux terminal agent. Complete the user's
task by running commands and verifying the result. When the task is
complete, call done.
```

Prompt G.2: META Prompt

```
# Tools

You have access to the following functions:
```

Prompt G.2: META Prompt (continued)

```
<tools>
{"type": "function", "function": {"name": "bash", "description":
"Execute a bash command in the terminal.", "parameters": {"type":
"object", "properties": {"meta_action": {"type": "string", "enum":
["EXECUTE", "INSPECT", "DIAGNOSE", "REVISE", "VERIFY"],
"description": "Meta-strategic goal of this command. Details
described in <IMPORTANT> tags."}, "command": {"type": "string",
"description": "The bash command to execute"}, "timeout": {"type":
"integer", "description": "Timeout in seconds (default 30)"}},
"required": ["meta_action", "command"]}}}
{"type": "function", "function": {"name": "done", "description":
"Mark the task as complete. Call this when you have verified the task
is solved.", "parameters": {"type": "object", "properties": {}}}}
</tools>

If you choose to call a function ONLY reply in the following format
with NO suffix:

<tool_call>
<function=example_function_name>
<parameter=example_parameter_1>
value_1
</parameter>
<parameter=example_parameter_2>
This is the value for the second parameter
that can span
multiple lines
</parameter>
</function>
</tool_call>

<IMPORTANT>
Reminder:
- Function calls MUST follow the specified format: an inner
<function=...></function> block must be nested within
<tool_call></tool_call> XML tags
- Required parameters MUST be specified
- You may provide optional reasoning for your function call in
natural language BEFORE the function call, but NOT after
- If there is no function call available, answer the question like
normal with your current knowledge and do not tell the user about
function calls
- In each turn involving a bash call, first think/reason (content in
<think> ... </think> tags) which meta-action is appropriate given the
interaction history, then think/reason about the specific command to
take conditioned on that meta-action
- Meta-action is the strategic purpose of the next action: EXECUTE,
INSPECT, DIAGNOSE, REVISE, or VERIFY; choose it according to the
primary purpose of the command.
- The `meta_action` parameter in the subsequent `bash` function call
MUST match the meta-action chosen in your reasoning/thinking
- Use meta-actions strategically based on the current situation:
continue executing when the approach is making progress; when
observations reveal missing information, unexpected behavior, failed
assumptions, or that the approach is no longer suitable, inspect,
diagnose, or revise as appropriate before proceeding; verify
important results when needed.
- Use EXECUTE when you already know what approach to use and the next
command directly carries it out
```

Prompt G.2: META Prompt (continued)

```
- Use INSPECT when you need additional information about the
environment or current state before deciding what action to take
- Use DIAGNOSE when a command failed, an observation was unexpected,
or an assumption appears false, and you need to determine the cause
- Use REVISE when evidence shows that the approach being used is
blocked, invalid, or unlikely to succeed, and the next command begins
carrying out a different approach
- Use VERIFY when checking whether a previous action succeeded or
whether an intermediate or final task requirement has been satisfied
</IMPORTANT>

You are a highly capable Linux terminal agent. Complete the user's
task by running commands and verifying the result. When the task is
complete, call done.
```

G.5 Qwen3-14B Generic Post-Training Results

Figure 7 reports the 14B clean-data comparison from Q6.1. GRPO and ECHO improve base and novel pass@1, but the adaptation gap widens under both methods.

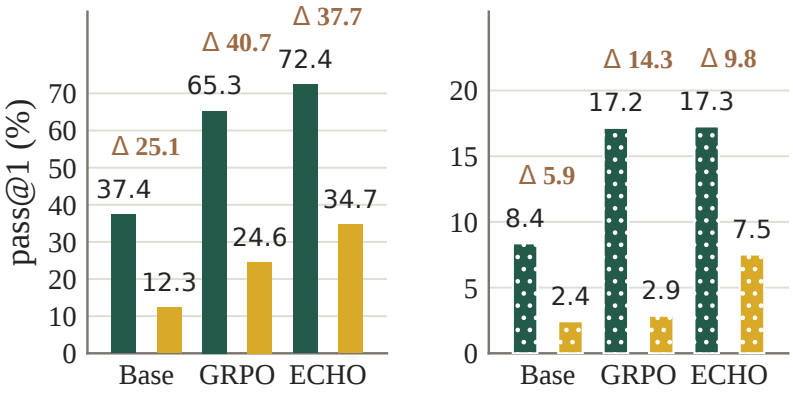


Figure 7: Qwen3-14B generic post-training on ET-eval (solid) and TB-Lite (dotted). Bars show base and novel pass@1 (%); Δ marks the adaptation gap.

H MODEL PROMPTS USED IN AGNI

We provide the prompts and structured output schemas used by each LLM-based component of AGNI. Each model call serves a distinct role in the pipeline. The inputs shown below are the complete prompt-level inputs for that call. The prompt text and JSON schemas are shown together so that each model call can be reproduced without consulting a separate schema. Line wrapping is adjusted only for typesetting; template placeholders are enclosed in braces.

Table 25 maps each prompt to its role and corresponding operation in Algorithm 1. The shared definition in Prompt P.1 is interpolated wherever the placeholder {`novelty_definition`} appears.

Prompt	**Purpose**	**Usage in Algorithm 1**
P.1	Defines novelty for reuse across prompts	Lines 3, 4, 6, 11, 16: shared model context
P.2	Characterizes the base strategy and assumptions	Line 3: M_{A}
P.3	Identifies applicable novelty dimensions	Line 4: $M_{\mathrm{N}}^{\mathrm{dim}}$
P.4	Synthesizes novelty specifications	Line 6: $M_{\mathrm{N}}^{\mathrm{spec}}$
P.5	Instantiates E_c' and adapted S_c'	Line 11: $M_{\mathrm{N}}^{\mathrm{impl}}$
P.6	Repairs a failed task instantiation	Line 11: $M_{\mathrm{N}}^{\mathrm{impl}}$ retry (implicit)
P.7	Validates the expected trajectory change	Lines 16–17: $M_{\mathrm{A}}^{\mathrm{obs}}$
P.8	Analyzes intent, outcome, and adaptation progress	Lines 18–20: ANALYZETRAJECTORIES
P.9	Discloses the change without suggesting a solution	After Algorithm 1: constructs the disclosed-novelty condition
P.10	Scores condition plausibility and realization fidelity	After Algorithm 1: audits novelty realism

Table 25: Prompt-to-operation mapping for the AGNI pipeline and disclosed-novelty evaluation. Line numbers refer to Algorithm 1.

Prompt P.1: Shared Novelty Definition

Definition

An environmental novelty is a task-preserving change to the execution environment that invalidates an assumption underlying a previously successful strategy. On task T, the agent's strategy succeeds in environment E. In the novel task T', the objective and success criterion remain unchanged, but environment E' invalidates an assumption used by that strategy. The agent must discover the change through interaction and adapt, for example by exploring, replanning, switching tools, resolving permissions, or handling inputs more robustly. If the original behavior still succeeds without meaningful revision, the change is not a strong test of adaptation.

Task Definition

A task T consists of:

- instruction: what the agent must accomplish;
- environment: its files, tools, permissions, dependencies, and runtime state;
- evaluator: the tests or grading harness that determine success; and
- reference solution: a known successful strategy for T.

Criteria for Valid Novel Task

A novel task T'=(I,E',G) is valid only if all of the following hold:
1. Task-objective preservation: T' has the same underlying objective as T=(I,E,G); only the execution environment changes.
2. Success-criterion preservation: the grader represents the same notion of success. Mechanical updates for a relocated path or changed invocation must not alter which behavior or final state counts as successful.
3. Environmental change: the novelty changes an operating condition in E and invalidates an assumption used by a successful trajectory; it does not change the task objective.
4. Task solvability: at least one adapted strategy can achieve the original objective in E'.
5. Base-strategy invalidation: behavior that relied on the targeted assumption is insufficient in E', producing an interpretable behavioral difference between T and T'.
6. Robustness and non-trivial adaptation: the change should measurably reduce pass@1 for a weaker or less-careful agent relative to T by breaking a default assumption on which it reflexively relies. Recovery must require the agent to inspect the environment, diagnose the violated assumption, or revise its strategy. Reject changes that are irrelevant, trivially discoverable, trivially bypassed, or leave successful behavior effectively unchanged.
7. Autonomous discovery: the change is not announced, described, or hinted at in the instruction, READMEs, comments, or other task-facing artifacts. The agent must discover it by inspecting state, executing commands, or interpreting errors and outputs. Modify the instruction only when necessary to keep the original objective coherent, never to reveal the novelty.

Constraint

Invalidate a trajectory-relevant assumption rather than introducing a change that is immediately discoverable and mechanically resolved.

- Weak novelty: relocate a file or rename a directory when one obvious inspection step reveals the new location. This does not meaningfully disrupt the strategy.
- Strong novelty: invalidate an assumption during execution and require revision after new evidence. Examples include a missing or shadowed tool with a non-obvious alternative; silent tool-behavior drift; a permission barrier encountered mid-task; a misleading decoy;

Prompt P.1: Shared Novelty Definition (continued)

or a runtime, version, locale, or interface change that alters semantics or output.

Novelty Dimensions

Assign the novelty to one broad dimension:

- perception: what information or state is present and where it appears, including file layout, filenames, decoys, clutter, hidden state, and stale artifacts;
- control: how the agent acts, including shell, tool, CLI, or API substitutions, command syntax, and utility availability;
- constraints: what the agent is allowed or able to do, including permissions, non-root access, write restrictions, time, memory, disk, and network access; or
- dynamics: how actions behave, including package or runtime versions, defaults, ordering, locale, timezone, and parser or runtime behavior.

Optional Modifier Tags

Use these only as tags, not as broad novelty dimensions:

- instruction_underspecified: the instruction provides fewer explicit hints while preserving the objective;
- residual_state: stale, temporary, or cached artifacts are already present; and
- idempotence_cleanup: repeated execution or cleanup of extra artifacts matters.

Required Novelty Description

Every proposed novelty must state:

- broken_assumption: what the original solution or successful agent assumed;
- required_adaptation: what the agent must notice or do differently; and
- expected_failure: how a non-adaptive agent fails.

Identifying a Consequential Novelty

Find an assumption used by successful trajectories, such as a clean initial state, exact file format, available tool, default permissions, or fixed path. It must not be required by the evaluator. Break only that assumption, keep the evaluator's substantive assertions unchanged, and force a different solution path already allowed by those assertions.

Example Patterns

These patterns are examples, not a checklist. Use diverse mechanisms within each dimension rather than repeatedly proposing the same changes. Every novelty must preserve the task objective and substantive success criterion.

- Pre-existing state: a required file or directory already exists with incorrect contents, type, or permissions. A naive mkdir, touch, append, or overwrite fails or produces the wrong result, requiring inspection and repair through chmod, truncate, replacement, or cleanup.
- Decoy resources: a sibling file, directory, executable, or artifact resembles the target. Globbing, broad copying, or selecting the most salient match acts on the decoy; the agent must identify the correct resource.
- Format perturbation: comments, indentation, leading whitespace, a header row, or another formatting convention changes structure without changing semantics. Fixed-column or exact-text parsing fails, requiring robust parsing or inspection.

Prompt P.1: Shared Novelty Definition (continued)

- Tool removal or shadowing: remove, shadow, or alter a reflexively used tool, such as tail, grep, sha256sum, date, or a gawk feature. The agent must diagnose the failure and use an alternative such as Python, sed, awk, or openssl.
- Tool discovery: the expected tool is unavailable but an equivalent exists; several similar tools exist but only one is appropriate; or the correct command or version must be inferred from --version, configuration, documentation, or other evidence.
- Permission or umask change: permissions, ownership, privilege level, or the default umask invalidates assumed write access, executability, or permissive defaults. The agent must inspect and correct the constraint.
- Renamed or undiscovered input: move or rename a hard-coded input. The agent must find it through ls, globbing, metadata, or context. Prefer cases requiring interpretation rather than one obvious lookup.
- Stale state or required cleanup: extra configuration lines, temporary files, locks, stale outputs, or partially completed state interfere with the base strategy and must be reconciled or removed.
- Semantic drift: the interface appears unchanged, but locale, timezone, encoding, sort order, runtime, or dependency version changes its behavior. The command still runs but produces a subtly incorrect result that must be recognized and addressed.
- Forced recovery: a natural first action from the successful strategy fails or produces an informative mismatch. The agent must interpret the error, output, or log and recover through diagnosis and replanning.

Prompt P.2: Base Strategy Characterizer

Role

You are analyzing a terminal task T and solver trajectories collected on T.

Objective

Characterize the successful base strategy and identify the environmental assumptions it relies on. Produce a compact, evidence-grounded summary that will support later novelty generation. Do not propose novel tasks or environmental changes in this step.

Definition of Novelty

{novelty_definition}

Inputs

- Original task files:
{task_files}
- Reference solution (solve.sh):
{reference}
- Base-task solver trajectories with pass/fail labels:
{trajectories}

Requirements

- Return **ONLY** valid **JSON**.
- Treat the tests and reference solution as a evidence of the task objective.
- Use only evidence supported by the task files, reference solution, or trajectories.
- Use exact commands/file paths only when important.
- Separate observed behavior from inferred assumptions.

Prompt P.2: Base Strategy Characterizer (continued)

```
• If something is unclear, say "unclear".
• If no solver trajectory passed, set fields about successful
trajectories to "" or [].
• Summarize behavior for a reviewer; do not reproduce exhaustive
logs.

Output Format
Return exactly one JSON object with this schema:
{
  "task_tldr":
    "<1-3 sentences: objective the tests actually check>",

  "solution_summary": {
    "reference":
      "<how solve.sh achieves the objective>",
    "frontier_successes":
      "<how successful agent runs achieved it; '' if none passed>",
    "frontier_failures":
      "<common reasons unsuccessful attempts failed; '' if none
failed>"
  },

  "pass_fail_contrast": {
    "what_successful_agents_did":
      "<main behaviors that led to success>",
    "what_failed_agents_missed":
      "<main behaviors/assumptions/errors behind failures>",
    "key_difference":
      "<clearest behavioral difference between pass and fail>"
  },

  "important_commands_and_files": {
    "commands": [
      "<important recurring commands only>"
    ],
    "files_or_paths": [
      "<important files/paths inspected, modified, or assumed>"
    ]
  },

  "observed_assumptions": [
    {
      "assumption":
        "<path/tool/shell/version/permission/network/cwd/etc.
assumption>",
      "who_relied_on_it":
        "reference | successful_agents | failed_agents | both",
      "evidence":
        "<brief concrete evidence>",
      "why_it_may_be_brittle":
        "<why changing it could require adaptation>",
      "how_can_assumption_be_broken_for_novelty":
        "<what an agent would need to notice or change if this
assumption broke>"
    }
  ]
}
```

Prompt P.3: Novelty Dimension Identifier

Objective

Given a terminal task T and its behavior summary, identify the broad novelty dimensions that admit a task-preserving, implementable novelty for this task and are likely to produce an observable change in the agent's trajectory.

Shared Novelty Definition

{novelty_definition}

Inputs

- Original task files:
{task_files}
- Base Strategy Characterizer output:
{summary}

Requirements

- Return **ONLY** valid **JSON**.
- Include only dimensions that can be implemented for this task with minimal changes.
- Do not include a dimension merely because it is generally possible.
- Ground each dimension in an observed assumption from the behavior summary.
- If no good dimension exists, return {"applicable": []}.
- Do not propose concrete file edits yet.

Output Format

Return exactly one **JSON** object with this schema:

```
{
  "applicable": [
    {
      "dimension":
        "perception | control | constraints | dynamics",
      "why_this_task":
        "<why this dimension can bite this task specifically>",
      "broken_assumption":
        "<assumption from reference/agents that could be broken>",
      "expected_adaptation":
        "<what the novelty agent would need to notice or do
differently>",
      "risk":
        "<main reason this dimension may fail or become
non-task-preserving>"
    }
  ]
}
```

Prompt P.4: Novelty Candidate Synthesizer

Objective

Propose {n} distinct, concrete, task-preserving novelty specifications for terminal task T. Every specification must belong to the broad novelty dimension {dimension}.

Each specification is a plan, not an implementation. Prefer minimal, auditable changes that can be implemented and are grounded in how agents actually solved T.

Constraints

Prompt P.4: Novelty Candidate Synthesizer (continued)

- Do not add irrelevant noise.
- Do not change the task objective.
- Do not reveal the answer or the environmental change to the agent.
- Do not require many unrelated edits.

Shared Novelty Definition

{novelty_definition}

Inputs

- Past failure context:
{failure_context}
- Base Strategy Characterizer output:
{summary}
- Original task files:
{task_files}

Output Format

Return **ONLY** valid **JSON** with this schema:

```
{
  "specs": [
    {
      "fine_grained_category":
        "<short kebab-case subtype within {dimension}, e.g.
root-and-target-relocation>",
      "concrete_novelty_plan":
        "<the operating-condition change, concretely>",
      "components_to_modify": [
        "environment/Dockerfile",
        "solution/solve.sh",
        "instruction.md",
        "tests/...",
        "harness"
      ],
      "expected_trajectory_change":
        "<how the successful agent run should visibly differ>",
      "why_S_prime_differs_from_S":
        "<why the adapted solution differs from the original>",
      "why_semantics_preserved":
        "<why objective + tests' meaning are unchanged>",
      "task_semantics_preserved": true,
      "minimal_modification_claim": true,
      "implementation_plausibility_score": <1-5>,
      "expected_behavioral_delta_score": <1-5>,
      "risks": [
        "<reasons this novelty may fail>"
      ]
    }
  ]
}
```

Scoring

- implementation_plausibility_score: 1 = impossible or too invasive; 3 = requires moderate care; 5 = trivial to implement.
- expected_behavioral_delta_score: 1 = identical solutions; 3 = some visible change; 5 = the original strategy fails.

Prompt P.5: Novel Task Instantiator

Objective

Implement the task-preserving novelty for terminal task T. Return the full new content of only the files that change.

Shared Novelty Definition

{novelty_definition}

Validity Requirements

1. preserve the same objective;
2. preserve what the tests assert, except for path/invocation updates required by the novelty;
3. be visible in the agent's token stream through commands such as ls, cat, find, which, --version, test errors, permission errors, or runtime behavior;
4. make the unmodified original solve.sh fail or become clearly insufficient under T';
5. include an adapted solve.sh' that succeeds under T';
6. remain realistic and plausible;
7. avoid telling the agent about the novelty in instruction.md unless the original instruction explicitly names something that became invalid.

Inputs

- Novelty dimension: {dimension}
- Novelty subtype: {subtype}
- Novelty plan:
{plan}
- Original task files:
{task_files}

Requirements

- Return **ONLY** valid **JSON**.
- Emit full new content only for changed files.
- Do not include unchanged files.
- tests/: change only paths/tool invocations/setup assumptions required by the novelty; never weaken or change what is asserted.
- instruction.md: edit only if the original instruction explicitly names a changed path/tool/interface; otherwise leave it unchanged.
- solve.sh: provide the adapted reference solution for T' if solve.sh is part of the task files.
- Keep the privileged novelty truth out of the agent-facing instruction.

Output Format

Return exactly one **JSON** object with this schema:

```
{
  "changed_files": {
    "<relpath>": "<full new file content>"
  },
  "novelty": {
    "dimension": "{dimension}",
    "subtype": "{subtype}",
    "intensity": "mild | moderate | severe",
    "changes": ["<exact concrete change made>"],
    "privileged_spec":
      "<full truth of what changed; not shown to agent>",
    "preservation_argument":
      "<why objective and test meaning are unchanged>",
    "why_original_solution_fails":
```

Prompt P.5: Novel Task Instantiator (continued)

```
      "<why unmodified original solve.sh fails or becomes
insufficient>",
    "why_adapted_solution_passes":
      "<why solve.sh' succeeds>",
    "harness_change":
      "none | terminus-2:<what> | echo-minimal:<what>"
  }
}
```

Prompt P.6: Novel Task Instantiation Repair

Objective

The novel task T' failed validation: the adapted solve.sh' did not pass the tests, or the build or solution produced an error. Repair the changed files while preserving the intended novelty.

Shared Novelty Definition

{novelty_definition}

Repair Criteria

1. the adapted solve.sh' passes in T';
2. the original objective remains unchanged;
3. the tests still assert the same behavior/state;
4. the intended novelty subtype remains present and visible;
5. the unmodified original solve.sh still fails or remains clearly insufficient;
6. changes are minimal and auditable.

Inputs

- Novelty subtype: {subtype}
- Original task files:
{task_files}
- Current changed files for T':
{current}
- Validation failure output:
{error}

Requirements

- Return **ONLY** valid **JSON**.
- Do not abandon the novelty unless it is impossible; repair it.
- Do not weaken tests to make them pass.
- Change tests only for path/invocation/setup compatibility with the preserved objective.
- If the proposed novelty is impossible to preserve, return "unrepairable": true with a brief reason.

Output Format

Return exactly one **JSON** object with this schema:

```
{
  "changed_files": {
    "<relpath>": "<full new file content>"
  },
  "novelty": {
    "dimension": "<dimension>",
    "subtype": "{subtype}",
    "intensity": "mild | moderate | severe",
    "changes": ["<exact concrete change made>"],
    "privileged_spec":
```

Prompt P.6: Novel Task Instantiation Repair (continued)

```
      "<full truth of what changed; not shown to agent>",
    "preservation_argument":
      "<why objective and test meaning are unchanged>",
    "why_original_solution_fails":
      "<why unmodified original solve.sh fails or becomes
insufficient>",
    "why_adapted_solution_passes":
      "<why solve.sh' succeeds>",
    "harness_change":
      "none | terminus-2:<what> | echo-minimal:<what>"
  },
  "unrepairable": false,
  "unrepairable_reason": ""
}
```

Prompt P.7: Adaptive Behavior Validator

Objective

Compare how agents solved the original task T and the novel task T'. Determine whether the novelty changed successful trajectories in the expected way.

Shared Novelty Definition

{novelty_definition}

Inputs

- Novelty dimension: {dimension}
- Novelty subtype: {subtype}
- Novelty plan:
{plan}
- Expected trajectory change:
{expected}
- Successful trajectories on T:
{base_traj}
- Successful trajectories on T':
{novelty_traj}

Behavioral Evidence

Check whether T' caused concrete differences such as:

- different commands
- different files inspected or modified
- discovery before editing
- avoiding removed tools
- handling changed permissions
- ignoring decoys
- adapting to version/runtime errors
- cleanup/idempotence behavior
- changed test/debug loop

Requirements

- Return **ONLY** valid **JSON**.
- Be faithful to trajectories; do not infer differences not visible in logs.
- If trajectories are too sparse to judge, say so.
- A novelty can be valid even if the behavioral delta is weak; distinguish validity from visibility.

Output Format

Prompt P.7: Adaptive Behavior Validator (continued)

```
Return exactly one JSON object with this schema:
{
  "novelty_visible": true,
  "matches_expected_delta": true,
  "behavioral_delta_strength": "strong | medium | weak | unclear",
  "observed_differences": [
    "<concrete difference seen in T' trajectory>"
  ],
  "unchanged_behaviors": [
    "<important behaviors that stayed the same>"
  ],
  "validity_concerns": [
    "<possible task drift, weak evidence, flaky behavior, etc.>"
  ],
  "rationale": "<1-3 sentences>"
}
```

Prompt P.8: Trajectory Adaptation Analyzer

Role

You are an expert in terminal tasks, shell behavior, command-line debugging, and agent-trajectory analysis.

Objective

Analyze each step of an agent trajectory in a novel environment. Summarize the step's main semantic intent, assign every substantive intent class, judge the outcome of each intent from visible evidence, and label progress in handling the environmental novelty. Do not summarize commands individually or invent unsupported details.

Novelty-Handling Stages

Assign zero or more stages to each step:

- observed: the observation newly exposes direct evidence of the novelty or a clearly novelty-attributable consequence.
- recognized: the agent explicitly treats prior evidence as unexpected, suspicious, or inconsistent with the task.
- diagnosed: the agent identifies the effective changed condition at the level needed to adapt.
- revised: the step successfully removes or bypasses the novelty obstacle, but the complete task remains unfinished or incorrect.
- solved: the step completes or verifies the full task under the novelty using a task-preserving method.

Use [] when no stage applies. Do not assign observed from hidden provenance or grader information, and do not mark the same evidence observed again unless the step reveals something new. Evidence exposed in step N's observation may be labeled recognized or diagnosed only when the agent subsequently engages with it. A failed attempted revision is not revised. Assign solved only when the complete task state is produced or verified by a task-preserving method. Deterministic command semantics may support solved even when the intent outcome is inconclusive because exact post-hoc verification is absent. An exploit may still be observed, recognized, or diagnosed, but it is never solved.

Intent Classes

Assign all substantive intents present in each agent step:

- inspect: examine ordinary task inputs, files, or requirements before attempting a solution.

Prompt P.8: Trajectory Adaptation Analyzer (continued)

- execute: make the first substantive task-producing attempt.
- diagnose: investigate the cause of a visible failure or anomaly.
- explore: look for another tool, file, path, resource, or method without applying it yet.
- retry: repeat the same failed method or a superficial variant that preserves the failed assumption.
- revise: apply a task-preserving changed method intended to overcome the novelty.
- repair: finish, format, permission-fix, regenerate, or clean up required artifacts after the novelty obstacle is handled.
- verify: check existing outputs against task requirements without materially changing them.
- finalize: request or confirm completion without making substantive environment changes.
- exploit: satisfy visible checks while replacing or violating the required task semantics.
- other: use only when no listed class applies, and explain why.

Write one intent_summary per step and one intent-analysis object per assigned class. When reasoning and actions disagree, classify the actions and note the mismatch. Ignore incidental setup, logging, or checks that do not affect the step's main purpose.

Use execute for the first substantive attempt even when it is already robust to the novelty. After a relevant failure, use retry when the new attempt preserves the failed assumption and revise only when it changes the novelty-relevant method. An unrelated typo, quoting error, or setup mistake does not by itself turn the next real attempt into retry. A step may receive multiple labels, such as diagnose and revise, when both behaviors are substantive. After the novelty obstacle is handled, use repair for remaining artifact work rather than revise.

Intent Outcomes

For each assigned intent, choose success, partial, failure, or inconclusive using the step's observation and visible deterministic command semantics. Judge each intent separately.

For verify, instead use:
- verify_valid: the check covers the relevant criteria and the artifact is correct.
- verify_invalid: the check correctly establishes that the artifact is wrong.
- verify_partial: the check covers only part of the relevant criteria.
- verify_inconclusive: the check fails, tests the wrong requirement, or provides no usable verdict.

For diagnose, success means identifying the effective cause and partial means narrowing it to a useful symptom or component. For explore, success means finding a usable alternative. An expected failing subcommand inside an explicit fallback does not by itself make the intent fail; judge whether the fallback succeeds.

For finalize only, the grader result may be used together with the observation. A grader pass obtained through an exploit is still a successful finalization, but it is not a genuine novelty fix.

Grounding Rules

- Base every per-step label and outcome on that step's reasoning, actions, observation, or deterministic command semantics.

Prompt P.8: Trajectory Adaptation Analyzer (continued)

- Use the novelty description, original solution, and grader result only as context; do not backfill per-step progress from them.
- A generic error, missing file, or empty output is not observed unless the evidence specifically connects it to the changed condition.
- Do not infer recognized or diagnosed merely because terminal output contains a clue; the agent must engage with it.
- Assign revise only when the changed method addresses the novelty-relevant assumption, not for superficial flag, path, ordering, or quoting changes.
- Assign exploit only when the action replaces the genuine required operation to satisfy a check. A different method that genuinely achieves the required state is not an exploit.
- An exploit is revised only if it genuinely clears or bypasses the novelty obstacle; merely fabricating evaluator-visible artifacts does not count as revision.
- Do not treat completion claims, completion-tool calls, or superficial checks as proof that the task is solved.
- When exactness matters and the relevant output is truncated or unverified, use inconclusive.
- Do not invent task requirements, constraints, tests, or failure causes absent from the visible inputs.

Constraint Violations

Separately list stated task constraints that the agent visibly violates, such as privilege, allowed-directory, forbidden-operation, package-source, or required-path constraints. Do not infer unstated constraints. Ordinary commands executed from a root shell do not alone violate a no-root constraint. Temporary use of /tmp or /var/tmp does not violate an allowed-directory constraint unless a required artifact is placed outside the allowed location. Set grader_enforced to false if the grader passes despite the violation. Constraint violations do not change intent outcomes or novelty-handling stages. Use [] when no stated constraint is violated.

Output Format

Return **ONLY** valid **JSON** with this structure:

```
{
  "task_goal": "<one or two sentences>",
  "constraint_violations": [
    {
      "step_id": 3,
      "constraint": "<stated constraint>",
      "violation": "<visible violation>",
      "grader_enforced": false
    }
  ],
  "step_analysis": [
    {
      "step_id": 1,
      "actor": "user",
      "intent_summary": "<summary of the task instruction>",
      "intent_analysis": [],
      "intent_outcomes": [],
      "novelty_handling_analysis": [],
      "novelty_handling_comment": "<why no novelty label applies>",
      "comment": ""
    },
    {
      "step_id": 2,
      "actor": "agent",
      "intent_summary": "<main subgoal and semantic strategy>",
```

Prompt P.8: Trajectory Adaptation Analyzer (continued)

```
      "intent_analysis": [
        {
          "intent_reasoning": "<evidence for this intent>",
          "intent_class": "execute"
        }
      ],
      "intent_outcomes": [
        {
          "intent_class": "execute",
          "outcome_justification": "<visible evidence for the
outcome>",
          "outcome": "success"
        }
      ],
      "novelty_handling_analysis": [
        {
          "reason": "<visible evidence for the stage>",
          "label": "observed"
        }
      ],
      "novelty_handling_comment": "<why no additional stages apply>",
      "comment": "<optional ambiguity or reasoning-action mismatch>"
    }
  ],
  "trajectory_summary": "<concise account of novelty handling over
time>"
}
```

Output Requirements

- step_id must match the trajectory, and actor must be user or agent.
- User-only instruction steps have empty intent_analysis and intent_outcomes arrays.
- Each agent intent has one intent_analysis object and one matching intent_outcomes object.
- Write intent_reasoning before intent_class, and outcome_justification before outcome.
- Every intent_class is one of inspect, execute, diagnose, explore, retry, revise, repair, verify, finalize, exploit, or other.
- novelty_handling_analysis contains zero or more objects with reason before label.
- Every novelty label is one of observed, recognized, diagnosed, revised, or solved.
- Ground every justification in visible evidence or deterministic command semantics.
- Return no Markdown fences, comments, or prose outside the **JSON**.

Inputs

```
Task instruction:
{{original_task}}

Novel environment details:
{{shift_description}}

Original successful solution S:
{{original_solution}}

Agent trajectory in the novel environment:
{{agent_trajectory}}

Grader result (context only, except for finalization outcome):
{{grader_result}}
```

Prompt P.9: Novelty Disclosure Generator

Role

You are preparing a novelty-disclosed version of a terminal-task instruction for an agentic evaluation.

Context

The task environment has been modified in a specific, non-standard way. The task objective and required final state are unchanged. In the standard condition, the agent is not told about the modification and must discover it. In this disclosed condition, the agent is told exactly what is non-standard, but not how to work around, fix, or adapt to it.

Objective

Write a short environment note of 2-6 sentences in plain prose to append to the task instruction.

Requirements

- State clearly and factually what is non-standard: identify the modified component and explain how it behaves differently from the standard environment.
- Describe only the environmental change and its observable behavior. Do not restate the task objective or required final state.
- Never reveal, recommend, or hint at a workaround, fix, alternative tool, command, flag, path, or adaptation strategy. Do not say "use X instead," "you should," or "to handle this." Ignore any adaptation or solution guidance in the internal description.
- Do not mention the reference solution, tests, graders, benchmarks, scoring, or that a novelty was applied. Write a neutral factual notice about the environment.

Inputs

- Original task instruction:
{instruction}
- Environmental modification (disclose the change, never the fix):
{novelty_changes}

Output Format

Output **ONLY** the environment-note prose. Do not include a heading, preamble, or Markdown fence.

Prompt P.10: Novelty Realism Judge

Role

You are auditing the realism of a synthetic environmental novelty injected into a terminal-based software-agent task. The original task T was changed into T' by modifying only the environment. The goal remains unchanged, but the agent's previously successful strategy should fail and require adaptation.

Objective

Evaluate two levels independently:
- Underlying condition: the semantic environmental change that invalidates the prior strategy, such as a tool interface differing from the assumed version.
- Realization mechanism: how that condition is concretely instantiated, such as a different installed version, modified CLI, or incompatible executable earlier in PATH.

Prompt P.10: Novelty Realism Judge (continued)

```
An observable symptom is evidence about the realization, not a
separate dimension. Follow this order: identify the underlying
condition, score its occurrence plausibility, score realization
fidelity, and classify the implementation mechanism.

A. Underlying Condition
Infer the underlying condition from the supplied artifacts and state
it independently of the benchmark-specific implementation. Cite the
files, paths, patch hunks, or interfaces that establish it. Do not
alter the inferred condition to justify a preferred score.

Judge from the diff and changed files rather than the generator's
claims. When the artifacts contradict or narrow the claimed
condition, the artifacts win. For example, if the specification
claims a general version change but the diff contains a wrapper that
special-cases one argument sequence, evaluate the narrower
implemented behavior.

B. Occurrence Plausibility
Question: could the underlying condition reasonably arise during real
deployment or maintenance of a terminal-based software agent?

1. Implausible: no credible real-world mechanism; mainly an evaluator
trick or behavior real systems would not reasonably exhibit.
2. Technically possible but contrived: requires an artificial setup,
unlikely coincidence, or mechanism unrepresentative of ordinary
deployment.
3. Plausible in some deployments: has a credible mechanism but is
uncommon, domain-specific, or somewhat stylized.
4. Clearly realistic: a recognizable deployment, configuration,
dependency, permission, filesystem, interface, or runtime change
encountered by practitioners.
5. Directly representative: closely matches a documented, observed,
or widely recognized deployment failure mode strongly supported by
the artifacts.

Do not assign 5 merely because the condition is imaginable. Score the
condition itself, independent of implementation quality, reference
solution, or recovery strategy.

C. Realization Fidelity
Question: does the concrete implementation faithfully reproduce how
the condition would manifest in a real system?

1. Unfaithful or incoherent: contradicts the condition, relies mainly
on evaluator-only machinery, or changes the task through an unrelated
mechanism.
2. Crude proxy: captures the broad idea but introduces major
artificial artifacts or substantially different behavior.
3. Controlled abstraction: preserves the central causal effect and
relevant adaptation but simplifies meaningful aspects coherently.
4. Faithful simulation: uses realistic mechanisms and produces
deployment-like observations and consequences, with only minor
simplification or scaffolding.
5. Direct or near-exact reproduction: directly instantiates the real
mechanism, or reproduces its behavior, scope, consequences, and
recovery options with negligible abstraction.

Judge the realization from the diff and changed files. A wrapper,
shim, or fixture is not automatically unfaithful; judge whether it
preserves the causal mechanism and agent-visible behavior. A wrapper
```

Prompt P.10: Novelty Realism Judge (continued)

```
that special-cases one exact command or anticipated solution is
generally a crude or controlled proxy, not a faithful simulation. Do
not raise the score merely because the goal is preserved, the task is
solvable, or recovery exists.

Required Evidence
For the condition and both scores, cite concrete evidence such as a
file, path, patch hunk, permission, command interface, package or
runtime version, error behavior, or configuration value. Explain:
• what condition the shift instantiates;
• how that condition could arise in real systems; and
• whether the concrete implementation matches that mechanism.

Distinguish observed evidence from assumptions. Do not use generic
claims such as "such issues happen in practice." The reference
solution, tests, and grader are withheld; judge only from the
environment construction and agent-facing instruction.

D. Mechanism Classification
Classify the implementation mechanism independently of the two scores:
• Choose exactly one mechanism family copied verbatim from the
supplied taxonomy, or other if none fits.
• For other, provide a plain-language family name and representative
example in proposed_family_if_other.
• Classify the realization visible in the diff, not the
specification's claimed dimension.
• adaptive_cluster must exactly match a taxonomy cluster. For other,
choose the cluster whose recovery loop best matches what the agent
must do.
• surface_binding must name the concrete surface touched, such as a
tool, binary, path, environment variable, file format, or encoding.

Taxonomy:
{taxonomy}

Inputs
• Task ID: {task_id}
• Agent-facing instruction for T':
{instruction}
• Generator's novelty specification:
{spec}
• Unified diff from T to T':
{diff}
• Full contents of changed files in T':
{changed_files}

Output Format
Return ONLY one JSON object with exactly this shape and no extra keys
or Markdown. Within each scored dimension, place evidence and
rationale before the score.

Allowed confidence values are low, medium, and high.

{
  "task_id": "{task_id}",
  "change_summary": "<one or two sentences describing the
environmental change>",
  "underlying_condition": "<semantic condition stated without
implementation details>",
  "occurrence_plausibility": {
    "condition_identification_evidence": [
```

Prompt P.10: Novelty Realism Judge (continued)

```
      {
        "source": "<file, path, hunk, or interface>",
        "observation": "<what the artifact shows>",
        "implication": "<what condition this establishes>"
      }
    ],
    "real_world_basis": "<how the condition could arise in real
systems>",
    "rationale": "<why the mechanism is or is not credible in
deployment>",
    "score": 1,
    "confidence": "low"
  },
  "realization_fidelity": {
    "evidence": [
      {
        "source": "<file, path, hunk, or interface>",
        "observation": "<what the artifact shows>",
        "implication": "<what this establishes about the realization>"
      }
    ],
    "rationale": "<how faithfully the realization implements the
condition>",
    "score": 1,
    "confidence": "low"
  },
  "mechanism": {
    "rationale": "<why the selected family matches, or why none
matches>",
    "mechanism_family": "<verbatim taxonomy family, or other>",
    "proposed_family_if_other": "<name and example, or empty if not
other>",
    "adaptive_cluster": "<verbatim taxonomy cluster>",
    "surface_binding": "<specific tool, path, variable, format, or
encoding>",
    "confidence": "low"
  }
}
```