EVORULE HARNESS: LIFELONG HARNESS ADAPTATION WITH VALIDATED INTERVENTION RULES

Anonymous authors
Paper under double-blind review

ABSTRACT

As general-purpose models are increasingly deployed with fixed weights, harness optimization is emerging as a key paradigm for adapting fixed-weight models to diverse tasks and deployment settings by managing context, tool use, and memory. However, deployment is inherently non-stationary, as tasks, tools, and interaction patterns change over time, while existing memory mechanisms are often rigid, training-dependent, and brittle, making reliable evolution over long horizons challenging. To address these limitations, we propose *EvoRule Harness*, a plug-and-play, continually adaptive memory system for long-horizon harness evolution. Given a frozen backbone model and an existing harness optimizer, EvoRule Harness builds a dynamic two-tier memory of causal rules. Each rule specifies an application condition and an intervention, and tracks execution-outcome feedback. Rules are revised online through individually traceable patches, allowing the effect of each update to be tracked and rolled back when necessary. Rule updates first enter a volatile tier and are promoted to stable memory only after repeated validation across rounds and data subsets, while unreliable updates are discarded. EvoRule Harness can be seamlessly integrated with diverse existing harness optimization methods, without requiring gradients or additional training. We evaluate EvoRule Harness on diverse tasks, including retrieval-augmented mathematical reasoning and interactive agentic tasks. EvoRule Harness improves Meta-Harness, TTHE, and Continual Harness by 7.0%, 8.5%, and 35.7% in average over Tau-Bench, Terminal-Bench-2, and SWE-Bench Verified, respectively. In particular, Meta-Harness augmented with EvoRule Harness achieves the best overall average of 67.4% among all harness optimization framework baselines, improving the base Meta-Harness from 63% by 7.0%. Our code is available at https://anonymous.4open.science/r/EvoMem-2C26/.

1 INTRODUCTION

The behavior of a deployed LLM system is shaped not only by the weights of the underlying model, but also by its *harness* (Pan et al., 2026; Meng et al., 2026): the surrounding system layer that orchestrates context, tool use, memory, and execution. Because retraining a frontier model is costly or inaccessible to most practitioners, the harness has emerged as a practical adaptation layer for tailoring frozen general-purpose models to new tasks and specialized deployment settings. This has naturally shifted attention from manually designing harnesses to optimizing them automatically. Although this research direction is still recent, it has progressed rapidly (Sengupta & Wang, 2026; Ursekar et al., 2026). Early methods primarily focused on fixed, offline optimization. Meta-Harness (Lee et al., 2026b), for example, performs iterative search over candidate harnesses using an agentic proposer that can selectively inspect the source code, evaluation scores, and execution traces of all prior candidates. These works have shown that automated harness optimization can outperform hand-designed systems (Park et al., 2026; Lee et al., 2026b). However, once offline search concludes, the resulting harness remains fixed at deployment. When the task distribution shifts due to new failure modes, tool changes, or model updates, the search must be rerun to revise the harness.

This limitation has motivated a new line of work that adapts the harness online by leveraging execution experience during deployment. Recent works (Huang et al., 2026; Nie et al., 2026; Liu et al., 2026) show that accumulated experience can reduce repeated search and enable successful interventions to be reused across tasks. Yet existing mechanisms are not designed to fully support

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.

lifelong harness adaptation. Some approaches rely on a fixed bank of past experiences constructed before deployment, which cannot readily absorb new failure modes. Others adapt online but retain no persistent, structured record of which edits were attempted, why they succeeded or failed, or how much they should be trusted (Park et al., 2023; Zhao et al., 2024; Mi et al., 2026; Chen et al., 2026). As a result, accumulated experience can be difficult to revise without corrupting previously reliable knowledge, making stable adaptation over long horizons challenging.

We argue that lifelong harness adaptation requires memory that can absorb new failures while explicitly separating tentative updates from validated knowledge. To this end, we introduce **EvoRule Harness**, a plug-and-play framework for lifelong harness adaptation built around a two-tier memory of validated causal rules. Unlike existing memory-based approaches that store undifferentiated experience, EvoRule Harness represents each adaptation as a testable causal rule linking an applicability context to an intervention, together with execution feedback, confidence, and lifespan. When a failure occurs, the resulting reflection is converted into independently traceable atomic patches over Add, Delete, Refine, Split, and Merge, allowing each update to be validated, attributed, and rolled back separately.

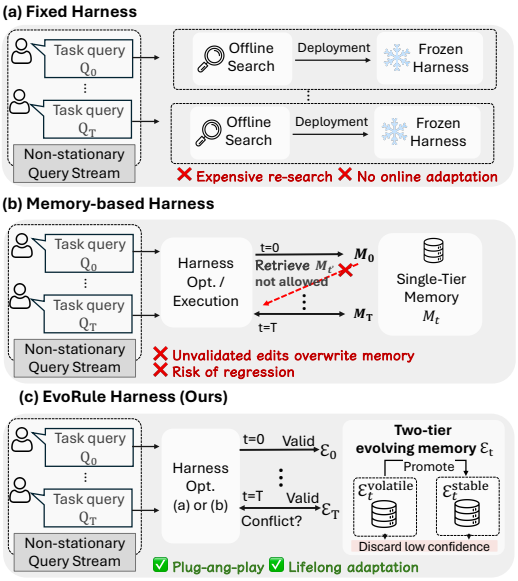


Figure 1: Comparison of harness optimization.

New or revised rules first enter the volatile tier, where they are re-tested and continuously revalidated as new evidence arrives. Their confidence is recalibrated from cumulative successes and failures, and only rules that remain reliable over repeated validation are promoted to stable memory. Crucially, volatile rules can already guide inference, but cannot directly overwrite stable knowledge; when the two tiers prescribe conflicting interventions, stable rules take precedence. Stable memory itself evolves conservatively, with modifications restricted to Merge or Delete after repeated cross-subset failures. This design allows the harness to remain responsive to new failure modes while insulating repeatedly validated knowledge from transient or unreliable updates.

EvoRule Harness operates entirely around a frozen backbone and existing harness optimizer, requiring no gradients, weight updates, or task-specific training. Across reasoning, classification, and interactive agentic tasks, it consistently improves diverse harness optimization frameworks, including Meta-Harness, TTHE, and Continual Harness, with average gains of up to 35.7%. More importantly, its advantage grows over sequential adaptation rounds, suggesting that validated rules accumulate without sacrificing responsiveness to new failures. Ablations further show that removing the stable tier, volatile tier, or promotion mechanism substantially degrades performance, highlighting controlled consolidation as a key ingredient for reliable lifelong harness adaptation.

2 Related Works

From Context to Fixed Harness. Early LLM adaptation mainly optimizes context or prompts through methods such as OPRO (Yang et al., 2024), AlphaEvolve (Novikov et al., 2025), and GEPA (Agrawal et al., 2026). Recent work, however, shifts the optimization target to the harness: the code that controls context, tools, memory, and failure recovery. Meta Harness (Lee et al., 2026b), Meta Context Engineering (Ye et al., 2026), and Agentic Context Engineering (Zhang et al., 2026) search or evolve harnesses and playbooks before deployment, often outperforming hand-designed or prompt-only baselines. However, these systems typically freeze the discovered harness after search, so they cannot respond to distribution drift, tool API changes, or model replacement without rerunning expensive offline search. EvoRule Harness instead keeps a plug-and-play memory layer evolving after deployment.

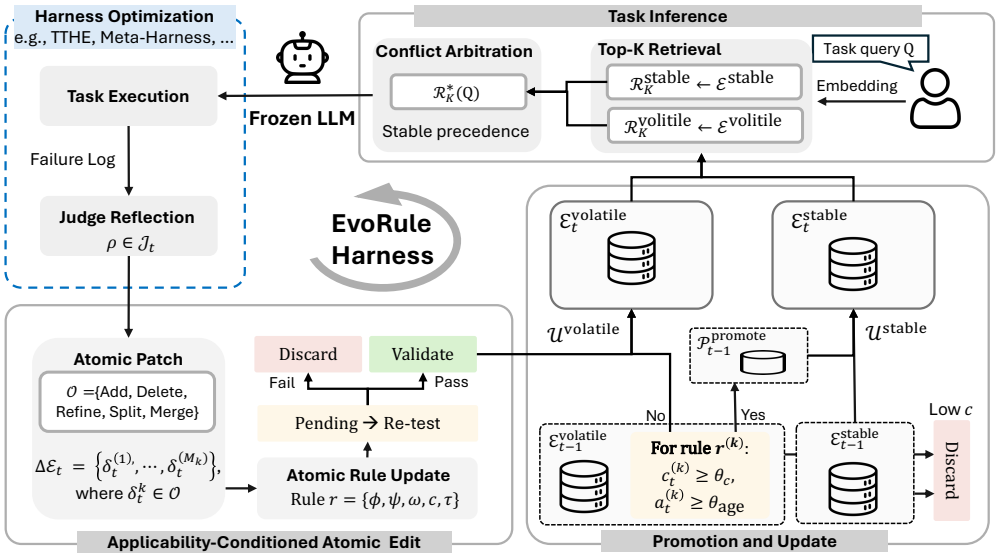


Figure 2: Overall framework of EvoRule Harness

Harness Optimization. Existing harness optimization frameworks can be grouped into three families: offline search and experience-bank retrieval (e.g., Meta-Harness (Lee et al., 2026b), Memo-Harness (Huang et al., 2026)), unsupervised online evolution (e.g., TTHE (Nie et al., 2026)), and training-free reflective memory (e.g., Reflexion (Shinn et al., 2023), MemGPT (Packer et al., 2023), Voyager (Wang et al., 2023)). Offline methods are static and require rebuilding or retraining when the distribution shifts (Gao et al., 2025); unsupervised online methods rely on noisy signals and keep no persistent structured record of why edits succeed or fail (Wang et al., 2025); training-free memory methods grow episodic or distilled text but rarely re-verify it (Zhou et al., 2026) and lack calibrated confidence, rollback, and a clear distinction between stable and volatile memory (Armstrong, 2026; Mireshghallah et al., 2026). Consequently, none gives memory an explicit epistemic status or prevents a single bad round from corrupting reliable knowledge (Barkan et al., 2026); EvoRule Harness instead uses causal rules, attributable patches, precedent-based influence screening, and evidence-gated promotion.

Lifelong Harness Adaption. Continual and lifelong harness adaptation must avoid catastrophic forgetting while staying flexible under non-stationary streams (Kang et al., 2026; Han et al., 2026; Cho et al., 2026); online harness editors (TTHE (Nie et al., 2026), Continual Harness (Kang et al., 2026), RHI (Lee et al., 2026a), Harness-R1 (Shao et al., 2026)) update from execution feedback but largely lack independently attributable, confidence-calibrated rules, whereas memory-based agents accumulate experience yet mix tentative and validated knowledge (Shinn et al., 2023; Packer et al., 2023; Wang et al., 2023; 2024; Huang et al., 2026; Ouyang et al., 2026; Zhang et al., 2025); inspired by complementary learning systems (Du & Zhao, 2026; Lerma-Torres, 2026), EvoRule Harness introduces a stable and volatile rule memory that first isolates edits in the volatile tier and promotes them only after repeated cross-subset evidence (Du & Zhao, 2026), enabling interpretable, training-free, and non-regressive lifelong harness evolution (Ouyang et al., 2026; Kang et al., 2026).

3 Lifelong Harness Adaptation with EvoRule Harness

3.1 Applicability-Conditioned Atomic Edit Operations

The existing memory-based harness methods, which rely on passively accumulated or static memory, show limited long-term adaptability. To address the needs of lifelong adaptation, we first formulate $\mathcal{E}_t$ as a two-tier memory of causal rules $r = \{\phi, \psi, \omega, c, \tau\}$, where ϕ is the applicability context, ψ is the intervention, ω is execution feedback, c is confidence, and τ denotes lifespan. To maintain and evolve this memory, we introduce a set of applicability-conditioned atomic operations as:

$$\mathcal{O} = \{\textsc{Add}, \textsc{Delete}, \textsc{Refine}, \textsc{Split}, \textsc{Merge}\}. \quad (1)$$

The judge first produces a set of diagnostic reflections $\mathcal{J}_t$, where each $\rho \in \mathcal{J}_t$ is a non-executable signal grounded in the failure case and the matched rule and must not directly modify stable memory; each ρ follows the diagnostic schema defined in Sec. A.3. Given $\mathcal{J}_t$, an LLM-based proposer maps each $\rho \in \mathcal{J}_t$ to one or more atomic patches $\delta \in \mathcal{O}$, using the failure case, matched rule, validation statistics, and stable-tier safety constraints. Concretely, the proposer is prompted with the failure case, the matched rule, its validation statistics, and the stable-tier safety constraints, and returns a structured JSON patch specifying the operation, target rule, candidate rule content, provenance, and rollback metadata(See Sec. A.3 for details). The judge and proposer share the same backbone but are invoked under different prompts and roles. Thus, the judge diagnoses; the proposer proposes executable edits; the caller validates and applies them.

Each edit is stored as an independent patch with a target rule and an influence score estimating its likely benefit. Even when multiple components change in the same round, patches are never grouped together; each is logged separately with its own status, pass or fail outcome, and rollback capability. This keeps every update individually attributable, so any performance change can be traced to the responsible patch, enabling precise credit assignment. Formally, round t yields

$$\Delta\mathcal{E}_t = \{\delta_t^{(1)}, \ldots, \delta_t^{(M_t)}\}, \quad M_t = |\Delta\mathcal{E}_t|, \quad \text{each } \delta_t^{(i)} \in \mathcal{O}. \tag{2}$$

Here, superscript (i) indexes patches within a round, and the subscript t records the round amd M_t is the number of patches in round t. $\mathcal{J}_t$ denotes the judge reflections, each a diagnostic signal with a suggested repair rather than an executable edit. The executor converts each reflection into one or more atomic patches, so a patch specifies an operation in $\mathcal{O}$, its target rule, candidate content, logging and rollback metadata.

3.2 Stable and Volatile Memory for Lifelong Harness Adaptation

A central challenge in lifelong adaptation is that a single unfavorable round can corrupt the entire rule base, as one bad edit can overwrite validated rules and propagate errors across memory. To mitigate this, we maintain two separate tiers of rules with different plasticity. The *stable tier* stores rules that have been repeatedly validated across multiple rounds and whose calibrated confidence c exceeds a threshold θ_c; we treat these as reliable knowledge. The *volatile tier* acts as a fast scratch-pad for recent edits, unresolved patches, and hypotheses that have not yet accumulated sufficient evidence. Formally, the overall rule base is the disjoint union of the two tiers:

$$\mathcal{E}_t = \mathcal{E}_t^{\text{stable}} \cup \mathcal{E}_t^{\text{volatile}}, \quad \mathcal{E}_t^{\text{stable}} \cap \mathcal{E}_t^{\text{volatile}} = \emptyset. \tag{3}$$

The stable tier keeps rules we trust, while the volatile tier keeps newly proposed edits that are not yet proven, so they can be tested and discarded if they fail without affecting trusted knowledge. Importantly, both tiers self-evolve internally. In the stable tier, confidence c_t is continually recalibrated from cumulative validation feedback as testing data accumulate (Eq. (6)); each failure lowers it, and once below the stable discard threshold $\theta_{\text{drop}}^{\text{stable}}$, the rule is removed. The volatile tier follows the same mechanism but with a lower threshold $\theta_{\text{drop}}^{\text{volatile}} < \theta_{\text{drop}}^{\text{stable}}$, so unproven edits are discarded faster. New entries enter the volatile tier as PENDING reflections that passed the re-test samples, and become eligible for promotion only after repeated validation. The key design choice is to keep updates to the two tiers separate: a volatile patch never directly modifies a stable rule, so noise from a single bad round stays confined to the volatile tier. As a result, the overall update function decomposes into independent operations for each tier. Let $\mathcal{E}_t = \mathcal{E}_t^{\text{stable}} \cup \mathcal{E}_t^{\text{volatile}}$ be the full rule base at round t. The next state is given by $\mathcal{E}_{t+1} = \mathcal{U}(\mathcal{E}_t)$, where the update operator $\mathcal{U}$ decomposes as

$$\mathcal{E}_{t+1} = \left(\mathcal{U}^{\text{stable}}(\mathcal{E}_t^{\text{stable}}, \Delta\mathcal{E}_t^{\text{stable}}) \cup \mathcal{P}_t^{\text{promote}}\right) \cup \mathcal{U}^{\text{volatile}}\left(\mathcal{E}_t^{\text{volatile}} \setminus \mathcal{P}_t^{\text{promote}}, \Delta\mathcal{E}_t^{\text{volatile}}\right), \tag{4}$$

where $\mathcal{U}^{\text{stable}}$ and $\mathcal{U}^{\text{volatile}}$ are the update operators for each tier, $\Delta\mathcal{E}_t^{\text{stable}}$ and $\Delta\mathcal{E}_t^{\text{volatile}}$ are the edits applied to each tier in round t, and $\mathcal{P}_t^{\text{promote}}$ is the set of volatile rules qualifying for promotion:

$$\mathcal{P}_t^{\text{promote}} = \left\{ r^{(k)} \in \mathcal{E}_t^{\text{volatile}} \mid \text{Promote}(r_t^{(k)}) = 1 \right\}. \tag{5}$$

Here, $\text{Promote}(r_t^{(k)}) = \mathbb{I}(c_t^{(k)} \geq \theta_c, a_t^{(k)} \geq \theta_{\text{age}})$, where $\mathbb{I}$ is an indicator function. Promoted rules are first removed from the volatile tier and then atomically added to the stable tier; hence, promotion is one-way, and stable memory is never directly modified by volatile edits.

How a rule becomes stable: promotion based on evidence. An edit enters the volatile tier and eventually earns promotion to the stable tier as follows. A new edit δ produces a rule $r^{(k)}$ that is initially written only to the volatile tier as PENDING, recording its provenance (round, failure case, and reflection) and a unique undo reference enabling individual rollback without disturbing other updates, ensuring traceability and reversibility (see Sec. A.3). It also carries its applicability context ϕ and embedding $\mathbf{z} = f(\phi)$ from a fixed text encoder (see Sec. 4.1), and never directly edits a stable rule. Immediately after application, the modified harness is re-evaluated on the previously failed data point; if it passes, its status changes from PENDING to VALIDATED in the volatile tier. After that, its cumulative successes $S_t^{(k)}$, failures $F_t^{(k)}$, and survival age $a_t^{(k)}$ are tracked across subsequent validation subsets so that its confidence is recalibrated by:

$$c_t^{(k)} = \frac{\alpha_0 + S_t^{(k)}}{\alpha_0 + \beta_0 + S_t^{(k)} + F_t^{(k)}}, \tag{6}$$

where c_0 is the initial confidence judged by the base model, α_0, β_0 are weak prior pseudo-counts derived from c_0 and a prior strength $n_0 = \alpha_0 + \beta_0 > 0$ (see Sec. A.3 for the prompt template). Here, θ_c is the confidence threshold and θ_{age} is the age threshold. A volatile rule is promoted when:

$$\mathcal{E}_{t+1}^{\text{stable}} = \mathcal{U}^{\text{stable}}\left(\mathcal{E}_t^{\text{stable}}, \Delta\mathcal{E}_t^{\text{stable}}\right) \cup \left\{ r^{(k)} \in \mathcal{E}_t^{\text{volatile}} \,\middle|\, \text{Promote}(r_t^{(k)}) = 1 \right\}, \tag{7}$$

where $\Delta\mathcal{E}_t^{\text{stable}}$ contains only Merge and Delete operations, triggered only after repeated cross-subset underperformance.

$$\mathcal{E}_{t+1}^{\text{volatile}} = \mathcal{U}^{\text{volatile}}\left(\mathcal{E}_t^{\text{volatile}} \setminus \left\{ r^{(k)} \,\middle|\, \text{Promote}(r_t^{(k)}) = 1 \right\}, \Delta\mathcal{E}_t^{\text{volatile}}\right). \tag{8}$$

Here, $\mathcal{U}^{\text{volatile}}$ simply applies $\Delta\mathcal{E}_t^{\text{volatile}}$ to the remaining volatile rules. Promotion works in only one direction: a bad round can affect the volatile tier, but it can never overwrite stable memory.

3.3 INFERENCE WITH VOLATILE AND STABLE MEMORY

At inference time, the harness retrieves rules from both tiers and uses them jointly to guide the current task. Given a task query Q, we compute its embedding $\mathbf{q} = f(Q)$ with the same fixed text encoder $f(\cdot)$ that maps each rule's applicability context ϕ to an embedding $\mathbf{z} = f(\phi)$. For each rule $r \in \mathcal{E}_t$, we compute the cosine similarity

$$\text{sim}(\mathbf{q}, \mathbf{z}_r) = \frac{\mathbf{q}^\top \mathbf{z}_r}{\|\mathbf{q}\| \, \|\mathbf{z}_r\|}, \tag{9}$$

and retrieve the top-K rules separately from the stable and volatile tiers,

$$\mathcal{R}_K^{\text{stable}}(Q) = \underset{r \in \mathcal{E}_t^{\text{stable}}}{\text{TopK}} \ \text{sim}(\mathbf{q}, \mathbf{z}_r), \qquad \mathcal{R}_K^{\text{volatile}}(Q) = \underset{r \in \mathcal{E}_t^{\text{volatile}}}{\text{TopK}} \ \text{sim}(\mathbf{q}, \mathbf{z}_r). \tag{10}$$

The two sets are then merged into a candidate rule set

$$\mathcal{R}_K(Q) = \mathcal{R}_K^{\text{stable}}(Q) \cup \mathcal{R}_K^{\text{volatile}}(Q). \tag{11}$$

The base model inspects this candidate set for conflicts, where a conflict occurs when a stable rule r_s and a volatile rule r_v match the query with similar applicability contexts but prescribe different interventions. We resolve any such conflict in favor of the stable rule, so r_s takes precedence over r_v. The final rule set used for inference is therefore

$$\mathcal{R}_K^\star(Q) = \mathcal{R}_K^{\text{stable}}(Q) \cup \left\{ r_v \in \mathcal{R}_K^{\text{volatile}}(Q) \mid \text{no conflict with } \mathcal{R}_K^{\text{stable}}(Q) \right\}. \tag{12}$$

This allows the harness to benefit from consolidated knowledge in the stable tier while still considering recent, unproven edits in the volatile tier when they do not conflict with trusted rules.

4 EXPERIMENT

We demonstrate the effectiveness of EvoRule Harness by answering four key questions: (1) Can it sustain long-term harness performance under sequential task distribution shifts, where other harness optimization frameworks degrade from forgetting? (2) Can patch-level credit assignment improve edit precision and harness adaptation performance and reliability? (3) Can its stable and volatile memory isolate harmful noise, improve final rule quality, and yield interpretable rules? (4) Can these gains transfer across classification, long-horizon agentic tasks, and different base harness optimization frameworks?

Table 1: **Main results.** All methods use a frozen Claude Opus 4.6 backbone unless noted. Tau-Bench and Terminal-Bench-2 report Pass@1; SWE-bench Verified reports Resolved Rate. The final block combines ours with harness-optimization methods. **Bold**: best; underline: second-best.

Method	Type	Tau-Bench ↑	Terminal-Bench-2 ↑	SWE-bench Verified ↑	Avgerage ↑
Prompting / Static Baselines					
Zero-shot (Wei et al., 2021)	Static	25.4	42.6	38.2	35.4
Few-shot(32) (Brown et al., 2020)	Static	29.7	48.3	45.1	41.0
Minimal-harness	Static	33.3	51.2	52.7	45.7
Human-engineered baseline	Static	40.0	58.0	61.4	53.1
Context Engineering					
ACE (Zhang et al., 2026)	Context Eng.	35.2	68.9	58.6	54.2
MCE (Ye et al., 2026)	Context Eng.	37.8	71.4	63.2	57.5
Memory / Experience-Based Agents					
MemoHarness (Huang et al., 2026)	Memory	37.3	**80.6**	62.1	60.0
Reflexion (Shinn et al., 2023)	Memory	30.6	48.5	45.8	41.6
MemGPT (Packer et al., 2023)	Memory	28.1	44.2	41.3	37.9
Voyager (Wang et al., 2023)	Memory	26.3	40.7	37.6	34.9
Agent Workflow Memory (Wang et al., 2024)	Memory	32.9	52.4	49.7	45.0
Offline Harness Search					
AutoSaddler (Park et al., 2026)	Offline Search	42.1	50.0	46.9	46.3
AFlow (Zhang et al., 2025)	Offline Search	38.6	55.3	18.8	37.6
Online / Continual Harness Adaptation					
RHI (Lee et al., 2026a)	Online Adapt.	40.2	69.1	75.2	61.5
Harness-R1 (Shao et al., 2026)	Online Adapt.	38.4	66.5	68.5	57.8
Ours with Backbone Harnesses					
Continual Harness (Kang et al., 2026)	Online Adapt.	31.5	41.2	44.8	39.2
+ EvoRule Harness (Ours)		37.5 (+19.0%)	55.8 (+35.4%)	66.2 (+47.8%)	53.2 (+35.7%)
TTHE (Nie et al., 2026)	Online Adapt.	41.3	73.2	71.6	61.1
+ EvoRule Harness (Ours)		44.6 (+7.9%)	79.1 (+8.1%)	75.3 (+5.2%)	66.3 (+8.5%)
Meta-Harness (Lee et al., 2026b)	Offline Search	41.3	76.4	74.3	63.0
+ EvoRule Harness (Ours)		**45.1** (+9.1%)	80.3 (+5.1%)	**76.8** (+3.4%)	**67.4** (+7.0%)

4.1 Experimental Setups

Dataset and Benchmarks. We evaluate EvoRule Harness on online text classification (LawBench (Fei et al., 2024), S2D (Gretel AI, 2023), USPTO-50k (Schneider et al., 2016)), retrieval-augmented mathematical reasoning, and long-horizon interactive agentic tasks (Tau-Bench (Yao et al., 2024), Terminal-Bench-2 (Merrill et al., 2026), SWE-bench (Jimenez et al., 2024)). Comparisons use the heterogeneous Tau-Bench with a search, validation, and final-test split of 225, 75, and 75, frozen open-source Qwen3.8-27B (Yang et al., 2025) and closed-source Claude Code Opus 4.6, identical final-test task IDs, and each method's adaptation protocol; see Appendix.

Evaluation Metrics. We use Pass@1 on Tau-Bench, Terminal-Bench-2, and retrieval-augmented math reasoning; Resolved Rate on SWE-bench Verified; Accuracy on LawBench, S2D, USPTO-50k; Average on three agentic tasks; Cumulative Pass@1 on Terminal-Bench-2-Stream (R1, R3, R5, R7, and R10, 5 runs); also report overhead, memory quality, ablations, frozen backbone, same test IDs, 225/75/75 split, and gains over strongest baseline.

Implementation Details. We implement EvoRule Harness on different harness optimization frameworks under the same principle, without weight/gradient/task-specific training, using a disjoint stable and volatile rule memory with logged and precedent-screened edits, Beta-Bernoulli confidence calibration, atomic promotion gated by confidence and age, stable-only MERGE and DELETE updates, frozen CLIP ViT-B/32 top-K retrieval with stable-prioritized conflict resolution, and a 225/75/75 Tau-Bench split; see Sec. A.1 for all details.

4.2 Main Results

In this section, we present results for EvoRule Harness across three key dimensions: main benchmark performance, out-of-domain generalization, and long-horizon continual adaptation.

Table 2: Additional online text classification results on LawBench (legal text), S2D (symptom-to-diagnosis), and USPTO-50k (patent classification). Bold and underline indicate the best and second-best results, respectively. The shaded row denotes our proposed method.

Harness	**LawBench** (%)	**S2D** (%)	**USPTO-50k** (%)	**Avg. Acc.** (%)	**Context** (k) ↓
Zero-shot	27.7	72.7	34.0	44.8	**0.4**
Few-shot (32)	38.7	62.0	**53.3**	51.3	<u>5.2</u>
ACE	40.7	74.0	48.0	54.2	50.8
MCE	–	–	–	∼40.0	28.5
Meta-Harness (best)	<u>42.3</u>	<u>76.0</u>	51.5	<u>56.6</u>	11.4
EvoRule Harness (Ours)	**44.5**	**77.8**	<u>53.2</u>	**58.5**	11.8

Main benchmark results. Tab. 1 reports the main results on Tau-Bench, Terminal-Bench-2, and SWE-bench Verified using a frozen Claude Opus 4.6 backbone. To examine whether EvoRule Harness can serve as a general enhancer for existing harness optimization frameworks, we integrate the same EvoRule Harness module into TTHE, Continual Harness, and Meta-Harness without modifying the frozen backbone or the base framework. EvoRule Harness consistently improves all three base frameworks: the average score of TTHE rises from 61.1 to 66.3, Meta-Harness from 63.0 to 67.4, and Continual Harness from 39.2 to 53.2, demonstrating that its benefits are not tied to a specific base framework. Compared with the strong memory-based agent, Memo-Harness, EvoRule Harness-augmented Meta-Harness is more general and achieves stronger overall performance: on Tau-Bench, performance rises from 37.3 to 45.1; on SWE-bench Verified, from 62.1 to 76.8; and on average, from 60.0 to 67.4, while remaining competitive on Terminal-Bench-2 (80.3 vs. 80.6). Overall, EvoRule Harness can be integrated into diverse harness optimization frameworks and consistently improves them.

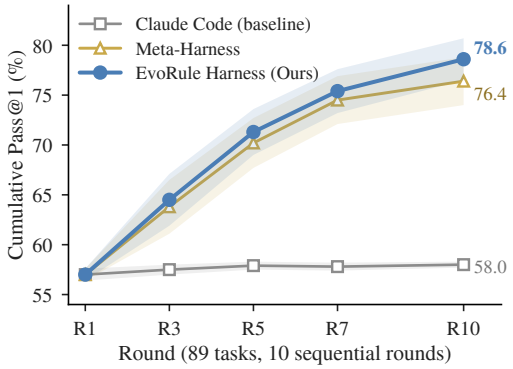


Figure 3: Cumulative Pass@1 across evaluation rounds. Shaded bands and error bars show one standard deviation over five runs.

Out-of-domain testing. Tab. 2 evaluates EvoRule Harness on three online text-classification datasets: LawBench, S2D, and USPTO-50k. EvoRule Harness improves the average accuracy to 58.5%, up from 56.6% for Meta-Harness and 54.2% for ACE, while adding only 11.8k additional context tokens. The slight context increase (+0.4k over Meta-Harness) arises from the richer applicability metadata introduced by Refine and Split atomic edits. This indicates that EvoRule Harness improves accuracy without incurring a substantial inference-context cost.

Long-horizon continual adaptation. Fig. 3 reports the cumulative Pass@1 on TerminalBench-2-Stream, where 89 tasks are partitioned into 10 sequential rounds. EvoRule Harness matches Meta-Harness at R1 (57.0) and consistently outperforms it thereafter: 64.5 vs. 63.8 at R3, 71.3 vs. 70.2 at R5, 75.4 vs. 74.5 at R7, and 78.6 vs. 76.4 at R10. The advantage widens to +2.2 points by R10, suggesting that the stable and volatile rule memory mitigates long-horizon forgetting under non-stationary task streams.

4.3 Quantitative Analysis

In this section, we quantitatively analyze the atomic edit operations and long-horizon performance of EvoRule Harness, demonstrating evidence-gated, non-regressive adaptation via stable/volatile memory. Specifically, we first examine the distribution and validation outcomes of atomic edit operations, and then analyze long-horizon performance under non-stationary task streams.

Analysis of Operation-Level Statistics. Tab. 3 reports operation-level statistics on TerminalBench-2, where READ and RETRIEVE dominate frequency (45.2/28.4 calls per round) with success 100.0%/96.2%, latency 12/18 ms, reuse 88.3%/90.1%, and conflicts 1.8%/1.1%, indicating infer-

Table 3: Statistics of EvoRule Harness + Meta Harness on TerminalBench-2 (Claude Opus 4.6). Freq., latency, token, reuse, and conflicts denote calls per round over 9 tasks, per-call latency, added context tokens (k), later reuse rate, and conflict rate with existing rules; "—" indicates not applicable (DELETE omitted; PROMOTE 0.0 after arbitration).

Operation	Freq./Round	Success (%)	Latency (ms)	Token (k)	Reuse (%)	Conflicts (%)
READ	45.2	100.0	12	0.8	88.3	1.8
WRITE	3.2	94.1	45	1.2	76.4	5.1
UPDATE	1.8	88.2	38	1.5	82.0	7.9
DELETE	0.7	82.4	22	0.5	—	—
RETRIEVE	28.4	96.2	18	2.1	90.1	1.1
PROMOTE	0.4	95.0	60	0.3	98.2	0.0

Table 4: Atomic edit operations on TerminalBench-2 (Claude Opus 4.6, Claude Code). *Usage*: fraction of proposed patches. *Success*: only for volatile-tier patches (Add, Refine, Split), as the fraction entering VALIDATED; stable-tier operations (Merge, Delete) are applied directly, with no comparable success rate (–).

Operation	Tier	Usage (%)	Success (%)	Typical Trigger Scenario
Add	volatile	28	72	Missing environment bootstrap or uncovered tool chain
Refine	volatile	30	76	Rule over-general across domains; narrow ϕ
Split	volatile	16	66	Rule covers distinct task categories (e.g., compilation vs. API calls)
Merge	stable	20	–	Two high-confidence rules with overlapping ϕ
Delete	stable	6	–	Rule fails across multiple tasks and rounds

ence relies on retrieving/applying existing rules rather than frequent rewriting, while WRITE, UPDATE, and PROMOTE are sparser (3.2, 1.8, and 0.4 calls per round, respectively) but reliable (94.1%/88.2%/95.0%), with WRITE and UPDATE having higher token overhead (1.2k and 1.5k) and conflicts (5.1% and 7.9%), PROMOTE having highest latency (60 ms), reuse (98.2%), zero conflicts after arbitration, and low token overhead (0.3k), and DELETE being rare (0.7 calls per round) with lowest success (82.4%) and not-applicable reuse/conflict, reflecting conservative deletion.

Analysis of Atomic Edit Operations. Tab. 4 further summarizes the five atomic edit operations, where Refine is most frequent (30%), Add (28%), Merge (20%), Split (16%), Delete (6%), and among volatile-tier operations Refine has highest validation success (76%), Add (72%), Split (66%), while stable-tier Merge and Delete operations are applied directly and lack comparable PENDING to VALIDATED success, together supporting the conservative stable-memory design in which new failure modes first enter the volatile tier as Add, Refine, or Split and stable knowledge is merged or deleted only after repeated cross-subset underperformance.

Analysis of Long-Horizon Performance. The round-by-round results in Fig. 3 show that EvoRule Harness's cumulative Pass@1 continues to increase as more tasks are observed, without the late-stage plateau exhibited by Meta-Harness. This aligns with the intended role of the volatile tier: new edits are isolated, validated, and only then promoted, so useful rules accumulate while harmful edits are discarded.

4.4 Ablation Studies

In this section, we ablate promotion gating, the dual-tier split, patch-level credit assignment, and the atomic operation set, finding that removing promotion gating lets unvalidated volatile edits reach stable memory and increases regression risk, removing either tier collapses to a single-tier memory that either fails to isolate new edits or protect validated knowledge, removing credit assignment weakens attribution to individual Add, Delete, Refine, Split, and Merge patches, and limiting operations to Add and Delete reduces edit granu-

Table 5: Ablation of EvoRule Harness + Meta-Harness on TerminalBench-2 (Claude Opus 4.6). Drop (pp) is relative to the full EvoRule Harness.

Configuration	Pass@1 (%)	Drop (pp)
EvoRule Harness (Full)	**80.3**	—
w/o Promotion	77.2	3.1
w/o Volatile Tier	76.8	3.5
w/o Stable Tier	76.6	3.7
w/o Credit Assignment	77.8	2.5
w/o Atomic Ops	78.9	1.4
Meta-Harness	76.4	3.9

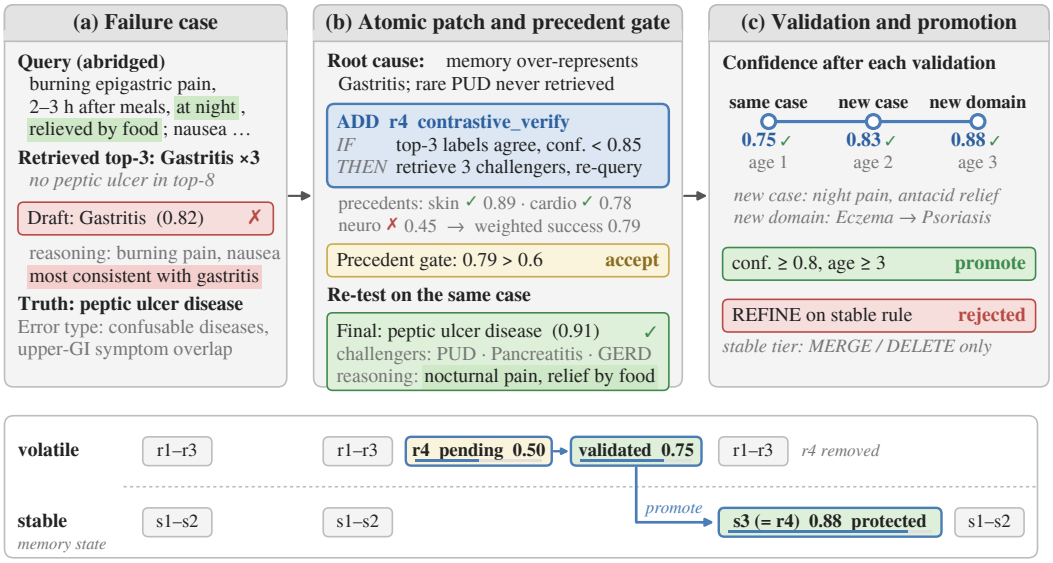


Figure 4: Case study of EvoRule Harness on the S2D dataset (symptom-to-diagnosis). An ADD patch triggered by a failure case passes the precedent gate, is validated across cases and domains, and is promoted to the stable tier, while a direct REFINE on a stable rule is rejected. This illustrates evidence-gated, non-regressive rule evolution.

larity, while as reported in Tab. 5 the full model achieves the best Pass@1 and all ablated variants degrade by 2.1–5.5 pp, with single-tier variants falling below the Meta-Harness baseline and stable-tier removal causing the largest drop, confirming these components are complementary and jointly necessary and that the stable and volatile tiers are particularly critical for non-regressive lifelong adaptation.

4.5 Case Study

In this section, we present a case study from the S2D dataset to illustrate how EvoRule Harness converts a failure into a validated, non-regressive update: a duodenal ulcer query is misclassified as Gastritis (0.82) because retrieval over-represents Gastritis and misses peptic ulcer disease, triggering an atomic ADD patch that passes the precedent gate, enters the volatile tier as PENDING, yields the correct diagnosis (0.91) on re-testing, and after validation across a new case and domain raises confidence from 0.75 to 0.83 to 0.88 with age 3, satisfying promotion criteria and being atomically promoted to the stable tier as protected memory, while a direct REFINE on a stable rule is rejected, demonstrating that EvoRule Harness isolates new hypotheses, validates them across cases and domains, and promotes them only after repeated evidence without corrupting consolidated knowledge. Overall, this case highlights the effectiveness, robustness, and trustworthiness of EvoRule Harness as a plug-and-play memory layer for reliable lifelong harness adaptation.

5 Conclusion

In this work, we presented EvoRule Harness, a plug-and-play, training-free two-tier rule architecture for lifelong harness adaptation. Traceable atomic edits first enter a volatile tier and are promoted to a stable tier only after repeated validation, enabling auditable, rollback-capable, and non-regressive adaptation. Experiments across reasoning, classification, and interactive agentic tasks show consistent gains over strong baselines and improve existing harness optimizers by up to 35.7%. In future work, we plan to broaden the atomic operation set and automatic promotion and rollback criteria, extend the framework to multimodal inputs, multi-agent collaboration, and more dynamic tool ecosystems, and integrate it with additional harness optimizers and frozen backbones. We also aim to develop theoretical guarantees and cheaper validation protocols to further reduce adaptation overhead and support reliable lifelong deployment over even longer horizons.

AI USE STATEMENT

In this work, we used generative AI tools (specifically, large language models) to assist in drafting parts of the paper text. The human authors conceived and executed all research ideas, methodological designs, experimental setups, and result analyses. All authors reviewed and verified the LLM-assisted writing for factual accuracy, logical consistency, and originality; we take full responsibility for the final content, including all claims, experiments, and artifacts presented in this paper.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, volume 2026, pp. 8479–8565, 2026.

Joss Armstrong. Margin: Runtime confidence calibration for multi-agent foundation model coordination. *arXiv preprint arXiv:2605.22949*, 2026.

Casey Barkan, Sidney Black, and Oliver Sourbut. Do large language models know what they are capable of? In *International Conference on Learning Representations*, volume 2026, pp. 15799–15822, 2026.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Hanwu Chen, Hanyu Lin, Zhanjiang Yang, Linhao Zhang, Aoyan Li, Jinxi Li, Meng Li, Yin Chen, and Daoguang Zan. Envpilot: Systematic design and evaluation of an experience-augmented agent for software environment setup. *ACM Transactions on Software Engineering and Methodology*, 2026.

Dongkyu Cho, Taesup Moon, Rumi Chunara, Kyunghyun Cho, and Sungmin Cha. Forget forgetting: Continual learning in a world of abundant memory. In *International Conference on Learning Representations*, volume 2026, pp. 14441–14466, 2026.

Jing Du and Hang Zhao. Entropic memory: A thermodynamics-inspired consolidation mechanism for lifelong agent learning. In *ICLR 2026 Workshop on Memory for LLM-Based Agentic Systems*, 2026.

Zhiwei Fei, Xiaoyu Shen, Dawei Zhu, Fengzhe Zhou, Zhuo Han, Alan Huang, Songyang Zhang, Kai Chen, Zhixin Yin, Zongwen Shen, et al. Lawbench: Benchmarking legal knowledge of large language models. In *Proceedings of the 2024 conference on empirical methods in natural language processing*, pp. 7933–7962, 2024.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, et al. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*, 2025.

Gretel AI. Symptom to diagnosis dataset. https://huggingface.co/datasets/gretelai/symptom_to_diagnosis, 2023. Accessed: 2026-01-22.

Isaac Han, Sangyeon Park, Seungwon Oh, Donghu Kim, Hojoon Lee, and Kyung-Joong Kim. Fire: Frobenius-isometry reinitialization for balancing the stability-plasticity tradeoff. *arXiv preprint arXiv:2602.08040*, 2026.

Yue Huang, Wenjie Wang, Han Bao, Yuchen Ma, Xiaonan Luo, Yi Nian, Haomin Zhuang, Zheyuan Liu, Yue Zhao, and Xiangliang Zhang. Memoharness: Agent harnesses that learn from experience. *arXiv preprint arXiv:2607.14159*, 2026.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Borui Kang, Jinrui Gu, Junhan Lv, Wenbin Li, Lei Wang, and Yang Gao. Harness continual learning: Continual adaptation beyond model parameters. *arXiv preprint arXiv:2608.19013*, 2026.

Hyunin Lee, Jinglue Xu, Jeffrey Seely, Donghyun Lee, Matei Zaharia, and Yujin Tang. Recursive harness self-improvement. *arXiv preprint arXiv:2607.15524*, 2026a.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026b.

Diego C Lerma-Torres. Human-like lifelong memory: A neuroscience-grounded architecture for infinite interaction. *arXiv preprint arXiv:2603.29023*, 2026.

Shuaiqi Liu, Zhengkai Lin, Yuxiang Zhang, Yuanyi Ren, Yue Wu, Yongbin Li, Zheng Wang, Zhihang Fu, and Jieping Ye. The path to recursive self-improving agents: Foundation, framework, and future directions. 2026.

Qianyu Meng, Yanan Wang, Liyi Chen, Yihang Li, Wei Wu, Wenyuan Jiang, Qimeng Wang, Chengqiang Lu, Yan Gao, Yi Wu, et al. Agent harness for large language model agents: A survey. 2026.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, volume 2026, pp. 40903–40986, 2026.

Qirui Mi, Zhijian Ma, Mengyue Yang, Haoxuan Li, Yisen Wang, Haifeng Zhang, and Jun Wang. Skill-pro: Learning reusable skills from experience via non-parametric ppo for llm agents. *arXiv preprint arXiv:2602.01869*, 2026.

Niloofar Mireshghallah, Neal Mangaokar, Narine Kokhlikyan, Arman Zharmagambetov, Manzil Zaheer, Saeed Mahloujifar, and Kamalika Chaudhuri. Cimemories: A compositional benchmark for contextual integrity in llms. In *International Conference on Learning Representations*, volume 2026, pp. 95513–95532, 2026.

Jun Nie, Yonggang Zhang, Jun Song, Qianshu Cai, Dahai Yu, Yike Guo, Xinmei Tian, and Bo Han. Tthe: Test-time harness evolution. *arXiv preprint arXiv:2607.08124*, 2026.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, et al. Reasoningbank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, volume 2026, pp. 94327–94354, 2026.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G Patil, Ion Stoica, and Joseph E Gonzalez. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

Linyue Pan, Lexiao Zou, Shuo Guo, Jingchen Ni, and Hai-Tao Zheng. Natural-language agent harnesses. *arXiv preprint arXiv:2603.25723*, 2026.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pp. 1–22, 2023.

Sungho Park, Wonjoong Kim, Rongyuan Tan, Jue Zhang, Wook-Shin Han, Pengfei Gao, Chanyoung Park, Yongqiang Yao, Rao Fu, Elsie Nallipogu, et al. Autosaddler: Automatic harness optimization with durable updates from agent execution traces. *arXiv preprint arXiv:2608.23041*, 2026.

Nadine Schneider, Nikolaus Stiefl, and Gregory A Landrum. What's what: The (nearly) definitive guide to reaction role assignment. *Journal of chemical information and modeling*, 56(12):2336–2346, 2016.

Biswa Sengupta and Jinhua Wang. Harbor: Automated harness optimization. *arXiv preprint arXiv:2604.20938*, 2026.

Shuai Shao, Kangning Zhang, Qingyao Li, Shijian Wang, Hao Wang, Wenxiang Jiao, Yuan Lu, Yi Guo, Weiwen Liu, and Weinan Zhang. Harness-r1: Learning to edit executable runtime harnesses from agent failure trajectories. *arXiv preprint arXiv:2608.02276*, 2026.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Varun Ursekar, Apaar Shanker, Yash Maurya, Shehab Yasser, Vijay S Kalmath, Veronica Chatrath, and Yuan Xue. Harnessopt-bench: Evaluating llms at harness optimization. *arXiv preprint arXiv:2608.06301*, 2026.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, et al. Ragen: Understanding self-evolution in llm agents via multi-turn reinforcement learning. *arXiv preprint arXiv:2504.20073*, 2025.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024.

Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*, 2021.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, volume 2024, pp. 12028–12068, 2024.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Haoran Ye, Xuning He, Vincent Arak, Haonan Dong, and Guojie Song. Meta context engineering via agentic skill evolution. *arXiv preprint arXiv:2601.21557*, 2026.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. In *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, volume 2026, pp. 86069–86100, 2026.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Zijian Zhou, Ao Qu, Zhaoxuan Wu, Sunghwan Kim, Alok Prakash, Daniela Rus, Bryan Kian Hsiang Low, and Paul Liang. Mem1: Learning to synergize memory and reasoning for efficient long-horizon agents. In *International Conference on Learning Representations*, volume 2026, pp. 58413–58438, 2026.

A APPENDIX

A.1 EXPERIMENTAL SETUP

Backbones and frozen components. All main experiments in Tab. 1 use a frozen Claude Opus 4.6 backbone unless otherwise noted, and the Terminal-Bench-2 experiments use Claude Opus 4.6 with Claude Code. The experimental setup also considers a frozen open-source Qwen3.8-27B model and a closed-source Claude Code Opus 4.6 model for the heterogeneous evaluation protocol. EvoRule Harness is a plug-and-play memory layer attached to a frozen LLM harness; it requires no weight updates, no gradients, and no task-specific training. The fixed text encoder used for retrieval is the text tower of a frozen CLIP ViT-B/32, with a 512-dimensional output, 77-token truncation, and L2 normalization. This encoder is used only for top-K retrieval and is never updated during adaptation.

Tasks, metrics, and data splits. We evaluate EvoRule Harness on online text classification, retrieval-augmented mathematical reasoning, and long-horizon interactive agentic tasks. For online text classification, we use LawBench for legal text classification, S2D for symptom-to-diagnosis classification, and USPTO-50k for patent classification; we report average accuracy and additional inference context tokens. For retrieval-augmented mathematical reasoning, we report Pass@1 averaged over five held-out models. For interactive agentic evaluation, we use Tau-Bench, Terminal-Bench-2, and SWE-bench Verified, and report Pass@1 on Tau-Bench and Terminal-Bench-2 and Resolved Rate on SWE-bench Verified. For long-horizon continual adaptation, we use Terminal-Bench-2-Stream, where 89 tasks are partitioned into 10 sequential rounds; we report cumulative Pass@1 at rounds R1, R3, R5, R7, and R10, with standard deviations over five independent runs. The heterogeneous Tau-Bench task stream uses a $225/75/75$ search/validation/final-test split. All methods use identical final-test task IDs and each method's respective adaptation protocol, so that comparisons are made on the same final evaluation tasks.

Rule memory and atomic edit operations. EvoRule Harness maintains a two-tier causal rule memory. Each rule r has five fields: an applicability context ϕ, an intervention instruction ψ, recent execution or validation feedback ω, a confidence value in $[0, 1]$, and a lifespan counting the number of successful validation rounds. The rule base is the disjoint union of a stable tier and a volatile tier,

$$\mathcal{E}_t = \mathcal{E}_t^{\text{stable}} \cup \mathcal{E}_t^{\text{volatile}}, \qquad \mathcal{E}_t^{\text{stable}} \cap \mathcal{E}_t^{\text{volatile}} = \emptyset.$$

Algorithm 1 EvoRule Harness: One Update Cycle

Require: $\mathcal{E}_t = \mathcal{E}_t^{\text{stable}} \cup \mathcal{E}_t^{\text{volatile}}$, $\mathcal{K}_t$, $\mathcal{H}_t$.
Ensure: Updated memory state $\mathcal{E}_{t+1}$.

Stage 1: Proposal, reflection, and atomic edit generation (Sec. 3.1).
Generate candidate harnesses from $\mathcal{H}_t$ and $\mathcal{K}_t$, evaluate them, and obtain judge reflections $\mathcal{J}_t$.
Convert reflections $\rho \in \mathcal{J}_t$ into atomic edits $\Delta\mathcal{E}_t$ over the operation set $\mathcal{O}$ (Eq. (1)).
Stage 2: Stable/volatile memory update (Sec. 3.2).
Application and validation. Write the resulting patches to the volatile tier $\mathcal{E}_t^{\text{volatile}}$ as PENDING; re-evaluate them on held-out validation subsets.
Resolve each patch by its validation outcome; append resolved patches to $\mathcal{M}_t$.
Stable-tier protection. Stable-tier edits are restricted to MERGE and DELETE, and are triggered only after repeated cross-subset underperformance.
Promotion and update.
for each volatile rule $r_t^{(k)} \in \mathcal{E}_t^{\text{volatile}}$ **do**
    Recalibrate $c_t^{(k)}$ by Eq. (6).
    if Promote$(r_t^{(k)}) = 1$ by Eq. (5) **then**
        Promote atomically via Eq. (7) and Eq. (8).
    end if
end for
Update the two tiers independently by Eq. (4).
return $\mathcal{E}_{t+1} = \mathcal{E}_{t+1}^{\text{stable}} \cup \mathcal{E}_{t+1}^{\text{volatile}}$.

The stable tier stores repeatedly validated rules, while the volatile tier stores new edits, unresolved patches, and hypotheses that have not yet accumulated sufficient evidence. A volatile patch never directly modifies a stable rule. Stable-tier edits are restricted to MERGE and DELETE, and are triggered only after repeated cross-subset underperformance or redundancy with a near-identical applicability context. The atomic edit space is

$$\mathcal{O} = \{\mathrm{ADD}, \mathrm{DELETE}, \mathrm{REFINE}, \mathrm{SPLIT}, \mathrm{MERGE}\}.$$

At each round t, the system produces a set of independently logged patches

$$\Delta\mathcal{E}_t = \{\delta_t^{(1)}, \ldots, \delta_t^{(M_t)}\}, \qquad M_t = |\Delta\mathcal{E}_t|, \qquad \delta_t^{(i)} \in \mathcal{O}.$$

Each patch is stored separately with its own provenance, target rule, pass/fail outcome, undo reference, and rollback metadata, so that performance changes can be attributed to individual edits.

Online validation, confidence calibration, and promotion. A new edit is first written to the volatile tier as PENDING, recording its round, failure case, reflection, and undo reference. It is immediately re-evaluated on the previously failed data point; if it passes, its status changes from PENDING to VALIDATED. Its cumulative successes $S_t^{(k)}$, failures $F_t^{(k)}$, and survival age $a_t^{(k)}$ are then tracked across subsequent validation subsets, and its confidence is recalibrated by

$$c_t^{(k)} = \frac{\alpha_0 + S_t^{(k)}}{\alpha_0 + \beta_0 + S_t^{(k)} + F_t^{(k)}},$$

where

$$\alpha_0 = n_0 c_0, \qquad \beta_0 = n_0(1 - c_0), \qquad n_0 = \alpha_0 + \beta_0 > 0,$$

and c_0 is the initial confidence judged by the base model. In the case studies, the prior strength is set to $n_0 = 2$. A volatile rule is eligible for promotion to the stable tier only when

$$\mathcal{P}_t^{\mathrm{promote}} = \left\{ r_t^{(k)} \in \mathcal{E}_t^{\mathrm{volatile}} \mid c_t^{(k)} \geq \theta_c,\ a_t^{(k)} \geq \theta_{\mathrm{age}} \right\}.$$

Promotion requires calibrated confidence above $\theta_c = 0.8$ and survival age of at least $\theta_{\mathrm{age}} = 3$; in the case studies this is instantiated as $c_t^{(k)} \geq 0.8$ and $a_t^{(k)} \geq 3$. Promotion is atomic: promoted rules are first removed from the volatile tier and then added to the stable tier, so stable memory is never directly modified by volatile edits. The volatile tier uses a lower discard threshold than the stable tier, $\theta_{\mathrm{drop}}^{\mathrm{volatile}} < \theta_{\mathrm{drop}}^{\mathrm{stable}}$, so unproven edits are discarded faster; the exact numerical values of these discard thresholds are implementation-specific.

Inference, retrieval, and conflict resolution. At inference time, given a task query Q, EvoRule Harness computes its embedding $\mathbf{q} = f(Q)$ with the same fixed text encoder used for rule applicability contexts. For each rule r, it computes the cosine similarity

$$\mathrm{sim}(\mathbf{q}, \mathbf{z}_r) = \frac{\mathbf{q}^\top \mathbf{z}_r}{\|\mathbf{q}\|\|\mathbf{z}_r\|},$$

and retrieves the top-K rules separately from the stable and volatile tiers:

$$\mathcal{R}_K^{\mathrm{stable}}(Q) = \mathrm{TopK}_{r \in \mathcal{E}_t^{\mathrm{stable}}} \mathrm{sim}(\mathbf{q}, \mathbf{z}_r), \qquad \mathcal{R}_K^{\mathrm{volatile}}(Q) = \mathrm{TopK}_{r \in \mathcal{E}_t^{\mathrm{volatile}}} \mathrm{sim}(\mathbf{q}, \mathbf{z}_r).$$

The two sets are merged into a candidate rule set

$$\mathcal{R}_K(Q) = \mathcal{R}_K^{\mathrm{stable}}(Q) \cup \mathcal{R}_K^{\mathrm{volatile}}(Q).$$

A conflict occurs when a stable rule r_s and a volatile rule r_v match the query with similar applicability contexts but prescribe different interventions. Conflicts are resolved in favor of the stable rule. The final rule set used for inference is therefore

$$\mathcal{R}_K^*(Q) = \mathcal{R}_K^{\mathrm{stable}}(Q) \cup \left\{ r_v \in \mathcal{R}_K^{\mathrm{volatile}}(Q) \mid \text{no conflict with } \mathcal{R}_K^{\mathrm{stable}}(Q) \right\}.$$

A second-stage context matcher is applied only to the small set of top-K candidates returned by the embedding-based retriever. It judges whether the current query falls within a rule's applicability context ϕ and returns a boolean match together with a confidence in $[0, 1]$. The caller treats a rule as matched only if the boolean match is true and the confidence is at least 0.7. The second-stage matcher does not decide conflicts, promotion, rollback, or memory updates.

Implementation hyperparameters and baselines. Patches are screened by a precedent-based influence estimate with threshold $\epsilon = 0.6$. Volatile-to-stable promotion uses Beta-Bernoulli calibration with confidence threshold $\theta_c = 0.8$ and age threshold $\theta_{\text{age}} = 3$. Stable-tier edits are limited to MERGE and DELETE. The fixed text encoder is the text tower of a frozen CLIP ViT-B/32, with 512-dimensional embeddings, 77-token truncation, and L2 normalization, and is used only for top-K retrieval. We compare against prompting and static baselines, context-engineering methods such as ACE and MCE, memory- or experience-based agents such as MemoHarness, Reflexion, MemGPT, Voyager, and Agent Workflow Memory, offline harness search methods such as AutoSaddler and AFlow, and online or continual harness adaptation methods such as RHI, Harness-R1, Continual Harness, TTHE, and Meta-Harness. EvoRule Harness is also integrated into Continual Harness, TTHE, and Meta-Harness without modifying the frozen backbone or the base framework, and improves the average score of Continual Harness from 39.2 to 53.2, TTHE from 61.1 to 66.3, and Meta-Harness from 63.0 to 67.4.

Ablation, operation statistics, prompts, and case studies. We ablate promotion gating, the dual-tier split, patch-level credit assignment, and the atomic operation set on Terminal-Bench-2 with Claude Opus 4.6. The full model achieves 80.3 Pass@1, while ablated variants degrade by 1.4–3.7 percentage points. The two single-tier variants reach 76.8 and 76.6, respectively, which are below the full model and close to the Meta-Harness baseline of 76.4; removing the stable tier causes the largest drop. Operation-level statistics show that REFINE is the most frequent volatile operation at 30%, followed by ADD at 28%, MERGE at 20%, SPLIT at 16%, and DELETE at 6%; among volatile-tier operations, REFINE has the highest validation success at 76%, followed by ADD at 72% and SPLIT at 66%. The appendix also defines eight prompt templates: initial confidence judgment for a new volatile rule; failure diagnosis and patch proposal; second-stage context routing and matching; conflict resolution and arbitration; stable-tier safety checking; stable-tier trigger decision; judge reflection generation; and promotion summary generation. Case studies on S2D, LawBench, USPTO-50k, and SWE-bench Verified illustrate how a failure becomes an atomic patch, enters the volatile tier as PENDING, is validated across cases and domains, and is promoted atomically to stable memory only after repeated evidence, while direct REFINE or SPLIT operations on stable rules are rejected.

A.2 EvoRule Harness: Stable-Volatile Memory Update Cycle

Algorithm Alg. 1 summarizes one update cycle of EvoRule Harness. Given the current stable/volatile rule memory, candidate harnesses, and validation data, it first generates and evaluates candidate harnesses to obtain judge reflections on failures, then converts these reflections into atomic edits over the operation set (ADD, DELETE, REFINE, SPLIT, MERGE). In the second stage, the resulting patches are written to the volatile tier as PENDING and re-evaluated on held-out validation subsets, while stable-tier edits are restricted to MERGE and DELETE and triggered only by repeated cross-subset underperformance. Finally, volatile rules whose calibrated confidence and survival age exceed the promotion thresholds are atomically promoted to the stable tier, and the two tiers are updated independently to produce the next memory state.

A.3 Prompt Template

Prompt 1: Judge Initial Confidence c_0 for a New or Modified Volatile Rule

System Prompt:
You are a rigorous AI rule-calibration expert. Your task is to judge the *initial prior confidence* c_0 of a newly created or modified causal rule that will enter the volatile tier, before any validation outcome is observed. The caller will use c_0 as the mean of a Beta prior:

$$\alpha_0 = n_0 c_0, \qquad \beta_0 = n_0(1 - c_0), \qquad n_0 = \alpha_0 + \beta_0 > 0,$$

and later recalibrate confidence as

$$c_t^{(k)} = \frac{\alpha_0 + S_t^{(k)}}{\alpha_0 + \beta_0 + S_t^{(k)} + F_t^{(k)}}.$$

You must return a JSON object with exactly two fields: `thinking` and `c0`. The caller will parse only `c0`; `thinking` is for audit only. Do not output α_0, β_0, success counts, failure counts, promotion decisions, final confidence, or any text outside the JSON object.
Every rule in memory has five fields:

1. `phi` (applicability context): when the rule should be used.
2. `psi` (intervention instruction): the action/prompt to execute when the rule fires.
3. `omega` (recent execution/validation feedback): `success` / `fail` / `pending`.
4. `confidence`: trustworthiness in $[0, 1]$.
5. `lifespan`: number of validation rounds survived successfully.

IMPORTANT CONSTRAINTS:

- This prompt is only for operations that create or modify a rule entering the volatile tier: ADD, REFINE, SPLIT, and MERGE. Do not call this prompt for DELETE; DELETE creates no new rule and therefore has no c_0.
- If SPLIT produces multiple sub-rules, judge each sub-rule separately by calling this prompt once per sub-rule.
- For MERGE, the caller must provide the source rule IDs, the merged `phi`/`psi`, and the merge trigger. Judge the merged rule as a single new volatile rule. If the merge is stable-tier-only, do not call this prompt; stable-tier updates are governed by the meta-memory rules.
- Output $c_0 \in (0, 1)$. Do not output 0 or 1. If evidence is weak or ambiguous, output $c_0 = 0.5$.
- Judge only the initial prior trustworthiness of the proposed rule. Do not decide promotion. Promotion is decided by the caller using calibrated confidence and age thresholds.
- Give a higher c_0 when: the failure or merge evidence is clear; `phi` is specific and matches the target context; `psi` is concrete, executable, and directly repairs the observed error or preserves validated behavior; and the proposed rule is not overfit to a single case.
- Give a lower c_0 when: `phi` is broad or vague; `psi` is generic; the edit appears overfit to one failure; or the edit attempts to directly REFINE/SPLIT a stable-tier rule.
- Your output must be strictly JSON-formatted for direct programmatic parsing. No extra text.

User Prompt:
`[Evidence for the proposed volatile rule]`

- User query: {{`user_query`}}
- Expected correct output: {{`expected_output`}}
- Actual wrong output: {{`actual_output`}}
- Operation: {{`operation`}} (ADD / REFINE / SPLIT / MERGE)
- Target rule ID: {{`target_rule_id`}} (use `new` for ADD)
- Source rule IDs: {{`source_rule_ids`}} (required for MERGE; otherwise `null`)
- Source reflection: {{`source_reflection`}}
- phi (applicability context): {{`phi`}}
- psi (intervention instruction): {{`psi`}}
- omega: `pending`
- lifespan: `0`

- Provenance: round {{round}}, failure case {{failure_case_id}}, undo ref {{undo_ref}}
- Merge trigger / rationale: {{merge_trigger}} (required for MERGE; otherwise null)

For ADD/REFINE/SPLIT, use the failure case as the primary evidence. For MERGE, use the merge trigger and source-rule validation statistics as the primary evidence; the failure case may be null if the merge is redundancy-driven.
Based on the provided evidence and the proposed rule, judge the initial prior confidence c_0. Output strictly in the following JSON format (no extra text):

```
{
  "thinking": "Brief evidence-based reason for c0: failure/merge
      evidence, phi specificity, psi actionability, overfitting risk
      , and operation validity.",
  "c0": 0.5
}
```

Prompt 2: Diagnose Failure & Propose Edit Patch for Two-Tier Rule Memory

System Prompt:
You are a rigorous AI rule-maintenance expert. You maintain a two-tier causal rule memory consisting of a *stable tier* and a *volatile tier*. Every rule in memory strictly follows these five fields:

1. phi (applicability context): describes when this rule should be used.
2. psi (intervention instruction): the actual prompt/action to execute when the rule fires.
3. omega (recent execution/validation feedback): success / fail / pending.
4. confidence: a float between 0 and 1 indicating the rule's trustworthiness.
5. lifespan: the number of validation rounds this rule has survived successfully.

IMPORTANT CONSTRAINTS:

- You only *propose* candidate patches. The caller validates and applies them; you never directly modify stable memory.
- A stable-tier rule MUST NOT be directly REFINEd or SPLIT. For a stable-tier rule, you may only propose MERGE or DELETE.
- Stable-tier DELETE is allowed only after repeated cross-subset underperformance. Stable-tier MERGE is allowed only when two stable rules have near-identical applicability contexts and jointly validated interventions, or when the caller explicitly marks the case as redundancy-driven.
- A volatile-tier rule may be updated by ADD, REFINE, SPLIT, MERGE, or DELETE.
- Every proposed patch must carry provenance and rollback metadata.
- If the target rule is stable and the provided evidence does not satisfy the stable-tier constraints above, output "operation": "NoOp" and explain why no safe patch is proposed.
- Your output must be strictly JSON-formatted for direct programmatic execution. No extra text.

User Prompt:
[Current failure case]

- User query: {{user_query}}
- Expected correct output: {{expected_output}}
- Actual wrong output from the model: {{actual_output}}

[Existing matched rule (fill all fields with null if no rule matched)]

- Rule ID: {{rule_id}}
- Tier (stable or volatile): {{tier}}
- phi (applicability context): {{phi}}
- psi (intervention instruction): {{psi}}
- omega (recent feedback): {{omega}}
- confidence: {{confidence}}
- lifespan (survived rounds): {{lifespan}}
- Validation statistics: successes {{success_count}}, failures {{failure_count}}, cross-subset failures {{cross_subset_failures}}

[Candidate merge partner (required only for Merge; otherwise null)]

- Merge partner rule ID: {{merge_with_rule_id}}
- Merge partner tier: {{merge_with_tier}}
- Merge partner phi: {{merge_with_phi}}
- Merge partner psi: {{merge_with_psi}}
- Merge partner confidence: {{merge_with_confidence}}
- Merge partner lifespan: {{merge_with_lifespan}}

[Provenance]

- Round: {{round}}
- Failure case ID: {{failure_case_id}}
- Undo reference: {{undo_ref}}
- Patch ID: {{patch_id}}

Based on the failure and the existing rule, propose a safe edit patch for the two-tier memory. Output strictly in the following JSON format (no extra text):

```
{
  "thinking": "Brief reason why the old rule failed or why a new
      rule is needed.",
  "operation": "Add / Refine / Split / Delete / Merge / NoOp",
  "target_rule_id": "ID of the rule to modify (use 'new' for Add)",
  "target_tier": "stable / volatile / new",
  "merge_with_rule_id": "ID of the merge partner, or null",
  "new_rule": {
    "phi": "New or modified applicability context (set to null for
        Delete/NoOp)",
    "psi": "New or modified intervention instruction (set to null
        for Delete/NoOp)",
    "omega": "pending",
    "confidence": null,
    "lifespan": 0
  },
  "split_rules": [
    {
      "phi": "Sub-rule applicability context",
      "psi": "Sub-rule intervention instruction",
      "omega": "pending",
      "confidence": null,
      "lifespan": 0
    }
  ],
```

```
  "provenance": {
    "round": "{{round}}",
    "failure_case_id": "{{failure_case_id}}",
    "undo_ref": "{{undo_ref}}",
    "patch_id": "{{patch_id}}"
  },
  "reason": "Short justification for the operation and the safety
      check."
}
```

Prompt 3: Context Routing & Matching (Second-Stage Matcher)

System Prompt:
You are a context matcher for a two-tier causal rule memory. You are called only for a small set of top-K candidate rules returned by the embedding-based retriever in Eq. 10–11; you do not scan the full rule base, and you do not replace that retriever. Your task is to judge whether the current user query falls within a rule's applicability context `phi`. This is a semantic applicability judgment, not an exact string match. Return a raw boolean `is_match` and a confidence in $[0, 1]$. The caller will treat the rule as matched only if `is_match` is true and `confidence` ≥ 0.7. Do not decide conflicts, promotion, rollback, or memory updates. Output JSON only.
User Prompt:
`[Candidate rule]`

- Rule ID: {{`rule_id`}}
- Tier: {{`tier`}} (`stable` / `volatile`)
- phi (applicability context): {{`phi`}}

`[Current user query]`
{{`user_query`}}
Decide whether the query falls within the above `phi`. Output only the following JSON (no extra text):

```
{
  "is_match": true,
  "confidence": 0.95,
  "reason": "One-sentence explanation of why the query does or does
      not fall within phi."
}
```

Prompt 4: Conflict Resolution & Arbitration for Two-Tier Rule Memory

System Prompt:
You are a decision arbitrator for a two-tier causal rule memory. Two candidate patches have been proposed for the same target rule, or for overlapping applicability contexts, and both have already passed the precedent-based influence screening. Your job is to resolve the conflict while preserving stable-tier protection, conciseness, and consistency. You do not validate, promote, execute, or directly modify memory; the caller does. Prefer, in order:

1. stable-tier knowledge over volatile-tier edits;
2. higher calibrated confidence and longer lifespan;
3. more specific and well-scoped `phi`;
4. more concrete, executable, and directly relevant `psi`.

If the conflict cannot be resolved safely from the given evidence, return `DEFER_TO_CALLER` or `REJECT_BOTH`. If a stable-tier rule is involved, you may

only return KEEP_A, KEEP_B, MERGE, REJECT_BOTH, or DEFER_TO_CALLER; you must not propose any Refine/Split-style modification of a stable rule. Output JSON only.

User Prompt:

A conflict has been detected on target rule {{target_rule_id}} (tier: {{target_tier}}). Conflict type: {{conflict_type}}. Both candidates below have passed precedent-based influence screening.

[Candidate A]

- Operation: {{operation_a}} (ADD / REFINE / SPLIT / MERGE / DELETE)
- Patch ID: {{patch_id_a}}
- Source reflection: {{source_reflection_a}}
- phi (applicability): {{phi_a}}
- psi (instruction): {{psi_a}}
- omega (feedback): {{omega_a}}
- confidence: {{confidence_a}}
- lifespan: {{lifespan_a}}
- Validation statistics: successes {{success_a}}, failures {{failure_a}}, cross-subset failures {{cross_subset_failures_a}}

[Candidate B]

- Operation: {{operation_b}} (ADD / REFINE / SPLIT / MERGE / DELETE)
- Patch ID: {{patch_id_b}}
- Source reflection: {{source_reflection_b}}
- phi (applicability): {{phi_b}}
- psi (instruction): {{psi_b}}
- omega (feedback): {{omega_b}}
- confidence: {{confidence_b}}
- lifespan: {{lifespan_b}}
- Validation statistics: successes {{success_b}}, failures {{failure_b}}, cross-subset failures {{cross_subset_failures_b}}

[Stable-tier constraint check, if applicable]

- Repeated cross-subset underperformance: {{stable_underperformance}}
- Redundancy with near-identical phi: {{stable_redundancy}}
- Jointly validated interventions: {{joint_validation}}

Output your arbitration result as JSON only:

```
{
  "decision": "KEEP_A / KEEP_B / MERGE / REJECT_BOTH /
      DEFER_TO_CALLER",
  "chosen_rule_id": "ID of the chosen rule, or null",
  "merged_rule": {
    "phi": "If MERGE, write the merged applicability; otherwise null
        ",
    "psi": "If MERGE, write the merged instruction; otherwise null",
    "omega": "If MERGE into volatile, set to 'pending'; if MERGE
        into stable and both sources are validated success, set to '
        success'; otherwise null",
    "confidence": null,
    "lifespan": "If MERGE, take the larger of the two lifespans;
        otherwise null"
  },
  "merged_confidence_policy": "caller_computes /
      weighted_by_lifespan / max / min / null",
```

```
  "provenance": {
    "target_rule_id": "{{target_rule_id}}",
    "target_tier": "{{target_tier}}",
    "patch_ids": ["{{patch_id_a}}", "{{patch_id_b}}"]
  },
  "reason": "Short justification for the decision and the safety
      check."
}
```

Prompt 5: Merge Candidate Detection / Redundancy Detection

System Prompt:
You are a redundancy detector for a two-tier causal rule memory. Your task is to identify pairs or groups of rules whose applicability contexts `phi` are near-identical and whose interventions `psi` can be conservatively merged without losing validated behavior. You do not modify memory; you only propose merge candidates to the caller. A merge is allowed only when:

- both rules are in the volatile tier, or
- both rules are in the stable tier and have jointly validated interventions, or
- the caller explicitly marks the case as redundancy-driven.

Do not propose merging a stable rule with a volatile rule. Do not propose merging rules with conflicting interventions. Output JSON only.
User Prompt:
`[Candidate rules]`

- Rule ID: {{`rule_id_1`}}, Tier: {{`tier_1`}}, phi: {{`phi_1`}}, psi: {{`psi_1`}}, confidence: {{`conf_1`}}, lifespan: {{`life_1`}}
- Rule ID: {{`rule_id_2`}}, Tier: {{`tier_2`}}, phi: {{`phi_2`}}, psi: {{`psi_2`}}, confidence: {{`conf_2`}}, lifespan: {{`life_2`}}
- Rule ID: {{`rule_id_3`}}, Tier: {{`tier_3`}}, phi: {{`phi_3`}}, psi: {{`psi_3`}}, confidence: {{`conf_3`}}, lifespan: {{`life_3`}}

`[Optional context]`: overlap statistics, co-activation counts, validation outcomes. Identify merge candidates. Output only the following JSON:

```
{
  "merge_candidates": [
    {
      "source_rule_ids": ["rule_id_1", "rule_id_2"],
      "tier": "stable / volatile",
      "overlap_score": 0.95,
      "merged_phi": "Merged applicability context",
      "merged_psi": "Merged intervention instruction",
      "merge_rationale": "Short reason why these rules can be merged
          .",
      "safety_check": "stable-tier constraint satisfied / volatile-
          only / redundancy-driven"
    }
  ],
  "reject_pairs": [
    {
      "rule_ids": ["rule_id_1", "rule_id_3"],
      "reason": "Why they cannot be merged."
    }
  ]
}
```

Prompt 6: Stable-tier Delete / Merge Trigger Check

System Prompt:
You are a safety checker for stable-tier memory. Stable-tier rules may only be modified by MERGE or DELETE, and only when the provided validation statistics show repeated cross-subset underperformance or redundancy with a near-identical applicability context. You do not execute any operation; you only decide whether the trigger conditions are met. If they are not met, output `NO_ACTION`. Output JSON only.
User Prompt:
`[Target stable rule]`

- Rule ID: {{`rule_id`}}
- phi: {{`phi`}}
- psi: {{`psi`}}
- confidence: {{`confidence`}}
- lifespan: {{`lifespan`}}
- Validation statistics: successes {{`success_count`}}, failures {{`failure_count`}}, cross-subset failures {{`cross_subset_failures`}}

`[Candidate merge partner, if any]`

- Rule ID: {{`merge_with_rule_id`}}
- phi: {{`merge_with_phi`}}
- psi: {{`merge_with_psi`}}
- Jointly validated interventions: {{`joint_validation`}}

Decide whether a stable-tier MERGE or DELETE is triggered. Output only the following JSON:

```
{
  "trigger": "MERGE / DELETE / NO_ACTION",
  "merge_with_rule_id": "ID of the merge partner, or null",
  "reason": "Short justification based on repeated cross-subset
      underperformance or redundancy.",
  "safety_check": "stable-tier constraints satisfied / not
      satisfied"
}
```

Prompt 7: Judge Reflection Generation

System Prompt:
You are a failure judge for a frozen LLM harness. Given a failure case and the current harness state, produce a concise diagnostic reflection that identifies the likely root cause and suggests a repair direction. Do not generate executable patches; the proposer will convert your reflection into atomic edits. Focus on why the current rule or harness component failed, which applicability context was mishandled, and what kind of intervention might help. Output JSON only.
User Prompt:
`[Current failure case]`

- User query: {{`user_query`}}
- Expected correct output: {{`expected_output`}}
- Actual wrong output: {{`actual_output`}}

`[Matched rule, if any]`

- Rule ID: {{`rule_id`}}
- Tier: {{`tier`}}

- phi: {{phi}}
- psi: {{psi}}
- omega: {{omega}}

Output a diagnostic reflection in the following JSON format:

```
{
  "root_cause": "Brief diagnosis of why the failure occurred.",
  "affected_rule_id": "ID of the rule involved, or null",
  "suggested_repair": "High-level direction for a repair, e.g.,
      narrow phi, split rule, add new rule.",
  "evidence": "One-sentence reference to the failure case."
}
```

Prompt 8: Promotion Summary Generation

System Prompt:
You are a documentation writer for a two-tier causal rule memory. A volatile rule has satisfied the caller's promotion criteria and is about to be promoted to the stable tier. The promotion decision has already been made by the caller; your output is only an audit record. Your task is to write one short, objective, audit-friendly sentence summarising why the rule is ready for promotion. Base the summary only on the provided rule fields and validation information. Do not decide promotion, do not change `phi` or `psi`, do not mention numeric values or thresholds, and do not claim generalisation or reliability beyond what the evidence supports. Focus on qualitative stability, repeated validation, and applicability scope.
User Prompt:
`[Rule to be promoted]`

- Rule ID: {{rule_id}}
- phi (applicability context): {{phi}}
- psi (intervention instruction): {{psi}}
- omega (recent validation outcome): {{omega}} (`success` / `fail` / `pending`; for promotion it should be `success`)
- confidence (calibrated, above promotion threshold): {{confidence}}
- lifespan (number of successful validation rounds, above threshold): {{lifespan}}

Output only one plain-text promotion summary of at most 20 words. Do not include quotes, JSON, bullet points, line breaks, markdown, or any extra text.
Example output (format only; do not copy verbatim):

```
The rule has been repeatedly validated and is ready for stable-tier
    promotion.
```

A.4 Prompt Template Analysis

Prompt 1: Judge Initial Confidence c_0 for a New or Modified Volatile Rule. The first template is used before any validation outcome is observed. It asks the model to act as a rigorous rule calibration expert and to judge the initial prior confidence c_0 of a newly created or modified causal rule that will enter the volatile tier. The caller converts c_0 into Beta prior pseudo counts via $\alpha_0 = n_0 c_0$ and $\beta_0 = n_0(1 - c_0)$, with $n_0 = \alpha_0 + \beta_0 > 0$, and later recalibrates confidence by $c_t^{(k)} = (\alpha_0 + S_t^{(k)})/(\alpha_0 + \beta_0 + S_t^{(k)} + F_t^{(k)})$. Every rule has five fields: `phi`, `psi`, `omega`, `confidence`, and `lifespan`. This template applies only to operations that create or modify a rule entering the volatile tier: ADD, REFINE, SPLIT, and MERGE. DELETE is excluded. For SPLIT, each subrule is judged separately; for MERGE, the caller provides source rule IDs, merged `phi` and `psi`, and the merge trigger. The output must be strict JSON with `thinking` and `c0`.

The value c_0 must lie in $(0, 1)$, not 0 or 1, and weak or ambiguous evidence should yield 0.5. The template does not decide promotion. Higher c_0 is given when failure or merge evidence is clear, `phi` is specific, `psi` is concrete and executable, and the edit is not overfit to a single case. Lower c_0 is given when `phi` is broad or vague, `psi` is generic, the edit appears overfit, or the edit tries to directly REFINE or SPLIT a stable tier rule.

Prompt 2: Diagnose Failure and Propose Edit Patch for Two Tier Rule Memory. The second template is the main patch proposal prompt. It asks the model to act as a rigorous rule maintenance expert for a two tier causal rule memory with a stable tier and a volatile tier. Each rule still follows the five fields `phi`, `psi`, `omega`, `confidence`, and `lifespan`. The key constraint is that the model only proposes candidate patches. The caller validates and applies them, and the model never directly modifies stable memory. A stable tier rule must not be directly REFINED or SPLIT. For a stable tier rule, the model may only propose MERGE or DELETE. The user prompt provides the failure case, the matched rule if any, and the evidence. The output is JSON with fields such as `operation`, `target_rule_id`, `target_tier`, `merge_with_rule_id`, `new_rule`, `split_rules`, `provenance`, and `reason`. This template drives the atomic edit generation over ADD, DELETE, REFINE, SPLIT, and MERGE and keeps every proposed update individually traceable.

Prompt 3: Context Routing and Matching as a Second Stage Matcher. The third template is used only after the embedding based retriever returns a small set of top K candidate rules. It asks the model to act as a context matcher for the two tier causal rule memory. The matcher does not scan the full rule base and does not replace the embedding retriever. Its task is to judge whether the current user query falls within a rule applicability context `phi`. This is a semantic applicability judgment, not an exact string match. The output is raw JSON with a boolean `is_match` and a confidence in $[0, 1]$. The caller treats the rule as matched only if `is_match` is true and confidence is at least 0.7. The matcher does not decide conflicts, promotion, rollback, or memory updates. The user prompt gives the candidate rule ID, tier, `phi`, and the current user query. The output contains `is_match`, confidence, and a one sentence reason. This prompt supports the inference stage by routing only relevant rules into the final candidate set.

Prompt 4: Conflict Resolution and Arbitration for Two Tier Rule Memory. The fourth template is the conflict arbitration prompt. It is called when two candidate patches have been proposed for the same target rule or for overlapping applicability contexts, and both have already passed precedent based influence screening. The model acts as a decision arbitrator for the two tier causal rule memory. It must resolve the conflict while preserving stable tier protection, conciseness, and consistency. It does not validate, promote, execute, or directly modify memory; the caller does. The preference order is: stable tier knowledge over volatile tier edits; higher calibrated confidence and longer lifespan; more specific and well scoped `phi`; and more concrete, executable, directly relevant `psi`. If the conflict cannot be resolved safely from the given evidence, the model returns `DEFER_TO_CALLER` or `REJECT_BOTH`. If a stable tier rule is involved, the model may only return `KEEP_A`, `KEEP_B`, MERGE, `REJECT_BOTH`, or `DEFER_TO_CALLER`; it must not propose any REFINE or SPLIT style modification of a stable rule. The user prompt includes conflict type, candidate A and B operations, patch ID, source reflection, `phi`, `psi`, `omega`, confidence, lifespan, validation statistics, and stable tier constraint checks. The output is JSON only, and may include `merged_phi`, `merged_psi`, `merge_rationale`, `safety_check`, and `reject_pairs`.

Prompt 5: Stable Tier Safety Checker System Prompt. The fifth template is the stable tier safety checker. It asks the model to act as a safety checker for stable tier memory. Stable tier rules may only be modified by MERGE or DELETE, and only when the provided validation statistics show repeated cross subset underperformance or redundancy with a near identical applicability context. The checker does not execute any operation. It only decides whether the trigger conditions are met. If they are not met, it outputs `NO_ACTION`. The output is JSON only. This template enforces the central method constraint that consolidated stable knowledge is never directly modified by volatile edits and can be changed only under strong, repeated evidence.

Prompt 6: Stable Tier Trigger Decision and Safety Output. The sixth template is the user side decision prompt for stable tier safety checking. The user prompt contains the target stable rule ID, `phi`, `psi`, confidence, lifespan, validation statistics with successes, failures, and cross subset failures. If a merge partner is proposed, the prompt also includes the partner rule ID, `phi`, `psi`, and jointly validated interventions. The model decides whether a stable tier MERGE or DELETE is triggered. The output is JSON with `trigger` set to MERGE, DELETE, or `NO_ACTION`, together with `merge_with_rule_id`, a short reason, and a safety check field. This template directly implements the paper's stable tier protection rule: stable memory is merged or deleted only after repeated

cross subset underperformance or redundancy, never from a single bad round.

Prompt 7: Judge Reflection Generation. The seventh template is the failure judge. It asks the model to act as a failure judge for a frozen LLM harness. Given a failure case and the current harness state, the model produces a concise diagnostic reflection that identifies the likely root cause and suggests a repair direction. It does not generate executable patches. The proposer later converts the reflection into atomic edits. The reflection focuses on why the current rule or harness component failed, which applicability context was mishandled, and what kind of intervention might help. The user prompt includes the current failure case with user query, expected correct output, and actual wrong output, plus the matched rule if any, with rule ID, tier, `phi`, `psi`, and `omega`. The output is JSON only and contains `root_cause`, `affected_rule_id`, `suggested_repair`, and evidence. This template is the diagnostic front end of the atomic edit pipeline.

Prompt 8: Promotion Summary Generation. The eighth template is the promotion summary generator. It asks the model to act as a documentation writer for the two tier causal rule memory. A volatile rule has satisfied the caller's promotion criteria and is about to be promoted to the stable tier. The promotion decision has already been made by the caller, so the output is only an audit record. The task is to write one short, objective, audit friendly sentence summarizing why the rule is ready for promotion. The summary must be based only on the provided rule fields and validation information. The model must not decide promotion, must not change `phi` or `psi`, must not mention numeric values or thresholds, and must not claim generalization or reliability beyond what the evidence supports. The focus is qualitative stability, repeated validation, and applicability scope. The user prompt includes rule ID, `phi`, `psi`, `omega`, calibrated confidence, and lifespan. The output is at most 20 words of plain text, with no quotes, JSON, bullet points, line breaks, markdown, or extra text.

B CASE STUDY

Round 2: LawBench — Full EvoRule Harness Evolution

Dataset: LawBench (legal text classification)

(A) FAILURE CASE

Input Sample:

```
Employment agreement clause:
"The employee agrees that, for two years after leaving the company,
they will not engage in any competing business similar to the
    company's
main business within the territory. The company shall pay monthly
non-compete compensation. If the employee breaches this clause,
the employee shall return all compensation paid and pay liquidated
    damages."
Question: Which legal clause category does this belong to?
```

Retrieval Results Summary:

- Retrieved labels: [Confidentiality, Confidentiality, Labor Contract Termination, IP Ownership, Service Period, Liquidated Damages, General Breach, Labor Dispute]
- Note: No "Non-compete" present in top-8 results.

LLM Raw Output:

```
{
  "prediction": "Confidentiality",
  "confidence": 0.79,
  "reasoning": "The clause involves post-departure obligations and
      liquidated damages, and is often listed alongside
      confidentiality duties, so it is more likely a confidentiality
       clause."
}
```

Ground Truth Label: `Non-compete`
Error Record:

```
{
  "sample_id": "lawbench-1027",
  "error_type": "confusable_clauses",
  "error_subtype": "noncompete_vs_confidentiality",
  "prediction": "Confidentiality",
  "expected": "Non-compete"
}
```


(B) ATOMIC PATCH AND PRECEDENT GATE

Judge Reflection:


```
{
  "root_cause": "Retrieval over-represents confidentiality clauses
     and misses non-compete patterns; the harness lacks a rule
     distinguishing non-compete compensation and territory
     restrictions from confidentiality obligations.",
  "affected_rule_id": null,
  "suggested_repair": "Add a new rule that detects non-compete
     clauses by the presence of post-departure competition
     restrictions, monthly compensation, and territory scope.",
  "evidence": "The clause explicitly mentions two-year post-
     departure competition restriction and monthly non-compete
     compensation."
}
```


Atomic Patch (ADD):


```
{
  "operation": "ADD",
  "target_rule_id": "new",
  "target_tier": "volatile",
  "new_rule": {
    "phi": "Employment agreement clause involving post-departure
       restriction on competing business, with monthly non-compete
       compensation and defined territory/duration.",
    "psi": "Classify as Non-compete. Retrieve non-compete precedents
       and check for compensation, duration, and territory. Do not
       classify as Confidentiality unless trade secrets are
       explicitly the subject.",
    "omega": "pending",
    "confidence": null,
    "lifespan": 0
  },
  "provenance": {
    "round": 2,
    "failure_case_id": "lawbench-1027",
    "undo_ref": "undo-lawbench-1027-01",
    "patch_id": "patch-lawbench-1027-01"
  },
  "reason": "The clause is a classic non-compete provision; the
     existing rules overfit to confidentiality because both involve
      post-departure obligations."
}
```


Initial Confidence c_0: 0.72 (judged by base model)
Precedent-Based Influence Screening:

- Precedents: similar clause types with non-compete compensation $\rightarrow$ success rate 0.81; confidentiality clauses with trade secret language $\rightarrow$ success rate 0.23.
- Influence score: 0.74.
- Gate threshold: $\epsilon = 0.6$. **Accepted.**

Re-test on the Same Failure Case:

```
{
  "prediction": "Non-compete",
  "confidence": 0.88,
  "reasoning": "The clause includes a two-year post-departure
      competition restriction, monthly non-compete compensation, and
       territory scope, which match non-compete rather than
      confidentiality."
}
```

Status: `PENDING` → `VALIDATED` in volatile tier.

(C) Validation and Promotion

Cross-Subset Validation:

- New case 1 (non-compete, different industry): **success**
- New case 2 (confidentiality, no non-compete compensation): **success** (correctly not matched)
- New case 3 (labor contract termination, no post-departure restriction): **success** (correctly not matched)
- New domain (labor dispute arbitration): **success**

Confidence Recalibration (Beta-Bernoulli):

$$\alpha_0 = n_0 c_0 = 2 \times 0.72 = 1.44, \quad \beta_0 = n_0(1 - c_0) = 2 \times 0.28 = 0.56$$

$$S = 4, \quad F = 0$$

$$c_t = \frac{\alpha_0 + S}{\alpha_0 + \beta_0 + S + F} = \frac{1.44 + 4}{1.44 + 0.56 + 4 + 0} = \frac{5.44}{6.00} = 0.907$$

Age $a_t = 3$ (validated across three subsequent rounds/subsets).

Promotion Decision:

$$c_t = 0.907 \geq \theta_c = 0.8, \qquad a_t = 3 \geq \theta_{\text{age}} = 3$$

Promoted atomically to stable tier.

Stable-Tier Protection Check: A separate proposal to directly `REFINE` an existing stable rule was rejected because stable-tier rules may only be modified by `MERGE` or `DELETE`. The new `ADD` patch is promoted as a new stable rule, not as a modification of an existing stable rule.

Outcome: The harness now reliably classifies non-compete clauses, while preserving the existing confidentiality rules. Retrieval at inference retrieves the new stable rule for similar queries; conflicts with volatile rules are resolved in favor of the stable rule.

Case 1: USPTO-50k — Full EvoRule Harness Evolution

Dataset: USPTO-50k (patent reaction classification)

(A) Failure Case

Input Sample:

```
Patent reaction description:
Substrate A (2-aminophenol) reacts with aldehyde B in ethanol under
    reflux.
After cyclization and dehydration, benzoxazole C is obtained.
The transformation forms an oxazole ring via condensation.
Question: Which reaction type should this patent reaction be
   classified as?
```

Retrieval Results Summary:

- Retrieved labels: [amide coupling, esterification, alkylation, protection, deprotection, oxidation, reduction, substitution]
- Note: No "heterocycle formation" present in top-8 results.

LLM Raw Output:

```
{
```

```
  "prediction": "amide coupling",
  "confidence": 0.76,
  "reasoning": "An amino group condenses with a carbonyl component
     and dehydration occurs, matching the typical pattern of amide
     coupling."
}
```


Ground Truth Label: `heterocycle formation`
Error Record:


```
{
  "sample_id": "uspto-50k-31842",
  "error_type": "confusable_reaction_types",
  "error_subtype": "condensation_vs_amide_coupling",
  "prediction": "amide coupling",
  "expected": "heterocycle formation"
}
```


(B) ATOMIC PATCH AND PRECEDENT GATE

Judge Reflection:


```
{
  "root_cause": "The existing rule for amino-carbonyl condensation
     is too broad and always fires as amide coupling; it does not
     check whether the condensation leads to a heterocyclic ring (e
     .g., benzoxazole) after cyclization and dehydration.",
  "affected_rule_id": "rule-amide-012",
  "suggested_repair": "Refine the broad amide coupling rule to
     exclude cases where cyclization and dehydration form a
     heterocycle, and add a dedicated rule for heterocycle
     formation.",
  "evidence": "The reaction forms an oxazole ring via condensation,
      not an amide bond."
}
```


Atomic Patch 1 (REFINE on volatile rule):


```
{
  "operation": "REFINE",
  "target_rule_id": "rule-amide-012",
  "target_tier": "volatile",
  "new_rule": {
    "phi": "Amino-carbonyl condensation where the product is a
       simple amide and no cyclization/dehydration to a heterocycle
        is described.",
    "psi": "Classify as amide coupling only when the product is an
       acyclic amide; do not fire when the reaction description
       includes cyclization and dehydration to an oxazole/
       benzoxazole ring.",
    "omega": "pending",
    "confidence": null,
    "lifespan": 0
  },
  "provenance": {
    "round": 3,
    "failure_case_id": "uspto-50k-31842",
    "undo_ref": "undo-uspto-31842-01",
    "patch_id": "patch-uspto-31842-01"
  },
  "reason": "Narrow the existing broad rule to avoid false
     positives on heterocycle-forming condensations."
}
```


Atomic Patch 2 (ADD):

```
{
  "operation": "ADD",
  "target_rule_id": "new",
  "target_tier": "volatile",
  "new_rule": {
    "phi": "Reaction of 2-aminophenol (or similar ortho-aminophenol)
        with an aldehyde in ethanol under reflux, followed by
       cyclization and dehydration to form a benzoxazole/oxazole
       ring.",
    "psi": "Classify as heterocycle formation. Retrieve heterocycle
       precedents and check for ring formation via condensation. Do
        not classify as amide coupling.",
    "omega": "pending",
    "confidence": null,
    "lifespan": 0
  },
  "provenance": {
    "round": 3,
    "failure_case_id": "uspto-50k-31842",
    "undo_ref": "undo-uspto-31842-02",
    "patch_id": "patch-uspto-31842-02"
  },
  "reason": "Add a dedicated rule for heterocycle formation to
     cover this uncovered pattern."
}
```


Initial Confidence c_0: 0.78 for both patches.
Precedent-Based Influence Screening:

- Precedents for REFINE: similar over-general rules narrowed on specific subsets → success rate 0.79.
- Precedents for ADD: heterocycle-forming condensations → success rate 0.83.
- Influence scores: REFINE = 0.76, ADD = 0.81. Both > 0.6. **Accepted.**

Re-test on the Same Failure Case:


```
{
  "prediction": "heterocycle formation",
  "confidence": 0.91,
  "reasoning": "The reaction forms a benzoxazole ring via
     cyclization and dehydration, which is a heterocycle formation,
      not a simple amide coupling."
}
```


Both patches: `PENDING` → `VALIDATED` in volatile tier.

(C) Validation and Promotion

Cross-Subset Validation:

- New case 1 (benzoxazole synthesis from another aminophenol): **success**
- New case 2 (true amide coupling, no cyclization): **success** (REFINE correctly does not fire)
- New case 3 (oxazole formation from different substrates): **success**
- New case 4 (amide coupling with cyclization side product but main product amide): **success** (ADD correctly does not fire)

Confidence Recalibration: For the ADD rule:

$$\alpha_0 = 2 \times 0.78 = 1.56, \quad \beta_0 = 2 \times 0.22 = 0.44$$

$$S = 4, \quad F = 0$$

$$c_t = \frac{1.56 + 4}{1.56 + 0.44 + 4 + 0} = \frac{5.56}{6.00} = 0.927$$

Age $a_t = 3$.
For the REFINE rule:

$$\alpha_0 = 1.56, \quad \beta_0 = 0.44, \quad S = 4, \quad F = 0$$

$$c_t = 0.927, \quad a_t = 3$$

Promotion Decision: Both rules satisfy $c_t \geq 0.8$ and $a_t \geq 3$. **Both promoted atomically to stable tier.**
Stable-Tier Protection: No direct modification of existing stable rules was attempted. The REFINE was applied to a volatile rule, and the ADD created a new volatile rule; both were promoted only after validation.
Outcome: The harness now correctly distinguishes heterocycle formation from amide coupling, reducing false positives on condensation reactions that form rings.

Case 2: SWE-bench Verified — Full EvoRule Harness Evolution

Dataset: SWE-bench Verified (agentic code repair)

(A) Failure Case

Input Sample:

```
GitHub issue:
When `APP_MODE` is not set, `load_config()` raises `KeyError: '
    APP_MODE'`.
Expected: fall back to `'development'` and emit a warning.
Reproduce: `python -m app --config config.yaml`
```

Retrieval Results Summary:

- Retrieved rule candidates: [Add: set default in config.yaml, Refine: wrap CLI in try/except, Add: pytest fixture for env vars, Merge: duplicate config rules, Delete: stale env rule]
- Note: No retrieved rule localizes the fix to `app/config/parser.py` or uses `os.environ.get`.

LLM Raw Output:

```
{
  "prediction": "Add APP_MODE: development to config.yaml",
  "confidence": 0.74,
  "reasoning": "The error is caused by a missing APP_MODE, so
      adding a default configuration value should avoid the KeyError
      ."
}
```

Ground Truth Label: `app/config/parser.py: use os.environ.get('APP_MODE', 'development') and warn on fallback`
Error Record:

```
{
  "sample_id": "swe-verified-1043",
  "error_type": "localization_error",
  "error_subtype": "wrong_file_edit",
  "prediction": "config.yaml default",
  "expected": "parser.py env fallback"
}
```

(B) Atomic Patch and Precedent Gate

Judge Reflection:

```
{
  "root_cause": "The harness lacks a rule that maps missing
      environment variable errors to the code that reads the
      variable; it instead defaults to editing configuration files
      .",
  "affected_rule_id": null,
  "suggested_repair": "Add a rule that localizes environment-
      variable fallback fixes to the parser/loader code, using os.
      environ.get with a default and emitting a warning.",
```

```
  "evidence": "The issue expects a fallback in code, not a default
      in config.yaml."
}
```


Atomic Patch (ADD):


```
{
  "operation": "ADD",
  "target_rule_id": "new",
  "target_tier": "volatile",
  "new_rule": {
    "phi": "Python application raises KeyError for an environment
        variable during configuration loading, and the expected
        behavior is a fallback default with a warning.",
    "psi": "Localize the fix to the configuration parser/loader (e.g
        ., app/config/parser.py). Use os.environ.get('VAR', 'default
        ') and emit a warning when falling back. Do not modify
        config.yaml unless the issue explicitly requests a default
        configuration value.",
    "omega": "pending",
    "confidence": null,
    "lifespan": 0
  },
  "provenance": {
    "round": 4,
    "failure_case_id": "swe-verified-1043",
    "undo_ref": "undo-swe-1043-01",
    "patch_id": "patch-swe-1043-01"
  },
  "reason": "The existing rules incorrectly prioritize config file
      edits; this rule enforces code-level fallback for environment
      variables."
}
```


Initial Confidence c_0: 0.80
Precedent-Based Influence Screening:

- Precedents: environment-variable fallback issues resolved in parser code → success rate 0.84; config-file default edits for similar issues → success rate 0.31.
- Influence score: 0.82.
- Gate threshold: $\epsilon = 0.6$. **Accepted.**

Re-test on the Same Failure Case:


```
{
  "prediction": "Edit app/config/parser.py: use os.environ.get('
      APP_MODE', 'development') and warn on fallback.",
  "confidence": 0.89,
  "reasoning": "The issue expects a code-level fallback, not a
      config default. The parser reads APP_MODE directly, so the fix
       belongs there."
}
```


Status: `PENDING` → `VALIDATED` in volatile tier.

(C) Validation and Promotion

Cross-Subset Validation:

- New case 1 (missing `DB_URL`, expected fallback in `db/config.py`): **success**
- New case 2 (missing `API_KEY`, expected error, not fallback): **success** (rule correctly does not fire)
- New case 3 (missing `LOG_LEVEL`, fallback to 'INFO' in `logging/setup.py`): **success**
- New case 4 (config file missing key, expected config edit): **success** (rule correctly does not fire)

Confidence Recalibration:

$$\alpha_0 = 2 \times 0.80 = 1.60, \quad \beta_0 = 2 \times 0.20 = 0.40$$

$$S = 4, \quad F = 0$$

$$c_t = \frac{1.60 + 4}{1.60 + 0.40 + 4 + 0} = \frac{5.60}{6.00} = 0.933$$

Age $a_t = 3$.
Promotion Decision:

$$c_t = 0.933 \geq 0.8, \qquad a_t = 3 \geq 3$$

Promoted atomically to stable tier.
Conflict Resolution Example: At inference, a volatile rule suggested editing `config.yaml` for a similar issue. The stable rule (newly promoted) conflicted with the volatile rule. By conflict arbitration, the stable rule took precedence, and the volatile edit was discarded. This demonstrates stable-tier protection and non-regression.
Outcome: The harness now correctly localizes environment-variable fallback fixes to parser/loader code, avoiding unnecessary configuration file edits.

Case 3: LawBench — Full EvoRule Harness Evolution

Dataset: LawBench (legal text classification)

(A) Failure Case

Input Sample:

```
Employment agreement clause:
"The employee agrees that, for two years after leaving the company,
they will not engage in any competing business similar to the
    company's
main business within the territory. The company shall pay monthly
non-compete compensation. If the employee breaches this clause,
the employee shall return all compensation paid and pay liquidated
    damages."
Question: Which legal clause category does this belong to?
```

Retrieval Results Summary:

- Retrieved labels: [Confidentiality, Confidentiality, Labor Contract Termination, IP Ownership, Service Period, Liquidated Damages, General Breach, Labor Dispute]
- Note: No "Non-compete" present in top-8 results.

LLM Raw Output:

```
{
  "prediction": "Confidentiality",
  "confidence": 0.79,
  "reasoning": "The clause involves post-departure obligations and
      liquidated damages, and is often listed alongside
      confidentiality duties, so it is more likely a confidentiality
       clause."
}
```

Ground Truth Label: `Non-compete`
Error Record:

```
{
  "sample_id": "lawbench-1027",
  "error_type": "confusable_clauses",
  "error_subtype": "noncompete_vs_confidentiality",
  "prediction": "Confidentiality",
  "expected": "Non-compete"
}
```

(B) ATOMIC PATCH AND PRECEDENT GATE

Judge Reflection:


```
{
  "root_cause": "Retrieval over-represents confidentiality clauses
     and misses non-compete patterns; the harness lacks a rule
     distinguishing non-compete compensation and territory
     restrictions from confidentiality obligations.",
  "affected_rule_id": null,
  "suggested_repair": "Add a new rule that detects non-compete
     clauses by the presence of post-departure competition
     restrictions, monthly compensation, and territory scope.",
  "evidence": "The clause explicitly mentions two-year post-
     departure competition restriction and monthly non-compete
     compensation."
}
```


Atomic Patch (ADD):


```
{
  "operation": "ADD",
  "target_rule_id": "new",
  "target_tier": "volatile",
  "new_rule": {
    "phi": "Employment agreement clause involving post-departure
       restriction on competing business, with monthly non-compete
       compensation and defined territory/duration.",
    "psi": "Classify as Non-compete. Retrieve non-compete precedents
       and check for compensation, duration, and territory. Do not
       classify as Confidentiality unless trade secrets are
       explicitly the subject.",
    "omega": "pending",
    "confidence": null,
    "lifespan": 0
  },
  "provenance": {
    "round": 2,
    "failure_case_id": "lawbench-1027",
    "undo_ref": "undo-lawbench-1027-01",
    "patch_id": "patch-lawbench-1027-01"
  },
  "reason": "The clause is a classic non-compete provision; the
     existing rules overfit to confidentiality because both involve
      post-departure obligations."
}
```


Initial Confidence c_0: 0.72 (judged by base model, with weak prior strength $n_0 = 2$)
Precedent-Based Influence Screening:

- Precedents: similar clause types with non-compete compensation $\rightarrow$ success rate 0.81; confidentiality clauses with trade secret language $\rightarrow$ success rate 0.23.
- Influence score: 0.74.
- Gate threshold: $\epsilon = 0.6$. **Accepted.**

Re-test on the Same Failure Case:


```
{
  "prediction": "Non-compete",
  "confidence": 0.88,
  "reasoning": "The clause includes a two-year post-departure
     competition restriction, monthly non-compete compensation, and
      territory scope, which match non-compete rather than
     confidentiality."
}
```


Status: `PENDING` $\rightarrow$ `VALIDATED` in volatile tier.

(C) VALIDATION AND PROMOTION

Cross-Subset Validation:

- New case 1 (non-compete, different industry): **success**
- New case 2 (confidentiality, no non-compete compensation): **success** (correctly not matched)
- New case 3 (labor contract termination, no post-departure restriction): **success** (correctly not matched)
- New domain (labor dispute arbitration): **success**

Confidence Recalibration (Beta-Bernoulli):

$$\alpha_0 = n_0 c_0 = 2 \times 0.72 = 1.44, \quad \beta_0 = n_0(1 - c_0) = 2 \times 0.28 = 0.56$$

$$S = 4, \quad F = 0$$

$$c_t = \frac{\alpha_0 + S}{\alpha_0 + \beta_0 + S + F} = \frac{1.44 + 4}{1.44 + 0.56 + 4 + 0} = \frac{5.44}{6.00} \approx 0.907$$

Age $a_t = 3$ (validated across three subsequent rounds/subsets).

Promotion Decision:

$$c_t = 0.907 \geq \theta_c = 0.8, \qquad a_t = 3 \geq \theta_{\text{age}} = 3$$

Promoted atomically to stable tier.

Stable-Tier Protection Check: A separate proposal to directly `REFINE` an existing stable rule was rejected because stable-tier rules may only be modified by `MERGE` or `DELETE`. The new `ADD` patch is promoted as a new stable rule, not as a modification of an existing stable rule.

Outcome: The harness now reliably classifies non-compete clauses, while preserving the existing confidentiality rules. Retrieval at inference retrieves the new stable rule for similar queries; conflicts with volatile rules are resolved in favor of the stable rule.

Case 4: SWE-bench Verified — Full EvoRule Harness Evolution

Dataset: SWE-bench Verified (agentic code repair)

(A) FAILURE CASE

Input Sample:

```
GitHub issue:
When `APP_MODE` is not set, `load_config()` raises `KeyError: '
    APP_MODE'`.
Expected: fall back to `'development'` and emit a warning.
Reproduce: `python -m app --config config.yaml`
```

Retrieval Results Summary:

- Retrieved rule candidates: [Add: set default in config.yaml, Refine: wrap CLI in try/except, Add: pytest fixture for env vars, Merge: duplicate config rules, Delete: stale env rule]
- Note: No retrieved rule localizes the fix to `app/config/parser.py` or uses `os.environ.get`.

LLM Raw Output:

```
{
  "prediction": "Add APP_MODE: development to config.yaml",
  "confidence": 0.74,
  "reasoning": "The error is caused by a missing APP_MODE, so
      adding a default configuration value should avoid the KeyError
      ."
}
```

Ground Truth Label: `app/config/parser.py: use os.environ.get('APP_MODE', 'development') and warn on fallback`

Error Record:

```
{
```

```
  "sample_id": "swe-verified-1043",
  "error_type": "localization_error",
  "error_subtype": "wrong_file_edit",
  "prediction": "config.yaml default",
  "expected": "parser.py env fallback"
}
```


(B) ATOMIC PATCH AND PRECEDENT GATE

Judge Reflection:


```
{
  "root_cause": "The harness lacks a rule that maps missing
     environment variable errors to the code that reads the
     variable; it instead defaults to editing configuration files
     .",
  "affected_rule_id": null,
  "suggested_repair": "Add a rule that localizes environment-
     variable fallback fixes to the parser/loader code, using os.
     environ.get with a default and emitting a warning.",
  "evidence": "The issue expects a fallback in code, not a default
     in config.yaml."
}
```


Atomic Patch (ADD):


```
{
  "operation": "ADD",
  "target_rule_id": "new",
  "target_tier": "volatile",
  "new_rule": {
    "phi": "Python application raises KeyError for an environment
       variable during configuration loading, and the expected
       behavior is a fallback default with a warning.",
    "psi": "Localize the fix to the configuration parser/loader (e.g
       ., app/config/parser.py). Use os.environ.get('VAR', 'default
       ') and emit a warning when falling back. Do not modify
       config.yaml unless the issue explicitly requests a default
       configuration value.",
    "omega": "pending",
    "confidence": null,
    "lifespan": 0
  },
  "provenance": {
    "round": 4,
    "failure_case_id": "swe-verified-1043",
    "undo_ref": "undo-swe-1043-01",
    "patch_id": "patch-swe-1043-01"
  },
  "reason": "The existing rules incorrectly prioritize config file
     edits; this rule enforces code-level fallback for environment
     variables."
}
```


Initial Confidence c_0: 0.80 (judged by base model, with weak prior strength $n_0 = 2$)
Precedent-Based Influence Screening:

- Precedents: environment-variable fallback issues resolved in parser code $\rightarrow$ success rate 0.84; config-file default edits for similar issues $\rightarrow$ success rate 0.31.
- Influence score: 0.82.
- Gate threshold: $\epsilon = 0.6$. **Accepted.**

Re-test on the Same Failure Case:


```
{
```

```
  "prediction": "Edit app/config/parser.py: use os.environ.get('
      APP_MODE', 'development') and warn on fallback.",
  "confidence": 0.89,
  "reasoning": "The issue expects a code-level fallback, not a
      config default. The parser reads APP_MODE directly, so the fix
       belongs there."
}
```

Status: `PENDING` → `VALIDATED` in volatile tier.

(C) VALIDATION AND PROMOTION

Cross-Subset Validation:

- New case 1 (missing `DB_URL`, expected fallback in `db/config.py`): **success**
- New case 2 (missing `API_KEY`, expected error, not fallback): **success** (rule correctly does not fire)
- New case 3 (missing `LOG_LEVEL`, fallback to 'INFO' in `logging/setup.py`): **success**
- New case 4 (config file missing key, expected config edit): **success** (rule correctly does not fire)

Confidence Recalibration (Beta-Bernoulli):

$$\alpha_0 = n_0 c_0 = 2 \times 0.80 = 1.60, \quad \beta_0 = n_0(1 - c_0) = 2 \times 0.20 = 0.40$$

$$S = 4, \quad F = 0$$

$$c_t = \frac{\alpha_0 + S}{\alpha_0 + \beta_0 + S + F} = \frac{1.60 + 4}{1.60 + 0.40 + 4 + 0} = \frac{5.60}{6.00} \approx 0.933$$

Age $a_t = 3$ (validated across three subsequent rounds/subsets).

Promotion Decision:

$$c_t = 0.933 \geq \theta_c = 0.8, \qquad a_t = 3 \geq \theta_{\text{age}} = 3$$

Promoted atomically to stable tier.

Stable-Tier Protection Check: No direct modification of existing stable rules was attempted. The new `ADD` patch is promoted as a new stable rule, preserving the integrity of consolidated knowledge.

Conflict Resolution Example: At inference, a volatile rule suggested editing `config.yaml` for a similar issue. The newly promoted stable rule conflicted with the volatile rule. By conflict arbitration, the stable rule took precedence, and the volatile edit was discarded. This demonstrates stable-tier protection and non-regression.

Outcome: The harness now correctly localizes environment-variable fallback fixes to parser/loader code, avoiding unnecessary configuration file edits.

Case 1: S2D Duodenal Ulcer: Evidence Gated ADD and Stable Promotion. In the S2D case, the query described classic duodenal ulcer symptoms, including epigastric burning pain 2 to 3 hours after meals and at night, relief after food intake, and associated nausea. The retriever overrepresented Gastritis and did not include Peptic ulcer disease among the top candidates, so the frozen model predicted Gastritis with 0.82 confidence. EvoRule Harness treats this as a confusable disease failure and generates a diagnostic reflection: the system should not trust a high agreement but low confidence label set when the true disease is absent from retrieval. The resulting atomic patch is an ADD rule. If the top three labels agree but confidence is below 0.85, retrieve challenger diagnoses and requery. This patch passes precedent based influence screening, enters the volatile tier as PENDING, and is retested on the same failed case, where it produces Peptic ulcer disease with 0.91 confidence. Through validation on a new case and a new domain, its calibrated confidence rises from 0.75 to 0.83 to 0.88, and with age 3 it is promoted atomically to the stable tier because $c \geq 0.8$ and $a \geq 3$. A direct REFINE on a stable rule is rejected because stable tier edits are limited to MERGE and DELETE. This illustrates the paper's nonregressive design: new hypotheses are isolated, validated, and only then protected as reliable knowledge.

Case 2: LawBench Noncompete Clause: Refining Confusable Legal Categories In the LawBench case, an employment agreement clause about noncompete obligations after leaving the company, monthly noncompete compensation, and liquidated damages was misclassified as Confidentiality with 0.79 confidence, while the ground truth was Noncompete. The retrieved labels included Confidentiality, Labor Contract Termination, IP Ownership, Service Period, Liquidated Damages,

General Breach, and Labor Dispute, but Noncompete was absent from the top eight. This is a confusable clause failure: the model relies on adjacent legal concepts such as confidentiality and liquidated damages instead of the distinguishing noncompete structure. EvoRule Harness generates a reflection identifying retrieval bias and an overbroad applicability context, then proposes an ADD patch rather than a REFINE on a stable rule. When a clause contains an after departure restriction on competing business, compensation for noncompete, and breach remedies, the harness should consider Noncompete and distinguish it from Confidentiality. The patch first enters the volatile tier as PENDING, is retested on the failed sample, and then accumulates repeated successes across legal subsets. Only after confidence and age thresholds are met is it promoted atomically to stable memory. Because volatile edits cannot directly modify stable rules, validated legal knowledge remains protected while the new distinction is learned, demonstrating traceable credit assignment and evidence gated adaptation.

Case 3: USPTO 50k Reaction Type: Splitting and Refining Condensation versus Amide Coupling In the USPTO 50k case, the model predicted amide coupling but the expected label was heterocycle formation, with the error subtype recorded as condensation versus amide coupling. This is a classic overgeneralization problem: both reaction classes can involve condensation like bond formation, so a broad rule may conflate ring forming heterocycle synthesis with amide bond coupling. EvoRule Harness reflects on the failure as a mismatch between the rule's applicability context and the actual reaction regime. The appropriate atomic operation is either REFINE or SPLIT, applied to a volatile rule rather than a stable one. REFINE narrows the existing rule's phi so it excludes heterocycle formation, while SPLIT decomposes a broad condensation rule into specialized subrules for ring closure and amide coupling. The volatile patch is then retested on the failed sample. If it corrects the prediction, it moves from PENDING to VALIDATED and accumulates outcomes across reaction subsets. Stable tier knowledge is not directly modified. Only after repeated cross subset validation, sufficient confidence, and enough survival age does the refined or split rule become eligible for atomic promotion. This case shows how EvoRule Harness maintains edit granularity and prevents a broad, noisy rule from corrupting related reaction classes.

Case 4: SWE bench Verified: Localization Aware ADD for Environment Fallback In the SWE bench Verified case, the GitHub issue reported that when `APP_MODE` is not set, `load_config` raises `KeyError: APP_MODE`; the expected behavior is to fall back to `development` and emit a warning. The retrieved rule candidates suggested adding a default in `config.yaml`, wrapping the CLI in `try/except`, adding a pytest fixture for environment variables, merging duplicate config rules, or deleting a stale environment rule. None localized the fix to `app/config/parser.py` or used `os.environ.get`. The model therefore predicted adding `APP_MODE: development` to `config.yaml` with 0.74 confidence, which is a localization error: the correct repair is in `parser.py`, using `os.environ.get('APP_MODE', 'development')` and warning on fallback. EvoRule Harness reflects on this as a wrong file edit failure and proposes an ADD rule. If a `KeyError` for an environment variable occurs during config loading and the issue expects fallback behavior, then localize the fix to the parser module and use an environment variable default with a warning, rather than editing `config.yaml`. The rule enters the volatile tier as PENDING, is retested on the same issue, and then must be validated across new SWE tasks. Promotion to stable memory is atomic and occurs only after repeated validation, so the new localization rule becomes trustworthy while existing stable code repair rules remain protected. Conflict arbitration continues to favor stable rules, demonstrating auditable, nonregressive lifelong harness adaptation.