CoDiMem: Evolving Agent Memory through Intervention-Grounded Diagnostic Learning

Anonymous authors
Paper under double-blind review

Abstract

LLM agents increasingly rely on external memory to reuse experience across tasks, motivating recent approaches that evolve memory architectures from interaction experience. However, architecture evolution requires repeatedly deciding what to change next, while existing methods largely use previous trials to select candidate architectures rather than to improve the diagnostic policy that guides subsequent search. We introduce **CoDiMem**, a self-evolving agentic memory system that reuses each architecture trial in two complementary ways: as evidence for selecting the next architecture and as a factual trial record for updating diagnostic guidance, thereby allowing subsequent diagnostic decisions to build on what earlier trials actually established rather than reasoning from each new trajectory in isolation. Across AppWorld and MemoryArena with two LLM backbones, CoDiMem improves upon its paired starting architectures while reducing search cost; on AppWorld with StepFun 3.7 Flash, held-out accuracy increases from 59.03% to 69.44%. Ablations further show that removing diagnostic-guidance updates degrades both task performance and search efficiency, supporting the value of learning from accumulated architecture trials.

1 Introduction

LLM agents increasingly rely on external memory to retain and reuse experience across tasks. Most memory systems, however, are built around fixed mechanisms for writing, retrieving, using, and managing past experience, making a single design difficult to adapt to heterogeneous task demands. Recent work therefore moves beyond accumulating memory content and instead evolves the memory architecture itself from interaction experience (Zhang et al., 2026b; Xiong et al., 2026). This shift introduces a new decision problem: given the current architecture and its execution traces, *what aspect of the memory system should be investigated or modified next?* We refer to the decision rule that answers this question as the *diagnostic policy* D_ψ. The policy is conditioned on *diagnostic guidance*, denoted by ψ and round-specific directional information d, which specifies how evidence should be interpreted when proposing the next architectural hypothesis.

Existing architecture-evolution methods already recognize that execution feedback should guide subsequent search, but they use it in different ways. MemEvolve analyzes trajectories produced by the current parent architecture to identify memory bottlenecks and then generates redesigned descendants (Zhang et al., 2026b). ALMA instead samples previously explored memory designs together with their evaluation logs and asks a meta-agent to reflect on them before proposing a new design (Xiong et al., 2026). These approaches demonstrate that evaluation feedback is useful for architecture search. However, neither formulation explicitly represents each architectural trial as evidence about a diagnostic hypothesis and uses accumulated outcomes to revise persistent diagnostic guidance. In MemEvolve, the diagnosis is primarily reconstructed from the current architecture's trajectories; in ALMA, evaluation logs provide context for generating subsequent designs, but are not organized as hypothesis-linked evidence for updating an explicit diagnostic policy.

This distinction matters for two reasons. First, a trace rarely pins down the architectural cause of a failure. An agent that ignores a useful past experience may never have stored it, may have failed to retrieve it under the current query, or may have retrieved it but not used it at the right step. All three explanations fit the same trajectory, yet each calls for a different change. A trajectory-derived diagnosis is thus a *hypothesis about what a modification will do*, not an established explanation.

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.

Second, testing a hypothesis does not by itself inform later diagnoses. Unless the outcome is retained and fed back, weakly supported directions resurface in later rounds (Section 4.5).

We therefore use architectural evaluation as supervision for diagnosis itself, and build **CoDiMem** on this idea. In each round, the diagnostic policy proposes hypotheses under its current guidance, and CoDiMem tests one targeted modification per hypothesis. Each test produces a *trial record* that links the hypothesis, the modification, and the observed outcome. These records drive architecture selection as usual; reflection over them also revises the diagnostic guidance for the next round. A modification that is not retained can therefore still change where the search goes next.

Our contributions are:

1) Intervention-grounded diagnostic learning for memory-architecture evolution. We formulate architecture search as a coupled process in which diagnostic hypotheses are tested through architectural interventions and their outcomes are retained as explicit trial records. CoDiMem uses these records to revise persistent diagnostic guidance, allowing future search decisions to incorporate evidence accumulated across rounds rather than relying only on the current execution traces.

2) Empirical evidence that learning diagnostic guidance affects both architecture quality and search efficiency. We evaluate CoDiMem on AppWorld and MemoryArena with two LLM backbones. On AppWorld with StepFun 3.7 Flash, the evolved architecture improves held-out accuracy from 59.03% to 69.44% relative to its paired starting architecture. Ablations further show that disabling diagnostic-guidance updates reduces held-out performance and increases search cost, while round-by-round analyses illustrate how accumulated trial records alter the hypotheses pursued in later rounds.

2 Related Work

Adaptive Agent Memory. Agent memory research has progressed from storing past experience to adapting how that experience is represented and used. AWM extracts reusable workflows, while A-MEM organizes and revises interconnected memories (Wang et al., 2025; Xu et al., 2025). ACE reduces information loss during incremental context updates (Zhang et al., 2026d); MemRL learns execution-derived utilities for selecting useful episodic memories (Zhang et al., 2026e); and AgeMem learns when to invoke different memory operations through reinforcement learning (Yu et al., 2026a). Recent evaluations further show that memory quality cannot be inferred from storage or recall alone. Repeated LLM-based consolidation can reduce the utility of otherwise useful experience (Zhang et al., 2026a), while MemoryArena reveals a substantial gap between conversational recall and the effective use of memory in interdependent agentic tasks (He et al., 2026). Together, these results motivate evaluating memory mechanisms through their downstream effects on agent behavior.

From Memory Management to Architectural Search. More recent work makes the memory mechanism itself a target of adaptation. MemSkill revises a bank of reusable memory skills from difficult cases (Zhang et al., 2026c). MemEvolve analyzes execution behavior and generates alternative memory architectures, jointly evolving experiential knowledge and the mechanisms that operate on it (Zhang et al., 2026b). ALMA searches an open-ended space of executable memory designs by using archived implementations and their evaluation logs to inform subsequent proposals (Xiong et al., 2026). Recuris uses structured execution evidence to localize failures to memory components and applies component-scoped, validation-gated updates for long-horizon agents (Yu et al., 2026b). Collectively, these approaches establish memory architecture as an editable object that can be improved from execution feedback.

CoDiMem focuses on a complementary question: *how should the outcomes of previous architectural trials change the diagnosis that directs the next search step?* Existing architecture-search methods use execution feedback to generate, compare, or validate candidate designs. CoDiMem additionally represents each tested diagnostic hypothesis as a *trial record* linking the proposed mechanism, the architectural change used to test it, and the observed outcome. Accumulated trial records are then used to revise *diagnostic guidance*, which conditions the diagnostic policy in subsequent rounds. The distinction therefore lies in the role assigned to trial outcomes: beyond evaluating candidate architectures, CoDiMem uses them as evidence for improving how future search directions are formulated.

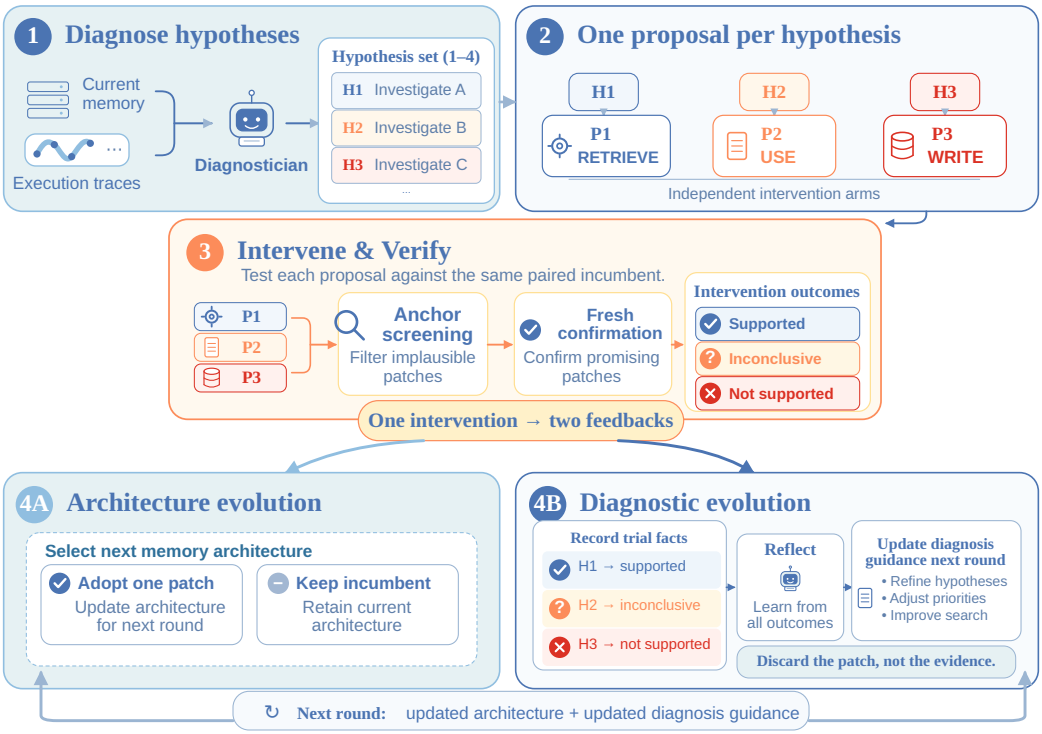

Figure 1: **Overview of CoDiMem.** Each round generates one to four diagnostic hypotheses, binds one executable proposal to each hypothesis, and evaluates the resulting intervention arms against the same paired incumbent. Trial outcomes provide two feedback paths: at most one patch is adopted for architecture evolution, while factual trial records are retained to revise diagnostic guidance for the next round.

From Evaluation Feedback to Diagnostic Guidance. Reflection-based methods demonstrate that natural-language feedback can improve subsequent decisions. Reflexion retains verbal feedback from previous attempts to guide future behavior (Shinn et al., 2023), while GEPA reflects on execution trajectories and evaluation outcomes to propose and test improved prompts (Agrawal et al., 2026). Complementary work in agent debugging emphasizes that such feedback should be grounded in intervention rather than trajectory interpretation alone. DoVer generates failure hypotheses and evaluates them through targeted interventions, showing that multiple distinct interventions may independently repair the same observed failure (Ma et al., 2026). This supports an outcome-oriented view in which a diagnosis specifies a testable direction rather than a unique ground-truth cause.

Work on recursive search further illustrates that *how* previous outcomes are reused matters. Dream-RSI converts recorded discovery histories into replay environments for improving exploration policies (Zheng et al., 2026); its ConvDiv analysis also shows that injecting high-level directional summaries can underperform the corresponding unguided search, suggesting that accumulated feedback should not be converted into irreversible search constraints. CoDiMem therefore separates *trial records*, which preserve factual outcomes of architectural tests, from *diagnostic guidance*, which remains revisable as additional evidence accumulates. This separation enables architectural outcomes to influence subsequent hypothesis generation while preserving the distinction between what was observed and how those observations should guide future search.

3 METHOD

CoDiMem couples memory-architecture evolution with diagnostic learning (Fig. 1). At round r, the memory architecture is $\mathcal{M}_r = (W_r, R_r, U_r, G_r)$, corresponding to WRITE, RETRIEVE, USE, and MANAGE. The diagnostic policy D_{ψ_r} proposes what mechanism to investigate next, conditioned

on persistent diagnostic guidance ψ_r and round-specific recommendations d_r. Architectural trials produce factual trial records E_r, which are used both for architecture selection and for revising the guidance used in later rounds.

3.1 Linking Hypotheses to Architectural Trials

Given current execution traces $\mathcal{T}_r$, architecture $\mathcal{M}_r$, bank B_r, previous trial records $E_{<r}$, and guidance, the diagnostic policy produces

$$\mathcal{H}_r = D_{\psi_r}(\mathcal{T}_r, \mathcal{M}_r, B_r, E_{<r}, d_r),$$

where $\mathcal{H}_r = \{h_{r,i}\}_{i=1}^{K_r}$ and $1 \le K_r \le 4$. Each hypothesis specifies a proposed mechanism, its operator scope, supporting and contrary evidence, and the memory and task effects expected if that mechanism is modified.

For example, a hypothesis may state that failures arise because unsuccessful source trajectories are excluded by WRITE; its predicted effect is that relaxing this rule should increase memory coverage for the affected tasks and, if useful, improve their downstream outcomes. This formulation makes a diagnosis a testable search direction rather than a ground-truth failure explanation.

For each $h_{r,i}$, the proposer generates exactly one bound modification $p_{r,i}$,

$$\mathcal{M}_{r,i} = \mathcal{M}_r \oplus p_{r,i},$$

restricted to the hypothesis's declared operator scope. Thus each hypothesis corresponds to one independent intervention arm. Candidates are checked for executability, scope, activation, and duplication before evaluation; bounded repair and other implementation checks are described in Appendix C. Invalid realizations are retained in the records as NOT_TESTED.

3.2 Fixed Evaluation Produces Trial Records

As illustrated in Fig. 1, all valid interventions are compared with the same paired incumbent rather than with one another. For WRITE interventions, the incumbent and candidate rebuild their banks from the same frozen source corpus; RETRIEVE, USE, and MANAGE interventions receive the same relevant bank or upstream inputs. This isolates the declared intervention as far as the executable architecture permits.

For task set Q, we measure

$$Y_Q(x) = \frac{1}{|Q|}\sum_{q\in Q} y_q(x), \qquad \Delta^Q_{r,i} = Y_Q(x_{r,i}) - Y_Q(x_{r,0}),$$

where $x_{r,0}$ is the incumbent control and $x_{r,i}$ is candidate i.

Verification has two stages. Stage 1 screens every valid candidate on the fixed anchor set A ($|A| = 24$) and advances at most the two candidates with positive paired gain:

$$\mathcal{J}_r = \mathrm{Top2}\{i : \Delta^A_{r,i} > 0\}.$$

Stage 2 reevaluates these candidates on

$$Q_r^{\mathrm{conf}} = A^{\mathrm{ret}} \cup F_r,$$

where $|A^{\mathrm{ret}}| = 12$ and $|F_r| = 12$. Each task is evaluated with three repeated executions, and the fresh sets are disjoint across the five evolution rounds.

The $\Delta > 0$ criterion is a fixed search gate rather than a significance test. In particular, positive anchor gain alone does not support a hypothesis: confirmation on Stage 2 is required, while task-level regressions and uncertainty estimates are retained in the record.

Each hypothesis receives one code-generated trial record $e_{r,i}$ with its tested modification, observed memory changes, stage-specific outcomes, and adoption decision. We assign SUPPORTED if both screening and confirmation gains are positive; NOT_SUPPORTED if an observed screening or confirmation gain is negative; NOT_TESTED if no valid realization is evaluated; and INCONCLUSIVE

Algorithm 1 CoDiMem

Require: $\mathcal{M}_0, B_0, \psi_0$, development tasks, search budget b
$E_{<0} \leftarrow \emptyset,\ d_0 \leftarrow \emptyset$
for round $r = 0, 1, \ldots$ within budget **do**
  Execute incumbent to obtain $\mathcal{T}_r$
  $\mathcal{H}_r \leftarrow D_{\psi_r}(\mathcal{T}_r, \mathcal{M}_r, B_r, E_{<r}, d_r)$
  Generate and validate one proposal for each $h_{r,i} \in \mathcal{H}_r$
  Screen valid interventions on fixed anchors A
  Confirm at most two finalists on $A^{\mathrm{ret}} \cup F_r$
  Adopt at most one supported patch; otherwise retain incumbent
  Construct factual trial records E_r
  $(\psi_{r+1}, d_{r+1}) \leftarrow \mathrm{Audit}[\mathrm{Reflect}(\psi_r, \mathcal{H}_r, E_{\le r}, \mathcal{L}_r)]$
end for
return $\mathcal{M}_r, B_r, E_{<r}, \psi_r, d_r$

otherwise. Thus a candidate with positive Stage 1 gain that does not enter the Top–2 is explicitly INCONCLUSIVE.

At most one supported candidate is adopted:

$$i^\star = \arg\max_{i \in \mathcal{S}_r} \Delta_{r,i}^{Q_r^{\mathrm{conf}}},$$

while all trial records, including those of rejected candidates, are retained in

$$E_{\le r} = E_{<r} \cup E_r.$$

3.3 FROM TRIAL RECORDS TO THE NEXT DIAGNOSIS

Trial records describe what was tested and what happened; they do not themselves determine what should be tested next. CoDiMem therefore reflects over accumulated records and selected paired cases to revise diagnostic guidance:

$$(\psi_{r+1}, d_{r+1}) = \mathrm{Audit}[\mathrm{Reflect}(\psi_r, \mathcal{H}_r, E_{\le r}, \mathcal{L}_r)]\,.$$

The distinction between records and guidance is central. For example, an initial rule may broadly suggest investigating WRITE when useful memory is missing. After trials show that some failures contain no relevant memory whereas others contain relevant memories that were not surfaced, the rule can be narrowed to first distinguish missing storage coverage from retrieval or use failures. The complete evolution of ψ is provided in Appendix E.

Audit prevents reflection from rewriting empirical evidence. Each guidance edit must reference registered hypotheses and trial records, preserve measured outcomes and evidence statuses, and use an allowed update operation (KEEP, NARROW, REPLACE, RETRACT, or RESTORE); otherwise the previous rule is retained.

The next round therefore uses both new trajectories and revised guidance:

$$\mathcal{H}_{r+1} = D_{\psi_{r+1}}(\mathcal{T}_{r+1}, \mathcal{M}_{r+1}, B_{r+1}, E_{\le r}, d_{r+1}).$$

A patch may consequently be rejected while its trial still changes how later hypotheses are formulated. Algorithm 1 summarizes the complete procedure.

4 EXPERIMENTS

We organize our evaluation around four questions: **(RQ1)** whether CoDiMem improves the held-out quality of the memory architectures returned by search; **(RQ2)** whether it does so with lower architecture-search cost; **(RQ3)** how much revising diagnostic guidance from accumulated trial records contributes beyond search with fixed guidance; and **(RQ4)** how the memory architecture and diagnostic guidance change over successive evolution rounds.

LLM backbone	Method	AppWorld			MemoryArena	
		Perf. (%) ↑	Cost ↓	Steps ↓	Perf. (%) ↑	Cost ↓
StepFun 3.7 Flash	NoMemory	52.08	**136.50**	21.12	22.06	**41.67**
	Voyager	<u>64.58</u>	191.75	14.82	27.74	<u>71.75</u>
	Dynamic Cheatsheet	60.42	347.20	**13.62**	29.34	126.58
	ExpeL	63.19	391.69	15.21	<u>35.97</u>	95.15
	MemoryBank	56.94	160.69	19.07	33.48	101.36
	A-MEM	61.11	722.91	<u>14.56</u>	35.55	200.28
	AWM	59.03	158.71	19.63	35.63	89.07
	MemEvolve	<u>64.58</u>	367.19	18.35	35.47	101.52
	CoDiMem	**69.44**	<u>155.40</u>	16.32	**37.36**	106.84
Qwen3.5-27B	NoMemory	52.08	268.59	25.98	28.74	**44.26**
	Dynamic Cheatsheet	<u>68.05</u>	642.06	**16.12**	39.30	225.01
	MemoryBank	54.16	265.70	20.15	42.51	<u>138.22</u>
	A-MEM	66.67	594.30	<u>16.30</u>	<u>45.89</u>	361.70
	AWM	65.28	<u>230.69</u>	23.50	45.20	184.89
	CoDiMem	**70.83**	**214.44**	21.06	**48.10**	220.60

Table 1: **Held-out results by backbone.** AppWorld evaluates 24 Test Normal and 24 Test Challenge tasks with three executions per task (48 tasks; 144 executions), Perf. is overall accuracy; Cost and Steps are execution cost and agent interactions per task. MemoryArena reports macro-averaged subproblem accuracy over 40 held-out Formal Reasoning groups and execution cost per complete group. AWM is CoDiMem's paired starting architecture; MemoryArena MemEvolve uses native evolution. Costs exclude architecture search. Bold and underlined values indicate best and second-best for each metric.

4.1 Experimental Setup

Benchmarks and evaluation. We evaluate on AppWorld and the Formal Reasoning subset of MemoryArena using StepFun 3.7 Flash and Qwen3.5-27B. AppWorld evaluation comprises 24 Test Normal and 24 Test Challenge tasks, each executed three times: 48 distinct tasks and 144 executions, MemoryArena reports the macro-average of within-group subproblem accuracy over 40 held-out task groups. Development data support memory construction and architecture search; held-out tasks are excluded from both.

Comparisons and cost. We compare with NoMemory, Voyager, Dynamic Cheatsheet, ExpeL, MemoryBank, A-MEM, and AWM; MemEvolve provides the architecture-evolution reference where results are available. AWM is CoDiMem's paired starting architecture. Within each benchmark and backbone, methods share the task-execution protocol. Table 1 reports execution cost, whereas Table 2 reports cumulative search tokens, including reflection. Search-cost comparisons describe the reported runs. Appendix A gives protocol details

4.2 RQ1: Held-Out Architecture Quality

Table 1 shows gains over the paired AWM architecture in all four evaluated settings. On AppWorld, it exceeds the strongest baselines, Voyager and MemEvolve (64.58%), by 4.86 points at a lower cost (155.40 versus 191.75 and 367.19); only NoMemory is cheaper. On MemoryArena, the margin over ExpeL (35.97%) is smaller, at 1.39 points, and the costs are comparable (106.84 versus 95.15). With Qwen, CoDiMem also ranks first on both benchmarks among the five reported baselines. On AppWorld, it outperforms Dynamic Cheatsheet (68.05%) at about one third of its cost (214.44 versus 642.06), and it has the lowest AppWorld cost of all methods, including NoMemory. On MemoryArena, it outperforms A-MEM (45.89%) at 61% of its cost (220.60 versus 361.70). CoDiMem does not minimize interaction steps, however: Dynamic Cheatsheet and A-MEM use fewer AppWorld steps with both backbones.

4.3 RQ2: Search Cost and Development Progress

Figure 2(a) tracks development accuracy on 24 fixed anchors. On AppWorld with StepFun, CoDiMem returns an architecture with 69.44% held-out accuracy using 155.7M search tokens, compared with 64.58% and 177.4M for MemEvolve. The reported run therefore uses 12.2% fewer search tokens, although the comparison does not measure cost to a shared performance target.

The role-level accounting in Table 2(b) identifies savings in both actor execution and search-side processing. CoDiMem executes more actor episodes (1,290 versus 912), but uses less aggregate actor input (142.50M versus 156.10M), corresponding to approximately 110k versus 171k input tokens per episode. Diagnosis uses 2.55M versus 9.61M tokens, and bank construction uses 1.99M versus 7.74M. CoDiMem's 3.28M proposal/repair cost and 1.88M diagnostic-guidance reflection cost are included in its total. Per-episode input sums all actor calls.

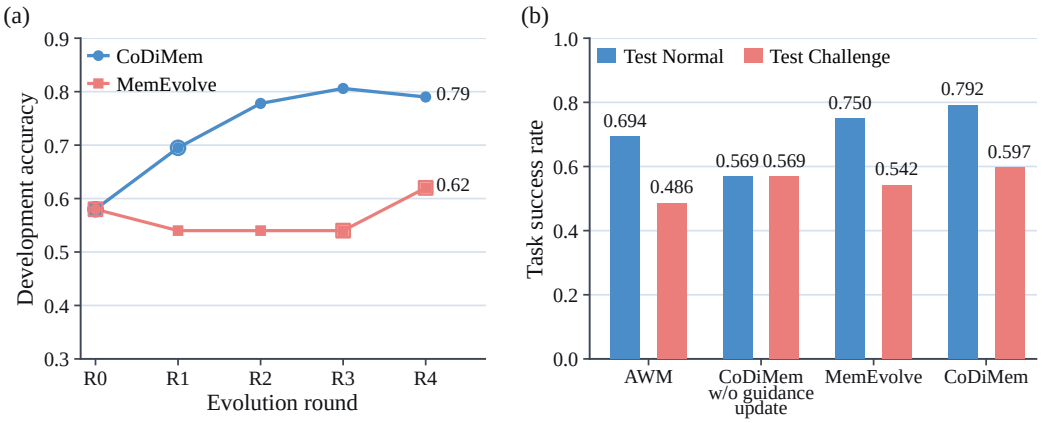

Figure 2: **Development progress and held-out evaluation on AppWorld. (a)** Accuracy on 24 fixed development anchors; open rings indicate architecture changes. **(b)** Test Normal and Test Challenge accuracy with StepFun 3.7 Flash, each measured on 24 tasks with three executions per task.

4.4 RQ3: Contribution of Diagnostic-Guidance Updates

The fixed-guidance variant disables reflection-based guidance updates while retaining architecture search. The available summaries do not specify its access to $E_{<r}$ and d_r, limiting the separation of guidance revision from history access. In Table 2, full CoDiMem exceeds the fixed-guidance variant by 12.50 percentage points in held-out accuracy (69.44% versus 56.94%) while using 19.5% fewer search tokens (155.7M versus 193.3M). Actor episodes decrease by 30.4%, from 1,854 to 1,290, despite higher input per episode (110k versus 98k). Reflection accounts for 1.88M tokens, approximately 1.2% of the full run; the net saving of 37.6M tokens includes this overhead.

4.5 RQ4: What Changes During Architecture Search?

Figure 3 presents a case study of CoDiMem's search. The questions under investigation shift from memory specificity to missing coverage, retrieval matching, and exposure, and later to the transfer of instance-specific answers. Coverage repair and answer abstraction are considered as alternative directions, rather than necessarily cumulative architectural changes.

The example "Alice / Jan 5 / $42" becoming "person / date / value" illustrates replacing instance-specific constants with reusable slots. Likewise, the trajectory illustrates changing search directions.

(a) Held-out performance and run-level search cost

Method	Perf. (%) ↑	Search tokens (M) ↓	Relative tokens	Actor episodes	Actor input per episode (k)
MemEvolve (reference)	<u>64.58</u>	<u>177.4</u>†	1.14×	912	171
CoDiMem (fixed guidance)	56.94	193.3	1.24×	1,854	98
CoDiMem	**69.44**	**155.7**	1.00×	1,290	110

(b) Search-token breakdown: input / output (M)

Role	CoDiMem	Fixed guidance	MemEvolve
Actor	142.50 / 3.47	182.40 / 5.00	156.10 / 3.42
Diagnosis	2.38 / 0.17 (12)	2.02 / 0.12 (10)	9.10 / 0.51 (393)
Proposal + repair	1.87 / 1.41 (62)	1.63 / 1.15 (46)	0.19 / 0.37 (51)
Diagnostic-guidance reflection	1.55 / 0.33 (19)	0 (disabled)	—
WRITE induction / bank construction	1.66 / 0.33	0.86 / 0.10	6.38 / 1.36

M and k denote millions and thousands of tokens. Parentheses in panel (b) give LLM-call counts. "—" marks a stage that MemEvolve does not have. Search tokens in panel (a) are the sum of all roles in panel (b) for each method. Relative tokens use CoDiMem as the reference, and actor input per episode is rounded.

Table 2: Diagnostic-guidance ablation and search-token accounting on AppWorld with StepFun 3.7 Flash. Performance is the held-out task accuracy of each returned architecture on 24 Test Normal and 24 Test Challenge tasks, with three executions per task (48 tasks; 144 executions). Search tokens aggregate input and output usage, including diagnostic-update overhead, rather than per-task deployment cost. Bold and underlined values mark the best and second-best performance and total-token values, respectively. Role-level coverage differs for the MemEvolve reference.

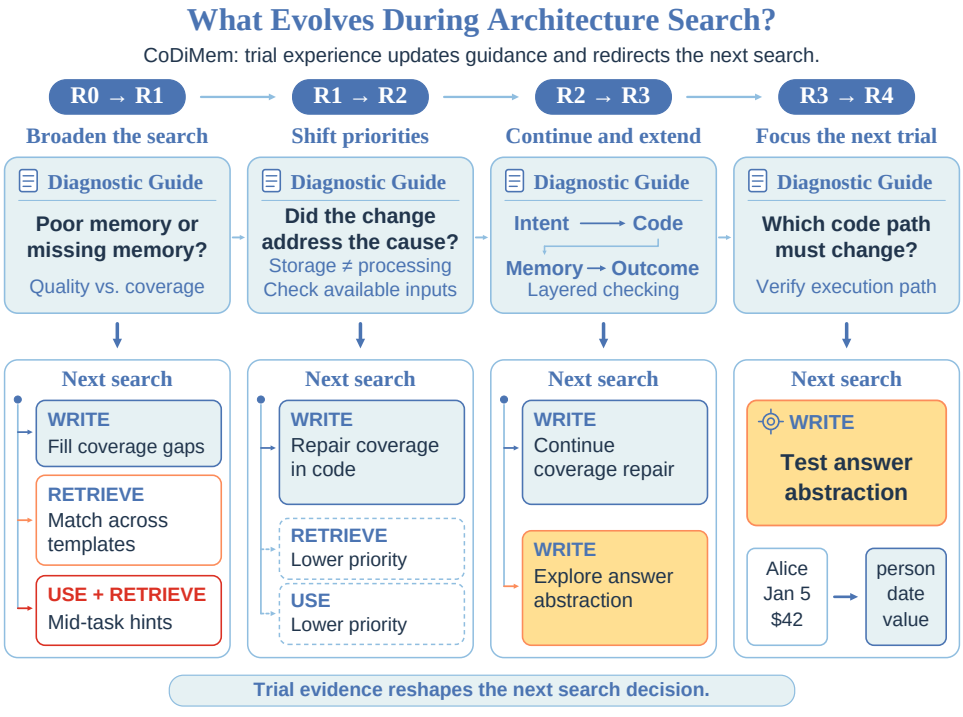

Figure 3: **Diagnostic guidance and subsequent search directions.** The final panel illustrates abstraction from instance-specific values to reusable slots. Arrows indicate round order.

5 CONCLUSION

We introduced **CoDiMem**, which uses architectural trial records to revise the diagnostic guidance that directs memory-architecture search. Its diagnostic policy can therefore draw on tested modifications and their outcomes, including interventions whose patches are not adopted. On AppWorld and the Formal Reasoning subset of MemoryArena, the evaluated architectures improve upon their paired starting points. In the reported AppWorld ablation, the full method achieves higher held-out accuracy with fewer search tokens than the frozen-guidance variant, providing initial evidence for the value of revising diagnostic guidance during search.

6 LIMITATIONS

Our findings are bounded by the tested implementations, benchmarks, backbones, and search horizons. An intervention evaluates a particular realization of a hypothesis, not a unique underlying cause of failure; its outcome does not establish or refute every implementation of the same direction. Natural-language reflection can also overgeneralize limited or noisy trial records, so revised diagnostic guidance need not produce better subsequent hypotheses. AppWorld evaluation covers 48 held-out tasks with three executions per task, rather than the full test sets; these repetitions do not characterize variability across independent architecture-search runs. The current ablation also does not fully distinguish guidance revision from access to historical trial records. Whether the observed benefits persist over longer searches, under changing task distributions, or when diagnostic guidance is transferred across architectures and backbones remains to be established.

ETHICS STATEMENT

This work evaluates agent memory on AppWorld's simulated application APIs and the mathematics and physics tasks in the Formal Reasoning subset of MemoryArena. It does not involve human participants or private user data. To support transparent interpretation, we retain trial records for unsuccessful, inconclusive, and non-adopted interventions, rather than considering only accepted patches. Intervention outcomes are interpreted as evidence about tested architectural changes, not as proof of a unique failure cause. Our benchmark results do not establish the safety or reliability of deployment with real users and data.

REPRODUCIBILITY STATEMENT

Sections 3 and 4.1 describe the search procedure and evaluation protocol. The repository records run manifests, execution commands, seeds, prompt hashes, architectural patch diffs, and task-level evaluation outcomes. Trial records link diagnostic hypotheses to tested modifications, measured effects, and adoption decisions. We distinguish repeated task executions from independent architecture-search runs and report search-token consumption separately from task-execution cost.

AI USE STATEMENT

Generative AI tools were used solely to assist with code debugging, refine the language of the manuscript, and search for relevant literature. All other work—including formulating the research questions, developing the method, implementing the software, designing and conducting the experiments, analyzing and interpreting the results, and writing the scientific content of the manuscript—was carried out by the human authors. The authors reviewed all AI-assisted outputs and take full responsibility for the manuscript, the reported findings, and their interpretation.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang,

Christopher Potts, Koushik Sen, Alex Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/0e9e708b6f48e14fd0ac29e167413f76-Abstract-Conference.html.

Zexue He, Yu Wang, Churan Zhi, Yuanzhe Hu, Tzu-Ping Chen, Lang Yin, Ze Chen, Tong Arthur Wu, Siru Ouyang, Zihan Wang, Jiaxin Pei, Julian McAuley, Yejin Choi, and Alex Pentland. MemoryArena: Benchmarking agent memory in interdependent multi-session agentic tasks. *arXiv preprint arXiv:2602.16313*, 2026. doi: 10.48550/arXiv.2602.16313. URL https://arxiv.org/abs/2602.16313.

Ming Ma, Jue Zhang, Fangkai Yang, Yu Kang, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. DoVer: Intervention-driven auto debugging for LLM multi-agent systems. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/2f5fdce256ff0d7fb774da76e5e63209-Abstract-Conference.html.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic Cheatsheet: Test-time learning with adaptive memory. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7080–7106, Rabat, Morocco, 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.eacl-long.333. URL https://aclanthology.org/2026.eacl-long.333/.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.850. URL https://aclanthology.org/2024.acl-long.850/.

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. *Transactions on Machine Learning Research*, 2024. URL https://openreview.net/forum?id=ehfRiF0R3a.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 63897–63911. PMLR, 2025. URL https://proceedings.mlr.press/v267/wang25bx.html.

Yiming Xiong, Shengran Hu, and Jeff Clune. Learning to continually learn via meta-learning agentic memory designs. *arXiv preprint arXiv:2602.07755*, 2026. URL https://arxiv.org/abs/2602.07755.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-Mem: Agentic memory for LLM agents. In *Advances in Neural Information Processing Systems*, volume 38, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/19909c36f51abc4856b4560aff3d36d6-Abstract-Conference.html.

Yi Yu, Liuyi Yao, Yuexiang Xie, Qingquan Tan, Jiaqi Feng, Yaliang Li, and Libing Wu. Agentic memory: Learning unified long-term and short-term memory management for large language model agents. *arXiv preprint arXiv:2601.01885*, 2026a. URL https://arxiv.org/abs/2601.01885.

Zhaochen Yu, Yingcheng Wu, Zhenfei Yin, Kaiyuan Chen, Zhe Zhao, Mengdi Wang, Shuicheng Yan, and Ling Yang. Recursive experiential-working memory evolution for long-horizon agent harnesses. *arXiv preprint arXiv:2608.24876*, 2026b. URL https://arxiv.org/abs/2608.24876.

Dylan Zhang, Yanshan Lin, Zhengkun Wu, Yihang Sun, Bingxuan Li, Dianqi Li, and Hao Peng. Useful memories become faulty when continuously updated by LLMs. *arXiv preprint arXiv:2605.12978*, 2026a. URL https://arxiv.org/abs/2605.12978.

Guibin Zhang, Haotian Ren, Chong Zhan, Zhenhong Zhou, Junhao Wang, He Zhu, Wangchunshu Zhou, and Shuicheng Yan. MemEvolve: Meta-evolution of agent memory systems. In *International Conference on Machine Learning*, 2026b. URL https://arxiv.org/abs/2512.18746.

Haozhen Zhang, Quanyu Long, Jianzhu Bao, Tao Feng, Weizhi Zhang, Haodong Yue, and Wenya Wang. MemSkill: Learning and evolving memory skills for self-evolving agents. *arXiv preprint arXiv:2602.02474*, 2026c. URL https://arxiv.org/abs/2602.02474.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, 2026d. URL https://arxiv.org/abs/2510.04618.

Shengtao Zhang, Jiaqian Wang, Ruiwen Zhou, Junwei Liao, Yuchen Feng, Zhuo Li, Yujie Zheng, Weinan Zhang, Ying Wen, Zhiyu Li, Feiyu Xiong, Yutao Qi, Bo Tang, and Muning Wen. MemRL: Self-evolving agents via runtime reinforcement learning on episodic memory. *arXiv preprint arXiv:2601.03192*, 2026e. URL https://arxiv.org/abs/2601.03192.

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. doi: 10.1609/aaai.v38i17.29936. URL https://ojs.aaai.org/index.php/AAAI/article/view/29936.

Tong Zheng, Xidong Wu, Zheng Zhang, Zhankui He, Chaoyi Zhang, Benjamin Coleman, Ruoqiao Wei, Di Bai, Haolin Liu, Rui Liu, Xue Wang, Yue Zhuan, Wang-Cheng Kang, Renkai Xiang, Heng Huang, Xinwu Cheng, and Yunsong Guo. Dream-RSI: Recursive self-improvement through evolving worlds. *arXiv preprint arXiv:2609.14858*, 2026. URL https://arxiv.org/abs/2609.14858.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. MemoryBank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19724–19731, 2024. doi: 10.1609/aaai.v38i17.29946. URL https://ojs.aaai.org/index.php/AAAI/article/view/29946.

A Experimental Protocol

Data splits. We evaluate on AppWorld (Trivedi et al., 2024) and the *Formal Reasoning* subset of MemoryArena (He et al., 2026). Development data are used for memory construction, diagnosis, and architecture selection; held-out tasks are never seen during search (Table 3).

On AppWorld, a 40-task source pool supplies trajectories for memory construction and diagnosis; each round runs 24 of these tasks once. Stage 1 screens candidates on 24 fixed anchors. Stage 2 uses the first 12 anchors plus 12 fresh tasks, drawn from 60 fresh tasks reserved for R0–R4 and never reused across rounds. Both stages run each task three times per architecture. All manifests are drawn with a fixed seed before R0, without reference to outcomes or difficulty. The held-out set is 24 Test Normal and 24 Test Challenge tasks, three executions each.

MemoryArena is split by task group to keep cross-session dependencies intact: 20 groups for development and 40 for held-out evaluation. Memory may evolve within a held-out group and is reset between groups.

Evolution protocol. CoDiMem runs five rounds (R0–R4). Each round proposes 1–4 hypotheses, realizes each as one candidate, and compares every valid candidate with the same incumbent. At most two candidates advance to Stage 2 and at most one is committed. Stage 2 was reached in R0 and R1. MemEvolve (Zhang et al., 2026b) uses the same data boundary and number of rounds with its native search procedure.

Execution and metrics. Within each benchmark–backbone setting, all methods share the actor, task instructions, tools, evaluator, and execution limits, with temperature 0 and seed 42. AppWorld reports task success, execution cost per task, and interaction steps; MemoryArena reports macro-averaged within-group subproblem accuracy and cost per group. Search cost is reported separately as cumulative token usage.

Benchmark	Data role	Usage
AppWorld	Source / diagnosis	Memory construction and diagnosis
	24 fixed anchors	Stage 1 screening
	Fresh tasks	Stage 2 confirmation
	Test Normal / Challenge	Final evaluation only
MemoryArena	20 groups	Development / search
	40 groups	Held-out evaluation

Table 3: **Data roles.** Held-out tasks are never used during memory construction or search.

Method	Implementation	Arch. search
NoMemory	No cross-task memory; same actor and within-task history.	No
Voyager (Wang et al., 2024)	Reusable procedural skills from source experience, retrieved per task.	No
Dynamic Cheatsheet (Suzgun et al., 2026)	A compact natural-language cheatsheet updated from development experience.	No
ExpeL (Zhao et al., 2024)	Experiential lessons extracted from source trajectories and retrieved at inference.	No
MemoryBank (Zhong et al., 2024)	Episodic text memories with native relevance-based retrieval and updates.	No
A-MEM (Xu et al., 2025)	Native linked-memory organization, retrieval, and updates.	No
AWM	Workflows induced from source trajectories; CoDiMem's starting architecture.	No
MemEvolve (Zhang et al., 2026b)	Native memory-architecture evolution under the same data boundary.	Yes

Table 4: **Baseline implementations.**

B Baseline Implementations

All baselines run in the same execution interface and receive the same source experience, actor, tools, and evaluator. Each builds and uses its own native memory; no bank is shared across methods (Table 4). Non-search baselines never see CoDiMem's anchor or confirmation outcomes, and held-out trajectories are never written back to memory. AWM is both a baseline and CoDiMem's starting architecture, so the AWM–CoDiMem gap measures the effect of evolution directly.

C Candidate Validation and Bounded Repair

Validation. Before any task evaluation, each candidate must pass four code-owned checks: it runs in a fresh process (*executability*); it changes only the operators its hypothesis declares (*scope*); its changed code is on the executed path and alters the derived bank or the exposed memory on fixed probes (*activation*); and its bank or exposure signature differs from every earlier candidate in the run (*duplication*). Configuration-only changes are rejected. None of these checks sees task outcomes.

Repair budget. Each hypothesis allows one initial implementation and up to two replacements, each with at most one constrained repair limited to interface or runtime defects. A hypothesis that exhausts this budget is recorded as `NOT_TESTED` and counts as no evidence either way.

Evidence status and commitment. Let Δ_A and Δ_C be the paired success differences in Stage 1 and Stage 2. Statuses follow Table 5. Ties in Δ_A or Δ_C are broken by fewer interaction steps, then fewer task losses. The committed candidate is the `SUPPORTED` one with the largest Δ_C; confidence intervals are recorded but do not veto adoption.

Round	Hyp.	Not tested	Repairs
R0	2	1	2 / 1
R1	3	0	0 / 0
R2	1	0	0 / 0
R3	2	1	0 / 0
R4	1	0	0 / 0
Total	9	2	2 / 1

Condition	Status
No valid candidate	`NOT_TESTED`
$\Delta_A < 0$, or $\Delta_A > 0$ and $\Delta_C < 0$	`NOT_SUPPORTED`
$\Delta_A > 0$ and $\Delta_C > 0$	`SUPPORTED`
Otherwise	`INCONCLUSIVE`

Table 5: **Per-round hypotheses and evidence status.** Repairs: repair attempts / attempts that then passed validation. `INCONCLUSIVE` covers $\Delta = 0$ and positive candidates that did not reach Stage 2.

Reflection input. The reflection input L_r contains the full verdict of every hypothesis plus up to four same-task incumbent/candidate pairs per executed hypothesis (improved, harmed, unchanged-failure, unchanged-success, in order of paired change), capped at ten pairs per round. Each pair carries outcomes, evaluator feedback, exposed memory, and trajectory excerpts. Five incumbent failures or success contrasts are added as context.

D TRIAL RECORDS

Trial records are built by code from the hypothesis, its realization, and its paired verification; reflection cannot edit them. A record is kept even when no valid candidate exists. Table 6 lists the main fields.

Group	Content
Hypothesis	Direction, predicted mechanism, expected effect, target operator, continued hypothesis.
Realization	Changed operators, mechanism summary, modified symbols, config changes, or the reason it was not tested.
Observed change	Differences in exposed memory and derived bank versus the incumbent.
Outcome	Stage 1 and Stage 2 deltas with 95% intervals; improved, harmed, and unchanged tasks.
Decision	Support status and whether the candidate was committed.

Table 6: **Trial record fields.**

For example, `DX_R00_H1` is a WRITE change with $\Delta_A = +0.097$ and $\Delta_C = +0.028$, recorded as `SUPPORTED` and committed. `DX_R02_H1` is a coverage change with $\Delta_A = 0$, recorded as `INCONCLUSIVE`; it grew the bank from 69 to 75 items and is cited by rule G25.

E EVOLUTION OF DIAGNOSTIC GUIDANCE

Each reflection produces reusable rules ψ and round-specific recommendations d, which the next diagnosis must adopt or decline. Guidance is capped at 7,000 characters and four new rules per round. Over R0–R4, all ten accepted edits were additions (Table 7); R4 returned no update. The final ψ_4 is ψ_0 plus G17–G26.

Initial diagnostic guidance ψ_0

Hypothesis formation (G01–G03).
Form architecture-level hypotheses from the whole batch of source traces, not from one memorable task. Each hypothesis names a direction (what the memory architecture does wrong or leaves unexploited and which way an intervention should move it), the mechanism you predict, and the effect an intervention should have on task success. Consider WRITE, RETRIEVE, USE and MANAGE without presuming one is faulty.

Evidence adjudication (G04–G05).
Support every hypothesis with exact trace ids and state its counterexamples or uncertainty. Weigh what the actor actually did after memory exposure against what the exposed memory contained; do not reconstruct task, memory, state, action, outcome or numeric facts, code owns them.

Target operator (G06–G07).
Name the target operator when the evidence determines it; otherwise mark it UNDETERMINED and list the operators that could plausibly realise the hypothesis. WRITE is a candidate only when the hypothesis is about what gets stored from source trajectories.

Prior experience use (G08–G11).
Read the code-owned outcome of every earlier hypothesis: which tasks improved, which were harmed, whether Stage 2 confirmed the screening. A realisation that was committed is part of the incumbent now. Use these facts to judge which evidence patterns have and have not translated into task success. Judge an earlier result through what its realisation actually changed in what the actor saw; a realisation that did not do what its hypothesis intended, or that changed nothing observable, leaves that reading untested rather than refuted.

Selection strategy (G12–G14).
Every listed hypothesis is realised as one controlled candidate and verified on the same paired tasks. List the hypotheses that are distinct in mechanism and testable within the budget; prefer the ones whose expected effect the fresh confirmation tasks can show. For each hypothesis, state what the next trial is meant to resolve: the problem still unexplained, the competing explanations and what earlier tests have and have not discriminated, what the intervention adds beyond the realisations already tested, and which observable results would support it, weaken it or leave it undetermined.

Uncertainty and abstention (G15–G16).
Propose only as many hypotheses as the evidence distinctly supports; never pad to the maximum. Keep unresolved directions unresolved and do not claim a verified cause or a unique root cause.

After	Added	Cited trial records
R0	G17–G19	DX_R00_H1
	G20	DX_R00_H2
R1	G21	DX_R01_H2
	G22–G23	DX_R01_H1
	G24	DX_R01_H1, DX_R00_H1
R2	G25	DX_R02_H1
R3	G26	DX_R01_H1, DX_R02_H1, DX_R03_H1, DX_R03_H2

Table 7: **Accepted guidance edits.** All earlier rules were kept in every round.

Accepted additions G17–G26

After R0, used in R1.

G17. Before citing a failed trace as support for a hypothesis about WRITE quality or USE timing, check whether the trace's task template has any memory in the bank (bank_items_from_same_task_template $>$ 0). A failed trace with no same-template memory in bank is evidence about bank coverage or RETRIEVE matching, not about the hypothesised defect. Cite it only against hypotheses that address bank coverage.

G18. When a hypothesis predicts that a memory deficit (e.g., empty API names, generic text) causes failure, weigh the ratio of same-template successes to same-template failures among traces with that deficit. A high success ratio weakens the causal claim and should be reported as a counterexample or uncertainty, not dismissed.

G19. Apply the decision table to your own evidence: (a) specific new evidence or a located repairable defect in a tested mechanism -> continue that line with the basis stated; (b) several valid tests without incremental gain while the earlier explanation does not cover the residual failures -> deprioritise further fine-tuning and compare other mechanisms; (c) results that cannot discriminate between explanations -> choose the intervention that discriminates them, not an arbitrary re-implementation.

G20. A hypothesis with no executable realisation (NOT_TESTED) carries no evidence about its direction. Do not cite it as support or counterexample, and do not use it to justify continuing a line. Re-attribute the direction to a new realizable hypothesis before testing.

After R1, used in R2.

G21. Before proposing a RETRIEVE or USE mechanism that depends on API names, observations or previous actions, check which phase the operator runs in and what inputs that phase receives in the incumbent. A mechanism whose required signal is absent from the operator's phase inputs cannot be tested by changing that operator alone; the phase itself must change.

G22. When a hypothesis targets a specific defect (e.g., bank coverage gaps, missing API names), verify that the realised code changes actually alter the condition that causes that defect. A WRITE change that keeps success_only=True and store_failed_trajectories=False does not test a coverage-gap reading, regardless of its effect on other aspects of memory.

G23. After a commit, compare the persisting failures' memory facts against the defect the committed hypothesis claimed to fix. If the same defect pattern (e.g., bank_items_from_same_task_template=0) is still present in the persisting failures, the commit did not address it and the reading is not confirmed.

G24. When a WRITE hypothesis targets bank coverage gaps, distinguish eligibility changes (which trajectories are stored) from quality changes (how stored content is processed). Only an eligibility change can test the coverage-gap reading; a quality change tests a different reading about content usefulness.

After R2, used in R3.

G25. For every hypothesis, separately state the four layers: (a) the intended mechanism and its expected effect on the actor's observed memory, (b) the actual code changes and effective configuration after default interception, (c) the observed memory changes (exposed item sets, block sizes, same-template exposure, raw dumps, derived bank differences), and (d) the paired result. A result does not test the reading if layer (b) diverged from layer (a) or if layer (c) changed nothing observable; classify such a result as untested, not merely inconclusive.

After R3, used in R4.

G26. A located repairable defect is a code-level condition whose current value directly causes the hypothesised failure pattern (e.g., eligibility flags that exclude the trajectories the hypothesis says are missing, phase inputs that lack the signal the mechanism requires, or a disabled code path guarded by an effective false flag). A result is untested if the realised code did not alter this condition.

Direction recommendations. Table 8 lists each recommendation and the next diagnosis's response. A DECLINED response can still agree in part, as in R02-D1.

Direction	Recommendation	Response	Reason
R00-D1	Deprioritise WRITE-quality tuning.	DECLINED	Pursue eligibility (coverage), not quality.
R00-D2	Compare coverage vs. cross-template matching.	ADOPTED	R1-H1 tests coverage; R1-H2 tests matching.
R01-D1	Deprioritise API-match ranking.	DECLINED	Ranking harmed success; pursue coverage.
R01-D2	Deprioritise error-triggered exposure.	DECLINED	Exposure harmed success; pursue coverage.
R02-D1	Keep matching/timing deprioritised.	DECLINED	Agrees; BEGIN inputs lack the needed signals.
R03-D1	No coverage tuning without eligibility change.	DECLINED	Requires a new WRITE procedure, not flag changes.

Table 8: **Recommendations and responses.** R4's recommendations have no R5 response.

Audit. The audit is deterministic code. It checks schema validity, fact references, identifiers, prescriptive rules, duplicates, and capacity. Across R0–R4 it rejected two malformed drafts, returned three over-budget additions, and dropped six recommendations, for example one that leaked a task ID.

F PROMPTS

We reproduce the diagnostician and reflection prompts. The diagnostic system prompt is followed by ψ_0 (Appendix E); from R1 onward, its last paragraph is replaced by the history clause. The proposer and repair prompts, whose rules are summarized in Appendix C, are released with the code.

Diagnostician / System prompt

Diagnose the shared memory architecture from observational execution traces. Code owns facts. Return diagnosis_result (the architecture-level diagnosis), evidence (exact trace ids with the fact each shows) and hypotheses. Each hypothesis states its direction, its supporting trace ids and evidence, its counterexamples or uncertainty, the predicted mechanism and the expected effect of an intervention on task success. Give target_operator when the evidence determines which operator must change; otherwise use UNDETERMINED and list the plausible candidate_operators.

Propose between 1 and 4 hypotheses: as many as the evidence distinctly supports, never padded. Every hypothesis you list will be realised as one controlled candidate and verified, so list only hypotheses worth a trial. A hypothesis worth a trial calls for a new or changed procedure inside an operator (how it ranks, queries, selects, synthesizes, decides when to act, or induces memories); a hypothesis that would be satisfied by changing the value of an existing configuration key (how many items, a threshold, a mode flag, an instruction string) is not worth a trial and must not be listed.

For every hypothesis, applies_guidance names the guidance component(s) it applies and says in one or two sentences how they shaped its reading of the evidence. Do not claim a verified cause, diagnostic ground truth or unique root cause. Do not design patches.

The evidence is an aligned, code-owned view of the source traces (serialization_contract): per trace the evaluator result with the failed requirements, the memory timeline (what RETRIEVE ranked, what USE exposed with per-item facts: source task template, size, whether it is a raw trajectory dump, the API names it mentions, and the actor's next commands), the complete step list (command, HTTP calls with status, execution errors, observation head/tail) and memory_use_facts (API names in the exposed memory vs the API names the actor called, first error, same-template exposure, and how many source trajectories of the same template the WRITE corpus holds and how many succeeded: the incumbent WRITE stores successful trajectories only, so a template with no successful source trajectory has no memory whatever RETRIEVE or USE do).

Whether the actor read or followed a memory is NOT_OBSERVABLE: never assert it; reason from the string-level facts and say when a reading rests on an unobservable. evidence_summary and current_bank.composition are computed tallies; operator_runtime_facts state what each operator does now, what its procedure receives and which runtime information is legally available per phase; current_configuration is the incumbent's present parameters, not the boundary of what can change.

Account for every failed trace: cite it as support or counterexample of a hypothesis or list it in residual_failures_not_explained with what it shows that none of your hypotheses explains. A trace whose episode ended in a provider/infrastructure error (evidence_quality INFRASTRUCTURE_FAILURE, listed in evidence_summary.infrastructure_failed_trace_ids) has no valid outcome: it is not evidence about the memory architecture and must not be cited as support, counterexample or evidence.

search_directions_from_reflection lists the search directions the last reflection recommended from the code-owned feedback of the previous round (each with the facts it rests on and the observable that would tell it apart), and line_verdicts its reading of every line of work so far. They are recommendations, not orders: answer every one in search_direction_responses (ADOPTED with the hypothesis that realises it, or DECLINED with the evidence in front of you that outranks it), and set responds_to_search_direction on the adopting hypothesis.

A hypothesis that continues a line which already ran (continues_hypothesis_id names an executed hypothesis) must fill discriminating_observable: the code-owned observable of the paired run that this trial will move differently from the recorded realisation (see observed_memory_change and the paired result of that realisation) and the result that would refute the line; a reworded instruction or a re-weighted score that moves the same observables the same way is the same trial again.

required_runtime_inputs names, from the runtime_inputs table, what the predicted mechanism reads; bank metadata keys must be keys the items actually carry (runtime_inputs.bank_metadata_keys); an IN-phase input exists only if the realisation runs the operator at IN (activates_in_phase true), which the incumbent does not.

There is no prior hypothesis experience in this initial round; write prior_experience_response as "NONE" and what_changes_vs_prior as null. No held-out outcomes are available. Follow the serialization contract, the operator ownership semantics and the diagnostic guidance below.

Diagnostician / History clause (R1 onward)

verified_hypothesis_experiences lists, per earlier hypothesis, the realisation that actually ran (operators, mechanism, code changes, configuration) and what the paired verification showed (improved, harmed and unchanged task ids, aggregate deltas with intervals, Stage-2 status, support status, whether it was committed). In prior_experience_response say, per experience_id, how it shaped this diagnosis.

Set continues_hypothesis_id when a hypothesis re-tests or varies an earlier one and then state in what_changes_vs_prior what it changes relative to that tested realisation (its reading of the evidence, its mechanism or its target) and why the tested outcome motivates it; otherwise null. How to use these outcomes is part of the diagnostic guidance below.

tested_realisations_by_operator indexes, per operator, every realisation that actually ran and what it bought; for a hypothesis whose target operator (or any candidate operator when UNDETERMINED) appears there, what_changes_vs_prior must state what the hypothesis adds beyond those realisations, whether or not it continues one of them; null only when nothing was tested on that operator.

Reflection / System prompt

You are the reflective optimiser of the diagnostic guidance of a memory-architecture evolution loop, in the manner of GEPA. The guidance is the only thing you optimise: how the diagnosis reads its evidence, forms hypotheses and chooses what is worth testing. It is a list of judgement items, each with the experience it rests on. You read the guidance in force, the diagnosis it produced, and code-owned feedback about what happened when its hypotheses were realised and verified, and you return (a) a re-examination of every existing item - kept, narrowed, replaced, retracted or restored, each with the fact that decides it - and (b) the judgement methods that are missing.

Keep four layers apart for every hypothesis: what the diagnosis intended, what code was actually changed, what actually changed in what the actor saw, and the paired result. A result is not the correctness of the diagnosis: an implementation that did not do what was intended, or that changed nothing the actor saw, leaves the reading untested; a supported result does not make the reading complete if the residual failures are not what it explained; a committed change makes judgements about the previous incumbent worth re-checking.

Retraction targets the judgements written into the guidance, never the execution facts. Insufficient evidence is a reason to keep an item, not to weaken it; there are no confidence scores and nothing decays with age. Whether the actor read or followed a memory is NOT_OBSERVABLE in this evidence; never build guidance on it.

Guidance never bans or forces an operator or direction, never encodes a conclusion about a specific task, never contains identifiers, and does not decide the next mechanism or patch: it teaches the checks. NOT_TESTED hypotheses carry no evidence about their direction. Return the structured result only.

Reflection / Task instructions

First fill hypothesis_review, one row per hypothesis of this round, from the code-owned feedback: (1) which evidence in front of the diagnosis it did not cite or misread (diagnosis_evidence_audit); (2) whether the executed code did what the hypothesis intended and what actually changed in what the actor saw (intended_vs_realised); (3) whether the residual failures, and the failures that persist after a commit, still look like what the hypothesis explained (residual_evidence, defect_persistence_since_last_commit); (4) why continuing this line would or would not be worth more than testing another explanation.

Then re-examine EVERY item of current_guidance.items (item_review, one row per item, retracted ones included): does it still have a basis; should its scope be narrowed to what the evidence supports; is it weakened by counterexamples; does it rest on a reading the four layers show untested; did the committed change make it inapplicable; should a retracted item be restored? Give the deciding fact for each; KEEP when the evidence is insufficient to change it.

Then add the judgement METHODS that are missing (new_items): which code facts to compare, what a located repairable defect looks like, when an unexplained residual outranks a refinement, how to tell an untested reading from a refuted one, and how to state what a trial resolves. Do not write conclusions about which operator is at fault, do not prescribe the next mechanism, and do not add qualifiers to how hypotheses are worded. Return NO_UPDATE when the feedback justifies no change.

Separately from the items, fill search_directions: what the facts recommend the NEXT diagnosis should test. Per line of work, CONTINUE_LINE only with a located repairable defect or an observable the next trial would move differently from the recorded realisation (observed memory change), DEPRIORITISE_LINE when the line was properly realised, moved what it meant to move and gained nothing, and nothing new was located; NEW_DIRECTION for a reading of the residual and uncited failures that no tested line explains, naming the operator whose procedure would change and the observable that would tell it apart.

Use the facts that the tested lines left open: the persisting failures' templates and their corpus counts (WRITE_CORPUS_RULE), the phases the operators run in and what they receive there (PHASE_INPUTS), the memory directions the candidates produced. These are recommendations with their basis, relayed verbatim to the next diagnosis, which must adopt or decline each one with a reason.