EMMA: STRUCTURED EPISODIC MEMORY FOR MULTI-AGENT WORKFLOW SYNTHESIS

Anonymous authors
Paper under double-blind review

ABSTRACT

Deploying LLM agents in specialized domains often require fine-grained task-specific procedures and knowledge a general model lacks, yet parameter training is costly and non-interpretable, and hand-engineered agent systems shift the cost to expert labor. Episodic memory updates at context level, but single agent adaptation struggles to alleviate cross-task interference, especially with complex tasks such as multi-application navigation or data pool analysis. Although multi-agent evolution alleviates this problem, existing methods lack clear definitions on update rules and rely on the backbone model to decide where and how much the system changes. This *ill-scoped evolution* makes learning follow the backbone's structural preferences rather than the observed failures: one backbone accumulates skills while another spawns agents. We present EMMA (structured Episodic Memory for Multi-Agent workflow synthesis), which treats the configuration of a multi-agent system (controller routing memory, specialists inheriting a generalist prompt, and per-step procedural skills) as episodic memory. EMMA makes the ownership and admissibility of updates explicit: a reflector assigns each diagnosed failure to the scope owning its correction and proposes one edit within it, and backbone-independent guards keep every accepted edit scoped, incremental and bounded. Across KramaBench, AppWorld and BBEH with five backbones and four baselines (ACE, GEPA, EvoMAS and ADAS), EMMA attains the best overall performance on every benchmark, up to 32.5% over the strongest baseline, maintains consistent improvements across backbones, and uses 17–38% fewer training executions than the most efficient baseline.

1 INTRODUCTION

Deploying LLM agents in specialized domains demands more than general capability. Real-world applications such as data-to-insight pipelines and multi-application tasks require tailored conventions and procedures that a general model does not possess (Lai et al., 2026; Trivedi et al., 2024). Two mechanisms are conventionally used to supply such knowledge, and each incurs a distinct cost. Parameter training, whether supervised fine-tuning or reinforcement learning under objectives such as GRPO and DAPO (Shao et al., 2024; Yu et al., 2025), is computationally expensive and yields an artifact that can be neither inspected nor selectively corrected. Human engineering of agent systems avoids that cost but transfers it to expert labor: hand-specified roles and topologies perform well (Du et al., 2024; Hong et al., 2024; Li et al., 2025) but must be re-authored for every new domain.

Episodic memory offers a more efficient mechanism, in which the experience of solving a task is persisted as text and reused on later tasks with the model weights held fixed. Reflexion, A-Mem and MemRL store retrievable reflections, notes and utility-ranked entries (Shinn et al., 2023; Xu et al., 2025; Zhang et al., 2026b), while ACE and GEPA adapt the context of a ReAct-style agent, as an accumulating playbook and as a reflectively optimized prompt (Zhang et al., 2026a; Agrawal et al., 2026). Both use a single generic agent and adapt only its context. Complex applications such as data lake analysis, however, involve thousands of files from various sources (Lai et al., 2026). A single agent cannot specialize its behavior per subtask, and its failure becomes observable only at the end of a long trajectory, so the learning signal cannot be attributed to the step that caused it.

Multi-agent evolution and learning systems supply the missing structure. EvoMAS, EvoAgent, ADAS and AFlow search over agent configurations or programs through mutation, recombination

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.

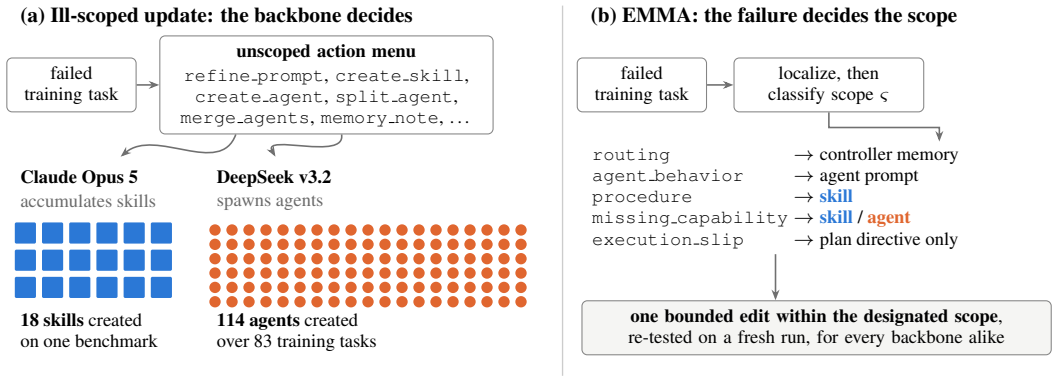

Figure 1: **Ill-scoped versus scoped updates.** (a) Without scope constraints, the update is a backbone habit: Opus 5 accumulates skills, DeepSeek v3.2 repeatedly creates agents (one mark per created artifact). (b) EMMA assigns the diagnosed failure to a scope and makes one bounded edit within it.

or meta-agent proposals guided by a task score (Hu et al., 2026; Yuan et al., 2025; Hu et al., 2025; Zhang et al., 2025b). In these systems, however, the optimizing backbone decides *where* and *how extensively* to change the system: a meta-model proposes a whole new configuration or program, and although EvoMAS constrains each update step to one component per mutation, there is no limit on which component is selected, a choice that relies entirely on the backbone model (Appendix D.1). This *ill-scoped evolution* makes learning heavily dependent on the backbone's own behavior. As demonstrated in Figure 1, without scope constraints, different backbone models have almost orthogonal structural preferences, which overshadow the knowledge actually learned from the task.

We present **EMMA**, structured **E**pisodic **M**emory for **M**ulti-**A**gent workflow synthesis, which makes the *ownership* and *admissibility* of updates explicit. EMMA treats reusable configurations as memory derived from task experience. Its controller selects planning knowledge, agents, and procedural skills, executes one step at a time, and revises unfinished work from intermediate outputs. Training assigns each diagnosed failure to a scope and permits a single corresponding edit. Generalization instructions encourage transferable corrections; programmatic guards enforce scope–action compatibility, preserve existing prompt content, bound memory growth, and require a change of strategy after repeated unsuccessful edits. Specialized agents inherit the current generalist prompt, allowing general improvements to propagate without duplicating them. Our contributions are:

- **Configuration as episodic memory.** We formulate multi-agent workflow synthesis as episodic learning over routing knowledge, inheriting agents and procedural skills.
- **Explicit, guarded updates for consistent update behavior.** Each diagnosed failure is assigned to a scope and corrected by a single guarded edit within it. This keeps reflection focused on the information a task actually provides, instead of superficial structural modifications, repetitive additions, or destructive rewrites of existing entries.
- **Consistent gains across model families.** EMMA achieves the best mean performance on every benchmark, improving over the strongest baseline by up to 32.5%, while ranking first in 17 of 20 settings and using the fewest training executions.

2 PRELIMINARIES

We consider a distribution $\mathcal{Q}$ over tasks. Each task $q \sim \mathcal{Q}$ is a natural-language request issued against an environment and is paired in training with a reference y_q (a gold answer or a set of unit tests). A gate $\mathsf{Gate}(q, \hat{y}) = (v, d)$ returns a binary verdict $v \in \{0, 1\}$ for a produced answer $\hat{y}$ and a short evaluator digest d of why it failed. θ denotes the backbone LLM's parameters and is held *fixed*; the controller, the sub-agents and the reflector are all instantiated from the same θ.

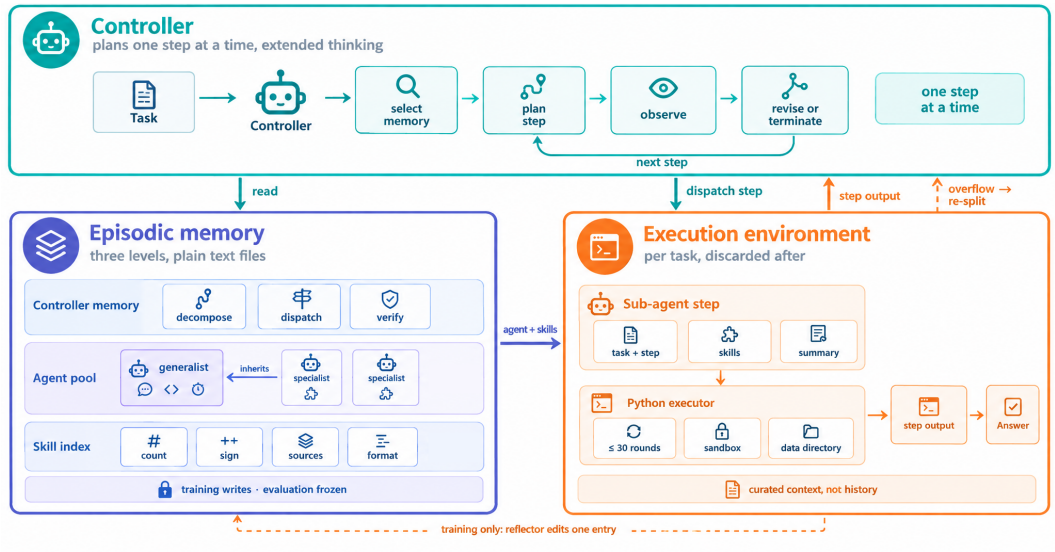

Figure 2: **EMMA architecture.** The controller selects controller-memory entries, plans, and dispatches one step at a time to a sub-agent with its assigned skills; step outputs drive replanning, and a step that overflows its sub-agent is re-split. The three memory levels persist across tasks and are written only during training; the execution environment is created per attempt and discarded.

2.1 Episodic Learning

Episodic learning improves an agent by accumulating experience in an external store of textual entries rather than by updating θ (Shinn et al., 2023; Zhang et al., 2026b). Executing the agent system on q under memory $\mathcal{M}$ is written $\mathsf{Exec}_\theta(\mathcal{M}, q) = (\hat{y}, \tau)$, with final answer $\hat{y}$ and trace τ, and learning is a write $\mathcal{M}' = \mathsf{Update}_\theta(\mathcal{M}, q, \tau, \mathsf{Gate}(q, \hat{y}))$.

Definition 1 (Episodic learning problem)**.** *Given fixed θ, a training set $\mathcal{D}_{\text{train}} = \{(q_i, y_{q_i})\}_{i=1}^{N}$ drawn from $\mathcal{Q}$, and an initial memory $\mathcal{M}_0$, form, using only $\mathcal{D}_{\text{train}}$, a memory*

$$\mathcal{M}^\star \in \arg\max_{\mathcal{M}} \mathbb{E}_{q\sim\mathcal{Q}}[v(q, \mathsf{Exec}_\theta(\mathcal{M}, q))], \tag{1}$$

where $v(q, \cdot)$ is the verdict of the benchmark's official metric. At evaluation the memory is frozen and Update *is never applied to a held-out task.*

With no gradient, *where* and *how much* Update changes the store is a design choice.

2.2 Multi-Agent Workflow with Orchestration

We adopt the orchestrated setting of Magentic-One, in which a lead agent plans, tracks progress and re-plans while directing sub-agents that operate the environment (Fourney et al., 2024), and add the per-step assignment of procedural skills. Let $\mathcal{A}$ be a pool of sub-agents, $\mathcal{S}$ a pool of skills and $\mathcal{C}$ a controller memory (defined in Section 3.1.1). A controller π_c, itself an LLM with parameters θ, solves q by interleaving planning with execution. After $t-1$ steps with history $h_{t-1} = \big((q_1, a_1, S_1, o_1), \ldots, (q_{t-1}, a_{t-1}, S_{t-1}, o_{t-1})\big)$, it emits a subtask, an executing sub-agent, a skill subset and a curated context,

$$(q_t,\ a_t,\ S_t,\ \kappa_t) = \pi_c(q, \mathcal{C}_q, \mathcal{A}, \mathcal{S}, h_{t-1}), \qquad a_t \in \mathcal{A},\ \ S_t \subseteq \mathcal{S}, \tag{2}$$

and the sub-agent returns $o_t = \mathsf{Exec}_\theta(a_t, q, q_t, S_t, \kappa_t)$, where $\mathcal{C}_q \subseteq \mathcal{C}$ is the controller memory loaded for q and κ_t is a controller-authored summary of the prior outputs that step t needs, rather than the full history. The loop continues until π_c terminates and returns $\hat{y}$.

Because step t is conditioned on h_{t-1}, no complete plan is committed in advance, and because S_t is chosen per step, procedural knowledge is a decision of the controller rather than a fixed property of a sub-agent. Magentic-One has no skill layer (Fourney et al., 2024), and multi-agent search methods fix $(a_t)_t$ before any o_t is observed (Hu et al., 2026; 2025; Zhang et al., 2025b).

3 EMMA: STRUCTURED EPISODIC MEMORY FOR MULTI-AGENT WORKFLOW SYNTHESIS

EMMA instantiates Definition 1 with a memory whose entries *are* the configuration of the multi-agent system of Section 2.2. Section 3.1 describes the architecture that reads this memory, Section 3.2 the training procedure that writes it, and Section 3.3 summarizes training and inference. Algorithms 1–3 in Appendix A give the exact inputs and outputs of each step.

3.1 ARCHITECTURE

As shown in Figure 2, EMMA consists of a controller, a persistent episodic memory, and a task-local execution environment. The memory persists across tasks and is written only during training; the execution environment is created for each attempt and discarded afterwards.

3.1.1 HIERARCHICAL EPISODIC MEMORY

The memory consists of three pools at decreasing levels of generality,

$$\mathcal{M} = \big(\underbrace{\mathcal{C}}_{\text{controller memory}}, \underbrace{\mathcal{A}}_{\text{agent pool}}, \underbrace{\mathcal{S}}_{\text{skill pool}} \big). \tag{3}$$

The **controller memory** $\mathcal{C}$ holds rules about how to decompose a task, which agent and skills to dispatch, and when to verify. Each entry carries a topic, a summary, domain tags and a generality label (general, domain-specific or task-specific). Rather than retrieving by similarity, the controller reads this index and chooses which entries to load for the current task; loading none is a valid choice, since irrelevant memory degrades planning.

The **agent pool** $\mathcal{A}$ holds sub-agents, each defined by a system prompt p_j, an input/output contract and a Python tool. Training starts from a single generalist a_0; every other agent is created during training as a *specialization* that inherits the generalist's current prompt,

$$\tilde{p}_j \;=\; p_0 \,\|\, p_j, \tag{4}$$

so a general habit written once in p_0 reaches every specialist; p_j holds only the role-specific method.

The **skill pool** $\mathcal{S}$ holds short procedures for *classes* of sub-problems (an algorithm, a rule system, a pitfall checklist, a format convention), indexed by a one-line "when to use" description. The controller sees only the index and assigns skills per step; the sub-agent receives their full text.

The three levels answer different questions: $\mathcal{C}$, *how the task is divided and routed*; the agent prompts, *how an agent works on any subtask*; and $\mathcal{S}$, *how a class of sub-problem is solved*. This separation gives every lesson a single home, which the training procedure in §3.2 exploits.

3.1.2 STEP-WISE SYNTHESIS WITH OVERFLOW DECOMPOSITION

Given a task, the controller loads its selected memory and emits an ordered plan in which each step names an agent, a subtask and the skills to assign. It dispatches one step at a time (Eq. 2), reads the output, and then continues, revises the remaining steps, or terminates. Each sub-agent receives the task, its subtask, its skills and a controller-curated summary of earlier outputs, which keeps context small while passing on what the step needs.

A step may exhaust its sub-agent, by filling the context window or by running out of rounds. EMMA reads this as evidence that the subtask was specified too coarsely, not that the agent failed, and applies *overflow decomposition*: the controller uses the partial trace to split the unfinished step into smaller steps over existing agents, splices them into the plan, and resumes at the same position with completed work preserved. Every overflow is also recorded as evidence for the reflector; the full procedure is given in Algorithm 2 in Appendix A.

3.2 TRAINING PROCEDURE

Figure 3 presents the training procedure. Each training task is solved, gated, diagnosed and revised in a loop, and every revision is one scoped, guarded edit that is re-tested before the next task.

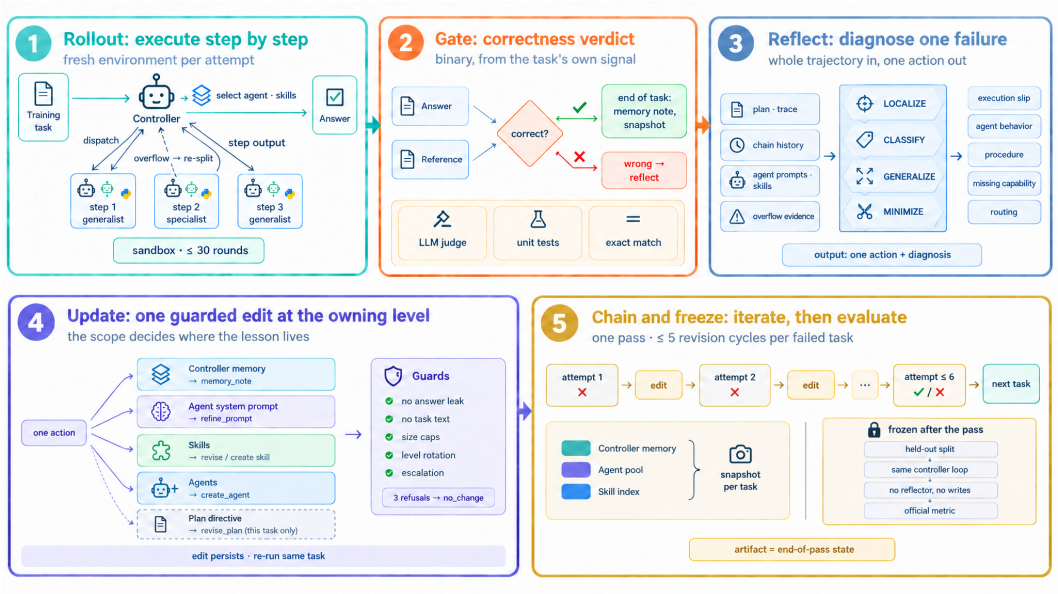

Figure 3: **EMMA training procedure.** (1) Rollout in a fresh environment; (2) a binary gate from the task's own signal; (3) structured reflection over the whole trajectory (localize, classify, generalize, minimize); (4) one guarded edit within the designated scope; (5) re-run until success or the revision budget is spent, then an end-of-task controller update; the memory is frozen after one pass.

Table 1: **Ownership rules: failure scopes z and available actions** $\mathrm{Act}(z)$. A lesson at the wrong level never fires again or contaminates unrelated tasks; the scope–action guard enforces this table.

Scope z	**Meaning**	**Level**	**Available actions** $\mathrm{Act}(z)$
`execution_slip`	knew the method, slipped once	plan directive (this task)	REVISE_PLAN, NO_CHANGE
`agent_behavior`	a general working habit failed	agent prompt	REFINE_PROMPT
`procedure`	method for a class of sub-problems wrong/missing	skill pool	REVISE_SKILL, CREATE_SKILL
`missing_capability`	no agent in the pool can do this work	skills, then agents	CREATE_SKILL, CREATE_AGENT
`routing`	subtasks correct; decomposition/dispatch wrong	controller memory	MEMORY_NOTE, REVISE_PLAN; CREATE_AGENT on overflow

3.2.1 Structured Reflection

Attempt chains. Training processes each task as an *attempt chain*, shown as steps 1–2 and 5 of Figure 3: the task is executed in a fresh environment and gated by its own correctness signal (an LLM judge, unit tests or exact match). A correct answer ends the chain; an incorrect one triggers reflection, one memory edit and a re-run, up to a fixed number of revisions. No edit is made after the final failure, since an edit that is never re-run is untested memory.

What the reflector sees. Unlike a scalar fitness, the reflector receives the whole trajectory of the failed attempt: the plan, the code each sub-agent issued, the evaluator's explanation of the failure, the history of earlier revisions of this task, the full prompts of the agents that ran, the assigned skills, and any overflow events. It may also inspect the full execution log of selected steps.

What it decides. As shown in step 3 of Figure 3, the reflector follows a fixed four-step procedure. It *localizes* the step where the outcome went wrong; *classifies* the failure into one of the five scopes listed in Table 1; *generalizes* the correction so that it applies to every task of the same kind, naming no entity or answer of this one, as the reflector's generalization instructions require; and *minimizes* it to the smallest change within the lesson's designated scope. It then outputs exactly one action. These *ownership rules* make the update scoped: each failure scope $z \in \mathcal{Z}$ has a set of available actions $\mathrm{Act}(z)$, which edit only the part of the memory that owns the correction. A one-off `execution_slip` is never written to durable memory; a failed working habit (`agent_behavior`) is written into a prompt, in p_0 if it is general and in a specialization if it is role-specific; a wrong or missing method (`procedure`) into a skill; and a wrong decomposition or dispatch (`routing`) into controller memory. New agents are reserved for work that no agent can do or that repeatedly overflows one agent.

3.2.2 GUARDED UPDATE

As shown in step 4 of Figure 3, a proposed action is applied only if it passes a set of programmatic guards, which depend on the memory, the task and the revision history but never on the backbone: *scope–action compatibility*, the action is one of the available actions $\mathrm{Act}(z)$; *prompt preservation*, a prompt change keeps most of the existing prompt and may only touch an agent that ran in the failed attempt; *bounded memory growth*, the agent and skill pools and every entry stay within fixed sizes, and a new agent must be a genuine specialization rather than a duplicate or a label; *no leakage*, no artifact may contain the reference answer or copy a span of the task text; and *change of strategy*, no memory level is edited in more than two consecutive failed revisions, and if the same step overflows twice in a row, the action must change how the work is split. A refused proposal returns its reason to the reflector, which tries again; persistent refusals default to no change. A newly created or revised skill or agent is named in the re-run's plan, so the edit is actually exercised.

Together, the scoped update and the guards fix the *envelope* of learning: every accepted edit touches one entry within its designated scope, is incremental, and keeps the memory bounded, whichever backbone proposed it. The backbone decides *which* available edit to make, so a stronger model can still learn better memory, but no backbone can make a larger, misplaced or destructive one, so what is learned follows the training feedback rather than the backbone's habits. Multi-agent search methods lack such an envelope, since the meta-model proposes a whole new configuration; as Figure 1a shows, when the update is ill-scoped even the *kind* of edit becomes a backbone habit.

3.3 TRAINING AND INFERENCE

As stated in Algorithm 1 in Appendix A, training makes a single pass over the training set in a fixed order, running one attempt chain per task; accepted edits persist regardless of the re-run's outcome. After each chain, one chain-aware controller update records a few routing lessons, and the memory is snapshotted. After the pass the memory is frozen, one artifact per benchmark and backbone, with no checkpoint selection. At inference on held-out tasks, the controller keeps memory selection, replanning and overflow decomposition, but no reflector is called and nothing is written.

4 RELATED WORK

LLM-based multi-agent systems. Hand-designed systems such as ChatDev (Qian et al., 2024), MetaGPT (Hong et al., 2024), AutoGen (Wu et al., 2024), debate and mixture-of-agents topologies (Du et al., 2024; Xu et al., 2023; Li et al., 2025) and Magentic-One (Fourney et al., 2024) established role specialization and orchestration, but remain task-specific and carry a coordination cost (Cemri et al., 2025; Tran et al., 2025; Wang et al., 2025a; Zhang et al., 2025a; Li, 2026). Automatic design treats the system as code to search, as in ADAS (Hu et al., 2025) and MAS-GPT (Ye et al., 2025); optimizes its structure, as in GPTSwarm (Zhuge et al., 2024), AFlow (Zhang et al., 2025b), AgentSquare (Shang et al., 2025), AutoAgents (Chen et al., 2024) and generated topologies (Li et al., 2026a); or evolves agents and whole configurations, as in EvoAgent (Yuan et al., 2025) and EvoMAS (Hu et al., 2026). As discussed in Section 1, these methods let the backbone propose whole candidates, with safeguards for executability or novelty rather than for the scope of a change (Appendix D.1), which makes what they learn sensitive to backbone behavior; EMMA instead admits one guarded edit within a diagnosed scope.

Episodic learning and procedural memory. Reflexion (Shinn et al., 2023), TextGrad (Yuksekgonul et al., 2024), MIPRO (Opsahl-Ong et al., 2024) and GEPA (Agrawal et al., 2026) improve agents from natural-language feedback on traces; Dynamic Cheatsheet (Suzgun et al., 2026), A-Mem (Xu et al., 2025), Agent Workflow Memory (Wang et al., 2025b), MemRL (Zhang et al., 2026b) and ACE (Zhang et al., 2026a) accumulate a store, ACE through incremental curation that avoids collapse under wholesale rewriting. Skill libraries study how procedures are authored (Zhou et al., 2026; Liu et al., 2026), selected (Li et al., 2026b) and routed (Zheng et al., 2026; Li, 2026). These serve a fixed, usually single ReAct agent (Yao et al., 2023); EMMA stores a multi-agent configuration and extends incremental updating with a level hierarchy and deterministic guards.

Table 2: **KramaBench** (macro accuracy over 76 held-out tasks) and **BBEH-mini** (accuracy, 368 held-out examples). Best per column in **bold**, second underlined; Mean averages over backbones.

	KramaBench						BBEH-mini					
Method	O-5	S-5	GLM-5	DSv3.2	K2.5	**Mean**	O-5	S-5	GLM-5	DSv3.2	K2.5	**Mean**
ACE	0.694	0.545	0.602	0.478	0.579	0.580	0.780	0.514	0.353	0.492	0.448	0.517
GEPA	0.663	0.509	0.634	0.514	**0.602**	0.584	0.764	**0.658**	0.527	0.501	0.576	0.605
EvoMAS	0.654	0.516	0.600	0.476	0.524	0.554	0.650	0.541	0.478	0.478	0.462	0.522
ADAS	0.709	0.584	0.530	0.481	0.504	0.562	0.690	0.595	0.500	0.516	0.519	0.564
EMMA (ours)	**0.766**	**0.641**	**0.693**	**0.585**	0.586	**0.654**	**0.804**	0.647	**0.617**	**0.563**	**0.685**	**0.663**

Table 3: **AppWorld**, TGC / SGC (%) on `test_normal` (168 tasks) and `test_challenge` (417 tasks). Best in **bold**, second underlined, ranked per metric. For ADAS on DeepSeek v3.2 and Kimi-K2.5, no discovered program beat the hand-written seed, which is reported.

Split	**Method**	O-5	S-5	GLM-5	DSv3.2	K2.5	**Mean**
normal	ACE	83.3 / 75.0	84.5 / 73.2	61.3 / 35.7	65.5 / 48.2	66.7 / 35.7	72.3 / 53.6
	GEPA	92.9 / 87.5	77.4 / 55.4	63.7 / 39.3	41.7 / 7.1	53.6 / 19.6	65.9 / 41.8
	EvoMAS	94.6 / 89.3	69.0 / 44.6	16.7 / 1.8	31.5 / 10.7	44.6 / 30.4	51.3 / 35.4
	ADAS	90.5 / 83.9	67.9 / 53.6	22.0 / 12.5	0.0 / 0.0	3.6 / 0.0	36.8 / 30.0
	EMMA (ours)	**96.4 / 94.6**	**89.9 / 82.1**	**78.6 / 57.1**	**77.4 / 57.1**	**78.6 / 64.3**	**84.2 / 71.0**
challenge	ACE	88.0 / 81.3	80.6 / 65.5	56.1 / 36.0	46.0 / 24.5	**66.2 / 46.0**	67.4 / 50.7
	GEPA	85.1 / 71.9	70.5 / 49.6	44.1 / 23.0	23.5 / 5.0	41.2 / 12.2	52.9 / 32.3
	EvoMAS	89.0 / 78.4	52.3 / 36.0	7.2 / 0.0	11.3 / 1.4	24.5 / 10.1	36.9 / 25.2
	ADAS	84.4 / 71.9	36.2 / 17.3	7.7 / 2.9	0.5 / 0.0	1.2 / 0.0	26.0 / 18.4
	EMMA (ours)	**89.2 / 83.5**	**84.7 / 73.4**	**71.5 / 51.1**	**62.6 / 40.3**	61.2 / 31.6	**73.8 / 56.0**

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

Benchmarks. KramaBench (Lai et al., 2026) (28 train / 76 held-out): data extraction, integration and analysis across six scientific domains; macro accuracy over domains. **AppWorld** (Trivedi et al., 2024) (90 train / 168 `test_normal` / 417 `test_challenge`): tasks across nine applications through APIs, where challenge tasks need an application unseen in training; task-goal (TGC) and scenario-goal completion (SGC, all variants of a scenario must pass) from state-based unit tests. **BBEH-mini** (Kazemi et al., 2025) (92 train / 368 held-out): 23 equally sized reasoning categories; accuracy. Splits are disjoint and shared by all methods (Appendix C).

Backbones and baselines. Claude Opus 5 (Anthropic, 2026a) and Sonnet 5 (Anthropic, 2026b), and the open-weight GLM-5 (GLM-5 Team, 2026), DeepSeek v3.2 (DeepSeek-AI, 2025) and Kimi-K2.5 (Kimi Team, 2026); within a run the controller, sub-agents and reflector share one backbone. We compare with four methods that learn persistent artifacts from the same training split with ground-truth feedback and freeze them (Appendix D): **ACE** (Zhang et al., 2026a), a curated playbook for a fixed ReAct agent; **GEPA** (Agrawal et al., 2026), reflective prompt evolution; **EvoMAS** (Hu et al., 2026), evolution of whole multi-agent configurations rewarded by the dataset evaluator minus a cost term; and **ADAS** (Hu et al., 2025), meta-agent search over agent programs with a budget matched to GEPA. EMMA's budgets, caps and guard settings are listed in Appendix B.

5.2 EXPERIMENTAL RESULTS

As shown in Tables 2 and 3, EMMA attains the **highest mean on all three benchmarks**: $+7.0$ accuracy points over the strongest baseline on KramaBench (0.654 vs. 0.584, GEPA), $+5.8$ on BBEH-mini (0.663 vs. 0.605, GEPA), and $+11.9$ TGC / $+17.4$ SGC on AppWorld normal and $+6.4$ TGC / $+5.3$ SGC on challenge (over ACE). Counting the AppWorld splits separately, EMMA ranks **first in 17 of 20 benchmark–backbone settings** and second in the other three (KramaBench and AppWorld challenge on Kimi-K2.5, BBEH on Sonnet 5); TGC and SGC give the same rankings. The runner-up changes with the benchmark (GEPA on KramaBench and BBEH, ACE on AppWorld), so the advantage does not rest on one weak comparison. The largest gains are on AppWorld challenge with open-weight backbones ($+16.6$ TGC on DeepSeek v3.2, $+15.4$ on GLM-5).

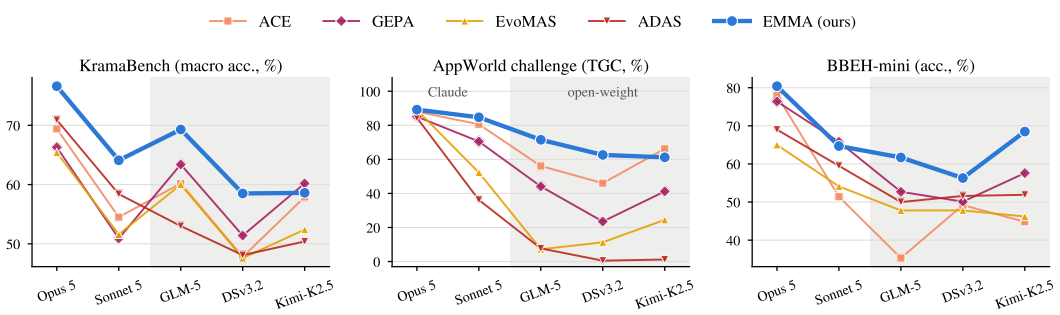

Figure 4: **Cross-family consistency.** Held-out performance per backbone (from Tables 2 and 3); shaded columns are open-weight backbones.

Table 4: **Training task executions**, mean over five backbones (lower is better). Best in **bold**, second underlined.

Method	KB	BBEH	AppW.
ACE	96.8	192.6	283.4
GEPA	156.4	222.8	229.8
EvoMAS	280.0	385.1	1,098.0
ADAS	141.4	200.0	206.0
EMMA (ours)	**61.8**	**159.6**	**127.8**

Table 5: **Ablations on KramaBench with Opus 4.5.** Top: learned memory levels. Bottom: training-split episodic learning vs. test-split adaptation.

Configuration		Acc.
EMMA, agent and skill pools untrained		0.506
EMMA, fixed agent pool (skills + ctrl. memory)		0.591
EMMA (ours), full		**0.647**
Within-task reflection (test)	DACode	0.443
	Reflexion	0.476
	DeepResearch	0.535
Cross-task self-evolution (test)	EvoMAS	0.532
Cross-task episodic (train)	EMMA (ours)	**0.647**

5.2.1 Consistency Across Model Families

Figure 4 shows how each method's held-out performance changes across backbones, and reveals three regularities. **(i) Multi-agent search methods are the most backbone-sensitive.** On AppWorld challenge, EvoMAS and ADAS approach EMMA with Opus 5 (89.0 and 84.4 vs. 89.2 TGC) but fall to 7.2 (EvoMAS, GLM-5) and 0.5 (ADAS, DeepSeek v3.2); on two open-weight backbones ADAS finds no program better than its hand-written seed. When the backbone authors the whole configuration, the result is only as good as that authorship. **(ii) EMMA's weakest backbone is stronger than every baseline's weakest backbone** on every benchmark (58.5 on KramaBench, 61.2 TGC on AppWorld challenge and 56.3 on BBEH), and on KramaBench and AppWorld normal it even exceeds every baseline's five-backbone mean. **(iii) EMMA's margin grows on open-weight backbones** on AppWorld and BBEH: over the strongest baseline per backbone, it leads by $+0.2$ TGC with Opus 5 but $+15.4$ and $+16.6$ with GLM-5 and DeepSeek v3.2 on AppWorld challenge, and by $+2.4$ with Opus 5 but $+9.0$ and $+10.9$ with GLM-5 and Kimi-K2.5 on BBEH. On KramaBench all methods degrade similarly across families, and EMMA's lead is a steady shift of about six points on four of the five backbones.

5.2.2 Update Efficiency

As reported in Table 4, EMMA achieves the best mean performance with the **fewest training executions on every benchmark**, 17–38% fewer than the most efficient baseline. EMMA re-runs only failed tasks, and each re-run tests one localized, guarded edit. This result demonstrates that these updates are on-point: each training execution yields a targeted correction rather than a speculative restructuring, which maximizes learning efficiency over the baselines.

5.3 Ablation Studies

All ablations are conducted on KramaBench with Claude Opus 4.5 (Anthropic, 2025), under which full EMMA attains 0.647; Table 5 reports all compared configurations.

Memory levels. This ablation asks whether treating the multi-agent configuration as memory contributes beyond what a fixed system can learn. Controller memory is learned in every arm, while

the skill and agent pools are either kept at their initial state or learned. Learning procedural skills accounts for the larger of the two gains, which indicates that much of the transferable knowledge concerns how recurring classes of sub-problems are solved. Learning the agent pool as well brings a further gain (0.591 to 0.647), showing that specialized agents capture knowledge that skills alone do not, and supporting our premise that the structure of the system, not only its context, should be learned. Because controller memory is present in all arms, its individual contribution is not isolated.

Episodic learning vs. test-time adaptation. This ablation asks whether the gains stem from memory learned on training tasks rather than from adaptation on the evaluated tasks themselves. We compare EMMA with within-task reflection (DACode, Reflexion and smolagents DeepResearch; Huang et al., 2024; Shinn et al., 2023; Roucher et al., 2025), which revises a solution within one trajectory, and with EvoMAS applied as cross-task self-evolution on the test split without ground truth. All test-time variants remain well below EMMA, indicating that corrections derived from verified training failures transfer to held-out tasks more effectively than additional adaptation at test time. As these arms also differ in feedback access and adaptation split, the comparison supports the transfer of learned memory rather than isolating persistence alone.

Case study: what training produces. In the Sonnet 5 run on BBEH, a single failed zebra puzzle led to three artifacts, each written at the level that owns it: a constraint-satisfaction skill, a specialist agent created from overflow evidence, and a routing rule in controller memory. Subsequent zebra tasks were solved on the first attempt, and a held-out puzzle was solved and verified unique in a single dispatched step (Appendix F). This illustrates how scoped updates distribute one lesson across the memory hierarchy and how the resulting artifacts are reused.

6 Conclusion and Limitations

In this paper, we present EMMA, structured Episodic Memory for Multi-Agent workflow synthesis, which treats the configuration of a multi-agent system (controller routing memory, inheriting agents and procedural skills) as episodic memory and updates it only within the scope of each diagnosed failure, through a single edit that passes backbone-independent guards. The experimental results show that EMMA achieves the highest mean performance on every benchmark and split across five backbones from two model families, ranks first in 17 of 20 benchmark–backbone settings, and uses the fewest training executions, while the multi-agent search baselines degrade sharply on open-weight backbones. These results indicate that scoped, guarded updates let what is learned follow the training feedback rather than the backbone's structural preferences.

Despite these advantages, EMMA narrows but does not eliminate the performance gap between backbone families, and pairing a strong reflector with weaker executor agents is a natural way to exploit this. In addition, accepted edits persist even when an attempt chain ends unsolved; we observed no systematic degradation, but ineffective or harmful entries may remain, and cross-verification on mini-batches or multiple epochs could identify them. We consider both directions as future work.

AI Use Statement

In this work, we used generative AI tools to aid and polish writing, to draft sections of the paper, for retrieval and discovery of related work, and for research execution, specifically to write code for running experiments. We have not used generative AI tools for research ideation, for generating synthetic datasets, or for proving mathematical claims. Separately from this assistance, LLMs appear in this work as objects and components of the study: as the backbones that instantiate the controller, sub-agents and reflector of EMMA and the learned baselines (Section 3 and Appendix D), and as the correctness judge that gates KramaBench training tasks (Appendix B). We have reviewed all AI-assisted work: AI-drafted text was checked and revised by the authors, related work surfaced by AI tools was verified against the original papers, and AI-written experiment code was reviewed and tested by the authors. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

Reproducibility Statement

Section 3 describes the method, and Appendix A states the training loop, the rollout and the guarded reflection step as algorithms. Appendix B lists every budget, cap and guard in the order in which the guards are checked; Appendix E reproduces the full reflector prompt; Appendix C gives the benchmark splits, including the deterministic KramaBench split rule; and Appendix D describes the configuration of each baseline. Learned artifacts from one training run are reproduced in Appendix F.

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, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2507.19457.

Anthropic. System card: Claude Opus 4.5. `https://www.anthropic.com/claude-opus-4-5-system-card`, November 2025.

Anthropic. System card: Claude Opus 5. `https://www-cdn.anthropic.com/c5fbac3f0b1280a933ebd26d3cb8bb9f5bdeaf48/Claude%20Opus%205%20System%20Card.pdf`, July 2026a.

Anthropic. System card: Claude Sonnet 5. `https://www.anthropic.com/claude-sonnet-5-system-card`, June 2026b.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2025. arXiv:2503.13657.

Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje F. Karlsson, Jie Fu, and Yemin Shi. AutoAgents: A framework for automatic agent generation. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 22–30, 2024. arXiv:2309.17288.

DeepSeek-AI. DeepSeek-V3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving Factuality and Reasoning in Language Models through Multiagent Debate. In *International Conference on Machine Learning (ICML)*, 2024.

Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Erkang Zhu, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, Peter Chang, Ricky Loynd, Robert West, Victor Dibia, Ahmed Awadallah, Ece Kamar, Rafah Hosn, and Saleema Amershi. Magentic-One: A Generalist Multi-Agent System for Solving Complex Tasks. *arXiv preprint arXiv:2411.04468*, 2024.

GLM-5 Team. GLM-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2308.00352.

Shengran Hu, Cong Lu, and Jeff Clune. Automated Design of Agentic Systems. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2408.08435.

Yuntong Hu, Yuting Zhang, Matthew Trager, Yi Zhang, Shuo Yang, Wei Xia, and Stefano Soatto. EvoMAS: Evolutionary Generation of Multi-Agent Systems. In *International Conference on Machine Learning (ICML)*, 2026. arXiv:2602.06511.

Yiming Huang, Jianwen Luo, Yan Yu, Yitong Zhang, Fangyu Lei, Yifan Wei, Shizhu He, Lifu Huang, Xiao Liu, Jun Zhao, and Kang Liu. DA-Code: Agent Data Science Code Generation Benchmark for Large Language Models. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2024. arXiv:2410.07331.

Mehran Kazemi, Bahare Fatemi, Hritik Bansal, John Palowitch, Chrysovalantis Anastasiou, Sanket Vaibhav Mehta, Lalit K. Jain, Virginia Aglietti, Disha Jindal, Peter Chen, Nishanth Dikkala, Gladys Tyen, Xin Liu, Uri Shalit, Silvia Chiappa, Kate Olszewska, Yi Tay, Vinh Q. Tran, Quoc V. Le, and Orhan Firat. BIG-Bench Extra Hard. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025. arXiv:2502.19187.

Kimi Team. Kimi K2.5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

Eugenie Lai, Gerardo Vitagliano, Ziyu Zhang, Om Chabra, Sivaprasad Sudhir, Anna Zeng, Anton A. Zabreyko, Chenning Li, Ferdi Kossmann, Jialin Ding, Jun Chen, Markos Markakis, Matthew Russo, Weiyang Wang, Ziniu Wu, Michael J. Cafarella, Lei Cao, Samuel Madden, and Tim Kraska. KramaBench: A Benchmark for AI Systems on Data-to-Insight Pipelines over Data Lakes. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2506.06541.

Dawei Li, Zhen Tan, Peijia Qian, Yifan Li, Kumar Satvik Chaudhary, Lijie Hu, and Jiayi Shen. SMoA: Improving Multi-agent Large Language Models with Sparse Mixture-of-Agents. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, pp. 54–65, 2025.

Shiyuan Li, Yixin Liu, Qingsong Wen, Chengqi Zhang, and Shirui Pan. Assemble your crew: Automatic multi-agent communication topology design via autoregressive graph generation. In *AAAI Conference on Artificial Intelligence (AAAI)*, volume 40, pp. 23142–23150, 2026a. arXiv:2507.18224.

Xiaoxiao Li. When Single-Agent with Skills Replace Multi-Agent Systems and When They Fail. *arXiv preprint arXiv:2601.04748*, 2026.

Yanchao Li, Wanhao Liu, Ben Gao, Jiaqing Xie, Zhehong Ai, Na Zou, Yuqiang Li, and Tianfan Fu. SkillsInjector: Dynamic Skill Context Construction for LLM Agents. *arXiv preprint arXiv:2605.29794*, 2026b.

Yuxuan Liu, Zhaochen Su, Lingyun Xie, Yuhao Zhang, Qing Zong, Jiahe Guo, Zhongwei Xie, Yiyan Ji, Yauwai Yim, Hongyu Luo, Xiyu Ren, Ruan Chenyu, Haoran Li, and Yangqiu Song. SkillRevise: Improving LLM-Authored Agent Skills via Trace-Conditioned Skill Revision. *arXiv preprint arXiv:2606.01139*, 2026.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing Instructions and Demonstrations for Multi-Stage Language Model Programs. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2024. arXiv:2406.11695.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 15174–15186, 2024. arXiv:2307.07924.

Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. smolagents: A smol library to build great agentic systems. `https://github.com/huggingface/smolagents`, 2025.

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. AgentSquare: Automatic LLM Agent Search in Modular Design Space. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2410.06153.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. *arXiv preprint arXiv:2402.03300*, 2024.

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.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic Cheatsheet: Test-time learning with adaptive memory. In *Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, pp. 7080–7106, 2026. arXiv:2504.07952.

Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O'Sullivan, and Hoang D. Nguyen. Multi-agent collaboration mechanisms: A survey of LLMs. *arXiv preprint arXiv:2501.06322*, 2025.

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 *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024. arXiv:2407.18901.

Qian Wang, Zhenheng Tang, Zichen Jiang, Nuo Chen, Tianyu Wang, and Bingsheng He. AgentTaxo: Dissecting and Benchmarking Token Distribution of LLM Multi-Agent Systems. In *ICLR 2025 Workshop on Foundation Models in the Wild (FM-Wild)*, 2025a. OpenReview: `https://openreview.net/forum?id=0iLbiYYIpC`.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent Workflow Memory. In *International Conference on Machine Learning (ICML)*, 2025b. arXiv:2409.07429.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversations. In *Conference on Language Modeling (COLM)*, 2024. arXiv:2308.08155.

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 (NeurIPS)*, 2025. arXiv:2502.12110.

Zhenran Xu, Senbao Shi, Baotian Hu, Jindi Yu, Dongfang Li, Min Zhang, and Yuxiang Wu. Towards reasoning in large language models via multi-agent peer review collaboration. *arXiv preprint arXiv:2311.08152*, 2023.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023. arXiv:2210.03629.

Rui Ye, Shuo Tang, Rui Ge, Yaxin Du, Zhenfei Yin, Siheng Chen, and Jing Shao. MAS-GPT: Training LLMs to build LLM-based multi-agent systems. In *International Conference on Machine Learning (ICML)*, 2025. arXiv:2503.03686.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Yonghui Wu, and Mingxuan Wang. DAPO: An Open-Source LLM Reinforcement Learning System at Scale. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. arXiv:2503.14476.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Dongsheng Li, and Deqing Yang. EvoAgent: Towards Automatic Multi-Agent Generation via Evolutionary Algorithms. In *Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2025. arXiv:2406.14228.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. TextGrad: Automatic "Differentiation" via Text. *arXiv preprint arXiv:2406.07496*, 2024.

Guibin Zhang, Yanwei Yue, Zhixun Li, Sukwon Yun, Guancheng Wan, Kun Wang, Dawei Cheng, Jeffrey Xu Yu, and Tianlong Chen. Cut the Crap: An Economical Communication Pipeline for LLM-based Multi-Agent Systems. In *International Conference on Learning Representations (ICLR)*, 2025a. arXiv:2410.02506.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating Agentic Workflow Generation. In *International Conference on Learning Representations (ICLR)*, 2025b. arXiv:2410.10762.

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 (ICLR)*, 2026a. arXiv: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*, 2026b.

YanZhao Zheng, ZhenTao Zhang, Chao Ma, YuanQiang Yu, JiHuai Zhu, Yong Wu, Tianze Xu, Baohua Dong, Hangcheng Zhu, Ruohui Huang, and Gang Yu. SkillRouter: Skill Routing for LLM Agents at Scale. *arXiv preprint arXiv:2603.22455*, 2026.

Huichi Zhou, Siyuan Guo, Anjie Liu, Zhongwei Yu, Ziqin Gong, Bowen Zhao, Zhixun Chen, Menglong Zhang, Yihang Chen, Jinsong Li, Runyu Yang, Qiangbin Liu, Xinlei Yu, Jianmin Zhou, Na Wang, Chunyang Sun, and Jun Wang. Memento-Skills: Let Agents Design Agents. *arXiv preprint arXiv:2603.18743*, 2026.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In *International Conference on Machine Learning (ICML)*, 2024. arXiv:2402.16823.

A ALGORITHMS

Algorithm 1 states the training loop of Figure 3. Algorithm 2 states ROLLOUT, which executes one attempt under the current memory, and Algorithm 3 states REFLECTGUARDED, which produces one available action for a failed attempt. Notation: a_0 is the generalist and $\tilde{p}_j$ an agent's effective prompt (Eq. 4); a plan $\Pi = (\pi_1, \ldots, \pi_m)$ has steps $\pi_t = (a_t, q_t, S_t)$ (agent, subtask, skills); $\mathrm{idx}(\cdot)$ denotes an index of topics or "when to use" descriptions; ω_t records whether step t overflowed its context or round budget; η is a task-local plan directive; (v, d) is the gate's verdict and evaluator digest; and (z, α, δ) are the reflector's scope, action and payload.

Algorithm 1 EMMA training. Only the memory $\mathcal{M} = (\mathcal{C}, \mathcal{A}, \mathcal{S})$ changes; θ is fixed. One pass, one attempt chain per task, at most one guarded edit per failed attempt.

Require: training tasks $\mathcal{D}_{\mathrm{train}} = \{(q_i, y_i)\}_{i=1}^{N}$; seed memory $\mathcal{M}_0 = (\varnothing, \{a_0\}, \varnothing)$; revisions K
Ensure: frozen memory $\mathcal{M}$

$\mathcal{M} \leftarrow \mathcal{M}_0$
for $i = 1$ **to** N **do** ▷ fixed order, single pass
  $H \leftarrow \langle\,\rangle$; $\eta \leftarrow \varnothing$
  for $k = 0$ **to** K **do**
    $(\hat{y}, \tau, \Omega) \leftarrow \textsc{Rollout}(q_i, \mathcal{M}, \eta)$ ▷ fresh environment; Alg. 2
    $(v, d) \leftarrow \mathsf{Gate}(q_i, \hat{y})$
    if $v = 1$ **or** $k = K$ **then break** ▷ no untested revision after the last failure
    end if
    $x \leftarrow (q_i, d, \Pi, \tau_{\mathrm{code}}, H, \{\tilde{p}_j\}_{a_j \in \tau}, \mathrm{idx}(\mathcal{S}), S_\tau, \Omega)$ ▷ reflection input; §3.2.1
    $(\alpha, \delta, z) \leftarrow \textsc{ReflectGuarded}(x)$ ▷ Alg. 3
    $\mathcal{M} \leftarrow \mathsf{Apply}(\alpha, \delta, \mathcal{M})$; $\eta \leftarrow \textsc{Directive}(\alpha, \delta)$; $H \leftarrow H \oplus (z, \alpha, v)$
  end for
  $\mathcal{C} \leftarrow \textsc{UpdateController}_\theta(\mathcal{C}, q_i, H, v)$ ▷ ≤ 2 entries, pass or fail
  $\textsc{Snapshot}(\mathcal{M})$
end for
return $\mathcal{M}$ ▷ frozen for evaluation

B CONFIGURATION

Table 6 lists the configuration of the final method. Model weights are never updated; the values govern the memory-editing loop and the per-task execution budget. Constants are pre-registered: a change defines a new condition, not a tuning.

Table 6: **Configuration of EMMA.**

Setting	Value
Training passes	1, fixed task order
Training gate	KramaBench: Opus 4.5 judge; AppWorld: unit tests; BBEH: exact match
Revisions per task K / log inspections per revision	5 ($\leq$ 6 executions) / 3
Refusal feedback rounds before forced NO_CHANGE	3
End-of-task controller-memory entries	≤ 2
Pool caps: agents (incl. seed) / skills; chars per prompt / skill	10 / 40; 6,000 / 3,000
Prompt-edit line retention / minimum specialization	0.6 / 150 chars
Task-text leak span	15 words
Level rotation	no level in > 2 consecutive failed cycles
Controller-memory loading budget	15 entries / 12,000 chars
Rounds per agent per step R (round cap = overflow)	30
Plan steps K_Π (final answer step always executed) / re-plan attempts	15 / 3
Executor timeout / child memory limit / captured output	600 s / 24 GiB / 2 MB per stream
Controller / sub-agent reasoning	high (adaptive thinking on Claude 5) / low
Transport retry ladder (KramaBench; BBEH, AppWorld)	12×10 s; 60×10 s

Action vocabulary. Table 7 lists the reflector's actions. The guards of Section 3.2.2 are checked in the following order: scope–action compatibility; change of strategy (escalation after repeated overflow, then level rotation); no leakage (answer in artifacts or directives, verbatim task text); prompt preservation (participant-only edit, line retention, length); bounded memory growth (skill bounds, pool cap, duplicate, locator and label-only agents, dataset or run paths in an artifact).

Algorithm 2 ROLLOUT: controller-mediated execution of one attempt. The plan is mutable; a step that overflows its sub-agent is re-split and re-entered at the same index.

Require: task q; memory $\mathcal{M} = (\mathcal{C}, \mathcal{A}, \mathcal{S})$; directive η; caps K_Π (plan steps), R (rounds per step), R_{re} (re-plans)
Ensure: answer $\hat{y}$; trace τ; overflow evidence Ω

$\mathcal{C}_q \leftarrow \mathsf{Select}_\theta(q, \text{idx}(\mathcal{C}))$ ▷ controller-selected; $\varnothing$ allowed
$\Pi \leftarrow \text{PLAN}_\theta(q, \mathcal{C}_q, \text{idx}(\mathcal{A}), \text{idx}(\mathcal{S}), \eta)$ ▷ $\pi_t = (a_t, q_t, S_t)$
$\tau \leftarrow \langle\rangle$; $\Omega \leftarrow \varnothing$; $t \leftarrow 1$
while $t \le |\Pi|$ **and** $t \le K_\Pi$ **do**
  if $a_t \notin \mathcal{A}$ **then**
    $\Pi \leftarrow \text{REPLAN}_\theta(\Pi, t)$; **continue** ▷ strict pool at training; $\le R_{\text{re}}$ times
  end if
  $\kappa_t \leftarrow \text{CURATE}_\theta(\tau, \pi_t)$
  $o_t \leftarrow \mathsf{Exec}_\theta(\tilde{p}_{a_t}, q, q_t, S_t, \kappa_t)$ ▷ $\le R$ rounds of Python; Eq. 4
  if $\omega_t \neq \varnothing$ **then** ▷ context or round overflow
    $\Omega \leftarrow \Omega \cup \{(t, a_t, \omega_t, q_t, \text{partial}(o_t))\}$
    $\Pi' \leftarrow \text{DECOMPOSE}_\theta(\pi_t, \text{partial}(o_t), \mathcal{A}, \mathcal{S})$
    if $\Pi' \neq \varnothing$ **then**
      $\Pi \leftarrow \Pi_{1:t-1} \oplus \Pi' \oplus \Pi_{t+1:|\Pi|}$; **continue** ▷ splice, re-enter at t
    end if
  end if
  $\tau \leftarrow \tau \oplus (\pi_t, o_t)$
  $dec \leftarrow \text{DECIDE}_\theta(q, \tau, \Pi_{t+1:})$ ▷ continue / revise remaining steps / terminate
  if $dec =$ TERMINATE **then break**
  else if $dec =$ REVISE **then**
    $\Pi \leftarrow \Pi_{1:t} \oplus \text{REVISE}_\theta(q, \tau)$
  end if
  $t \leftarrow t + 1$
end while
return $\text{FINALANSWER}_\theta(\tau)$, τ, Ω

Algorithm 3 REFLECTGUARDED: one available action for a failed attempt.

Require: reflection input x: task, digest, plan, per-round code τ_{code}, chain history H, prompts of participating agents, skill-pool index and injected skills S_τ, overflow evidence Ω
Ensure: action α, payload δ, scope z

fb $\leftarrow \varnothing$
for $r = 1$ **to** 3 **do**
  $(t^\star, z, \alpha, \delta) \leftarrow \mathsf{Reflect}_\theta(x, fb)$ ▷ localize, classify, generalize, minimize; may inspect logs
  why $\leftarrow$ FIRSTVIOLATEDGUARD$(z, \alpha, \delta, \mathcal{M}, H, \Omega)$ ▷ order in Appendix B
  if *why* $= \varnothing$ **then**
    return (α, δ, z)
  end if
  fb $\leftarrow$ *fb* $\oplus$ *why* ▷ refusal reason fed back
end for
return (NO_CHANGE, $\varnothing$, z)

C BENCHMARKS AND SPLITS

Every split is disjoint: no training task appears in any evaluation set, and every system is trained and evaluated on the same split. Split membership is generated once from pinned sources and loaded through a fingerprint-verifying loader that refuses to run on a mismatch. Table 8 summarizes the splits.

KramaBench. The benchmark supplies 104 tasks across six domains and defines no split. Within each domain, tasks are separated into the `easy` and `hard` groups the benchmark labels, sorted by identifier, and every fifth task is assigned to training. The rule is index-based, requires no random seed, and stratifies over both domain and difficulty; the per-domain counts are given in Table 9.

AppWorld. We use the benchmark's own `test_normal` and `test_challenge` splits without modification and train on its 90-task training split. TGC and SGC are computed by the benchmark's state-based unit tests.

Table 7: **Reflector action vocabulary.**

Action	Level	Durable	Payload
NO_CHANGE	none	no	the failure is not addressable by the program
REVISE_PLAN	plan directive	this task only	a directive the next attempt's planner must follow
MEMORY_NOTE	controller memory	yes	topic, one to three sentences, generality label
REFINE_PROMPT	agent prompt	yes	the complete edited prompt of a participating agent
REVISE_SKILL / CREATE_SKILL	skill pool	yes	skill text and its "when to use" description
CREATE_AGENT	agent pool	yes	name, description, specialization (inherits the generalist)

Table 8: **Benchmark splits.**

Benchmark	Total	Train	Eval	Split construction
KramaBench	104	28	76	deterministic, stratified by domain and difficulty
AppWorld	—	90	168 (normal) / 417 (challenge)	upstream splits, unmodified
BBEH-mini	460	92	368	seed 42, 4 train / 16 eval per category

BBEH-mini. 23 categories $\times$ 20 examples $=$ 460, split per category into 4 training and 16 evaluation examples under seed 42, so all categories appear on both sides. Because every category contributes exactly 16 evaluation examples, macro and micro accuracy coincide on this split.

D Baseline Methods

All four baselines in the main comparison learn persistent artifacts from training feedback and freeze them for evaluation. DACode, smolagents DeepResearch and Reflexion appear only in the ablation of Section 5.3.

ACE (Zhang et al., 2026a) grows a playbook of identified bullets for a fixed ReAct agent with code execution. A generator solves the task with the playbook in context, a reflector tags bullets as helpful or harmful, and a curator applies incremental edits; the playbook with the best validation accuracy is frozen. Feedback is answer correctness on KramaBench and, on AppWorld, the no-ground-truth mode in which the reflector sees only unit-test pass/fail and the trace.

GEPA (Agrawal et al., 2026) evolves the prompt texts of a compound system. Candidates are evaluated on training mini-batches; a reflection model reads the traces and rewrites one component; a Pareto front over validation tasks (drawn from the training split) selects survivors. Feedback is the benchmark metric plus the execution traces.

EvoMAS (Hu et al., 2026) evolves complete multi-agent configurations (topology, prompts, models, tools) by meta-model selection, mutation and crossover for three steps, keeping successful configurations in an experience memory. In the main comparison the reward is the dataset evaluator (oracle mode) minus a token-and-latency cost term, and the best configuration and memory are frozen. The separate EvoMAS arm of Table 5 adapts on test tasks through self-reflection without ground truth.

ADAS (Hu et al., 2025) has a meta-agent write complete agent programs in code, evaluates each on the training split, and keeps an archive that conditions the next proposal, with a budget matched to GEPA. The best archived program is frozen; where no discovered program beat the hand-written seed (DeepSeek v3.2 and Kimi-K2.5 on AppWorld), the seed is reported.

D.1 Update Constraints in Learned Baselines

Table 10 compares how each method constrains a single update, as described in the respective papers (Zhang et al., 2026a; Agrawal et al., 2026; Hu et al., 2026; 2025). "Not described" means that the paper specifies no such mechanism. The two multi-agent search methods leave both the locus and the magnitude of a change to the backbone: ADAS checks a proposal only for novelty (two self-reflection steps) and executability (up to five debugging rounds), and EvoMAS restricts a mutation to one component type (prompts, models, tools or topology) but not how much of that component changes. Neither types the update by a diagnosed failure, bounds the resulting system, or checks for

Table 9: **KramaBench per-domain split.**

	archeo.	astro.	biomed.	environ.	legal	wildfire	total
Train	4	4	3	5	7	5	28
Eval	8	8	6	15	23	16	76

leakage. ACE's incremental delta bullets with embedding de-duplication are the closest analogue of an edit guard, but act on a single context level.

Table 10: **How a single update is constrained.**

Method	Unit of update	Where to edit	Bound on edit	Leakage check	Acceptance
ACE	delta bullets in one playbook	single level	incremental; de-duplication	not described	best-validation playbook
GEPA	one component prompt	round-robin scheduler	none (full rewrite)	not described	minibatch improvement
EvoMAS	whole configuration	meta-model	one component type per mutation	not described	evaluator reward − cost
ADAS	whole agent program (code)	meta-agent	none (novelty, debugging)	not described	archived after evaluation
EMMA	one entry at one level	diagnosed scope, as in Table 1	prompt preservation, bounded growth, change of strategy	answer and task text	re-run on a fresh environment

E REFLECTOR PROMPT

The prompt below drives the reflection step. Run-time inputs appear in angle brackets, pre-registered constants are filled in, and the two conditional sections (overflow evidence; escalation) are shown where they appear when present.

```
A training task FAILED its correctness gate. Diagnose the failure from the execution
     trajectory, decide which LEVEL of the program is responsible, and make exactly ONE
     improvement before the task is re-run on a fresh environment.

## The program you are improving
The program is `P + router + {skills}`, plus the ability to add agents. Each level has a
     home for lessons; a lesson written at the wrong level either never fires again or
     poisons unrelated tasks.
- ROUTER (controller memory): how the controller decomposes a task and dispatches subtasks
     to agents and skills. Home for lessons about decomposition, sequencing, which agent or
     skill to use, and verification steps in the plan. Action: memory_note.
- AGENT PROGRAM P (an agent's system prompt): that agent's general working method -- how it
     reads a subtask, plans, executes, checks and reports -- independent of any topic. Home
     for behaviors that would recur on ANY subtask the agent handles. Action: refine_prompt
     (an EDIT of the existing prompt: keep what works, add or amend a few lines).
- SKILLS (procedural modules the router assigns to an agent per step): a self-contained
     method for a recognizable CLASS of sub-problems, written so it transfers to unseen
     instances of that class -- an algorithm, a rule system, a checklist of pitfalls, a
     format convention. Actions: revise_skill (a listed skill covers this class but is wrong
     or incomplete), create_skill (no skill covers it).
- AGENTS (roles): a new agent only when a kind of work needs its own working method and
     context that does not fit any existing agent's P and cannot be expressed as a skill.
     Action: create_agent (rare; the pool is capped at 10).
Non-durable outcomes: revise_plan -- a directive that only the next attempt of THIS task
     follows (a one-off slip, or a different decomposition to try); no_change -- the failure
     is not addressable by the program (e.g. the reference or gate looks wrong).

## Procedure -- follow in order and fill every field of the output
1. LOCALIZE. From the plan, the action trace and the evaluator digest, identify the step and
     agent where the outcome went wrong, or conclude that the plan itself was wrong. Call
     inspect_step only when the code and digest are not diagnostic by themselves.
2. CLASSIFY the scope -- exactly one:
   - execution_slip: the agent knew the right method and slipped once (misread, off-by-one,
     typo); would not recur systematically. -> revise_plan or no_change
   - agent_behavior: a general working habit failed (did not verify, did not read the whole
     input, answered before computing, ignored the requested format, stopped early); recurs
     across topics. -> refine_prompt
   - procedure: the method for this class of sub-problem was wrong or missing (a domain
     algorithm, a rule system, a convention). -> revise_skill if a listed skill covers the
     class, else create_skill
   - missing_capability: a kind of work nobody in the pool can do. -> create_skill
     (preferred) or create_agent (only if it needs its own working method and context)
   - routing: the agents did their subtasks correctly but the decomposition or dispatch was
     wrong (wrong agent, missing step, no verification step, a relevant skill not assigned).
     -> memory_note or revise_plan
```

```
3. GENERALIZE. Phrase the change so it applies to every future task of the same KIND. It
     must not mention this task's entities, values, names, options or answer, and must not
     quote the task text. Prefer principles and procedures over facts about this instance.
4. MINIMIZE. The smallest change, at the lowest level that actually owns the lesson. One
     action.

## Hard rules (violations are refused and you must redo the action)
- Neither artifacts nor plan directives may contain this task's answer; artifacts never
     contain verbatim spans of the task text. A directive that tells the next attempt what
     the answer is teaches nothing and is refused.
- refine_prompt is an edit of an agent that PARTICIPATED in this attempt (their complete
     prompts are shown below): at least 60% of the existing prompt's lines must survive; add
     or amend a few lines; the prompt stays under 6000 characters.
- Level rotation: no level may be used in more than 2 consecutive cycles of one chain. If
     the chain history shows the last 2 cycles at the same level and the task still fails,
     that level is refused now -- move to another level, or use revise_plan / no_change.
- Skills stay under 3000 characters, are procedural and generic to their class, and come
     with a one-line index description saying WHEN to use the skill.
- create_agent is refused when the pool holds 10 agents or the name exists; prefer a skill.
- The declared scope and the action must match the table above.

## Inheritance and targeting (v1.4.6)
- Every created agent RUNS with the generalist's CURRENT prompt followed by its own
     specialization. The prompt shown below for a created agent is ONLY its specialization;
     the generalist's prompt is shown read-only when the generalist did not participate.
- A lesson about a general working habit (verifying, reading the whole input, computing
     before answering, the requested output format, persisting when a first search finds
     nothing, which source to trust) belongs in the GENERALIST's prompt -- refine_prompt on
     `generalist` -- because inheritance carries it to every specialist. This is allowed even
     when the generalist did not participate, provided the failing agent inherits from it.
- A lesson specific to a specialist's role belongs in THAT specialist's specialization --
     refine_prompt on the agent that ran the failing step. Never write a general rule into a
     specialization; never write a role-specific procedure into the generalist.
- create_agent under inheritance: details.system_prompt is ONLY the specialization (the
     generalist's method is inherited automatically).

## Task
<task text>

## Evaluator digest (why the gate failed this attempt)
<EVALUATOR DIGEST>

## Plan (this attempt)
<PLAN OF THIS ATTEMPT>

## Action trace (per step, per round: the code issued -- environment output deliberately
     omitted)
<ACTION TRACE>

## Chain history (this task's prior revision cycles)
<PRIOR REVISION CYCLES OF THIS TASK>

## Agent pool (<n>/10)
<AGENT POOL SUMMARY>

## Current prompts of the agents that participated (complete text -- edit, do not rewrite)
<FULL PROMPTS OF PARTICIPATING AGENTS>

## Skills index (<k>/40; the router assigns these per step)
<SKILL INDEX>

## Skills injected in this attempt (complete text)
<FULL TEXT OF INJECTED SKILLS>

## OVERFLOW evidence (v1.4.8) -- steps that exhausted an agent in this attempt
<one line per overflowed step: step * agent * kind * rounds used * decomposition outcome;
     subtask; partial trace>
With this evidence, create_agent is admissible under scope routing as well as
     missing_capability: a kind of work that repeatedly exhausts one agent's rounds or
     context needs its own agent with a narrower context, and when the same step overflowed
     in two consecutive attempts create_agent is the EXPECTED structural response. A new
     agent must be a genuinely narrower capability: not a duplicate of an existing agent,
     never a "find/locate the files" role (the inputs are given -- no agent searches for
     them), and its prompt must name no paths, datasets or runs.

## ESCALATION -- structural action REQUIRED
Step(s) <step numbers> overflowed in two consecutive attempts, even after mid-task overflow
     decomposition. A prompt or skill edit is not acceptable here: you MUST choose a
     structural action -- create_agent (scope missing_capability) or revise_plan (scope
     routing / execution_slip) that changes how the work is split.
```

```
## Drill-down (optional)
You may call inspect_step(step_index, round_start, round_end) to see the full execution log
     (code + stdout + stderr) for specific rounds. Budget: 3 inspections this cycle.

## Output -- JSON only
~~~json
{
  "diagnosis": {
    "failing_step": <step number or null>,
    "failing_agent": "<agent name or null>",
    "what_went_wrong": "one or two sentences",
    "scope": "execution_slip | agent_behavior | procedure | missing_capability | routing",
    "recurrence": "why this would or would not recur on other tasks of this kind"
  },
  "generality_check": "confirm in one sentence that the change names no task-specific
     entities, values or answers",
  "action": "<action>",
  "details": {...}
}
~~~
Actions and their details:
- no_change: {}
- revise_plan: {"plan_directive": "a concrete directive the next attempt's planner must
     follow (this task only)"}
- memory_note: {"topic": "short topic", "content": "1-3 sentences: the routing/decomposition
     rule", "generality": "general | domain_specific"}
- refine_prompt: {"agent": "existing agent name", "new_prompt": "the COMPLETE edited prompt
     text"}
- revise_skill: {"skill_name": "existing-skill-name", "new_content": "the complete revised
     skill (markdown)", "description": "one line: when to use it"}
- create_skill: {"skill_name": "kebab-case-name", "content": "the skill (markdown)",
     "description": "one line: when to use it", "agent": "existing agent that should use it"}
- create_agent: {"name": "snake_case_name", "description": "one line: what kind of work it
     does", "system_prompt": "its general working method"}
```

F What Training Produces: Learned Artifacts

In the Sonnet 5 BBEH run, one failure chain populated all three levels . A failed training zebra puzzle was classified as `procedure` and produced the skill `zebra-csp-encode-and-verify`; the re-run split the puzzle across many steps and overflowed, and with this evidence the reflector created `csp_solver`, a constraint-satisfaction specialist inheriting the generalist. The task passed on attempt 5, and the end-of-task update wrote a routing rule: *solve large zebra puzzles in a single `csp_solver` step*. The next three zebra training tasks passed on the first attempt. On a held-out puzzle with 52 clues, the controller loaded 8 of 101 entries, dispatched one `csp_solver` step with the skill, and obtained a solution verified unique by a blocking re-solve in 9 rounds. None of the three entries names an entity or answer of the training puzzles.

The artifacts below are the frozen final memory of this run and the corresponding evaluation log.

Controller-memory entry (type `workflow_pattern`, generality `domain_specific`, domain `bbeh_zebra_puzzles`). Two later entries restate the rule, noting that it "succeeded on first attempt with no revision needed" and "scales fine even with 15+ categories per position".

```
For large zebra/logic-grid puzzles (100+ clues), route to a single csp_solver step that
     transcribes, encodes, and solves all clues in one continuous pass rather than splitting
     into transcription/collision-check/build/diagnose/resolve sub-steps -- checkpoint files
     frequently fail to persist across steps, forcing costly re-derivation and wasted rounds
     on unsat-core chasing before eventually re-solving from scratch anyway.
```

Agent specialization `csp_solver` (inherits `generalist`; 31 recorded invocations), appended to the generalist's current prompt at dispatch. Its description reads: "Handles large constraint-satisfaction problems (e.g. many-clue logic-grid/zebra puzzles): incremental model construction, solving, and unsat-core-based debugging, split into small checkpointed batches so no single call needs to do transcription+encoding+solving+diagnosis all at once."

```
You specialize in building and solving LARGE constraint-satisfaction problems (CSP), such as
     logic-grid/zebra puzzles with many categories and many numbered clues, using OR-Tools
     CP-SAT or equivalent.

Working method:
```

```
1. Never attempt to transcribe the source, encode every constraint, solve, AND debug
     infeasibility inside a single call. Split the work yourself into checkpointed phases
     and persist state to files between phases: (a) transcribe categories/values and clue
     text once, save verbatim; (b) build one small set of well-tested helper functions per
     relational pattern (identity, negation, adjacency, ordered-between, at-end,
     fixed-position) and unit-test each on a tiny toy example before using it; (c) encode
     constraints in fixed-size batches (e.g. 25-40 clues at a time), saving the partial
     model/constraint list to a file after each batch so later batches or later steps can
     resume without re-deriving; (d) only after all batches are encoded, solve once.

2. Track your round budget explicitly. If you are more than half your round allowance into a
     phase without finishing it, stop expanding scope, write out your current partial
     artifact (encoded constraints so far, or the current unsat core) to a file, and report
     exactly what is done and what remains, so a follow-up call can continue from that
     checkpoint instead of restarting.

3. If the full model reports INFEASIBLE, do not iteratively hand-drop clues or hand-derive
     chains of inequalities as a substitute for solving. Instead: extract the unsat core
     programmatically (e.g. CP-SAT SufficientAssumptionsForInfeasibility) exactly once, then
     for every clue in that core re-derive its encoding independently from the literal clue
     text, diff it against the original, fix any mismatch, and re-solve the FULL model. Do
     at most one such re-audit cycle per call; if still infeasible after that, checkpoint
     the core and mismatches found and stop rather than manually enumerating chains of hand
     arithmetic.

4. Before reporting a solution as final, re-solve with the found assignment excluded to
     confirm uniqueness. Report clearly which phase you completed, what artifact you saved,
     and what the next call should resume with.
```

Procedural skill `zebra-csp-encode-and-verify`, indexed as "Use when a subtask requires encoding a multi-category zebra/logic-grid puzzle (many numbered clues, many categories, positions in a row/grid) into a CSP and verifying it".

```
## Zebra/logic-grid CSP encoding and verification (CP-SAT or equivalent)

1. UNIQUE NAMES PER CATEGORY: give every value a name unique across ALL categories (e.g.
     prefix with category), never reuse a bare word like 'biking' in two different
     categories (hobby vs activity). Category-value collisions silently corrupt constraints
     and are a common source of false INFEASIBLE.

2. PIN DOWN PRIMITIVES ONCE, THEN REUSE: before encoding any clue, write exactly one helper
     function per relational pattern actually used by the clues (identity/'is the',
     negation/'is not', 'immediately left/right of' as pos(A)+1==pos(B), 'next
     to'/'adjacent' as |pos(A)-pos(B)|==1, 'X is somewhere in between A and B in that order'
     as pos(A)<pos(X)<pos(B), 'at one of the ends', fixed absolute position references like
     'the person at position K'). Test each helper on a tiny 2-3 variable toy example that
     has a known expected SAT/UNSAT outcome before applying it to all clues.

3. ENCODE ALL CLUES IN ONE PASS, VERBATIM: extract literal clue text once, then encode every
     clue directly against its exact wording using only the verified helpers from step 2. Do
     not re-interpret or 'improve' phrasing case by case.

4. TREAT A FIRST INFEASIBLE AS A BUG SIGNAL, NOT A PUZZLE FACT: these puzzles are
     constructed to be satisfiable. If the full model reports INFEASIBLE, do NOT start
     dropping/relaxing individual clues as if some clue is 'wrong' -- that treats an assumed
     defect in the puzzle as ground truth. Instead: (a) recompute the unsat core, (b) for
     every clue in the core, re-derive its encoding from the literal text completely
     independently (fresh helper call, fresh mental parse) and diff against the original
     encoding, (c) specifically re-check direction (which side is 'immediately left' vs
     'right'), identity vs adjacency conflation, and any off-by-one in absolute position
     indices. Fix any mismatch found and re-solve the FULL model (not a dropped-clue
     variant). Only conclude a clue is genuinely unsatisfiable with the rest after this
     re-audit still reproduces the same minimal core with all encodings independently
     re-verified as literal-correct.

5. VERIFY UNIQUENESS ONLY AFTER SAT: once SAT, re-solve with the found full assignment
     excluded (add a blocking clause) to confirm UNSAT (unique solution) before reporting
     any derived attribute as the final answer. If a second solution exists, the model is
     underconstrained -- check for an unencoded or misencoded clue.

6. BUDGET DISCIPLINE FOR LARGE PUZZLES: encoding 100+ clues and solving is one continuous
     task; do not spend rounds re-deriving the same helper twice. If you are more than half
     your round budget in without a completed, verified SAT solve, stop broad exploration
     (e.g. stop trying clue-drop variants) and instead do the targeted re-audit in step 4 on
     the actual unsat core -- this converges faster than guessing which clue to relax.
```

Evaluation trace on held-out task `bbeh_zebra_puzzles-00`.

```
[v143 Memory] controller selected 8/101 entries

[Controller] Decomposing task: bbeh_zebra_puzzles-00

[Controller] Plan: 2 steps

[Controller] Step 1: Solve this 8-position zebra/logic-grid puzzle. There are 8 people in
    positions 1... (agent: csp_solver, skills: ['zebra-csp-encode-and-verify'])

[Agent csp_solver] Completed in 9 rounds

[Controller] Terminating: The csp_solver already produced a fully verified, unique solution
    confirming the person who likes beavers is at position 6. All 52 clues were checked and
    uniqueness was verified by blocking and ...

[Controller] Task completed in 82.3s

  [bbeh_zebra_puzzles-00] correct=1
```

Table 11 summarizes how the three levels cooperate on this task.

Table 11: **Three memory levels on a held-out zebra puzzle.**

Level	Entry	Responsibility	Observed evidence
Controller memory	single `csp_solver` step, one continuous pass	decomposition and dispatch	the plan uses one solving step and repeats the routing rule
Agent configuration	`csp_solver` specialization	model construction, debugging, verification	the specialist completes in 9 rounds and reports a verified solution
Procedural skill	`zebra-csp-encode-and-verify`	clue encoding and correctness checks	the skill is assigned; the trace reports clue checks and a blocking re-solve for uniqueness