𝕏olver: Generalist Reasoning and Problem Solving through Federated Multi-Agent Dynamics and Holistic Experience Learning

Anonymous Authors[1]

Abstract

LLMs solve every problem in two senses of *isolation*: across problems they discard prior reasoning traces, and within a single problem they cannot compare or reuse parallel attempts. We close this gap with 𝕏**olver**, a training-free framework with three contributions. (i) Holistic Experience, a typed formalism mapping four human experience sources—mentorship, prior-self, tool mastery, peer collaboration—onto retrieval, self-retrieval, tool use, and multi-agent solving. (ii) Federated Agent Learning: agents independently solve the full problem and a judge curates a fixed-size top-m ones; subsequent rounds condition on this buffer rather than on free-form peer messages. (iii) A dual-memory inference algorithm coupling episodic and intermediate, with quality-gated insertion and a relevance/recency/novelty eviction rule. Extensive experiments across 14 reasoning, agentic, and coding benchmarks demonstrate that 𝕏olver consistently outperforms not only collaborative baseline methods such as debate, panel discussion, and cascading with identical setups (e.g., agents, tools, and iterations) but also specialized evolving/iterative, memory-augmented, and tool-using agents, including OctoTools, Search-o1, AWorld, OpenHands, OAgents, and Agent S2.5. Even with lightweight backbones (e.g., QWQ-32B), 𝕏olver surpasses state-of-the-art proprietary models, and with stronger backbones (e.g., o3-mini-high), it achieves new state-of-the-art performance across a broad range of benchmarks. We will release all code/data.

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

1. Introduction

Expert human problem-solvers (e.g., Olympiad teams, programming-contest squads, research groups) rarely attempt a hard problem alone or in one shot. They draw on coaches, prior puzzles, tool fluency, peer attempts, and several rounds of refinement, and only then commit to an answer. Frontier large language models, in contrast, solve every problem in *isolation*. *Across problems*, they discard the reasoning trace, tool calls, and verifier feedback the moment a query is answered—so two structurally similar instances are solved from scratch. *Within a problem*, a single forward pass commits to one trajectory; even sampled k times, the model has no channel to compare, score, and reuse partial work. Existing remedies close only one half of this gap. Retrieval and reflection provide cross-problem signal but remain single-agent and single-pass (Lewis et al., 2020; Guu et al., 2020; Shinn et al., 2024; Suzgun et al., 2025; Li et al., 2025b). Tool-augmented agents gain precision but accumulate no reusable trajectory (Lu et al., 2025; 2024; Yao et al., 2023b). Existing multi-agent collaboration frameworks such as debate, panel, and cascaded coordination (Du et al., 2024; Wang et al., 2024a; Wu et al., 2023; Hong et al., 2024) share work within a problem but propagate confidently-wrong intermediate steps (Cemri et al., 2025).

The shared flaw of multi-agent collaboration. Debate, panel discussion, and cascaded handoff differ in surface form but agree on one design choice: agents condition the next round on *raw peer text*. That single substrate produces three different pathologies: debate steers agents toward defending or rebutting initial answers rather than truly investigating the problem, so a confidently-wrong claim becomes a position the team must argue around; panel discussion summarises outputs through a moderator, which under consensus framing tends to flatten the often-minority correct path; cascade conditions every later agent on the previous one's output, so an early mistake frames the chain. The fix is not to patch the pathologies one by one; it is to remove the substrate. Replace raw peer text with a *judge-curated, score-bounded* shared buffer, and the three pathologies dissolve at once.

In order to close the gap in generalist problem solving, we instantiate this principle as 𝕏**olver**, a training-free multi-agent

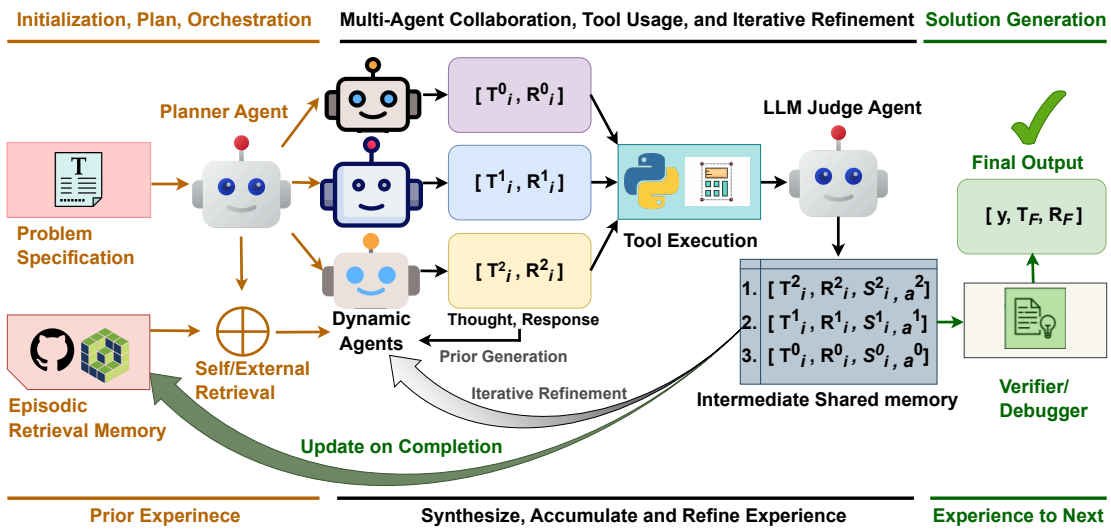

Figure 1. **𝕏olver Scaffold.** At each iteration, agents receive their past reasoning history and top-ranked exemplars to generate new thoughts and responses, using tools (e.g., code) as needed. A judge model ranks outputs, and an intermediate memory maintains the best responses over time. Exemplars are initialized via episodic retrieval and continually updated with high-quality solutions from memory. Iteration stops when convergence or max steps are reached, followed by final verification.

inference framework (Fig. 1) with three contributions. (i) Holistic Experience (HE): a typed formalism that maps the four sources of human problem-solving experience (mentorship, prior-self, tool mastery, peer collaboration) onto concrete inference-time mechanisms (retrieval+judge, parametric self-recall, tool calls etc.,). HE is the missing *vocabulary* for attributing gains to specific sources. (ii) Federated Agent Learning (FAL): a data-parallel replacement for debate/panel/cascade in which every agent independently re-solves the full problem and a judge curates a fixed-size top-m ones (the *only* peer signal a later round ever sees). Independence breaks anchoring; judge-gating filters error; top-m retention preserves diverse strategy lineages.[1] (iii) A dual-memory inference algorithm pairs a dynamic shared memory with an episodic store (cross-problem), governed by an explicit *quality gate* (insert into only if it passes by the judge agents) and a relevance/recency/novelty eviction policy accumulating holistic experiences continuously.

We conduct large-scale experiments across numerous agentic, reasoning, and coding benchmarks—including GSM8K, Math-500, AIME (2024 and 2025), LiveCodeBench (v5), GAIA, BrowseComp, OSWorld, Humanities Last Exams (HLE), SWE-bench Verified (bash only), GPQA Diamond, SQuAD2.0, 2WIKI and Bamboogle—using both proprietary (o3-mini and o4-mini) and open-weight (QWQ-32B and QVQ-72B-preview) backbone models. **𝕏olver** consistently outperforms specialized reasoning systems such as OctoTools, Search-o1, OpenHands, Agentless, WebSailor, OAgents, Alita, AWorld, GTA1, and Agent S2.5. Remarkably, even when instantiated with lightweight models such as QWQ-32 or o3-mini-med, 𝕏olver often outperforms substantially larger state-of-the-art LLMs, including Qwen3-235B, Gemini 2.5 Pro, o3, and o4-mini-high. Moreover, with stronger backbones such as o3-mini-high or o4-mini-high 𝕏olver establishes new state-of-the-art results across all benchmarks. Finally, our comprehensive ablations and analyses reveal how 𝕏olver 's experiential components shape its performance, highlighting both its strengths and limitations.

[1]We use *federated* in the data-parallel sense, not the privacy-FL sense.

2. The 𝕏olver Framework

2.1. Analogical Holistic Experiences

We define the following analogical forms of human learning as holistic experiences: (1) *Mentorship:* Human solvers learn from teachers both via their instructions or lectures as well as from their feedback. We model the former as retrieval from resources external to the model and the later as a judge agent. (2) *Prior own experiences:* Each member leverages their prior studies and knowledge. 𝕏olver maps this as a retrieval from own memory (e.g., self-retrieval via prompting the model itself). (3) *Tool knowledge:* Solvers are capable of tool usage (e.g., a calculator). We also equip the models with different tools, libraries, and apis (e.g., Python) (4) *Peer-collaboration:* In classroom settings, students first attempt problems independently, forming diverse solution strategies, and subsequently learn by observing peers and receiving teacher feedback. In other words, this process involves a "federated solving" and "aggregated refinement".

2.2. Framework Overview

Given a problem query $q \in \mathcal{Q}$ and a pretrained language model $\text{LLM}_\theta(\cdot)$, a conventional approach generates a solution via single-step inference: $y \sim \text{LLM}_\theta(q)$. In contrast,

𝕏**olver** executes a dynamic, multi-agent reasoning process that iteratively accumulates and leverages symbolic experience to solve complex problems more effectively. To support structured collaborative reasoning, 𝕏olver maintains two complementary forms of memory: an *episodic memory* $\mathcal{D}_E$, which stores a library of past problems, solutions, and reasoning traces; and an intermediate dynamic *shared memory* $\mathcal{D}_S$, which evolves during inference to retain high-quality agent trajectories—comprising reasoning thoughts, responses, agent metadata, and feedback. In 𝕏olver, a multi-agent team $\mathcal{A}$ is orchestrated adaptively by a planner agent $\mathcal{P}$, which assigns roles and configures memory access. During inference, $\mathcal{A}$ agents leverage an external toolset $\mathcal{T}$ (e.g., Python interpreter) to support accurate computation. Finally, a verifier or external debugger $\mathcal{V}$ is invoked to extract and format the final answer, and to validate correctness for executable outputs. Below, we first describe the 𝕏olver agents and tools in Section 2.3, followed by the memory components in Section 2.4, and the inference cycle in Section 2.5.

2.3. Agents and Tools

Planner Agent $\mathcal{P}$. The planner agent $\mathcal{P}$ is responsible for initiating, planning, and orchestrating the 𝕏olver multi-agent architecture. Given the problem q and the number of agents m, it constructs a team $\mathcal{A}$ of m dynamic agents, each assigned a distinct expert role (e.g., algebra solver, mathematician, theorist, programmer, algorithm designer) tailored to the demands of q. To ensure sufficient task coverage and role diversity, $\mathcal{P}$ first prompts the underlying LLM to over-generate $M > m$ candidate agents, from which it then selects the most effective subset $\mathcal{A} \subset \{a_1, \ldots, a_M\}$ such that $|\mathcal{A}| = m$. A summary of the most frequently generated and selected roles is provided in Appx. H.15.

Dynamic Reasoning Agents $\mathcal{A}$. The set $\mathcal{A} = \{a^1, a^2, \ldots, a^m\}$ represents a team of dynamic reasoning agents constructed by the planner agent $\mathcal{P}$. Each agent $a^j \in \mathcal{A}$ is assigned a distinct expert role (e.g., algebra solver, programmer, counter-example generator) tailored to the task query q. Agents are instantiated using a standardized prompting template (see Appx. C) that incorporates the task description, assigned role, retrieved examples, prior reasoning attempts, and shared memory feedback—enabling iterative self-correction and role specialization. We note that all agents get the same template prompt but their solution differs due to their strategy reflecting the roles.

At each iteration i, agent a^j receives a context $\mathcal{C}_i^j$ and generates a structured reasoning trace T_i^j and a response R_i^j. For the first iteration ($i = 0$), the context is initialized using the task query and relevant retrieved exemplars:

$$\mathcal{A} \leftarrow \mathcal{C}_0^j = \{q\} \cup \mathcal{R}(\mathcal{D}_E). \qquad \text{(BUILDCONTEXT)}$$

For subsequent iterations ($i \geq 1$), the context evolves by incorporating its prior generation (history) and the shared memory:

$$\mathcal{A} \leftarrow \mathcal{C}_i^j = \{q\} \cup \{T_{i-1}^j, R_{i-1}^j\} \cup \mathcal{D}_S. \quad \text{(BUILDCONTEXT)}$$

Judge Agent $\mathcal{J}$. The judge agent $\mathcal{J}$ evaluates intermediate outputs from each agent and returns structured feedback to guide refinement and memory updates. Given a query q, a reasoning trace T, and a response R, it produces a feedback tuple $S = (T_S, s)$, where T_S is a natural language explanation (e.g., critique, justification, correction), and s is a dynamic quality score. The interpretation of s is task-dependent: for math and other tasks, $s \in [0, 1]$ reflects an LLM-estimated correctness probability; for code tasks, $s \in \{0, 1, \ldots, N_{\text{test}}\}$, where N_{test} denotes the total number of test cases including problem-provided samples and 10 synthesized test cases (LiveCodeBench) generated using AceCode-RM-32B (Zeng et al., 2025). This structured feedback enables agents to identify failures, receive localized corrections, and improve reasoning over iterations.

Verifier Agent $\mathcal{V}$. Due to linguistic complexity and varying answer specification formats, a response may be incorrect even when the underlying reasoning or open-ended response is valid. For instance, answer formats may require multiple-choice letters (e.g., "(A)" or "Choice B"). Such challenges are even more pronounced in code generation tasks, where predicted code may fail to execute or pass any sample test cases. To mitigate this, 𝕏olver includes a Verifier Agent $\mathcal{V}$, which operates differently based on the output type. For math and other problems, $\mathcal{V}$ extracts the final reasoning T_F, response R_F, and answer y from the response associated with the top-ranked entry BESTRESPONSE in $\mathcal{D}_S$, ensuring adherence to the expected output format. For executable code, 𝕏olver invokes an external debugger (e.g., LDB (Zhong et al., 2024)), where $\mathcal{V}$ interacts w/ a Python runtime to get execution feedback & iteratively fix errors.

Tools $\mathcal{T}$. Integrating natural language reasoning with tools like Python execution is a proven way to boost performance on complex reasoning tasks (Moshkov et al., 2025; Toshniwal et al., 2024). We observe that even advanced reasoning models often make mistakes in intermediate steps, particularly when computations become non-trivial. To address this, each dynamic agent a^j is explicitly instructed to use Python execution during reasoning when needed. While for math and code tasks, 𝕏olver limits $\mathcal{T}$ to Python (e.g., numerical libraries, interpreters, debuggers), our prompting strategy is tool-agnostic, allowed us an interface for extensions to richer toolsets such as (e.g., search engines, patch zoomer, clicking, typing, scrolling web pages, image captioner, file handler, knowledge databases, port transferer, reverse engineer) for other tasks. All prompts are 0-shot and provided in Appx. C.

2.4. Memory Components

Episodic Memory $\mathcal{D}_E$. 𝕏olver maintains two forms of episodic (long-term) memory: (1) an external memory corpus $\mathcal{D}_E^{\text{ext}} = \{(q', T', R')\}$, which consists of past problem instances q', their corresponding reasoning traces T' (optional), and solution responses R' (or an adaptive external corpus; see Appx. H.16); and (2) the internal parametric memory encoded in the weights of the agent-specific language model LLM_j.

We define a general retrieval operator $\mathcal{R}(\mathcal{D}_E)$ that returns a set of K examples relevant to the query q. When $\mathcal{D}_E^{\text{ext}}$ is available, retrieval is conducted using similarity-based search (e.g., BM25):

$$\mathcal{R}(\mathcal{D}_E) = \{(q'_k, T'_k, R'_k)\}_{k=1}^{K} \leftarrow \text{Retrieve}_j(q, \mathcal{D}_E^{\text{ext}}).$$

Otherwise, adaptive external retrieval (see Appx. H.16) or 𝕏olver falls back to internal self-retrieval by sampling from the agent model itself:

$$\mathcal{R}(\mathcal{D}_E) = \{(q'_k, T'_k, R'_k)\}_{k=1}^{K} \sim \text{LLM}_j(q).$$

In the case of an external episodic memory, $\mathcal{D}_E$ can also be updated by adding the top-ranked reasoning & response from $\mathcal{D}_S$, paired with the problem q, into the external corpus $\mathcal{D}_E^{\text{ext}}$:

$$\mathcal{D}_E^{\text{ext}} \leftarrow \mathcal{D}_E^{\text{ext}} \cup (q, T, R), \quad \text{(UPDATEEPISODICMEMORY)}$$

where (T, R, S, a) is the top-ranked entry in $\mathcal{D}_S$. To prevent pseudo-correct solutions from polluting the external episodic memory, 𝕏olver promotes an experience from the shared memory $\mathcal{D}_S$ to $\mathcal{D}_E$ only if it passes a conservative quality gate. For executable code, the response must pass all available execution checks, including public tests and synthesized tests when applicable. For math or agentic tasks, the judge score must be a threshold $\tau_q = 1$.

𝕏olver evicts entries with low expected utility. The keep score of an experience e is defined as:

$$\text{Keep}(e) = \alpha \widetilde{s}_e + \beta\, \text{Rel}_W(e) + \gamma \exp\left(-\frac{t_{\text{now}} - t_e}{H}\right) + \delta\, \text{Div}(e) - \rho\, \text{Red}(e),$$

where $\widetilde{s}_e$ is the normalized quality score, $\text{Rel}_W(e)$ is the average relevance of e to a sliding window W of recent queries, H is the temporal decay horizon, $\text{Div}(e)$ rewards coverage of underrepresented task domains, and $\text{Red}(e)$ penalizes near-duplicates solution strategies. The lowest-scoring entries are removed. Further details at Appx. H.16).

Intermediate Shared Memory $\mathcal{D}_S$. The shared memory $\mathcal{D}_S$ maintains a fixed-size set of high-quality intermediate reasoning, responses, and metadata generated by the dynamic agents during inference on the current query q. For simplicity and to preserve the dynamic nature of the framework, we constrain $|\mathcal{D}_S| = m$, where m is the number of dynamic agents in $\mathcal{A}$. Initially, $\mathcal{D}_S \leftarrow \emptyset$. At each iteration i, each agent $a_j \in \mathcal{A}$ produces a reasoning trace T_i^j, response R_i^j, and receives structured feedback $S_i^j = (T_S^{(i,j)}, s_{i,j})$ from the judge agent $\mathcal{J}$, where $T_S^{(i,j)}$ is a natural language explanation and $s_{i,j}$ is a dynamic score reflecting the quality of the tuple (T_i^j, R_i^j). After collecting the new outputs

$$\tau_i^j = (T_i^j, R_i^j, S_i^j, a^j), \quad j = 1, \ldots, m, \quad \text{(RUNAGENTS)}$$

we form the candidate pool $\mathcal{M} = \mathcal{D}_S \cup \{\tau_i^1, \ldots, \tau_i^m\}$. We then update the fixed-size shared memory by keeping only the top-m tuples by score

$$\mathcal{D}_S \leftarrow \text{TopK}\big(\mathcal{M},\ m;\ \text{key}(e) = s(e)\big),, \quad \text{(UPDATESHAREDMEMORY)}$$

where $s(e)$ extracts the dynamic score from $e = (T, R, (T_S, s), a)$.

This replacement mechanism ensures that $\mathcal{D}_S$ always contains exactly m entries with the highest observed scores across all iterations. By maintaining only the strongest reasoning-response-feedback tuples, the shared memory facilitates knowledge transfer between agents and across iterations, enabling collaborative improvement through exposure to diverse high-quality solutions.

2.5. Inference Protocol

Algorithm 1 𝕏olver Inference Protocol

1: **Input:** Query q, Tools $\mathcal{T}$, Episodic Memory $\mathcal{D}_E$, parameters m, k, I
2: **Init:** $\mathcal{D}_S \leftarrow \emptyset$
3: $\mathcal{A} \leftarrow$ PLANNER(q, m)
4: **for** $i = 0$ **to** $\mathcal{I}$ **do**
5: $\quad \{\mathcal{C}_i\}_{c=1}^{m} \leftarrow$ BUILDCONTEXT$(\mathcal{A}, \mathcal{D}_E, \mathcal{D}_S, q, i)$
6: $\quad \{\tau_i^j\}_{j=1}^{m} \leftarrow$ RUNAGENTS$(\mathcal{A}, \mathcal{C}_i, \mathcal{T}, \mathcal{J})$
7: $\quad \mathcal{D}_S \leftarrow$ UPDATESHAREDMEMORY$(\mathcal{D}_S, \{\tau_i^j\})$
8: $\quad$ **if** CONVERGED$(\mathcal{D}_S)$ **then**
9: $\quad\quad$ **break**
10: $\quad$ **end if**
11: **end for**
12: $y \leftarrow \mathcal{V}$(BESTRESPONSE$(\mathcal{D}_S)$)
13: UPDATEEPISODICMEMORY$(\mathcal{D}_E, q, \mathcal{D}_S)$
14: **Return** y

Algorithm 1 summarizes the 𝕏olver inference protocol, which operates in three structured stages. **Stage-1**, which emulates initialization with prior experience, involves the planner constructing a team of agents $\mathcal{A}$ (lines 2–3). **Stage-2**, embodying symbolic *federated* experience accumulation and refinement, iterates for $\mathcal{I}$ rounds (lines 4–10). In each round, all agents receive access to $\mathcal{D}_S$ and $\mathcal{D}_E$, build their symbolic *aggregated* contexts, and generate structured trajectories and responses. ($\mathcal{D}_E$ is only used for context con-

struction at the first iteration). These are evaluated by the judge agent $\mathcal{J}$, and $\mathcal{D}_S$ is updated with the resulting feedback tuples (line 7). Upon convergence[2] or after $\mathcal{I}$ rounds, **Stage-3** invokes the verifier agent $\mathcal{V}$, which extracts the final answer from the top-ranked entry in $\mathcal{D}_S$ (line 11), and updates $\mathcal{D}_E$ with the new experience.

3. Experiments

3.1. Evaluation Setup

Evaluation Benchmarks. We evaluate 𝕏olver on math, coding, open-ended, multimodal, browsing, multihop, and agentic tasks. Math: GSM8K (Cobbe et al., 2021), Math-500 (Hendrycks et al., 2021), AIME 2024 (MAA, 2024), and AIME 2025 (MAA, 2025) (multi-step high-school competition problems). Coding: SWE-bench Verified (bash only) (Jimenez et al., 2023) and LiveCodeBench v5 (Jain et al., 2024) (dynamic, no-leakage). Open-ended: GPQA Diamond (Rein et al., 2024), SQuAD2.0 (Rajpurkar et al., 2018). Multimodal: GAIA (Mialon et al., 2023), HLE (Phan et al., 2025). Tool-integrated browsing: BrowseComp (Wei et al., 2025). Multihop: 2WIKI (Ho et al., 2020), Bamboogle (Press et al., 2022). Agentic: OSWorld (Xie et al., 2024). These span arithmetic, algebra, number theory, geometry, combinatorics, and algorithmic problem solving.

Comparison set. We compare against three groups (Tab. 1 positions 𝕏olver feature-by-feature against the specialized agents): (i) *frontier reasoning models* for context (o3, Gemini 2.5 Pro, o4-mini-high); (ii) the strongest *specialized agent per task family*—Self-Reflexion, OctoTools, Search-o1, CheatSheet, CodeSim, Alita, AWorld, GTA1, Agent S2.5, OpenHands, Agentless, WebSailor, OAgents—each evaluated under our identical backbone; and (iii) three *matched-budget* multi-agent communication schemas (MAD (Du et al., 2024), Panel D. (Wang et al., 2024a), Cascaded (Wu et al., 2023; Hong et al., 2024; Yao et al., 2023b)) re-implemented under the same backbone, planner-selected roles, agents m, rounds $\mathcal{I}$, tool access, retrieval, judge, verifier, and decoding parameters as 𝕏olver —differing *only* in the communication schema; their per-benchmark gaps to federated aggregation are presented in Appx. H.8 (Tab. 13). The feature matrix above shows that the only distinguishing axis among the three rich-comm schemas is the substrate (debate / panel / cascade vs. federated); accordingly we drop those rows from Tab. 1 to avoid restating the matched-budget point.

Baselines and Metrics. We compare 𝕏olver to: (a) proprietary reasoning models—Gemini 2.5 (Pro, Flash Think) (DeepMind, 2025), Grok-3 Beta Think/Mini (Beta) Think (xAI, 2025), Claude 3.7 Sonnet Think (Anthropic, 2025), o1 (OpenAI, 2024), o3-mini/o3/o4-mini (OpenAI, 2025); (b) open-weight LLMs—Qwen3-235B (Qwen Team, 2024), QWQ-32B (Qwen Team, 2025), DeepSeek-R1 (DeepSeek-AI, 2025); (c) specialized models—AlphaOne (Zhang et al., 2025), OpenMathReason (Moshkov et al., 2025), rStar-Math (Guan et al., 2025), rStar-Coder (Liu et al., 2025), OpenCodeReason (Ahmad et al., 2025), Kimi-K1-1.6 (Kimi Team, 2025), Kimi-k2-0905 (Team et al., 2025), Deep Research, ChatGPT Agent, and OpenCua-Qwen2-7B/32B (Wang et al., 2025); and (d) agent/framework baselines—Self-Reflexion (Shinn et al., 2023), Search-o1 (Li et al., 2025b), OctoTools (Lu et al., 2025), CheatSheet (Suzgun et al., 2025), CodeSim (Islam et al., 2025), OpenHands (Wang et al., 2024b), Agentless (Xia et al., 2024), WebSailor (Li et al., 2025a), OAgents (Zhu et al., 2025), Alita (Qiu et al., 2025), AWorld (Yu et al., 2025), GTA1 (Yang et al., 2025), and Agent S2.5 (Agashe et al., 2025). Moreover we implement MAD (Du et al., 2024), Panel Dicussion (Wang et al., 2024a), and Cascaded (Yao et al., 2022; Hong et al., 2024; Wang et al., 2024b; Lei et al., 2024; Wu et al., 2023) communication under *exactly the same backbone, planner-selected agents and rounds, tool access, retrieval mode, verifier, and decoding parameters* as 𝕏olver–differing only in communication schema (full implementation details: Appx. H.8). For (d) we re-implement with the same backbone LLMs as ours; for (a–c) we report official numbers. Metrics: accuracy for math and agentic tasks (judged by o3-pro (OpenAI, 2025)) and *pass@1* for code. In addition, for fair comparison, we reproduce baseline CodeSim using the same verifier LDB as ours. To prevent data leakage, we ensure no solutions are included in our retrieval corpora.

Method	Iter/ Evolving	Mem-within	Mem-cross	Multi-Agent	Judge/ Reflection	Tool-call
Self-Reflexion	✓	✓	✗	✗	✓	✗
OctoTools	✓	✗	✗	✗	✗	✓
Search-o1	✓	✗	✗	✗	✗	✓
CheatSheet	✓	✓	✓	✗	✓	✓
CodeSim	✓	✗	✗	✓	✓	✓
Alita	✓	✓	✗	✗	✗	✓
AWorld	✓	✓	✗	✗	✓	✓
GTA1	✗	✗	✗	✗	✗	✓
Agent S2.5	✓	✓	✗	✓	✗	✓
OpenHands	✓	✓	✗	✗	✗	✓
WebSailor	✓	✗	✗	✗	✗	✓
OAgents	✗	✗	✓	✗	✗	✓
Agentless	✓	✗	✗	✗	✗	✗
Xolver(-)	✓	✓	✗	✓	✓	✓
Xolver(+)	✓	✓	✓	✓	✓	✓

Table 1. **Feature comparison against specialized-agent baselines.** Iter/Evolving: iterative refinement loop where context evolves across rounds; Mem-within: within-problem state persisted across iterations (the dynamic shared $\mathcal{D}_S$ in 𝕏olver); Mem-cross: cross-problem episodic memory accumulation across the test set ($\mathcal{D}_E$); Multi-Agent: ≥ 2 distinct agents collaborate; Judge: *external* scoring agent. Only 𝕏olver combines all five. Communication schemas (debate, panel, cascade, federated) are compared separately under identical agents and iterations in Appx. H.8 (Tab. 13).

[2] 𝕏olver converges when each dynamic agent in $\mathcal{A}$ holds the very same scores (same at max)—terminates when each agent is at max possible scores in the intermediate shared memory $\mathcal{D}_S$ or max iterations $\mathcal{I}$ elapsed.

Inference Details. We evaluate **𝕏olver** across open-weight and proprietary reasoning backbones under the default setting: temperature 0.2, agents $m = 3$, and max iterations $\mathcal{I} = 2$. We report averaged results for stochastic benchmarks and use greedy decoding for simpler or instructional tasks. Unless otherwise stated, 𝕏olver (+) denotes our default variant with dynamically updated episodic memory, while 𝕏olver (–) uses static memory. Full decoding settings, retrieval protocols, benchmark-specific run counts, and implementation details are provided in Appx. B.

3.2. Main Results

Tab. 2 evaluates 𝕏olver across diverse agentic, reasoning, & coding tasks, compared to state-of-the-art LLMs and strongest specialized agents (for full table, see Appx. I). Whereas Appx. H.1 presents results in more open-ended and multi-hop reasoning, showing effectiveness & generality.

Strong Gains Across Benchmarks. Across all datasets, both 𝕏olver (–) and 𝕏olver (+) surpass the backbone LongCoT baselines: for o3-mini-medium, AIME'24 improves 75.8→93.8 and LiveCodeBench 66.3→87.3; for o4-mini-medium, GAIA 54.7→85.6 and HLE 18.1→49.0; for QWQ-32B, AIME'24 78.1→93.6 and LiveCodeBench 63.4→79.2; and for QVQ-72B-preview, GAIA 45.2→77.3 and HLE 11.7→38.5. **Surpassing Prior Agents and Other Multi-Agent Collaboration Methods Under Identical Protocols.** Relative to Search-o1, OctoTools, CheatSheet, Agent S2.5, and AWorld, 𝕏olver (+) delivers consistent gains, exceeding the best baseline by +12.7 on AIME'25 and +13.5 on LiveCodeBench with o3-mini-medium, and by +14.8 on OSWorld and +9.8 on GAIA with o4-mini-medium. Matched-budget MAD/Panel D./Cascaded baselines are presented separately in Appx. H.8 (Tab. 13); across all 11 benchmarks 𝕏olver (–)/(+) exceed MAD by **+14.7/+19.7**, Panel D. by **+13.5/+18.5**, Cascaded by **+11.5/+16.4** avg pp on o3-mini-medium. **In Comparison to Leading LLMs.** Despite weaker backbones, 𝕏olver (+) matches or surpasses frontier models, outperforming o4-mini-high with o3-mini-medium on AIME'24 (93.8 vs. 93.4) and substantially on LiveCodeBench (87.3 vs. 69.5). **Backbone Agnostic** The framework yields consistent benefits across backbones, as 𝕏olver (+) attains 97.1 on GSM8K with o3-mini-medium and 98.0 with QWQ-32B, both markedly above their respective baselines. **Effectiveness of Episodic Memory.** The cross-problem variant 𝕏olver (+) reliably outperforms the problem-specific 𝕏olver (–), averaging +4.8 points across backbones, with the largest gain of +8.2 on BrowseComp using o3-mini-medium. **Scales with Backbone LLM's Strength.** Performance scales w/ stronger backbones—as shown by o3-mini-high (GSM8K 98.1, AIME'24 94.4, AIME'25 93.7, Math-500 99.8, LiveCodeBench 91.6, BrowseComp 71.7, SWE-bench 84.9, GPQA 94.6) and o4-mini-high (GAIA 90.1, OSWorld 74.4, HLE 57.3)—and remains backbone-agnostic, with GSM8K reaching 97.1 on o3-mini-medium and 98.0 on QWQ-32B, collectively setting new SoTA results across all benchmarks.

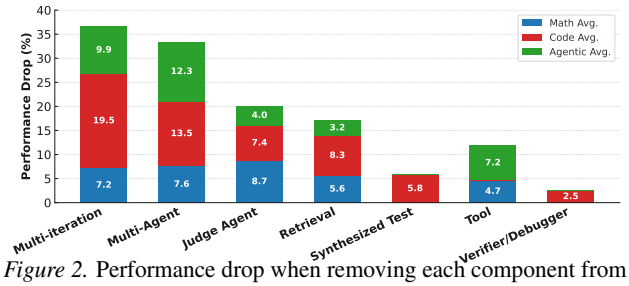

Figure 2. Performance drop when removing each component from 𝕏olver. Bars show average drop on Math (bottom), Code (middle) and Agentic (top).

4. Ablation and Analyses

Quantifying Component Impact. In Fig. 2, we present an ablation study quantifying the contribution of individual components in **𝕏olver** to overall performance, measured by the average performance drop on agentic (Agentic Avg), math reasoning (Math Avg) and programming (LCB) tasks. Each component plays a necessary role, with the most significant degradation observed when removing Multi-iteration and Multi-Agent followed by Judge Agent, highlighting their central importance. In contrast, removing Verifier/Debugger and Tool leads to comparatively smaller drops, suggesting a more auxiliary role in the overall system. Verifier is critical for Math tasks and cannot be removed, while Tool apply only to Math and Agentic tasks, whereas Test only for Code tasks. Likewise retrieval where self-retrieval can also work in-place of external retrieval w/ some drop in accuracy.

Impact of Agent and Tool Count with Iterations, and Emerging Benefits of Collaboration. Fig. 3, performance improves consistently on both AIME '25 and LIVECODEBENCH with more agents or iterations. From Fig. 3, it is evident that for coding tasks, increasing the number of iterations is more crucial than increasing the number of agents. However, for mathematical tasks, both agents and iterations are similarly important. To investigate further, we conducted a more granular experiment on the AIME '25 dataset. We found that, when keeping the total budget (i.e., agents × iterations) constant, increasing the number of agents yields better performance—achieving over 4% improvement compared to increasing iterations. More details as well as the impact of tool count vs. shared memory size (see Fig. 8) on 𝕏olver performance are in Appx. H.3.

Effect of Retrieval Strategies on 𝕏olver Performance. Fig. 9 shows that, performance on both AIME '25 and LIVECODEBENCH follows the trend: *External Retrieval* > *Self-Retrieval* > *No Retrieval*, indicating that external retrieval significantly enhances 𝕏olver's performance. Nevertheless, our dual-memory design gives flexibility about the dependence on retrieval corpora: while external retrieval gives the best results, self-retrieval from the model's own

Backbone	Method	AIME'24	AIME'25	LCB	GPQA	HLE*	Browse Comp	GAIA*	OS World*	SWE-V (bash)
Frontier reference										
	o3	91.6	88.9	–	82.7	20.3	49.7	–	23.0	58.4
	Gemini 2.5 Pro	92.0	86.7	70.4	84.4	21.6	–	–	–	45.0
	o4-mini-high	93.4	92.7	69.5	81.4	18.1	28.3	–	–	–
o3-mini-medium	LongCoT	75.8	70.4	66.3	75.0	18.1	24.4	54.7	31.4	56.2
	Best Spec. Agent	82.2 *CheatSheet*	76.7 *Search-o1*	73.8 *CodeSim*	81.2 *Search-o1*	30.4 *Search-o1*	47.1 *OAgents*	75.8 *AWorld*	59.0 *Agent S2.5*	70.0 *Agentless*
	MAD	80.5	76.8	69.0	77.8	21.3	29.3	57.1	34.0	58.5
	Panel D.	84.4	78.5	68.5	71.3	23.1	33.5	61.0	40.3	60.1
	Cascade	82.5	77.5	74.0	82.0	31.0	32.3	60.5	39.1	61.0
	𝕏olver(–)	**87.2**	**85.1**	**79.6**	**85.4**	**42.6**	**60.7**	**82.4**	**69.9**	**73.9**
	𝕏olver(+)	**93.8**	**89.4**	**87.3**	**89.1**	**49.0**	**68.9**	**85.6**	**73.8**	**81.5**
QWQ-32B	LongCoT	78.1	65.8	63.4	58.1	11.7	17.6	45.2	14.1	30.1
	Best Spec. Agent	84.4 *Search-o1*	72.2 *CheatSheet*	70.5 *CodeSim*	63.6 *Search-o1*	21.2 *Search-o1*	38.5 *OAgents*	67.3 *AWorld*	39.2 *Agent S2.5*	57.4 *Agentless*
	MAD	80.0	76.7	66.5	62.0	13.0	20.4	48.0	17.6	32.2
	Panel D.	82.5	72.5	66.5	61.5	15.1	25.0	52.2	22.6	38.4
	Cascaded	83.1	72.8	66.0	68.1	13.5	19.6	47.5	16.2	33.0
	𝕏olver(–)	**89.9**	**79.5**	**76.2**	**70.1**	**30.9**	**49.3**	**75.6**	**46.4**	**68.2**
	𝕏olver(+)	**93.6**	**82.7**	**79.2**	**75.4**	**38.5**	**57.4**	**77.3**	**50.9**	**71.6**
o3-mini-high	**𝕏olver(+)**	**94.4**	**93.7**	**91.6**	**94.6**	**57.3**	**71.7**	**90.1**	**74.4**	**84.9**

Table 2. **Main results across reasoning, coding, agentic tasks.** 𝕏olver vs. a frontier-model reference and the strongest specialized agent per benchmark; 𝕏olver (–) is the strict single-pass protocol identical to every baseline, 𝕏olver (+) adds sequential cross-problem $\mathcal{D}_E$ updates. *: multimodal subsets use o4-mini-medium / QVQ-72B-preview. *Best Spec. Agent* per column is in superscript.

parametric memory or even no retrieval still gets SOTA results. More details can be found in Appx. H.4.

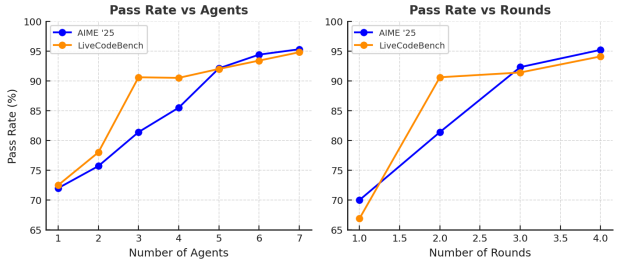

Figure 3. Impact of iterations and agents in 𝕏olver on AIME '25 (QWQ-32B) and LIVECODEBENCH (o3-mini-medium).

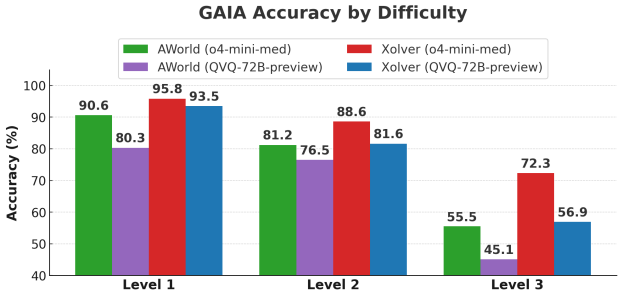

Figure 4. Performance comparison per difficulty levels in GAIA.

Fine-grained Performance Analysis. In Fig. 4, we present fine-grained performance comparisons on GAIA dataset, evaluating 𝕏olver against the agentic baseline framework AWorld. As observed, 𝕏olver consistently outperforms AWorld across all difficulty levels using both QVQ-72B-preview and o4-mini-medium backbones. The performance gap is particularly notable with the open-weight QVQ-72B-preview model and on more challenging problems where AWorld struggles—such as Level 3 problems in GAIA. More details and further fine-grained analysis on MATH-500 (Fig. 10), LCB (Fig. 11) & GPQA DIAMOND (Fig.. 12) have shown in Appx. H.5.

Judge Reliability and Calibration. As the judge gates which trajectories enter $\mathcal{D}_S$ (and via the quality gate, $\mathcal{D}_E$), judge reliability is central to our framework. We surface here the static numbers; the full analysis is in Appx. H.6.

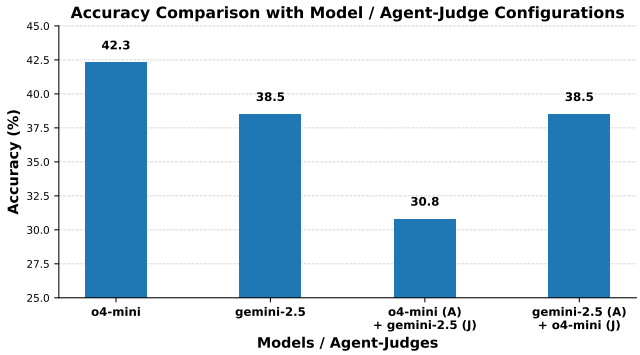

Figure 5. Impact of judge's strength.

Validation and impact of different judges. As validating judges for an agentic task is a separate open research question, our judge only judges the reasoning of the agents. In Fig 5, on 50% of GAIA-level1 validation set, we show how performance changes with different judges. *Calibration vs. humans.* On 300 human-annotated GAIA validation trajectories stratified by difficulty and length, the o3-pro judge achieves **91.8%** agreement with human correctness labels, **90.6%** endpoint accuracy, **88.4%** late-error detection, and **0.041** calibration error—substantially above self-judging (72.4%) and a 3-judge majority is only marginally better (92.5%) at 3× cost (Tab. 5). *Robustness to judge noise.* Injecting random preference flips at rate ρ degrades 𝕏olver gracefully: $\rho = 10\%$ drops GAIA by only **2.7** points, and even at $\rho = 30\%$ 𝕏olver still beats the no-judge baseline by 15+ points (Tab. 9). *Top-m memory protects correct*

trajectories. The default top-m retention reduces the rate of correct-trajectory eviction from 11.9% (top-1) to 5.7% (top-m), and a human-verified gate further drops it to 1.8%—we report the audited number to make the upper bound visible (Tab. 10). *Self-judging is not a substitute.* Letting agents judge themselves drops accuracy by 8.7 / 4 / 7.4 points on math / agentic / code respectively (Appx. H.7), confirming the value of a separate judge.

Cost Analysis and How Long Do 𝕏olver Agents Think? Appx. H.9 report input, reasoning, and output token usage for 𝕏olver across datasets. With m agents and $\mathcal{I}$ iterations, token complexity is $O(m\mathcal{I})$, while runtime is $O(\mathcal{I})$ due to parallel agent execution. This is cheaper than self-consistency (Wang et al., 2022), which typically uses 32–64 generations per example versus our 3×2, and than CheatSheet, whose usefulness estimation requires $O(n^2)$ memory updates, where test size n is usually much larger than m or $\mathcal{I}$. 𝕏olver spends about 25% of tokens on core reasoning and the rest on context sharing and inter-agent communication. Compared with the single-agent Search-o1 baseline, 𝕏olver uses about $1.5\times$ more tokens, offset by substantial performance gains. Details are in Appx. H.9.

Does Data Shuffling Affect 𝕏olver Performance? While we report without any shuffling, shuffling can change the results but minimally (see Tab. 14). Further details can be found at Appx. H.10. **Qualitative Examples.** As in Appx. D, we present examples of full cycle 𝕏olver output on both math and code reasoning tasks demonstrating a full view of how 𝕏olver gradually accumulates.

More Error Analysis. We analyze errors across math, code, and agentic tasks beyond accuracy or pass@1. Although both backbone LLMs generate solution, that are not optimized and hence did not pass. More details as well as further error analysis on math and code (see Fig. 15) and agentic tasks (see Fig. 16) can be found at Appx. H.11.

Dynamics of Reasoning Patterns in 𝕏olver Traces. We study how 𝕏olver adapts its reasoning on LCB by tracking reasoning-pattern frequencies across difficulty tiers (see Tab. 15). Details of pattern collection are in Appx. H.12. As difficulty rises, 𝕏olver increases *self-evaluation* and *exploratory strategies* (e.g., trying new approaches). Correct solutions show reduced rephrasing and subgoal decomposition, indicating more direct, confident reasoning; incorrect ones exhibit more subgoal setup and rephrasing, reflecting failure recognition and restructuring. Unlike OpenCodeReasoning, which stagnates or regresses in key patterns (e.g., self-evaluation), 𝕏olver remains robust and adaptive, aided by multi-agent collaboration and judge feedback, highlighting its generality and flexibility. Moreover, a detailed analysis on 𝕏olver's knowledge acquisition patterns can be found at Appx. E.

Case-Study: How 𝕏olver Enhances Reasoning and Problem Solving To understand how 𝕏olver enhances performance, we conducted case studies on AIME '25 and LiveCodeBench. As shown in our ablation study, at each iteration, the Judge Agent ranks the responses, and in the following round, all agents attempt to match at least one of the previously top-ranked outputs. As shown in Fig. 17 (see Appx. H.13), over time, this leads to improved outputs from the diverse agents (see Appx. H.14), increased agreement among them, and eventual convergence (for why 𝕏olver performs better, we showed some scenarios from GPQA Diamond in Appx. F). Notably, removing the Judge Agent significantly degrades performance, as demonstrated in our first ablation.

5. Related Work

This section concisely reviews related work; full details are in Appx. A.

Memory-Augmented and Retrieval-Augmented LLMs. Prior work spans static retrieval, dynamic reflection, and scratch-space memories, but largely remains single-agent and task-isolated (Lewis et al., 2020; Guu et al., 2020; Shinn et al., 2024; Packer et al., 2023; Nye et al., 2021; Zhao et al., 2023; Shinn et al., 2024; Yao et al., 2023c; Feng et al., 2025). **𝕏olver** extends this line with episodic long-term and dynamic intermediate memories shared across specialized agents. **Multi-Agent Problem Solving.** Existing multi-agent systems improve collaboration, specialization, debate, and code/problem solving, but typically solve isolated tasks without retaining experience (Guo et al., 2024; Li et al., 2023; Chen et al., 2023; Wu et al., 2023; Liu et al., 2023; Islam et al., 2024; 2025; Du et al., 2023; Rahman et al., 2025; Subramaniam et al., 2025; Feng et al., 2025). **𝕏olver** adds judge-mediated selection, dual-memory integration, and continual episodic expansion. **LLM Reasoning Enhancement Techniques.** CoT, self-consistency, tree search, reflection, and retrieval-enhanced reasoning broaden exploration but are mostly single-agent or single-pass (Wei et al., 2022; Wang et al.; Yao et al., 2023a; Shinn et al., 2024; Suzgun et al., 2025; Li et al., 2025b). **𝕏olver** embeds these ideas in a multi-agent loop that supports cross-problem learning. **Tool-Augmented Reasoning.** Tool-using LLMs progressed from browsing and code execution to multi-tool orchestration, yet usually lack persistent, shared experience (Nakano et al., 2021; Gao et al., 2023b; Yao et al., 2023b; Lu et al., 2024; 2025). **𝕏olver** treats tools as memory-enriched shared assets whose outcomes become reusable strategies.

6. Conclusion

We present 𝕏olver, an open-source multi-agent inference framework for experience-aware reasoning. Current limits are higher token usage and dependence on backbone LLMs. To strengthen validity, we will add advanced external verifiers and structured guardrails, moving 𝕏olver toward more efficient, robust, and human-expert-like reasoning.

Impact Statement

This work introduces 𝕏olver, a training-free, generalist reasoning framework that equips large language models with persistent, holistic experience through federated multi-agent collaboration, tool use, and iterative self-refinement. By enabling models to reuse prior reasoning trajectories, evaluations, and feedback—both within and across problem instances—𝕏olver advances the state of the art in reliable reasoning, coding, and agentic task performance without requiring additional model training or parameter updates.

Positive societal impact. 𝕏olver has the potential to substantially improve the robustness and efficiency of AI systems deployed in high-stakes reasoning settings, such as education, scientific discovery, software engineering, and decision support. Its training-free design lowers barriers to adoption, making advanced reasoning capabilities accessible even with lightweight or open-weight models, which may help democratize access to high-quality AI tools. Moreover, the explicit use of shared memory, judge-based evaluation, and verification encourages transparency and error correction, contributing to safer and more accountable AI behavior compared to single-pass or opaque reasoning pipelines.

Risks and limitations. Like other powerful reasoning systems, 𝕏olver could be misused to automate complex tasks in ways that may amplify existing inequalities or enable harmful activities (e.g., generating exploitative code or misinformation at scale). Additionally, its reliance on external tools, retrieval sources, and judge models introduces dependencies that may propagate biases or errors present in those components. While 𝕏olver emphasizes evaluation and verification, it does not guarantee correctness, and failures may still occur in adversarial or poorly specified environments.

Mitigation and responsible use. We mitigate these risks by designing 𝕏olver as a modular framework that allows practitioners to control agent roles, tool access, retrieval sources, and stopping criteria. The use of explicit evaluation signals, verifier agents, and bounded iteration budgets provides practical safeguards against unchecked reasoning loops or tool misuse. We encourage future deployments to pair Xolver with domain-specific constraints, curated retrieval corpora, and human oversight, especially in safety-critical applications.

Overall, we view 𝕏olver as a step toward more reliable, transparent, and generalist AI systems that reason and learn from experience in ways closer to expert human problem solvers, while acknowledging the importance of careful deployment and continued study of its societal implications.

References

Saaket Agashe, Kyle Wong, Vincent Tu, Jiachen Yang, Ang Li, and Xin Eric Wang. Agent s2: A compositional generalist-specialist framework for computer use agents. *arXiv preprint arXiv:2504.00906*, 2025.

Wasi Uddin Ahmad, Sean Narenthiran, Somshubra Majumdar, Aleksander Ficek, Siddhartha Jain, Jocelyn Huang, Vahid Noroozi, and Boris Ginsburg. Opencodereasoning: Advancing data distillation for competitive coding. *arXiv preprint arXiv:2504.01943*, 2025.

Anthropic. Claude 3.7 Sonnet, 2025. URL https://www.anthropic.com/news/claude-3-7-sonnet.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. Why do multi-agent llm systems fail? *arXiv preprint arXiv:2503.13657*, 2025.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chen Qian, Chi-Min Chan, Yujia Qin, Yaxi Lu, Ruobing Xie, et al. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents. *arXiv preprint arXiv:2308.10848*, 2(4):6, 2023.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*, a.

Zhipeng Chen, Kun Zhou, Beichen Zhang, Zheng Gong, Xin Zhao, and Ji-Rong Wen. Chatcot: Tool-augmented chain-of-thought reasoning on chat-based large language models. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, b.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Google DeepMind. Gemini 2.5, 2025. URL https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/.

DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. *arXiv preprint arXiv:2305.14325*, 2023.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. In *Forty-first international conference on machine learning*, 2024.

Shangbin Feng, Wenxuan Ding, Alisa Liu, Zifeng Wang, Weijia Shi, Yike Wang, Zejiang Shen, Xiaochuang Han, Hunter Lang, Chen-Yu Lee, Tomas Pfister, Yejin Choi, and Yulia Tsvetkov. When one llm drools, multi-llm collaboration rules. *ArXiv*, abs/2502.04506, 2025. URL https://api.semanticscholar.org/CorpusID:276235808.

Z. Fountas, Martin A Benfeghoul, Adnan Oomerjee, Fenia Christopoulou, Gerasimos Lampouras, Haitham Bou-Ammar, and Jun Wang. Human-like episodic memory for infinite context llms. *ArXiv*, abs/2407.09450, 2024. URL https://api.semanticscholar.org/CorpusID:271162171.

Chen Gao, Xiaochong Lan, Zhi jie Lu, Jinzhu Mao, Jing Piao, Huandong Wang, Depeng Jin, and Yong Li. S3: Social-network simulation system with large language model-empowered agents. *ArXiv*, abs/2307.14984, 2023a. URL https://api.semanticscholar.org/CorpusID:260202947.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR, 2023b.

Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. rstar-math: Small llms can master math reasoning with self-evolved deep thinking, 2025. URL https://arxiv.org/abs/2501.04519.

Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint arXiv:2402.01680*, 2024.

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. REALM: Retrieval-augmented language model pre-training. In *International Conference on Machine Learning*. JMLR.org, 2020. URL https://dl.acm.org/doi/abs/10.5555/3524938.3525306.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. In J. Vanschoren and S. Yeung, editors, *Proceedings of the Neural Information Processing*

Systems Track on Datasets and Benchmarks, volume 1, 2021.

Xanh Ho, Anh-Khoa Duong Nguyen, Saku Sugawara, and Akiko Aizawa. Constructing a multi-hop qa dataset for comprehensive evaluation of reasoning steps. *arXiv preprint arXiv:2011.01060*, 2020.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 3(4):6, 2023.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. International Conference on Learning Representations, ICLR, 2024.

Jie Huang and Kevin Chen-Chuan Chang. Towards reasoning in large language models: A survey. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 1049–1065, 2023.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. MapCoder: Multi-agent code generation for competitive problem solving. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4912–4944, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.269. URL https://aclanthology.org/2024.acl-long.269/.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. CodeSim: Multi-agent code generation and problem solving through simulation-driven planning and debugging. In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 5113–5139, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-195-7. doi: 10.18653/v1/2025.findings-naacl.285. URL https://aclanthology.org/2025.findings-naacl.285/.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swebench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. In *The Eleventh International Conference on Learning Representations*.

Kimi Team. Kimi k1.5: Scaling reinforcement learning with llms, 2025.

Md Tahmid Rahman Laskar, Sawsan Alqahtani, M Saiful Bari, Mizanur Rahman, Mohammad Abdullah Matin Khan, Haidar Khan, Israt Jahan, Amran Bhuiyan, Chee Wei Tan, Md Rizwan Parvez, Enamul Hoque, Shafiq Joty, and Jimmy Huang. A systematic survey and critical review on evaluating large language models: Challenges, limitations, and recommendations. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 13785–13816, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.764. URL https://aclanthology.org/2024.emnlp-main.764/.

Bin Lei, Yuchen Li, Yiming Zeng, Tao Ren, Yi Luo, Tianyu Shi, Zitian Gao, Zeyu Hu, Weitai Kang, and Qiuwu Chen. Infant agent: A tool-integrated, logic-driven agent with cost-effective api usage. *arXiv preprint arXiv:2411.01114*, 2024.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36:51991–52008, 2023.

Kuan Li, Zhongwang Zhang, Huifeng Yin, Liwen Zhang, Litu Ou, Jialong Wu, Wenbiao Yin, Baixuan Li, Zhengwei Tao, Xinyu Wang, et al. Websailor: Navigating super-human reasoning for web agent. *arXiv preprint arXiv:2507.02592*, 2025a.

Xiaoxi Li, Guanting Dong, Jiajie Jin, Yuyao Zhang, Yujia Zhou, Yutao Zhu, Peitian Zhang, and Zhicheng Dou. Search-o1: Agentic search-enhanced large reasoning models. *arXiv preprint arXiv:2501.05366*, 2025b.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Yifei Liu, Li Lyna Zhang, Yi Zhu, Bingcheng Dong, Xudong Zhou, Ning Shang, Fan Yang, and Mao Yang. rstar-coder: Scaling competitive code reasoning with a large-scale verified dataset. *arXiv preprint arXiv:2505.21297*, 2025.

Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. Dynamic llm-agent network: An llm-agent collaboration framework with agent team optimization. *arXiv preprint arXiv:2310.02170*, 2023.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. Chameleon: Plug-and-play compositional reasoning with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

Pan Lu, Bowen Chen, Sheng Liu, Rahul Thapa, Joseph Boen, and James Zou. Octotools: An agentic framework with extensible tools for complex reasoning. *arXiv preprint arXiv:2502.11271*, 2025.

MAA. American invitational mathematics examination - aime. In *American Invitational Mathematics Examination - AIME 2024*, February 2024. URL https://maa.org/math-competitions/american-invitational-mathematics-examination-aime.

MAA. American invitational mathematics examination - aime. In *American Invitational Mathematics Examination - AIME 2025*, February 2025. URL https://maa.org/math-competitions/american-invitational-mathematics-examination-aime.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594, 2023.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*, 2023.

Ali Modarressi, Ayyoob Imani, Mohsen Fayyaz, and Hinrich Schütze. Ret-llm: Towards a general read-write memory for large language models. *ArXiv*, abs/2305.14322, 2023. URL https://api.semanticscholar.org/CorpusID:258841042.

Ivan Moshkov, Darragh Hanley, Ivan Sorokin, Shubham Toshniwal, Christof Henkel, Benedikt Schifferer, Wei Du, and Igor Gitman. Aimo-2 winning solution: Building state-of-the-art mathematical reasoning models with openmathreasoning dataset, 2025. URL https://arxiv.org/abs/2504.16891.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

OpenAI. Learning to reason with llms, 2024. URL https://openai.com/index/learning-to-reason-with-llms/.

OpenAI. Introducing openai o3 and o4-mini, 2025. URL https://openai.com/index/introducing-o3-and-o4-mini/.

Charles Packer, Vivian Fang, Shishir G. Patil, Kevin Lin, Sarah Wooders, and Joseph Gonzalez. Memgpt: Towards llms as operating systems. *ArXiv*, abs/2310.08560, 2023. URL https://api.semanticscholar.org/CorpusID:263909014.

Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*, 2023.

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

Long Phan, Alice Gatti, Ziwen Han, Nathaniel Li, Josephina Hu, Hugh Zhang, Chen Bo Calvin Zhang, Mohamed Shaaban, John Ling, Sean Shi, et al. Humanity's last exam. *arXiv preprint arXiv:2501.14249*, 2025.

Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah A Smith, and Mike Lewis. Measuring and narrowing the compositionality gap in language models. *arXiv preprint arXiv:2210.03350*, 2022.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2023.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*, 2023.

Jiahao Qiu, Xuan Qi, Tongcheng Zhang, Xinzhe Juan, Jiacheng Guo, Yifu Lu, Yimin Wang, Zixin Yao, Qihan Ren, Xun Jiang, et al. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution. *arXiv preprint arXiv:2505.20286*, 2025.

Qwen Team. Qwq: Reflect deeply on the boundaries of the unknown, 2024. URL https://qwenlm.github.io/blog/qwq-32b-preview/.

Qwen Team. QwQ-32B: Embracing the power of reinforcement learning, March 2025. URL https://qwenlm.github.io/blog/qwq-32b/.

Salman Rahman, Sheriff Issaka, Ashima Suvarna, Genglin Liu, James Shiffer, Jaeyoung Lee, Md Rizwan Parvez, Hamid Palangi, Shi Feng, Nanyun Peng, et al. Ai debate aids assessment of controversial claims. *arXiv preprint arXiv:2506.02175*, 2025.

Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for squad. *arXiv preprint arXiv:1806.03822*, 2018.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*, 2024.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36: 68539–68551, 2023.

Yunfan Shao, Linyang Li, Junqi Dai, and Xipeng Qiu. Character-llm: A trainable agent for role-playing. *ArXiv*, abs/2310.10158, 2023. URL https://api.semanticscholar.org/CorpusID:264145862.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

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, 2024.

Vighnesh Subramaniam, Yilun Du, Joshua B. Tenenbaum, Antonio Torralba, Shuang Li, and Igor Mordatch. Multiagent finetuning: Self improvement with diverse reasoning chains. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=JtGPIZpOrz.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. *arXiv preprint arXiv:2504.07952*, 2025.

Jihoon Tack, Jaehyung Kim, Eric Mitchell, Jinwoo Shin, Yee Whye Teh, and Jonathan Richard Schwarz. Online adaptation of language models with a memory of amortized contexts. *ArXiv*, abs/2403.04317, 2024. URL https://api.semanticscholar.org/CorpusID:268264809.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

Shubham Toshniwal, Ivan Moshkov, Sean Narenthiran, Daria Gitman, Fei Jia, and Igor Gitman. Openmathinstruct-1: A 1.8 million math instruction tuning dataset. *arXiv preprint arXiv:2402.10176*, 2024.

Qineng Wang, Zihao Wang, Ying Su, Hanghang Tong, and Yangqiu Song. Rethinking the bounds of llm reasoning: Are multi-agent discussions the key? In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6106–6131, 2024a.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024b.

Xinyuan Wang, Bowen Wang, Dunjie Lu, Junlin Yang, Tianbao Xie, Junli Wang, Jiaqi Deng, Xiaole Guo, Yiheng Xu, Chen Henry Wu, et al. Opencua: Open foundations for computer-use agents. *arXiv preprint arXiv:2508.09123*, 2025.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Huai hsin Chi, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *ArXiv*, abs/2203.11171, 2022.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Jason Wei, Zhiqing Sun, Spencer Papay, Scott McKinney, Jeffrey Han, Isa Fulford, Hyung Won Chung, Alex Tachard Passos, William Fedus, and Amelia Glaese. Browsecomp: A simple yet challenging benchmark for browsing agents. *arXiv preprint arXiv:2504.12516*, 2025.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 3(4), 2023.

xAI. Grok 3 beta — the age of reasoning agents, 2025. URL https://x.ai/news/grok-3.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

Yan Yang, Dongxu Li, Yutong Dai, Yuhao Yang, Ziyang Luo, Zirui Zhao, Zhiyuan Hu, Junzhe Huang, Amrita Saha, Zeyuan Chen, et al. Gta1: Gui test-time scaling agent. *arXiv preprint arXiv:2507.05791*, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*, 2023a.

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)*, 2023b.

Weiran Yao, Shelby Heinecke, Juan Carlos Niebles, Zhiwei Liu, Yihao Feng, Le Xue, Rithesh Murthy, Zeyuan Chen, Jianguo Zhang, Devansh Arpit, Ran Xu, Phil L Mui, Haiquan Wang, Caiming Xiong, and Silvio Savarese. Retroformer: Retrospective large language agents with policy gradient optimization. *ArXiv*, abs/2308.02151, 2023c. URL https://api.semanticscholar.org/CorpusID:260611249.

Chengyue Yu, Siyuan Lu, Chenyi Zhuang, Dong Wang, Qintong Wu, Zongyue Li, Runsheng Gan, Chunfeng Wang, Siqi Hou, Gaochi Huang, et al. Aworld: Orchestrating the training recipe for agentic ai. *arXiv preprint arXiv:2508.20404*, 2025.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. Acecoder: Acing coder rl via automated test-case synthesis. *arXiv preprint arXiv:2502.01718*, 2025.

Junyu Zhang, Runpei Dong, Han Wang, Xuying Ning, Haoran Geng, Peihao Li, Xialin He, Yutong Bai, Jitendra Malik, Saurabh Gupta, and Huan Zhang. Alphaone: Reasoning models thinking slow and fast at test time. 2025. URL https://arxiv.org/abs/2505.24863.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Gaetan Lin, Y. Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *AAAI Conference on Artificial Intelligence*, 2023. URL https://api.semanticscholar.org/CorpusID:261048772.

Li Zhong, Zilong Wang, and Jingbo Shang. Ldb: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*, 2024.

Wanjun Zhong, Lianghong Guo, Qi-Fei Gao, He Ye, and Yanlin Wang. Memorybank: Enhancing large language models with long-term memory. *ArXiv*, abs/2305.10250, 2023. URL https://api.semanticscholar.org/CorpusID:258741194.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V Le, et al. Least-to-most prompting enables complex reasoning in large language models. In *The Eleventh International Conference on Learning Representations*.

He Zhu, Tianrui Qin, King Zhu, Heyuan Huang, Yeyi Guan, Jinxiang Xia, Yi Yao, Hanhao Li, Ningning Wang, Pai Liu, et al. Oagents: An empirical study of building effective agents. *arXiv preprint arXiv:2506.15741*, 2025.

Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, Y. Qiao, Zhaoxiang Zhang, and

Jifeng Dai. Ghost in the minecraft: Generally capable agents for open-world environments via large language models with text-based knowledge and memory. *ArXiv*, abs/2305.17144, 2023. URL https://api.semanticscholar.org/CorpusID:258959262.

A. Related Work

Memory-augmented language models have evolved from static retrieval systems like RAG (Lewis et al., 2020) and REALM (Guu et al., 2020) to more dynamic approaches such as self-reflective memory in Reflexion (Shinn et al., 2024), hierarchical memory in MemGPT (Packer et al., 2023), and working memory in Scratchpads (Nye et al., 2021). Recent advances explore diverse memory forms from textual (Zhong et al., 2023; Modarressi et al., 2023) to parametric encodings (Tack et al., 2024; Shao et al., 2023), with sophisticated operations for memory management (Gao et al., 2023a; Zhu et al., 2023). However, these systems share fundamental limitations: they operate on single tasks in isolation, lack persistent cross-problem experience accumulation, employ single-agent architectures, and discard valuable reasoning traces after each problem. 𝕏**olver** addresses these gaps through a novel dual-memory architecture that combines episodic long-term memory (Fountas et al., 2024) with dynamic intermediate memory, enabling multiple specialized agents to collectively build, share, and iteratively refine experiential knowledge. While prior work has explored cross-trial information sharing (Zhao et al., 2023; Shinn et al., 2024) and multi-source memory integration (Yao et al., 2023c; Park et al., 2023), these approaches remain confined to single-agent settings. Unlike existing methods where problems are solved independently, our framework creates a persistent knowledge base through multi-agent collaboration (Feng et al., 2025). This allows agents to accumulate expertise from every solved problem and leverage collective experience for future tasks, shifting from isolated inference to collaborative experience-driven reasoning.

Multi-Agent Problem Solving. Multi-agent LLM systems emerge from the fundamental limitation that single models cannot adequately represent diverse real-world data, skills, and perspectives, with collaborative approaches offering improved reliability through cross-validation and efficiency via task specialization (Guo et al., 2024; Feng et al., 2025). Early frameworks like CAMEL (Li et al., 2023) introduced role-playing agents focusing on two-agent assistant-user interactions with fixed role assignments. In contrast, AgentVerse (Chen et al., 2023) advanced this field by enabling dynamic role adjustment and flexible multi-agent groups that adapt based on task requirements. More advanced systems such as AutoGen (Wu et al., 2023) added code execution capabilities, while MetaGPT (Hong et al., 2023) and ChatDev (Qian et al., 2023) specialized in software development workflows with structured coordination through standard operating procedures. Recent advances include DyLAN (Liu et al., 2023) which introduces layered agent networks with dynamic selection and early-stopping mechanisms, and multi-agent debate frameworks (Du et al., 2023; Subramaniam et al., 2025) for enhanced reasoning capabilities. While these frameworks demonstrate effective collaboration through various communication structures—layered, centralized, decentralized, or shared message pools (Guo et al., 2024)—they primarily operate on isolated problems without accumulating experience across tasks. 𝕏**olver** introduces: (1) a dual-memory architecture with episodic retrieval and evolving intermediate memory that maintains top-ranked responses across iterations, (2) holistic experience integration combining external/self-retrieval, multi-agent collaboration, tool use, and iterative refinement in a unified framework, (3) judge-mediated selection where responses are evaluated and only the best are retained in memory for subsequent rounds, and (4) continuous expansion of the episodic corpus with newly solved problems and their reasoning traces. This transforms multi-agent systems from single-problem solvers into experience-aware agents that accumulate and leverage knowledge across problems.

LLM Reasoning Enhancement Techniques. Various techniques have emerged to enhance LLM reasoning capabilities beyond standard prompting, focusing on systematic approaches to improve accuracy and reliability. Chain-of-Thought (Wei et al., 2022) introduced step-by-step reasoning using intermediate natural language rationales (Huang and Chang, 2023), while Self-Consistency (Wang et al.) explores multiple reasoning paths through diverse rationale sampling with majority voting for improved robustness (Huang and Chang, 2023). Tree of Thoughts (Yao et al., 2023a) enables exploration of reasoning branches, extending the basic CoT paradigm—yet all remain limited to single-pass generation without error correction or experience accumulation. Self-reflective approaches like Reflexion (Shinn et al., 2024) and Self-Refine (Madaan et al., 2023) enable iterative improvement, but operate within single tasks without cross-problem learning or collaborative refinement. Recent retrieval-enhanced methods such as CheatSheet (Suzgun et al., 2025) and Search-o1 (Li et al., 2025b) augment reasoning with external knowledge, though they remain confined to single-agent architectures without shared memory or iterative refinement. Problem decomposition techniques like least-to-most prompting (Zhou et al.) and decomposed prompting (Khot et al.) tackle complex tasks by breaking them into manageable subproblems, but lack persistent memory across problems. These approaches share fundamental limitations: no cross-problem learning, no persistent memory, and no multi-agent collaboration. 𝕏**olver** unifies these enhancements within a multi-agent framework where agents collaboratively refine solutions through judge-mediated iterations, leverage dual memory systems for cross-problem learning, and dynamically adapt strategies based on accumulated experience—transforming isolated reasoning techniques into a holistic collaborative system.

Tool-Augmented Reasoning. Tool integration has become essential for extending LLM capabilities beyond language processing, enabling computational precision and external interaction. Early systems like WebGPT (Nakano et al., 2021) and Toolformer (Schick et al., 2023) introduced single-tool integration, while program synthesis approaches such as PAL (Gao et al., 2023b) and Program of Thoughts (Chen et al., a) enabled code execution for mathematical reasoning. Multi-tool frameworks emerged with ReAct (Yao et al., 2023b) interleaving reasoning with actions, Chameleon (Lu et al., 2024) composing multiple tools, and OctoTools (Lu et al., 2025) standardizing tool planning, ART (Paranjape et al., 2023) for automatic tool selection, and recent systems like ToolLLM (Qin et al., 2023) handling thousands of real-world APIs—yet all remain limited to single-agent execution without iterative refinement. Other approaches like ChatCoT (Chen et al., b) model tool use through conversations, but these still lack collaborative debugging, memory integration, and cross-problem learning. 𝕏**olver** transforms tool use into a collaborative multi-agent process where agents simultaneously execute tools, share results through intermediate memory, iteratively refine based on judge feedback, and accumulate successful tool strategies in episodic memory. This integration enables dynamic tool selection, cross-problem pattern learning, and external verification—creating an adaptive tool ecosystem that evolves with experience rather than relying on fixed demonstrations or single-pass execution.

B. Inference Details

Backbones include open-weight QWQ-32B (Qwen Team, 2024), QVQ-72B-preview, and proprietary o3-mini/o4-mini (medium/high) (OpenAI, 2025). To reduce variance, we report averages over 32 runs on LIVECODEBENCH and AIME '25, and 16 runs on AIME '24, with $\sim 1\%$ std. dev. (Appx. H.12). For simpler or instructional tasks—GSM8K, MATH-500, GAIA, BrowseComp, OSWorld, SWE-bench, HLE, and GPQA Diamond—we follow DeepSeek-v3 (Liu et al., 2024) with single greedy decoding. Defaults: temperature 0.2, agents $m = 3$, max iterations $\mathcal{I} = 2$; terminate on reaching $\mathcal{I}$ or when shared memory $\mathcal{D}_s$ converges (score 1.0 for non-code tasks; all tests passed for code). For external retrieval in coding (LiveCodeBench), we build a 9M-token corpus of algorithmic problems with C++ solutions from GitHub `https://github.com/cp-algorithms/cp-algorithms` (see Appx. G); for math we use OPENMATHREASON (Moshkov et al., 2025). Other tasks use adaptive external retrieval (GAIA, OSWorld, HLE, 2WIKI, Bamboogle, BrowseComp; see Appx. H.16) with search engines or self-retrieval (SWE-bench, SQuAD2.0, GPQA Diamond). We evaluate two variants of 𝕏**olver**: (i) 𝕏*olver with in-competition cross-problem experience* (𝕏olver (+)), which dynamically updates the episodic memory after solving each problem to utilize accumulated knowledge across problems; and (ii) 𝕏*olver (–)*, which keeps the episodic memory static, focusing solely on problem-specific experience. 𝕏olver (+) refers to our default settings.

C. Lists of Prompts

This section provides the list of prompts for planning, dynamic, judge, verifier and reasoning segmentation we have used in the experimental period. These are crucial to ensure the reproducibility (Laskar et al., 2024) of the framework 𝕏**olver**.

C.1. Planner Agent

Prompt for PLANNER AGENT

You are a planner to solve a {coding/math/agentic} problem. Here is the problem for which you have to plan:
`{problem_dict[query['problem_id']]['description']}`

First draft required strictly greater than `{m}` specialized roles to solve the problem collaboratively with reasoning behind your draft of each role.

Then select the highly influential `{m}` roles by re-checking the reasoning behind your selection and assign them to each agent to solve the problem.

C.2. Dynamic Agent

Prompt for DYNAMIC AGENT

You are a {role}. Your task is to solve a {coding/math/agentic} problem. Here is the problem that you have to solve:
problem_dict[query[‘problem_id’]][‘description’]

If external retrieval: You were also given a couple of similar problems to the problem above along with their reasoning and solutions to aid you in solving the problem at hand. Here are the similar problems you were given:
retrieved_dict[query[‘problem_id’]][‘retrieval_text’]

If self-retrieval: Further, recall a relevant and distinct problem (different from the problem mentioned above) along with its reasoning and solution.

And here was your original response:
query[[‘role’]][‘original_thought’,‘original_response’]

If iteration $i \geq 1$ (i.e., $\mathcal{D}_S$ is not empty):
Also here is the leading responses with execution results from the response store:
response_dict[‘role’,‘thought’,‘response’,‘score’]

If coding task:
Think carefully about where you went wrong, relating with responses in the response store. Then, try to fix the solution producing a thought later reply with a {Python} solution to be executed and judged again.

Make sure to wrap your code in ```python ``` block and Markdown delimiters, and include exactly one block of code with the entire solution (in the final code step).

If math/agentic task:
Think carefully about where you went wrong, relating with responses in the response store. Then, try to fix the solution producing a thought later reply with a solution to be executed and judged again. You can integrate required tools to execute the task while replying your solution if required.

Make sure to wrap your final answer in \boxed{} block with the entire solution (in the final answer step).

C.3. Judge Agent

Prompt for JUDGE AGENT

You are a judge. Your task is to judge the candidate solution of a {coding/math/agentic} problem. Here is the problem for which the candidate solution you have to judge:
`problem_dict[query[‘problem_id’]][‘description’]`

If coding task:
And here is the candidate response along with test cases against which to judge:
`query[[‘candidate_role’]]`
`[‘candidate_thought’,‘candidate_response’,‘test_case’]`

Please produce a score (based on the number of test cases passed) with reasoning behind your judgement of the candidate solution to the problem.

If math/agentic task:
And here is the candidate response which to judge:
`query[[‘candidate_role’]][‘candidate_thought’,‘candidate_response’`

Please produce a score between $[0, 1]$ with reasoning behind your judgement of the candidate solution to the problem.

C.4. Verifier Agent

Prompt for VERIFIER AGENT

Your are an answer extractor. Your task is to extract answer from the response to a {coding/math/agentic} problem. Here is the response for which the answer you have to extract:
`response_dict[query{‘role’}]{‘thought’, ‘response’, ‘score’}`

If coding task:
Please extract the answer from inside ```python ``` block from the response.

If math/agentic task:
Please extract the answer from inside \boxed{} block from the response.

C.5. Reasoning Segmentation

Prompt for INITIAL REASONING SEGMENTATION

Below is a chain of thought for solving a question. Figure out what are the different reasoning patterns that are used like problem rephrasing, new approach, self-evaluation, verification, backtracking, subgoal setup, solving a simpler problem first, and more. Then your task is to segment the entire chain of thought into different reasoning patterns. Rewrite the chain of thought in the following format:

```
<pattern> pattern name </pattern>
<content> the entire text
that corresponds to the pattern </content>
```

Thought: {thoughts}

Prompt for Final Reasoning Segmentation

Below is a thought for solving a question. For the segment between the <unannotated> and </unannotated> tags, figure out what is reasoning pattern used in that segment like problem rephrasing, new approach, self-evaluation, verification, backtracking, subgoal setup, solving a simpler problem first, or something else. Then your task is to identify the reasoning pattern used in the unannotated segment. Rewrite the unannotated segment in the following format:

```
<content> The text within the unannotated segment
that corresponds to the pattern. </content>
<reasoning> Reasoning for
what the pattern should be for the content </reasoning>
<pattern> *single* pattern name </pattern>
```

Thought: {thoughts}

D. Qualitative Examples

The subsequent section presents some qualitative examples–agentic interaction breakdown on how 𝕏olver can approach and solve a problem of math, agentic and coding tasks when it encounters. These examples illustrate how 𝕏olver initiates reasoning from external or self-retrieved exemplars, engages in multi-agent collaboration, and periodically accumulates experiences through inter-agent propagation and refinement. The full interaction trace highlights 𝕏olver's ability to iteratively decompose, solve, and adapt solutions across reasoning steps, showcasing its capacity for dynamic knowledge construction and generalizable problem solving.

Example for Correct Math Task Green: Correct Answer. Red: Wrong Answer.

Query: Bob is in charge of doing laundry for a large hotel. Each room has two sheets, one comforter, twice as many pillow cases as sheets and twice as many towels as pillow cases. How many pieces of laundry are there in 80 rooms? Make sure to wrap your final answer in `\boxed{}` block with the entire solution (in the final answer step).

Iteration 1

Mathematical Modeler: Each room contains 2 sheets, 1 comforter, 2 pillow cases (which is twice the number of sheets), and 4 towels (which is twice the number of pillow cases). Hence, the total number of laundry items in each room is:
$2 + 1 + 2 + 4 = 9$ pieces of laundry.
When considering 80 rooms, the total number of pieces of laundry is:
$80 \times 9 = 720$ pieces of laundry.
Thus, the final answer is: $\boxed{720}$.

Judge: The candidate's solution contains an error in calculating the number of pillow cases and towels per room. Here's the breakdown:
Sheets: 2 (correct). Comforter: 1 (correct). Pillow Cases: The problem states "twice as many pillow cases as sheets." Since there are 2 sheets, pillow cases should be $2 \times 2 = 4$. The candidate incorrectly used 2 pillow cases (half the required amount). Towels: The problem states "twice as many towels as pillow cases." With the correct pillow case count (4), towels should be $2 \times 4 = 8$. The candidate used 4 towels (based on their incorrect pillow case count of 2).
The correct per-room total is:
2 (sheets) + 1 (comforter) + 4 (pillow cases) + 8 (towels) = 15
items. The total for 80 rooms is:
$80 \times 15 = 1200$.

The candidate's answer is incorrect due to miscalculating pillow cases and towels.

Score: 0.

Numerical Analyst: For each room, Bob needs to wash 2 sheets, 1 comforter, 4 pillow cases (since $2 \times 2 = 4$), and 8 towels (since $4 \times 2 = 8$). Therefore, for each room, the total amount of laundry Bob needs to wash is:
$2 + 1 + 4 + 8 = 15$ pieces of laundry.
For 80 rooms, the total number of pieces of laundry Bob must wash is:
$80 \times 15 = 1200$ pieces of laundry.
Thus, the final answer is: $\boxed{1200}$.

Judge: The candidate correctly interprets the problem's relationships: sheets: 2 (given), comforter: 1 (given), pillow cases: twice the sheets, $2 \times 2 = 4$ (correct), towels: twice the pillow cases, $2 \times 4 = 8$ (correct). Total per room: $2 + 1 + 4 + 8 = 15$ (correct). Total for 80 rooms: $80 \times 15 = 1200$ (correct). All steps align with the problem's specifications, and the final answer is accurate. The candidate's solution is error-free.

Score: 1.

Mathematician: Every room is equipped with 2 sheets, 1 comforter, 4 pillow cases (calculated as 2×2), and 8 towels (since $4 \times 2 = 8$). Summing these, the total number of laundry items for one room is:
$2 + 4 + 8 = 14$ pieces of laundry.
Multiplying by the number of rooms, the total quantity for 80 rooms becomes:
$80 \times 14 = \boxed{1120}$.

Judge: The candidate incorrectly omitted the comforter in their final sum. Here's the breakdown: sheets: 2 (correct), comforter: 1 (given but not included in the candidate's calculation), pillow cases: $2 \times 2 = 4$ (correct), towels: $2 \times 4 = 8$ (correct). The candidate's total per room was $2 + 4 + 8 = 14$, but the correct total should include the comforter: $2 + 1 + 4 + 8 = 15$. Thus, the total for 80 rooms should be $80 \times 15 = 1200$, not 1120. The error stems from excluding the comforter in the summation.

Score: 0.

$\mathcal{D}_{S_1}$ rank: [Numerical Analyst, Mathematical Modeler, Mathematician]

Iteration 2

Mathematical Modeler: Each room contains 2 sheets, 1 comforter, 2 pillow cases (which is twice the number of sheets), and 4 towels (which is twice the number of pillow cases). Hence, the total number of laundry items in each room is:
$2 + 1 + 4 + 8 = 15$ pieces of laundry.
When considering 80 rooms, the total number of pieces of laundry is:
$80 \times 15 = 1200$ pieces of laundry.
Thus, the final answer is: $\boxed{1200}$.
This confirms the accuracy of the revised calculation.

Judge: The candidate's final answer is correct, and the mathematical calculations align with the problem's requirements: 1. Sheets: 2 (correct). 2. Comforter: 1 (included in the total). 3. Pillow Cases: The problem states "twice as many pillow cases as sheets." Since there are 2 sheets, this should be $2 \times 2 = 4$. The candidate correctly uses 4 pillow cases in their total calculation (despite a typographical inconsistency in the explanation, where they initially wrote "2 pillow cases"). 4. Towels: "Twice as many towels as pillow cases" gives $2 \times 4 = 8$ (correct).

The candidate's total per room is $2 + 1 + 4 + 8 = 15$ (correct), leading to $80 \times 15 = 1200$ (correct).

While the explanation contains a minor inconsistency in wording ("2 pillow cases" instead of 4), the numerical calculations are accurate. Since the final answer is correct and the mathematical steps resolve the inconsistency, the

solution is valid.

Score: 1.

Numerical Analyst: Based on the information provided, we know that each room includes 2 sheets, 1 comforter, 4 pillow cases (computed as 2×2), and 8 towels (since $4 \times 2 = 8$). Adding these together, the total number of laundry items required for one room is:
$2 + 1 + 4 + 8 = 15$ pieces.
Multiplying by the total number of rooms, the overall amount of laundry for 80 rooms is:
$80 \times 15 = \boxed{1200}$.

Although the other agents correctly determined the number of laundry items required for each room, there was an error in the concluding step–calculate the total number of laundry pieces.

Judge: The candidate's calculations and final answer are correct: sheets: 2 (given), comforter: 1 (given), pillow cases: $2 \times 2 = 4$ (twice the sheets), towels: $2 \times 4 = 8$ (twice the pillow cases), total per room: $2 + 1 + 4 + 8 = 15$ (correct), total for 80 rooms: $80 \times 15 = 1200$ (correct).

Score: 1.

Mathematician: According to the provided details, every room is equipped with 2 sheets, 1 comforter, 4 pillow cases (calculated as 2×2), and 8 towels (since $4 \times 2 = 8$). Summing these, the total number of laundry items for one room is:
$1 + 4 + 8 = 13$ pieces of laundry.
Multiplying by the number of rooms, the total quantity for 80 rooms becomes:
$80 \times 13 = \boxed{1040}$.

Judge: The candidate incorrectly omitted the sheets in their final sum. Here's the breakdown: 1. Sheets: 2 (given but not included in the candidate's calculation). 2. Comforter: 1 (included). 3. Pillow Cases: $2 \times 2 = 4$ (correct). 4. Towels: $2 \times 4 = 8$ (correct). The candidate's total per room was $1 + 4 + 8 = 13$, but the correct total should include the sheets: $2 + 1 + 4 + 8 = 15$. Thus, the total for 80 rooms should be $80 \times 15 = 1200$, not 1040. The error stems from excluding the sheets in the summation.

Score: 0.

$\mathcal{D}_{S_2}$ rank: [Mathematical Modeler, Numerical Analyst, Mathematician]

Iteration 3

Mathematical Modeler: From the details provided, it is clear that each room contains 2 sheets, 1 comforter, 4 pillow cases (as $2 \times 2 = 4$), and 8 towels (calculated as $4 \times 2 = 8$). Adding these together, the total number of laundry items per room amounts to:
$2 + 1 + 4 + 8 = 15$ items.
Consequently, for 80 rooms, the total number of laundry pieces becomes:
$80 \times 15 = \boxed{1200}$.

I agree with the other agent's corrected calculation.

Judge: The candidate's solution is mathematically accurate and complete: sheets: 2 (correct), comforter: 1 (correct), pillow cases: $2 \times 2 = 4$ (twice the sheets), towels: $2 \times 4 = 8$ (twice the pillow cases), total per room: $2 + 1 + 4 + 8 = 15$ (correct), total for 80 rooms: $80 \times 15 = 1200$ (correct).

Score: 1.

Numerical Analyst: Given the information, we know that each room includes 2 sheets, 1 comforter, 4 pillow cases (computed as 2×2), and 8 towels (since $4 \times 2 = 8$). Adding these together, the total number of laundry items required for one room is:
$2 + 1 + 4 + 8 = 15$ pieces.
Multiplying by the total number of rooms, the overall amount of laundry for 80 rooms is:
$80 \times 15 = \boxed{1200}$.

Although the other agent correctly determined the number of laundry items required for each room, there was an error in the concluding step. Specifically, they incorrectly used 80×13 instead of the appropriate multiplication of 80×15 to calculate the total number of laundry pieces.

Judge: The candidate's solution is fully correct: sheets: 2 (given), comforter: 1 (given), pillow cases: $2 \times 2 = 4$ (twice the sheets), towels: $2 \times 4 = 8$ (twice the pillow cases), total per room: $2 + 1 + 4 + 8 = 15$ (correct), total for 80 rooms: $80 \times 15 = 1200$ (correct). All steps align with the problem's specifications, and the final answer is accurate. No errors were made.

Score: 1.

Mathematician: According to the given details, each room contains 2 sheets, 1 comforter, 4 pillow cases (calculated as 2×2), and 8 towels (determined by 4×2). Summing these amounts, the total number of laundry items per room is:
$2 + 1 + 4 + 8 = 15$ pieces of laundry.
To find the total for 80 rooms, we multiply:
$80 \times 15 = \boxed{1200}$.

I concur with the other agents' results and the computations they presented.

Judge: The candidate's solution is fully correct and aligns with the problem's requirements: sheets: 2 (given), comforter: 1 (given), pillow cases: $2 \times 2 = 4$ (twice the sheets), towels: $2 \times 4 = 8$ (twice the pillow cases), total per room: $2 + 1 + 4 + 8 = 15$ (correct), total for 80 rooms: $80 \times 15 = 1200$ (correct). All calculations are accurate, and the final answer matches the problem's constraints.

Score: 1.

$\mathcal{D}_{S_3}$ rank: [Mathematician, Mathematical Modeler, Numerical Analyst]

Example for CORRECT CODING TASK GREEN: CORRECT SOLUTION. RED: WRONG SOLUTION.

Query: Yarik is a big fan of many kinds of music. But Yarik loves not only listening to music but also writing it. He likes electronic music most of all, so he has created his own system of music notes, which, in his opinion, is best for it.

Since Yarik also likes informatics, in his system notes are denoted by integers of 2^k, where $k \geq 1$ — a positive integer. But, as you know, you can't use just notes to write music, so Yarik uses combinations of two notes. The combination of two notes (a, b), where $a = 2^k$ and $b = 2^l$, he denotes by the integer a^b.

For example, if $a = 8 = 2^3$, $b = 4 = 2^2$, then the combination (a, b) is denoted by the integer $a^b = 8^4 = 4096$. Note that different combinations can have the same notation, e.g., the combination $(64, 2)$ is also denoted by the integer $4096 = 64^2$.

Yarik has already chosen n notes that he wants to use in his new melody. However, since their integers can be very large, he has written them down as an array a of length n, then the note i is $b_i = 2^{a_i}$. The integers in array a can be repeated.

The melody will consist of several combinations of two notes. Yarik was wondering how many pairs of notes b_i, b_j $(i < j)$ exist such that the combination (b_i, b_j) is equal to the combination (b_j, b_i). In other words, he wants to count the number of pairs (i, j) $(i < j)$ such that $b_i^{b_j} = b_j^{b_i}$. Help him find the number of such pairs.

Input

The first line of the input contains one integer t $(1 \le t \le 10^4)$ — the number of test cases.

The first line of each test case contains one integer n $(1 \le n \le 2 \cdot 10^5)$ — the length of the arrays.

The next line contains n integers $a_1, a_2, \ldots, a_n$ $(1 \le a_i \le 10^9)$ — array a.

It is guaranteed that the sum of n over all test cases does not exceed $2 \cdot 10^5$.

Output

For each test case, output the number of pairs that satisfy the given condition.

Sample Input 1:

5

1

2

4

3 1 3 2

2

1000 1000

3

1 1 1

19

2 4 1 6 2 8 5 4 2 10 5 10 8 7 4 3 2 6 10

Sample Output 1:

0
2
1
3

19

Make sure to wrap your code in ```python ``` block and Markdown delimiters, and include exactly one block of code with the entire solution (in the final code step).

Iteration 1

Implementation Specialist: To solve this problem, we need to determine the number of pairs of notes (b_i, b_j) such that $b_i^{b_j} = b_j^{b_i}$. The notes are given as $b_i = 2^{a_i}$, where a_i is an integer. The key insight is to recognize the conditions under which the equation $b_i^{b_j} = b_j^{b_i}$ holds. This equation can be broken down into two main cases. The first case is when the values are the same, meaning $a_i = a_j$. In this situation, the equation trivially holds because both sides will be identical. The number of such pairs is given by the combination formula $C(n, 2) = \frac{n(n-1)}{2}$ for each group of identical values. The second case occurs when the values are different but satisfy a special condition: the equation holds when one value is 1 and the other is 2. This is derived from the mathematical simplification of the equation $a_i \cdot 2^{a_j} = a_j \cdot 2^{a_i}$, which only holds true for $a_i = 1$ and $a_j = 2$ (or vice versa) when $a_i \neq a_j$.

```python
import sys
from collections import Counter

def main():
    # Read all input tokens at once
    x = sys.stdin.read().split()
    i = 0  # Index to track position in input list

    t = int(x[i])  # Number of test cases
    i += 1

    for _ in range(t):
        n = int(x[i])  # Number of elements in this test case
        i += 1

        # Extract the list of integers for current test case
        l = list(map(int, x[i:i+n]))
        i += n

        # Count frequencies of each element
        c = Counter(l)

        s = 0  # Sum of pairs

        # Sum pairs of identical elements: count choose 2
        for v in c.values():
            s += v * (v - 1) // 2

        # Add pairs formed by one element '1' and one element '2'
        s += c.get(1, 0) * c.get(2, 0)

        print(s)

if __name__ == "__main__":
    main()
```

```

Judge: There are 10 test cases. The candidate passes 5 test cases.

Score: 5.

Programmer: To solve this problem, we need to determine the number of pairs of notes such that the combination of two notes $(b_i, b_j)$ results in the same integer as the combination $(b_j, b_i)$. The notes are given as $b_i = 2^{a_i}$, and we are looking for pairs $(i, j)$ where $b_i^{b_j} = b_j^{b_i}$. The condition $b_i^{b_j} = b_j^{b_i}$ can be simplified to checking if $a_i \cdot 2^{a_j} = a_j \cdot 2^{a_i}$, which holds true in two scenarios: first, when $a_i = a_j$, producing identical pairs; and second, when one value is 1 and the other is 2, creating valid cross pairs. To count the identical pairs, for each unique value $x$ in the array, we use the combination formula $\binom{count}{2}$, where $count$ is the frequency of $x$. For cross pairs, which occur specifically between values 1 and 2, the number of such pairs is calculated as the product of the counts of 1 and 2.

```python
import sys
from collections import defaultdict

def main():
    x = sys.stdin.read().split()  # Read all input tokens
    i = 0  # Input index pointer

    t = int(x[i])  # Number of test cases
    i += 1

    for _ in range(t):
        n = int(x[i])  # Number of elements
        i += 1

        # List of integers for this test
        l = list(map(int, x[i:i+n]))
        case
        i += n

        f = defaultdict(int)  # Frequency dictionary

        # Count occurrences of each number
        for v in l:
            f[v] += 1

        s = 0  # Count pairs of identical elements

        # Sum pairs from frequencies (nC2)
        for v in f.values():
            s += v * (v - 1) // 2

        # Count pairs formed by one '1' and one '2'
        c = f.get(1, 0) * f.get(2, 0)

        print(s + c)

if __name__ == "__main__":
```
```

```
    main()
```

Judge: There are 10 test cases. The candidate passes 5 test cases.

Score: 5.

Data Structure Specialist: To solve this problem, we need to determine the number of pairs of music notes, represented by integers of the form 2^k, such that the combination of two notes (b_i, b_j) results in the same value as (b_j, b_i). In other words, we seek pairs (i, j) where $b_i^{b_j} = b_j^{b_i}$. By applying logarithms and using the properties of exponents, for notes defined as $b_i = 2^{a_i}$ and $b_j = 2^{a_j}$, the condition simplifies to $a_i \cdot 2^{a_j} = a_j \cdot 2^{a_i}$. Two key observations arise from this: first, if $a_i = a_j$, the equation holds true, meaning all pairs of identical values satisfy the condition; second, the equation also holds when one value is 1 and the other is 2, since $1 \cdot 2^2 = 2 \cdot 2^1$ simplifies to $4 = 4$. The algorithm to find the number of such pairs involves two parts. For pairs with the same value v, we count the number of pairs using the combination formula $C(n, 2) = \frac{n(n-1)}{2}$, where n is the frequency of v. For cross-value pairs where one note corresponds to 1 and the other to 2, the number of valid pairs is simply the product of their counts.

```python
import sys
from collections import Counter

def main():
    x = sys.stdin.read().split()  # Read all input tokens at once
    i = 0  # Input index pointer

    t = int(x[i])  # Number of test cases
    i += 1

    for _ in range(t):
        n = int(x[i])  # Number of elements
        i += 1

        # List of integers for this test
        l = list(map(int, x[i:i+n]))
        case
        i += n

        c = Counter(l)  # Frequency counter

        c1 = c.get(1, 0)  # Count of 1's
        c2 = c.get(2, 0)  # Count of 2's

        x = 0

        # Combine counts of 1 and 2, calculate pairs among them
        m = c1 + c2
        x += m * (m - 1) // 2

        # Calculate pairs for other numbers except 1 and 2
        for k in c:
            if k not in (1, 2):
                v = c[k]
```

```
                x += v * (v - 1) // 2

        print(x)

if __name__ == "__main__":
    main()
```

Judge: There are 10 test cases. The candidate passes 5 test cases.

Score: 5.

$\mathcal{D}_{S_1}$ rank: [Implementation Specialist, Programmer, Data Structure Specialist]

Iteration 2

Implementation Specialist: : After reviewing the solutions provided by other agents, I agree that to solve this problem, we need to determine the number of pairs of notes (b_i, b_j) such that $b_i^{b_j} = b_j^{b_i}$, where each note is given by $b_i = 2^{a_i}$. We are interested in counting pairs (i, j) with $i < j$ that satisfy this condition. By applying properties of exponents and substituting $b_i = 2^{a_i}$ and $b_j = 2^{a_j}$, the condition simplifies to checking when $\frac{2^{a_i}}{a_i} = \frac{2^{a_j}}{a_j}$. There are two key observations: first, if $a_i = a_j$, the condition is automatically met; second, pairs where one value is 1 and the other is 2 also satisfy the condition, since $\frac{2^1}{1} = 2$ and $\frac{2^2}{2} = 2$. The algorithm involves counting the frequency of each a_i, then calculating the number of valid pairs for identical values using the combination formula $\binom{f}{2}$, where f is the frequency of the value. Finally, to account for the special case of pairs between 1 and 2, we add the product of their counts to the total.

```python
import sys
from collections import Counter

def main():
    x = sys.stdin.read().split()  # Read all input tokens at once
    i = 0  # Index pointer

    t = int(x[i])  # Number of test cases
    i += 1

    for _ in range(t):
        n = int(x[i])  # Number of elements in this test case
        i += 1

        l = list(map(int, x[i:i+n]))  # List of elements
        i += n

        c = Counter(l)  # Frequency count

        s = 0  # Sum of pairs

        # Calculate pairs from identical elements: nC2 for each count
        for v in c.values():
            s += v * (v - 1) // 2
```

```
        # Add pairs formed by one '1' and one '2'
        s += c.get(1, 0) * c.get(2, 0)

        print(s)

if __name__ == "__main__":
    main()
```

Judge: There are 10 test cases. The candidate passes 6 test cases.

Score: 6.

Programmer: I agree with the other agents that, to solve this problem, we need to determine the number of valid pairs of notes such that the combination of two notes (b_i, b_j) results in the same integer as the combination (b_j, b_i). This is equivalent to finding pairs (i, j) where $b_i^{b_j} = b_j^{b_i}$. The key insight is that this equation holds under two conditions: first, when the exponents a_i and a_j are equal, the pairs (b_i, b_j) will always satisfy the equation; second, when one exponent is 1 and the other is 2, the equation also holds. To solve the problem, the approach involves three steps. First, use a frequency counter to count the occurrences of each value in the array. Second, for each value x, compute the number of valid pairs among identical values using the combination formula $\binom{f}{2}$, where f is the frequency of x. Third, calculate the number of specific pairs where one value is 1 and the other is 2 by multiplying their counts.

```python
import sys
from collections import Counter

def main():
    t = int(sys.stdin.readline())  # Number of test cases

    for _ in range(t):
        # Number of elements
        n = int(sys.stdin.readline().strip())
        # List of integers
        l = list(map(int, sys.stdin.readline().split()))

        c = Counter(l)  # Frequency counter

        s = 0  # Sum of pairs

        # Calculate pairs among identical elements:
        nC2 for each frequency
        for v in c.values():
            s += v * (v - 1) // 2

        # Add pairs formed by one '1' and one '2'
        s += c.get(1, 0) * c.get(2, 0)

        print(s)

if __name__ == "__main__":
    main()
```

```

Judge: There are 10 test cases. The candidate passes 5 test cases.

Score: 5.

Data Structure Specialist: To solve this problem, we need to determine the number of valid pairs of notes such that the combination of two notes $(b_i, b_j)$ results in the same integer as the combination $(b_j, b_i)$. This is equivalent to finding pairs $(i, j)$ where $b_i^{b_j} = b_j^{b_i}$. The key insight comes from the equation $b_i^{b_j} = b_j^{b_i}$, where $b_i = 2^{a_i}$ and $b_j = 2^{a_j}$. This equation simplifies to $a_i \cdot 2^{a_j} = a_j \cdot 2^{a_i}$. To solve this efficiently, we follow several steps. First, for each $a_i$, we perform prime factorization, expressing it as $2^y \times x$, where $x$ is the odd part of $a_i$ and $y$ is the exponent of 2 in its factorization. Next, we generate a key for each $a_i$ in the form $(x, y - a_i)$. This key is designed such that if two numbers $a_i$ and $a_j$ share the same key, they satisfy the equation $a_i \cdot 2^{a_j} = a_j \cdot 2^{a_i}$. We then use a frequency dictionary to count the occurrences of each key. Finally, for each key, we compute the number of valid pairs using the combination formula $\binom{f}{2}$, where $f$ is the frequency of that key.

```python
import sys
from collections import defaultdict

def main():
    x = sys.stdin.read().split()  # Read all input tokens at once
    i = 0  # Input index pointer

    t = int(x[i])  # Number of test cases
    i += 1

    for _ in range(t):
        n = int(x[i])  # Number of elements in this test case
        i += 1

        l = list(map(int, x[i:i+n]))  # List of integers
        i += n

        f = defaultdict(int)  # Frequency dictionary

        for v in l:
            a = v
            b = 0
            # Count how many times 2 divides the number
            while a % 2 == 0:
                a //= 2
                b += 1
            # Key is tuple of (odd part, difference of exponents)
            k = (a, b - v)
            f[k] += 1

        s = 0  # Result accumulator

        # Count pairs for each unique key using nC2 formula
        for c in f.values():
            s += c * (c - 1) // 2
```
```

```
        print(s)

if __name__ == "__main__":
    main()
```

Judge: There are 10 test cases. The candidate passes all 10 test cases with correct result.

Score: 10.

$\mathcal{D}_{S_2}$ rank: [Data Structure Specialist, Implementation Specialist, Programmer]

Iteration 3

Implementation Specialist: I concur with the other agent that to tackle this problem, our goal is to find the number of valid pairs of notes (b_i, b_j) such that raising one note to the power of the other yields the same result when the order is reversed; that is, pairs (i, j) satisfying $b_i^{b_j} = b_j^{b_i}$. Given that each note can be expressed as $b_i = 2^{a_i}$, the problem reduces to analyzing the equation $a_i \cdot 2^{a_j} = a_j \cdot 2^{a_i}$. To approach this effectively, we begin by decomposing each exponent a_i into its prime factors, specifically writing it as $2^y \times x$, where x is the odd component and y indicates the power of two within the factorization. Using this factorization, we create a unique key for each a_i defined by the pair $(x, y - a_i)$. This construction ensures that any two exponents sharing the same key satisfy the original equation. After generating these keys for all elements, we count their frequencies using a dictionary. Finally, to determine the number of valid pairs, we apply the combination formula $\binom{f}{2}$ for each key, where f is the frequency of that key, summing these results to obtain the total count of valid pairs.

```python
import sys
from collections import defaultdict

def main():
    d = sys.stdin.read().strip().split()  # Read and split all input
    t = int(d[0])  # Number of test cases
    p = 1  # Current position in input list

    for _ in range(t):
        n = int(d[p])  # Number of elements in this test case
        p += 1

        l = list(map(int, d[p:p+n]))  # Current list of elements
        p += n

        f = defaultdict(int)  # Frequency dictionary

        for v in l:
            o = v  # Original value
            c = 0  # Count of factors 2

            # Count how many times 2 divides the value
            while v % 2 == 0:
                v //= 2
                c += 1
```

```
            # Key: (odd part, count_twos - original_value)
            k = (v, c - o)
            f[k] += 1

        s = 0  # Result accumulator

        # Calculate pairs for each unique key using nC2
        for x in f.values():
            s += x * (x - 1) // 2

        print(s)

if __name__ == "__main__":
    main()
```

Judge: There are 10 test cases. The candidate passes all 10 test cases with correct result.

Score: 10.

Programmer: I agree with the other agent that to address this problem, we aim to find the number of valid pairs of notes (b_i, b_j) such that the expression $b_i^{b_j}$ equals $b_j^{b_i}$. In other words, we seek pairs (i, j) where the exponentiation relation $b_i^{b_j} = b_j^{b_i}$ holds true. Since each note is defined as $b_i = 2^{a_i}$, the equation simplifies to $a_i \cdot 2^{a_j} = a_j \cdot 2^{a_i}$. To efficiently solve this, we first factorize each exponent a_i into the form $2^y \times x$, where x represents the odd component and y is the power of two in its prime factorization. Using this decomposition, we construct a key for each a_i as $(x, y - a_i)$. This key is crafted so that if two exponents a_i and a_j share the same key, they satisfy the equality condition. Next, we tally the frequency of each key using a dictionary or hash map. Finally, the number of valid pairs corresponding to each key is computed using the combination formula $\binom{f}{2}$, where f denotes the frequency of that key. Summing these values yields the total count of valid pairs.

```python
import sys
from collections import defaultdict

def main():
    d = sys.stdin.read().strip().split()  # Read all input tokens
    t = int(d[0])  # Number of test cases
    p = 1  # Position pointer in input list

    for _ in range(t):
        n = int(d[p])  # Number of elements
        p += 1

        l = list(map(int, d[p:p+n]))  # List of elements
        p += n

        f = defaultdict(int)  # Frequency dictionary

        for v in l:
            b = v  # Base value
            c = 0  # Count of factors 2
```

```
            # Count how many times 2 divides b
            while b % 2 == 0:
                b //= 2
                c += 1

            # Key is tuple: (odd part, power difference)
            k = (b, c - v)
            f[k] += 1

        s = 0  # Result accumulator

        # Count pairs for each frequency using nC2 formula
        for x in f.values():
            s += x * (x - 1) // 2

        print(s)

if __name__ == "__main__":
    main()
```

Judge: There are 10 test cases. The candidate passes all 10 test cases with correct result.

Score: 10.

Data Structure Specialist: To solve this problem, the goal is to identify the number of valid pairs of notes (b_i, b_j) for which the value of $b_i^{b_j}$ equals that of $b_j^{b_i}$. This translates to finding pairs (i, j) satisfying the equation $b_i^{b_j} = b_j^{b_i}$. Given that each note b_i can be expressed as 2^{a_i}, the equation simplifies to $a_i \cdot 2^{a_j} = a_j \cdot 2^{a_i}$. To approach this efficiently, each exponent a_i is factorized into the form $2^y \times x$, where x denotes the odd part of a_i and y is the exponent of 2 in its prime factorization. Based on this factorization, we construct a unique key $(x, y - a_i)$ for each a_i. This key is carefully designed so that two numbers sharing the same key satisfy the given equality condition. Once the keys are generated, a frequency map is used to count how many times each key appears. Finally, for each key, the number of valid pairs is calculated using the combination formula $\binom{f}{2}$, where f represents the frequency of that key. Summing these counts across all keys gives the total number of valid pairs.

```python
import sys
from collections import defaultdict

def main():
    d = sys.stdin.read().split()  # Read all input tokens
    i = 1  # Index pointer after reading t

    t = int(d[0])  # Number of test cases

    for _ in range(t):
        n = int(d[i])  # Number of elements
        i += 1

        l = list(map(int, d[i:i+n]))  # Current list of elements
        i += n
```

```
        f = defaultdict(int)  # Frequency dictionary

        for v in l:
            x = v  # Current value
            p = 0  # Count of factor 2

            # Count how many times 2 divides x
            while x % 2 == 0:
                x //= 2
                p += 1

            k = (x, p)  # Key as (odd part, power of two)
            f[k] += 1

        s = 0  # Result accumulator

        # Count pairs for each unique key using nC2 formula
        for c in f.values():
            s += c * (c - 1) // 2

        print(s)

if __name__ == "__main__":
    main()
```

While the previous codes are correct for the given problem constraints, it implicitly assumes that only pairs of equal values or (1, 2) satisfy the equation later searching iteratively. However, explicitly grouping all valid pairs using a mathematical key, making it more robust and efficient for all scenarios, including possible edge cases not covered by the first approach.

Judge: There are 10 test cases. The candidate passes all 10 test cases with correct result.

Score: 10.

$\mathcal{D}_{S_3}$ rank: [Data Structure Specialist, Programmer, Implementation Specialist]

E. Inside 𝕏olver Memory

We check inside 𝕏olver memory trace on episodic retrieval improvements, which constitute our strongest inference-time technique. Specifically, we isolate three coding examples of problems that become newly solvable with 𝕏olver.

Models can follow established structures and strategies. Models demonstrate the capacity to emulate prior code structures and algorithmic approaches. In the present scenario, both the target problem and the retrieved case necessitate the implementation of Dijkstra's algorithm. The baseline generation exhibits vulnerabilities stemming from a lack of functional decomposition, as the solution is confined to a single scope. This reduction in modularity correlates with a higher probability of errors. Through episodic retrieval from prior experience, the model grounds its output in the provided context, effectively assimilating the robust structural and logical patterns found in the retrieved solution.

Example for CORRECT CODING TASK GREEN: CORRECT SOLUTION. RED: WRONG SOLUTION.

Query: Farmer John's farm has an outdated network of M pipes (1 <= M <= 500) for pumping milk from the barn to his milk storage tank. He wants to remove and update most of these over the next year, but he wants to leave exactly one path worth of pipes intact, so that he can still pump milk from the barn to the storage tank.

The pipe network is described by N junction points (1 <= N <= 500), each of which can serve as the endpoint of a set of pipes. Junction point 1 is the barn, and junction point N is the storage tank. Each of the M bi-directional pipes runs between a pair of junction points, and has an associated latency (the amount of time it takes milk to reach one end of the pipe from the other) and capacity (the amount of milk per unit time that can be pumped through the pipe in steady state). Multiple pipes can connect between the same pair of junction points.

For a path of pipes connecting from the barn to the tank, the latency of the path is the sum of the latencies of the pipes along the path, and the capacity of the path is the minimum of the capacities of the pipes along the path (since this is the "bottleneck" constraining the overall rate at which milk can be pumped through the path). If FJ wants to send a total of X units of milk through a path of pipes with latency L and capacity C, the time this takes is therefore L + X/C.

Given the structure of FJ's pipe network, please help him select a single path from the barn to the storage tank that will allow him to pump X units of milk in a minimum amount of total time.

INPUT FORMAT:

* Line 1: Three space-separated integers: N M X (1 <= X <= 1,000,000).

* Lines 2..1+M: Each line describes a pipe using 4 integers: I J L C. I and J (1 <= I,J <= N) are the junction points at both ends of the pipe. L and C (1 <= L,C <= 1,000,000) give the latency and capacity of the pipe.

SAMPLE INPUT:

3 3 15
1 2 10 3
3 2 10 2
1 3 14 1

INPUT DETAILS:

FJ wants to send 15 units of milk through his pipe network. Pipe #1 connects junction point 1 (the barn) to junction point 2, and has a latency of 10 and a capacity of 3. Pipes #2 and #3 are similarly defined.

OUTPUT FORMAT:

* Line 1: The minimum amount of time it will take FJ to send milk along a single path, rounded down to the nearest integer.

SAMPLE OUTPUT:

27

OUTPUT DETAILS:

The path 1->3 takes 14 + 15/1 = 29 units of time. The path 1->2->3 takes 20 + 15/2 = 27.5 units of time, and is therefore optimal.

Make sure to wrap your code in ```python ``` block and Markdown delimiters, and include exactly one block of code with the entire solution (in the final code step).

Iteration 1

Programmer:

```python
            ...
def r(n, m, x, p):
    g = {i: [] for i in range(1, n+1)}
    for i, j, l, c in p:
        g[i].append((j, l, c))
        g[j].append((i, l, c))
    v = {i: 0 for i in range(1, n+1)}
    while q:
            ...
```

...

Iteration $\mathcal{I}$

Programmer:

```python
            ...
def d(g, s, n, c):
    t = [float('inf')] * (n+1)
    t[s] = 0
    h = [0, s]
    while h:
            ...
```

Memory:

```python
            ...
def d(n, e, s, t...
    l = [float('inf')] * (n+1)
    v = [[0, s]]
    q.heapify(v)
    l[s] = 0
    while v:
            ...
```

Models can transfer reasoning from similar problems. The capacity to transfer reasoning strategies across similar problem domains enables models to improve performance. Notably, the target task and the most pertinent retrieved case from previous experience share a common requirement: operations on parenthesis-exclusive strings. Accessing a verified solution for this specific problem class provides the model with a valid reasoning prototype. By exposing the model to correct handling of this nuanced environment, the likelihood of errors is reduced, facilitating the production of more reliable code.

Example for CORRECT CODING TASK GREEN: CORRECT SOLUTION. RED: WRONG SOLUTION.

Query: Bessie the cow has escaped and is hiding on a ridge covered with tall grass. Farmer John, in an attempt to recapture Bessie, has decided to crawl through the grass on his hands and knees so he can approach undetected. Unfortunately, he is having trouble spotting Bessie from this vantage point. The grass in front of Farmer John looks like a string of N (1 <= N <= 50,000) parentheses; for example:

)((()())())

Farmer John knows that Bessie's hind legs look just like an adjacent pair of left parentheses ((, and that her front legs look exactly like a pair of adjacent right parentheses)). Bessie's location can therefore be described by a pair of indices x < y such that ((is found at position x, and)) is found at position y. Please compute the number of different such possible locations at which Bessie might be standing.

INPUT FORMAT:

* Line 1: A string of parentheses of length N (1 <= N <= 50,000).

SAMPLE INPUT:

)((()())())

OUTPUT FORMAT:

* Line 1: The number of possible positions at which Bessie can be standing – that is, the number of distinct pairs of indices x < y at which there is the pattern ((at index x and the pattern)) at index y.

SAMPLE OUTPUT:

4

OUTPUT DETAILS:

There are 4 possible locations for Bessie, indicated below:

```
1. )((()())())
    ^^   ^^

2. )((()())())
     ^^  ^^

3. )((()())())
     ^^     ^^

4. )((()())())
    ^^      ^^
```

Make sure to wrap your code in ```python ``` block and Markdown delimiters, and include exactly one block of code with the entire solution (in the final code step).

Iteration 1

Implementation Specialist:

```python
            ...
# Iterate into grass...
for i in range(len(g)-1, -1, -1):
    if g[i:i+2] == "))":
        f += 1
    elif g[i:i+2] == "((":
        c += f
            ...
```

...

Iteration $\mathcal{I}$

Implementation Specialist:

```python
            ...
for i in range(1, len(g)):
    if g[i] == '('
    and g[i-1] == '(':
        b += 1
    elif g[i] == ')'
    and g[i-1] == ')':
        l += b
            ...
```

Memory: Perhaps the simplest definition is that there must be the same total number of (’s and)’s, and for any prefix of the string, there must be at least as many (’s as)’s. For example, the following strings are all balanced:

```
()
(())
()(())
```

Models can leverage ideas and reasoning from resources. The integration of previous knowledge enables models to apply algorithmic concepts during problem-solving. For a grid-based path-finding scenario involving obstacles, the system retrieved text regarding Lowest Common Ancestor techniques rather than the standard shortest-path algorithms. Analysis indicates this deviation resulted from the sparse nature of the Dijkstra-specific content, which yielded a diminished retrieval score relative to the denser LCA chapter. Despite the conceptual divergence, the retrieved text offered applicable logic for graph traversal. This instance underscores the adaptability of the retrieval process, demonstrating its ability to secure functionally appropriate knowledge even when the most direct references are substantively lacking.

Example for CORRECT CODING TASK GREEN: CORRECT SOLUTION. RED: WRONG SOLUTION.

Query: Every morning, FJ wakes up and walks across the farm from his house to the barn. The farm is a collection of N fields (1 <= N <= 250) connected by M bidirectional pathways (1 <= M <= 25,000), each with an associated length. FJ’s house is in field 1, and the barn is in field N. No pair of fields is joined by multiple redundant pathways, and it is possible to travel between any pair of fields in the farm by walking along an appropriate sequence of pathways. When traveling from one field to another, FJ always selects a route consisting of a sequence of pathways having minimum total length.

Farmer John's cows, up to no good as always, have decided to interfere with his morning routine. They plan to build a pile of hay bales on exactly one of the M pathways on the farm, doubling its length. The cows wish to select a pathway to block so that they maximize the increase in FJ's distance from the house to the barn. Please help the cows determine by how much they can lengthen FJ's route.
INPUT FORMAT:

* Line 1: Two space-separated integers, N and M.

* Lines 2..1+M: Line j+1 describes the jth bidirectional pathway in terms of three space-separated integers: A_j B_j L_j, where A_j and B_j are indices in the range 1..N indicating the fields joined by the pathway, and L_j is the length of the pathway (in the range 1...1,000,000).

SAMPLE INPUT:

5 7
2 1 5
1 3 1
3 2 8
3 5 7
3 4 3
2 4 7
4 5 2

INPUT DETAILS:

There are 5 fields and 7 pathways. Currently, the shortest path from the house (field 1) to the barn (field 5) is 1-3-4-5 of total length 1+3+2=6.

OUTPUT FORMAT:

* Line 1: The maximum possible increase in the total length of FJ's shortest route made possible by doubling the length of a single pathway.

SAMPLE OUTPUT:

2

OUTPUT DETAILS:

If the cows double the length of the pathway from field 3 to field 4 (increasing its length from 3 to 6), then FJ's shortest route is now 1-3-5, of total length 1+7=8, larger by two than the previous shortest route length.

Make sure to wrap your code in ```python ``` block and Markdown delimiters, and include exactly one block of code with the entire solution (in the final code step).

Iteration 1

Data Structure Specialist:

```python
            ...
def main():
    g, n, m = read_input()
    d, p = dijkstra(...
```

```
    c = 0
    for u, v in p:
        l = g[u][v]
        g[u][v] *= 2
            ...
```

...

Iteration $\mathcal{I}$

Data Structure Specialist:

```python
            ...
def main(f, r):
    g = {i: {} for i in range(1,f+1)}
    for a, b, l in r:
        g[a][b] = 1
        g[b][a] = 1
    p, min_dist = dijkstra(g...)
    c = 0
    for i in range(1, len(p))
            ...
```

Memory: Let G be a tree. For every query of the form (u, v) we want to find the lowest common ancestor of the nodes u and v, i.e. we want to find the node w that lies on the path from u to the root node, that lies on the path from v to the root node, and if there are multiple nodes we pick the one that is farthest away from the root node.

F. Why 𝕏olver Performs Better

Federated sharing filters errors before they spread. In MAD, agents see other agents' previous solutions and explicitly critique, defend, and revise. The original MAD formulation is built around multiple model instances proposing and debating responses over rounds to reach a common answer. That is useful when disagreement exposes mistakes, but also injected wrong intermediate reasoning is directly into other agents' context. 𝕏olver avoids that by not passing free-form peer messages directly. Each agent solves independently, the judge scores each trajectory, and only the top-scoring reasoning-response-feedback tuples enter shared memory. Thus, MAD and Panel Discussion expose agents to unfiltered peer responses, while 𝕏olver uses a judge-filtered memory bottleneck that reduces error propagation.

Example for GPQA Diamond Communication Schemata Green: Correct Trajectory. Red: Wrong Trajectory.

Problem Interpretation: The item requires identifying the correct mechanistic interpretation from a spectral shift and a reagent/catalyst clue. A tempting distractor over-interprets the shift as direct evidence for a late-transition-metal-only mechanism.

MAD

...

Iteration 2

Spectroscopy Specialist: The downfield movement implies a more electron-withdrawing environment. This strongly suggests oxidation at the relevant site, so the answer should be the option involving a late transition metal

only.

Organic Mechanism Specialist: I agree. The downfield interpretation is consistent with an industrial late-transition-metal-catalyzed transformation. The additive clue is probably secondary.

Physical Chemist: I am not fully convinced. The observed shift could also arise from coordination or insertion. The question seems to ask for the role of both the catalyst and the additive, not just the metal center.

Spectroscopy Specialist: The magnitude of the shift is too characteristic to ignore. The additive is likely present only to tune the catalyst, not to define the mechanism.

Organic Mechanism Specialist: Yes, the industrial clue supports the metal-only option. The non-metal additive should not be selected as part of the answer.

Physical Chemist: Given both agents agree that the key evidence is the spectral shift, I revise to the late-transition-metal-only option.

Failure Pattern: The initially useful uncertainty from the Physical Chemist is overridden by two confident but wrong agents. Debate amplifies a plausible heuristic instead of filtering it.

Panel D.

...

Iteration 2

Spectroscopy Specialist: The spectral shift points to a more electron-withdrawing product environment, which is most consistent with the late-transition-metal-only choice.

Organic Mechanism Specialist: The industrial-process clue also supports a transition-metal catalyst. I would not include the additive as mechanistically decisive.

Physical Chemist: There is a possible coordination/insertion pathway. The additive may be essential, so the answer may need to include both the metal and the non-metal component.

Moderator: The panel mostly agrees that the spectral shift indicates a late-transition-metal-catalyzed process. Physical Chemist raised an additive-based alternative, but the consensus favors the late-transition-metal-only answer.

Final Panel Answer: Select the late-transition-metal-only option.

Failure Pattern: The moderator compresses disagreement into a majority-consensus summary. The minority-correct clue is preserved only as a weak caveat.

Cascaded

...

Iteration 2

Spectroscopy Specialist: The spectral shift implies electron withdrawal. Therefore, the correct answer is the late-transition-metal-only mechanism.

Organic Mechanism Specialist: Building on Spectroscopy Specialist, the high-pressure industrial setting further supports a homogeneous transition-metal process. I keep the same answer.

Physical Chemist: The previous reasoning is coherent. I verify the catalyst-class mapping and choose the same late-transition-metal-only option.

Failure Pattern: The first agent's wrong framing anchors the entire cascade. Later agents polish the explanation rather than independently challenging the initial assumption.

Federated

Iteration 1

Spectroscopy Specialist: The spectral shift implies a more electron-withdrawing environment. I choose the late-transition-metal-only option.

Judge: The candidate uses a plausible spectral heuristic but does not reconcile the additive clue. The reasoning overweights one observation and ignores a mechanistic constraint.
Score: 0.58.

Organic Mechanism Specialist: The industrial clue suggests a metal-mediated step, but the additive is not incidental. The correct answer should include both the metal catalyst class and the non-metal additive/reagent role.

Judge: The candidate better integrates the catalyst and additive clues. The answer is more consistent with the full problem statement.
Score: 0.83.

Physical Chemist: The shift alone is insufficient. Coordination or insertion can produce the same spectral direction, so the answer must be selected using both the spectroscopic and reagent constraints.

Judge: The candidate correctly avoids over-weighting the spectral cue and identifies the missing mechanistic component.
Score: 0.79.

$\mathcal{D}_{S_1}$ rank: [Organic Mechanism Specialist, Physical Chemist, Spectroscopy Specialist]

Iteration 2

Spectroscopy Specialist: My first answer over-interpreted the shift. The retained trajectories show that the additive clue changes the mechanistic interpretation. I revise to the answer including both the catalyst class and the non-metal additive role.

Judge: The candidate corrects the earlier omission and now uses both key clues.
Score: 1.

$\mathcal{D}_{S_1}$ rank: [Spectroscopy Specialist, Organic Mechanism Specialist, Physical Chemist]

Why Federated Wins: Unlike MAD, Panel D., and Cascaded, the wrong high-confidence trajectory is not freely propagated. It is scored lower and pushed below stronger alternatives in shared memory.

Federated aggregation preserves diversity better than debate or panel consensus. MAD and Panel Discussion both tend to push agents toward agreement. MAD does it adversarially: agents defend, attack, and revise. Panel Discussion does it cooperatively: the panel-style method emphasizes expert discussion, consistency analysis, and consensus-building among agents. That consensus pressure can be harmful for hard reasoning because the correct path can initially be a minority path. 𝕏olver keeps multiple high-scoring trajectories in shared memory rather than collapsing everything into one debate transcript or one moderator summary. This lets later agents borrow strong partial solutions while still independently re-solving the

full problem. Fixed-size memory D_S retains the top-m highest-scoring trajectories across agents and iterations, enabling transfer through diverse high-quality solutions.

Example for GPQA DIAMOND COMMUNICATION SCHEMATA GREEN : CORRECT TRAJECTORY. RED : WRONG TRAJECTORY.

Problem Interpretation: The item requires identifying a perturbation-specific regulatory exception in a molecular biology / genetics setting. A tempting distractor follows the canonical textbook pathway, but the correct answer depends on recognizing that a regulator is epistatic under the stated condition.

MAD

...

Iteration 2

Molecular Biologist: The phenotype points to the canonical pathway. The expected downstream readout follows the standard textbook mechanism, so the answer should be the canonical-pathway option.

Cell Biologist: I agree. The perturbation should increase activity in the standard pathway, and the observed readout is exactly what we would expect from that pathway.

Geneticist: I disagree. The perturbation changes the regulatory hierarchy. Under this condition, regulator Y is epistatic to the canonical pathway, so the canonical answer is likely a distractor.

Molecular Biologist: The stem does not explicitly say that regulator Y is knocked out or disabled. Without that, the safest interpretation is the standard pathway.

Cell Biologist: Yes, the options appear designed around the canonical mechanism. The exception seems too speculative. I stay with the canonical-pathway answer.

Geneticist: I still think regulator Y may matter, but since both agents interpret the problem canonically, I revise to the standard pathway answer.

Failure Pattern: The minority-correct trajectory is suppressed by majority pressure. MAD preserves disagreement at first, but the debate dynamic pushes agents toward consensus even when the minority trajectory contains the key exception.

Panel D.

...

Iteration 2

Molecular Biologist: The phenotype is consistent with canonical pathway activation. The downstream readout matches the standard biological mechanism.

Cell Biologist: The perturbation should affect the standard pathway, so the canonical-pathway option is the most direct answer.

Geneticist: The perturbation invokes regulator Y, which can override the canonical pathway. The correct answer may be the counterintuitive regulatory-output option.

Moderator: Two panelists support the canonical pathway answer. One panelist proposes an epistasis-based exception. Since the stem does not explicitly describe a complete knockout of regulator Y, the panel consensus is the canonical interpretation.

Final Panel Answer: Select the canonical-pathway option.

Failure Pattern: Panel D. compresses disagreement into a consensus summary. The correct minority path is mentioned but not retained as a strong candidate trajectory.

Cascaded

...

Iteration 2

Molecular Biologist: The phenotype follows canonical pathway X. Therefore, the answer is the standard downstream-effect option.

Cell Biologist: Building on Molecular Biologist, the observed readout is downstream of pathway X, and the perturbation is consistent with pathway activation. I keep the same answer.

Geneticist: The previous reasoning is coherent. Since the chain already identifies pathway X as causal, I choose the canonical-pathway option.

Failure Pattern: The cascade never explores the epistatic exception because every later agent inherits the first agent's canonical framing. The correct minority trajectory is never independently developed.

Federated

Iteration 1

Molecular Biologist: The phenotype points to canonical pathway X. I select the standard downstream-effect option.

Judge: The candidate gives a plausible default explanation, but ignores the perturbation-specific regulatory exception. The answer relies on the canonical pathway without checking whether regulator Y changes the causal ordering.
Score: 0.61.

Cell Biologist: The perturbation should increase canonical-pathway activity. The standard pathway explains the observed readout, so I choose the canonical answer.

Judge: The candidate repeats the same canonical-pathway assumption and does not address the role of regulator Y. The reasoning is incomplete.
Score: 0.60.

Geneticist: Regulator Y is epistatic under the stated perturbation. Therefore, the canonical pathway answer is a distractor. The correct answer is the regulatory-exception option, where the readout follows regulator Y, not the default pathway.

Judge: The candidate correctly uses the perturbation condition to override the default pathway. This trajectory identifies why the canonical answer is tempting but wrong.
Score: 0.86.

$\mathcal{D}_{S_1}$ rank: [Geneticist, Molecular Biologist, Cell Biologist]

Iteration 2

Molecular Biologist: My first answer incorrectly followed the canonical pathway. The retained high-score trajectory shows that regulator Y is epistatic under the perturbation. Therefore, the canonical answer is a distractor,

and the regulatory-exception option is correct.

Judge: The candidate now correctly incorporates the perturbation-specific exception and abandons the misleading canonical pathway.
Score: 1.

Cell Biologist: The standard pathway is not sufficient here. Under the stated condition, the readout follows regulator Y. I revise to the regulatory-exception answer.

Judge: The candidate correctly revises using the retained high-quality minority trajectory.
Score: 1.

Geneticist: The key is the epistatic relationship. Because regulator Y overrides the canonical pathway under this perturbation, the correct answer is the counterintuitive regulatory-output option.

Judge: The candidate remains correct and clearly identifies the distractor mechanism.
Score: 1.

$\mathcal{D}_{S_2}$ rank: [Geneticist, Molecular Biologist, Cell Biologist]

Why Federated Wins: Federated aggregation preserves diversity because a single high-quality minority trajectory can survive in shared memory. MAD and Panel D. tend to collapse toward consensus, while Cascaded communication may never explore the minority path at all.

Federated aggregation avoids the anchoring problem of Cascaded communication. Cascaded communication is especially vulnerable to path dependence. In 𝕏olver 's controlled baseline, the first agent solves, the second revises the first, the third revises the second, and so on. That means an early mistake can become the frame for every later agent. 𝕏olver breaks that chain. Every agent independently attempts the whole problem at every round, conditioned only on its own history plus judge-filtered shared memory. So a later agent is not forced to continue a bad trajectory; it can restart from the original problem while using only curated experience. Hence, cascaded reasoning strongly conditions later agents on earlier errors, while 𝕏olver preserves diversity and reduces error propagation through judge-filtered memory.

Example for GPQA DIAMOND COMMUNICATION SCHEMATA GREEN: CORRECT TRAJECTORY. RED: WRONG TRAJECTORY.

Problem Interpretation: The item requires selecting the correct limiting regime in a quantum / condensed-matter physics setting before applying an approximation. A tempting distractor uses a clean weak-coupling derivation, but the condition in the problem actually places the system in a low-temperature strong-coupling regime.

MAD

...

Iteration 2

Theorist: This is the weak-coupling limit. Using approximation W, the observable scales according to the weak-coupling expression.

Mathematical Physicist: I obtain a different scaling if the system is in the low-temperature strong-coupling regime. The condition in the stem appears to rule out approximation W.

Experimental Physicist: The measured observable is dominated by the low-temperature regime. I think the strong-coupling approximation S is more appropriate.

Theorist: The inequality in the stem looks like the weak-coupling condition. Strong coupling would require an additional assumption that is not explicitly stated.

Mathematical Physicist: The inequality is ambiguous. If weak coupling is intended, the derivation with W is cleaner and maps more directly to the answer options. I revise to the weak-coupling answer.

Experimental Physicist: I will follow the cleaner weak-coupling derivation.

Failure Pattern: Debate resolves ambiguity by favoring the cleaner derivation, even though the initial regime selection is wrong. The debate focuses on defending derivations rather than re-checking the premise.

Panel D.

...

Iteration 2

Theorist: Weak-coupling approximation W gives a direct scaling law and maps cleanly to one of the options.

Mathematical Physicist: The stated condition implies the strong-coupling / low-temperature regime. Approximation S, not W, should determine the scaling.

Experimental Physicist: The observable is regime-dependent. The weak-coupling result is likely included as a distractor.

Moderator: The panel debated weak versus strong coupling. The weak-coupling derivation is more explicit and easier to map to the answer choices. The consensus answer follows approximation W.

Final Panel Answer: Select the weak-coupling scaling option.

Failure Pattern: Panel D. rewards an internally coherent and easy-to-summarize derivation, but it fails to verify whether the derivation's starting regime is valid.

Cascaded

...

Iteration 2

Theorist: Assume weak coupling. Approximation W gives the scaling in option 3.

Mathematical Physicist: Using Theorist's weak-coupling assumption, the algebra is correct. I refine the notation and keep option 3.

Experimental Physicist: The previous chain is internally consistent. The weak-coupling expression maps to option 3, so I verify the same answer.

Failure Pattern: Cascaded communication is highly path-dependent. The first agent's approximation becomes the scaffold, and later agents validate conditional correctness instead of re-checking whether the condition applies.

Federated

Iteration 1

Theorist: This is weak coupling. Approximation W gives the observable scaling, so I choose the weak-coupling answer.

Judge: The algebra is coherent under the assumed regime, but the candidate selects a regime that conflicts with the low-temperature condition in the problem.
Score: 0.55.

Mathematical Physicist: The stated condition places the system in the low-temperature strong-coupling regime. Approximation S, not W, determines the correct scaling.

Judge: The candidate correctly identifies the dominant regime. The final option mapping is mostly correct but could be explained more explicitly.
Score: 0.82.

Experimental Physicist: The observable is regime-dependent. Comparing W and S, the problem's condition rules out the weak-coupling distractor and supports the strong-coupling expression.

Judge: The candidate explicitly rejects the distractor approximation and uses the limiting condition correctly.
Score: 0.88.

$\mathcal{D}_{S_1}$ rank: [Experimental Physicist, Mathematical Physicist, Theorist]

Iteration 2

Theorist: My first solution assumed weak coupling without checking the limiting condition. The retained trajectories show that the strong-coupling / low-temperature regime applies. Therefore, the weak-coupling answer is a distractor.

Judge: The candidate correctly abandons the invalid starting approximation and now uses the appropriate regime.
Score: 1.

Mathematical Physicist: The correct first step is identifying the regime. Once the strong-coupling condition is recognized, approximation S gives the correct scaling.

Judge: The candidate gives the correct regime and correctly maps the approximation to the final answer.
Score: 1.

Experimental Physicist: The weak-coupling expression is internally consistent but irrelevant. The experimental observable is governed by the low-temperature strong-coupling limit.

Judge: The candidate clearly distinguishes conditional algebraic validity from applicability to the problem.
Score: 1.

$\mathcal{D}_{S_2}$ rank: [Experimental Physicist, Mathematical Physicist, Theorist]

Why Federated Wins: Federated aggregation reduces anchoring because every agent independently re-solves the full problem using judge-filtered shared memory. Cascaded communication instead carries forward the first agent's framing, so a wrong initial approximation can become locked into the entire chain.

Judge-mediated memory beats persuasion and summarization. MAD relies heavily on agents persuading or correcting one another. Panel Discussion relies on a moderator-style transcript. Cascaded communication relies on the next agent improving the previous agent's output. In all three, the communication schema is mostly language produced by agents. 𝕏olver 's communication schema is different: a scored tuple containing reasoning, response, feedback, and agent metadata. The judge produces structured feedback and a dynamic score, and D_S retains the best items. This makes communication more like tournament selection than conversation.

Example for GPQA DIAMOND COMMUNICATION SCHEMATA GREEN : CORRECT TRAJECTORY. RED : WRONG TRAJECTORY.

Problem Interpretation: The item requires oxidation-state bookkeeping before applying ligand-field reasoning in an inorganic / organometallic chemistry setting. A fluent distractor assumes the wrong electron count and then gives a persuasive but incorrect ligand-field explanation.

MAD

...

Iteration 2

Ligand-Field Chemist: The metal center is d^6 low spin. Therefore, the transition is spin-allowed, and the correct answer should be option 1.

Inorganic Chemist: I calculate d^4 after charge balancing the ligands. That changes the selection-rule conclusion, so I choose option 3.

Spectroscopist: The d^6 low-spin explanation sounds more standard for this class of complex. The transition pattern described in the problem is familiar.

Ligand-Field Chemist: The ligand-field argument is decisive. The d^4 count may come from misreading the ligand charge.

Inorganic Chemist: I may have misread the charge assignment. If the complex is neutral overall, d^6 is plausible. I revise to option 1.

Spectroscopist: I agree with the d^6 low-spin interpretation. It provides the most familiar spectral explanation.

Failure Pattern: MAD allows a fluent but wrong ligand-field explanation to persuade the group before the electron count is verified. The debate rewards verbal plausibility over bookkeeping accuracy.

Panel D.

...

Iteration 2

Ligand-Field Chemist: The complex is likely d^6 low spin, giving the familiar spin-allowed transition.

Inorganic Chemist: Charge balance gives d^4, so the ligand-field conclusion changes.

Spectroscopist: The d^6 ligand-field story is more complete and matches common examples of this class of complex.

Moderator: The panel leans toward the d^6 low-spin interpretation based on ligand-field considerations. There is a dissenting d^4 count, but the consensus explanation is more complete and better aligned with the familiar transition pattern.

Final Panel Answer: Select the d^6 low-spin option.

Failure Pattern: Panel D. selects the more narratively complete explanation. The moderator summarizes away the key issue: charge balance must precede ligand-field reasoning.

Cascaded

...

Iteration 2

Ligand-Field Chemist: Assign the metal as d^6. The transition is spin-allowed, so choose option 1.

Inorganic Chemist: Using Ligand-Field Chemist's d^6 count, the ligand-field splitting supports the same option. The explanation can be strengthened by noting the low-spin configuration.

Spectroscopist: The d^6 low-spin interpretation is internally consistent and matches the expected spectral pattern. Final answer: option 1.

Failure Pattern: Once the wrong electron count is accepted by the first agent, later agents improve the explanation around that count rather than recomputing the oxidation state from scratch.

Federated

Iteration 1

Ligand-Field Chemist: The metal center is d^6 low spin. The transition is spin-allowed, so I choose option 1.

Judge: The candidate applies ligand-field reasoning before verifying charge balance. The electron count is likely wrong, which undermines the rest of the explanation.
Score: 0.52.

Inorganic Chemist: Charge balance gives a d^4 metal center. Once the correct electron count is established, the ligand-field conclusion changes. I choose option 3.

Judge: The candidate correctly performs oxidation-state bookkeeping before applying ligand-field reasoning. This trajectory identifies the hidden error in the tempting d^6 answer.
Score: 0.89.

Spectroscopist: A d^6 assignment initially seems familiar, but charge balance gives d^4. The correct spectral consequence follows from the d^4 configuration, not the familiar d^6 template.

Judge: The candidate identifies and corrects the tempting d^6 distractor while preserving the relevant spectroscopic reasoning.
Score: 0.84.

$\mathcal{D}_{S_1}$ rank: [Inorganic Chemist, Spectroscopist, Ligand-Field Chemist]

Iteration 2

Ligand-Field Chemist: My earlier ligand-field argument assumed the wrong electron count. The correct sequence is charge balance first, then ligand-field analysis. With the d^4 configuration, the answer is option 3.

Judge: The candidate corrects the hidden bookkeeping error and reaches the correct answer.
Score: 1.

Inorganic Chemist: The decisive step is the oxidation-state calculation. The familiar d^6 low-spin pattern is a distractor because it is based on an incorrect charge assignment.

Judge: The candidate correctly explains why the persuasive distractor fails.
Score: 1.

Spectroscopist: After charge balancing, the spectral conclusion must be derived from d^4, not d^6. Thus the correct answer is option 3.

Judge: The candidate gives a correct final trajectory and no longer relies on the misleading familiar template. Score: 1.

$\mathcal{D}_{S_2}$ rank: [Inorganic Chemist, Spectroscopist, Ligand-Field Chemist]

Why Federated Wins: Federated aggregation uses judge-scored reasoning-response-feedback tuples rather than persuasion, majority agreement, or moderator summarization. This allows the system to retain the chemically correct trajectory even when the wrong explanation is more fluent or familiar.

G. External Retrieval Corpus for Code

As shown in Fig. 6, our coding external retrieval corpus has 156 fine grained algorithmic tutorials with relevant problems along with their solutions collections from 10 different categories (algebrea, data structure, dynamic programming, string processing, linear algebrea, combinatorics, numerical methods, geometry, graphs and miscellaneous).

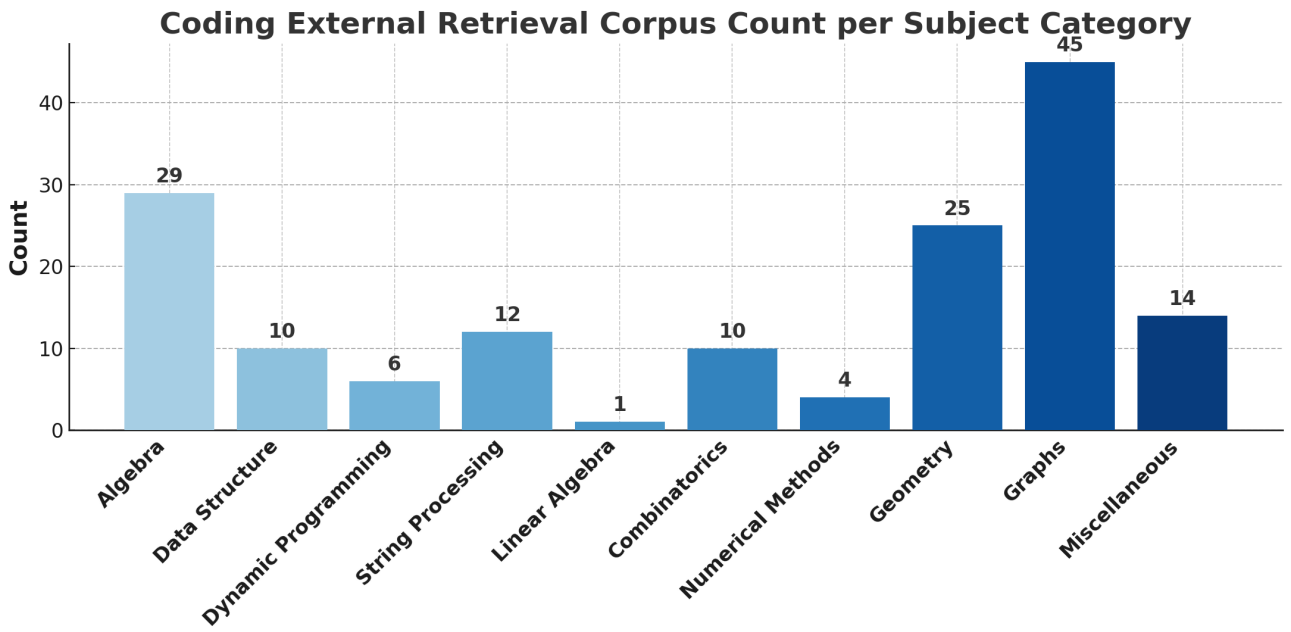

Figure 6. Coding External Retrieval Corpus Count per Subject Category.

H. Additional Analysis

This section presents our multiple runs' results which we tested and list of roles which LLMs selected during the experiments.

H.1. Additional Results

Tab. 3 presents results against some open-ended and multi-hop reasoning tasks, highlighting 𝕏olver's effectiveness on diverse domains.

Strong Gains Across Benchmarks. Overall, 𝕏olver consistently delivers significant improvements over the backbone LLMs' standard LongCoT prompting. Both the problem-specific 𝕏olver (–) and the cross-problem 𝕏olver (+) variants outperform their respective LongCoT baselines across all datasets. For example, with the o3-mini-medium backbone, 𝕏olver (+) improves from 84.8 to 95.9 EM on SQuAD2.0, from 41.8 to 78.9 EM on 2WIKI, and from 46.6 to 77.0 EM on Bamboogle. Similarly, with the QWQ-32B backbone, 𝕏olver (+) boosts EM from 79.4 to 90.2 on SQuAD2.0, from 34.4 to 70.2 on 2WIKI, and from 38.4 to 64.1 on Bamboogle.

Surpassing Prior Agents. Compared to prior frameworks such as Search-o1, 𝕏olver demonstrates consistent and significant gains. With o3-mini-medium, 𝕏olver (+) outperforms Search-o1 by +5.0 EM on SQuAD2.0 (95.9 vs. 90.9), +12.6 EM on 2WIKI (78.9 vs. 66.3), and +13.3 EM on Bamboogle (77.0 vs. 63.7). With QWQ-32B, 𝕏olver (+) also yields clear improvements, surpassing Search-o1 by +5.2 EM on SQuAD2.0, +12.2 EM on 2WIKI, and +8.1 EM on Bamboogle.

In Comparison to Leading LLMs. Despite using weaker backbones, 𝕏olver (+) matches or surpasses the strongest proprietary variants. For instance, with o3-mini-high, 𝕏olver (+) achieves state-of-the-art results across all benchmarks: 97.5 EM on SQuAD2.0, 84.4 EM on 2WIKI, and 82.3 EM on Bamboogle—outperforming all previously reported results.

Backbone Agnostic. Improvements from 𝕏olver are consistent across different backbone LLMs. Both o3-mini-medium and QWQ-32B benefit substantially from the framework, demonstrating its model-agnostic design. On 2WIKI, for example, 𝕏olver (+) reaches 78.9 EM (o3-mini-medium) and 70.2 EM (QWQ-32B), a dramatic gain over the respective LongCoT baselines of 41.8 and 34.4.

Effectiveness of Dynamic Episodic Memory. While both variants excel, the cross-problem variant 𝕏olver (+) consistently outperforms the problem-specific version 𝕏olver (–) in all benchmarks. On average, episodic memory integration yields an improvement of +4.6 EM across datasets and backbones, with the largest gain being +6.3 EM on 2WIKI with o3-mini-medium (78.9 vs. 72.6).

Scales with Backbone LLM's Strength. 𝕏**olver**'s performance scales consistently with the strength of its backbone. With o3-mini-high, 𝕏olver (+) achieves 97.5 EM / 98.8 F1 on SQuAD2.0, 84.4 EM / 97.6 F1 on 2WIKI, and 82.3 EM / 93.8 F1 on Bamboogle, setting new state-of-the-art results across all multihop retrieval benchmarks.

Model	Appr.	SQuAD2.0		2WIKI		Bamboogle	
		EM	F1	EM	F1	EM	F1
o3-mini-medium	LongCoT	84.8	87.3	41.8	48.3	46.6	60.3
	Search-o1 (C, E)	90.9	93.0	66.3	78.0	63.7	74.9
	𝕏**olver**(–) (F, E)	<u>92.2</u>	<u>96.3</u>	<u>72.6</u>	<u>87.5</u>	<u>70.9</u>	<u>81.1</u>
	𝕏**olver** (+) (F, E, M)	**95.9**	**98.3**	**78.9**	**96.3**	**77.0**	**88.7**
QWQ-32B	LongCoT	79.4	83.9	34.4	40.9	38.4	53.7
	Search-o1 (C, E)	85.0	88.4	58.0	71.4	56.0	67.8
	𝕏**olver**(–) (F, E)	88.4	91.5	63.2	78.3	60.8	72.6
	𝕏**olver**(+) (F, E, M)	90.2	94.6	70.2	86.6	64.1	79.5
o3-mini-high	𝕏**olver (+)**	**97.5**	**98.8**	**84.4**	**97.6**	**82.3**	**93.8**

Table 3. Comparison of 𝕏**olver** against SoTA reasoning agents across multihop retrieval tasks. Best results are boldfaced and second-best results are underlined. LongCoT: standard prompting for reasoning models, D: w/ MAD, P: w/ Panel D., C: w/ Cascaded, F: w/ Federated, E: w/ Evolving/Iterative, M: w/ Memory.

H.2. Performance Variance Statistics.

In this experiment on the variance of 𝕏olver performance, we tested 𝕏olver against multiple runs (16 for AIME '24 and 32 for AIME '25 and LiveCodeBench) in AIME and LiveCodeBench dataset. Results shows in Tab. 4 that it has small scale performance change with multiple runs which is a strong sign on the robustness of 𝕏olver.

As in Fig. 7, 𝕏olver (m) achieves 91.6% average accuracy on the AIME '24 and '25 benchmarks—an 18.5-point gain over o3-mini-medium—while 𝕏olver (h) reaches 94.1%, outperforming o3-mini-high by 7.2 points. On LiveCodeBench, 𝕏olver (m) improves upon its base by 21 points (66.3% to 87.3%), with 𝕏olver (h) achieving 91.6%, a 22.1-point lift over o3-mini-high.

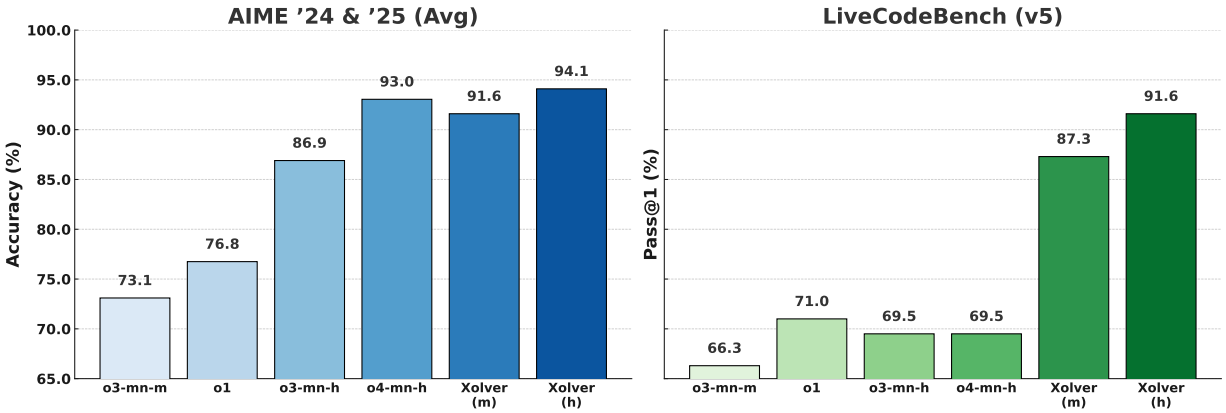

Figure 7. **Results Summary on AIME '24 (16 runs), AIME '25 and LiveCodeBench (32 runs).** Our framework **Xolver**, built on `o3-mini-medium` and `o3-mini-high` backbones (denoted (m) and (h)), achieves up to 30.9% gain over the baseline and often outperforms leading models on both tasks.

Model	Appr.	AIME '24	AIME '25	LiveCodeBench (v5)
o3-mini-medium	Xolver (-)	87.2 ± 1.2	85.1 ± 1.3	79.6 ± 1.0
	Xolver (+)	93.8 ± 0.3	89.4 ± 0.7	87.3 ± 0.4
QWQ-32B	Xolver (-)	89.9 ± 0.8	79.5 ± 1.1	76.2 ± 0.9
	Xolver (+)	93.6 ± 0.2	82.7 ± 0.8	79.2 ± 0.5
o3-mini-high	Xolver (+)	94.4 ± 0.6	93.7 ± 0.5	91.6 ± 0.3

Table 4. Xolver average performance with multiple trials.

H.3. Impact of Agent and Tool Count with $\mathcal{D}_{\mathcal{S}}$ Instances, and Emerging Benefits of Collaboration

We analyze the effect of varying the number of agents , tools, reasoning iterations, and the size of the shared memory $\mathcal{D}_{\mathcal{S}}$ on Xolver's performance. In a controlled setup, we fix one variable (e.g., 3 agents, 4 tools 2 iterations, or 3 instances) and incrementally increase the other. As shown in Fig. 3, performance improves consistently on both AIME '25 and LiveCodeBench with more agents or iterations, whereas Fig. 8 (see Section 4) shows steadily improving performance on both GAIA and Bamboogle with more tools or $\mathcal{D}_{\mathcal{S}}$ instances, highlighting the advantage of collaborative and iterative problem solving.

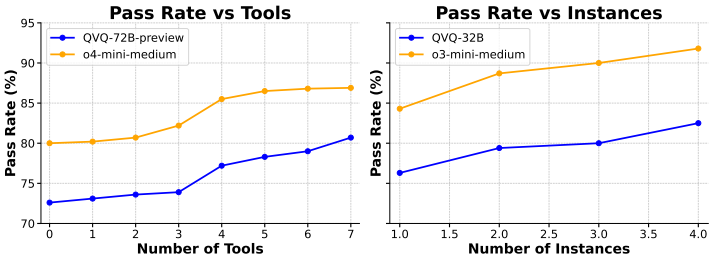

Figure 8. Impact of tools and iterations in Xolver on GAIA (left) and Bamboogle (right).

To probe deeper, we conduct a budget-controlled experiment on the Bamboogle dataset, where the total reasoning budget (i.e., number of agents $\times$ number of instances) is fixed. While tool integration remains a crucial factor for Xolver's performance, we find that increasing the number of instances—particularly beyond a minimum of three—yields additional, emergent improvements, leading to over a 6% performance gain. This suggests that shared memory diversity and parallelism complement tool breadth, together producing stronger collaborative problem-solving benefits than either alone.

H.4. Effect of Retrieval Strategies on Xolver Performance

We evaluate the impact of different retrieval strategies on Xolver by comparing three settings: (1) *External Retrieval*, where the model retrieves the top-k (e.g., $k = 5$) most similar problems and their solutions from an external corpus using a BM25 retriever; (2) *Self-Retrieval*, where the model recalls the top-k most similar problems and solutions from its own internal memory; and (3) *No Retrieval*, where neither external nor self-retrieval is used. As shown in Fig. 9 (see Section 4), performance on both AIME '25 and LiveCodeBench follows the trend: *External Retrieval* > *Self-Retrieval* > *No Retrieval*, indicating that external retrieval significantly enhances Xolver's performance.

We note that for code tasks, although the external retrieval corpus contains solutions written in C++—a different language from the target Python—external retrieval still provides a substantial performance boost. Nonetheless, while self-retrieval results in a notable performance drop compared to external retrieval, it still outperforms the no-retrieval baseline with notable margins, serving as a viable alternative when external resources are unavailable.

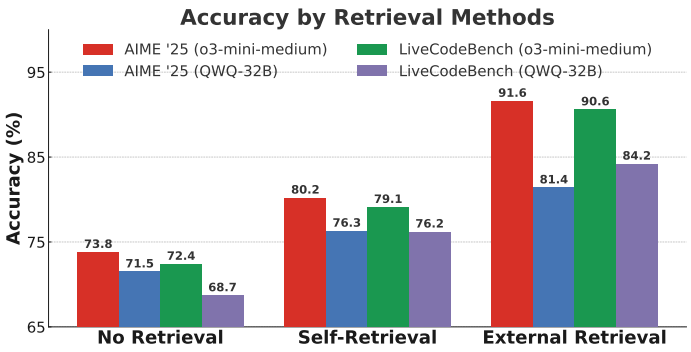

Figure 9. Impact of different retrievals in Xolver.

H.5. Fine-grained Performance Analysis

Figures shows that, on MATH-500 (Fig. 10), LIVECODEBENCH (Fig. 11) and GPQA DIAMOND (Fig. 12), Xolver (both o3-mini-medium and QWQ-32B) consistently outperforms CHEATSHEET across nearly all seven subject categories, despite the latter relying on costly per-problem memory updates. The only exception is in *Number Theory*, where `o3-mini-medium` scores 99.2 compared to CHEATSHEET's 99.5. As for QWQ-32B, Xolver achieves substantial accuracy gains over CheatSheet across all categories, with improvements of +9.0% in Prealgebra, +8.5% in Algebra, +11.0% in Number Theory, +8.5% in Counting and Probability, +8.8% in Geometry, +10.0% in Intermediate Algebra, and +7.5% in Precalculus. These consistent gains highlight **Xolver**'s strong performance across both symbolic and numerical reasoning. On LiveCodeBench, Xolver demonstrates even more pronounced gains. The o3-mini-medium variant achieves 95.6%, 90.4%, and 85.8% accuracy on Easy, Medium, and Hard problems respectively, significantly outperforming CodeSim by +4.5%, +11.9%, and a striking +32.3% margin on hard examples. Even with a weaker QWQ-32B backbone, Xolver (95.2%, 87.5%, 70.0%) surpasses all baselines and achieves similar gains. In contrast to CheatSheet and CodeSim, Xolver leverages multi-agent collaborations and holistic experience learning.

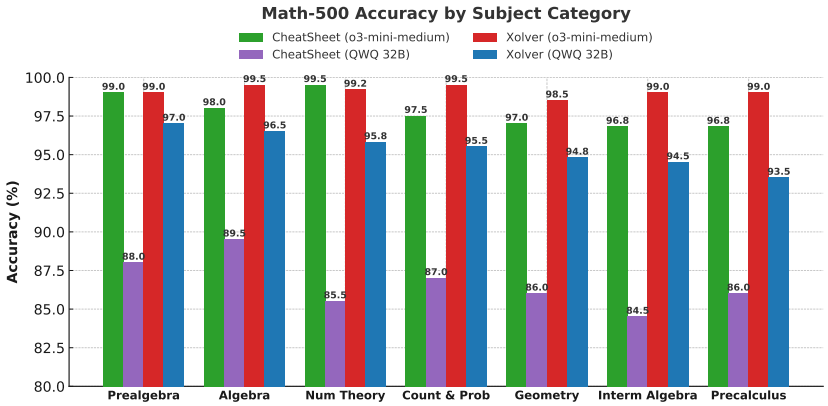

Figure 10. Fine-grained performance comparison in MATH-500.

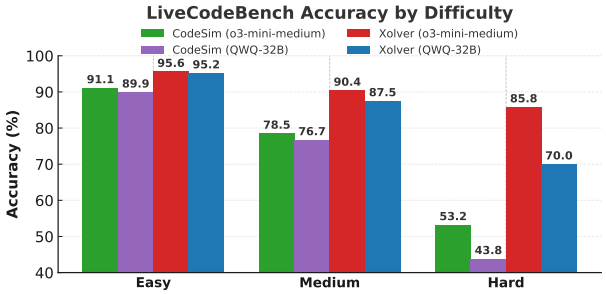

Figure 11. Performance comparison per difficulty levels in LiveCodeBench.

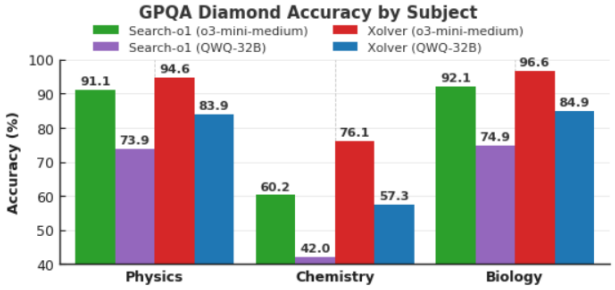

Figure 12. Fine-grained performance comparison in GPQA Diamond.

On GPQA Diamond, Xolver improves accuracy across all three subjects for both backbones (see Fig. 12). With the o3-mini-medium backbone, Xolver attains 94.6% (Physics), 76.1% (Chemistry), and 96.6% (Biology), outperforming Search-o1 by +3.5%, +15.9%, and +4.5% points, respectively. Even with the weaker QVQ-32B backbone, Xolver reaches 83.9%, 57.3%, and 84.9%—gains of +10.0%, +15.3%, and +10.0% points over the corresponding Search-o1 results. These consistent, backbone-agnostic improvements highlight **Xolver**'s strong performance on multi-disciplinary scientific QA task.

H.6. Judge Reliability, Calibration, and Sensitivity Analysis

As 𝕏olver uses the Judge Agent both to guide iterative refinement and to determine which trajectories are retained in shared memory, Judge reliability is central to the robustness of the framework. In the main inference loop, each dynamic agent independently produces a reasoning trace and response, after which the Judge assigns structured feedback and a dynamic score. The shared memory is then updated by retaining the top-scoring trajectories. Thus, an unreliable Judge could in principle preserve incorrect solutions, suppress correct but lower-scored alternatives, or introduce misleading trajectories into the episodic memory. To address this concern, we conduct a dedicated analysis of Judge correctness, long-context robustness, scoring sensitivity, and alternative trajectory-selection mechanisms. Reported backbones are o3-mini or o4-mini.

H.6.1. Human Validation of Judge Accuracy

To evaluate whether the Judge can assess long, multi-hop trajectories without access to ground truth, we annotate 300 GAIA validation trajectories stratified by difficulty level and trajectory length. Human annotators label final-answer correctness, evidence consistency, unsupported claims, tool-use errors, and late-stage reasoning failures. We then compare Judge decisions against human labels.

Judge	**Human Agr** ↑	**Endpoint Acc.** ↑	**Late-Error Det.** ↑	**Hallucination Det.** ↑	**CE** ↓
Self-judge	72.4	69.8	61.5	64.2	0.183
o4-mini	83.7	81.9	76.8	79.5	0.091
Gemini-2.5 Pro	85.1	83.6	78.2	81.0	0.083
o3-pro	**91.8**	**90.6**	**88.4**	**89.1**	**0.041**
3-judge majority	92.5	91.4	89.0	89.7	0.038

Table 5. Judge reliability on 300 human-annotated GAIA trajectories. o3-pro achieves high agreement with human labels and substantially outperforms self-judging and smaller judge models. CE denotes calibration error.

As shown in Tab. 5, the o3-pro Judge achieves 91.8% agreement with human correctness labels and 90.6% endpoint accuracy, substantially outperforming self-judging and smaller judge models. Its calibration error is also considerably lower, indicating that its dynamic scores are better aligned with empirical correctness. The 3-judge majority ensemble provides a small additional improvement, but at substantially higher inference cost.

H.6.2. Reliability Across Difficulty Levels

We further examine Judge reliability by GAIA difficulty level. This is important because Level 3 tasks typically require longer trajectories, more tool calls, and more complex multi-hop reasoning.

GAIA Split	**Human Agreement** ↑	**Endpoint Acc.** ↑	**Late-Error Det.** ↑
Level 1	94.6	93.8	92.1
Level 2	91.5	90.4	88.0
Level 3	87.9	86.6	83.7
Overall	**91.8**	**90.6**	**88.4**

Table 6. Judge reliability across GAIA difficulty levels. Reliability decreases mildly as task difficulty and trajectory length increase, but remains strong even on Level 3 examples.

Judge reliability decreases mildly as trajectory length and task difficulty increase, but remains strong even on Level 3 examples. On the hardest GAIA split, the Judge achieves 87.9% human agreement and 86.6% endpoint accuracy. This suggests that the Judge is not merely matching surface-level final answers, but is able to evaluate whether the final response is supported by the accumulated trajectory.

H.6.3. Effect of Full-Trajectory Context

To test whether full trajectories are necessary, we compare several Judge input formats.

Passing only the final answer is insufficient for GAIA-style tasks, especially when errors occur late in the reasoning process. Full-trajectory judging improves endpoint accuracy by 12.3 points over final-answer-only judging and reduces the false accept rate from 22.8% to 7.3%. Including tool logs provides an additional improvement, showing that explicit grounding in

Judge Input	Endpoint Acc. ↑	Late-Error Det. ↑	False Accept Rate ↓
Final answer only	78.3	41.7	22.8
Final answer + summary	84.9	68.5	15.6
Truncated trajectory, first 8k tokens	81.6	52.4	19.7
Truncated trajectory, last 8k tokens	86.8	79.1	12.9
Full trajectory	**90.6**	**88.4**	**7.3**
Full trajectory + tool logs	**92.1**	**90.2**	**6.5**

Table 7. Effect of Judge input context on long-context GAIA trajectory evaluation. Full-trajectory judging substantially improves endpoint accuracy and late-error detection compared with final-answer-only judging.

intermediate observations helps the Judge identify unsupported or contradicted conclusions.

H.6.4. Stress Test with Injected Reasoning Errors

We also evaluate whether the Judge can detect plausible but incorrect trajectories. Starting from correct GAIA trajectories, we inject controlled corruptions into the final 20% of the reasoning trace while preserving the earlier context. This tests whether the Judge overlooks late-stage hallucinations after a mostly plausible reasoning prefix.

Injected Corruption Type	Detection Rate ↑	False Accept Rate ↓
Wrong final entity	94.2	4.8
Unsupported tool observation	91.6	6.1
Contradiction with earlier observation	89.4	7.8
Late-stage answer swap	88.1	8.5
Plausible fabricated intermediate fact	84.7	10.9
Irrelevant but correct-looking reasoning	82.5	12.4
Average	**88.4**	**8.4**

Table 8. Injected-error stress test on GAIA trajectories. The Judge detects most late-stage answer swaps, contradictions, unsupported observations, and fabricated intermediate facts.

The Judge detects 88.1% of late-stage answer swaps and 89.4% of contradictions with earlier observations. Detection is strongest for concrete endpoint errors and slightly weaker for fabricated intermediate facts, which are more difficult to distinguish from valid retrieved context. These results indicate that the Judge is sensitive to both final-answer correctness and trajectory-level consistency.

H.6.5. Sensitivity to Judge Error

Since Judge scores directly determine shared-memory retention, we analyze how Xolver behaves when the Judge is imperfect. We simulate Judge noise by randomly flipping Judge preferences before the shared-memory update.

Injected Judge Error Rate	GAIA Acc. ↑	Δ vs. Clean Judge	Memory Corruption Rate ↓
0%	**85.6**	–	**3.8**
5%	84.7	-0.9	5.1
10%	82.9	-2.7	7.6
20%	78.4	-7.2	13.9
30%	72.5	-13.1	22.4
Random Judge	61.3	-24.3	41.7
No Judge	57.1	-28.5	–

Table 9. Sensitivity of Xolver to Judge errors on GAIA. Xolver degrades gracefully under moderate Judge noise, but severe Judge corruption substantially harms performance.

Xolver is robust to moderate Judge error. With a 10% injected error rate, GAIA accuracy drops by only 2.7 points. Larger error rates cause more substantial degradation, confirming that Judge reliability is important. However, performance remains above the no-Judge setting even under 30% injected error, suggesting that multi-agent redundancy and iterative refinement partially mitigate occasional mis-scoring.

H.6.6. Can Incorrect Judge Scores Overwrite Correct Solutions?

Incorrect Judge scores can degrade Xolver when a wrong trajectory receives a high score and is retained in memory. We therefore analyze the overwrite rate under different memory-update rules.

Memory Update Rule	GAIA Acc. ↑	Correct Trajectory Overwritten ↓	Incorrect Trajectory Retained ↓
Scalar Judge Top-1 only	80.8	11.9	14.6
Scalar Judge Top-m	**85.6**	**5.7**	**7.9**
Top-m + human-verified	**89.0**	**1.8**	**2.1**

Table 10. Effect of memory-update rules on overwrite behavior. Retaining multiple high-scoring trajectories reduces the probability that correct solutions are overwritten by incorrectly scored alternatives.

Retaining only a single top-ranked trajectory is more fragile, because one Judge mistake can eliminate correct alternatives. Xolver's default Top-m shared memory substantially reduces this risk by preserving multiple high-scoring trajectories.

H.6.7. Independent Versus Joint Evaluation of Agent Outputs

Xolver evaluates each agent output independently rather than jointly. This design keeps judging simple, scalable, and parallelizable. However, joint evaluation may capture consensus and diversity signals that independent scoring ignores. We therefore compare independent scalar judging with several joint or consensus-aware alternatives.

Selection Mechanism	GAIA Acc. ↑	Diversity Score ↑	Consensus Correctness ↑	Cost Multiplier ↓
Independent dynamic scoring	**85.6**	**0.71**	**82.9**	**1.0**×
Pairwise ranking	86.3	0.69	84.1	1.8×
Majority vote	83.5	0.58	86.0	1.2×
Diversity-aware Top-m	87.0	0.79	84.8	1.3×
3-judge ensemble	87.4	0.72	85.2	3.0×
Joint consensus	87.8	0.76	86.4	2.2×

Table 11. Comparison of independent scalar judging with joint and consensus-aware selection mechanisms. Joint judging improves accuracy but requires substantially higher cost.

Joint consensus judging produces the highest GAIA accuracy, improving over independent scoring by 2.2 points, but requires longer prompts and more expensive Judge calls. Majority voting improves consensus correctness but reduces diversity and performs worse than dynamic scoring, because agents often converge on the same plausible but incorrect endpoint. Diversity-aware Top-m selection recovers most of the joint-evaluation gain at modest cost. These results support independent scalar judging as a strong accuracy–cost tradeoff, while suggesting that diversity-aware selection is a useful extension.

H.6.8. Dynamic Scoring Versus Alternative Selection Rules

Finally, we compare scalar Judge scoring with rank aggregation, uncertainty-weighted scoring, majority voting, and ensemble judging across representative math, coding, and agentic tasks.

Selection Method	AIME'25 ↑	LiveCodeBench ↑	GAIA ↑	Average ↑	Cost Multiplier ↓
Random memory retention	76.4	70.2	61.3	69.3	1.0×
Majority vote	84.9	78.5	83.5	82.3	1.2×
Pairwise rank	88.8	85.9	86.3	87.0	1.8×
3-judge ensemble	90.1	87.2	87.4	88.2	3.0×
Scalar o3-pro	**89.4**	**87.3**	**85.6**	**87.4**	**1.0**×

Table 12. Comparison of scalar Judge scoring with alternative trajectory-selection mechanisms. Dynamic scoring is competitive with more complex methods while requiring fewer Judge calls.

Scalar Judge scoring performs competitively with more complex selection methods while requiring fewer Judge calls. Pairwise ranking and ensemble judging provide small improvements on some tasks, but the cost increase is substantial.

The dynamic score is therefore an effective default for Xolver because it gives a simple, low-overhead signal for memory retention while preserving compatibility with parallel multi-agent execution.

H.7. Can a Self-Judge Replace a Judge Agent?

We analyze the effect of different judging mechanisms on 𝕏olver's performance by comparing two setups: (1) *self-judging*, where each dynamic agent evaluates its own response through self-reflection without altering its role, and (2) *external judging*, where a separate judge agent is used to assess the responses. We find that self-judging agents tend to be biased in favor of their own outputs, occasionally validating incorrect solutions. This self-bias leads to a noticeable drop in overall performance—specifically, a 7.4% decrease in coding tasks, 4% decrease in agentic tasks and a 8.7% decrease in math tasks, on average.

H.8. Gains over Richer Communication Multi-Agents

Model	Appr.	MAD		Panel D.		Cascaded	
		Avg. gap	Wins	Avg. gap	Wins	Avg. gap	Wins
o3-mini-medium*	𝕏**olver**(–)	↑14.69 pp	09/11	↑13.50 pp	11/11	↑11.47 pp	09/11
	𝕏**olver** (+)	↑19.65 pp	11/11	↑18.46 pp	11/11	↑16.44 pp	11/11
QWQ-32B*	𝕏**olver**(–)	↑15.88 pp	10/11	↑14.04 pp	11/11	↑14.67 pp	09/11
	𝕏**olver**(+)	↑19.92 pp	11/11	↑18.07 pp	10/11	↑18.71 pp	11/11
o3-mini-high*	𝕏**olver (+)**	↑22.92 pp	11/11	↑21.73 pp	11/11	↑19.70 pp	11/11

Table 13. Controlled communication tests on 𝕏olver's federated schema with direct-interaction multi-agent baselines. Average gaps are reported as percentage-point improvements of 𝕏olver (+) over each baseline across the 11 mathetical, coding, knowledge, agentic, web, and multimodal validation benchmarks. 𝕏olver (+) consistently outperforms all three direct-interaction schemes, suggesting that federated independent solving with experience aggregation is more effective than unfiltered debate, panel discussion, or sequential handoff under the same inference agents and iterations. *: multimodal model, so we use `o4-mini` instead `o3-mini` and use `QVQ-72-preview` instead `QWQ-32B` for multimodal dataset.

Implementation of interactive multi-agent baselines. To isolate the effect of 𝕏olver 's federated shared-memory schema from other confounders, we implement MAD, Panel D., and Cascaded baselines under the same backbone model, planner-selected roles, number of agents, number of rounds, tool access, retrieval setting, verifier/debugger, and decoding parameters as 𝕏olver.

MAD: Each agent first independently proposes a solution. In subsequent rounds, each agent observes the other agents' previous solutions and is prompted to explicitly critique, defend, and revise its answer in response to disagreements.

Panel D: Agents first independently generate candidate solutions. A moderator prompt then summarizes the panel's proposed solutions, disagreements, and judge feedback into a shared discussion transcript. In the next round, each agent revises its answer using this transcript. Unlike MAD, the prompt emphasizes collaborative consensus rather than adversarial critique.

Cascaded: Agents are executed sequentially. The first agent solves the problem from the original context; the second receives the first agent's solution and revises or extends it; the third receives the second agent's output, and so forth. For multiple rounds, the cascade is repeated with the same total agent-call budget.

Federated: In contrast, 𝕏olver does not pass free-form messages directly among agents. Each agent independently solves the full problem at every round, while the judge scores each trajectory and updates $\mathcal{D}_S$ by retaining the $top - m$ reasoning-response-feedback tuples across all agents and iterations. Agents in later rounds condition on this judge-filtered shared memory, rather than on an unfiltered debate or sequential transcript.

Tab. 13 suggest that the advantage of 𝕏olver is not simply "more agents talking more in more rounds." In MAD and Panel D., agents receive unfiltered peer responses, which can propagate incorrect intermediate reasoning or encourage premature consensus. In Cascaded reasoning, later agents are strongly conditioned on earlier

agents' errors. 𝕏olver instead uses a judge-filtered memory bottleneck: only high-scoring trajectories are retained in $\mathcal{D}_{\mathcal{S}}$, and each agent still independently re-solves the full problem in the next round preserving diversity while reducing error propagation. 𝕏olver's federated scheme is a communication mechanism that is less interactive in the conversational sense, but more selective and stable as an experience-aggregation mechanism.

H.9. Cost Analysis and How Long Do 𝕏olver Agents Think?

In Fig. 14 and Fig. 13, we report input, reasoning, and output token statistics for 𝕏olver across all datasets. Token complexity scales as $O(m\mathcal{I})$ (with m agents and $\mathcal{I}$ iterations), while runtime remains $O(\mathcal{I})$ because agents run in parallel. This is more efficient than self-consistency (Wang et al., 2022), which typically needs 32–64 generations per example, and than the CheatSheet baseline with $O(n^2)$ memory updates from usefulness estimation over prior examples (for this clear superiority we haven't tested 𝕏olver's token count against of those). As a multi-agent system, 𝕏olver spends most tokens on context sharing and inter-agent communication; about 25% go to core reasoning. We compare total token usage to the single-agent Search-o1 (Li et al., 2025b), counting tokens with `tiktoken` (o3-mini-medium) and `AutoTokenizer` (QWQ-32B). 𝕏olver uses roughly $1.5\times$ more tokens, reflecting collaborative, iterative reasoning, yet this moderate increase yields substantial gains: +32.3% absolute on hard coding with o3-mini-medium and +9.05% accuracy across all Math-500 categories with QWQ-32B (see Fig. 10; Fig. 11). Thus, 𝕏olver 's slightly higher reasoning cost is well justified by its broad, superior performance.

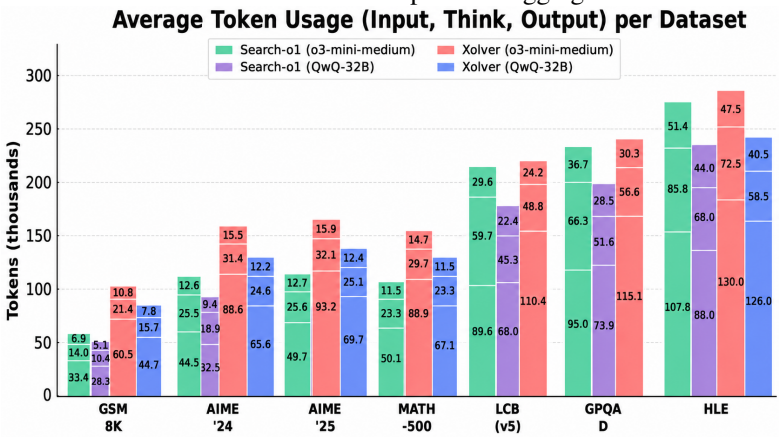

Figure 13. Average numbers of token usage across math, coding, and knowledge datasets in 𝕏olver.

H.10. Impact of Data-Shuffling in 𝕏olver (+) Performance

𝕏olver (+) updates its external memory incrementally after solving each new problem. To examine whether the order of test instances impacts performance, we conduct an ablation study by randomly shuffling the sequence of problems in each task. This helps determine if there is any dependency on the data order. During this experiment on the impact of shuffling data on 𝕏olver performance, we randomly shuffled the test instances and conducted the experiment with 5 runs. Results shows in Tab. 14 that 𝕏olver has limited performance change (STD ~ 1) with shuffling data–a strong sign on the robustness of the framework.

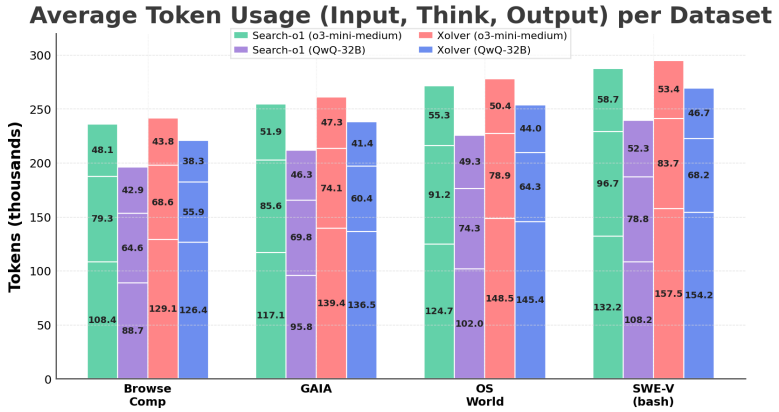

Figure 14. Average numbers of token usage across agentic browsing, multimodal, OS, SWE datasets in 𝕏olver.

Mean ± STD	**(With only 5 Runs, all STD ~ 1)**				
Model	**GSM8K**	**AIME '24**	**AIME '25**	**MATH-500**	**LiveCodeBench (v5)**
o3-mini-medium	97.6 ± 1.3	92.2 ± 0.4	91.0 ± 0.3	98.3 ± 0.6	90.9 ± 1.1
QWQ-32B	97.2 ± 0.6	93.7 ± 0.5	82.7 ± 2.0	95.1 ± 0.6	83.6 ± 1.6

Table 14. Impact of using intermediate shared memory with shuffle of order in test set in **𝕏olver**.

H.11. More Error Analysis

In Fig. 15 and Fig. 16, we present an error analysis across math, code and agentic tasks that goes beyond simple accuracy or pass@1 metrics. While 𝕏olver significantly improves reasoning and generation capabilities in these domains, both (o3-mini-medium and QWQ-32B equivalent) backbone LLMs can still produce solutions that are syntactically correct yet semantically flawed, resulting in failed executions due to incorrect reasoning, incomplete logic, unoptimized implementations, or misaligned tool usage. In agentic tasks, failure mode include mouse click, instructual, visual and other errors, whereas, code tasks include incorrect final code, time limit exceeded (TLE), runtime errors (RTE), and syntax issues. In math tasks, remaining errors are primarily due to flawed logical derivations or faulty intermediate calculations. Although Python-based tools are available, such calculation errors often occur when agents choose not to invoke these tools—highlighting that tool usage remains decoupled from the model's core reasoning process (see Appx. C for our prompt design). These findings provide insights for future improvements by exposing the variety of failure modes across domains, and further emphasize the importance of robust self-verification and refinement mechanisms, as employed by 𝕏olver.

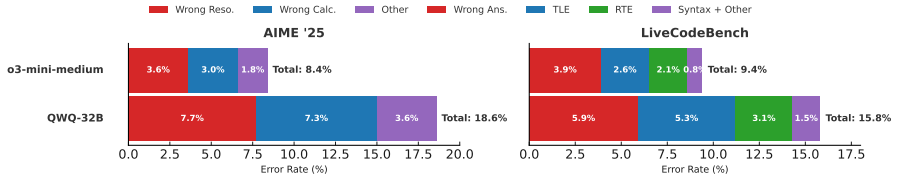

Figure 15. 𝕏olver Math and Code error distribution.

H.12. Patterns in 𝕏olver Reasoning Traces

To understand how 𝕏olver adapts its reasoning process to perform complex reasoning, we analyze the dynamics of reasoning pattern frequencies across difficulty levels in LIVECODEBENCH, as shown in Tab. 15. Our analysis reveals that 𝕏olver dynamically increases *self-evaluation* and *exploratory strategies* (e.g., trying new approaches) as problem difficulty grows. Correct solutions demonstrate a declining need for problem rephrasing and subgoal decomposition, indicating more direct and confident reasoning. In contrast, incorrect solutions show increased subgoal setup and rephrasing attempts—suggesting that the system recognizes failure and attempts recovery through restructuring. Compared to OPENCODEREASONING, which shows stagnation or regression in key patterns (e.g., self-evaluation), 𝕏olver exhibits robust and adaptive reasoning behavior, supported by multi-agent collaboration and judge feedback. This behavior highlights the generality and flexibility of 𝕏olver 's reasoning model.

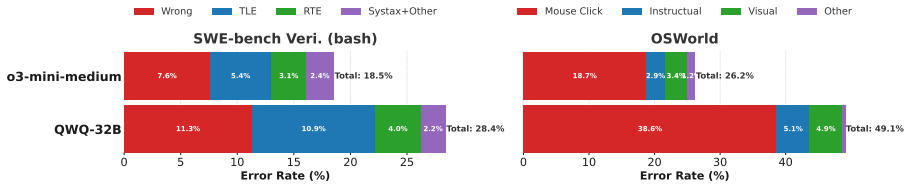

Figure 16. 𝕏olver Code and Agentic task error distribution.

We employ the QWQ-32B model for the initial segmentation of the chain of thought, guided by the first prompt, subsequently, for segments that remain unannotated, a second segmentation pass is performed using the second prompt depicted in Appx. C.5. Reasoning patterns are identified by extracting content enclosed within the <pattern> tags. Verification and reasoning patterns are combined into a unified self-evaluation category. If the model assigns multiple patterns to a single segment, that segment is excluded due to ambiguity in pattern classification. For each generated output, we calculate the proportion of occurrences of each pattern relative to the total patterns present, resulting in a frequency vector representing pattern distribution per generation. In examining the relationship between pattern usage and problem difficulty, we compute the mean frequencies separately for correct and incorrect generations and assess significance through a t-test. To evaluate pattern prevalence on a per-problem basis, a binary matrix is constructed where rows correspond to problems and entries indicate whether a pattern is more common in correct (1) or incorrect (0) solutions. The statistical significance of these findings is evaluated using a binomial test.

H.13. Case-Study: How 𝕏olver Enhances Reasoning

To further understand the reasoning and problem-solving strategies behind our multi-agent, iterative framework 𝕏olver, we conduct an in-depth analysis combining qualitative runtime inspection with controlled experiments. We begin by manually studying 𝕏**olver**'s agent interaction traces on AIME '25 and LiveCodeBench. These case stud-

Reasoning Pattern	Correct Solutions		Incorrect Solutions	
	Easy → Medium	Medium → High	Easy → Medium	Medium → High
(a) OpenCodeReasoning				
Self-Evaluation (↓)	0.39 → 0.37	0.37 → 0.34	0.36 → 0.37	0.34 → 0.31
New Approach (↑)	0.16 → 0.20	0.20 → 0.23	0.16 → 0.22	0.22 → 0.25
Problem Rephrasing (↓↑)	0.21 → 0.20	0.20 → 0.20	0.21 → 0.22	0.22 → 0.23
Subgoal Setup (↓)	0.13 → 0.12	0.12 → 0.10	0.13 → 0.10	0.10 → 0.10
(b) 𝕏olver				
Self-Evaluation (↑)	0.35 → 0.38	0.38 → 0.40	0.35 → 0.37	0.32 → 0.35
New Approach (↑)	0.18 → 0.21	0.21 → 0.24	0.17 → 0.24	0.24 → 0.26
Problem Rephrasing (↓↑)	0.20 → 0.17	0.18 → 0.18	0.23 → 0.24	0.24 → 0.25
Subgoal Setup (↓↑)	0.14 → 0.13	0.13 → 0.11	0.11 → 0.12	0.11 → 0.11

Table 15. Demonstrating how the frequency of major reasoning pattern changes as problem difficulty increases. Green indicates statistically significant increases and red indicates significant decreases ($p < 0.05$). Gray boxes highlight opposing trends between OpenCodeReasoning (decrease) and 𝕏olver (increase). Direction arrows indicate the expected trend direction: ↑ = increase, ↓ = decrease, ↓↑ = mixed trend (minor decrease then elevated recovery), ↕ = fluctuating trend (major decrease then recovery). While solving problems, OpenCodeReasoning struggles at Self-Evaluation and Subgoal Setup whereas 𝕏olver overcomes it with increasing Self-Evaluation in both correct and incorrect solutions and elevated recovery in Subgoal Setup in incorrect solutions. Both OpenCodeReasoning and 𝕏olver adapts New Approach while struggles at Problem Rephrasing.

ies reveal that at each iteration, dynamic agents attempt to improve upon earlier failures by leveraging Judge agent feedback and by aligning with top-ranked outputs stored in the shared memory $\mathcal{D}_S$. This process results in progressively refined outputs, increased agent alignment, and eventual convergence toward correct solutions. To verify this behavior systematically, we conduct a controlled experiment across both math and code tasks. We instantiate two dynamic agents with complementary strengths: a Coder agent and a Mathematician agent, each proficient in one domain but suboptimal in the other. We then measure their performance and agreement across iterations—defined as the percentage of problems in which both agents independently produce the same correct answer (for math) or code that passes the same test cases (for code). As shown in Fig. 17, both agents demonstrate consistent accuracy improvements over time, accompanied by a rising agreement rate. This not only illustrates mutual influence and learning-by-alignment but also validates the emergence of collaborative synergy.

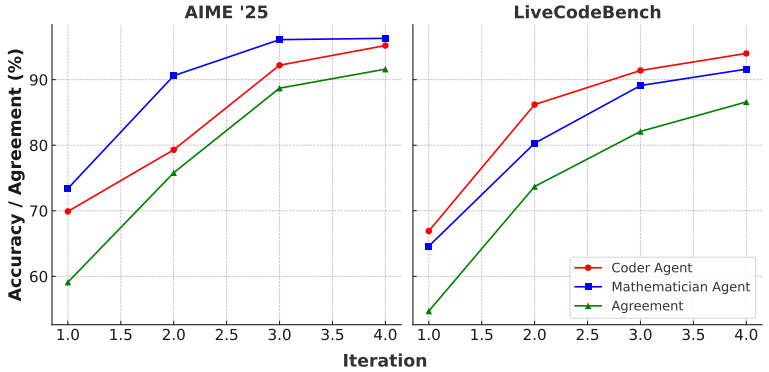

Figure 17. Agents Accuracy and Agreement over iterations.

Crucially, we observe that the presence of the Judge agent plays a vital role in this convergence process. When the Judge agent is removed—as shown in our first ablation—performance degrades significantly. These findings collectively affirm that **𝕏olver**'s iterative memory-sharing, feedback-driven refinement, and role-specialized agents contribute to its strong reasoning performance across domains, making it a compelling framework for general-purpose, self-improving problem solving.

H.14. Functional Role Diversity

To validate that planner-assigned roles are not merely stylistic variants of the same underlying solver, we introduce a shared-memory-based role-diversity diagnostic. Since **𝕏olver** stores the top-ranked agent trajectories in the intermediate shared memory D_S, the roles that survive in D_S provide a direct proxy for which role-conditioned strategies are judged useful during inference.

For each problem, each dynamic agent generates a reasoning trace, response, and judge feedback tuple. The shared memory D_S then retains the top-m tuples across agents and iterations. We therefore measure role diversity using four statistics.

First, **Unique Roles in Final** D_S counts the average number of distinct planner-assigned roles present in the final shared memory. Second, **Normalized Role Entropy** measures whether final D_S entries are evenly distributed across roles rather than dominated by a single role:

$$H_{\text{role}} = -\frac{1}{\log m} \sum_{r \in \mathcal{R}} p(r) \log p(r),$$

where $p(r)$ is the proportion of final D_S entries assigned to role r. Third, **Role-Switch Rate** measures the percentage of problems where the final verifier-selected role differs from the best first-round role. Finally, **Non-Dominant Final-Best** measures the percentage of problems where the final selected answer comes from a role other than the most frequently retained role for that benchmark.

Tab. 16 shows that role diversity in **𝕏olver** is functional. Across both o3-mini-medium and QWQ-32B, the final shared memory typically retains more than two distinct roles out of three agents. This indicates that the judge does not collapse 𝕏olver to a single dominant role, but instead preserves multiple high-value reasoning trajectories. The normalized role entropy is also high, ranging from 0.76 to 0.88 for o3-mini-medium and from 0.78 to 0.89 for QWQ-32B, further suggesting that useful trajectories are distributed across different roles. The trend also follows the broader performance behavior of **𝕏olver** (optimal performance retains with three roles combination; Fig. 18). Moreover, role diversity is lowest on relatively saturated mathematical benchmarks such as MATH-500, where many problems can be solved through a compact set of standard symbolic or algebraic strategies. It increases on AIME'25, where problems require more varied reasoning, and rises further on LiveCodeBench, where algorithm design, implementation, testing, and debugging roles often contribute complementary corrections. The strongest diversity appears on agentic benchmarks such as GAIA and OSWorld. These tasks naturally require heterogeneous subskills, including planning, tool use, evidence gathering, source validation, and final verification. Consistent with this, GAIA and OSWorld obtain the highest role entropy and role-switch rates for both backbones.

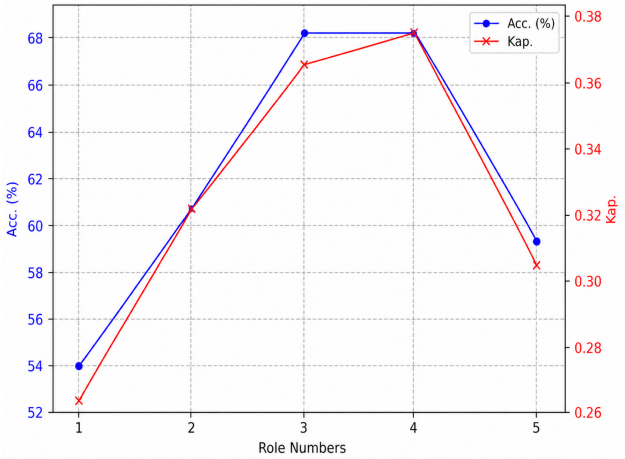

Figure 18. Performance with Increased Different Roles on BrowseComp Dataset.

The role-switching results further support the benefit of federated multi-agent refinement. For o3-mini-medium, the final selected role differs from the best first-round role in 28.4% of MATH-500, 34.7% of AIME'25, 39.2% of LiveCodeBench, 48.5% of GAIA, and 45.1% of OSWorld. QWQ-32B shows an even stronger role-switching pattern, with rates increasing to 31.7%, 38.9%, 42.6%, 50.4%, and 48.2%, respectively. This suggests that the final answer is often not produced by the initially strongest role. Instead, later iterations allow other roles to exploit the shared memory, incorporate judge feedback, and overtake earlier solutions.

Finally, the non-dominant final-best rate shows that less frequent roles can still be decisive. For example, on GAIA and OSWorld, non-dominant roles produce the final selected answer in more than 42% of cases for o3-mini-medium and more than 44% of cases for QWQ-32B. This is important because it shows that low-frequency or specialized roles are not simply auxiliary; they can become the source of the final solution when the problem requires their specific strategy.

H.15. List of Roles of Selected by Dynamic Agents

Tab. 17 shows some selected specialized roles by the dynamic agents while testing on math, coding and agentic tasks along with their most frequently selected roles.

H.16. Episodic Memory and Retrieval Protocol

XOLVER uses the term *episodic memory* D_E to denote the set of prior problem-solving experiences that are made available to the agents before or during inference. Episodic memory is distinct from the intermediate shared memory D_S which is created within a single test instance and stores the current agents' intermediate reasoning, responses, scores, and feedback, whereas episodic memory provides prior experience from outside the current reasoning trajectory. In our implementation, episodic retrieval has three modes: static external retrieval, self-retrieval, and adaptive external retrieval.

Benchmark	Unique Roles in Final D_S	Norm. Role Entropy	Role-Switch Rate	Non-Dominant Final-Best
(a) o3-mini-medium				
AIME'25	2.50 / 3	0.82	38.9	34.8
MATH-500	2.42 / 3	0.78	31.7	29.5
LCB (v5)	2.61 / 3	0.85	42.6	38.9
GAIA*	2.73 / 3	0.89	50.4	45.6
OSWorld*	2.72 / 3	0.88	48.2	44.9
(b) QWQ-32B				
AIME'25	2.43 / 3	0.79	34.7	31.5
MATH-500	2.38 / 3	0.76	28.4	26.9
LCB (v5)	2.56 / 3	0.83	39.2	36.1
GAIA*	2.71 / 3	0.88	48.5	44.3
OSWorld*	2.69 / 3	0.86	45.1	42.7

Table 16. 𝕏**olver**'s role-diversity. Green cells highlight the strongest role-diversity effects, which appear on heterogeneous agentic tasks such as GAIA and OSWorld. *: multimodal dataset, so we use `o4-mini` instead `o3-mini` and use `QVQ-72-preview` instead `QWQ-32B`.

(a) Specialized Roles		
Math	**Coding**	**Agentic**
Problem Analyzer	Problem Analyzer	Snippet Extractor
Mathematical Modeler	Algorithm Designer	Web Search Strategist
Algorithm Designer	Solution Architect	Evidence Collector
Numerical Analyst	Implementation Specialist	Source Credibility Assessor
Symbolic Solver	Data Structure Specialist	Multi-Document Synthesizer
Mathematician	Optimization Engineer	Verification Critic
Computational Tools Specialist	Unit Tester	Browser Orchestrator
	Debugging Expert	Automation Specialist
	Programmer	Workspace Manager
	Debugging Expert	User Interface Navigator
	Code Reviewer	Rubric Aligner
		Query Rewriter
(b) Most Frequent Roles		
Math	**Coding**	**Agentic**
Mathematical Modeler	Algorithm Designer	Web Search Strategist
Numerical Analyst	Implementation Specialist	Query Rewriting Strategist
Symbolic Solver	Data Structure Specialist	Evidence Extractor
Mathematician	Programmer	Consistency Verifier
Computational Tools Specialist	Optimization Engineer	Multi-Document Synthesizer

Table 17. List of math, coding and agentic roles selected by LLMs.

Static external episodic retrieval. Static external retrieval is used when a fixed corpus D_E^{ext} is available before evaluation. Each item in this corpus is an experience unit

$$e = (q', c', r'),$$

where q' is a previous problem or instructional query, c' is contextual information such as a tutorial, rationale, or reasoning trace, and r' is a solution or response. Importantly, D_E^{ext} is constructed before evaluation and does not contain the ground-truth solution of any target test instance. When the source corpus contains worked solutions, these solutions correspond only to disjoint non-test examples or public instructional materials, not to the benchmark instances being evaluated.

Given a target query q, we retrieve the top-K most relevant experience units using BM25:

$$\mathcal{R}_{\text{ext}}(q, D_E^{\text{ext}}; K) = \text{TopK}_{e \in D_E^{\text{ext}}} \text{BM25}(\phi(q), \phi(e)),$$

where $\phi(\cdot)$ denotes the tokenized textual representation of the target query or memory item. Unless otherwise stated, we use $K = 5$. The retrieved examples are serialized into the initial agent prompt as analogical exemplars. They are used only as

in-context experience and are not used by the judge or verifier (the judge or verifier extracts or validates the answer from the agents' generated responses; it does not retrieve answers from the corpus) to access labels.

For the math benchmarks, the static external corpus is built from external math reasoning resources, excluding the evaluated benchmark solutions. For LiveCodeBench, the external corpus is a fixed algorithmic knowledge corpus derived from competitive-programming tutorials and examples; the target benchmark solutions are never included. Thus, retrieval supplies potentially useful analogies, algorithms, and solution patterns, but not the answer to the evaluated problem.

Self-retrieval from parametric memory. Self-retrieval is used when no fixed external corpus is used. In this setting, XOLVER asks each agent to recall or construct relevant analogous experiences from the backbone model's own parametric memory:

$$\mathcal{R}_{\text{self}}(q; \text{LLM}_j, K) \sim \text{LLM}_j\big(\text{“recall } K \text{ relevant but distinct problems, reasoning patterns, and solutions”} \mid q\big).$$

Self-retrieval therefore does not query a database, search engine, or benchmark file. It is simply an in-context prompting mechanism that asks the model to generate analogous prior examples or strategies before solving the target query. This mode is less controlled than static external retrieval because the retrieved content is generated by the model itself; however, it cannot access any external source beyond the model's pretrained parametric knowledge and the current prompt. In our experiments, self-retrieval is used for benchmarks where a fixed, leakage-free external experience corpus is not used, such as SWE-bench, SQuAD2.0, and GPQA Diamond.

Adaptive external retrieval. Adaptive external retrieval is used for benchmarks whose task definition naturally requires interactive information seeking or web/tool use. Unlike static external retrieval, adaptive retrieval does not retrieve from a pre-indexed episodic corpus. Instead, the planner or tool-using agents issue search queries or browser/tool actions during inference:

$$\mathcal{R}_{\text{adapt}}(q; \mathcal{S}) = \text{SearchOrBrowse}(\text{QueryGen}(q), \mathcal{S}),$$

where $\mathcal{S}$ denotes the permitted search or browsing environment. The returned evidence may include web snippets, documents, pages, or environment observations. These retrieved materials are appended to the agent context as external evidence for the current problem only. They are not treated as benchmark labels, are not used to modify the evaluation script, and are not allowed to query hidden answer keys or private benchmark annotations.

We use adaptive external retrieval for agentic, browsing, multi-hop, and multimodal benchmarks where search or environment interaction is part of the intended task setting, including GAIA, OSWorld, HLE, 2WIKI, Bamboogle, and BrowseComp. This mode should therefore be interpreted as tool-mediated evidence acquisition rather than retrieval from the static episodic memory corpus D_E^{ext}.

No-retrieval setting. For ablation, we also evaluate a no-retrieval setting:

$$\mathcal{R}_{\varnothing}(q) = \varnothing.$$

In this setting, agents receive the target problem, their assigned roles, tool instructions, and later the intermediate shared memory D_S, but they receive no external exemplars and no self-retrieved examples.

Online episodic-memory update in XOLVER(+). We distinguish two evaluation variants. In XOLVER(-), the episodic memory is fixed throughout evaluation. In XOLVER(+), the system performs online experience accumulation: after a problem q_t has been solved and its final answer has already been committed, the top-ranked trajectory from D_S be appended to the episodic memory for use on later problems:

$$D_E^{\text{ext}} \leftarrow D_E^{\text{ext}} \cup \{(q_t, \hat{T}_t, \hat{R}_t)\}.$$

We report XOLVER(-) separately to show performance under a static-memory protocol, and XOLVER(+) to measure the benefit of online cross-problem experience accumulation.

Quality-gated memory insertion. To prevent low-quality or pseudo-correct solutions from polluting the long-term episodic memory, Xolver promotes an experience from the shared memory $\mathcal{D}_S$ to $\mathcal{D}_E$ only if it passes a conservative quality gate. For executable code, the response must pass all available execution checks, including public tests and synthesized tests

when applicable. For mathematical or agentic tasks, the verifier must extract a valid final answer and the judge score must be a threshold $\tau_q = 1$. When multiple agents independently converge to the same final answer, the agreement is treated as an additional confidence signal. Otherwise, the trajectory may still be useful inside the current problem through the transient shared memory $\mathcal{D}_S$, but it is not inserted into the persistent episodic memory.

Eviction mechanism. Xolver evicts entries with low expected utility. The keep score of an experience e is defined as:

$$\begin{aligned}\mathrm{Keep}(e) = \alpha \widetilde{s}_e + \beta\, \mathrm{Rel}_W(e) + \gamma \exp\left(-\frac{t_{\mathrm{now}} - t_e}{H}\right) \\ + \delta\, \mathrm{Div}(e) - \rho\, \mathrm{Red}(e),\end{aligned}$$

where $\widetilde{s}_e$ is the normalized quality score, $\mathrm{Rel}_W(e)$ is the average relevance of e to a sliding window W of recent queries, H is the temporal decay horizon, $\mathrm{Div}(e)$ rewards coverage of underrepresented task clusters, and $\mathrm{Red}(e)$ penalizes near-duplicates solution strategies. Also, the lowest-scoring entries are removed. We also preserve small per-domain quotas to avoid deleting rare but important reasoning skills. This eviction policy balances two factors: *experience timeliness*, through the exponential time-decay term, and *task relevance*, through the sliding-window relevance term.

Retrieval process. Given a query problem q, Xolver retrieves demonstrations from $\mathcal{D}_E$ using a two-stage retrieve-then-filter procedure. First, we build a lexical BM25 index over the problem field q_e and lightweight metadata d_e of each memory item. The solution response R_e and trajectory T_e are returned after retrieval, but they are not the primary matching fields, which avoids matching a new problem to an old solution merely because of shared generic solution language.

We normalize the query and memory problem statements with a task-preserving preprocessing function $\phi(\cdot)$ that lowercases natural language tokens, removes non-informative boilerplate, and preserves mathematical symbols, constraints, variable names, function names, code identifiers, etc. The initial candidate set is

$$\mathcal{C}_L(q) = \mathrm{TopL}_{e \in \mathcal{D}_E^{\mathrm{act}}}\, \mathrm{BM25}(\phi(q), \phi(q_e))\,,$$

where $L \geq k$ and k is the number of demonstrations inserted into the prompt.

The BM25 score is computed as

$$\mathrm{BM25}(q, e) = \sum_{w \in \phi(q)} \mathrm{IDF}(w) \frac{f(w, e)(k_1 + 1)}{f(w, e) + k_1 \left(1 - b + b \frac{|e|}{\mathrm{avgdl}}\right)},$$

where $f(w, e)$ is the frequency of term w in the indexed text of memory item e, $|e|$ is the indexed document length, avgdl is the average indexed document length, and (k_1, b) are the standard BM25 saturation and length-normalization parameters. In our implementation, we use standard BM25 defaults unless otherwise stated.

Second, the candidate set is filtered by task compatibility, duplicate hash, and quality score. The final retrieved demonstrations are selected as

$$\mathcal{R}(\mathcal{D}_E, q) = \mathrm{Topk}_{e \in \mathcal{C}_L(q)} \left[\mathrm{BM25}(\phi(q), \phi(q_e)) + \lambda_s \widetilde{s}_e + \lambda_t \exp\left(-\frac{t_{\mathrm{now}} - t_e}{H}\right)\right].$$

Thus, retrieval is a bounded, quality-aware, and time-aware retrieval procedure over the external memory.

I. Full Per-Baseline Main Results

This Appx. contains the two complete result tables, including all 14 baselines, four backbones, and all 11 benchmarks. Saturated GSM8K and MATH-500 (Tab. 18) and the multi-hop QA suite (SQuAD2.0, 2WIKI, Bamboogle: Tab. 3) are reported here in full. The slim main-paper Tab. 2 corresponds to the rows of these tables marked with Xolver (–) and Xolver (+).

Model	Appr.	GSM 8K	AIME '24	AIME '25	Math -500	LCB (v5)	GPQA D	GPQA HLE
Proprietary Models								
Claude 3.5 Sonnet	LongCoT	96.4	16.0	3.3	78.0	36.5	–	–
Claude 3.7 Sonnet T.	LongCoT	–	61.3	49.5	96.2	51.4	75.3	8.0
Grok-3 Beta T.	Direct	–	83.9	77.3	–	70.6	73.7	–
Grok-3-mini(Beta) T.	LongCoT	–	89.5	82.0	–	–	–	–
Gemini 2.5 F. T.	LongCoT	–	88.0	78.0	–	63.5	–	–
o1	LongCoT	96.4	74.3	79.2	96.4	71.0	73.0	8.0
o3-mini-high	LongCoT	–	87.3	86.5	–	69.5	–	–
Gemini 2.5 Pro	Direct	–	92.0	86.7	–	70.4	84.4	21.6
o3	LongCoT	96.7	91.6	88.9	–	–	82.7	20.3
o4-mini-high	LongCoT	–	93.4	**92.7**	–	69.5	81.4	18.1
Open Weights Models								
DeepSeek-R1	LongCoT	–	79.8	70.0	97.3	64.3	81.3	8.5
Qwen3-235B-A22B	LongCoT	–	85.7	81.5	–	70.7	79.0	11.8
Specialized Models (with Best Reported Results)								
rStar-Math		95.2	53.3	–	90.0	–	–	–
OpenMathReason	–	–	93.3	80.0	–	–	–	–
AlphaOne	–	–	53.3	–	89.4	75.8	–	–
OpenCodeReason	–	–	–	–	–	61.8	–	–
rStar-Coder	–	–	–	–	–	62.5	–	–
Kimi-k1.6-IOI-high	–	–	–	–	–	73.8	–	–
Kimi-k2-0905	–	–	–	–	–	–	76.7	–
Deep Research	–	–	–	–	–	–	–	26.6
Reasoning Agents/Frameworks								
o3-mini-medium	LongCoT	95.2	75.8	70.4	97.3	66.3	75.0	18.1*
	Self-Refl. (C, E, M)	93.1	79.4	76.5	95.2	73.2	–	–
	OctoTools (C, E)	95.4	81.7	75.3	97.5	–	–	–
	Search-o1 (C, E)	95.8	81.8	76.7	97.6	73.6	81.2	30.4*
	CheatSheet (C, E, M)	95.9	82.2	75.8	97.7	–	–	–
	CodeSim (C, E)	–	–	–	–	73.8	–	–
	MAD (D, E, M)	96.4	80.5	76.8	97.8	69.0	77.8	21.3*
	Panel D. (P, E, M)	95.6	84.4	78.5	95.3	68.5	71.3	23.1*
	Cascade (C, E, M)	96.0	82.5	77.5	98.0	74.0	82.0	31.0*
	𝕏olver(–) (F, E)	95.6	87.2	85.1	97.7	79.6	85.4	42.6*
	𝕏olver(+) (F, E, M)	97.1	**93.8**	89.4	**99.2**	**87.3**	**89.1**	**49.0***
QWQ-32B	LongCoT	96.1	78.1	65.8	83.2	63.4	58.1	11.7*
	Self-Refl. (C, E, M)	94.0	79.3	66.3	80.4	69.2	–	–
	OctoTools (C, E)	96.3	83.0	71.7	86.1	–	–	–
	Search-o1 (C, E)	96.4	84.4	71.8	87.1	69.3	63.6	21.2*
	CheatSheet (C, E, M)	96.8	83.5	72.2	86.5	–	–	–
	CodeSim (C, E)	–	–	–	–	70.5	–	–
	MAD (D, E, M)	97.8	80.0	76.7	86.8	66.5	62.0	13.0*
	Panel D. (P, E, M)	96.5	82.5	72.5	88.5	66.5	61.5	15.1*
	Cascaded (C, E, M)	96.8	83.1	72.8	97.7	66.0	68.1	13.5*
	𝕏olver(–) (F, E)	96.5	89.9	79.5	93.1	76.2	70.1	30.9*
	𝕏olver(+) (F, E, M)	**98.0**	93.6	82.7	95.5	79.2	75.4	38.5*
o3-mini-high	**𝕏olver**(+)	**98.1**	**94.4**	**93.7**	**99.8**	**91.6**	**94.6**	**57.3***

Table 18. Comparison of **𝕏olver** against SoTA reasoning models, specialized models, and other reasoning agents on mathematical, coding, and knowledge tasks. Best results are boldfaced and second-best results are underlined. F: Flash, T: Think models, LongCoT: standard prompting for reasoning models, D: w/ MAD, P: w/ Panel D., C: w/ Cascaded, F: w/ Federated, E: w/ Evolving/Iterative, M: w/ Memory, *: multimodal dataset, so we use `o4-mini` instead `o3-mini` and use `QVQ-72-preview` instead `QWQ-32B`. "–" denotes either n/a (only specialized agents like CodeSim for code) or results not reported (only specialized models).

Model	Appr.	Browse Comp	GAIA	OS World	SWE-V (bash)
Proprietary Models					
Claude 3.7 Sonnet T.	LongCoT	4.8	53.0	35.6	52.8
Gemini 2.5 F. T.	LongCoT	79.0	12.1	–	–
o1	LongCoT	9.9	–	–	–
Gemini 2.5 Pro	Direct	–	–	–	45.0
o3	LongCoT	49.7	–	23.0	58.4
o4-mini-high	LongCoT	28.3	–	–	–
Open Weights Models					
DeepSeek-R1	LongCoT	9.5	–	–	–
Specialized Models (with Best Reported Results)					
Deep Research-Smolagent	–	–	52.0	–	–
Deep Research	–	51.5	63.7	–	–
ChatGPT Agent	–	**68.9**	–	–	–
OpenCua-Qwen2-7B	–	–	–	–	23.1
UI-TARS-1.5-7B	–	–	–	–	27.4
OpenCua-32B	–	–	–	–	34.8
Reasoning Agents/Frameworks					
o3-mini-medium	LongCoT	24.4	54.7*	31.4*	56.2
	Alita (E, M)	–	74.1*	–	–
	AWorld (C, E, M)	–	75.8*	–	–
	GTA1	–	–	55.3*	–
	Agent S2.5 (E, M)	–	–	59.0*	–
	OpenHands (E, M)	–	–	–	62.4
	Agentless (C, E)	–	–	–	70.0
	WebSailor (E)	45.5	–	–	–
	OAgents (M)	47.1	–	–	–
	MAD (D, E, M)	29.3	57.1*	34.0*	58.5
	Panel D. (P, E, M)	33.5	61.0*	40.3*	60.1
	Cascaded (C, E, M)	32.3	60.5*	39.1*	61.0
	𝕏**olver**(–) (F, E)	60.7	82.4*	69.9*	73.9
	𝕏**olver**(+) (F, E, M)	**68.9**	**85.6***	**73.8***	**81.5**
QWQ-32B	LongCoT	17.6	45.2*	14.1*	30.1
	Alita (E, M)	–	66.3*	–	–
	AWorld (C, E, M)	–	67.3*	–	–
	GTA1	–	–	37.1*	–
	Agent S2.5 (E, M)	–	–	39.2*	–
	OpenHands (E, M)	–	–	–	50.7
	Agentless (C, E)	–	–	–	57.4
	WebSailor (E)	37.0	–	–	–
	OAgents (M)	38.5	–	–	–
	MAD (D, E, M)	20.4	48.0*	17.6*	32.2
	Panel D. (P, E, M)	25.0	52.2*	22.6*	38.4
	Cascaded (C, E, M)	19.6	47.5*	16.2*	33.0
	𝕏**olver**(–) (F, E)	49.3	75.6*	46.4*	68.2
	𝕏**olver**(+) (F, E, M)	57.4	77.3*	50.9*	71.6
o3-mini-high	𝕏**olver**(+)	**71.7**	**90.1***	**74.4***	**84.9**

Table 19. Comparison of 𝕏**olver** against SoTA reasoning models, specialized models, and other reasoning agents on agentic browsing, multimodal, OS, and SWE tasks. Best results are boldfaced and second-best results are underlined. F: Flash, T: Think models, LongCoT: standard prompting for reasoning models, D: w/ MAD, P: w/ Panel D., C: w/ Cascaded, F: w/ Federated, E: w/ Evolving/Iterative, M: w/ Memory, *: multimodal dataset (uses `o4-mini` instead of `o3-mini` and `QVQ-72-preview` instead of `QWQ-32B`). "–" denotes either n/a (only specialized agents like GTA1 and Agent S2.5 for OS, OpenHands and Agentless for SWE) or results not reported (only specialized models).