PROCEDURAL GRAPHS: SELF-EVOLVING EXECUTION STRUCTURES FOR LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models are increasingly deployed as agents that plan over long horizons and act through external tools. Most agents select actions through unconstrained generation over an accumulating history, leaving implicit the procedural knowledge of what to do, in what order, and under which conditions. As trajectories lengthen, agents can lose track of their objectives, invoke tools out of order, and repeat unproductive actions. We introduce the **Procedural Graph**: just as a knowledge graph organizes factual knowledge into (entity, relation, entity) triplets for *what-is* questions, a Procedural Graph organizes procedural knowledge into (procedure, relation, procedure) triplets for *what-to-do* questions. At each decision step, the framework localizes the agent's active node, and a guidance model translates the surrounding subgraph into step-level situational guidance that biases the solver's next action without dictating it. The graph is self-evolving: an LLM refiner contrasts failed trajectories with successful ones and edits the graph's topology and attributes, committing edits that preserve or improve held-out validation performance while retaining rejected ones to discourage repetition. Starting from a minimal skeleton, the loop builds graphs that match or surpass hand-designed ones. It can also repair a flawed expert prior. Across multiple datasets, task types, and LLMs, the Procedural Graph delivers consistent gains over memory-based baselines, and self-evolution further improves performance without manual engineering.

1 INTRODUCTION

Large language models (LLMs) are increasingly deployed as autonomous agents that plan over long horizons and act through external tools (Sumers et al., 2023; Qin et al., 2024). Most agents make decisions through unconstrained generation conditioned on a flat, growing log of prior actions and observations. This places the burden of procedural coherence on free-form generation: the agent must identify relevant observations, infer which steps remain, and choose an action that respects their dependencies. As trajectories lengthen, agents can lose track of their objectives, invoke tools out of order, and repeat unproductive actions.

Existing approaches provide procedural structure through textual memory, conditional guidelines, and explicit workflows. Memory and self-reflection methods (Shinn et al., 2023; Zhao et al., 2024) record past experience as free-form text and retrieve it for reuse in the context. Although these records preserve useful experience, the solver must still reconstruct how it applies to the current step and how it constrains the steps that follow. State-conditioned guidelines (Fu et al., 2024) provide more targeted advice, but retrieve rules without explicitly connecting successive procedural steps. Workflows and state machines (Xiao et al., 2024; Zhang et al., 2023) make those steps explicit and constrain execution, but often require manual design. Automated workflow search reduces manual design effort by optimizing workflow structure offline (Zhang et al., 2025). The remaining challenge is to combine an editable procedure representation with guidance that is conditioned on the agent's current progress.

We argue that an agent needs procedural knowledge that is structured enough to steer it away from invalid behavior, flexible enough to preserve reasoning freedom, responsive to its current progress, and able to improve from experience. We address these requirements with the **Procedural Graph (PG)**, an explicit and editable directed graph of procedural knowledge. The design mirrors a familiar

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

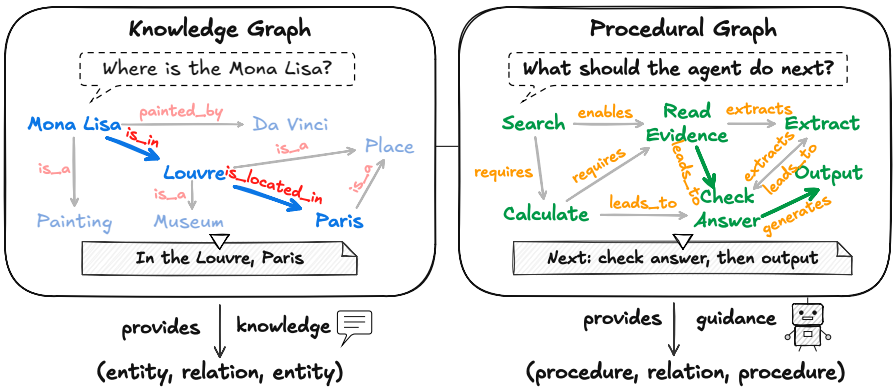

Figure 1: **From knowledge to procedure.** A Knowledge Graph organizes facts into triplets that answer *what-is* questions. A Procedural Graph organizes task procedures into triplets that answer *what-to-do* questions.

structure (Figure 1): just as a knowledge graph organizes factual knowledge into (entity, relation, entity) triplets to answer *what-is* questions, a PG organizes procedural information into (procedure, relation, procedure) triplets to answer *what-to-do* questions. Its nodes abstract tool actions, reasoning steps, and states; its edges encode permissible transitions, each annotated with textual attributes describing how and when the transition should be taken. The PG keeps a task domain's procedural knowledge outside the model weights, where it can be inspected, retrieved at each step, and edited without retraining.

Our framework puts this prior to work in two complementary phases. During online inference, the framework localizes the active node from the agent's trajectory, and a guidance model reads the surrounding subgraph in its topological context and translates the relevant edge attributes into situational guidance for the next step. During offline self-evolution, after each batch of training tasks, an LLM refiner contrasts failed trajectories with successful ones and proposes edits to the graph's topology and attributes, adding missing nodes and edges, pruning failure-inducing ones, and revising edge attributes. A structurally valid candidate graph is adopted if it matches or improves performance on a held-out validation set, and rejected candidates are retained as negative constraints. The validation gate filters out candidates that reduce the measured score, while rejection memory discourages repeated unsuccessful proposals. We summarize our contributions as follows:

- We introduce the Procedural Graph (PG), an explicit and editable graph of procedural knowledge that steers LLM-agent execution while preserving reasoning flexibility.
- We propose Generative PG Guidance, an online mechanism that converts the static graph and the live trajectory into step-level situational guidance.
- We develop a self-evolution loop that refines graph topology and attributes via execution feedback.
- Across different tasks and LLMs, PG consistently outperforms other memory baselines; evolution from scratch produces graphs that match or surpass hand-designed ones, and the loop can also repair flawed expert priors.

2 Related Work

LLM Agents and Action Selection. The dominant paradigm follows a free-form action-selection loop: ReAct (Yao et al., 2023b) interleaves reasoning with environment actions, and successors extend it with self-critique, branching search, or richer action spaces (Shinn et al., 2023; Yao et al., 2023a; Wang et al., 2024a; Qin et al., 2024). These approaches rely on the LLM to select valid next actions from in-context information, leaving admissible transitions implicit. Documented failure modes include planning hallucination, drift, and repetitive loops (Zhu et al., 2025; Xiao et al., 2024).

Structured Priors for Agent Planning. A second line provides explicit structure for planning, including textual procedure rules (Zhu et al., 2025), workflow knowledge in text, code, or flowchart

form (Xiao et al., 2024), searched workflow graphs (Zhang et al., 2025), and graph-organized tool catalogs (Liu et al., 2024a;b; Lumer et al., 2025). These methods organize procedural knowledge as action rules, workflows, or tool graphs. PG combines attributed procedure transitions, local retrieval from the current execution context, and refinement of graph topology and attributes.

Self-Improving Agents from Trajectories. A third line distills reusable knowledge from past trajectories, stored as self-critiques (Shinn et al., 2023), insights (Zhao et al., 2024), state-conditioned guidelines (Fu et al., 2024), workflows (Wang et al., 2025b), or procedural memories (Fang et al., 2025; Zhong et al., 2024). These artifacts retain different forms of structure, including conditional rules and ordered steps within workflows. PG connects transitions across procedure steps in an explicitly editable graph. Its typed, attributed edges support local structural retrieval and refinement from execution feedback. An extended survey and an eight-dimension comparison of 24 methods are provided in Appendix A.

3 The Procedural Graph Framework

We introduce the **Procedural Graph (PG)**, a directed graph of procedural knowledge that guides agent execution online while iteratively optimizing its topology and attributes offline. As illustrated in Figure 2, the framework operates in two complementary phases:

Online Inference (Section 3.2): During task solving, the graph is frozen. The agent combines the PG with its trajectory to generate dynamic situational guidance.

Offline Evolution (Section 3.3): After executing a batch of training tasks, an LLM refiner analyzes the diagnostic traces and modifies the graph topology and attributes via an automated feedback loop.

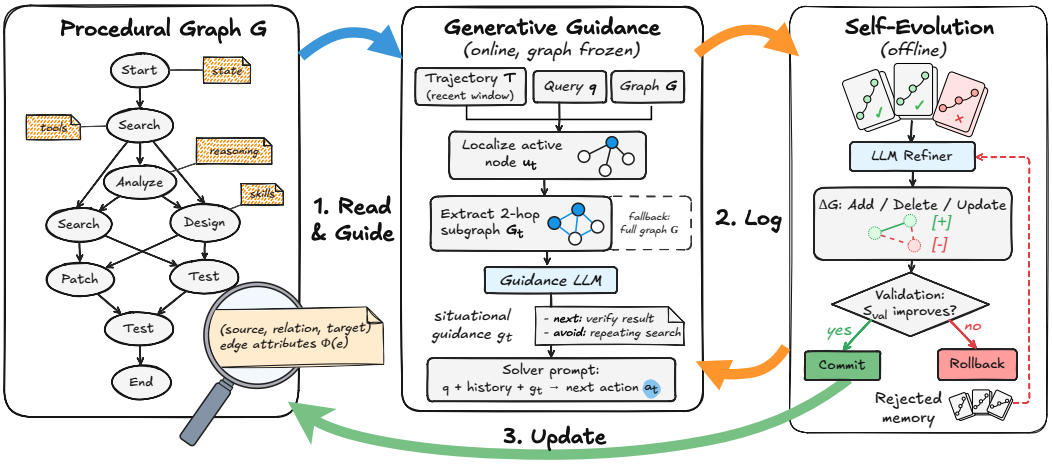

Figure 2: **Overview of the Procedural Graph framework.** *Left:* procedural triplets define $\mathcal{G}$. *Middle:* the framework localizes u_t and retrieves its 2-hop neighborhood $\mathcal{G}_t$ (or the full graph if matching fails); a guidance LLM translates it into guidance g_t for the solver. *Right:* the refiner proposes edits from execution trajectories. Structurally valid candidates are committed when validation performance does not decrease; rejected candidates inform subsequent proposals through memory.

3.1 Formal Representation of Procedural Graphs

Formally, a Procedural Graph is a directed, attributed graph

$$\mathcal{G} = (\mathcal{V}, \mathcal{R}, \mathcal{E}, \Phi), \qquad \mathcal{E} \subseteq \mathcal{V} \times \mathcal{R} \times \mathcal{V}, \tag{1}$$

where $\mathcal{V}$ is the set of abstract nodes, $\mathcal{R}$ is a vocabulary of transition relations, and each element of $\mathcal{E}$ is a directed, attributed triplet: an edge $e = (u, r, v) \in \mathcal{E}$ states that node v is admissible after node u under relation r. Each node abstracts a tool function, a skill, an internal reasoning step, or a task status. The attribute mapping Φ associates each edge with a set of named attributes whose schema

can be specified for the task. In our implementation, we use three textual fields: `condition`, `guidance`, and `pitfalls`, describing when the transition applies, how to proceed, and what to avoid. As an illustrative example, a financial-planning edge `(cash_flow_forecast, LEADS_TO, fund_raising_request)` could carry the attributes "*condition:* projected runway falls below the safety buffer; *guidance:* submit the request early to allow for the financing delivery delay; *pitfalls:* do not stack a second request while one is pending."

By structuring task knowledge into triplets, $\mathcal{G}$ makes admissible transitions explicit and guides the agent toward valid tool calls and action sequences. The graph can be initialized from an expert prior or from scratch; Section 5.3 compares these construction strategies.

3.2 Generative Guidance at Inference Time

Independent retrieval of transition attributes, such as top-k similarity search, can omit the connections between procedural steps. For example, retrieving guidance for `submit` without the preceding `check_answer` transition can omit the verification step that makes submission appropriate. Retrieving the connected neighborhood exposes both the action and its procedural prerequisites.

Generative Procedural Graph Guidance addresses this by combining three operations: *locate*, *extract*, and *generate*. Let q be the user query and $\mathcal{T}_t = (a_1, o_1, \ldots, a_{t-1}, o_{t-1})$ be the interleaved history of actions and observations up to decision step t. We use $a_0 = \texttt{Start}$ as an initialization marker, so the first step is localized at $u_1 = \texttt{Start}$. At each step,

$$u_t = \mathrm{Match}\big(a_{t-1},\, \mathcal{V}\big), \qquad \mathcal{G}_t = \begin{cases} \mathcal{N}_h(u_t), & u_t \neq \varnothing, \\ \mathcal{G}, & \text{otherwise}, \end{cases} \qquad g_t = \Psi\big(\mathcal{G}_t,\; q,\; \mathcal{T}_{t-w:t}\big), \quad (2)$$

where Match locates the agent by exactly matching its most recent procedure (e.g., a tool call) to a node in $\mathcal{V}$. The directed edge neighborhood $\mathcal{N}_h(u_t)$ contains u_t and the outgoing transitions reached by expanding for up to h steps. The window $\mathcal{T}_{t-w:t}$ contains the last w trajectory steps, and Ψ is the guidance language model. This connected neighborhood lets Ψ read transitions in their topological context and consider possible next steps up to h transitions ahead. Ψ translates the static attributes $\Phi(e)$ of the surrounding edges into situational guidance g_t, identifying the agent's immediate goal and the formatting or logical errors to avoid.

The guidance g_t is appended to the task solver's prompt. The solver then selects its next action from the query, trajectory, and guidance:

$$a_t \sim P_{\text{solver}}\big(\cdot \;\big|\; q,\; \mathcal{T}_t,\; g_t\big). \quad (3)$$

This soft integration allows the agent to maintain flexible reasoning while being steered toward the procedural structure encoded in $\mathcal{G}$. Guidance uses the localized neighborhood when matching succeeds and the full graph otherwise, together with a recent trajectory window. Section 5.5 evaluates the resulting performance and efficiency; Appendix F provides execution cases.

3.3 Self-Evolution of Procedural Graphs

An offline self-evolution loop adapts graph topology and attributes from execution feedback, reducing the need for manual design (Algorithm 1). Let $\mathcal{G}_0$ be the initial Procedural Graph and $\mathcal{G}_k$ the retained graph after round k. Each round starts from $\mathcal{G}_{k-1}$; a rejected candidate never becomes the starting graph of the next round. Across generations $k = 1, \ldots, K$, the evolution engine executes a four-step loop:

Step 1: Diagnostic Rollout. Using the retained graph $\mathcal{G}_{k-1}$, the solver runs on a batch of training tasks $\mathcal{B}_k \subset \mathcal{D}_{\text{train}}$, and we record the diagnostic traces together with their evaluation scores, $\mathcal{E}_k = \big\{(q_i,\, \mathcal{T}_i^{(k)},\, S_i^{(k)})\big\}_{i=1}^{|\mathcal{B}_k|}$, where $S_i^{(k)} \in [0, 1]$ is the final task score. The refiner compares high-scoring traces with low-scoring ones; for tasks with binary outcomes, this reduces to successes versus failures.

Step 2: Feedback-Driven Mutation. An offline LLM refiner inspects the partitioned traces to identify repeated error loops in failure trajectories and multi-step reasoning shortcuts in successful runs. Based on this feedback, the refiner generates a structured edit set $\Delta\mathcal{G}_k$ comprising two topological edit operations: ***Add***: Inserting missing verification nodes or edges; ***Delete***: Removing nodes or edges that repeatedly steer trajectories into failure or prevent progress.

Attribute revisions use the same edit interface: an edge is deleted and re-added with updated attribute values. This operation applies to any attribute defined by the chosen schema. The candidate graph is obtained by applying the proposed edits, $\mathcal{G}_k^{\text{cand}} = \mathcal{G}_{k-1} \oplus \Delta\mathcal{G}_k$, where $\oplus$ applies edits to a copy and performs any configured cycle repair.

Step 3: Validation Gating. To assess whether structural mutations improve performance beyond the training batch, a candidate that passes edit application and structural checks is evaluated on an independent validation set $\mathcal{D}_{\text{val}}$. Invalid candidates are discarded before validation rollout, leaving the retained graph and its cached validation score unchanged. The mean validation task score is computed as:

$$S_{\text{val}}(\mathcal{G}) = \frac{1}{|\mathcal{D}_{\text{val}}|} \sum_{(q,y)\in\mathcal{D}_{\text{val}}} S\big(f_{\text{solver}}(q \mid \mathcal{G}),\ y\big). \tag{4}$$

The initial graph is evaluated once to establish the reference score. For a structurally valid candidate, the retained graph is updated as follows:

$$\mathcal{G}_k = \begin{cases} \mathcal{G}_k^{\text{cand}}, & \text{if } S_{\text{val}}(\mathcal{G}_k^{\text{cand}}) \geq S_{\text{val}}(\mathcal{G}_{k-1}) \\ \mathcal{G}_{k-1}, & \text{otherwise} \end{cases}. \tag{5}$$

The gate retains candidates whose measured validation score matches or exceeds the cached score of the current graph. Section 5.4 examines these decisions under stochastic evaluation.

Step 4: Rejection Memory as a Safeguard. Iterative self-correction can repeatedly propose equivalent unsuccessful edits. If a candidate graph is rejected by the validation gate ($S_{\text{val}}(\mathcal{G}_k^{\text{cand}}) < S_{\text{val}}(\mathcal{G}_{k-1})$), we log the candidate graph and its proposed edits, together with the associated training trajectories $\mathcal{E}_k$ and validation outcomes, into a rejection memory $\mathcal{H}_{\text{rejected}}$. For the trajectory context supplied to the refiner, we concatenate the training trajectories and, if the configured maximum token length $L_{\max}$ is exceeded, discard tokens from the beginning while preserving the final $L_{\max}$ tokens in their original order. This retains the trajectory ending rather than an initial prefix; the resulting context is denoted as $\mathcal{C}_k$. When proposing edits for round $k+1$, the refiner receives $\mathcal{H}_{\text{rejected}}$ as negative evidence:

$$\Delta\mathcal{G}_{k+1} \sim P_{\text{refiner}}\big(\cdot \mid \mathcal{G}_k,\ \mathcal{C}_{k+1},\ \mathcal{H}_{\text{rejected}}\big). \tag{6}$$

The rejection history $\mathcal{H}_{\text{rejected}}$ helps the refiner avoid previously unsuccessful edits. Proposed changes are evaluated by the gate before they enter the retained graph.

4 Experimental Setup

Implementation details are provided in Appendix B.

Benchmarks. We evaluate procedural reasoning across seven benchmarks: **HotpotQA** (Yang et al., 2018) for multi-hop question answering with search tools; **MultiChallenge** (Deshpande et al., 2025) for instruction retention across multi-turn conversations; **GDPval** (Patwardhan et al., 2025) for open-ended professional tasks scored against expert rubrics; **ALFWorld** (Shridhar et al., 2021) for embodied household tasks with strict action ordering; **τ-bench** (Yao et al., 2024) for policy-compliant tool use under live user interaction; **BFCL** (Patil et al., 2025) for multi-turn function calling; and **EnterpriseArena** (Han et al., 2026) for long-horizon financial decision-making under delayed feedback and macroeconomic shocks. Dataset splits and preprocessing are detailed in Appendix B.1, and all metrics are defined in Appendix B.2.

Baselines. All methods share an identical ReAct solver (Yao et al., 2023b) and differ only in how procedural experience is stored and reused; every learning-based baseline consumes the same training trajectories as our self-evolution loop. Ordered by increasing structure, we compare: **Vanilla ReAct** (no memory) (Yao et al., 2023b), **MemoryBank** (Zhong et al., 2024), which maintains summarized experience with forgetting; **RAP** (Kagaya et al., 2024), which retrieves past trajectories as in-context exemplars; **ExpeL** (Zhao et al., 2024), which distills trajectories into natural-language insights; **AutoGuide** (Fu et al., 2024), which retrieves state-conditioned guidelines; **AWM** (Wang et al., 2025b), which induces linear workflows; and **KnowAgent** (Zhu et al., 2025), which maintains textual action-transition rules. Implementation and adaptation details for each baseline are given in Appendix B.3.

Models. We evaluate four LLMs: Claude Sonnet 4.6, Gemini 3.1 Pro, Gemini 3.5 Flash, and Grok 4.1 Fast. The guidance model and the offline refiner always share the same underlying LLM as the solver. All calls use greedy decoding (temperature 0) for reproducibility.

Procedural Graph Configuration. Online guidance uses the h=2 hop neighborhood of the localized node and a recent trajectory window of w=3. Different construction strategies are compared in Section 5.3 and formalized in Appendix D.2. Statistics of the graphs used for each benchmark (node and triplet counts, relation types, attribute coverage) are given in Appendix B.4.

5 Results

5.1 Main Results across Benchmarks and Models

Table 1: **Main results across LLMs and benchmarks.** Brackets give 95% confidence intervals; the best value is highlighted.

Model & Method	HotpotQA Acc. (↑)	MultiChallenge Acc. (↑)	GDPval Rubric Score (↑)	ALFWorld Success (↑)	τ-bench Pass@1 (↑)	BFCL v3 Acc. (↑)
Claude Sonnet 4.6						
Vanilla ReAct (Yao et al., 2023b)	74.60 [71.83, 77.37]	83.73 [78.06, 89.41]	46.23 [33.42, 59.04]	86.57 [80.76, 92.37]	66.09 [57.54, 74.64]	61.00 [51.29, 70.71]
MemoryBank (Zhong et al., 2024)	74.20 [71.50, 76.90]	89.16 [84.50, 93.82]	50.13 [37.04, 63.22]	86.57 [80.72, 92.41]	63.48 [54.78, 72.18]	62.00 [52.55, 71.45]
RAP (Kagaya et al., 2024)	71.80 [69.07, 74.53]	86.75 [81.56, 91.93]	40.97 [28.37, 53.56]	79.10 [72.28, 85.93]	60.87 [52.12, 69.62]	65.00 [55.68, 74.32]
ExpeL (Zhao et al., 2024)	75.20 [72.49, 77.91]	89.16 [84.45, 93.86]	48.22 [35.66, 60.78]	91.79 [87.30, 96.28]	71.30 [63.05, 79.56]	60.00 [50.46, 69.54]
AutoGuide (Fu et al., 2024)	**75.40 [72.72, 78.08]**	88.55 [83.71, 93.40]	41.39 [27.57, 55.21]	84.33 [78.12, 90.54]	65.22 [56.54, 73.89]	59.00 [49.19, 68.81]
AWM (Wang et al., 2025b)	73.90 [71.21, 76.59]	**89.76 [85.17, 94.34]**	40.61 [27.54, 53.68]	67.16 [59.23, 75.10]	66.96 [58.51, 75.40]	62.00 [52.56, 71.44]
KnowAgent (Zhu et al., 2025)	74.30 [71.59, 77.01]	87.35 [82.36, 92.34]	42.14 [29.50, 54.78]	87.31 [81.58, 93.05]	61.74 [52.67, 70.81]	60.00 [50.29, 69.71]
Procedural Graph (Ours)	74.50 [71.81, 77.19]	**89.76 [85.21, 94.31]**	**51.49 [38.44, 64.54]**	**93.28 [88.98, 97.59]**	**73.91 [65.84, 81.99]**	**67.00 [57.65, 76.35]**
Gemini 3.1 Pro						
Vanilla ReAct (Yao et al., 2023b)	85.90 [83.76, 88.04]	87.95 [83.00, 92.91]	56.39 [45.23, 67.55]	94.78 [91.03, 98.52]	72.17 [63.92, 80.43]	59.00 [49.33, 68.67]
MemoryBank (Zhong et al., 2024)	85.10 [82.90, 87.30]	95.18 [92.00, 98.36]	61.97 [50.54, 73.39]	85.07 [78.87, 91.28]	67.83 [59.38, 76.27]	63.00 [53.69, 72.31]
RAP (Kagaya et al., 2024)	83.60 [81.28, 85.92]	94.58 [91.17, 97.99]	71.37 [62.99, 79.76]	80.60 [73.87, 87.32]	73.04 [64.98, 81.11]	57.00 [47.20, 66.80]
ExpeL (Zhao et al., 2024)	86.00 [83.85, 88.15]	92.77 [88.84, 96.70]	64.69 [54.33, 75.06]	97.76 [95.25, 100.00]	64.35 [55.46, 73.24]	63.00 [53.60, 72.40]
AutoGuide (Fu et al., 2024)	85.80 [83.63, 87.97]	93.37 [89.54, 97.21]	56.18 [44.73, 67.62]	91.04 [86.17, 95.92]	65.22 [56.59, 73.84]	63.00 [53.60, 72.40]
AWM (Wang et al., 2025b)	85.10 [82.89, 87.31]	93.98 [90.45, 97.50]	63.85 [54.13, 73.57]	99.25 [97.78, 100.00]	67.83 [59.32, 76.33]	64.00 [54.41, 73.59]
KnowAgent (Zhu et al., 2025)	85.00 [82.82, 87.18]	95.18 [91.89, 98.47]	69.10 [60.25, 77.95]	95.52 [91.99, 99.06]	64.35 [55.42, 73.28]	61.00 [51.44, 70.56]
Procedural Graph (Ours)	**87.30 [85.18, 89.42]**	**95.78 [92.78, 98.79]**	**78.78 [74.42, 83.15]**	**100.00 [100.00, 100.00]**	**80.00 [72.83, 87.17]**	**66.00 [56.85, 75.15]**
Gemini 3.5 Flash						
Vanilla ReAct (Yao et al., 2023b)	83.10 [80.79, 85.41]	81.33 [75.33, 87.33]	59.33 [47.49, 71.16]	82.84 [76.42, 89.25]	31.30 [22.77, 39.84]	56.00 [46.51, 65.49]
MemoryBank (Zhong et al., 2024)	82.50 [80.14, 84.86]	89.16 [84.33, 93.99]	59.27 [47.81, 70.72]	76.87 [69.89, 83.84]	34.78 [26.03, 43.53]	54.00 [44.24, 63.76]
RAP (Kagaya et al., 2024)	82.30 [79.97, 84.63]	89.16 [84.45, 93.86]	60.23 [48.63, 71.82]	76.12 [69.00, 83.24]	37.39 [28.45, 46.34]	54.00 [44.11, 63.89]
ExpeL (Zhao et al., 2024)	83.40 [81.09, 85.71]	89.16 [84.42, 93.89]	62.45 [50.97, 73.93]	90.30 [85.34, 95.26]	32.17 [23.61, 40.73]	58.00 [48.50, 67.50]
AutoGuide (Fu et al., 2024)	83.10 [80.82, 85.38]	89.16 [84.32, 93.99]	50.29 [37.54, 63.04]	80.60 [73.87, 87.32]	26.09 [18.08, 34.10]	56.00 [46.01, 65.99]
AWM (Wang et al., 2025b)	82.50 [80.14, 84.86]	**91.57 [87.38, 95.75]**	50.25 [37.54, 62.95]	82.09 [75.55, 88.63]	33.91 [25.37, 42.46]	52.00 [42.24, 61.76]
KnowAgent (Zhu et al., 2025)	82.70 [80.39, 85.01]	**91.57 [87.26, 95.88]**	54.00 [41.68, 66.31]	81.34 [74.86, 87.83]	38.26 [29.23, 47.29]	58.00 [48.31, 67.69]
Procedural Graph (Ours)	**84.50 [82.30, 86.70]**	**91.57 [87.33, 95.80]**	**64.42 [53.66, 75.19]**	**94.03 [90.03, 98.03]**	**44.35 [35.20, 53.50]**	**67.00 [57.89, 76.11]**
Grok 4.1 Fast						
Vanilla ReAct (Yao et al., 2023b)	72.70 [69.89, 75.51]	68.07 [61.04, 75.10]	57.23 [47.11, 67.34]	26.12 [18.64, 33.60]	64.35 [55.58, 73.11]	52.00 [42.29, 61.71]
MemoryBank (Zhong et al., 2024)	70.70 [67.84, 73.56]	84.94 [79.47, 90.41]	55.19 [44.40, 65.98]	23.88 [16.79, 30.97]	58.26 [49.48, 67.04]	56.00 [46.10, 65.90]
RAP (Kagaya et al., 2024)	70.20 [67.32, 73.08]	80.12 [74.01, 86.24]	66.79 [58.25, 75.33]	14.93 [8.90, 20.95]	64.35 [55.41, 73.28]	55.00 [45.22, 64.78]
ExpeL (Zhao et al., 2024)	73.50 [70.81, 76.19]	84.94 [79.53, 90.35]	65.66 [57.00, 74.32]	**42.54 [34.08, 50.99]**	63.48 [54.72, 72.24]	52.00 [42.36, 61.64]
AutoGuide (Fu et al., 2024)	70.00 [67.15, 72.85]	80.72 [74.73, 86.72]	53.58 [42.90, 64.26]	30.60 [22.74, 38.45]	61.74 [52.90, 70.57]	56.00 [45.99, 66.01]
AWM (Wang et al., 2025b)	74.30 [71.60, 77.00]	83.73 [78.26, 89.21]	61.55 [52.66, 70.44]	17.16 [10.84, 23.49]	63.48 [54.87, 72.09]	52.00 [42.20, 61.80]
KnowAgent (Zhu et al., 2025)	73.10 [70.34, 75.86]	83.13 [77.55, 88.72]	68.15 [61.15, 75.16]	18.66 [12.11, 25.20]	**68.70 [60.45, 76.94]**	54.00 [44.44, 63.56]
Procedural Graph (Ours)	**74.40 [71.62, 77.18]**	**86.75 [81.74, 91.76]**	**71.19 [65.24, 77.14]**	39.55 [31.11, 47.99]	67.83 [59.27, 76.39]	**60.00 [50.51, 69.49]**

Table 1 compares the Procedural Graph against seven baselines across six benchmarks and four LLM families, all using the same ReAct solver. PG ranks first or joint first in 21 of 24 model–benchmark settings. Compared with the strongest baseline in each setting, PG records 19 wins, two ties, and three losses (one-sided exact binomial sign test excluding ties, $p = 4.3 \times 10^{-4}$). Its largest margins are on BFCL v3 with Gemini 3.5 Flash (67.00% vs. 58.00%, +9.00 points), GDPval with Gemini 3.1 Pro (78.78 vs. 71.37, +7.41 points), and τ-bench with the same model (80.00% vs. 73.04%, +6.96 points). Baseline rankings vary across tasks and models, with no single method consistently placing second. These results suggest that combining conditional guidance, reusable action sequences, and explicit transitions in a connected graph is useful across diverse settings.

The gains also extend across model families. On GDPval and BFCL v3, PG outperforms every baseline under all four LLMs, indicating that its advantage on these tasks is not confined to a particular solver. On MultiChallenge, PG ranks first or joint first across all four models, matching AWM on

Claude Sonnet 4.6 and both AWM and KnowAgent on Gemini 3.5 Flash. HotpotQA shows a different pattern: margins over the strongest baseline range from -0.90 to $+1.30$ points. The magnitude of the gains therefore varies substantially across benchmarks.

5.2 Long-Horizon Decision Making and Resilience

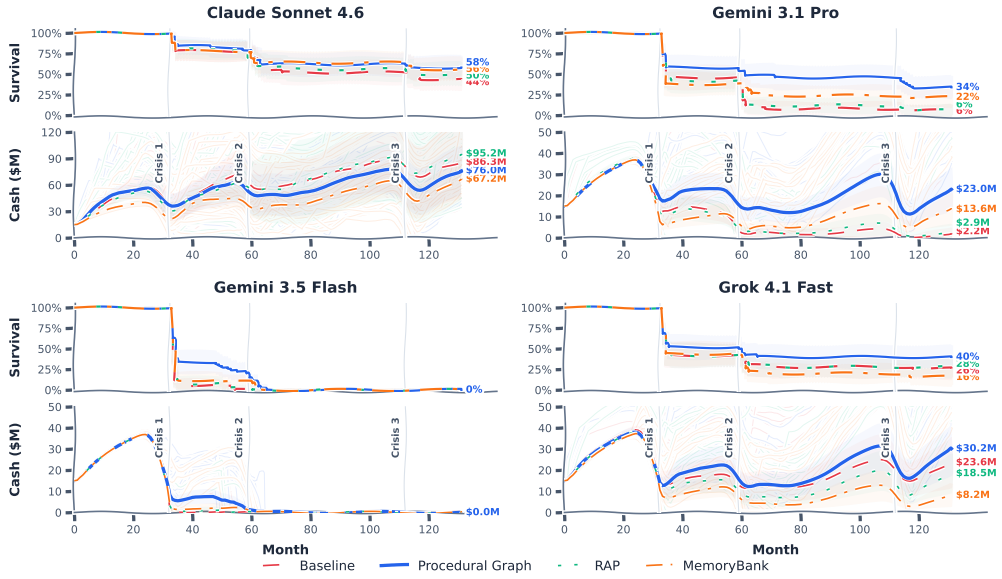

Figure 3: **Ensemble cash trajectories and Kaplan–Meier survival curves across four LLMs.** Bold lines and shaded areas show means and 95% confidence intervals. Vertical lines mark macroeconomic crises. Colors identify PG (blue), the baseline (red), and memory-based methods (green, orange).

To evaluate agent resilience on long-horizon tasks, we deploy agents in EnterpriseArena (Han et al., 2026), a simulator in which the agent makes monthly financial decisions over up to 132 months under strict liquidity constraints and three scheduled crises that are not disclosed to the agent. Figure 3 plots the Kaplan-Meier survival curves and the ensemble cash trajectories for all four models; complete metrics are reported in Table 8 in Appendix C. PG achieves the highest or joint-highest full-horizon survival and the longest average lifespan across all four LLMs. It raises survival from 44.0% to 58.0% for Claude Sonnet 4.6, from 6.0% to 34.0% for Gemini 3.1 Pro, and from 26.0% to 40.0% for Grok 4.1 Fast, where it also delivers the best average enterprise score ($39.62M).

What changes under guidance is *which* tools are called and when, rather than simply how many. The unguided Gemini 3.5 Flash baseline repeatedly queries cash and market state within a single turn, adding redundant observations to its context. It issues 18.94 tool calls per month, which the graph reduces to 12.53 while improving the average enterprise score. On Claude Sonnet 4.6 and Gemini 3.1 Pro, tool calls instead increase from 0.13 to 0.36 and from 0.89 to 3.18 per month, respectively, while survival also improves. In these settings, the graph guides the agent to run forecast and market checks before a financing decision (Appendix C.2).

The behavior that does track survival across all four models is anticipatory fundraising. Because capital arrives one to six months after it is requested, surviving a crisis requires asking well before liquidity runs out. The full trajectories show that the unguided Gemini 3.5 Flash baseline does not initiate fundraising sufficiently early, whereas PG-guided agents initiate requests during stable months. Average capital raised is $0.00M for the Flash baseline, compared with $9.39M for PG-guided Flash and $30.11M for PG-guided Grok 4.1 Fast. Appendix C.3 contrasts step-by-step traces of the unguided baseline, a memory-summarization agent, and a PG-guided agent entering the first crisis.

5.3 Procedural Graph Construction Strategies

We compare five PG construction strategies against the unguided baseline, spanning expert versus minimal initialization and fixed, one-time, or iterative refinement. Modes 3 and 5 instantiate the full evolution loop, while Modes 1, 2, and 4 provide fixed or one-time alternatives (supplementary results and mode definitions are provided in Appendix D). The performance metrics of HotpotQA and MultiChallenge are presented in Table 2.

Table 2: **PG construction modes on HotpotQA and MultiChallenge.** Self-evolution improves PGs and repairs a flawed expert prior.

Construction Mode	HotpotQA		MultiChallenge				
	Ans EM	Ans F1	Inference Memory	Instruction Retention	Reliable Versioned Editing	Self Coherence	Overall
Unguided Baseline (w/o PG)	58.8	71.21	86.96	86.67	71.43	**100.00**	87.50
Mode 1: Hand-crafted Expert	62.80	76.61	52.17	66.67	42.86	72.73	58.93
Mode 2: Expert + Static Update	63.80	77.16	47.83	53.33	28.57	81.82	53.57
Mode 3: Expert + Self-Evolution	63.10	76.34	**95.65**	**100.00**	**85.71**	81.82	**92.86**
Mode 4: Scratch + Static Build	55.40	69.49	91.30	86.67	**85.71**	90.91	89.29
Mode 5: Scratch + Self-Evolution	**66.30**	**78.79**	**95.65**	93.33	71.43	90.91	91.07

We compare initialization from an expert prior (Modes 1–3) with initialization from scratch (Modes 4–5). On HotpotQA, Mode 5 (Scratch + Self-Evolution) achieves the highest performance across all configurations, scoring 78.79% Ans F1 and 66.30% Ans EM (gains of 7.58 F1 points and 7.50 EM points over the unguided baseline). On MultiChallenge, which requires retaining multiple constraints across dialogue turns, Mode 5 achieves an Overall Success Rate of 91.07% without a human prior, while Mode 3 (Expert + Self-Evolution) performs best at 92.86%.

The loop also self-corrects from a flawed expert prior. On MultiChallenge, using the hand-crafted expert graph (Mode 1) lowers success from 87.50% to 58.93%. A single offline update (Mode 2) further lowers success to 53.57%. The iterative configuration (Mode 3), which combines fresh execution feedback with validation gating, recovers to 92.86%, a gain of 33.93 points over the expert initialization. The loop therefore recovers from an expert prior that initially reduces performance (Appendix F.2). Resource and stability statistics for all five modes, including a 45.7% reduction in parsing failures and the token overhead carried by expert-written guidance, are reported in Appendix D.1.

5.4 Procedural Graph Self-Evolution

We examine ten rounds of PG self-evolution (Section 3.3) on EnterpriseArena, where the agent manages liquidity through successive macroeconomic crises. Figure 4 tracks the resulting changes in lifespan and capital raised; Appendix E reports the per-round results.

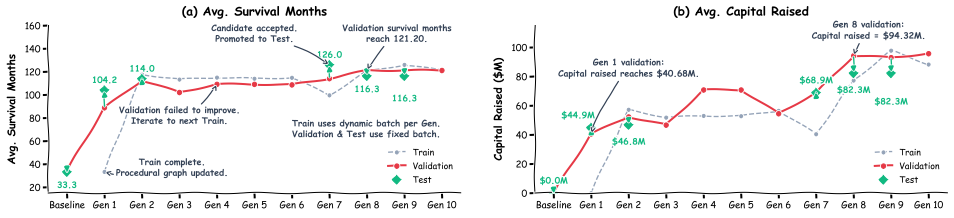

Figure 4: **Mean lifespan and capital raised across ten rounds of PG self-evolution.** Gray dashed lines show training results; red lines show validation results; and green diamonds show test results for the baseline and accepted checkpoints.

On the validation split, the unguided baseline has a full-horizon survival rate of 0.0% and a mean lifespan of 34.8 months. Since capital takes one to six months to arrive, fundraising must begin before cash runs out. Validation performance improves in several distinct rounds, separated by periods with no accepted update. Round 1 discovers the sequential backbone that guides the agent to audit cash and forecast runway before a financing decision, producing the largest single jump ($0.0\% \rightarrow 45.0\%$ validation survival); Round 2 adds `recall_notes` to reuse the notes saved by

the Round 1 graph and lifts survival to 80.0%, with tool usage falling from 17.23 to 3.08 calls per month relative to the unguided baseline. Rounds 3–6 produce no committed update; one candidate fails structural verification before rollout. Round 7 prunes the `pass_action` branch, and Round 8 introduces the administrative bypass described in Appendix E.3. Validation survival reaches 90.0% in Round 8 and remains at that level in Round 9 before Round 10 is rejected and the loop terminates.

The test results distinguish the returned graph from intermediate candidates. The returned graph reaches 85.0% test survival against the baseline's 0.0% (Fisher's exact $p = 2.6 \times 10^{-8}$), while the best single round observed during the search reached 95.0%; we report the former, since quoting the latter would amount to selecting on the test set. With 20 episodes per split, individual accept/reject decisions turn on one or two episodes and should be read as a search trace rather than as significance tests. Topological changes and cross-dataset validation are detailed in Appendices E.3 and E.4.

5.5 Efficiency Analysis

Table 3 examines two key design choices behind our guidance mechanism: what portion of the graph the agent sees (full graph vs. localized subgraph) and how it is consumed (raw injection vs. generative guidance). The three PG configurations compare generative guidance with raw injection for the full graph and assess localization under generative guidance, alongside a no-graph baseline. All configurations use Gemini 3.5 Flash and a shared solver prompt template; the PG configurations use the same underlying graph (Appendix B.5).

Table 3: **PG usage ablation with Gemini 3.5 Flash.** On fixed subsets, we report MultiChallenge accuracy, GDPval rubric score, and ALFWorld success rate (all ↑), alongside average tokens and solver steps per sample (both ↓).

Graph Configuration	MultiChallenge			GDPval			ALFWorld		
	Acc.	Avg. Tok	Avg. Step	Rubric	Avg. Tok	Avg. Step	Success	Avg. Tok	Avg. Step
Baseline (no graph)	80.27	6,629	3.87	54.80	275,638	28.20	72.58	18,055	21.84
Full graph, raw injection	86.60	10,164	4.54	57.17	264,680	33.55	70.34	21,062	25.00
Full graph, generative	87.35	14,434	**3.08**	56.75	448,972	22.07	54.48	96,360	30.05
Subgraph, generative (Ours)	**89.31**	12,295	4.22	**63.99**	367,738	**18.57**	**81.53**	28,064	**18.80**

Injecting the raw full graph improves performance on structured dialogue (MultiChallenge rises from 80.27 to 86.60) but lowers success on embodied execution (ALFWorld drops from 72.58 to 70.34). Full-graph generative guidance further reduces ALFWorld success to 54.48 while increasing token consumption. These results favor guidance grounded in the agent's local graph neighborhood over guidance generated from the full graph. Given the same graph, the localized generative configuration achieves the highest performance across all three benchmarks (89.31, 63.99, and 81.53), exceeding the best alternative in each setting, including the no-graph baseline, by 2.0, 6.8, and 9.0 points, respectively.

Localization reduces total tokens relative to full-graph generative guidance on all three benchmarks: by 70.9% on ALFWorld, 18.1% on GDPval, and 14.8% on MultiChallenge. It also shortens trajectories on GDPval and ALFWorld. The additional guidance call introduces token overhead relative to the no-graph baseline. On GDPval and ALFWorld, localized guidance reduces average solver steps from 28.20 to 18.57 and from 21.84 to 18.80, respectively, while total token consumption remains 33.4% and 55.4% higher.

6 Conclusion

We introduced the Procedural Graph, an explicit and editable representation of procedural knowledge that gives LLM agents a queryable answer to what to do next. PG connects the agent's current progress with relevant transitions and execution advice while preserving reasoning flexibility. Across tasks and model families, it delivers consistent gains over memory-based baselines. Self-evolution builds effective graphs from minimal initializations and repairs expert priors that initially hinder performance. These results support learning and revising procedural knowledge from execution feedback without updating model weights. Guidance increases token use even when it reduces solver steps; future work could reuse guidance across steps or generate it selectively. Evaluating transfer across solvers and tool interfaces would clarify how widely the learned procedures can be reused.

AI USE STATEMENT

We used generative AI tools to edit the language of the manuscript for grammar, clarity, and readability. AI assistance in manuscript preparation was limited to editorial tasks; it did not contribute to the research ideas, methodology, experiments, or interpretation of results. The authors take full responsibility for the content of this paper.

REFERENCES

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pp. 17682–17690, 2024.

Kaustubh Deshpande, Ved Sirdeshmukh, Johannes Baptist Mols, Lifeng Jin, Ed-Yeremai Hernandez-Cardona, Dean Lee, Jeremy Kritz, Willow E Primack, Summer Yue, and Chen Xing. MultiChallenge: A realistic multi-turn conversation evaluation benchmark challenging to frontier LLMs. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 18632–18702, 2025.

Yu Du, Fangyun Wei, and Hongyang Zhang. AnyTool: Self-reflective, hierarchical agents for large-scale API calls. *arXiv preprint arXiv:2402.04253*, 2024.

Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. From local to global: A graph RAG approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*, 2024.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. Memp: Exploring agent procedural memory. *arXiv preprint arXiv:2508.06433*, 2025.

Yao Fu, Dong-Ki Kim, Jaekyeom Kim, Sungryull Sohn, Lajanugen Logeswaran, Kyunghoon Bae, and Honglak Lee. AutoGuide: Automated generation and selection of context-aware guidelines for large language model agents. *Advances in Neural Information Processing Systems*, 37: 119919–119948, 2024.

Linfeng Gao, Yaoxiang Wang, Minlong Peng, Jialong Tang, Yuzhe Shang, Mingming Sun, and Jinsong Su. Tool graph retriever: Exploring dependency graph-based tool retrieval for large language models. *arXiv preprint arXiv:2508.05152*, 2025.

Yi Han, Yan Wang, Lingfei Qian, Haohang Li, Yupeng Cao, Yueru He, Xueqing Peng, Nanhan Shen, Yitao Xu, Yankai Chen, et al. Can LLM agents be CFOs? benchmarking long-horizon resource allocation in an uncertain enterprise environment. *arXiv preprint arXiv:2603.23638*, 2026.

Yue Huang, Jiawen Shi, Yuan Li, Chenrui Fan, Siyuan Wu, Qihui Zhang, Yixin Liu, Pan Zhou, Yao Wan, Neil Gong, et al. MetaTool benchmark for large language models: Deciding whether to use tools and which to use. In *International Conference on Learning Representations*, volume 2024, pp. 42978–43007, 2024.

Yan Jiang, Hao Zhou, LiZhong GU, Ai Han, and TianLong Li. NaviAgent: Bilevel planning on tool navigation graph for large-scale orchestration. *arXiv preprint arXiv:2506.19500*, 2025.

Tomoyuki Kagaya, Thong Jing Yuan, Yuxuan Lou, Jayashree Karlekar, Sugiri Pranata, Akira Kinose, Koki Oguri, Felix Wick, and Yang You. RAP: Retrieval-augmented planning with contextual memory for multimodal LLM agents. In *NeurIPS 2024 Workshop on Open-World Agents*, 2024.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. API-Bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 3102–3116, 2023.

Xukun Liu, Zhiyuan Peng, Xiaoyuan Yi, Xing Xie, Lirong Xiang, Yuchen Liu, and Dongkuan Xu. ToolNet: Connecting large language models with massive tools via tool graph. *arXiv preprint arXiv:2403.00839*, 2024a.

Zhaoyang Liu, Zeqiang Lai, Zhangwei Gao, Erfei Cui, Ziheng Li, Xizhou Zhu, Lewei Lu, Qifeng Chen, Yu Qiao, Jifeng Dai, et al. ControlLLM: Augment language models with tools by searching on graphs. In *European Conference on Computer Vision*, pp. 89–105. Springer, 2024b.

Elias Lumer, Pradeep Honaganahalli Basavaraju, Myles Mason, James A Burke, and Vamse Kumar Subbiah. Graph RAG-tool fusion. *arXiv preprint arXiv:2502.07223*, 2025.

Zheng Nie, Ruolin Shen, Xinlei Yu, Bo Yin, Jiangning Zhang, and Xiaobin Hu. SkillGraph: Self-evolving multi-agent collaboration with multimodal graph topology. *arXiv preprint arXiv:2604.17503*, 2026.

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

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. Gorilla: Large language model connected with massive APIs. *Advances in Neural Information Processing Systems*, 37:126544–126565, 2024.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. The Berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*, 2025.

Tejal Patwardhan, Rachel Dias, Elizabeth Proehl, Grace Kim, Michele Wang, Olivia Watkins, Simón Posada Fishman, Marwan Aljubeh, Phoebe Thacker, Laurance Fauconnet, Natalie S Kim, et al. GDPval: Evaluating AI model performance on real-world economically valuable tasks. *SuperIntelligence-Robotics-Safety & Alignment*, 2(4), 2025.

Archiki Prasad, Alexander Koller, Mareike Hartmann, Peter Clark, Ashish Sabharwal, Mohit Bansal, and Tushar Khot. ADaPT: As-needed decomposition and planning with language models. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pp. 4226–4252, 2024.

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. In *International Conference on Learning Representations*, volume 2024, pp. 9695–9717, 2024.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. Towards completeness-oriented tool retrieval for large language models. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, pp. 1930–1940, 2024.

Preston Rasmussen, Pavlo Paliychuk, Travis Beauvais, Jack Ryan, and Daniel Chalef. Zep: a temporal knowledge graph architecture for agent memory. *arXiv preprint arXiv:2501.13956*, 2025.

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.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. HuggingGPT: Solving AI tasks with ChatGPT and its friends in Hugging Face. *Advances in Neural Information Processing Systems*, 36:38154–38180, 2023.

Yongliang Shen, Kaitao Song, Xu Tan, Wenqi Zhang, Kan Ren, Siyu Yuan, Weiming Lu, Dongsheng Li, and Yueting Zhuang. TaskBench: Benchmarking large language models for task automation. *Advances in Neural Information Processing Systems*, 37:4540–4574, 2024.

Taiwei Shi, Sihao Chen, Bowen Jiang, Linxin Song, Longqi Yang, and Jieyu Zhao. Experiential reinforcement learning. *arXiv preprint arXiv:2602.13949*, 2026.

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

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Cote, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations*, 2021.

Theodore Sumers, Shunyu Yao, Karthik R Narasimhan, and Thomas L Griffiths. Cognitive architectures for language agents. *Transactions on Machine Learning Research*, 2023.

Zeyi Sun, Ziyu Liu, Yuhang Zang, Yuhang Cao, Xiaoyi Dong, Tong Wu, Dahua Lin, and Jiaqi Wang. SE-Agent: Self-evolving computer use agent with autonomous learning from experience. *arXiv preprint arXiv:2508.04700*, 2025.

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

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: long papers)*, pp. 2609–2634, 2023b.

Renxi Wang, Xudong Han, Lei Ji, Shu Wang, Timothy Baldwin, and Haonan Li. ToolGen: Unified tool retrieval and calling via generation. In *International Conference on Learning Representations*, volume 2025, pp. 73473–73498, 2025a.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. *arXiv preprint arXiv:2402.01030*, 2024a.

Zhaoyang Wang, Qianhui Wu, Xuchao Zhang, Chaoyun Zhang, Wenlin Yao, Fazle Elahi Faisal, Baolin Peng, Si Qin, Suman Nath, Qingwei Lin, et al. WebXSkill: Skill learning for autonomous web agents. *arXiv preprint arXiv:2604.13318*, 2026.

Zhiruo Wang, Daniel Fried, and Graham Neubig. TroVE: Inducing verifiable and efficient toolboxes for solving programmatic tasks. *arXiv preprint arXiv:2401.12869*, 2024b.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *International Conference on Machine Learning*, pp. 63897–63911. PMLR, 2025b.

Brandon T Willard and Rémi Louf. Efficient guided generation for large language models. *arXiv preprint arXiv:2307.09702*, 2023.

Rong Wu, Xiaoman Wang, Jianbiao Mei, Pinlong Cai, Daocheng Fu, Cheng Yang, Licheng Wen, Xuemeng Yang, Yufan Shen, Yuxin Wang, et al. EvolveR: Self-evolving LLM agents through an experience-driven lifecycle. *arXiv preprint arXiv:2510.16079*, 2025.

Xixi Wu, Yifei Shen, Caihua Shan, Kaitao Song, Siwei Wang, Bohang Zhang, Jiarui Feng, Hong Cheng, Wei Chen, Yun Xiong, et al. Can graph learning improve planning in LLM-based agents? *Advances in Neural Information Processing Systems*, 37:5338–5383, 2024.

Ruixuan Xiao, Wentao Ma, Ke Wang, Yuchuan Wu, Junbo Zhao, Haobo Wang, Fei Huang, and Yongbin Li. FlowBench: Revisiting and benchmarking workflow-guided planning for LLM-based agents. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 10883–10900, 2024.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-Mem: Agentic memory for LLM agents. *Advances in Neural Information Processing Systems*, 38:17577–17604, 2026.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, pp. 2369–2380, 2018.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 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.

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

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

Kexun Zhang, Hongqiao Chen, Lei Li, and William Wang. Don't fine-tune, decode: Syntax error-free tool use via constrained decoding. *arXiv preprint arXiv:2310.07075*, 2023.

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

Boyuan Zheng, Michael Y Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, et al. SkillWeaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*, 2025.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. MemoryBank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pp. 19724–19731, 2024. Issue 17.

Yuqi Zhu, Shuofei Qiao, Yixin Ou, Shumin Deng, Shiwei Lyu, Yue Shen, Lei Liang, Jinjie Gu, Huajun Chen, and Ningyu Zhang. KnowAgent: Knowledge-augmented planning for LLM-based agents. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 3709–3732, 2025.

A Extended Related Work and Comparison

This appendix expands Section 2. It covers procedural memory (Appendix A.1), action selection (Appendix A.2), structured planning (Appendix A.3), and self-improvement from trajectories (Appendix A.4). A comparison along eight design dimensions appears in Appendix A.5.

A.1 Procedural Graphs as Procedural Memory: A CoALA View

Sumers et al. (2023) propose CoALA (Cognitive Architectures for Language Agents), which adapts the classical memory taxonomy of ACT-R and SOAR to LLM-based agents: *working memory* holds the current context, *episodic memory* holds past experiences, *semantic memory* holds factual knowledge, and *procedural memory* holds the skills and procedures that govern how to act. This taxonomy distinguishes the roles of different agent memories. Retrieval augmentation, including GraphRAG and its variants, operates on semantic memory; trajectory-based reflection methods such as Reflexion (Shinn et al., 2023) and ExpeL (Zhao et al., 2024) operate on episodic memory. Procedural memory is the quadrant that has received the least explicit treatment: it remains largely implicit in model weights, or is scattered across ad hoc artifacts such as prompt templates, skill libraries, and workflow scripts. PG implements CoALA's procedural-memory module by storing state-conditioned action transitions in a structure that can be retrieved and updated.

A.2 LLM Agents and Action Selection

The dominant execution template is reason-then-act interleaving, established by ReAct (Yao et al., 2023b) and extended along several axes. Reflexion (Shinn et al., 2023) inserts verbal self-criticism between trials; Toolformer (Schick et al., 2023) shows that tool-call decisions can be learned by self-supervised filtering of LM-generated API calls; Tree-of-Thoughts (Yao et al., 2023a) and Graph-of-Thoughts (Besta et al., 2024) generalize chain-style reasoning into search over branching alternatives; and Plan-and-Solve (Wang et al., 2023b) and ADaPT (Prasad et al., 2024) separate planning from execution, with the latter recursively decomposing sub-tasks only once the executor fails. A parallel line redesigns the action space itself rather than the control flow over it: CodeAct (Wang et al., 2024a) unifies actions as executable Python so as to inherit the compositionality of a programming language, and HuggingGPT (Shen et al., 2023) treats models hosted on Hugging Face as callable tools coordinated by an LLM controller.

A third body of work scales the tool catalog. ToolLLM (Qin et al., 2024) contributes a 16,464-API benchmark together with a depth-first decision tree for tool selection; Gorilla (Patil et al., 2024) fine-tunes LLaMA on APIBench with retrieval-augmented training; AnyTool (Du et al., 2024) adds a hierarchical three-tier API retriever; and ToolGen (Wang et al., 2025a) collapses retrieval and invocation into next-token generation via virtual tool tokens. API-Bank (Li et al., 2023) and MetaTool (Huang et al., 2024) supply complementary evaluation axes covering when to invoke a tool and which one to invoke.

These methods generally leave admissible transitions implicit, relying on the model to select the next action from in-context information. The resulting failure modes are well documented, and include planning hallucination (Zhu et al., 2025; Xiao et al., 2024), trajectory drift on long-horizon tasks, and repetitive loops when execution feedback is ambiguous. These failures motivate the explicit structural priors discussed next.

A.3 Structured Priors for Agent Planning

One approach is to represent procedures explicitly. KnowAgent (Zhu et al., 2025) maintains a textual action knowledge base of admissible action rules and pairs it with knowledgeable self-learning to constrain the agent's action path during trajectory synthesis. FlowBench (Xiao et al., 2024) formalizes workflow knowledge in three formats, text, code, and flowchart, and shows empirically across six domains and 51 scenarios that flowcharts reduce planning hallucination most effectively. AFlow (Zhang et al., 2025) recasts workflow construction as Monte Carlo Tree Search over code-represented graphs whose nodes are LLM-invoking operators. Decoding constraints provide another form of structure: TOOLDEC (Zhang et al., 2023) compiles tool syntax schemas into finite-state automata and constrains generation to syntactically valid calls, using the efficient FSM-guided gener-

ation algorithm of Outlines (Willard & Louf, 2023). Such methods guarantee surface-form validity but say nothing about whether an action is semantically admissible given the task state.

Other methods organize tool collections as graphs. ToolNet (Liu et al., 2024a) mines a directed tool-transition graph from LLM-generated trajectories and lets the agent walk it at inference time. ControlLLM (Liu et al., 2024b) pre-builds a tool dependency graph from parameter-type matching and introduces *Thoughts-on-Graph* search over it. COLT (Qu et al., 2024) targets retrieval completeness through a dual-view query-tool-scene graph trained with LightGCN and contrastive losses. Graph RAG-Tool Fusion (Lumer et al., 2025) hybridizes vector retrieval with graph traversal over a hand-designed tool knowledge graph, extending GraphRAG (Edge et al., 2024) to tool selection. The Tool Graph Retriever (Gao et al., 2025) learns a tool-dependency discriminator and propagates embeddings over the resulting graph, while NaviAgent (Jiang et al., 2025) fuses API schema structure with historical invocations into a continuously evolving heterogeneous dependency graph. On the evaluation side, TaskBench (Shen et al., 2024) provides a graph-structured tool-automation benchmark with explicit node and edge scoring, and GNN4TaskPlan (Wu et al., 2024) demonstrates that GNN-based sub-task selection improves over LLM-only planning on it. SkillGraph (Nie et al., 2026) addresses multimodal multi-agent collaboration: it retrieves reasoning skills from an evolving skill bank and predicts a query-conditioned communication graph over agents.

These methods use graphs for different purposes: tool-transition and dependency graphs support tool selection, workflow graphs organize operator execution, and SkillGraph models communication among agents. Conditional action knowledge also appears in KnowAgent's textual rules and FlowBench's branching workflows. PG combines a graph over tool calls and reasoning steps with condition, guidance, and pitfall attributes on transitions. At each step, it localizes the current procedure and verbalizes the surrounding neighborhood. Per-step retrieval is also used by AutoGuide, which selects state-matched guidelines (Appendix B.3); PG instead retrieves connected transitions and supports explicit edits to their topology and attributes.

A.4 Self-Improving Agents from Trajectories

Another line learns reusable knowledge from the agent's execution history, often through verbal reflection and episodic memory. Reflexion (Shinn et al., 2023) writes self-critiques into an episodic buffer and re-attempts the task; Generative Agents (Park et al., 2023) maintain a memory stream ranked by recency, importance, and relevance, with periodic reflection for abstraction; MemoryBank (Zhong et al., 2024) keeps a long-term store of experience summaries governed by an Ebbinghaus-inspired forgetting schedule; and RAP (Kagaya et al., 2024) retrieves whole past trajectories as in-context exemplars. Two methods push toward explicitly contrastive distillation: ExpeL (Zhao et al., 2024) contrasts success and failure pairs to extract natural-language insights, and AutoGuide (Fu et al., 2024) sharpens these into *context-aware guidelines* of explicit conditional form ("in context X, action Y is appropriate") retrieved at test time from the agent's current state. ERL (Shi et al., 2026) uses reflection to guide a second attempt, then trains the base policy to retain the resulting improvements.

A second family stores reusable skills and workflows. Voyager (Wang et al., 2023a) maintains an ever-growing library of Minecraft skills indexed by embeddings of their natural-language descriptions, retrieved top-k per task; TroVE (Wang et al., 2024b) induces a verified Python toolbox and trims it to stay compact; AWM (Wang et al., 2025b) induces reusable workflows combining natural-language descriptions with program-form actions and adds them to prompt memory in both offline and online modes; and SkillWeaver (Zheng et al., 2025) and WebXSkill (Wang et al., 2026) refine skill discovery and execution for web agents. Several frameworks explicitly model the lifecycle of procedural memory: MemP (Fang et al., 2025) formalizes build, retrieve, and update as an optimization target, distilling trajectories into both fine-grained step instructions and higher-level script abstractions; EvolveR (Wu et al., 2025) closes the loop with offline self-distillation, online retrieval of strategic principles, and policy reinforcement; and SEAgent (Sun et al., 2025) learns computer-use policies from autonomously collected experience. A-Mem (Xu et al., 2026) and Zep/Graphiti (Rasmussen et al., 2025) bring knowledge-graph-style structure to agent memory, though their focus is episodic and semantic rather than procedural.

Among these methods, AutoGuide (Fu et al., 2024) is the closest to our approach. Like our edge attributes, its conditional guidelines associate situations with actions. PG connects these transitions in

a typed graph, supporting structural retrieval, dependency inspection, and refinement through graph edits. MemP shares our emphasis on lifecycle operations over procedural memory, storing trajectories and script-like abstractions for retrieval without an explicit procedure graph. The seven baselines we compare empirically in Section 4 are drawn from across this design space, spanning episodic summarization (MemoryBank), trajectory retrieval (RAP), insight distillation (ExpeL), conditional guidelines (AutoGuide), workflow induction (AWM), and textual transition rules (KnowAgent); Table 6 summarizes their storage and injection designs.

A.5 Detailed Comparison Table

Table 4 compares 24 representative methods with PG along eight design dimensions. It summarizes the representations and access mechanisms described in the cited work; the baseline configurations used in our experiments are specified separately in Appendix B.3.

The comparison highlights differences in what is stored, how it is accessed, and what can be updated. Textual rules, code libraries, workflows, and graphs each preserve useful procedural structure. For example, KnowAgent supplies action-transition knowledge as text in the prompt, while FlowBench includes flowcharts with branch conditions. AFlow searches over code-represented operator workflows, and tool graphs support navigation through tool catalogs. PG stores transitions between tool calls and reasoning steps, with conditions, guidance, and pitfalls attached to edges, and retrieves a neighborhood around the current procedure.

The target of improvement also differs across methods. Some revise insights, skills, or workflows; ToolNet derives tool transitions from trajectories, while SkillGraph couples skill-bank updates with query-conditioned communication among agents. PG updates the procedure graph itself through node and edge edits. Its contribution is this combination of attributed transitions, localized guidance, and structural refinement, rather than graph structure or conditional knowledge alone.

Column Definitions. "Form" describes the representation used for action selection, memory, or coordination; "Granularity" is the unit size of stored knowledge; and "Source" describes how the structure is obtained. "Updatable" indicates whether the structure can be revised from new trajectories, and "Retrieval" specifies the inference-time access mode. "Edge Semantics" describes the meaning of edges or relations, if any, while "Scope" identifies the intended deployment setting. "Graph" distinguishes explicit workflow, tool, or communication graphs (yes), auxiliary hierarchies or transitions encoded only in text (partial), and representations without explicit graph organization (no). FlowBench is shown in its flowchart form. For KnowAgent, "Updatable" refers to the action knowledge base, not model self-training. In our implementation, PG edge attributes comprise conditions, guidance, and pitfalls.

B Experimental Details

This appendix provides the dataset splits, evaluation metrics, baseline implementation details, prompt templates, and the full self-evolution algorithm (Appendix B.6).

B.1 Datasets and Splits

Table 5 summarizes the sample counts and splits used in all experiments.

During self-evolution, the training split is processed in sequential strides of $S{=}100$ samples on HotpotQA and $S{=}20$ on MultiChallenge. The validation set $\mathcal{D}_{\text{val}}$ consumed by the acceptance gate is held out separately from both splits above and never overlaps the test set; its size is $1{,}000$ for HotpotQA and 100 for MultiChallenge (Appendix D.3), and 20 episodes for EnterpriseArena. For EnterpriseArena, each configuration runs 50 training and 50 test episodes; the full environment mechanics are given in Appendix C.1.

B.2 Evaluation Metrics

For **HotpotQA**, the main comparison reports LLM-judged answer accuracy: a Gemini 3.1 Pro judge model receives the question, the gold answer, and the agent's answer, and returns a binary equiva-

Table 4: **Comparison of representative methods along eight design dimensions.** Column definitions and method-specific conventions are described in the accompanying text.

Method	Form	Granularity	Source	Updatable	Retrieval	Edge Semantics	Scope	Graph
ReAct (Yao et al., 2023b)	none	action	none	none	none	none	general	no
Reflexion (Shinn et al., 2023)	text	reflection	trajectory	yes	prompt buffer	none	task	no
MemoryBank (Zhong et al., 2024)	text	summary	trajectory	yes	embedding	none	task	no
RAP (Kagaya et al., 2024)	text	trajectory	trajectory	yes	embedding	none	task	no
Toolformer (Schick et al., 2023)	weights	API call	trajectory	no	none	none	catalog	no
CodeAct (Wang et al., 2024a)	code	action	manual	no	none	none	general	no
ToolLLM (Qin et al., 2024)	document	API	manual	no	embedding	none	catalog	no
Gorilla (Patil et al., 2024)	document	API	manual	no	embedding	none	catalog	no
AnyTool (Du et al., 2024)	tree	API	manual	no	structural	taxonomy	catalog	partial
ToolGen (Wang et al., 2025a)	tokens	API	trajectory	no	none	none	catalog	no
Voyager (Wang et al., 2023a)	code	skill	trajectory	yes	embedding	none	task	no
TroVE (Wang et al., 2024b)	code	skill	trajectory	yes	prompt / import	none	task	no
AWM (Wang et al., 2025b)	text + code	workflow	trajectory	yes	prompt memory	sequence	task	no
ExpeL (Zhao et al., 2024)	text	insight	trajectory	yes	embedding	none	task	no
AutoGuide (Fu et al., 2024)	text	rule	trajectory	yes	LLM selection	implicit cond.	task	no
MemP (Fang et al., 2025)	text + trajectories	procedure	trajectory	yes	embedding	none	task	no
EvolveR (Wu et al., 2025)	text	principle	trajectory	yes	embedding	none	task	no
KnowAgent (Zhu et al., 2025)	text	action	hybrid	no	static prompt	action transition	task	partial
FlowBench (Xiao et al., 2024)	flowchart	workflow	manual	no	prompt	branches	task	yes
AFlow (Zhang et al., 2025)	code	operator	trajectory	yes	execution	control/ data flow	task	yes
ToolNet (Liu et al., 2024a)	graph	tool	trajectory	yes	structural	co-occurrence	catalog	yes
ControlLLM (Liu et al., 2024b)	graph	tool	manual	no	structural	parameter dep.	catalog	yes
Graph RAG-Tool (Lumer et al., 2025)	graph	tool	hybrid	no	hybrid	dependency	catalog	yes
SkillGraph (Nie et al., 2026)	graph + text	agent/skill	hybrid	yes	embedding	communication	multi-agent	yes
Procedural Graph (ours)	**graph**	**action**	**hybrid**	**yes**	**hybrid**	**transition attributes**	**task**	**yes**

Table 5: **Dataset statistics and splits.**

Dataset	Domain	Train	Test	Notes
HotpotQA (Yang et al., 2018)	Open-domain QA	1,000	1,000	Disjoint samples from the official validation pool, de-duplicated by question id.
MultiChallenge (Deshpande et al., 2025)	Multi-turn dialogue	100	166	The main table evaluates all 166 test items; the construction study (Table 2) uses the 56-sample fast split.
GDPval (Patwardhan et al., 2025)	Professional deliverables	88	44	All tasks split deterministically by occupation.
ALFWorld (Shridhar et al., 2021)	Embodied household	238	134	Test uses the standard *unseen* split; loaded counts reflect internal filtering by the ALFWorld library.
τ-bench (Yao et al., 2024)	Customer service	500	115	Test uses the *retail* domain.
BFCL v3 (Patil et al., 2025)	Function calling	100	100	Base-category multi-turn tasks.
EnterpriseArena (Han et al., 2026)	Financial simulation	50	50	Simulator episodes under disjoint random-seed sets. The self-evolution study (Table 11) uses a smaller $20/20/20$ configuration.

lence verdict; the construction study (Section 5.3) additionally reports strict string Exact Match and word-level F1. For **MultiChallenge**, an LLM judge based on Gemini 3.1 Pro scores the Overall Success Rate together with four axes: Inference Memory, Instruction Retention, Reliable Versioned Editing, and Self Coherence. For **GDPval**, deliverables are scored against per-task rubrics and we report the mean rubric score. For **ALFWorld**, we report the task success rate on the test games. For τ**-bench**, we report Pass@1, i.e., the fraction of episodes whose final database state matches the annotated goal state. For **BFCL v3**, we report the official multi-turn accuracy. For **EnterpriseArena**, we report the full-horizon survival rate, the average lifespan in months, the mean time-averaged enterprise score, and the average capital raised (see Appendix C.1 for definitions).

B.3 Baseline Implementation Details

All baselines share the same ReAct solver, tool interface, and decoding configuration as our method, and every learning-based baseline consumes exactly the same training split that our self-evolution loop uses; they differ only in the artifact distilled from those trajectories and in how that artifact is injected at inference time. Table 6 summarizes the compared mechanisms: what each method stores, how that knowledge is organized, and how it enters the solver's context at inference time.

Table 6: **Memory mechanisms compared in the main experiments.**

Method	Stored Artifact	Organization	Used at Inference
Vanilla ReAct (Yao et al., 2023b)	none	–	–
MemoryBank (Zhong et al., 2024)	experience summaries	unstructured pool with forgetting	retrieved and prepended
RAP (Kagaya et al., 2024)	raw trajectories	similarity index	top-k in-context exemplars
ExpeL (Zhao et al., 2024)	natural-language insights	unordered list	injected into system prompt
AutoGuide (Fu et al., 2024)	"in state X, do Y" guidelines	condition-indexed pool	state-matched retrieval
AWM (Wang et al., 2025b)	workflows with text and actions	sequence library	workflow memory in prompt
KnowAgent (Zhu et al., 2025)	action rules and transitions	single text document	static prompt prefix
Procedural Graph (Ours)	**(procedure, relation, procedure) triplets**	**connected graph**	**localized subgraph → generated guidance**

MemoryBank (Zhong et al., 2024) maintains a long-term store of per-task experience summaries updated after each completed task; at inference, relevant summaries are retrieved with recency-weighted relevance and prepended to the solver prompt.

RAP (Kagaya et al., 2024) embeds completed trajectories and, at each new task, retrieves the most similar past trajectories as in-context exemplars.

ExpeL (Zhao et al., 2024) contrasts success and failure trajectories to distill a pool of natural-language insights, which are injected into the system prompt alongside retrieved successful exemplars.

AutoGuide (Fu et al., 2024) extracts state-conditioned guidelines from contrastive trajectory pairs; at each step, the current state is summarized and the applicable guidelines are retrieved and injected.

AWM (Wang et al., 2025b) induces reusable workflows from successful trajectories, retaining natural-language descriptions and action sequences as workflow memory in the solver prompt.

KnowAgent (Zhu et al., 2025) maintains a textual action-knowledge base describing available actions and admissible transition rules, injected as a static prompt prefix.

B.4 Procedural Graph Statistics

Table 7 summarizes the size of the Procedural Graph used for each benchmark in the main experiments. The graphs are compact: outside of BFCL v3, whose 131 nodes mirror its large function catalog, every graph has between 7 and 17 nodes and between 7 and 27 triplets. Across all graphs, the relation vocabulary $\mathcal{R}$ comprises four types: `LEADS_TO`, `TRIGGERS`, `PROVIDES_INPUT_FOR`, and `CONVERGES_TO`. Most edges carry the full *condition*/*guidance*/*pitfalls* attribute triple of Section 3.1, with *guidance* the most consistently populated field.

Table 7: **Sizes of the Procedural Graphs used in the main experiments.**

	HotpotQA	MultiChallenge	GDPval	ALFWorld	τ-bench	BFCL v3	EnterpriseArena
Nodes	9	7	15	11	17	131	11
Triplets	9	7	22	27	18	265	13

B.5 Prompt Templates

We use three families of prompts: a solver execution prompt that governs the ReAct loop, guidance generation prompts that translate the (sub)graph into situational guidance at each step, and a refiner prompt that drives offline self-evolution. The same templates are shared across all benchmarks; only the tool lists and task descriptions vary. Curly braces denote runtime placeholders.

Solver Execution Prompt (ReAct loop)

`{system_prompt}`

Procedural Graph Guidance: `{procedural_graph_guidance}`

You must interleave Thought and Action. Your output format must be exactly:

```
Thought:  <your reasoning about what to do next>
Action:  <tool_name>(arg1=val1, arg2=val2, ...)
```

Example:

```
Thought:  I need to check the files in the workspace directory to
locate the source documents.
Action:  list_dir(path=".")
```

DO NOT write any "Observation:" block or any subsequent steps. Only output exactly one Thought and one Action block. Do NOT simulate the environment's responses.

Current Trajectory: `{trajectory}`

```
Thought:
```

Guidance Generation Prompt (Local subgraph; default)

You are an expert cognitive architect and execution guide for an AI agent solving the task: `{task_description}`

Here is `{graph_context_desc}`**:** `{subgraph_summary}`
Here is the current active query / observation: `{query}`
Here is the agent's recent execution trajectory: `{recent_context}`

Analyze this `{graph_source}` in the context of the agent's current progress. Using the `condition`, `guidance`, and `pitfalls` attributes carried by the edges in the graph context, generate clear, detailed, and actionable guidance advising the agent on exactly what step or strategy to pursue next, what pitfalls to avoid, and how to recover from recent failures if any. You must include any specific command patterns, file paths, tools, or arguments defined in the graph context if they are relevant to the next steps.

Serialized Graph Context Example. We reconstruct the graph-context text below from the saved HotpotQA Mode 2 graph using the implemented local serializer. The active-node header supplies the node ID, type, and description. Directed transitions are grouped by hop and followed by their condition, guidance, and pitfalls, preserving the checkpoint's field text. The two stored relation labels, `LEADS_TO` and `PROVIDES_INPUT_FOR`, are not printed by this serializer.

Serialized Local Graph Context (HotpotQA; excerpt)

```
Active Cognitive Node: [First_Hop_Retrieve] (Type: ACTION)
Description: Execute first_hop_retrieve to fetch primary evidence
passages.
Immediate Transition Options (Hop 1):
- Transition: [First_Hop_Retrieve] → [Scan_Index] (Condition:
first_hop_retrieve)
* Guidance: Review the retrieved primary passages via Scan_Index to
locate specific bridge terms (such as birth dates, locations, or
associated entities).
* Pitfalls to Avoid: Do not skip reading evidence details; missing
the exact bridge entity name causes second-hop search failure.
Subsequent Horizon (Hop 2):
- Transition: [Scan_Index] → [Bridge_Extract] (Condition:
scan_index)
* Guidance: Extract the explicit connecting entity or bridge term
linking the first passage to the target question.
* Pitfalls to Avoid: Ensure the extracted bridge term matches exact
Wikipedia capitalization conventions.
```

Guidance Generation Prompt (Full graph variant)

The full-graph variant is *textually identical* to the prompt above; the only difference is the content bound to the graph context slot, which is the complete Procedural Graph rather than the localized subgraph. Concretely, `{graph_context_desc}` is instantiated as "the complete Procedural Graph governing the task structure and strategic guidance", `{subgraph_summary}` as the full `{graph_summary}`, and `{graph_source}` as "complete Procedural Graph". The task description, query, recent-trajectory window, analysis instruction, and output requirements are unchanged, so the two ablation rows in Table 3 differ only in graph scope and not in prompt wording or requested output length.

Refiner Prompt (Self-evolution)

You are an expert cognitive architect optimizing a Procedural Graph for an intelligent agent. The Procedural Graph encodes structured procedural guidance.

Task context: `{task_description}`
Refinement mode: `{mode}`
Available Tool Actions (the agent can only execute these actions): `{available_tools_list}`
Recent execution trajectories: `{attempts_block}`
Current Procedural Graph representation: `{current_graph_json}`
Previously rejected candidates: `{rejected_block}`

Your job is to refine the Procedural Graph. Follow these guidelines based on the mode:

- `static_onetime / static_incremental`: Prune edges/nodes that lead to loops, deadlocks, or failures. Add missing nodes and edges that could fix the failures and improve performance for future tasks.
- `scratch_onetime / scratch_incremental`: If starting from scratch (the graph contains only Start → End), synthesize a brand new, complete Procedural Graph using the Available Tool Actions list, Status, and successful patterns in the trajectories. Otherwise, prune edges/nodes that lead to loops, deadlocks, or failures, and add missing nodes and edges based on the given graph.

Rules for nodes and edges. Rules 2–4 describe the edge attributes in $\Phi(e)$: `condition`, `guidance`, and `pitfalls`. The remaining rules govern node compatibility, generality, and graph structure.

1. **Action Nodes.** Any node of type `ACTION` must match one of the action/tool names in the "Available Tool Actions" list above.
2. **Transition Conditions.** If an edge has a `condition`, provide a natural-language semantic precondition under which this transition should fire (e.g., "When dialogue history has been parsed but target constraints are unknown"). Use `null` if the transition is unconditional.
3. **Execution Guidance.** For every edge added in `add_edges`, you MUST provide a `guidance` string detailing exactly what action to take next and the strategic rationale behind it.

Refiner Prompt (Self-evolution) (continued)

4. **Pitfalls.** Provide a `pitfalls` string warning about premature actions, forbidden words, or common formatting pitfalls to avoid during this step.
5. **Generality & Leak Prevention.** The updated Procedural Graph must guide the agent effectively without overfitting to specific details of a single trajectory. Use high-level conceptual descriptions.
6. **Node ID Compatibility.** If refining an existing graph (static modes), you MUST preserve the existing node IDs (such as `Month_Start`, `Decide_Capital`, and the tool names) so they remain compatible with the environment's state tracker. Do not rename them.
7. **Graph Structure.** Follow the task's configured cycle policy. Every edge must reference existing nodes, and every node must have a directed path to a terminal node. The environment loop handles repetition across simulation cycles.

Please propose the exact set of edits to perform. You must output your edits as a single valid JSON block containing four arrays: `add_nodes`, `delete_nodes`, `add_edges`, and `delete_edges`. Output format must be exactly:

```
{ "add_nodes":   [{"id":..., "type":  "ACTION", "description":...}],
  "delete_nodes":  ["node_id"],
  "add_edges":   [{"source":..., "target":..., "relation":...,
                  "condition":..., "guidance":..., "pitfalls":...}],
  "delete_edges":  [{"source":..., "target":...}] }
```

Make sure to output ONLY the raw JSON block.

Each entry in `delete_edges` removes all edges with the specified source and target, regardless of relation. To retain selected transitions between the same endpoints, include them in `add_edges`, which is applied after deletion.

B.6 Self-Evolution Algorithm

Algorithm 1 makes the retained-checkpoint state explicit. S_k is the cached validation score of $\mathcal{G}_k$, and c specifies whether cycles are allowed. $\mathrm{Tail}_{L_{\max}}$ preserves the ending of its input by removing excess tokens from the beginning, leaving shorter inputs unchanged. The graph stays fixed during each training or validation episode. SerializeRejections supplies prior candidate graphs and their validation scores, or structural-failure diagnostics, to the refiner; the associated training traces remain part of the rejection record.

Algorithm 1 Offline Closed-Loop Procedural Graph Self-Evolution

Require: Initial graph $\mathcal{G}_0$; training/validation sets $\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{val}}$; round budget K; trajectory limit $L_{\max}$; cycle policy c.

$S_0 \leftarrow \mathrm{Evaluate}(\mathcal{G}_0, \mathcal{D}_{\text{val}})$
$\mathcal{H}_{\text{rejected}} \leftarrow [\,]$
for $k = 1, \ldots, K$ **do**
  $\mathcal{G}_k \leftarrow \mathcal{G}_{k-1}$; $S_k \leftarrow S_{k-1}$ {Retain unless accepted}
  Select training batch $\mathcal{B}_k \subset \mathcal{D}_{\text{train}}$
  $\mathcal{E}_k \leftarrow \mathrm{Rollout}(\mathcal{G}_{k-1}, \mathcal{B}_k)$ {Training traces and scores}
  $\mathcal{C}_k \leftarrow \mathrm{Tail}_{L_{\max}}(\mathrm{ConcatTrajectories}(\mathcal{E}_k))$
  $\mathcal{R}_k \leftarrow \mathrm{SerializeRejections}(\mathcal{H}_{\text{rejected}})$
  $\Delta\mathcal{G}_k \leftarrow \mathrm{Refiner}(\mathcal{G}_{k-1}, \mathcal{C}_k, \{S_i^{(k)}\}_i, \mathcal{R}_k)$
  $(\mathcal{G}_k^{\text{cand}}, d_k) \leftarrow \mathrm{PrepareCandidate}(\mathcal{G}_{k-1}, \Delta\mathcal{G}_k, c)$
  if $d_k \neq \varnothing$ **then**
    Append $(\Delta\mathcal{G}_k, \mathcal{G}_k^{\text{cand}}, \mathcal{E}_k, d_k)$ to $\mathcal{H}_{\text{rejected}}$
    continue {No validation rollout; retained state is unchanged}
  end if
  $S_k^{\text{cand}} \leftarrow \mathrm{Evaluate}(\mathcal{G}_k^{\text{cand}}, \mathcal{D}_{\text{val}})$
  if $S_k^{\text{cand}} \geq S_{k-1}$ **then**
    $\mathcal{G}_k \leftarrow \mathcal{G}_k^{\text{cand}}$; $S_k \leftarrow S_k^{\text{cand}}$ {Accept, including ties}
  else
    Append $(\Delta\mathcal{G}_k, \mathcal{G}_k^{\text{cand}}, \mathcal{E}_k, S_k^{\text{cand}})$ to $\mathcal{H}_{\text{rejected}}$
  end if
end for
return $\mathcal{G}_K$

Candidate Preparation and Structural Checks. $\mathrm{PrepareCandidate}$ applies edits to a copy of the retained graph, deleting edges and nodes before adding nodes and edges. It reports malformed edits, invalid node or relation types, and missing edge endpoints as failures. When cycles are disallowed, the implementation removes detected cycle-closing edges before validation; when cycles are allowed, that repair and the acyclicity check are skipped. The remaining checks require valid edge endpoints and a directed path from every node to a terminal node, defined by zero out-degree. This is a reachability check to a terminal node, not specifically to the node named `End`. Matching action-node names to the available tool list is a refiner-prompt requirement; the generic structural validator does not independently enforce tool-catalog membership. On failure, d_k contains diagnostics and $\mathcal{G}_k^{\mathrm{cand}}$ may be unavailable; on success, $d_k = \varnothing$.

C Long-Horizon Analysis on EnterpriseArena

This appendix describes the EnterpriseArena simulator and evaluation metrics, reports survival and cash trajectories, and compares traces from the baseline, memory-summarization, and PG-guided agents.

C.1 EnterpriseArena Mechanics and Crisis Schedule

The Chief Financial Officer (CFO) simulator models the balance sheet dynamics of a microfinance lending institution over a long-term horizon of up to 132 months. The state of the environment at month t is formalized as a multi-dimensional tuple:

$$\mathcal{S}_t = \big(C_t,\, L_t,\, A_t,\, IR_t,\, PR_t,\, AP_t,\, D_t,\, E_t,\, U_t\big), \tag{7}$$

where C_t is the cash balance, L_t is the gross loan portfolio, A_t is the allowance for loan losses, IR_t and PR_t are interest and principal receivables, AP_t is accounts payable, D_t is total outstanding debt, E_t is total equity raised, and U_t is the active user base.

The agent interacts with the simulator through a discrete action space. The primary state-advancing action is `book_closing()`, which simulates the transition from month t to $t+1$. During this transition, the environment executes the following operations:

1. **Loan Amortization**: A fraction of the loan portfolio L_t matures, generating principal payments and interest income based on the lending rate.
2. **User and Operational Costs**: The active user base U_t grows or decays organically. Fixed operational costs and user acquisition costs are deducted from the cash balance C_t.
3. **Write-Offs**: Defaulted loans are written off against the allowance A_t, and new provisions are calculated.

To manage liquidity, the agent can invoke `fund_raising_request(type, amount)`, where the type is either `'equity'` or `'debt'`. Fundraising is subject to two realistic constraints:

- **Market Delivery Lag**: Capital is not delivered immediately. There is a stochastic delay of 1 to 6 months between the request step and the cash injection.
- **Market Capacity Cap**: The maximum amount of capital that can be raised in a single request is dynamically capped by the environment based on the current macroeconomic state and the institution's financial health.

The simulation terminates immediately if the cash balance goes negative ($C_t < 0$), representing corporate bankruptcy. The environment simulates three successive macroeconomic crises to test the agent's long-term resilience:

- **Crisis 1 (Month 32)**: A mild contraction where loan repayment rates drop slightly from 98% to 90%.
- **Crisis 2 (Month 59)**: A severe economic recession. Repayment rates plunge to 60%, write-offs surge, and organic user growth turns negative.
- **Crisis 3 (Month 112)**: A systemic liquidity freeze. Repayment rates drop to 40%, and the market capacity cap for fundraising is severely restricted, making new capital acquisition extremely difficult.

Evaluation Metrics. **Full Surv.** is the percentage of runs that complete the 132-month horizon without bankruptcy; the crisis columns report the fractions reaching months 32, 59, and 112. **Avg. Months** averages run duration, including early terminations. **Tools/Mo** averages the per-run ratio of arena information-tool calls to simulated months, excluding memory operations and state-changing actions. **Raised** is the mean cumulative equity and debt financing actually received per run, in millions of dollars.

We use EnterpriseArena's revenue-based valuation and tool-use penalty (Han et al., 2026) to compute the score at each recorded month:

$$s_{i,t} = \begin{cases} 0, & \text{if run } i \text{ has gone bankrupt,} \\ 5\,\mathrm{Rev}_{i,t}^{(12)} + C_{i,t} - 5{,}000\,N_{i,t}, & \text{otherwise,} \end{cases} \tag{8}$$

where $\mathrm{Rev}_{i,t}^{(12)}$ is trailing-twelve-month revenue (annualized from the available monthly average when fewer than twelve months are recorded), $C_{i,t}$ is cash in dollars, and $N_{i,t}$ is the cumulative number of arena information-tool calls. **Avg. Score** first averages these monthly scores within each run up to termination, then averages across runs, reporting the result in millions of dollars. Thus, a run ending in bankruptcy can still have a positive time-averaged score.

C.2 Results and Survival Analysis

Table 8: **Performance comparison on EnterpriseArena.** Within each model block, higher survival, lifespan, and score and lower Tools/Mo are highlighted. Raised highlights the largest financing volume. Metric definitions are given in Appendix C.1.

Model & Config	Overall			Multi-Crisis Survival			Agent Performance	
	Full Surv.	Avg. Months	Avg. Score	1st Crisis	2nd Crisis	3rd Crisis	Tools/Mo	Raised
Claude Sonnet 4.6								
Baseline	44.0%	89.80	**$78.86M**	**100.0%**	78.0%	52.0%	0.13	$152.12M
RAP (Kagaya et al., 2024)	50.0%	93.24	$76.76M	**100.0%**	70.0%	54.0%	**0.12**	**$157.54M**
MemoryBank (Zhong et al., 2024)	56.0%	97.82	$56.57M	**100.0%**	78.0%	**62.0%**	0.27	$97.01M
Procedural Graph (Ours)	**58.0%**	**98.58**	$70.38M	**100.0%**	**80.0%**	60.0%	0.36	$130.44M
Gemini 3.1 Pro								
Baseline	6.0%	50.28	$31.85M	**100.0%**	38.0%	8.0%	0.89	$21.82M
RAP (Kagaya et al., 2024)	6.0%	51.34	$32.27M	**100.0%**	38.0%	12.0%	0.83	$23.16M
MemoryBank (Zhong et al., 2024)	22.0%	59.78	$33.11M	**100.0%**	38.0%	24.0%	**0.39**	$14.53M
Procedural Graph (Ours)	**34.0%**	**79.22**	**$37.21M**	**100.0%**	**54.0%**	**46.0%**	3.18	**$38.20M**
Gemini 3.5 Flash								
Baseline	0.0%	33.58	$28.59M	**100.0%**	0.0%	0.0%	18.94	$0.00M
RAP (Kagaya et al., 2024)	0.0%	34.04	$28.75M	**100.0%**	2.0%	0.0%	17.71	$0.69M
MemoryBank (Zhong et al., 2024)	0.0%	35.60	**$29.34M**	**100.0%**	10.0%	0.0%	**11.99**	$1.69M
Procedural Graph (Ours)	0.0%	**40.62**	$29.08M	**100.0%**	**14.0%**	0.0%	12.53	**$9.39M**
Grok 4.1 Fast								
Baseline	26.0%	63.76	$39.42M	**100.0%**	42.0%	28.0%	0.47	$27.24M
RAP (Kagaya et al., 2024)	28.0%	64.08	$35.47M	**100.0%**	40.0%	28.0%	0.41	$21.72M
MemoryBank (Zhong et al., 2024)	16.0%	58.24	$31.95M	**100.0%**	42.0%	20.0%	1.03	$11.65M
Procedural Graph (Ours)	**40.0%**	**75.14**	**$39.62M**	**100.0%**	**50.0%**	**40.0%**	**0.40**	**$30.11M**

Figure 3 compares cash trajectories and Kaplan–Meier survival curves for four planning configurations on Claude Sonnet 4.6, Gemini 3.1 Pro, Gemini 3.5 Flash, and Grok 4.1 Fast. Each model panel places survival curves above cash trajectories, which show individual sample paths, the mean, and the 95% confidence interval. Survival gains vary across the four solvers.

The unguided baseline reaches full-horizon survival rates of 44.0% for Claude Sonnet 4.6, 6.0% for Gemini 3.1 Pro, 26.0% for Grok 4.1 Fast, and 0.0% for Gemini 3.5 Flash (Table 8). In the Flash baseline, mean lifespan is 33.58 months and no episode reaches the second crisis.

The Procedural Graph configuration (solid blue line) improves full-horizon survival for three solvers and mean lifespan for all four:

- **Claude Sonnet 4.6**: PG achieves the highest survival rate of 58.0% (a 14.0-point gain over the baseline). Its mean cash trajectory remains stable and ends at $76.03M, well above the liquidity warning threshold of $5.0M.

- **Gemini 3.1 Pro**: The Procedural Graph provides the largest absolute survival gain, raising the survival rate from 6.0% (baseline) to 34.0%, and securing a final mean cash of $22.99M, approximately 10.5 times the baseline's $2.18M.

- **Grok 4.1 Fast**: Survival improves from 26.0% to 40.0%, with the final mean cash increasing from $23.62M to $30.20M.

- **Gemini 3.5 Flash**: All four configurations have 0.0% full-horizon survival in this evaluation, while PG increases mean lifespan from 33.58 to 40.62 months. This result describes the configurations evaluated here; the separate self-evolution experiment is reported in Section 5.4.

MemoryBank achieves higher full-horizon survival than RAP on Gemini 3.1 Pro (22.0% vs. 6.0%) and Claude Sonnet 4.6 (56.0% vs. 50.0%). PG exceeds both on these two solvers and on Grok 4.1 Fast, while all methods tie at 0.0% on Gemini 3.5 Flash.

Relative to the unguided baseline, PG raises full-horizon survival by 28 points on Gemini 3.1 Pro and 14 points on both Claude Sonnet 4.6 and Grok 4.1 Fast. Appendix C.3 examines the actions taken in one matched task instance.

C.3 Case Study: Three Agents Facing the First Crisis

To understand the mechanisms driving the quantitative performance gap, we contrast the step-by-step reasoning traces of a baseline agent, a memory summarization agent, and a Procedural Graph guided agent. The following traces are extracted directly from real experimental logs of `Grok 4.1 Fast` operating on an identical task instance (Seed 14) during the critical transition into the first macroeconomic crisis (Months 29 to 33). Traces are condensed from the raw logs; bracketed italic text is editorial annotation, not model output.

In this baseline trace, the agent reports no bankruptcy risk despite declining cash reserves. It later submits requests that violate the environment's constraints:

Baseline Agent Trace (Sample 14) – Myopia & Rule Violation

Step 45 (Month 31):
Thought: Month 31 confirms trajectory: cash $14.2M (down ~$7.4M but **infinite runway >12mos**) ... **no bankruptcy risk**. With 1 action left, `book_closing` to advance month, lock YTD financials, sustain momentum, and maximize valuation...
Action: `book_closing()`

Step 46 (Month 32):
Thought: Month 32: cash $6.2M (down sharply ~$8M, runway ~2-3mos, bankruptcy risk emerging)... Prior `book_closing` advanced month but cash burn accelerating; cannot risk another `book_closing` or pass. Raise equity to bolster cash...
Action: `fund_raising_request(type='equity', amount=20000000)`

Step 47 (Month 33):
Thought: Month 33: cash critically low at $683K (runway <1mo, imminent bankruptcy risk)... Equity raise ($20M requested, pending 1-6mos) submitted but not received; cannot pass or `book_closing`. Submit debt raise for immediate bridge funding to avoid bankruptcy...
Action: `fund_raising_request(type='debt', amount=10000000)`
Observation: Your fundraising request was rejected: you already have a pending fundraising request.

[Observed behavior: The agent requests debt financing while an equity request is still pending. The environment rejects the additional request under the single-pending-request constraint.]

The memory summarization agent uses an LLM-generated running summary of past events. In this example, it recognizes the shrinking cash runway but submits fundraising amounts inconsistent with its stated intentions and repeats requests while earlier funding is still pending:

Memory Summarization Agent (Sample 14) – Hallucination & Constraint Blindness

Step 51 (Month 29):
Thought: Cash $25.3M (MoM -$5.4M, runway ∼4mo per updated forecast)... pending equity ($18M, submitted M26, now ∼3mo in)... Runway critically eroding (4mo < 6mo buffer); pending raise may arrive soon but unreliable. Check market for raise viability...
Action: `check_market_data(data_type='VIX')`

Step 53 (Month 30):
Thought: Cash $25.3M (runway 4mo)... VIX 10.51 very low... Submit equity raise now per rules (prefer equity $15-20M)...
Action: `fund_raising_request(type='equity', amount=`**18**`)`
Observation: Your fundraising request was rejected: you already have a pending fundraising request.

Step 57 (Month 32):
Thought: Cash $11.8M (runway 2mo)... low-risk equity raise $18M fits rules... submit to extend runway...
Action: `fund_raising_request(type='equity', amount=`**18**`)`

Step 59 (Month 32):
Thought: Cash $4.3M... runway 1mo critical... must raise now despite pending... submit $20M equity to buffer urgency...
Action: `fund_raising_request(type='equity', amount=`**20**`)`
Observation: Your fundraising request was rejected: you already have a pending fundraising request.

[Observed behavior: The agent submits amounts of $18 and $20 despite referring to $18M and $20M in its reasoning, and repeatedly submits requests while earlier funding is pending.]

Guided by the Procedural Graph, the proposed agent maintains a clear structural prior of the task's temporal rules. It initiates fundraising with the correct scale well in advance, chaining sequential reasoning steps while remaining strictly compliant with environment constraints during the stochastic delivery lag:

Procedural Graph Agent Trace (Sample 14) – Strategic Foresight & Adherence

[At the opening of Month 30, an equity request submitted in Month 27 delivers: $28.8M after the market-capacity cap.]

Step 37 (Month 30):
Thought: Equity raise succeeded (capped at ∼$28.8M, added to cash for $47.6M total)... VIX at 10.51 is exceptionally low (indicating calm markets, ideal for equity raise to minimize dilution cost)... **with escalating burn ahead of the projected contraction, raise equity again now for ∼$50M**, preserving no-debt status...
Action: `fund_raising_request(type='equity', amount=50000000)`

Step 38 (Month 30):
Thought: Request submitted (pending, 1–6 month delivery)... run forecast to confirm new runway...
Action: `cash_flow_forecast_calculation(months=12)`

Step 42 (Month 32):
Thought: ... forecast confirms 6-mo runway... pending $50M equity (submitted M30) likely delivers soon... **advance month via `book_closing` to progress raise delivery**...
Action: `book_closing()`

Step 44 (Month 33):
Thought: ... pending $50M equity (submitted M30, now 3 mos in) likely delivers soon... no bankruptcy imminent... **advance via `book_closing` to progress raise delivery**...
Action: `book_closing()`

*[Result: The agent **strictly adheres to the delivery-lag and single-pending constraints, successfully bridges the crisis months, and survives** to the end of the simulation at Month 132.]*

In this task instance, the PG-guided agent tracks pending funding, checks projected runway, and advances the month while awaiting delivery. The baseline and memory-summary traces show invalid requests. These observations illustrate different behaviors on the same instance; they do not establish how frequently each behavior occurs across runs.

D PROCEDURAL GRAPH CONSTRUCTION: SUPPLEMENTARY RESULTS

This appendix provides quantitative results, diagnostic case studies of construction failure modes, formulations of the five PG construction strategies, and the detailed experimental configurations.

D.1 EFFICIENCY AND ROBUSTNESS STATISTICS

Tables 9 (HotpotQA) and 10 (MultiChallenge) compare task performance, resource use, and parsing failures across the five construction modes and the unguided baseline.

Table 9: **HotpotQA test set performance, resource, and robustness statistics.** Highlighted cells mark column optima across all configurations, including the unguided baseline; ties are highlighted equally.

Construction Mode	Ans F1 (↑)	Tokens (↓)	Steps (↓)	Parsing Fail. (↓)	Latency (s, ↓)
Unguided Baseline (w/o PG)	71.21	**4,003.24**	4.88	0.016	**18.06**
Mode 1: Hand-crafted Expert	76.61	9,045.69	4.12	0.007	39.34
Mode 2: Expert + Static Update	77.16	8,942.79	4.11	**0.005**	37.73
Mode 3: Expert + Self-Evolution	76.34	10,658.36	4.11	0.009	39.61
Mode 4: Scratch + Static Build	69.49	6,396.90	4.30	0.016	36.98
Mode 5: Scratch + Self-Evolution	**78.79**	10,115.89	**3.97**	0.016	31.53

Table 10: **MultiChallenge test set performance, resource, and robustness statistics.** Bold blue cells mark column optima across all configurations, including the unguided baseline; ties are highlighted equally. Pale blue marks Mode 5's second-best values where Mode 2 ranks first.

Construction Mode	Overall Success (↑)	Tokens (↓)	Steps (↓)	Parsing Fail. (↓)	Latency (s, ↓)
Unguided Baseline (w/o PG)	87.50	7,403.98	7.05	1.05	127.59
Mode 1: Hand-crafted Expert	58.93	11,039.82	6.60	0.88	117.70
Mode 2: Expert + Static Update	53.57	**5,990.79**	**4.32**	**0.36**	**65.93**
Mode 3: Expert + Self-Evolution	**92.86**	12,157.52	6.73	0.98	128.50
Mode 4: Scratch + Static Build	89.29	14,859.80	8.02	1.07	166.57
Mode 5: Scratch + Self-Evolution	91.07	7,984.50	5.05	0.57	92.81

Solver Steps and Latency. On MultiChallenge, Mode 5 uses 5.05 steps and 92.81 seconds per sample, compared with 8.02 steps and 166.57 seconds for Mode 4. Mode 2 has the lowest average step count (4.32) and latency (65.93 seconds), but also the lowest success rate (53.57%). Mode 3 achieves the highest success rate (92.86%), with 6.73 steps and 128.50 seconds per sample. These results show why resource use should be read alongside task performance; aggregate statistics alone do not identify the graph edits responsible for the differences.

Compared with the hand-crafted Mode 1 graph (6.60 steps and 117.70 seconds), Mode 5 uses fewer steps and less time on MultiChallenge. On HotpotQA, Mode 5 has the lowest latency among the PG configurations (31.53 seconds).

Action Formatting and Parsing Robustness. On HotpotQA, expert-initialized Modes 1–3 have fewer parsing failures per sample than scratch-initialized Modes 4–5. Mode 2 has the lowest value (0.005), followed by Mode 1 (0.007) and Mode 3 (0.009); Modes 4 and 5 each have 0.016. On MultiChallenge, Mode 5 records 0.57 parsing failures per sample compared with Mode 4's 1.07. The association between initialization and formatting therefore differs across tasks, and does not by itself establish that a particular graph schema causes the reduction.

Token Use. On MultiChallenge, Modes 1 and 3 use approximately 11.0k and 12.2k tokens per sample, respectively. Mode 2 uses approximately 6.0k, alongside its lower success rate. Mode 5 uses 7,984.50 tokens, compared with 14,859.80 for Mode 4 and 7,403.98 for the unguided baseline. The Mode 4–5 comparison reverses on HotpotQA: Mode 5 uses 10,115.89 tokens versus Mode 4's 6,396.90. Total token use reflects the complete interaction, including the number of steps and generated outputs, and cannot alone establish that a graph representation is more compact.

D.2 Graph Construction Modes

We provide the formulation and implementation details for each of the five PG construction modes. Here, Self-Evolution refers to incremental graph updates between training batches; the graph remains fixed within each episode and during test evaluation.

Mode 1: Hand-crafted Expert PG (Zero-shot Baseline)

Initialization: A human-engineered directed graph $G_{expert} = (V, E)$ containing expert-designed tool nodes, valid transitions, and manually annotated natural language guidance.
Training Strategy: None. Zero-shot execution on the test set.
Evolutionary Mutation: None. The graph structure and guidance remain static.
Validation & Safeguard: None.

Mode 2: Expert PG + Static One-time Update (Offline)

Refinement Mode: `static_onetime`.
Initialization: Initialized with the hand-crafted expert graph G_{expert} from Mode 1.
Training Strategy: Static offline execution. The agent runs on the entire training split in a single pass to collect all successful and failed trajectories.
Evolutionary Mutation: Single-pass. The LLM refiner ingests all trajectories in a single large-context window to perform a one-time global offline update to transitions and guidance.
Validation & Safeguard: None. The refined graph is directly committed.

Mode 3: Expert PG + Self-Evolution (Incremental)

Refinement Mode: `static_incremental`.
Initialization: Initialized with the hand-crafted expert graph G_{expert} from Mode 1.
Training Strategy: Online incremental batches. The training split is partitioned into sequential strides ($S = 100$ samples for HotpotQA; $S = 20$ samples for MultiChallenge).
Evolutionary Mutation: After each stride, the refiner uses the latest failure logs to update nodes, transitions, and local guidance.
Validation & Safeguard: The candidate is evaluated on the validation split. If performance declines, the previous best graph is restored.

Mode 4: Scratch + Static One-time Build (Offline)

Refinement Mode: `scratch_onetime`.
Initialization: Minimal skeleton graph $G_{skeleton} = (\text{Start} \rightarrow \text{End})$ with no intermediate nodes or additional transitions.
Training Strategy: Static offline execution. The agent runs on the entire training split in a single pass to collect all successful and failed trajectories.
Evolutionary Mutation: Single-pass. The LLM refiner ingests all trajectories in a single large-context window to perform a one-time global offline update to transitions and guidance.
Validation & Safeguard: None. The refined graph is directly committed.

Mode 5: Scratch + Self-Evolution (Incremental)

Refinement Mode: `scratch_incremental`.
Initialization: Minimal skeleton graph $G_{skeleton} = (\text{Start} \rightarrow \text{End})$ with no intermediate nodes or additional transitions.
Training Strategy: Online incremental batches. The training split is partitioned into sequential strides ($S = 100$ samples for HotpotQA; $S = 20$ samples for MultiChallenge).
Evolutionary Mutation: After each stride, the refiner uses the latest failure logs to update nodes, transitions, and local guidance.
Validation & Safeguard: The candidate is evaluated on the validation split. If performance declines, the previous best graph is restored.

D.3 Experimental Setup

To ensure reproducibility, we detail the core experimental settings and dataset splits below:

Model Selection and Hyperparameters. All construction-strategy experiments in this appendix, including agent execution and evolutionary refinement, are powered by the frozen `Gemini 3.5 Flash` model snapshot. To ensure deterministic and reproducible tool-calling reasoning paths, the sampling temperature is set strictly to 0 and top-k sampling is configured with $k = 1$, enforcing greedy decoding across all model calls. Additionally, to prevent random execution deviations, the API safety filtering thresholds are unified and held constant to eliminate premature trajectory blocking, and the maximum generation length is locked at $2,048$ tokens for the agent and $8,192$ tokens for the evolutionary refiner to guard against truncated reasoning paths.

HotpotQA Environment Setup. The HotpotQA experiments are conducted on a standard multi-hop reasoning split consisting of $1,000$ training samples, $1,000$ validation samples, and $1,000$ test samples. The agent's performance is measured using strict string Exact Match (EM) and word-level overlap F1 score between the agent's generated answer and the ground truth.

MultiChallenge Environment Setup. The MultiChallenge dialogue experiments are conducted on a dataset split consisting of 100 training samples, 100 validation samples, and 56 test samples. Task performance is evaluated via an LLM judge based on Gemini 3.1 Pro, measuring the Overall Success Rate, alongside four core challenge subcategory axes: Inference Memory (IM), Instruction Retention (IR), Reliable Versioned Editing (RVE), and Self Coherence (SC).

E Round-by-Round Self-Evolution on EnterpriseArena

Table 11 presents the round-by-round metrics of the 10-generation CFO evolution. The delta (Δ) rows track metric changes relative to the active best validation checkpoint. Appendix E.1 examines tool use, and Appendix E.2 discusses candidate screening.

E.1 Computational Efficiency

Beyond task performance, the evolved graphs also help reduce redundant tool use. As detailed in Table 11, the average number of tool invocations per simulation month (`Tools/Mo`) on the validation split drops from 17.23 in the Baseline to 3.08 in Round 2, stabilizing at 3.13 by Round 8. The baseline agent, operating under zero-shot ReAct, repeatedly queries the environment for cash balances and market metrics within the same turn. The evolved PG guides the agent through these queries once per cycle. This is an 81.8% **reduction in tool invocations** per simulation month, showing that the evolved graph reduces environment interactions in this experiment. We report tool calls rather than tokens here because the simulation logs record invocation counts but not per-month token consumption.

E.2 Candidate Screening: Three Cases

The evolution loop combines validation checks with structural verification to filter unsuccessful candidates, as illustrated by three cases:

- **Performance Rollback (Rounds 3, 4, 6)**: In Round 3, the mutation engine attempted to lower the cash threshold that triggers fundraising. While this succeeded on specific training seeds, it led to premature dilution and cash shortages on validation seeds, dropping validation survival by 15.0 **points** and average lifespan by 10.45 **months**. The gate rejected the candidate and restored the Round 2 checkpoint.
- **Execution Constraint Rollback (Round 5)**: In Round 5, the candidate failed structural verification before simulation, so validation was skipped. The reported validation metrics for this round carry forward the Round 4 results.
- **Validation Safeguard (Round 10)**: In Round 10, training survival was 90.0%, but validation survival fell to 85.0% (-5.0 points) and the mean time-averaged enterprise score on the validation

Table 11: **PG self-evolution across ten rounds with Gemini 3.5 Flash.** The Δ rows report metric changes relative to the previous best validation checkpoint. Gray rows mark rounds without a committed graph update.

Evolution Round	Overall			Multi-Crisis Survival			Agent Performance		
	Full Surv.%	Avg. Months	Avg. Score	1st Crisis	2nd Crisis	3rd Crisis	Tools/Mo	Actions	Raised
Baseline									
Validate	0.0%	34.80	$28.377M	100.0%	0.0%	0.0%	17.23	35.7	$0.47M
Test	0.0%	33.30	$28.754M	100.0%	0.0%	0.0%	18.60	34.2	$0.00M
Round 01 [✓ Evolved]									
Train	0.0%	33.40	$28.777M	100.0%	0.0%	0.0%	16.25	34.4	$0.00M
Validate	45.0%	88.90	$38.594M	100.0%	60.0%	45.0%	7.92	89.9	$40.68M
Δ *vs. Base*	+45.0	+54.10	+$10.217M	–	+60.0	+45.0	-9.31	+54.2	+$40.21M
Test	**70.0%**	**104.15**	**$43.519M**	100.0%	**80.0%**	**70.0%**	**3.62**	105.2	**$44.90M**
Round 02 [✓ Evolved]									
Train	85.0%	117.75	$46.524M	100.0%	90.0%	85.0%	3.39	118.8	$57.27M
Validate	80.0%	112.60	$46.038M	100.0%	85.0%	80.0%	3.08	113.6	$52.33M
Δ *vs. R01*	+35.0	+23.70	+$7.444M	–	+25.0	+35.0	-4.84	+23.7	+$11.65M
Test	**80.0%**	**114.00**	**$44.992M**	100.0%	**90.0%**	**80.0%**	**3.11**	115.0	**$46.83M**
Round 03 [× Rolled Back]									
Train	80.0%	113.35	$46.857M	100.0%	85.0%	80.0%	3.08	114.3	$51.88M
Validate	65.0%	102.15	$43.965M	100.0%	75.0%	70.0%	3.30	103.2	$46.74M
Δ *vs. R02*	-15.0	-10.45	-$2.073M	–	-10.0	-10.0	+0.22	-10.4	-$5.59M
Round 04 [× Rolled Back]									
Train	80.0%	115.05	$46.631M	100.0%	90.0%	80.0%	3.06	116.0	$53.04M
Validate	75.0%	109.20	$49.831M	100.0%	85.0%	75.0%	3.05	110.2	$70.85M
Δ *vs. R02*	-5.0	-3.40	+$3.793M	–	–	-5.0	-0.03	-3.4	+$18.52M
Round 05									
Train	80.0%	114.05	$44.565M	100.0%	90.0%	80.0%	3.12	115.0	$52.85M
Validate (R04)	75.0%	109.20	$49.831M	100.0%	85.0%	75.0%	3.05	110.2	$70.85M
Round 06 [× Rolled Back]									
Train	80.0%	114.90	$46.345M	100.0%	90.0%	80.0%	3.09	115.9	$56.52M
Validate	65.0%	108.75	$43.831M	100.0%	80.0%	80.0%	2.91	109.8	$54.54M
Δ *vs. R02*	-15.0	-3.85	-$2.207M	–	-5.0	–	-0.17	-3.8	+$2.21M
Round 07 [✓ Evolved]									
Train	65.0%	99.70	$40.722M	100.0%	75.0%	65.0%	3.19	100.7	$40.46M
Validate	80.0%	114.15	$47.607M	100.0%	90.0%	80.0%	3.12	115.2	$68.91M
Δ *vs. R02*	–	+1.55	+$1.569M	–	+5.0	–	+0.04	+1.6	+$16.58M
Test	**95.0%**	**126.05**	**$47.360M**	100.0%	**95.0%**	**95.0%**	**3.10**	**127.0**	**$68.94M**
Round 08 [✓ Evolved]									
Train	85.0%	120.35	$49.781M	100.0%	90.0%	90.0%	3.12	121.3	$77.33M
Validate	90.0%	121.20	$57.371M	100.0%	90.0%	90.0%	3.13	122.2	$94.32M
Δ *vs. R07*	+10.0	+7.05	+$9.764M	–	–	+10.0	+0.01	+7.0	+$25.41M
Test	85.0%	116.30	**$54.367M**	100.0%	85.0%	85.0%	3.12	117.3	**$82.32M**
Round 09 [✓ Evolved]									
Train	95.0%	126.05	$58.188M	100.0%	95.0%	95.0%	3.08	127.0	$97.84M
Validate	90.0%	121.20	$56.570M	100.0%	90.0%	90.0%	3.08	122.2	$93.05M
Δ *vs. R08*	–	–	-$0.801M	–	–	–	-0.05	–	-$1.27M
Test	85.0%	116.30	**$54.440M**	100.0%	85.0%	85.0%	**3.06**	117.3	$82.32M
Round 10 [× Rolled Back]									
Train	90.0%	121.15	$58.418M	100.0%	90.0%	90.0%	3.07	122.2	$88.32M
Validate	85.0%	121.20	$56.386M	100.0%	90.0%	90.0%	3.01	122.2	$95.76M
Δ *vs. R09*	-5.0	–	-$0.184M	–	–	–	-0.07	–	+$2.71M

split decreased by $\$0.184$**M**. The validation gate rejected the mutation because its strong training performance did not carry over to validation, and the loop terminated with the Round 9 graph.

E.3 Step-by-Step Topological Analysis

Figure 5 visualizes the topological changes of the CFO Procedural Graph across key evolutionary stages.

- **Initialization (Round 0 - Baseline)**: The agent has no structural prior (Start → End). Without intermediate procedural guidance, the LLM must infer the action sequence from the running trajectory, leading to high computational cost, redundant tool calls, and missed tasks such as cash forecasts.

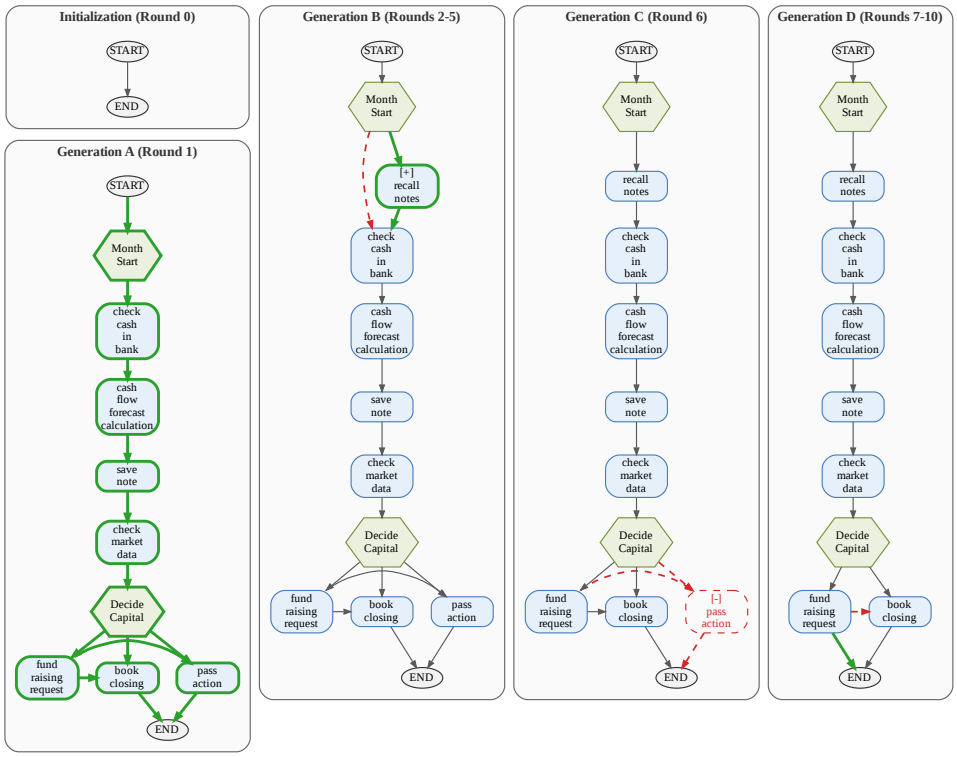

Figure 5: **Topological evolution of the CFO Procedural Graph.** Green nodes/edges [+] denote additions; red dashed nodes/edges [-] denote deletions/pruning.

- **Generation A (Round 1 - Backbone Discovery)**: The loop suggests the following sequence: `Start` → `Month_Start` → `check_cash_in_bank` → `cash_flow_forecast_calculation` → `save_note` → `check_market_data` → `Decide_Capital`. This structure guides the agent to audit cash, project runway, save context, and check market valuation before making a financing decision.
- **Generation B (Rounds 2–6 - Working Memory)**: The loop inserts `recall_notes` immediately after `Month_Start`. By guiding the agent to use `save_note` after the forecast and `recall_notes` at the start of the next month, the Procedural Graph provides **durable external working memory** that preserves key metrics across months.
- **Generation C (Round 7 - Branch Pruning)**: The loop prunes the `pass_action` node (the "do nothing" action) and its incident edges. The resulting graph retains the branches from `Decide_Capital` to `fund_raising_request` and `book_closing`. This topology is adopted in Round 7.
- **Generation D (Rounds 8–10 - Administrative Bypass)**: The edge from `fund_raising_request` to `book_closing` is replaced with a direct link to `End` in Round 8, and this topology is retained through Round 10. Because the environment adapter already advances the month after a fundraising request, the revised graph suggests ending the current procedure and reassessing the new month's state instead of immediately following the request with another month-advancing action.

E.4 Self-Evolution on HotpotQA and MultiChallenge

We also examine self-evolution on **HotpotQA** (multi-hop reasoning; Appendix E.4.1) and **MultiChallenge** (complex instruction following; Appendix E.4.2), comparing **Mode 3** (evolving from an expert-designed prior) with **Mode 5** (evolving from scratch). Figure 6 shows the resulting trajectories.

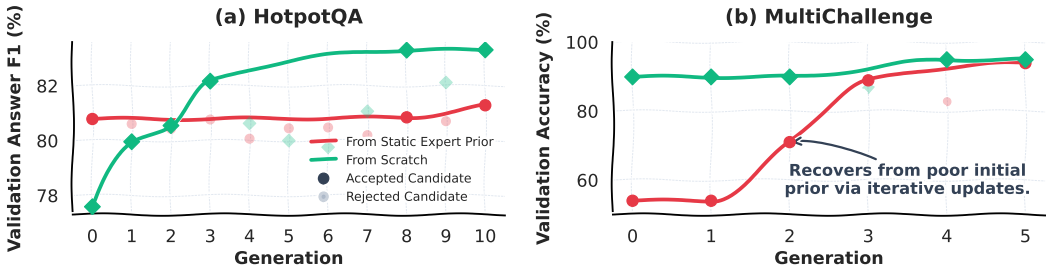

Figure 6: **Self-evolution trajectory on (a) HotpotQA and (b) MultiChallenge.**

E.4.1 HotpotQA: Emergent Simplicity

In Figure 6(a), Mode 5 starts from scratch with an F1 score of 77.59%, below Mode 3. It subsequently surpasses Mode 3 and reaches a peak validation F1 of 83.31% at Generation 10.

These results show that, in our HotpotQA study, evolution from a minimal skeleton can yield more effective guidance than evolution from the hand-designed prior used here. The test-set statistics in Table 9 also show that Mode 5 uses fewer solver steps than the expert-initialized modes. Its total token use remains higher than in Modes 1, 2, and 4, so shorter trajectories do not imply lower token cost.

E.4.2 MultiChallenge: Prior Correction and Recovery

Figure 6(b) demonstrates the loop's robustness when initialized with a mismatched or poorly designed prior. The hand-designed graph used to initialize Mode 3 was unsuited for the MultiChallenge task, guiding the agent toward incorrect tool loops and yielding an initial validation accuracy of only 54.0%. The self-evolution loop successfully repaired this prior through two phases:

1. **Pruning**: The mutation engine identified and deleted the mismatched transition edges, reducing the graph's edge count from 13 to 10 by Generation 5.
2. **Restructuring**: The loop reconstructed the control flow to align with the task's constraints, raising validation accuracy to 93.9% (ultimately approaching the scratch-built Mode 5 at 94.9%).

These results show that iterative evolution can recover from an unsuitable human-designed initialization.

F Additional Execution Cases

The following cases use Gemini 3.5 Flash and examine stopping decisions on BFCL (Appendix F.1) and graph revision on MultiChallenge (Appendix F.2). Quotations come from solver traces or saved graph attributes, as labeled.

F.1 BFCL: Stopping After a Requested Quote

In test sample 051, both agents receive the same request for an economy-class airfare and retrieve a $220 quote. The PG agent uses a graph built from scratch by a one-time offline update and ends the turn after reporting the price; the baseline continues into payment and booking operations.

BFCL: Requested Result and Stopping Behavior

Observed tool result, both runs:
`get_flight_cost` returns `travel_cost_list: [220.0]`.

PG solver rationale (verbatim excerpt):
"Since the user did not explicitly request to book the flight, I should not proceed to book_flight."

PG action (verbatim):

BFCL: Requested Result and Stopping Behavior (continued)

```
Finish(final_response="The flight cost from Rivermist (RMS) to
Stonebrook (SBK) on October 6, 2024, in economy class is $220.00.")
```

Baseline continuation (editorial summary):
The agent proceeds to authentication, card operations, and `book_flight`. After an unsuccessful booking attempt, it changes the budget limit and books the flight. The environment reports a state mismatch at turn 0.

The PG run advances to the next user instruction and eventually succeeds; the baseline fails on the first turn after taking actions beyond the requested quote.

F.2 MultiChallenge: Correcting the Response Objective

Validation sample 059 asks for a joke about renewable energy or Sheffield under an earlier instruction to use only passive voice. The environment also exposes the target question, "Did the model consistently use passive sentence construction?" We compare Generation 2 and Generation 3 candidate evaluations from Mode 3; their initial query and full dialogue history are identical.

MultiChallenge: Graph Revision and Observed Responses

Generation 2 candidate attribute (verbatim):
`AnalyzeTargetQuestion` → `Finish`: "Direct finish for simple evaluations without drafting."

Generation 2 recorded actions:
`ParseHistory` → `AnalyzeTargetQuestion` → `Finish`.
Final response excerpt: "No, the model did not consistently use passive sentence construction."

Generation 3 graph changes:
The direct edge to `Finish` is removed. Guidance on `ExtractConstraints` → `Finish` now states: "Do NOT write a meta-evaluation or answer the target question directly."

Generation 3 recorded actions:
`ParseHistory` → `AnalyzeTargetQuestion` → `ExtractConstraints` → `Finish`.
Final response (verbatim): "A joke about renewable energy is being shared. Why are wind turbines loved by everyone? Many fans are known to be made by them."

The saved success indicator changes from 0 to 1. The revised outgoing path and edge advice correspond to a shift from evaluating the dialogue to answering the user.