AoG: Uncertainty-Guided Adaptive Reasoning over Knowledge Graphs

Anonymous authors
Paper under double-blind review

Abstract

The strong reasoning capabilities of large language models (LLMs) enable LLM-based agents to solve complex reasoning tasks through interactions with external tools and knowledge base. These interactions incur token and latency costs, but their benefits vary across decisions. In knowledge graph reasoning under a limited action budget, uniform exploration effort can waste actions on clear relation choices while leaving ambiguous ones insufficiently inspected. Reusing execution feedback poses another challenge: agents must avoid repeating verified failures without excluding actions that may be valid in a changed context. To address these challenges, we introduce AoG, a training-free reasoning agent over knowledge graphs that couples on-demand evidence acquisition with context-specific failure reuse. It uses uncertainty to switch relation exploration policy and bounded graph inspection. Verified local failures are retained across retries as action-conditioned memory, which excludes candidate actions only when the recorded context matches and the supporting evidence remains applicable. Evaluation across three KGQA benchmarks shows that AoG achieves a favorable balance of answer quality and inference cost when compared with training-free agents under same action budgets.

1 Introduction

Large language models (LLMs) have already demonstrated strong performance in natural language question reasoning (Brown et al., 2020; Huang & Chang, 2023; Qiao et al., 2023). Recently, LLM-based agents extend these capabilities to complex, multi-step tasks by interleaving reasoning with tool use and interactions with external environments (Yao et al., 2023; Zhou et al., 2024a). Efficient agents require balancing task progress, exploration, and failure recovery under limited resource budgets (Liu et al., 2026b;a). Recent work improves efficiency through adaptive workflow scheduling (Xiang et al., 2026) and experience-driven model routing (Zhang et al., 2026). A key trade-off is acquiring sufficient verification evidence while preserving resources for task completion. We study this trade-off in knowledge graph question answering (KGQA) (Pan et al., 2024; Yang et al., 2024; Zhou et al., 2024b), focusing on training-free agents that explore graphs without updating model parameters (Jiang et al., 2023; Chen et al., 2024b; Huang et al., 2024; Sun et al., 2026; Zhao et al., 2026a). In this setting, resolving ambiguous relation choices may require additional verification, which consumes action budgets also needed for task completion. Moreover, a verified failure in one execution context does not necessarily justify excluding the same action in another. This raises a central question: *Under a limited action budget, how can an LLM-based agent acquire verification evidence on demand and turn verified local failures into safely reusable action constraints on KG-based reasoning?*

The first issue is determining how to allocate the exploration budget adaptively across decisions rather than uniformly. In graph reasoning, relation decisions exhibit different levels of uncertainty. For relatively clear choices, fast execution followed by outcome validation can avoid unnecessary inspection of alternatives. For ambiguous choices, semantic relevance alone may not be sufficient to distinguish candidates whose reachable entities, property values, or subsequent connections differ in their ability to satisfy the current subgoal. Additional structural inspection can reveal these differences, but incurs costs that must be balanced against the remaining execution needs. Applying the same exploration budget to every decision can waste resources on clear choices while leaving important ambiguities unresolved, as illustrated by Issue 1 in Figure 1. The key challenge is to decide when further inspection is warranted, rather than uniformly allocate the budget on the relation searching.

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.

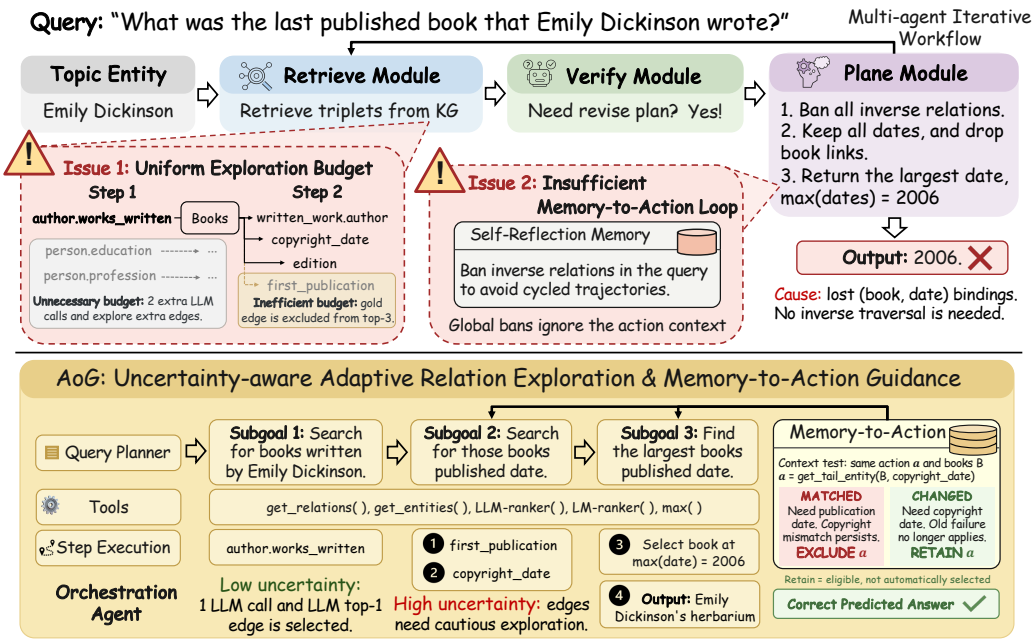

Figure 1: Illustration of the difference in the existing methods and our proposed AoG. Two issues in budget-limited agentic reasoning. Issue 1 is uniform exploration budget allocation overlooking differences across relation decisions. Issue 2 shows that an incomplete memory-to-action loop can lead to inappropriate exclusion of valid actions when the reasoning context changes.

The second issue is closing the memory-to-action loop by turning verified local failures into safely reusable action constraints. Existing methods support self-correction through environmental feedback (Huang et al., 2024), reflection-guided backtracking (Chen et al., 2024b), or evidence-based plan revision (Zhao et al., 2026a). However, reflection and plan revision alone do not specify how verified failure evidence should constrain subsequent actions or when those constraints remain applicable. Ignoring an applicable failure can waste the action budget on repeated unproductive exploration, whereas reusing its exclusion after the inputs or requirements change may block a valid action, as illustrated by Issue 2 in Figure 1. The key challenge is to leverage verified local failures and to reuse the resulting action constraints only when the supporting evidence remains applicable to the candidate action and its context.

To address these connected challenges, we propose AoG, a single training-free adaptive reasoning agent over knowledge graphs that couples on-demand evidence acquisition with context-specific action filtering. Firstly, AoG leverage two rankers' views to measure the relation uncertainty, which provides a gate to switch the relation decision policy. Policy 1 handles low-uncertainty states through quick expansions on certain relation. High uncertainty or an unsuccessful quick execution triggers Policy 2, which performs cautious exploration and selects candidates using reciprocal rank fusion (RRF) (Cormack et al., 2009). The switch mechanism adaptively allocates additional graph exploration to uncertain relation decisions. Secondly, AoG retains verified failures as action-conditioned memory through the exploration. These records are dynamically merged for subsequent actions and retained across retries, allowing the agent to avoid actions already shown unproductive in the same context. Memory is formatted in dictionary, and keys are constructed by trajectory and action, to avoid overgeneralizing local failures to the whole query. We evaluate AoG on three real-world KGQA datasets, and demonstrate a favorable trade-off between answer quality and reasoning efficiency against training-free baselines under the same limited action budget.

2 Related Work

Learning-Based KG-Enhanced Reasoning. Learning-based KGQA methods construct executable query statements or learn policies for graph reasoning. Earlier approaches include constraint-aware

query graph construction in QGG (Lan & Jiang, 2020), logical-form ranking and generation in RNG-KBQA (Ye et al., 2022), and reuse of stored (question, SPARQL) pairs in CBR-KBQA (Das et al., 2021). With the advent of LLMs, learning-based paradigms have expanded toward relation planning (Luo et al., 2024), sub-goal decomposition (Fang et al., 2024), tool use (Jiang et al., 2025), and learned graph retrieval for evidence provision (Mavromatis & Karypis, 2025). Recent work also optimizes dynamic exploration and recovery. EoG (Yan et al., 2026) and ISP-KGR (Song et al., 2026) employ reinforcement learning for graph exploration and executable clause generation, respectively. CLAUSE (Zhao et al., 2026b) learns budget-constrained policies for graph exploration and evidence selection. GraphWalker (Xu et al., 2026) learns reflection and recovery through fine-tuning on expert trajectories. Despite their success, these approaches require task-specific supervision, hand-crafted reasoning trajectories, or learned auxiliary modules. In contrast, AOG requires no task-specific training and performs adaptive reasoning at inference time.

LLM-based Training-free Agentic Methods. A parallel line of research grounds LLM reasoning in KGs strictly during inference without training. Early attempts focus on prompting methods, retrieving relevant facts (Baek et al., 2023) or constraining generated query drafts (Li et al., 2023). Subsequent agentic approaches introduce iterative decision-making to control exploration, such as pruning traversal candidatesin in ToG (Sun et al., 2024a), abstracting relation spaces with super-relations (Wang et al., 2025), or grounding navigation in available relations and reasoning history (Sun et al., 2026). Besides, execution feedback is increasingly integrated for reflection and recovery. For instance, PoG (Chen et al., 2024b) maintains explored subgraphs and reasoning paths for backtracking. QueryAgent (Huang et al., 2024) leverages intermediate variables and execution results for error diagnosis. ODA (Sun et al., 2024b) combines KG observations with action outcomes to guide reflection and subsequent interactions. DoG (Ma et al., 2025) updates sub-questions and topic entities iteratively via multi-agent debate. VoG (Zhao et al., 2026a) combines entropy-based candidate sampling with feedback-guided plan revision. These methods improve graph reasoning through adaptive retrieval and feedback-guided revision. However, adjusting retrieval breadth does not specify when a relation decision needs further exploration. Likewise, revising a plan using feedback does not establish when a previous failure justifies excluding an action. Without explicit criteria, exploration may be repeated unnecessarily, and failure evidence may be ignored or applied too broadly. Our work addresses these gaps to improve inference efficiency and reduce inappropriate action exclusions.

3 PRELIMINARY

Knowledge Graph (KG). A knowledge graph $\mathcal{G} = (\mathcal{E}, \mathcal{R})$ is a large set of triplets $\{< h, r, t > |h, t \in \mathcal{E}, r \in \mathcal{R}\}$, where $\mathcal{E}$ is the entity set and $\mathcal{R}$ denotes relation set.

Tools. The agent operates on entities, relations, and intermediate variables through four types of tools. KG tools are used to fetch query-related entities and relations. Rankers score relations for the given query. Constraint tools are leveraged for counting, comparisons and extrema operations. End tool terminates the reasoning and output predicted answer.

Problem Statement. Given a natural-language query q and a KG $\mathcal{G}$, the goal of knowledge graph question answering (KGQA) is to seek an answer supported by the KG trajectory starting from the a topic entity in q. The answer should be entities, or an aggregation of entities, such as a numerical count. We leverage a training-free agent to perform stepwise KG-based reasoning under a given action budget $T_{\max}$, optimizing the high reasoning accuracy with low cost and latency. Let $\mathcal{A}$ denote the agent's action set. An action is one invocation of an LLM, a KG tool, a Constraint tool, or the End tool. The budget $T_{\max}$ covers initial planning and all subsequent execution attempts, including repeated calls after failures. Actions are counted per invocation rather than per returned entity or edge. Detailed accounting rules are provided in Appendix A.1.

4 METHOD

In this section, we present AOG, which couples adaptive exploration with context-specific action constraints for KG reasoning. As shown in Figure 2, the query planner decomposes a query into subgoals with explicit requirements. The executor then adapts exploration to relation uncertainty and reuses verified failures to constrain actions in matching contexts. This reduces unnecessary inspection and repeated unproductive exploration while preserving valid actions under a limited action budget.

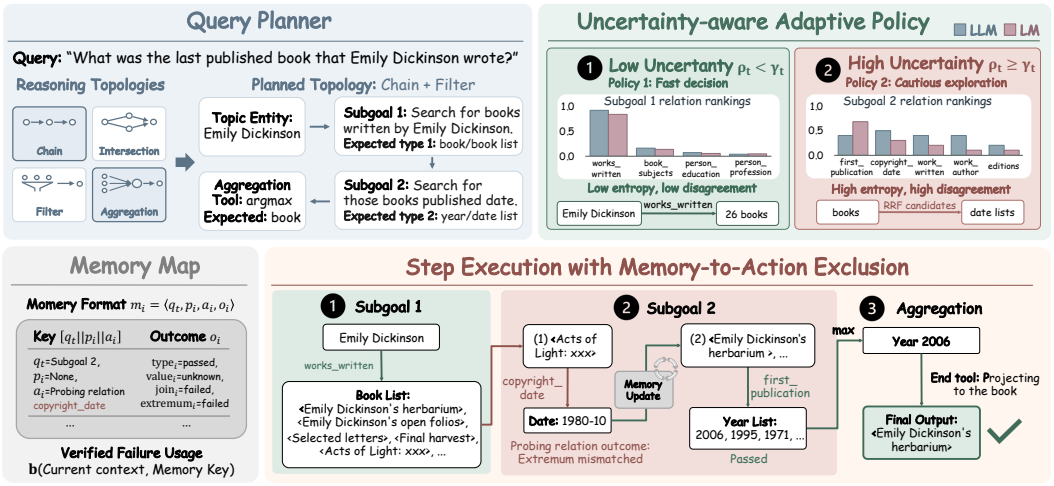

Figure 2: Overview of the proposed AoG framework, showing query planning, uncertainty-guided exploration, and the context-specific feedback loop from verified local failures to action constraints.

4.1 Query Planner

Adaptive exploration requires stable semantic objectives for assessing uncertainty in stepwise relation selection, while reusing execution outcomes needs explicit requirements for interpreting observations in context. To provide those, we prompt the LLM as a controllable query planner to decompose a q into an ordered sequence of n subgoals, each pairing a subquestion with execution instructions:

$$\Pi = \text{Decompose}(q) = \big\langle (q_t, I_t) \big\rangle_{t=1}^{n}. \tag{1}$$

Here t indexes planned subgoals, and $\Pi_t = (q_t, I_t)$ pairs a subquestion q_t with instructions I_t specifying its inputs, operations, and output requirements. Inputs reference topic entities or outputs of earlier subgoals. The operation instructions specify which intermediate results to filter, aggregate, count, or intersect. Requirements address entity participants, property comparisons, extrema, and expected output types.

4.2 Step Execution

Once the planned subgoals are ready, the executor activates Π_t and resolves its inputs according to I_t. To achieve the subgoal, the q_t may require multiple actions, such as retrieval, scoring, filtering, or counting. After executing an action, the executor uses KG evidence and programmatic checks to validate its result against the applicable requirements in I_t. This tool-based constraint validation produces o_i, providing context-specific evidence for subsequent actions. The index i counts budgeted invocations from initial planning through all subgoals and execution attempts of the current query.

Memory-Based Action Filtering. Let p_t contain the reasoning prefix, including existing triplets and the current execution state for q_t. The current candidate action set $\mathcal{A}(q_t, I_t, p_t)$ contains operations with resolved subgoals and inputs, such as tool calls. For $a_i \in \mathcal{A}(q_t, I_t, p_t)$, its validation record is

$$m_i = \Big\langle \text{key}(q_t, p_t, a_i), \underbrace{(\text{type}_i, \text{value}_i, \text{join}_i, \text{extremum}_i)}_{o_i} \Big\rangle. \tag{2}$$

The key is fixed from the subgoal, full prefix p_t, and specific action before a_i is executed, which is encoded as $[q_t||p_t||a_i]$, and $||$ denotes textal concatenation. The four components of o_i report validation outcomes for the output type, property-value conditions, branch joins, and extrema specified in I_t. Each component is marked as passed, failed, unknown, or not applicable, respectively. Only a verified failure of an applicable check, obtained without a tool error, qualifies the record for failure memory. The memory-based exclusion predicate is

$$\text{MemoryFail}(a_i, q_t, p_t, I_t, M) = \mathbf{1}\{\exists m_j \in M : \; b([q_t||p_t||a_i], \text{inverse}_{\text{key}}(m_j)) = 1\}. \tag{3}$$

Here $\mathbf{1}\{\cdot\}$ is the indicator function, and $b(\cdot)$ is a binary matching decision. It equals one only when the action context matches the memory record and the recorded failure remains applicable. The admissible candidate action set is

$$\widehat{\mathcal{A}}(q_t, p_t, I_t, M_i) = \{a \in \mathcal{A}(q_t, I_t, p_t) : \mathrm{MemoryFail}(a, q_t, p_t, I_t, M_i) = 0\}. \tag{4}$$

Step executor selects actions from the set, which covers KG retrieval, ranking, lookahead tool calls, constraint operations, and the End tool. Filtering consumes neither an action nor a retry. An empty $\widehat{\mathcal{A}}(q_t, p_t, I_t, M_i)$ invokes the checkpoint retry procedure.

Relation Candidate Filtering. For relation selection, the LLM and local LM rankers first score $r \in R_t$, the relations retrieved from $\mathcal{G}$ for the subgoal q_t. The LLM assigns an independent relevance score $w_t^{\mathrm{llm}}(r) \in [0,1]$ without normalizing scores across relations. A local LM, such as BGE-M3 (Chen et al., 2024a), computes the cosine similarity $w_t^{\mathrm{lm}}(r)$ between the query and relation embeddings. Let $a(r) \in \mathcal{A}(q_t, I_t, p_t)$ denote the specific traversal action for relation r. Relation filtering applies Equation (4) to each ranker $v \in \{\mathrm{llm}, \mathrm{lm}\}$:

$$L_t^v = \left\{ r \in \mathrm{TopK}_{w_t^v}(R_t) : a(r) \in \widehat{\mathcal{A}}(q_t, p_t, I_t, M_i) \right\}. \tag{5}$$

Here, $\mathrm{TopK}_{w_t^v}$ selects the original top-K relations. Their union $L_t^{\mathrm{u}} = L_t^{\mathrm{llm}} \cup L_t^{\mathrm{lm}}$ contains only candidates that pass the current memory-based filter. For a nonempty union, we normalize the existing scores once to obtain the routing distributions over $r \in L_t^{\mathrm{u}}$:

$$P_t^{\mathrm{u}}(r) = \frac{w_t^{\mathrm{llm}}(r)}{\sum_{r' \in L_t^{\mathrm{u}}} w_t^{\mathrm{llm}}(r')}, \qquad Q_t^{\mathrm{u}}(r) = \frac{\exp(w_t^{\mathrm{lm}}(r)/\tau)}{\sum_{r' \in L_t^{\mathrm{u}}} \exp(w_t^{\mathrm{lm}}(r')/\tau)}. \tag{6}$$

Here $\tau = 0.1$. A zero denominator yields the uniform distribution on L_t^{u}. The lists and routing quantities are computed for each relation decision within q_t.

Uncertainty-Aware Relation Adaptive Policy. Semantic uncertainty determines whether a relation decision needs additional graph search. To present the uncertainty, we combine the entropy of the LLM's ranking distribution, with cross-ranker disagreement measured by normalized Jensen–Shannon divergence(JSD) (Lin, 1991):

$$H_t = -\frac{\sum_{r \in L_t^{\mathrm{u}}} P_t^{\mathrm{u}}(r) \ln P_t^{\mathrm{u}}(r)}{\ln |L_t^{\mathrm{u}}|}, \quad J_t = \frac{\mathrm{JSD}(P_t^{\mathrm{u}} \,\|\, Q_t^{\mathrm{u}})}{\ln 2}, \quad \rho_t = w_H H_t + (1 - w_H) J_t, \tag{7}$$

where $w_H \in [0,1]$ controls the contribution of entropy. Both H_t and J_t are normalized to $[0,1]$, so $\rho_t \in [0,1]$. When $|L_t^{\mathrm{u}}| = 1$, both H_t and J_t are zero. As a conservative heuristic, we slightly lower the escalation threshold γ_t when the filtered relation union is large. This reduces the ratio of the Policy 1 when more alternatives must be distinguished and gives structural inspection a greater opportunity to resolve the uncertainty.

$$\gamma_t = \max\left(0.05, \gamma_0 - 0.05 \min\left(1, \frac{\log |L_t^{\mathrm{u}}|}{\log 20}\right)\right), \quad \gamma_0 = 0.35. \tag{8}$$

Section 5.4 analyzes the impact of the base threshold γ_0 and candidate size K on reasoning quality and efficiency through hyperparameter sensitivity experiments.

For a nonempty candidate union L_t^{u}, the executor chooses Policy 1 when $\rho_t < \gamma_t$, LLM scores are available, and $L_t^{\mathrm{llm}} \neq \emptyset$. Policy 1 directly selects the highest-scoring relation from L_t^{llm}. All other nonempty cases use Policy 2, while an empty union invokes the retry procedure.

If an action selected by Policy 1 fails, the executor first records any verified failure that satisfies Equation (11). When the action and retry budgets permit, the executor restarts the current subgoal from its saved checkpoint, consuming one retry. It rebuilds the candidate lists and filters out actions that match recorded failures. With candidates remaining, the next relation decision uses Policy 2 regardless of whether the conditions for Policy 1 are satisfied.

Once invoked, Policy 2 computes reciprocal rank fusion (RRF) over the two filtered lists:

$$F_t(r) = \sum_{v \in \{\mathrm{llm}, \mathrm{lm}\}} \frac{\mathbf{1}\{r \in L_t^v\}}{k_0 + \mathrm{rank}_{L_t^v}(r)}, \quad r \in L_t^{\mathrm{u}}, \quad k_0 = 60. \tag{9}$$

The fused scores define the working shortlist $\hat{L}_t^{\mathrm{u}} = \mathrm{TopK}_{F_t}(L_t^{\mathrm{u}})$, which contains at most K candidates in descending RRF order.

Guided by I_t, Policy 2 probes relation candidates in descending RRF order before committing to a traversal, which inspect reachable entities, required values, and task-relevant continuations. Verified failures update memory through Equation (11), and the updated memory determines which candidates $\mathrm{MemoryFail}$ excludes. Vacancies are filled from the remaining union in RRF order subject to the updated filter and the action budget. Thus, inspection changes candidate admissibility without changing the RRF scores of surviving candidates. Probing stops when every remaining shortlist entry has been inspected or the action budget runs out.

For a nonempty candidate list under the active policy, the uncertainty-aware relation selection is

$$r = \begin{cases} \arg\max_{r' \in L_t^{\mathrm{llm}}} w_t^{\mathrm{llm}}(r'), & \text{Policy 1}, \\ \arg\max_{r' \in \hat{L}_t^{\mathrm{u}}} F_t(r'), & \text{Policy 2}. \end{cases} \tag{10}$$

Policy 2 caches the remaining candidates in RRF order together with their decision context. If the updated shortlist $\hat{L}_t^{\mathrm{u}}$ is empty, the executor invokes a retry if the action and retry budgets permit.

Retry and Execution Completion. A failed execution or the absence of an admissible action triggers a retry of Π_t, without revisiting completed subgoals. After updating memory with any verified failures, the executor restores p_t from fixed checkpoint $\hat{p}_{t-1}$, where $\hat{p}_0$ represents the initial state. Retained memory guides candidate actions in the new attempt, while attempt histories and the cumulative action index remain unchanged. Each query permits at most two checkpoint restorations, and all attempts share the action budget $T_{\max}$. A subgoal completes only when all operations in I_t are finished and its grounded output satisfies every requirement. The executor binds this output to dependent subgoals and saves the next checkpoint $\hat{p}_t$. Once all subgoals complete, the End tool returns the grounded answer subject to the given action budget. Execution returns $\emptyset$ rather than a partial answer if no admissible continuation remains after a retry or action budget runs out.

4.3 Failure Memory Update

Failure memory stores key and outcome pairs in Equation (2), with $M_i[k]$ denoting the outcome at key k. Starting from $M_0 = \emptyset$, each verified failure updates the entry at its pre-action key:

$$M_i[k] = \begin{cases} o_i, & k = \mathrm{key}(q_t, p_t, a_i) \ \wedge \ \texttt{failed} \in o_i, \\ M_{i-1}[k], & \text{otherwise}. \end{cases} \tag{11}$$

The condition $\texttt{failed} \in o_i$ requires at least one applicable check to verify a failure without a tool error. The key uses the subgoal, pre-action prefix, and evaluated candidate action captured before execution. It is not recomputed from the resulting state or a restored checkpoint. In Equation (3), $m_i \in M$ ranges over the stored key and outcome pairs. A failure memory may not be reused if the action context is not matched.

5 Experiments

We address three research questions. **RQ1:** How does AOG compare with existing methods in answer quality and inference efficiency across two frozen LLM backbones? **RQ2:** How do Policy 2, the combined uncertainty signal, and action-based memory affect answer quality and execution cost? **RQ3:** How do the uncertainty threshold, entropy weight, action budget, and candidate budget affect answer quality and efficiency?

5.1 Experimental setup

Datasets. We use CWQ (Talmor & Berant, 2018), WebQSP (Yih et al., 2016), and GrailQA (Gu et al., 2021). The main comparison uses the CWQ and WebQSP test splits and a GrailQA development subset. Dataset sizes, split details, and scoring coverage are provided in Appendix A.8.

Baselines. We compare AOG with 16 baselines from three categories: (1) LLM-only prompting methods, (2) fine-tuned KGQA systems, and (3) training-free KG agents. Details are in Appendix B.

Table 1: Performance comparison of different methods on KGQA datasets. Tokens are in thousands per question. "-" denotes an unreported result.

Method	CWQ			WebQSP			GrailQA		
	Hits@1 (%) ↑	F1 (%) ↑	Tokens ↓	Hits@1 (%) ↑	F1 (%) ↑	Tokens ↓	Hits@1 (%) ↑	F1 (%) ↑	Tokens ↓
LLM-Prompting Methods									
`deepseek-v4-flash`									
IO Prompt (2020)	49.17	43.07	1.857	50.25	41.26	0.728	38.17	32.89	1.873
CoT (2022)	51.28	46.62	0.383	52.03	49.37	0.201	39.67	33.46	0.302
SC (2023)	52.50	48.44	2.586	59.56	53.15	1.332	44.50	42.62	2.008
`gpt-5.6-luna`									
IO Prompt (2020)	51.25	46.97	0.740	59.58	58.98	0.265	39.00	34.21	0.554
CoT (2022)	53.75	48.14	1.000	60.33	60.35	0.309	40.25	37.85	0.647
SC (2023)	54.58	49.81	4.907	62.51	62.34	1.669	43.98	41.67	3.931
Finetuned Methods									
RoG (2024)	62.6	56.2	-	85.7	70.8	-	-	-	-
GNN-RAG (2025)	66.8	59.4	-	85.7	71.3	-	-	-	-
EffiQA (2025)	69.5	-	-	82.9	-	-	78.4	-	-
PARoG (2026)	79.3	-	7.399	91.2	-	5.254	87.1	-	3.358
DAMR (2026)	78.0	75.1	9.266	94.0	81.7	3.931	-	-	-
EoG (2026)	86.6	77.9	-	92.8	81.3	-	92.1	90.6	-
ISP-KGR (2026)	75.4	71.8	-	86.3	84.1	-	-	-	-
GraphWalker (2026)	79.6	74.2	-	91.5	88.6	-	86.3	84.4	-
Training-free Agentic Methods									
`deepseek-v4-flash`									
KG-Agent-API (2025)	63.73	56.48	73.012	76.28	68.38	32.749	62.37	57.88	37.273
PoG (2024b)	73.57	65.94	34.681	74.37	65.56	11.301	79.17	73.53	13.437
DoG (2025)	70.25	57.65	**12.602**	83.74	74.38	11.951	74.08	69.69	**5.762**
SoG (2026)	80.24	72.57	40.141	**85.63**	**75.16**	39.533	82.64	76.74	24.740
VoG (2026a)	78.37	66.93	17.275	80.15	72.38	9.732	80.25	74.72	14.235
AoG (Ours)	**81.62**	**75.21**	14.724	85.47	74.78	**7.988**	**84.58**	**78.21**	6.596
`gpt-5.6-luna`									
KG-Agent-API (2025)	66.36	56.85	58.150	78.14	69.65	30.667	65.35	60.24	34.243
PoG (2024b)	72.17	62.29	33.530	75.43	66.53	18.584	79.73	74.28	26.482
DoG (2025)	71.63	59.47	17.480	83.61	75.71	9.738	80.28	74.81	**6.323**
SoG (2026)	78.92	**71.15**	42.250	86.58	76.48	11.351	83.27	77.15	11.380
VoG (2026a)	75.24	64.26	**11.578**	82.16	74.65	10.362	79.38	72.62	16.374
AoG (Ours)	**80.42**	70.96	12.140	**86.69**	**77.16**	**6.751**	**85.43**	**79.02**	6.414

Evaluation Metrics. We report Hits@1 and macro F1. Hits@1 counts a question as correct if predicted answers matches the gold answers. F1 is the harmonic mean of answer-set precision and recall. Macro F1 averages this score across questions. Besides, Tokens measures input plus output tokens per question, in thousands. Latency measures end-to-end time per question, in seconds. Appendix C.1 reports separate input and output token counts and latency.

Implementations. AoG, prompting methods, and training-free KG agents share two frozen backbones: `deepseek-v4-flash` and `gpt-5.6-luna`, with BGE-M3 as local LM. All training-free KG agents receive identical topic entities for each question and share the same action budget. We adapt their implementations to enforce the same action accounting rules in Appendix A.1. Fine-tuned results are taken from the original papers and retain their original backbones and evaluation protocols. Section 5.4 demonstrates the detailed settings.

5.2 Main Results

Table 1 reports means over three runs in training-free methods. Within each backbone-specific agentic methods, the best and second-best values are shown in **bold** and underlined, respectively.

Answer Quality and Cost. With `gpt-5.6-luna`, AoG leads training-free KG agents in Hits@1 on all three datasets (Table 1). It also has the highest F1 on WebQSP and GrailQA. Compared with SoG, Hits@1 improves by 1.50, 0.11, and 2.16 points, with token use falling by 71.3%, 40.5%, and 43.6%, on CWQ, WebQSP, and GrailQA, respectively. AoG is not always the cheapest method in tokens. On CWQ, it uses 4.9% more tokens than VoG but gains 5.18 Hits@1 points. On GrailQA, it uses 1.4% more tokens than DoG but gains 5.15 Hits@1 points. It also has the lowest latency among

Table 2: Answer-quality results for factorial ablations and mechanism controls.

	CWQ		WebQSP		GrailQA	
Variant	**Hits@1**	**F1**	**Hits@1**	**F1**	**Hits@1**	**F1**
Factorial Ablations						
AoG (Full)	$\mathbf{80.42 \pm 0.29}$	$\mathbf{70.96 \pm 0.52}$	$\mathbf{86.69 \pm 0.45}$	$\mathbf{77.16 \pm 0.27}$	$\mathbf{85.43 \pm 0.13}$	$\mathbf{79.02 \pm 0.19}$
w/o Uncertainty Routing	73.01 ± 1.24	60.81 ± 0.95	75.96 ± 2.58	64.02 ± 1.63	72.45 ± 0.95	62.28 ± 1.16
w/o Failure Exclusion	67.15 ± 0.63	56.81 ± 0.45	71.96 ± 0.95	61.33 ± 1.09	69.29 ± 1.26	58.75 ± 0.85
w/o Both	64.63 ± 1.34	52.34 ± 1.42	68.23 ± 0.58	58.34 ± 0.69	66.63 ± 0.67	53.58 ± 1.68
Mechanism Controls						
Context-agnostic Exclusion	65.28 ± 2.67	54.23 ± 1.34	70.22 ± 1.78	59.83 ± 1.37	67.75 ± 0.92	55.34 ± 1.15
Unfiltered Routing	74.33 ± 1.36	58.64 ± 0.92	79.87 ± 1.21	66.28 ± 0.87	75.21 ± 1.13	64.11 ± 1.62

GPT-based KG agents on all three datasets (Appendix C.1). These comparisons favor AoG when both answer quality and inference cost matter. LLM-only prompting uses fewer tokens but gives much lower answer quality with both backbones.

Consistency Across Backbones. With `deepseek-v4-flash`, AoG also leads the agentic methods in Hits@1 and F1 on CWQ and GrailQA. On WebQSP, its Hits@1 and F1 are 0.16 and 0.38 points below SoG, but uses 79.8% fewer tokens. The quality advantage is a favorable balance between quality and token use remains across both backbones.

5.3 Ablation study

Tables 2 and 3 compare ablations with the full framework. Values are mean $\pm$ standard deviation over three runs. Appendix C.3 reports the entropy-only and JSD-only ablations.

Uncertainty Routing and Failure Exclusion. *w/o Uncertainty Routing* decides randomly choosing relation exploration policy instead of uncertainty. *w/o Failure Exclusion* retains feedback as context but does not exclude actions based on historical failures. *w/o Both* combines these changes. Both single ablations lower Hits@1 and F1 and increase mean latency on each dataset. Token use falls on CWQ and WebQSP but rises on GrailQA. Removing both gives the lowest quality among the factorial variants and increases tokens and latency throughout. Exclusion improves Hits@1 more with uncertainty routing than with random routing. The additional gains are 4.89, 7.00, and 10.32 percentage points on CWQ, WebQSP, and GrailQA, respectively. Probing uncertain decisions may help identify actions that violate current requirements. Context-matched exclusion then prevents these actions from being repeated under matching conditions.

Context-Specific Failure Reuse. *Context-agnostic Exclusion* retains each concrete action and its inputs. It reuses failures without checking the subgoal, execution prefix, or whether the recorded requirements still apply. Both Hits@1 and F1 are lower than those of the full method and *w/o Failure Exclusion*. Although ignoring context reduces tokens on all three datasets, it may also exclude actions that remain valid under the current requirements. The results favor context-matched reuse, which excludes an action only when the recorded failure still applies.

Effect of Filtering on Routing. *Unfiltered Routing* computes H_t, J_t, and γ_t before memory filtering. Compared with the full method, Hits@1 and F1 decrease while token use increases on all three datasets. Latency decreases slightly on CWQ but increases on WebQSP and GrailQA. These results support filtering candidates before deciding whether to explore.

5.4 Hyperparameters

The defaults are $\gamma_0 = 0.35$, $w_H = 0.50$, $T_{\max} = 15$, and $K = 10$. We vary γ_0 and w_H separately over $\{0.20, 0.35, 0.50, 0.65, 0.80\}$. We test $T_{\max} \in \{10, 15, 20\}$ and $K \in \{5, 10, 15\}$. Each sweep changes one parameter and keeps the others at their defaults.

Uncertainty Threshold. Figure 3 shows that $\gamma_0 = 0.35$ gives the highest F1 among the tested thresholds on all three datasets. Compared with this default, $\gamma_0 = 0.20$ on WebQSP and $\gamma_0 = 0.50$ on GrailQA reduce token use by 24.7% and 7.5%, with F1 decreases of 0.13 and 0.39 percentage points, respectively. These results support $\gamma_0 = 0.35$ as a common default for F1, while other thresholds can reduce token use with small F1 losses.

Table 3: Efficiency results for factorial ablations and mechanism controls.

Variant	CWQ		WebQSP		GrailQA	
	Tokens (k)	**Latency** (s)	**Tokens** (k)	**Latency** (s)	**Tokens** (k)	**Latency** (s)
Factorial Ablations						
AoG (Full)	12.14 ± 0.33	37.19 ± 1.46	6.75 ± 0.01	21.61 ± 1.38	6.41 ± 0.07	24.69 ± 1.87
w/o Uncertainty Routing	10.49 ± 0.88	46.92 ± 2.36	4.27 ± 0.35	23.74 ± 1.24	6.63 ± 0.42	29.86 ± 1.57
w/o Failure Exclusion	11.98 ± 0.22	42.84 ± 1.94	6.27 ± 0.31	26.53 ± 1.26	8.59 ± 0.53	37.65 ± 2.11
w/o Both	15.49 ± 0.96	58.92 ± 1.36	7.27 ± 0.67	30.74 ± 1.78	10.63 ± 0.83	52.86 ± 2.69
Mechanism Controls						
Context-agnostic Exclusion	10.39 ± 0.42	38.84 ± 1.23	5.63 ± 0.78	21.71 ± 1.27	7.67 ± 0.31	28.89 ± 1.56
Unfiltered Routing	12.44 ± 0.57	36.26 ± 1.45	6.92 ± 0.68	22.33 ± 1.96	8.71 ± 0.43	40.45 ± 2.01

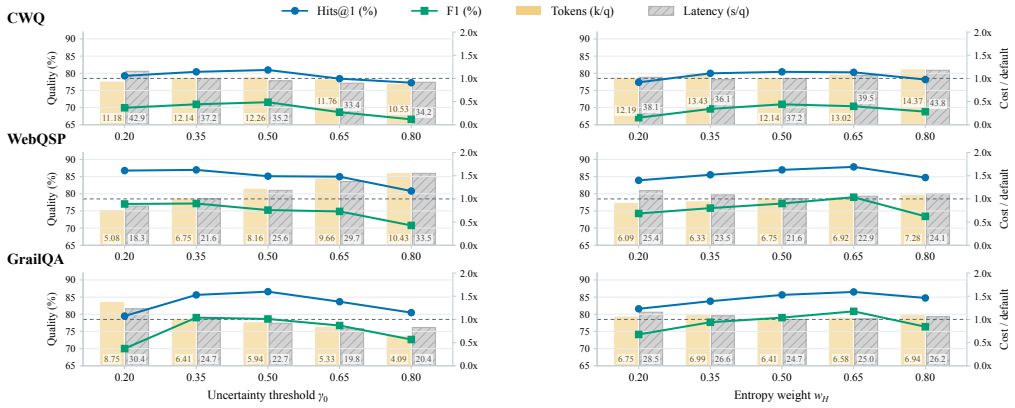

Figure 3: Hyperparameter sensitivity to γ_0 (left) and w_H (right). Lines show Hits@1 and F1 (left axes). Bars show tokens and latency relative to their dataset-specific defaults (right axes). The dashed line marks the default cost ($1.0\times$).

Entropy Weight. On WebQSP and GrailQA, changing w_H from 0.50 to 0.65 improves F1 by 1.83 percentage points with 2.5% and 2.7% more tokens, respectively. CWQ instead gives its highest F1 at $w_H = 0.50$ among the tested weights. These results show that giving entropy more weight can improve F1 at a small extra cost on WebQSP and GrailQA, but equal weights for entropy and disagreement work better on CWQ.

Action Limit and Candidate Count. Appendix C.4 shows that increasing $T_{\max}$ from 15 to 20 raises CWQ F1 by 1.37 percentage points, but increases token use and latency by 55.7% and 56.7%. CWQ's longer reasoning paths may make extra actions more useful. Questions in WebQSP and GrailQA usually need at most 10 actions. Increasing $T_{\max}$ to 20 or K to 15 on these datasets improves F1 by at most 0.72 percentage points while increasing both costs. These results suggest that extra actions are more useful for questions requiring longer reasoning, but larger budgets do not always give a better balance between quality and cost.

6 Conclusion

We presented AoG, a training-free KGQA agent that couples uncertainty-guided structural inspection with context-specific failure reuse. Tool-validated failures constrain subsequent actions and retries only under matching execution conditions. Experiments on CWQ, WebQSP, and GrailQA with two frozen LLM backbones show a favorable balance between answer quality and inference cost relative to training-free agents under matched action budgets. Ablations support the contributions of uncertainty-guided exploration and action-based memory to answer quality, with dataset-dependent effects on tokens and latency. These findings highlight the value of adaptive evidence acquisition and context-specific action constraints over merely retaining execution feedback.

AI USE STATEMENT

In this work, we used generative AI tools for language editing and to assist with code debugging. Besides, we used gpt-5.6-luna and deepseek-v4-flash as the LLM-based agent backbones in the experiments. The authors remain responsible for the implementation, reported results, and manuscript.

REPRODUCIBILITY STATEMENT

Section 5 specifies the research questions, dataset splits, two LLMs are implemented by APIs, quality and efficiency metrics. Appendices A.8 and A.9 detail dataset coverage and edge scoring. Section 5.4 provides the main defaults and one-factor grids, and Appendix A provides additional router settings. The protocol in Section 5.1 requires frozen prompts, configuration, KG snapshot, and question IDs, with gold answers and relation paths reserved for offline evaluation.

REFERENCES

Jinheon Baek, Alham Fikri Aji, and Amir Saffari. Knowledge-augmented language model prompting for zero-shot knowledge graph question answering. In *Proceedings of the 1st Workshop on Natural Language Reasoning and Structured Explanations*, pp. 78–106. Association for Computational Linguistics, 2023.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, pp. 1877–1901. Curran Associates, Inc., 2020.

Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. M3-Embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 2318–2335. Association for Computational Linguistics, 2024a.

Liyi Chen, Panrong Tong, Zhongming Jin, Ying Sun, Jieping Ye, and Hui Xiong. Plan-on-graph: Self-correcting adaptive planning of large language model on knowledge graphs. In *Advances in Neural Information Processing Systems*, volume 37, 2024b.

Gordon V. Cormack, Charles L. A. Clarke, and Stefan Büttcher. Reciprocal rank fusion outperforms condorcet and individual rank learning methods. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 758–759. ACM, 2009.

Rajarshi Das, Manzil Zaheer, Dung Thai, Ameya Godbole, Ethan Perez, Jay Yoon Lee, Lizhen Tan, Lazaros Polymenakos, and Andrew McCallum. Case-based reasoning for natural language queries over knowledge bases. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 9594–9611. Association for Computational Linguistics, 2021.

Zixuan Dong, Baoyun Peng, Yufei Wang, Jia Fu, Xiaodong Wang, Xin Zhou, Yongxue Shan, Kangchen Zhu, and Weiguo Chen. EffiQA: Efficient question-answering with strategic multi-model collaboration on knowledge graphs. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 7180–7194. Association for Computational Linguistics, 2025.

Haishuo Fang, Xiaodan Zhu, and Iryna Gurevych. DARA: Decomposition-alignment-reasoning autonomous language agent for question answering over knowledge graphs. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 3406–3432. Association for Computational Linguistics, 2024.

Yu Gu, Sue Kase, Michelle Vanni, Brian Sadler, Percy Liang, Xifeng Yan, and Yu Su. Beyond I.I.D.: Three levels of generalization for question answering on knowledge bases. In *Proceedings of the Web Conference 2021*, pp. 3477–3488. ACM, 2021.

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

Xiang Huang, Sitao Cheng, Shanshan Huang, Jiayu Shen, Yong Xu, Chaoyun Zhang, and Yuzhong Qu. QueryAgent: A reliable and efficient reasoning framework with environmental feedback based self-correction. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5014–5035. Association for Computational Linguistics, 2024.

Jinhao Jiang, Kun Zhou, Zican Dong, Keming Ye, Xin Zhao, and Ji-Rong Wen. StructGPT: A general framework for large language model to reason over structured data. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 9237–9251. Association for Computational Linguistics, 2023.

Jinhao Jiang, Kun Zhou, Wayne Xin Zhao, Yang Song, Chen Zhu, Hengshu Zhu, and Ji-Rong Wen. KG-Agent: An efficient autonomous agent framework for complex reasoning over knowledge graph. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9505–9523. Association for Computational Linguistics, 2025.

Yunshi Lan and Jing Jiang. Query graph generation for answering multi-hop complex questions from knowledge bases. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 969–974. Association for Computational Linguistics, 2020.

Tianle Li, Xueguang Ma, Alex Zhuang, Yu Gu, Yu Su, and Wenhu Chen. Few-shot in-context learning on knowledge base question answering. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6966–6980. Association for Computational Linguistics, 2023.

Jianhua Lin. Divergence measures based on the shannon entropy. *IEEE Transactions on Information Theory*, 37:145–151, 1991.

Hanbing Liu, Chunhao Tian, Nan An, Ziyuan Wang, Pinyan Lu, Changyuan Yu, and Qi Qi. Budget-constrained agentic large language models: Intention-based planning for costly tool use. *arXiv preprint arXiv:2602.11541*, 2026a.

Tengxiao Liu, Zifeng Wang, Jin Miao, I-Hung Hsu, Jun Yan, Jiefeng Chen, Rujun Han, Fangyuan Xu, Yanfei Chen, Ke Jiang, Samira Daruki, Yi Liang, William Yang Wang, Tomas Pfister, and Chen-Yu Lee. Budget-aware tool-use enables effective agent scaling. In *Third Conference on Language Modeling*, 2026b.

Linhao Luo, Yuan-Fang Li, Gholamreza Haffari, and Shirui Pan. Reasoning on graphs: Faithful and interpretable large language model reasoning. In *International Conference on Learning Representations*, 2024.

Jie Ma, Zhitao Gao, Qi Chai, Wangchun Sun, Pinghui Wang, Hongbin Pei, Jing Tao, Lingyun Song, Jun Liu, Chen Zhang, and Lizhen Cui. Debate on graph: A flexible and reliable reasoning framework for large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 2025.

Costas Mavromatis and George Karypis. GNN-RAG: Graph neural retrieval for efficient large language model reasoning on knowledge graphs. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 16682–16699. Association for Computational Linguistics, 2025.

Shirui Pan, Linhao Luo, Yufei Wang, Chen Chen, Jiapu Wang, and Xindong Wu. Unifying large language models and knowledge graphs: A roadmap. *IEEE Transactions on Knowledge and Data Engineering*, 2024.

Shuofei Qiao, Yixin Ou, Ningyu Zhang, Xiang Chen, Yunzhi Yao, Shumin Deng, Chuanqi Tan, Fei Huang, and Huajun Chen. Reasoning with language model prompting: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5368–5393. Association for Computational Linguistics, 2023.

Yuxin Shi, Han Fu, Zhuo Li, Chenghao Liu, Xiaoxue Ren, and Jianling Sun. Plan-Answer-Refine-on-Graph: Structured planning and self-refinement for large language model reasoning on knowledge graphs. In *International Conference on Learning Representations*, 2026.

Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. MPNet: Masked and permuted pre-training for language understanding. In *Advances in Neural Information Processing Systems*, volume 33, pp. 16857–16867, 2020.

Yurun Song, Xiangqing Shen, Jianfei Yu, and Rui Xia. Interactive semantic parsing with reinforcement learning for knowledge graph reasoning. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 20464–20484. Association for Computational Linguistics, 2026.

Jia Ao Sun, Hao Yu, Fabrizio Gotti, Fengran Mo, Yihong Wu, Yuchen Hui, Zhan Su, Lingfeng Xiao, and Jian-Yun Nie. Search-on-graph: Iterative informed navigation for large language model reasoning on knowledge graphs. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*. ACM, 2026.

Jiashuo Sun, Chengjin Xu, Lumingyuan Tang, Saizhuo Wang, Chen Lin, Yeyun Gong, Lionel M. Ni, Heung-Yeung Shum, and Jian Guo. Think-on-graph: Deep and responsible reasoning of large language model on knowledge graph. In *International Conference on Learning Representations*, 2024a.

Lei Sun, Zhengwei Tao, Youdi Li, and Hiroshi Arakawa. ODA: Observation-driven agent for integrating LLMs and knowledge graphs. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 7417–7431. Association for Computational Linguistics, 2024b.

Alon Talmor and Jonathan Berant. The web as a knowledge-base for answering complex questions. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pp. 641–651. Association for Computational Linguistics, 2018.

Song Wang, Junhong Lin, Xiaojie Guo, Julian Shun, Jundong Li, and Yada Zhu. Reasoning of large language models over knowledge graphs with super-relations. In *International Conference on Learning Representations*, 2025.

Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. MiniLM: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. In *Advances in Neural Information Processing Systems*, volume 33, pp. 5776–5788, 2020.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023.

Yingxu Wang, Shiqi Fan, Mengzhu Wang, Siyang Gao, Chao Wang, and Nan Yin. DAMR: Efficient and adaptive context-aware knowledge graph question answering with LLM-guided MCTS. In *International Conference on Learning Representations*, 2026.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837, 2022.

Dawei Xiang, Kexin Chu, Wenyan Xu, Wenhui Zhang, and Wei Zhang. LLM-as-scheduler: Agentic workflow dynamic scheduling. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12752–12763. Association for Computational Linguistics, 2026.

Shitao Xiao, Zheng Liu, Peitian Zhang, Niklas Muennighoff, Defu Lian, and Jian-Yun Nie. C-Pack: Packed resources for general chinese embeddings. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 641–649. ACM, 2024.

Shuwen Xu, Yao Xu, Jiaxiang Liu, Chenhao Yuan, Wenshuo Peng, Jun Zhao, and Kang Liu. GraphWalker: Agentic knowledge graph question answering via synthetic trajectory curriculum. In *Conference on Language Modeling*, 2026.

Shiqi Yan, Yubo Chen, Ruiqi Zhou, Zhengxi Yao, Shuai Chen, Tianyi Zhang, Shijie Zhang, Wei-Qiang Zhang, Yongfeng Huang, Haixin Duan, and Yunqi Zhang. Explore-on-Graph: Incentivizing autonomous exploration of large language models on knowledge graphs with path-refined reward modeling. In *International Conference on Learning Representations*, 2026.

Linyao Yang, Hongyang Chen, Zhao Li, Xiao Ding, and Xindong Wu. Give us the facts: Enhancing large language models with knowledge graphs for fact-aware language modeling. *IEEE Transactions on Knowledge and Data Engineering*, 2024.

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

Xi Ye, Semih Yavuz, Kazuma Hashimoto, Yingbo Zhou, and Caiming Xiong. RNG-KBQA: Generation augmented iterative ranking for knowledge base question answering. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6032–6043. Association for Computational Linguistics, 2022.

Wen-tau Yih, Matthew Richardson, Christopher Meek, Ming-Wei Chang, and Jina Suh. The value of semantic parse labeling for knowledge base question answering. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 201–206. Association for Computational Linguistics, 2016.

Guibin Zhang, Haiyang Yu, Kaiming Yang, Bingli Wu, Fei Huang, Yongbin Li, and Shuicheng Yan. EvoRoute: Experience-driven self-routing LLM agent systems. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 38213–38225. Association for Computational Linguistics, 2026.

Wenxin Zhao, Jiachuan Wang, Yongqi Zhang, Shuangyin Li, Cheng Deng, Jun Wang, and Lei Chen. VoG: Enhancing LLM reasoning through stepwise verification on knowledge graphs. In *International Conference on Learning Representations*, 2026a.

Yang Zhao, Chengxiao Dai, Wei Zhuo, Yue Xiu, and Dusit Niyato. CLAUSE: Agentic neuro-symbolic knowledge graph reasoning via dynamic learnable context engineering. In *International Conference on Learning Representations*, 2026b.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 62138–62160. PMLR, 2024a.

Tong Zhou, Yubo Chen, Kang Liu, and Jun Zhao. CogMG: Collaborative augmentation between large language model and knowledge graph. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pp. 365–373. Association for Computational Linguistics, 2024b.

A REPRODUCTION DETAILS

Table 4: Default relation policy settings used in the experiments.

Symbol / Descriptions	Value	Role
γ_0 / uncertainty threshold	0.35	Base escalation threshold
$T_{\max}$ / max actions	15	Action budget per query
K / candidate top-K	10	Per-ranker and working shortlist
τ	0.1	LM ranker softmax temperature
k_0	60	RRF rank constant
w_H / entropy weight	0.5	JS weight is $1 - w_H = 0.5$

A.1 METHOD PSEUDOCODE

Algorithm 1 summarizes the inference procedure in Section 4, using the defaults in Table 4 and the notation defined in the main text. RRF uses ranks in the filtered lists L_t^v, as specified in Equation (9). Routing distributions are computed once per decision. New probe evidence updates admissibility without recomputing F_t.

Every action is selected from $\widehat{\mathcal{A}}(q_t, p_t, I_t, M_i)$ in Equation (4). This includes retrieval, ranking, probing, constraint operations, and the end tool. Filtering actions consumes neither an action nor a retry. An unavailable required action invokes retry only within the remaining budgets.

Action budget accounting. Each LLM, KG tool, Constraint tool, or End tool invocation counts as one action. A KG tool call counts once regardless of the number of entities, relations, or triples it returns. One LLM call that scores the candidate relation set also counts as one action. It is not counted again as a separate ranking action. The accompanying local LM scoring does not increase any action. It is an internal computation of the ranking procedure.

Local normalization, uncertainty computation, rank fusion, and memory operations add no actions. Selecting a relation from existing scores also adds no action. Expanding along that relation requires a KG tool call and counts as one action. Retrieving relations, scoring them with one LLM call, and expanding along the selected relation are considered as three actions. An additional LLM call for action selection is counted separately. Lookahead counts each tool invocation that it issues. Wrapping several such invocations in one procedure does not combine them into a single action.

Budget enforcement and retries. The counter i starts at zero before planning. Before every LLM or tool invocation, including each call within probing and End, the runtime requires $i < T_{\max}$ and increments i once, regardless of success. If no budget remains for a required call, the query returns $\emptyset$ without a retry or partial answer. No completion budget is reserved.

Restoring a checkpoint is a local state operation and adds no action. It still consumes one of the permitted checkpoint retries. Calls made during the restarted attempt are counted normally. Planner calls leave failure memory unchanged. They precede validation against subgoal requirements.

Shared execution protocol. Each executor action follows Equation (4). The executor captures the evaluated action's key before execution. The checks in Appendix A.5 then produce o_i. Equation (11) updates memory at the captured key. This update precedes any change to p_t or restoration of a checkpoint. Only a verified violation of an applicable requirement updates failure memory. Otherwise $M_i = M_{i-1}$. Local validation of available evidence adds no action. Additional KG or Constraint tool calls needed for validation follow the same filtering and budget rules.

A subgoal completes only after all operations in I_t finish and its grounded output meets every requirement. Each query permits at most two checkpoint retries across all subgoals. A failed Policy 1 action records any verified failure before retry and requires Policy 2 at the first relation decision of the new attempt. This requirement is consumed at that decision, not applied to every later decision.

Baseline alignment. The implementations of training-free KG baselines use the same rules at their LLM and tool invocation boundaries. These baselines are tried to keep the same action accounting and budget checks, but equal action budgets do not strictly ensure equal computational cost. Therefore, token usage and end-to-end latency are important metrics to show the cost.

Algorithm 1 AoG. Uncertainty-Aware Relation Adaptive Policy with Memory-to-Action

Require: Query q, KG $\mathcal{G}$, action budget $T_{\max}$, router settings in Table 4
Ensure: Grounded answer, or $\emptyset$ if execution cannot be completed within budget
Budget rule. Count every LLM/tool call; if $i \geq T_{\max}$ before any call, return $\emptyset$.
Execution rule. Executor actions also follow Eq. (4).
Initialize $i \leftarrow 0$, $M_0 \leftarrow \emptyset$, $\hat{p}_0$, and the checkpoint retry count to zero
Obtain $\Pi \leftarrow \mathrm{Decompose}(q)$ under the shared budget. Count each planner attempt.
if no valid plan is obtained **then**
    return $\emptyset$
for $t = 1, \ldots, n$ **do**
    $p_t \leftarrow \hat{p}_{t-1}$ and resolve the inputs specified by I_t
    while Π_t is unfinished **do**
        if $i \geq T_{\max}$ **then**
            return $\emptyset$
        Clear any previously selected action
        if the next operation requires relation selection **then**
            Retrieve R_t through a budgeted KG tool call
            Obtain w_t^{llm} through budgeted LLM calls. Compute w_t^{lm} locally.
            Form L_t^v for each ranker using Eq. (5)
            $L_t^{\mathrm{u}} \leftarrow L_t^{\mathrm{llm}} \cup L_t^{\mathrm{lm}}$
            if $L_t^{\mathrm{u}} \neq \emptyset$ **then**
                Compute $P_t^{\mathrm{u}}, Q_t^{\mathrm{u}}, \rho_t, \gamma_t$ by Eqs. (6)–(8)
                if $\rho_t < \gamma_t$, LLM scores are available, $L_t^{\mathrm{llm}} \neq \emptyset$, and no retry requires Policy 2 **then**
                    Select $a(r)$ with $r = \arg\max_{r' \in L_t^{\mathrm{llm}}} w_t^{\mathrm{llm}}(r')$ ▷ Policy 1
                else ▷ Policy 2
                    Consume any pending retry requirement to use Policy 2
                    Compute F_t by Eq. (9) using ranks in L_t^{llm} and L_t^{lm}
                    $\hat{L}_t^{\mathrm{u}} \leftarrow \mathrm{TopK}_{F_t}(L_t^{\mathrm{u}})$ ▷ Keep F_t fixed during probing
                    while $\hat{L}_t^{\mathrm{u}}$ has an uninspected entry and $i < T_{\max}$ **do**
                        $r \leftarrow$ highest-RRF uninspected entry of $\hat{L}_t^{\mathrm{u}}$
                        Probe r and validate the evidence using the shared protocol, then mark r inspected
                        Retain shortlist entries with $a(r) \in \widehat{\mathcal{A}}(q_t, p_t, I_t, M_i)$
                        Refill with admissible entries from L_t^{u} by RRF, up to K if budget permits
                    if $i \geq T_{\max}$ **then**
                        return $\emptyset$
                    if $\hat{L}_t^{\mathrm{u}} \neq \emptyset$ **then**
                        Select $a(r)$ with $r = \arg\max_{r' \in \hat{L}_t^{\mathrm{u}}} F_t(r')$
                        Cache $\hat{L}_t^{\mathrm{u}} \setminus \{r\}$ in RRF order with the current decision context
        else
            Select $a \in \widehat{\mathcal{A}}(q_t, p_t, I_t, M_i)$ for the required operation in I_t
        while an action a is selected and $a \notin \widehat{\mathcal{A}}(q_t, p_t, I_t, M_i)$ **do**
            Exclude a and select the next candidate for the same operation
        if an action a has been selected **then**
            Execute a, validate its result, and update M_i using the shared protocol
            if execution succeeds without a verified applicable violation **then**
                Update p_t with the observed result
                continue
        if no admissible continuation remains after a retry **then**
            return $\emptyset$
        if $i \geq T_{\max}$ or two retries have already been used **then**
            return $\emptyset$
        Restore $p_t \leftarrow \hat{p}_{t-1}$ and consume one retry for the unfinished subgoal
        Retain M_i, i, and attempt histories, then rebuild candidates for the restored context
        Require Policy 2 at the next relation decision when retry follows a failed Policy 1 action
    Bind the completed output to dependent subgoals and save $\hat{p}_t$
if $i \geq T_{\max}$ **then**
    return $\emptyset$
Execute `End` using the shared protocol
return its grounded answer on successful completion, otherwise $\emptyset$

A.2 Experimental Environment and Repeated Runs

Unless explicitly noted otherwise, each experiment run by this work, including the main evaluation, ablations and hyperparameter sensitivity analysis, uses `run=3` with random seeds $\{1, 2, 3\}$. The software environment uses Python 3.12. Experiments run on one NVIDIA RTX A6000 GPU with 48 GB of memory. The LM ranker is loaded on this GPU, query planner and LLM ranker decisions are made through the configured external LLM API.

A.3 Prompt Templates and Schema

The query planner first decides whether the question requires multiple dependent or independent reasoning branches. The core decomposition instruction is:

Prompt Template: Query Decomposition

```
Decompose the query when the answer requires a dependent
intermediate result, independent branches, threshold filtering, an
aggregation, or multiple named entities with distinct reasoning roles.
For each chain, write one logical 1-hop sub-question per step. A later
step can use depends_on to reference the step whose EntitySet it consumes.
Keep CVT physical hops inside a semantic subgoal. Do not emit entity
IDs or entity names as traversal values when gold topic references are
available. Use topic_ref and let the runtime bind the EntitySet.
Every operation must be one of project, filter, or aggregate. Use filter
only for an explicit literal constraint, and use aggregate only for count,
argmax, or argmin. Count is terminal and has no property field.
```

The query planner output follows the given schema. The chain-level fields bind the input topic, while the step-level `depends_on` field binds the output of a preceding semantic step to the next step's input. The planner does not emit generic `input` or `output` fields. The runtime derives these bindings from `start` and `depends_on` and carries the resulting EntitySet.

Output Schema: Query Planner

```
{
  "strategy": "direct" | "decompose",
  "entities": {
    "e1": {"mention": "<entity name exactly as it appears>"}
  },
  "chains": [
    {
      "entity_refs": ["<keys from entities>"],
      "start": {"topic_ref": "topic_1"},
      "steps": [
        {
          "id": "c1s1",
          "sub_question": "<one logical 1-hop sub-question>",
          "operation": {"type": "project"},
          "expected_output_type": "<common entity types>"
        },
        {
          "id": "c1s2",
          "sub_question": "<next logical 1-hop sub-question>",
          "depends_on": "c1s1",
          "operation": {
            "type": "project" | "filter" | "aggregate",
            "operator": "<select according to operation type>",
            "value": "<filter literal only>",
            "property": "<filter, argmax, or argmin property only>"
          },
          "expected_output_type": "<common entity types>"
        }
      ]
    }
  ],
  "final_expected_answer_type": "<according the core object in the given query>",
}
```

The operation bindings are intentionally narrow:

Operation Schema: Filter and Aggregation

```
filter:  {"type":"filter", "operator":"=|>|>=|<|<=",
          "value":"<literal copied from the question>",
          "property":"<concise semantic property>"}
argmax:  {"type":"aggregate", "operator":"argmax",
          "property":"<property to maximize>"}
argmin:  {"type":"aggregate", "operator":"argmin",
          "property":"<property to minimize>"}
count:   {"type":"aggregate", "operator":"count",
          "property":"<total number of matched property>"}
```

For a question with available gold topic references, every chain includes `entity_refs` and starts with a `topic_ref`. The actual topic EntitySet is supplied by the executor. Each semantic subgoal represents one logical relation even when its execution traverses a compound value type (CVT). The `count` operation is placed last, and its output is the cardinality of the current EntitySet rather than a relation property.

Executable requirements. The executor resolves the planner fields into the requirements in I_t before applying the checks in Appendix A.5. Table 5 specifies the inputs to each check. Type names and property descriptions must resolve to KG identifiers or supported literal types. Entity references resolve to the IDs bound by `topic_ref` or earlier subgoal outputs. An unresolved mapping leaves the affected check unknown rather than producing a semantic guess.

The schema illustrates projection, literal filtering, and aggregation. A join requirement additionally identifies the participant role and bound entity, or the branch outputs and shared variable to be matched. `depends_on` supplies an input dependency, not an equality constraint by itself. The comparison set for an extremum is the input EntitySet obtained by executing the preceding plan steps over the KG, not an entity set predicted by the planner.

Table 5: Executable requirements and tool-based checks for the four components of o_i.

Component	Requirement resolved from I_t	Evidence and programmatic check
Type	`expected_output_type` and the stage at which it applies	Check the returned structure or KG type evidence against the required type.
Value	Filter `property`, `operator`, and `value`	Retrieve the property values and compare normalized literals using the specified operator.
Join	Required entity IDs and participant roles, or branch outputs sharing a variable	Match the resolved bindings or compute the required join or intersection over entity IDs.
Extremum	`argmax` or `argmin`, the comparison property, and the plan-specified EntitySet	Compare the candidate's property value with values in the specified comparison set.

A.4 LLM Relation-Ranker Prompt and Output Schema

The LLM ranker receives the subquestion and the complete candidate list. Candidate relations are sorted and rendered with compact numeric indices. Repeated dotted prefixes are factored only for display, without losing the mapping to the complete relation. The full prompt is:

Prompt Template: LLM Relation Ranking

```
Please evaluate how relevant each knowledge graph relation is to
answering the given question.
Use only the question and the given relation list. Assign every relation
an independent continuous relevance weight. Equal relevance may share the same
weight. Do not normalize the weights into a distribution.
Predicates are grouped as `shared.prefix: index=suffix`. The complete predicate
for an item is `shared.prefix.suffix`. `(no-prefix)` items use suffix as-is.
Assign every numeric index exactly one independent relevance weight in [0, 1]
and use only numeric index strings as JSON keys. The weights are independent
and do not need to sum to 1.

Question: <question>
Indexed candidate predicates:
```

```
<complete indexed candidate list>

Reply ONLY with JSON matching the output schema below.
```

The expected ranker output uses the following schema:

Output Schema: LLM Relation Ranking

```
{
  "scores": {
    "0": <float in [0, 1]>,
    "1": <float in [0, 1]>,
    ...,
    "<n-1>": <float in [0, 1]>
  }
}
```

The response must contain every numeric index exactly once, must contain no unknown index, and must use finite numeric values in $[0, 1]$. The values are independent scores, not probabilities, so they are not required to sum to one. If the first response is invalid JSON or violates the schema, the ranker makes one repair attempt within the remaining action budget.

Before policy selection, Equation (5) filters each ranker's original top-K list to obtain L_t^{llm} and L_t^{lm}. A shared relation uses the same check result in both lists. This is the traversal-specific application of Equation (4). Verified failures from executed actions and probes have already updated memory through the shared execution protocol. Filtering reuses these records under matching contexts and inspection scopes. It makes no additional KG or LLM calls and retains candidates whose relevant checks are unknown. The retained entries keep their original scores and relative order. Their positions in the filtered lists define the ranks in Equation (9). Both rankers' scores are normalized once over the filtered union L_t^{u}, without calling either ranker again. Missing, negative, or non-finite LLM scores contribute zero. If all retained LLM scores are zero, the distribution is uniform over this union. LM similarities are converted with a softmax temperature $\tau > 0$. Normalized entropy and JSD are zero for a singleton union. JSD is computed from the normalized distributions and clipped to $[0, 1]$. An empty union bypasses routing. It triggers a retry if the action and retry budgets permit. Otherwise, execution terminates. Both ρ_t and γ_t use the filtered union. For a nonempty union, Policy 1 selects directly from L_t^{llm} when its routing conditions hold and no retry requires Policy 2. All other nonempty cases use Policy 2.

A.5 Tool-Based Constraint Validation

Validation is an executor operation, not a separate LLM self-verification stage. KG tools supply factual evidence, while Constraint tools and local programmatic checks evaluate the applicable requirements in I_t. The interface is

$$o_i = \mathrm{Validate}\big(\mathrm{result}(a_i), I_t, p_t\big),$$

where p_t is the captured pre-action context and $\mathrm{result}(a_i)$ denotes the observed execution evidence, including any tool error and coverage information. The function returns the four components in Equation (2) without an LLM correctness judgment.

Each applicable check evaluates only the requirements assigned to the current action and output stage. An intermediate entity need not satisfy the final answer type, and gathering candidates does not yet require selecting an extremum. Joint requirements are checked for the same candidate and consistent participant bindings. The components have the following meanings:

- `passed`: error-free evidence establishes that all applicable requirements within the component hold.
- `failed`: error-free evidence establishes a violation of at least one applicable requirement within the component.
- `unknown`: the component is applicable, but unresolved requirements, insufficient evidence, or tool errors prevent a conclusion.
- `not_applicable`: the component has no requirement to check for the current action.

A condition is marked `failed` only when the evidence shows that it is not satisfied. If the condition requires at least one match, finding none in a limited sample leaves the result `unknown`. Likewise, a join is considered empty only when no match is found after checking all relevant entity bindings or running an exact join query. EExtremum checks use the input EntitySet obtained by executing the plan over the KG. Finding a higher value in this set disproves a proposed maximum, while finding a lower value disproves a proposed minimum. Confirming that a candidate is the maximum or minimum requires checking the entire set or running an exact aggregate query over it. These checks establish compliance with I_t, not the correctness of the planner's interpretation of the original question.

A.6 Memory Records and Checkpoint State

Equation (2) pairs $\mathrm{key}(q_t, p_t, a_i)$ with the four validation outcomes. Memory stores these outcomes by key. Filtering ranges over the resulting key. The prefix p_t contains existing triplets on valid and active paths. The key is captured before execution and is not recomputed from the resulting state or a restored checkpoint. Before textual concatenation, key construction preserves the full ordered prefix, resolves temporary entity variables to IDs, and normalizes entity-set ordering. Serialization retains the field boundaries so that $\mathrm{inverse}_{\mathrm{key}}$ in Equation (3) can recover the stored context. The action identifies the tool or operation, resolved inputs, parameters, and applicable requirements, including traversal direction where relevant. Thus the conditions in I_t relevant to the evaluated action are part of its identity.

Equation (4) checks these contexts before every tool call, not only during relation selection. For example, a failed constraint-tool action can exclude the same call with matching inputs and requirements. It does not invalidate the tool for other inputs. Before probing a candidate, the executor also captures the key of the evaluated candidate action. When a probe confirms that a candidate action violates a requirement, the failure is recorded under the candidate action's key, not the probe tool's key. This allows the system to exclude the same candidate action when the same context recurs.

The four components follow Appendix A.5. Equation (11) inserts or updates only the matching key when an applicable component has a verified failure. An existing outcome at that key is replaced by the new o_i, and all other keys remain unchanged. The same rule applies to Policy 1 execution and Policy 2 probes. Passed, unknown, or inapplicable checks alone do not update failure memory.

The following illustrative record uses placeholders for the prefix and entity IDs. Here I_t requires a film released after 2000. Suppose the candidate has verified film-type evidence and a complete, error-free property query returns only a 1995 release date. The type check passes, while comparing the date with the required threshold produces a failed value check. No join or extremum requirement applies. Supporting tool observations retain the checked values and coverage information. The memory item summarizes their validation, with its serialized key shown as decoded fields for readability.

Memory Case: Verified Literal-Property Failure

```
{
  "key": {
    "q_t": "Which films were released after 2000?",
    "p_t": "<pre-action prefix with valid and active triplets>",
    "a_i": {
      "tool": "get_entity_by_constraint",
      "relation": "film.film.release_date",
      "direction": "out",
      "source_entity_ids": ["<film-id>"],
      "operator": ">",
      "value": "2000",
      "requirement": "film released after 2000"
    }
  },
  "o_i": {
    "type": "passed",
    "value": "failed",
    "join": "not_applicable",
    "extremum": "not_applicable"
  }
}
```

The failed value component triggers the keyed update in Equation (11). In contrast, a timeout of the release-date query leaves the value check unknown. With the same independently verified type evidence, the outcome is (`passed`, `unknown`, `not_applicable`, `not_applicable`), which does not update failure memory.

During action filtering, the executor evaluates b in Equation (3) for each candidate action and stored failure record. Its output is computed during filtering rather than stored in the record. It returns 1 only when the candidate's pre-action context matches the record and the recorded failure still applies to the current requirements and inspection scope. A failure involving one film or condition does not automatically apply to another. If later error-free evidence disproves the recorded violation, the function returns 0 for that candidate and record. Existing triplets in p_t provide context but do not by themselves prove a validation failure.

A retry restores p_t from the fixed subgoal-entry checkpoint $\hat{p}_{t-1}$, with $\hat{p}_0$ holding the initial state. Only the unfinished subgoal restarts. Memory, attempt histories, and the cumulative action index are retained. For example, a failure in `c1s2` restores the checkpoint after `c1s1`. Execution resumes at `c1s2`. Each query permits at most two restorations (`max_retries=2`). Retries after Policy 1 fails count toward this limit and use Policy 2 at the first subsequent relation decision when candidates remain. All attempts share $T_{\max}$. Restoring a checkpoint adds no action. Calls issued after restoration consume the remaining action budget. Execution returns $\emptyset$ if a required call would exceed $T_{\max}$. It also returns $\emptyset$ if another restoration would exceed the retry limit. No admissible continuation after retry also results in $\emptyset$. The executor does not return a partial answer. Completion binds the grounded output to dependent subgoals and saves $\hat{p}_t$ for the next subgoal.

A.7 Lookahead Implementations

Policy 2 initializes its working shortlist as $\hat{L}_t^{\mathrm{u}} = \mathrm{TopK}_{F_t}(L_t^{\mathrm{u}})$ from the already filtered union. The RRF scores use ranks in the filtered lists L_t^v according to Equation (9) and remain fixed during probing for the current decision. Programmatic checks are reused while context and evidence remain unchanged. After a failed Policy 1 action triggers a retry, candidate lists are reconstructed for the restored subgoal-entry state before Policy 2 resumes. Retained observations are reused only under matching contexts.

New observations are collected through neighbor sampling, next-relation recording, and second-hop type checks. Policy 2 probes the shortlist in descending RRF order rather than stopping at the first surviving candidate. Each probe records sampled neighbors, literal availability, a preferred traversal direction, possible compound value type (CVT) continuations, and a bounded set of next-hop predicates. It also records the inspection scope and coverage status. We first collect an initial neighbor sample for each shortlisted candidate. Additional neighbors are sampled only when the candidate ranks highly under RRF and the initial evidence is insufficient to assess it.

Probe evidence is processed using the validation and memory-update rules in Appendices A.5 and A.6. Verified violations update M_i through Equation (11). The executor then filters the shortlist using the updated memory, retaining only entries whose traversal actions belong to $\widehat{\mathcal{A}}(q_t, p_t, I_t, M_i)$.

Within the remaining action budget, vacancies are filled in RRF order. Each replenished candidate is filtered using the updated memory before probing. Final relation selection follows Equation (10), while retry and termination follow Appendix A.6. No actions are reserved for subsequent traversal, constraint operations, or End. If probing exhausts $T_{\max}$, the query returns $\emptyset$ without further tool calls, following the budget-enforcement rule in Appendix A.1. Completing shortlist inspection therefore does not guarantee that the remaining execution can finish within budget.

A.8 Datasets

CWQ contains complex questions constructed from WebQuestions through composition, conjunction, comparative, and superlative operations (Talmor & Berant, 2018). Its official split contains 27,639 training, 3,519 validation (val), and 3,531 test questions. The main comparison in Tables 1 and 6 uses all 3,531 test questions.

WebQSP provides natural-language questions paired with semantic parses over Freebase (Yih et al., 2016), with 3,098 training and 1,639 test questions. It primarily tests one or two-hop reasoning with

additional constraints. The available answer-based evaluation covers all 1,628 test questions with nonempty answer annotations.

GrailQA contains 64,331 questions, split into 44,337 training, 6,763 development, and 13,231 test questions (Gu et al., 2021). It evaluates i.i.d., compositional, and zero-shot generalization, including unseen combinations of known schema elements and unseen schema elements. The complete test split remains separate because its answers and logical forms are hidden. The reported comparison uses 5,609 questions from the public development split.

A.9 Gold Edge Evaluation

Gold Edge Hits@1/5/10 measures the coverage of annotated directed relations in the top-k candidate lists, and **Gold Edge MRR@10** measures how highly those relations are ranked. Gold paths are aligned monotonically to the recorded relation decisions, without reusing a decision within a path. For gold edge e of question q, let $s_{q,e}$ be its rank in the aligned candidate list, with $s_{q,e} = \infty$ for a missing or unreached edge. For the m_q annotated edges being evaluated, we compute

$$\text{EdgeHits@}\,k(q) = \frac{1}{m_q} \sum_{e=1}^{m_q} \mathbf{1}[s_{q,e} \le k], \quad k \in \{1, 5, 10\}, \tag{12}$$

$$\text{EdgeMRR@10}(q) = \frac{1}{m_q} \sum_{e=1}^{m_q} \frac{\mathbf{1}[s_{q,e} \le 10]}{s_{q,e}}. \tag{13}$$

Missing ranks contribute zero. These question-level scores are macro-averaged over questions with evaluable gold paths. Alternative WebQSP parses use the best-aligned valid path, whereas jointly required paths in compositional queries are all scored. Strict matching requires both predicate identity and traversal direction. Any inverse-aware variant additionally accepts schema-certified inverse predicates with reversed direction and is reported separately. Edge Hits and MRR both lie in $[0, 1]$ before scaling. In the tables, their means and standard deviations are multiplied by 100 to report percentages, and higher is better for both. IO Prompt, CoT, and SC have no relation ranking, so their edge metrics are not applicable. These metrics diagnose relation retrieval, not the correctness of the final answer.

B Baselines

We divide the baselines into three categories: LLM-only prompting, fine-tuned KGQA systems, and training-free KG agents.

B.1 LLM-only methods

- **IO Prompt** ((Brown et al., 2020)): It performs few-shot prompting by input-output pair examples without any reasoning steps.
- **Chain-of-Thought (CoT)** ((Wei et al., 2022)): It prompts LLMs to produce step-by-step reasoning steps to obtain a final answer.
- **Self-Consistency (SC)** ((Wang et al., 2023)): This approach samples multiple reasoning chains from the same CoT prompt and aggregates their final answers by majority voting.

B.2 Finetuned methods

- **RoG** ((Luo et al., 2024)): It fine-tunes an LLM to generate relation-path plans, retrieves matching KG paths, and uses the retrieved evidence to produce a final answer.
- **GNN-RAG** ((Mavromatis & Karypis, 2025)): It trains a query-conditioned graph neural retriever to identify answer candidates and extracts paths connecting them to topic entities.
- **Plan-Answer-Refine-on-Graph (PARoG)** ((Shi et al., 2026)): It trains a structured planner to decompose non-linear questions and uses retrieved graph evidence to refine tentative subanswers.
- **DAMR** ((Wang et al., 2026)): It combines Monte Carlo tree search with a context-aware path scorer updated through pseudo-path feedback during inference.

- **Explore-on-Graph (EoG)** ((Yan et al., 2026)): It performs supervised fine-tuning on long graph-reasoning trajectories and further optimizes exploration using outcome and path-refined rewards.

- **ISP-KGR** ((Song et al., 2026)): It trains an interactive semantic parser through reinforcement learning to construct and execute logical clauses over the KG.

- **GraphWalker** ((Xu et al., 2026)): It initializes graph-tool reasoning with a synthetic trajectory curriculum and refines this capability through reinforcement learning.

- **EffiQA** ((Dong et al., 2025)): It pairs an LLM planner with a smaller graph executor, using fine-grained entity types to prune relations and entities and reflection to replan failed or oversized searches.

B.3 Training-free methods

- **KG-Agent-API** ((Jiang et al., 2025)): This adaptation replaces KG-Agent's fine-tuned controller with a general-purpose LLM API while retaining its planner, KG tools, execution feedback, and memory. It is not a reproduction of the original learned checkpoint and must not load task-specific KG-Agent weights, supervised trajectories, or task-trained auxiliary checkpoints.

- **Plan-on-Graph (PoG)** ((Chen et al., 2024b)): It decomposes complex questions into structured subgoals to guide KG exploration and uses reasoning memory, reflection, and backtracking to revise the plan when retrieved evidence is insufficient.

- **Debate-on-Graph (DoG)** ((Ma et al., 2025)): It uses multiple LLM agents to propose, inspect, and critique graph-grounded reasoning paths, simplifying unresolved questions and reconciling evidence through debate.

- **Search-on-Graph (SoG)** ((Sun et al., 2026)): It prompts a tool-using LLM to iteratively inspect local graph neighborhoods and select promising continuations until sufficient evidence is collected or the search budget is exhausted.

- **Verify-on-Graph (VoG)** ((Zhao et al., 2026a)): It verifies predicted observations against retrieved KG triples and uses this evidence to correct unsupported assumptions and update its reasoning plan.

C Extended Results

C.1 Additional Metrics

Table 6 complements Table 1 with separate input and output token counts and mean latency. It uses the same methods and groups. Prompting and training-free results are means over three runs. Fine-tuned entries retain the original papers' aggregation. Within each backbone-specific agentic block, the best and second-best values are shown in **bold** and underlined, respectively. Dataset coverage is specified in Appendix A.8. Output token means are derived by subtracting the input token means in this table from the total token means in Table 1. Missing values remain unreported when this difference cannot be computed. Year-only citations identify the baseline publications. The KG-Agent-API citation refers to the original KG-Agent method rather than a separately published API adaptation.

Table 6: Inference cost comparison on KGQA datasets. Input and output token counts are in thousands per question. Latency is in seconds per question. Methods are grouped by evaluation family, with the corresponding backbone shown as a sub-row before each backbone-specific block. "-" denotes an unreported result. AoG rows are shaded for readability.

Method	CWQ			WebQSP			GrailQA		
	In. ↓	Out. ↓	Latency ↓	In. ↓	Out. ↓	Latency ↓	In. ↓	Out. ↓	Latency ↓
LLM-Prompting Methods									
`deepseek-v4-flash`									
IO Prompt (2020)	0.827	1.030	16.022	0.510	0.218	5.842	0.404	1.469	16.962
CoT (2022)	0.121	0.262	4.272	0.124	0.077	2.116	0.122	0.180	3.377
SC (2023)	1.923	0.663	10.283	1.162	0.170	5.842	0.811	1.197	8.236
`gpt-5.6-luna`									
IO Prompt (2020)	0.157	0.583	12.750	0.144	0.121	6.203	0.155	0.399	9.232
CoT (2022)	0.175	0.825	17.162	0.122	0.187	5.687	0.126	0.521	14.026
SC (2023)	0.833	4.074	48.743	0.719	0.950	14.617	0.823	3.108	34.154
Finetuned Methods									
RoG (2024)	-	-	-	-	-	-	-	-	-
GNN-RAG (2025)	-	-	-	-	-	-	-	-	-
EffiQA (2025)	-	-	-	-	-	-	-	-	-
PARoG (2026)	7.1107	0.2883	-	5.0123	0.2417	-	3.1809	0.1771	-
DAMR (2026)	-	-	-	-	-	-	-	-	-
EoG (2026)	-	-	-	-	-	-	-	-	-
ISP-KGR (2026)	-	-	-	-	-	-	-	-	-
GraphWalker (2026)	-	-	-	-	-	-	-	-	-
Training-free Agentic Methods									
`deepseek-v4-flash`									
KG-Agent-API (2025)	47.124	25.888	39.741	28.523	4.226	27.045	26.374	10.899	125.342
PoG (2024b)	29.371	5.310	243.240	9.273	2.028	44.556	8.760	4.677	95.957
DoG (2025)	**8.302**	4.300	**22.441**	10.873	**1.078**	16.858	**4.188**	1.574	35.645
SoG (2026)	38.108	2.033	45.581	37.148	2.385	35.253	22.867	1.873	54.337
VoG (2026a)	15.273	**2.002**	123.235	**5.273**	4.459	52.381	11.324	2.911	82.382
AoG (Ours)	12.534	2.190	41.356	6.283	1.705	**12.711**	6.305	**0.291**	**11.393**
`gpt-5.6-luna`									
KG-Agent-API (2025)	48.220	9.930	229.349	27.964	2.703	79.151	32.642	1.601	93.246
PoG (2024b)	30.310	3.220	211.052	17.366	1.218	128.709	23.134	3.348	148.284
DoG (2025)	**9.220**	8.260	63.352	8.764	0.974	30.962	**5.503**	0.820	33.327
SoG (2026)	41.050	**1.200**	41.628	10.806	**0.545**	28.570	10.910	**0.470**	30.374
VoG (2026a)	9.398	2.180	118.821	7.827	2.535	50.824	12.362	4.012	85.861
AoG (Ours)	10.812	1.328	**37.194**	**6.0678**	0.6832	**21.610**	5.724	0.690	**24.690**

C.2 Policy 2 Usage and Random Routing

We measure Policy 2 usage in historical AoG runs with `gpt-5.6-luna` as the backbone and BGE-M3 as the local ranker. For each run, the rate is the number of recorded Policy 2 decisions divided by the total number of recorded Policy 1 and Policy 2 decisions. The count includes retry decisions and escalations caused by topology checks. The rates are $(51.16 \pm 0.85)\%$ on CWQ, $(77.33 \pm 0.28)\%$ on WebQSP, and $(40.12 \pm 1.49)\%$ on GrailQA. Each value reports the mean and sample standard deviation over three runs. The standard deviations are in percentage points.

The historical CWQ configuration uses $w_H = 0.75$. The WebQSP and GrailQA configurations use $w_H = 0.50$. These rates are historical references rather than measurements under a single shared default configuration.

These means define the reference probabilities for the random routing baseline. We set $p_{\mathrm{CWQ}} = 0.5116$, $p_{\mathrm{WebQSP}} = 0.7733$, and $p_{\mathrm{GrailQA}} = 0.4012$. At each decision eligible for random routing, *w/o Uncertainty Routing* selects Policy 2 with probability p and Policy 1 with probability $1 - p$. The probability is fixed for each dataset and does not depend on the current uncertainty or answer correctness. *w/o Both* uses the same probabilities and also disables failure exclusion. Candidate ranking, mandatory escalation rules, and action accounting remain unchanged. The realized Policy 2 usage rate can differ from p because mandatory escalations remain active. These reference probabilities do not guarantee matched token use or latency.

C.3 Uncertainty Signal Ablations

Tables 7 and 8 compare the combined signal with entropy-only and JSD-only uncertainty. Values are mean ± standard deviation over three runs under the protocol in Section 5.3. Within each table, the best and second-best means in each column are shown in **bold** and underlined, respectively. We repeat the full method for comparison.

Both single-signal variants reduce Hits@1 and F1 and increase tokens on all three datasets. JSD-only routing also increases mean latency throughout. Entropy-only routing is 5.9% faster on CWQ and 9.0% faster on WebQSP, but sacrifices answer quality and token efficiency. Entropy captures ambiguity within the LLM ranking, while JSD captures disagreement between rankers. The results favor combining these signals under the evaluated configuration.

Table 7: Answer-quality ablations of the uncertainty signal. All entries are percentages. Higher is better.

	CWQ		WebQSP		GrailQA	
Variant	**Hits@1**	**F1**	**Hits@1**	**F1**	**Hits@1**	**F1**
AoG	$\mathbf{80.42 \pm 0.29}$	$\mathbf{70.96 \pm 0.52}$	$\mathbf{86.69 \pm 0.45}$	$\mathbf{77.16 \pm 0.27}$	$\mathbf{85.43 \pm 0.13}$	$\mathbf{79.02 \pm 0.19}$
Entropy-only Uncertainty	71.87 ± 0.93	62.18 ± 1.12	75.35 ± 0.63	64.25 ± 0.54	70.49 ± 0.61	62.34 ± 1.03
JSD-only Uncertainty	73.98 ± 1.18	63.04 ± 1.42	77.76 ± 1.37	63.36 ± 1.58	71.97 ± 1.29	64.86 ± 1.76

Table 8: Efficiency ablations of the uncertainty signal. Tokens are in thousands per question, and latency is in seconds per question. Lower is better.

	CWQ		WebQSP		GrailQA	
Variant	**Tokens** (k)	**Latency** (s)	**Tokens** (k)	**Latency** (s)	**Tokens** (k)	**Latency** (s)
AoG	$\mathbf{12.14 \pm 0.33}$	$\underline{37.19 \pm 1.46}$	$\mathbf{6.75 \pm 0.01}$	$\underline{21.61 \pm 1.38}$	$\mathbf{6.41 \pm 0.07}$	$\mathbf{24.69 \pm 1.87}$
Entropy-only Uncertainty	16.72 ± 0.76	34.99 ± 0.88	8.26 ± 0.63	19.67 ± 1.47	6.92 ± 0.09	30.64 ± 2.59
JSD-only Uncertainty	20.50 ± 0.56	38.64 ± 2.47	7.20 ± 0.34	23.26 ± 2.69	7.26 ± 0.62	34.64 ± 2.21

C.4 Action Limit and Candidate Count Sensitivity

Tables 9 and 10 extend the one-factor analysis in Section 5.4 to the action budget $T_{\max}$ and candidate number K. Each sweep changes only the indicated parameter while retaining $\gamma_0 = 0.35$, $w_H = 0.50$, and all other default settings.

Table 9: Sensitivity to the action budget $T_{\max}$. Hits@1 and F1 are percentages. Token cost is in thousands of tokens per question, and latency is in seconds per question. Higher is better for Hits@1 and F1, while lower is better for tokens and latency. The default setting is shaded.

	CWQ				WebQSP				GrailQA			
$T_{\max}$	Hits@1 ↑	F1 ↑	Tokens ↓	Latency ↓	Hits@1 ↑	F1 ↑	Tokens ↓	Latency ↓	Hits@1 ↑	F1 ↑	Tokens ↓	Latency ↓
10	78.94	68.11	9.365	31.272	84.62	74.31	4.266	17.634	79.25	72.46	4.834	18.714
15	80.42	70.96	12.140	37.194	86.69	77.16	6.751	21.610	85.43	79.02	6.414	24.690
20	82.87	72.33	18.907	58.296	87.15	77.53	8.805	26.658	86.10	79.21	9.971	30.738

Reducing the budget from 15 to 10 actions lowers token use and latency, but also produces substantial Hits@1 and F1 degradation on all three datasets. Increasing the budget from 15 to 20 yields only small Hits@1 gains and a little F1 improvement, while increasing both token cost and latency. The default $T_{\max} = 15$ therefore captures most of the available quality gain without paying the full cost of the larger exploration budget.

Table 10: Sensitivity to the candidate count K. Hits@1 and F1 are percentages. Token cost is in thousands of tokens per question, and latency is in seconds per question. Higher is better for Hits@1 and F1, while lower is better for tokens and latency.

	CWQ				WebQSP				GrailQA			
K	Hits@1 ↑	F1 ↑	Tokens ↓	Latency ↓	Hits@1 ↑	F1 ↑	Tokens ↓	Latency ↓	Hits@1 ↑	F1 ↑	Tokens ↓	Latency ↓
5	79.83	70.80	10.605	29.419	85.36	75.47	5.324	19.781	82.06	75.86	6.169	27.861
10	80.42	70.96	12.140	37.194	86.69	77.16	6.751	21.610	85.43	79.02	6.414	24.690
15	82.58	73.24	17.380	59.147	87.13	77.88	7.981	26.509	85.92	79.43	9.244	33.589

Using only five relation candidates reduces token cost on all three datasets and latency on CWQ and WebQSP, but consistently lowers answer quality. Expanding the shortlist from $K = 10$ to $K = 15$ produces only modest Hits@1 and F1 gains while increasing token cost and latency. GrailQA also shows that a smaller candidate set need not reduce latency, because additional retries can offset the cheaper per-step shortlist. Overall, $K = 10$ provides the more balanced operating point.

D Embedding Model Sensitivity

The local LM relation ranker is an important source of inductive bias in AoG. We compare four checkpoints spanning compact English sentence encoders and larger retrieval-oriented models: the six-layer `multi-qa-MiniLM-L6-cos-v1`, based on MiniLM (Wang et al., 2020), `all-mpnet-base-v2`, based on MPNet (Song et al., 2020), `bge-large-en-v1.5`, from the BGE model family introduced with C-Pack (Xiao et al., 2024), and `bge-m3`, whose training unifies dense, sparse, and multi-vector retrieval (Chen et al., 2024a). Table 11 summarizes the model differences relevant to this study. Although BGE-M3 supports multiple retrieval modes, AoG uses its dense single-vector representation here so that the router interface remains unchanged.

Table 11: Embedding models used in the sensitivity study. Parameter counts and maximum lengths characterize the underlying models. Only dense single-vector retrieval is used in these experiments.

Model Name	Checkpoint	Backbone	Layers	Params.	Dim.	Max. length
MiniLM	`multi-qa-MiniLM-L6-cos-v1`	MiniLM/BERT	6	22.7M	384	512
MPNet	`all-mpnet-base-v2`	MPNet-base	12	109M	768	384
BGE-large	`bge-large-en-v1.5`	BERT-large	24	335M	1,024	512
BGE-M3	`bge-m3`	XLM-RoBERTa	24	568M	1,024	8,192

D.1 Evaluation Protocol

We evaluate AoG on 400 questions randomly sampled from each of CWQ test, WebQSP test, and GrailQA dev. Each sampled subset is kept fixed across embedding checkpoints and runs. All results in this section use these 400-question subsets, not the complete dataset splits. The sampled CWQ questions comprise 50 comparative, 150 composition, 150 conjunction, and 50 superlative questions. The GrailQA subset contains 94 i.i.d., 85 compositional, and 221 zero-shot questions. Each dataset–checkpoint pair uses `gpt-5.6-luna` for three runs with temperature 0, uncertainty threshold 0.35, entropy weight 0.5, candidate top-$K = 10$, maximum 15 actions, and two default retries.

D.2 Answer quality

Results are reported as mean ± standard deviation over three runs. Within each dataset, the best and second-best means are shown in **bold** and underlined, respectively. Ties share the same rank.

Table 12 shows a consistent ordering in the aggregate answer metrics: BGE-M3 is strongest on all three datasets, and BGE-large ranks second. Relative to MiniLM, BGE-M3 improves macro F1 by 11.86 points on CWQ, 14.41 points on WebQSP, and 8.90 points on GrailQA. The smaller GrailQA gap indicates that the effect of checkpoint choice depends on the schema and question distribution, rather than model size alone.

Table 12: Answer quality under different embedding models. All entries are percentages.

Dataset	Embedding	Hits@1	Macro F1
CWQ test (400)	MiniLM	64.58 ± 0.14	60.45 ± 0.73
	MPNet	69.58 ± 1.18	63.54 ± 0.43
	BGE-large	$\underline{79.58 \pm 0.95}$	$\underline{69.40 \pm 1.11}$
	BGE-M3	$\mathbf{81.33 \pm 0.42}$	$\mathbf{72.31 \pm 0.63}$
WebQSP test (400)	MiniLM	68.92 ± 1.38	60.57 ± 0.99
	MPNet	71.75 ± 0.50	62.92 ± 0.23
	BGE-large	$\underline{81.75 \pm 0.43}$	$\underline{70.21 \pm 0.75}$
	BGE-M3	$\mathbf{85.28 \pm 0.36}$	$\mathbf{74.98 \pm 0.73}$
GrailQA dev (400)	MiniLM	72.92 ± 1.13	70.61 ± 0.76
	MPNet	73.92 ± 0.76	71.31 ± 0.51
	BGE-large	$\underline{77.67 \pm 0.83}$	$\underline{74.07 \pm 0.74}$
	BGE-M3	$\mathbf{86.14 \pm 0.93}$	$\mathbf{79.51 \pm 0.36}$

D.3 Retrieval Quality

Both tables report mean $\pm$ standard deviation over three runs. Within each dataset, the best and second-best means are shown in **bold** and underlined, respectively.

The answer-quality trend is accompanied by stronger strict gold-edge retrieval (Table 13). BGE-M3 has the highest H@1 and MRR@10 on every dataset. BGE-large also improves substantially over the compact encoders on GrailQA, where strict H@1 rises from 49.97 for MiniLM to 56.22. The inverse-aware results in Table 14 increases some differences, especially on CWQ. This shows that schema directionality and inverse-relation handling can mask or amplify checkpoint differences under strict matching.

Table 13: Strict gold-edge retrieval quality. H@k and MRR@10 are reported as percentages.

Dataset	Embedding	H@1	H@5	H@10	MRR@10 (%)
CWQ test (400)	MiniLM	44.30 ± 0.13	50.53 ± 0.24	51.41 ± 0.23	46.98 ± 0.17
	MPNet	44.24 ± 0.83	49.50 ± 0.95	51.90 ± 0.65	46.66 ± 0.69
	BGE-large	$\underline{45.40 \pm 0.63}$	$\underline{51.54 \pm 0.78}$	$\underline{52.29 \pm 0.85}$	$\underline{48.09 \pm 0.56}$
	BGE-M3	$\mathbf{48.38 \pm 0.28}$	$\mathbf{54.23 \pm 0.14}$	$\mathbf{54.72 \pm 0.15}$	$\mathbf{50.67 \pm 0.19}$
WebQSP test (400)	MiniLM	65.63 ± 0.99	89.33 ± 0.59	$\underline{91.50 \pm 0.70}$	75.66 ± 0.75
	MPNet	67.38 ± 0.75	88.21 ± 0.89	90.21 ± 0.51	76.21 ± 0.46
	BGE-large	$\underline{71.75 \pm 1.23}$	$\underline{89.50 \pm 0.65}$	91.29 ± 0.47	$\underline{79.41 \pm 0.90}$
	BGE-M3	$\mathbf{75.08 \pm 0.64}$	$\mathbf{90.33 \pm 0.26}$	$\mathbf{93.13 \pm 0.25}$	$\mathbf{81.24 \pm 0.30}$
GrailQA dev (400)	MiniLM	49.97 ± 0.18	52.96 ± 0.10	56.26 ± 0.18	51.37 ± 0.12
	MPNet	41.15 ± 0.22	48.74 ± 0.13	58.84 ± 0.26	46.43 ± 0.19
	BGE-large	$\underline{56.22 \pm 0.10}$	$\underline{58.93 \pm 0.05}$	$\underline{63.10 \pm 0.10}$	$\underline{58.16 \pm 0.38}$
	BGE-M3	$\mathbf{57.53 \pm 0.14}$	$\mathbf{62.18 \pm 0.19}$	$\mathbf{69.06 \pm 0.13}$	$\mathbf{60.77 \pm 0.14}$

Table 14: Inverse-aware gold-edge retrieval quality on the 400-question subsets. H@k and MRR@10 are reported as percentages.

Dataset	Embedding	H@1	H@5	H@10	MRR@10 (%)
CWQ test (400)	MiniLM	64.49 ± 0.72	73.80 ± 0.58	75.12 ± 0.54	68.58 ± 0.54
	MPNet	63.71 ± 0.72	72.54 ± 1.04	75.68 ± 0.57	67.72 ± 0.66
	BGE-large	$\underline{69.89 \pm 1.25}$	$\mathbf{77.23 \pm 1.10}$	$\underline{83.41 \pm 1.06}$	$\underline{73.95 \pm 0.97}$
	BGE-M3	$\mathbf{76.74 \pm 0.83}$	$\mathbf{82.95 \pm 0.38}$	$\mathbf{88.85 \pm 0.21}$	$\mathbf{79.96 \pm 0.37}$
WebQSP test (400)	MiniLM	74.24 ± 0.45	90.04 ± 0.67	$\underline{93.50 \pm 0.70}$	78.34 ± 1.83
	MPNet	75.69 ± 0.81	90.21 ± 0.72	92.57 ± 0.68	78.96 ± 0.28
	BGE-large	$\underline{77.32 \pm 0.95}$	$\underline{91.72 \pm 0.84}$	93.29 ± 0.58	$\underline{81.26 \pm 1.70}$
	BGE-M3	$\mathbf{80.66 \pm 0.72}$	$\mathbf{92.48 \pm 0.38}$	$\mathbf{95.62 \pm 0.47}$	$\mathbf{83.72 \pm 0.65}$
GrailQA dev (400)	MiniLM	67.13 ± 0.53	75.63 ± 0.22	80.24 ± 0.10	71.46 ± 0.39
	MPNet	69.56 ± 0.81	77.38 ± 0.88	83.46 ± 0.91	74.18 ± 0.82
	BGE-large	$\underline{74.99 \pm 0.11}$	$\underline{78.66 \pm 0.10}$	$\underline{88.95 \pm 0.17}$	$\underline{77.39 \pm 0.08}$
	BGE-M3	$\mathbf{79.85 \pm 0.57}$	$\mathbf{87.97 \pm 0.67}$	$\mathbf{92.26 \pm 0.53}$	$\mathbf{82.97 \pm 0.46}$

D.4 Efficiency and Interpretation

Results are reported as mean ± standard deviation over three runs. Within each dataset, the best and second-best means are shown in **bold** and underlined, respectively.

Table 15 shows that model size does not induce a monotonic end-to-end cost increase. LLM calls vary only modestly, and BGE-M3 is faster than the compact checkpoints on WebQSP but slower than BGE-large on GrailQA. These latency measurements include LLM inference, graph queries, routing, retries, and embedding computation. They are not isolated encoder benchmarks. Input and output are thousands of tokens per question. Cached tokens are included in the input counts rather than added to them.

Taken together, the results suggest that stronger relation representations can improve both gold-edge ranking and final answers. BGE-M3 is the strongest LM ranker backbone in the sensitivity test, followed by BGE-large.

Table 15: LLM usage and end-to-end latency per question. Input and output are thousands of tokens. Lower is better.

Dataset	Embedding	LLM calls	Input	Output	Latency (s)
CWQ test (400)	MiniLM	6.578 ± 0.022	$\mathbf{12.836 \pm 0.050}$	$\mathbf{1.308 \pm 0.008}$	54.552 ± 0.515
	MPNet	6.409 ± 0.050	14.000 ± 0.059	1.645 ± 0.030	$\mathbf{48.727 \pm 0.809}$
	BGE-large	6.446 ± 0.006	16.068 ± 0.050	1.481 ± 0.026	50.089 ± 0.464
	BGE-M3	$\mathbf{6.260 \pm 0.091}$	15.646 ± 1.081	1.428 ± 0.095	49.197 ± 1.758
WebQSP test (400)	MiniLM	2.739 ± 0.035	$\mathbf{5.863 \pm 0.024}$	0.740 ± 0.098	24.377 ± 0.150
	MPNet	2.753 ± 0.016	5.976 ± 0.018	0.744 ± 0.036	22.631 ± 0.433
	BGE-large	$\mathbf{2.704 \pm 0.029}$	6.060 ± 0.061	0.736 ± 0.029	23.057 ± 0.157
	BGE-M3	2.856 ± 0.029	6.206 ± 0.021	$\mathbf{0.653 \pm 0.18}$	$\mathbf{21.135 \pm 0.023}$
GrailQA dev (400)	MiniLM	2.382 ± 0.024	5.543 ± 0.025	0.605 ± 0.008	18.285 ± 0.371
	MPNet	2.465 ± 0.043	$\mathbf{4.721 \pm 0.083}$	$\mathbf{0.527 \pm 0.039}$	28.712 ± 1.165
	BGE-large	$\mathbf{2.372 \pm 0.048}$	5.558 ± 0.030	0.608 ± 0.005	$\mathbf{17.622 \pm 0.126}$
	BGE-M3	2.638 ± 0.009	4.903 ± 0.039	0.673 ± 0.003	23.245 ± 0.347

E Case Study and Mechanism Analysis

These two cases illustrate adaptive relation exploration and memory-to-action loop using recorded traces. Costs report mean ± sample standard deviation over all three runs, including unsuccessful ones. Hits@1 counts the answer accuracy across three runs. All methods use `gpt-5.6-luna` as the LLM backbone.

E.1 Adaptive Exploration: Allocating Exploration Budget

Table 16: CWQ case: AoG and three methods' exploration paths. Each method select paths from the best performance from three runs.

Question. What college with the latest founding date did Theodore Schwann attend?
CWQ ID. `WebQTest-1732_2cab9ad36ae89d0762bb7b30e74a7e32`
Gold answer. University of Bonn (`m.0m7yh`).

Decision / selected relation	H	J	ρ	γ	Route
1. Retrieve education records `people.person.education`	0.000	0.542	0.135	0.312	Policy 1
2. Resolve attended institutions `education.education.institution`	0.000	0.516	0.129	0.327	Policy 1
3. Retrieve founding dates `organization.organization.date_founded`	0.301	0.537	0.360	0.312	Policy 2

Method	Recorded reasoning path	Output / Hits@1 (Count)	Latency (s) / Tokens (k)
AoG	Schwann → education records → attended institutions → founding dates → latest institution. Policies are Policy 1 → Policy 1 → Policy 2, as logged above.	University of Bonn `m.0m7yh` Hits@1 (Count): 3/3	34.10 ± 2.65 5.534 ± 0.054
SoG	Retrieve three education records, and resolve all three institution IDs. List their properties and query `date_founded` for each institution. Finally return Bonn (11 `Search` calls).	University of Bonn `m.0m7yh` Hits@1 (Count): 3/3	31.78 ± 3.27 10.204 ± 0.016
PoG	`people.person.education` → three CVT IDs. Stop at depth 1: memory lacks institutions and dates, yet the evaluator labels a CVT ID "University of Bonn." No institution or founding-date traversal is logged.	Label: Bonn `m.0n0tzlj` (education record) Hits@1 (Count): 0/3	51.20 ± 1.40 4.372 ± 0.119
VoG	Follow `people.person.education`, and query `education.education.start_date` on two records (1829, 1830). The verifier flags attendance, not founding dates. Two plan revisions fail to recover institution or founding-date evidence, and stop at depth 2.	No answer (∅) Insufficient evidence Hits@1 (Count): 0/3	151.81 ± 45.98 15.082 ± 11.680

Note. This case uses $w_H = 0.75$ rather than the default 0.50, so $\rho = 0.75H + 0.25J$. The reported H, J, and ρ are rounded independently to three decimal places.

Analysis. Exploration budget is allocated unevenly across the reasoning path. AoG probes one relation at each of the first two decisions through Policy 1. When founding dates and visiting scholars receive tied LLM scores, the uncertainty exceeds the threshold and Policy 2 probes five shortlisted relations. Additional exploration is thus concentrated on the ambiguous decision that determines the date comparison. SoG also succeeds through 11 Search calls, spending its budget on explicit property inspection across three institutions. PoG stops before resolving institutions or retrieving founding dates, leaving the required comparison unsupported. VoG spends two plan revisions on a trajectory that retrieves attendance dates, but never obtains the missing founding dates. The distinction is not simply a larger budget. It is whether additional exploration is triggered at the right decision and produces the evidence needed to answer.

E.2 CONTEXTUAL RETRY AFTER A FAILED BRANCH

Table 17: GrailQA case: AOG and three methods' exploration paths. Each method select paths from the best performance from three runs.

Question. A variant of ATmega168PA is in what computer processor family?
GrailQA ID. `3204168005000` (compositional).
Gold answer. Atmel AVR (`m.0pk4w`).
Subgoals. `c1s1`: identify the variant. `c1s2`: identify its processor family.

Step	Attempt / subgoal	Recorded action and path	Result / control effect
1	Attempt 1 `c1s1`	ATmega168PA → megaAVR. Outgoing `processor_family`.	Subgoal unfinished. `expected_output_type` triggers the memory update.
2	Attempt 2 `c1s1`	Resume at ATmega168PA. Exclude the outgoing `processor_family` action at this source and subgoal.	No outgoing candidate remains. Switch to incoming relations.
3	Attempt 2 `c1s1`	ATmega168PA ← megaAVR. Incoming `variants` via `get_head_entity`.	Complete `c1s1`. Pass megaAVR to `c1s2`.
4	Attempt 2 `c1s2`	megaAVR → Atmel AVR. Outgoing `processor_family` remains usable at the new source.	Complete `c1s2`. Return `m.0pk4w`.

Method	Recorded reasoning and stopping behavior	Output / Hits@1 (Count)	Latency (s) / Tokens (k)
AOG	**Contextual action exclusion.** Run 2 blocks the failed action, switches direction, and reuses `processor_family` at megaAVR.	Atmel AVR `m.0pk4w` Hits@1 (Count): 3/3	20.43 ± 1.97 4.633 ± 0.350
SoG	**Tool-response history.** One outgoing `Search` at ATmega168PA retrieves megaAVR. The agent stops at this intermediate entity in all three runs, without the required second hop.	megaAVR `m.06qhzs3` Hits@1 (Count): 0/3	18.64 ± 1.35 2.239 ± 0.113
DoG	**Debate-capable design, but no debate invoked.** The first answer trial accepts the direct `processor_family` triple and returns megaAVR in all three runs, before any rewriting or correction.	megaAVR `m.06qhzs3` Hits@1 (Count): 0/3	23.31 ± 0.79 0.781 ± 0.007
VoG	**Verification and revision.** Run 2 reaches megaAVR, then follows the inverse `processor_family` edge back to ATmega168PA. Two revisions still yield megaAVR. Run 3 instead follows the outgoing edge to Atmel AVR and succeeds.	megaAVR `m.06qhzs3` Hits@1 (Count): 1/3	81.42 ± 4.55 6.375 ± 2.017

Analysis. AOG uses the failed branch to constrain its next action. It excludes `processor_family` at ATmega168PA in `c1s1` as the first subgoal is to find the `variants` type, and reuses `processor_family` in `c1s2` at megaAVR to reach the correct answer Atmel AVR. The exclusion is specific to the source and subgoal, preserving a useful relation elsewhere. SoG treats the first retrieved family as the final answer and misses the remaining compositional step. DoG makes the same premature sufficiency judgment, so debate and rewriting are never triggered. VoG runs 1 and 2 revisit ATmega168PA and return megaAVR despite verification and revision. Its third run follows the outgoing edge and succeeds. These traces show that detecting an inconsistency does not guarantee a corrective graph action. The case illustrates context-specific action exclusion during retry.

F LIMITATIONS AND SCOPE

The method is evaluated under a deliberately bounded setting. First, the entity and relation semantics, topic links, and executable graph operations depend on Freebase and on the fixed KG snapshot used in this work. Changes in the underlying schema, aliases, or snapshot contents may change both the available candidate relations and the final answers. The reported results should not be interpreted as snapshot-independent KGQA performance.

Second, the Query Planner and LLM relation ranker rely on an external LLM API. Consequently, API model revisions, service availability, latency, rate limits, and nondeterministic output behavior are part of the operational environment. The local LM ranker removes a local embedding-service dependency, but it does not make the planner/ranker stack fully self-contained. Exact reproduction requires recording the external model/version and API configuration in addition to the local code and checkpoint state.

Third, the empirical study covers only three datasets: CWQ, WebQSP, and GrailQA. Several diagnostic, embedding, and sensitivity analyses use a finite question subset rather than the complete public split, and the repeated-run protocol uses only three seeds, $\{1, 2, 3\}$. These choices support controlled comparisons but do not establish robustness across all question distributions, larger seed ensembles, other knowledge graphs, or alternative LLM providers. The scope of the claims is therefore limited to the stated Freebase snapshot, the evaluated datasets and subsets, and the reported three-seed protocol.