KGVoyager: Knowledge Graph Agnostic Question Answering via Agentic Navigation

Anonymous ACL submission

Abstract

Knowledge Graph Question Answering (KGQA) over RDF graphs remains challenging in domain-specific settings, where formal ontologies and curated text–SPARQL pairs are often unavailable. We present KGVoyager, a KG-agnostic agentic architecture that generates SPARQL queries from natural language questions by dynamically discovering graph structure and semantics, requiring only a query endpoint of the underlying graph. Using a think–act–observe loop with search, exploration, and execution tools, KGVoyager maps terms to graph IRIs, uncovers structure, and refines queries through execution feedback—all without pre-existing ontologies or examples. Unlike the prior state of the art, KGVoyager requires only a lightweight class index which renders it applicable for far more real-world endpoints. Across four benchmarks, KGVoyager improves F1 by ∼8 points while cutting cost and runtime by ∼22% each.

1 Introduction

RDF knowledge graphs (KGs) provide a robust framework for representing complex, interconnected data across diverse domains [15, 33]. However, utilizing them effectively requires understanding graph-specific data models and technical query languages such as SPARQL [17, 12, 20, 1]. Natural language interfaces that map user questions to SPARQL queries lower this barrier and have driven significant research interest in knowledge graph question answering (KGQA) [8, 13, 21, 40, 30, 18].

LLM-based KGQA systhems have achieved state-of-the-art benchmark performance [8] over earlier machine-learning [5], deep-learning [39, 30, 3], template, and rule-based approaches [12]. These solutions fundamentally leverage ontologies and example text-query pairs for grounding and guiding query generation. Ontologies encode classes, attributes, relationships, and constraints that help models produce semantically valid queries [11, 22, 25]. In addition to ontologies, example text-SPARQL pairs improve performance of such natural language interfaces by serving as in-context examples or facilitating fine-tuning [37, 40, 32].

This dual reliance on ontologies and text-query pairs parallels SQL-based DBQA, where systems depend on the database schema and text-SQL pairs to ground query construction [41].

However, unlike SQL-based systems, which benefit from inherent database schemas, SPARQL interfaces for knowledge graphs often lack such resources, due to the flexible, schema-optional design of the RDF framework. In particular, we identify three distinct challenges, as follows. (1) *Ontology deficiency*: Many knowledge graphs lack a formal ontological structure. They may define entities, classes, and properties without specifying pivotal semantic relationships such as class hierarchies, property characteristics, and other semantic attributes. (2) *Ontology opacity*: Even when knowledge graphs do incorporate ontological structure, they predominantly lack documentation that exposes or explains such structure and related constraints [14]. (3) *Example scarcity*: Creating example text-SPARQL pairs for new domain-specific knowledge graphs is expensive and challenging. It requires human experts who possess deep domain knowledge, advanced SPARQL proficiency, and awareness of the underlying ontology. This is corroborated by the pronounced scarcity of query examples for domain-specific knowledge graphs [28, 26].

These challenges hinder the development of effective KGQA systems for *domain-specific* settings. Most recent natural language to SPARQL systems [21, 40, 30, 18, 3, 9, 32, 37] avoid these constraints by focusing on *open-domain* knowledge graphs such as Wikidata and DBpedia, which benefit from inherent knowledge in LLMs [13, 37], curated text-SPARQL datasets [10, 34, 35], and

community ontology tooling [21, 40]. These assets are rarely available for arbitrary domain-specific knowledge graphs.

As ontologies may be unavailable or opaque and example pairs scarce, we treat a SPARQL query endpoint as the sole universally assumable resource. Our system thereby aims to enable natural language interfaces for arbitrary knowledge graphs *given only query endpoint access*. To achieve this goal, our system must simulate the exploratory learning process a human expert would undertake when first encountering an unfamiliar knowledge graph. To this end, we present KGVoyager, an agentic architecture that uses a suite of KG-agnostic tools to dynamically discover latent graph structure, semantics, and instance data. KGVoyager then uses this discovered information to generate SPARQL queries that answer user questions. Specifically, we design a set of searching, sampling, and navigation tools that enable systematic exploration of arbitrary knowledge graphs. Using these tools within a think-act-observe loop [38], the agent identifies relevant classes and properties, maps local graph structure, and validates query drafts through execution feedback until it derives an accurate SPARQL query.

To the best of our knowledge, the only prior work sharing a similar endpoint-only, KG-agnostic objective is GRASP [36], which also targets zero-shot SPARQL generation over arbitrary knowledge graphs but relies on a complete entity index that builds for only ∼21% of Jena-served and 0% of QLever-served okn.us [26] endpoints. KGVoyager's lightweight class-level index builds on up to ∼97% of the same endpoints; other fundamental differences are detailed in Section 2. Across four domain-specific benchmarks, KGVoyager improves F1 by ∼8 points on average while reducing cost and runtime by ∼22% each.

In summary, our key contributions are:

- **KGVoyager**, an agentic architecture that leverages a suite of KG-agnostic tools to generate SPARQL queries that answer user questions without relying on pre-existing ontologies or curated example pairs.
- **Evaluation** of KGVoyager against the current state-of-the-art across diverse domain-specific knowledge graphs, demonstrating superior performance in correctness, efficiency, and applicability.
- **Open-source release** of KGVoyager, enabling practitioners to deploy the system against any SPARQL endpoint with minimal configuration. The codebase, datasets, experiment results, and demo recordings are available at https://anonymous.4open.science/r/nl2sparql2-C171.

2 Related Work

GRASP [36] is the prior work most similar to KGVoyager, employing an LLM agent with KG-agnostic tools for zero-shot SPARQL generation over arbitrary knowledge graphs. KGVoyager differs in three respects: the scale of required indexing, the inference of property semantics, and the retrieval mechanism. *First*, GRASP relies on a complete index of all entities, which is impractical because the sheer volume of entities makes exhaustive retrieval computationally infeasible for many real-world endpoints; our feasibility study (Section 4.1) shows it succeeds for only ∼21% of Jena-served and 0% of QLever-served endpoints in the OKN registry [26]. KGVoyager instead uses a class-level index that scales with the ontology rather than the data, and is typically orders of magnitude smaller. *Second*, GRASP differentiates properties using ontological domain/range annotations, which are not always present (the ontology-deficiency challenge); KGVoyager infers them directly from data via dedicated exploration tools. *Third*, GRASP uses semantic and keyword search as separate signals across its tools, whereas KGVoyager combines them jointly via hybrid retrieval, which has been shown to outperform either signal alone in various applications [19, 29].

ToG [32] pursues KG-agnostic question answering through an agentic LLM that traverses the knowledge graph hop-by-hop. Unlike KGVoyager, which explores latent structure and semantics to construct an executable SPARQL query, ToG traverses instance-level data node-by-node to locate answers, and the number of LLM calls therefore grows with the hop distance from the initial node to the answer. Despite its KG-agnostic intent, ToG was evaluated exclusively on open-domain benchmarks over Wikidata and Freebase, which provides only limited evidence of generalization to domain-specific knowledge graphs. It also requires downloading and locally deploying the entire knowledge graph, which is infeasible in endpoint-only settings.

Other systems rely heavily on KG-specific engineering. **Spinach** [21] uses Wikidata-specific tools and an in-context learning agent to simulate expert exploration for question answering over Wikidata, while **Expasy** [11] presents a bioinformatics

text-to-SPARQL system that mitigates hallucinations via schema validation, but relies on curated text-query pairs and KG-specific schemas. Their tight coupling to specific domains limits generalization to arbitrary knowledge graphs without substantial re-engineering, motivating KGVoyager's KG-agnostic architectures.

3 Methodology

We present KGVoyager, a KG-agnostic agentic reasoning architecture equipped with dynamic exploration tools to discover latent graph structure, semantics, and instances to reflectively generate SPARQL queries from natural language questions. The framework relies on the minimal assumption that the target knowledge graph adheres to W3C RDFS standards for class definitions—a convention adopted across almost all RDF knowledge graphs. Under this assumption, search and exploration tools can be implemented via KG-agnostic SPARQL queries that enable the tools to be universally executable across knowledge graphs.

Given a natural language question, KGVoyager formulates SPARQL generation as an iterative exploration problem. The agent operates in a think-act-observe loop [38], reasoning about what to do next, invoking a tool, and incorporating the result before deciding its next step. Its KG-agnostic tools are organized into three groups: (i) *search tools*, which retrieve candidate classes, node instances, and properties from natural language cues; (ii) *exploration tools*, which inspect how these candidates are used and connected in the knowledge graph; and (iii) an *execution tool*, which runs candidate SPARQL queries and returns feedback. The agent uses search to map terms from the user question to KG IRIs, exploration to fill gaps related to how different discovered KG IRIs interact, and execution to validate and refine query drafts until producing a SPARQL query that it determines correctly answers the question.

Figure 1 summarizes the tool suite. Search tools resolve classes, properties, and class-scoped instances from natural-language cues. Exploration tools inspect entity neighborhoods and property usage to expose structure that is not immediately apparent from search results. *execute_query* validates partial or complete query drafts. The complete KGVoyager system prompt and tool specification are included in the Appendix. In line with the ReAct framework [38], the agent may invoke tools in any order and at any frequency. However, we prompt the agent to generally follow a search, exploration, and execution sequence, mirroring the natural workflow of a human expert encountering an unfamiliar knowledge graph for the first time. The agent submits the final query through a dedicated `answer` tool to mark task completion. Following GRASP [36], we also allow the agent to terminate the process early via a `cancel` tool when generating a query is unfeasible for a given question under the knowledge graph.

3.1 Search Tools

Class Search. The *search_all_classes* tool accepts a natural language search term and returns a ranked list of classes whose IRIs or labels are semantically or lexically related to the term. Each result includes the class label, aliases, and instance count. On its first invocation, the tool fetches and locally caches a lightweight class index from the endpoint via SPARQL. Subsequent searches operate directly on this cached index, avoiding repeated SPARQL requests. To retrieve and rank results, we employ a hybrid approach that combines dense vector similarity with sparse BM25 lexical scoring via reciprocal rank fusion [7], which has been shown to consistently outperform either signal alone in various applications [19, 29].

Each result is returned with a relevance score. To prevent low-quality candidates from polluting the agent's context—a phenomenon known as context rot [16]—we apply an adaptive filtering mechanism. This mechanism ensures a minimum number of results by pruning candidates whose relevance score ratio relative to the top-ranked result is less than δ.

Instance Lookup. The *lookup_class_instances* tool retrieves instances of a given class, optionally filtered by an *exact_term*. Once the agent has identified a candidate class IRI via *search_all_classes*, it uses this tool either to verify that the class contains the expected content or to locate specific entities mentioned in the natural language question. The default lookup matches the *exact_term* against instance labels and IRIs. Since this matching relies on lexical SPARQL string filters, abbreviations, alternative spellings, or overly specific names may return no results. To handle such cases, the tool also supports a broader *fallback_term* which is used to retrieve a larger candidate set when the *exact_term* search fails. This larger candidate set is then filtered by scoring the similarity between each

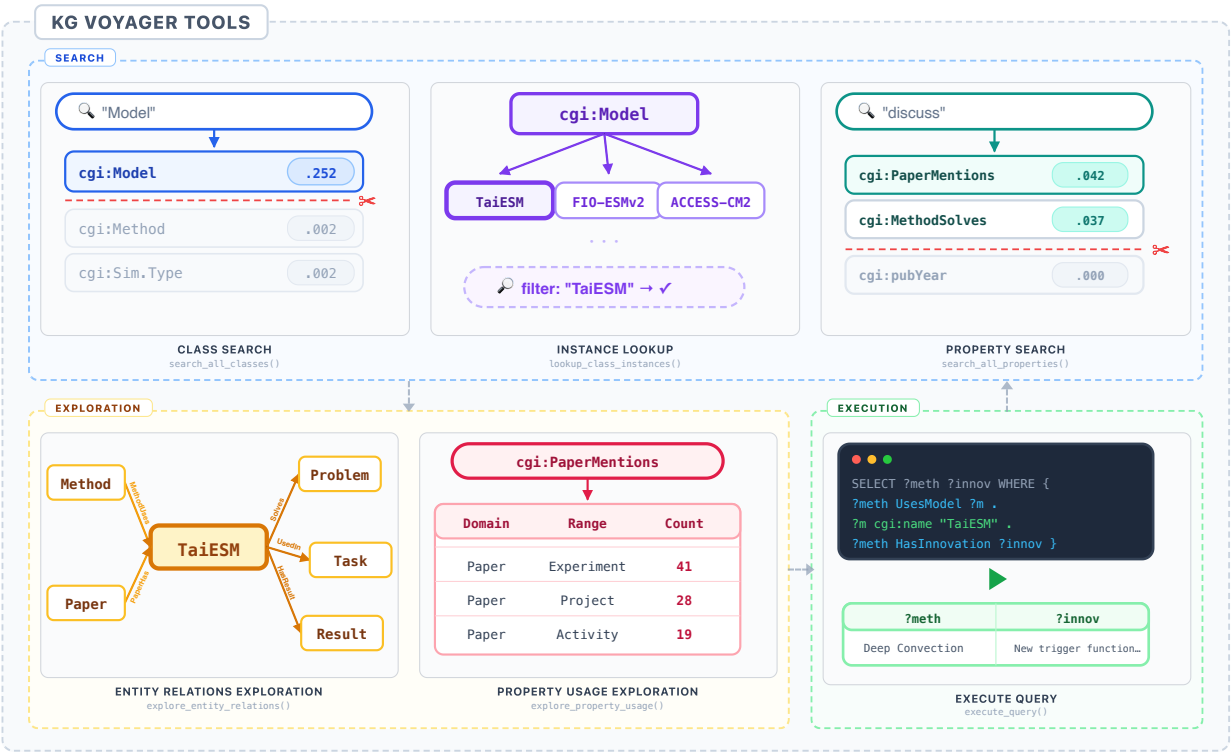

Figure 1: The KGVoyager tool suite. Examples inspired from operation on the Climate Models knowledge graph.

candidate labels and the *exact_term*. For knowledge graphs that store useful names or identifiers in literal-valued attributes rather than standard labels, the *lookup_class_instances* tool also supports an *all_attributes* mode that further extends the search to all literal values of class instances, at the cost of slower execution. These class-scoped mechanisms enable precise entity resolution without requiring an exhaustive entity index over the entire knowledge graph, which is unfeasible for most endpoints.

Property Search. The *search_all_properties* tool closely mirrors the class search functionality for properties. Given a search term, it returns a ranked list of property IRIs along with labels, aliases, usage counts as well as domains and ranges, if available. As with class search, the property index is fetched and cached on first use by enumerating all distinct predicates from the endpoint; the same hybrid retrieval and adaptive filtering are then applied locally when performing a search.

3.2 Exploration Tools

While the search tools resolve natural language terms to candidate class, property, or instance IRIs, they do not reveal how those IRIs are structurally embedded and connected in the graph. To understand the interaction of properties and the neighborhoods of entities, the agent requires tools that inspect the graph directly.

Property Usage Exploration. Many knowledge graphs lack explicit or accurate domain and range annotations, leading to ambiguity in property interpretation. The *explore_property_usage* tool bridges this gap by examining how a property is actually used. Given a property IRI, it samples triples with that property and aggregates the subject and object classes involved, along with representative instances and their labels. This allows the agent to verify that a candidate property empirically connects the kinds of entities expected by the question, before committing it to the query as well as to disambiguate properties with similar labels. For instance, while a property labeled "located in" connects organizations and companies to cities, a different property "location" may connect events to venues.

Entity Relation Exploration. The tool *explore_entity_relations* inspects the local neighborhood of a given entity. From an entity IRI, it retrieves the incoming and outgoing properties along with example subjects or objects reached through each, including labels when available. This helps understand how the entity is embedded in the graph, particularly when relation search has not surfaced the property needed for the question. The tool can also be used in reverse: given a literal value, it

retrieves entities that hold that value as an attribute.

3.3 Execution Tools

The *execute_query* tool runs a SPARQL query against the endpoint and returns the results. It serves a dual role: executing the final candidate query to obtain the answer, and providing execution-based feedback throughout the reasoning process. The agent can run intermediate queries to test hypotheses about graph structure, verify that IRIs produce non-empty results, and diagnose errors in partial queries. This iterative feedback is central to the agent's ability to self-correct and converge on accurate queries.

3.4 Running Example

Figure 2 shows KGVoyager resolving the question *"Count the number of papers that Jake Grozdanovski has published in the International Conference on Software and Data Technologies"* in ten tool calls, over the DBLP knowledge graph. During **Search**, the agent decomposes the question into semantic search terms and resolves them to KG elements. The agent infers *author*—implied by "Jake Grozdanovski has published" though not a literal keyword—to map it to `ds:Creator` (step 1), then maps *paper* to `ds:Publication` (step 2) and *conference* to `ds:Conference` (step 3). It reuses *author* to surface the authorship property `ds:authoredBy` (step 4), while a search on *"published"* (step 5) returns only journal/book properties, leaving the property linking publications to conference venues unresolved. The agent then grounds the two named entities: the conference name to `dsc:icsoft` (step 6) and the person name to `dp2:9368` (step 7). In the **Explore** phase, the agent inspects `dsc:icsoft` directly, which reveals the incoming relation `ds:publishedInStream`; a targeted follow-up search confirms it is the Publication → Stream property required for the publication-conference relationship. In **Execution**, the agent has fully resolved the triple pattern `?paper ds:authoredBy dp2:9368; ds:publishedInStream dsc:icsoft`, executes the COUNT query, and obtains the answer **1**.

4 Experiment Results

4.1 Feasibility Analysis

As discussed in Section 2, GRASP requires a comprehensive entity index, whereas KGVoyager only requires a lightweight class index. We assessed the

Figure 2: A complete KGVoyager example. Numbered blocks showtool calls in the order the agent invoked them. Short-hands such as 'ds:' represent shortened namespace pre-fixes used in the actual graph.

Knowledge Graph	Jena Entity	Jena Class	QLever Entity	QLever Class
biobricks-aopwiki	✓	✓	×	✓
biobricks-ice	×	×	×	✓
biobricks-mesh	×	×	×	✓
biobricks-pubchem-annotations	×	✓	×	✓
biobricks-tox21	✓	✓	×	✓
biobricks-toxcast	×	×	×	✓
biohealth	×	✓	×	✓
climatemodelskg	✓	✓	×	✓
dream-kg	✓	✓	×	✓
fio-kg	×	×	×	✓
gene-expression-atlas-okn	×	✓	×	✓
geoconnex	×	✓	×	✓
hydrology-kg	⏲	✓	×	✓
identifier-mappings	×	✓	×	✓
nasa-gesdisc-kg	×	✓	×	✓
nde	×	✓	×	✓
neighborhood-information-kg	✓	✓	×	✓
rural-kg	✓	✓	×	✓
sawgraph	×	×	×	✓
scales	×	✓	×	✓
secure-chain-kg	×	✓	×	✓
semopenalex	×	×	×	✓
sockg	×	✓	×	✓
spatial-kg	⏲	⏲	×	✓
spoke-genelab	×	✓	×	✓
sud-okn	×	×	×	✓
ubergraph	×	✓	×	✓
uf-okn	⏲	✓	×	✓
wikidata	×	×	×	⏲
Success Rate	**20.7%**	**69.0%**	**0%**	**96.6%**

Table 1: Indexing feasibility (✓: Success, ×: Failed, ⏲: Timeout) across 29 domain-specific SPARQL endpoints from okn.us under two settings.

practical feasibility of both by attempting to build their respective indices over 29 domain-specific SPARQL endpoints from okn.us [26], a catalog of graphs developed under the U.S. NSF Proto-OKN initiative. For GRASP, we used its official entity index query. We did the comparison under two settings: (i) the individual Apache Jena endpoints deployed for okn.us, and (ii) the unified federated endpoint that aggregates all graphs and is powered by QLever [4]. Citing general performance advantages, okn.us plans to deprecate the Jena endpoints in favor of QLever.

Table 1 reports the results. Under Jena, class indexing succeeded for 69.0% of endpoints while entity indexing succeeded for only ∼21%, confirming the impracticality of GRASP's entity-index requirement; the Jena entity-indexing failures returned opaque server errors with no actionable diagnostic information. The gap widens under QLever: class indexing succeeded for 96.6% of endpoints (28/29), while entity indexing failed on all 29, with failures manifesting as server memory crashes—an interpretable outcome given the size of entity indices. These results show that the feasibility of entity-level indexing is highly sensitive to the underlying SPARQL engine and knowledge graph.

4.2 Benchmarks and Models

We evaluated question answering on four domain-specific benchmarks: Climate Models KG (52 questions on climate-science literature), SOCKG [31] (54 questions on soil-carbon modeling), DREAM-KG (14 questions on local government service data), and DBLP-QuAD (100 questions on scholarly metadata). The first three are associated with okn.us [26], with question-SPARQL pairs supplied by the respective KG developers. DBLP-QuAD is a public question-SPARQL benchmark; we randomly sampled 100 questions from its test split [2].

Although our feasibility study (Section 4.1) showed that GRASP's entity index generally cannot be built from endpoint access alone, we deliberately compared KGVoyager against GRASP under GRASP's *best-case* scenario—one in which the entity index is available. We obtained this index through one of three routes: (i) building it directly on relatively small KGs whose Jena endpoint can serve the entity-index query, as for Climate Models KG and DREAM-KG; (ii) locally redeploying the knowledge graph and extracting the index from the deployment, as for SOCKG, whose okn.us endpoint fails entity index creation (Section 4.1); or (iii) using a pre-built index, as provided by GRASP's authors for DBLP-QuAD.

We ran our experiments on these benchmarks with four backbone models spanning closed and open systems of varying size and agentic capability. GPT-5.3 Codex (OpenAI, 2026) is an agentic coding model for tool-based coding and computer-use tasks [27]. GPT-5.4 Mini (OpenAI, 2026) is a smaller GPT-5.4 variant optimized for lower latency, cost, and developer-agent workloads [24]. Gemini 3 Flash Preview (Google, 2025) is an efficient Flash-family model reported to approach or surpass GPT-5.2 on selected benchmarks at a lighter cost-and-latency profile [6]. Devstral Small (Mistral AI, 2025) is an open 24B-parameter model specialized for coding agents [23].

4.3 QA Performance

We compared KGVoyager against the prior state of the art, GRASP [36], on the benchmarks and models above. For each combination of method, benchmark and model, we average three independent runs and report F1, monetary cost, agent iterations,

Benchmark	F1 (%)			Cost ($)			Iterations			Time (s)		
	KG Voyager	GRASP	Δ	KG Voyager	GRASP	Δ	KG Voyager	GRASP	Δ	KG Voyager	GRASP	Δ
Climate Models KG	**67.0**	52.9	+14.0	**0.66**	1.01	−35.1%	**14**	23	−9	**27.6**	58.8	−53.1%
SOCKG	**57.3**	53.3	+4.1	**0.68**	0.99	−31.2%	**15**	21	−6	**26.6**	47.6	−44.1%
DREAM-KG	**67.4**	58.3	+9.1	**0.27**	0.31	−13.1%	**17**	22	−5	**31.4**	49.4	−36.4%
DBLP	**60.4**	56.4	+3.9	**1.03**	1.11	−7.5%	**11**	12	−1	63.0	**43.8**	+43.9%
Average	**63.0**	55.2	+7.8	**0.66**	0.86	−21.7%	**14.3**	19.5	−5.3	**37.2**	49.9	−22.4%

Table 2: Performance by benchmark, averaged across four backbone models.

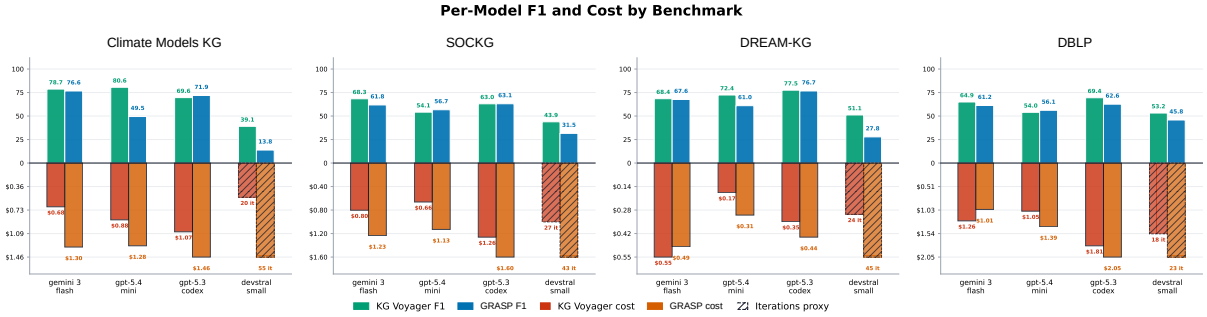

Figure 3: Per-model F1 and monetary cost by benchmark. F1 bars shown above the baseline, cost below; hatched bars use iteration count as a proxy where monetary cost is unavailable.

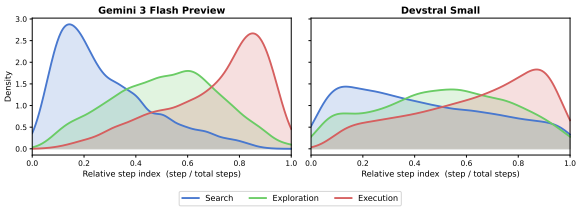

Figure 4: Relative timing of tool use by model: kernel density estimates of the relative step index for the search, exploration, and execution tool groups.

and wall-clock time. Following GRASP, we adopt the row-major F1 metric of Spinach [21].

Table 2 reports model-averaged results for each benchmark, while Figure 3 shows per-model F1 and monetary cost. KGVoyager improves F1 on all four benchmarks, with gains ranging from 3.9 percentage points on DBLP to 14.0 points on Climate Models KG. It also reduces cost on every benchmark and requires fewer agent iterations throughout. Runtime is lower on three benchmarks; DBLP is the exception because its template-based questions focus heavily on specific entity instances (e.g., paper and author names), which GRASP resolves instantly from its pre-supplied entity index, while KGVoyager resolves them through class-level search.

To better understand the aggregate F1 and cost differences, we manually inspected contrast cases where KGVoyager outperformed GRASP. We found three recurring patterns. First, KGVoyager's separate class, instance, and property tools help avoid cases where GRASP must distinguish a specific class from a large pool of instance entities. Second, *explore_entity_relations* gives immediate neighborhood information and concrete example values that are essential in some cases; GRASP can approximate this via its *list* tool or custom queries, but often with more iterations and fragility. Finally, GRASP is more prone to context rot because it lacks adaptive filtering over search candidates, which likely contributes to weaker and longer Devstral Small runs.

4.4 Tool-Use Dynamics

We further analyze whether the agent follows the search–explore–execute progression described in Section 3. For each tool call, we compute its relative step index, defined as position divided by total run iterations, and pool values across the three KGVoyager runs per benchmark-model pair.

Figure 4 contrasts Gemini 3 Flash Preview and Devstral Small. Both follow the intended temporal structure: search early, exploration in the middle, and execution later. But Gemini shows cleaner phase separation while Devstral exhibits more overlap. This aligns with benchmark performance: Devstral's benchmark-averaged KGVoyager F1 is lower (46.8% vs. 70.1%) and it requires more iterations on average (22.30 vs. 12.39), suggesting that phase mixing is a useful signal that the model finds the task challenging.

4.5 Ablation Study

We conducted three targeted ablations with Gemini 3 Flash Preview to examine design choices beyond the foundational tools: entity-neighborhood exploration, property-usage exploration, and hybrid retrieval. Each ablation result averages three independent runs. We do not ablate the search tools,

Variant	F1 (%)	Cost ($)	Iter.	Time (s)
Full KGVoyager	**68.3**	**0.80**	**11.0**	**19.7**
Without entity relations	66.5	0.84	11.2	24.0

Table 3: Ablation of *explore_entity_relations* on SOCKG using Gemini 3 Flash Preview.

Variant	F1 (%)	Cost ($)	Iter.	Time (s)
Full KGVoyager	**64.9**	1.26	**9**	57.3
Without entity relations	62.8	**1.11**	9	**55.3**

Table 4: Ablation of *explore_entity_relations* on DBLP using Gemini 3 Flash Preview.

Variant	F1 (%)	Cost ($)	Iter.	Time (s)
With property usage	**62.7**	1.25	17	66.7
Without property usage	54.0	**0.91**	**15**	**57.6**

Table 5: Ablation of *explore_property_usage* on SOCKG using Gemini 3 Flash Preview.

Variant	F1 (%)	Cost ($)	Iter.	Time (s)
Hybrid retrieval	**68.3**	**0.80**	**11**	**19.7**
Keyword-only retrieval	65.6	1.77	21	42.9

Table 6: Ablation of hybrid retrieval on SOCKG using Gemini 3 Flash Preview.

since resolving question terms to graph IRIs is fundamental to any endpoint-only text-to-SPARQL system. We also retain query execution, whose feedback is required both to validate the final answer and to keep the agentic setting comparable to GRASP. The ablations therefore focus on the additional mechanisms that distinguish how KGVoyager navigates and interprets unfamiliar graphs.

Table 3 shows that removing *explore_entity_relations* on SOCKG reduces F1 from 68.3% to 66.5% while increasing both iterations and wall time, confirming its role in guiding the agent efficiently through complex, heterogeneous schemas. We observe a similar F1 drop on DBLP (Table 4). Table 5 evaluates whether inspecting observed property usage helps when domain and range annotations are missing or unreliable. Since DBLP has a comparatively simpler schema, we ran this ablation on SOCKG. Adding *explore_property_usage* raises F1 from 54.0% to 62.7%, at the cost of additional time, money, and iterations. This shows that property-usage exploration recovers useful semantic information directly from graph data when ontological annotations are unavailable, though this benefit is not free. Finally, Table 6 evaluates how candidate retrieval quality affects downstream reasoning. On SOCKG, keyword-only retrieval lowers F1, more than doubles cost, and nearly doubles iterations, showing that combining semantic and lexical signals improves both candidate quality and search efficiency.

5 Conclusion

We presented KGVoyager, a KG-agnostic framework that answers natural language questions over arbitrary RDF knowledge graphs using only SPARQL endpoint access. Our feasibility study shows that KGVoyager's lightweight class index succeeds on 96.6% of okn.us endpoints under QLever, compared to 0% for the entity index required by the prior state-of-the-art system (GRASP). Across four domain-specific benchmarks and four backbone models, KGVoyager improves F1 over GRASP while generally reducing runtime, cost, and agent iterations, demonstrating that endpoint-only KGQA is viable and efficient for domain-specific knowledge graphs where curated resources are often unavailable.

Limitations

Although KGVoyager is KG-agnostic in that it requires no KG-specific ontology document, text-query examples, or graph-specific tools, its applicability is bounded by the structural assumptions built into the tool suite. The current implementation assumes that the target endpoint exposes classes and instance membership through standard RDFS/RDF typing patterns, making it best suited to RDFS-based or RDFS-compatible knowledge graphs. It is therefore not broadly applicable to open-domain graphs with substantially different modeling conventions, most notably Wikidata, whose entity, type, and property model is not RDFS-based. Supporting such graphs would likely require additional adapter tools or graph-specific normalization layers, which we leave for future work.

A second limitation is data availability: domain-specific text-SPARQL benchmarks remain scarce, and despite reaching out to many KG authors and maintainers, we could not gather larger datasets. More public domain-specific KGQA benchmarks are needed and we envision that automated benchmark generation could be promising.

Ethical Considerations

All knowledge graphs endpoints and benchmarks used in this work are publicly available (see links in our public repository[1]), and no private or personal data was collected or processed in any part of our experiments. We use existing artifacts only

[1] https://anonymous.4open.science/r/nl2sparql2-C171

for their intended research and benchmarking purposes; released artifacts are intended for research, evaluation, and reproducibility, with third-party resources remaining subject to their original terms.

Users of KGVoyager should be aware that, like any LLM-based system, it may generate syntactically valid SPARQL queries that are nonetheless semantically incorrect. In domains where query results inform decisions—such as scientific or policy contexts—outputs should be verified rather than trusted uncritically.

Finally, LLM inference carries an environmental cost. We note, however, that KGVoyager reduces monetary cost and wall-clock runtime by approximately 22% relative to the prior state-of-the-art system, partially mitigating this concern.

References

[1] Bilal Abu-Salih. 2021. Domain-specific knowledge graphs: A survey. *Journal of Network and Computer Applications*, 185:103076.

[2] Sushil Awale. 2026. DBLP-QuAD: A question answering dataset over the DBLP scholarly knowledge graph. Hugging Face dataset card. https://huggingface.co/datasets/awalesushil/DBLP-QuAD.

[3] Debayan Banerjee, Pranav Ajit Nair, Jivat Neet Kaur, Ricardo Usbeck, and Chris Biemann. 2022. Modern baselines for SPARQL semantic parsing. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2260–2265.

[4] Hannah Bast and Björn Buchhold. 2017. QLever: A query engine for efficient SPARQL+Text search. In *Proceedings of the 2017 ACM International Conference on Information and Knowledge Management*, pages 647–656.

[5] Jonathan Berant, Andrew Chlipala, Percy Liang, Bhuwan K Patel, and Luke Zettlemoyer. 2013. Semantic parsing on freebase from question-answer pairs. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1533–1544.

[6] Igor Bonifacic. 2025. Google's Gemini 3 Flash model outperforms GPT-5.2 in some benchmarks. Engadget. https://www.engadget.com/ai/googles-gemini-3-flash-model-outperforms-gpt-52-in-some-benchmarks-160000000.html.

[7] Gordon V. Cormack, Charles L. A. Clarke, and Stefan Büttcher. 2009. 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*, pages 758–759.

[8] Jacopo D'Abramo, Andrea Zugarini, and Paolo Torroni. 2025. Investigating large language models for text-to-SPARQL generation. In *Proceedings of the 4th International Workshop on Knowledge-Augmented Methods for Natural Language Processing*, pages 66–80.

[9] Papa Abdou Karim Karou Diallo, Samuel Reyd, and Amal Zouaq. 2025. FRASE: Structured representations for generalizable SPARQL query generation. *Preprint*, arXiv:2503.22144.

[10] Mohnish Dubey, Debayan Banerjee, Abdelrahman Abdelkawi, and Jens Lehmann. 2019. LC-QuAD 2.0: A large dataset for complex question answering over Wikidata and DBpedia. In *Proceedings of the 18th International Semantic Web Conference*, pages 69–78.

[11] Vincent Emonet, Jerven Bolleman, Severine Duvaud, Tarcisio Mendes de Farias, and Ana Claudia Sima. 2024. LLM-based SPARQL query generation from natural language over federated knowledge graphs. In *Proceedings of the Special Session on Harmonising Generative AI and Semantic Web Technologies (HGAIS 2024) co-located with ISWC 2024*, volume 3953 of *CEUR Workshop Proceedings*, pages 1–10.

[12] Sébastien Ferré. 2013. SQUALL: A controlled natural language as expressive as SPARQL 1.1. In *Natural Language Processing and Information Systems (NLDB)*, volume 7934 of *Lecture Notes in Computer Science*, pages 114–125.

[13] Aleksandr Gashkov, Aleksandr Perevalov, Maria Eltsova, and Andreas Both. 2025. SPARQL query generation with LLMs: Measuring the impact of training data memorization and knowledge injection. In *Proceedings of the 25th International Conference on Web Engineering*, pages 177–192.

[14] Graph.build. 2023. Ontology in graph models and knowledge graphs. Website: https://graph.build/resources/ontology.

[15] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutiérrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Urooj Rashid, Anisa Rula, Lukas Schmelzeisen, Juan F. Sequeda, Steffen Staab, and Antoine Zimmermann. 2021. Knowledge graphs. *ACM Computing Surveys*, 54(4):71:1–71:37.

[16] Kelly Hong, Anton Troynikov, and Jeff Huber. 2025. Context rot: How increasing input tokens impacts LLM performance. Technical report, Chroma.

[17] Xiangru Jian, Zhengyuan Dong, and M. Tamer Özsu. 2025. InteracSPARQL: An interactive system for SPARQL query refinement using natural language explanations. *Preprint*, arXiv:2511.02002.

[18] Endri Kacupaj, Joan Plepi, Kuldeep Singh, Harsh Thakkar, and Jens Lehmann. 2023. SPARQLGEN:

Generating SPARQL queries from natural language questions. In *Proceedings of the ACM Web Conference 2023*, pages 1131–1141.

[19] Saar Kuzi, Mingyang Zhang, Cheng Li, Michael Bendersky, and Marc Najork. 2020. Leveraging semantic and lexical matching to improve the recall of document retrieval systems: A hybrid approach. *Preprint*, arXiv:2010.01195.

[20] Harry Li, Gabriel Appleby, Camelia Daniela Brumar, Remco Chang, and Ashley Suh. 2024. Knowledge graphs in practice: Characterizing their users, challenges, and visualization opportunities. *IEEE Transactions on Visualization and Computer Graphics*, 30(1):584–594.

[21] Shicheng Liu, Sina Semnani, Harold Triedman, Jialiang Xu, Isaac Dan Zhao, and Monica Lam. 2024. SPINACH: SPARQL-based information navigation for challenging real-world questions. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 15977–16001.

[22] Divyansh Shankar Mishra, Abhinav Agarwal, B. P. Swathi, and K. C. 2022. Natural language query formalization to SPARQL for querying knowledge bases using Rasa. *Progress in Artificial Intelligence*.

[23] Mistral AI and All Hands AI. 2025. Devstral: Fine-tuning language models for coding agent applications. *Preprint*, arXiv:2509.25193.

[24] Naomi Moneypenny. 2026. Introducing OpenAI's GPT-5.4 mini and GPT-5.4 nano for low-latency AI. Microsoft Tech Community. https://techcommunity.microsoft.com/blog/azure-ai-foundry-blog/introducing-openai%E2%80%99s-gpt-5-4-mini-and-gpt-5-4-nano-for-low-latency-ai/4500569.

[25] Pauline Ochieng, Clifford Kyanda, and Carole Lawrence. 2020. PAROT: Translating natural language to SPARQL. *Expert Systems with Applications*, 150:113281.

[26] OKN.US. 2026. OKN registry: Knowledge graphs registered with the open knowledge network. Website: https://registry.okn.us/registry/.

[27] OpenAI. 2026. Introducing GPT-5.3-Codex.

[28] Julio Cesar Rangel, Tarcisio Mendes de Farias, Ana Claudia Sima, and Norio Kobayashi. 2024. SPARQL generation: an analysis on fine-tuning OpenLLaMA for question answering over a life science knowledge graph. In *Proceedings of the 15th International Conference on Semantic Web Applications and Tools for Health Care and Life Sciences (SWAT4HCLS 2024)*, volume 3890 of *CEUR Workshop Proceedings*, pages 36–45.

[29] Jhon Rayo, Raul de la Rosa, and Mario Garrido. 2025. A hybrid approach to information retrieval and answer generation for regulatory texts. In *Proceedings of the 1st Regulatory NLP Workshop (RegNLP 2025)*, pages 31–35.

[30] Md Rashad Al Hasan Rony, Uttam Kumar, Roman Teucher, Liubov Kovriguina, and Jens Lehmann. 2022. SGPT: A generative approach for SPARQL query generation from natural language questions. *IEEE Access*, 10:70712–70723.

[31] Nasim Shirvani-Mahdavi, Devin Wingfield, Juan Guajardo Gutierrez, Mai Tran, Zhengyuan Zhu, and Zeyu Zhang. 2025. A knowledge graph informing soil carbon modeling. In *Proceedings of the 25th International Conference on Web Engineering*, pages 226–241.

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

[33] Houcemeddine Turki, Mohamed Ali Hadj Taieb, and Mohamed Ben Aouicha. 2024. A decade of scholarly research on open knowledge graphs. *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation*, pages 446–456.

[34] Ricardo Usbeck, Ria Hari Gusmita, Axel-Cyrille Ngonga Ngomo, and Muhammad Saleem. 2018. 9th challenge on question answering over linked data (QALD-9) (invited paper). In *Joint Proceedings of the 4th Workshop on Semantic Deep Learning (SemDeep-4) and NLIWoD4: Natural Language Interfaces for the Web of Data (NLIWOD-4) and 9th Question Answering over Linked Data Challenge (QALD-9) co-located with 17th International Semantic Web Conference (ISWC 2018)*, volume 2241 of *CEUR Workshop Proceedings*, pages 58–64. CEUR-WS.org.

[35] Ricardo Usbeck, Xi Yan, Aleksandr Perevalov, Longquan Jiang, Julius Schulz, Angelie Kraft, Cedric Möller, Junbo Huang, Jan Reineke, Axel-Cyrille Ngonga Ngomo, Muhammad Saleem, and Andreas Both. 2024. QALD-10 - the 10th challenge on question answering over linked data: Shifting from DBpedia to Wikidata as a KG for KGQA. *Semantic Web*, 15(6):2193–2207.

[36] Sebastian Walter and Hannah Bast. 2025. GRASP: Generic reasoning and SPARQL generation across knowledge graphs. In *Proceedings of the 24th International Semantic Web Conference*, pages 271–289.

[37] Silei Xu, Shicheng Liu, Theo Culhane, Elizaveta Pertseva, Meng-Hsi Wu, Sina Semnani, and Monica Lam. 2023. Fine-tuned LLMs know more, hallucinate less with few-shot sequence-to-sequence semantic parsing over Wikidata. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 5778–5791.

[38] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023.

ReAct: Synergizing reasoning and acting in language models. In *Proceedings of the International Conference on Learning Representations*.

[39] Xiaoyu Yin, Dagmar Gromann, and Sebastian Rudolph. 2021. Neural machine translating from natural language to SPARQL. *Future Generation Computer Systems*, 117:510–519.

[40] Hamada Mohamed Abdelsamee Zahera, Ali Manzoor, Mohamed Sherif, Diego Moussallem, and Axel-Cyrille Ngonga Ngomo. 2024. Generating SPARQL from natural language using chain-of-thoughts prompting. In *Knowledge Graphs in the Age of Language Models and Neuro-Symbolic AI - Proceedings of the 20th International Conference on Semantic Systems, 17-19 September 2024, Amsterdam, The Netherlands*, Studies on the Semantic Web, pages 353–368. IOS Press.

[41] Xiaohu Zhu, Qian Li, Lizhen Cui, and Yongkang Liu. 2024. Large language model enhanced text-to-SQL generation: A survey. *Preprint*, arXiv:2410.06011.

A KGVoyager System Prompt and Tool Specification

You are a SPARQL query generation assistant. Your job is to generate an accurate SPARQL query that answers the user's question by dynamically discovering the knowledge graph's structure.
You operate in a think-act-observe loop: reason about what you need, invoke a tool, then interpret the results before deciding the next action. Follow this workflow:

1. SEARCH: Identify the classes, properties, and entities implied by the question. Search for candidate class and property URIs using natural-language terms. If the question mentions named entities, resolve them by first finding their class, then looking up instances by name. Use relevance scores and usage counts to select the most likely candidates.

2. EXPLORE: If the search results do not clearly reveal how properties connect entity types, explore property usage patterns to see actual domain/range types and example triples. If you need to understand the neighborhood of a specific entity, explore its relations.

3. CONSTRUCT AND TEST: Build the SPARQL query incrementally using the discovered URIs. Execute intermediate drafts to verify correctness. If results are unexpected, revisit earlier steps – you may have the wrong property, class, or triple pattern. Once the query returns the expected results, call the answer function. If you cannot find a satisfactory query, call cancel.

Keep in mind:

- Always strategically decide which tool call to make next. Include reasoning about your objective with the tool call whenever possible
- Never make a redundant tool call. You will be highly penalized if you repeat a tool call or perform tool calls of very little value (eg, verifying something already obvious). It's absolutely normal if you only use a subset of the available tools to solve the problem.
- Overall solving this with the least amount of tool calls but the perfect accuracy is the ultimate goal.
- Never return a SPARQL query before verifying it is correct and returns results that answer the question.
- For questions with a "True" or "False" answer the SPARQL query should be an ASK query.

Available knowledge graphs: {{ knowledge_graphs }}

Figure 5: KGVoyager system prompt.

SEARCH

Class Search `search_all_classes(search_term, limit)`

Search for classes in the Knowledge Graph by name or concept. Uses hybrid retrieval (semantic + keyword) to find relevant classes. Low-relevance results are automatically pruned.

Purpose: Use natural-language terms from the question to find candidate class URIs.

INPUT SCHEMA

search_term string REQ
The concept, keyword, or class name to search for. Utilizes semantic search to find relevant matches.

limit number OPT
Maximum number of results to return (default: 10).

OUTPUT FORMAT

TABLE WITH COLUMNS:

`class_uri` `label` `count` `score`

Instance Lookup `lookup_class_instances(class_uri, search_term, limit, mode)`

Retrieve instances of a specific class, optionally filtered by name.

Purpose: The primary way to resolve named entities mentioned in the question (e.g., 'France', 'Albert Einstein') to their URIs. First identify the entity's class via class search, then use this tool with the entity name as search_term.

INPUT SCHEMA

class_uri string REQ
The full absolute URI of the class. Do not use prefixed/abbreviated URIs.

search_term string OPT
Optional keyword to filter entities by label/name.

limit number OPT
Maximum number of entities to return (default: 10).

mode string OPT
Result mode: 'labels_only' (default) searches only entity label; 'all_attributes' searches across all literal values.

OUTPUT FORMAT

TABLE WITH COLUMNS:

`entity` `displayLabel` `searchProp` `searchVal`

Property Search `search_all_properties(search_term, limit)`

Search for properties in the Knowledge Graph by name or concept. Uses hybrid retrieval (semantic + keyword) to find relevant properties. Low-relevance results are automatically pruned.

Purpose: Find candidate property URIs. Use the domain and range columns to understand how the property connects different entity types.

INPUT SCHEMA

search_term string OPT
The concept, keyword, or property name to search for. Utilizes semantic search to find relevant matches.

limit number OPT
Maximum number of results to return (default: 10).

OUTPUT FORMAT

TABLE WITH COLUMNS:

`property_uri` `label` `aliases` `domain` `range` `count` `score`

Figure 6: KGVoyager tool specification: search tools.

EXPLORATION

Property Usage Exploration

`explore_property_usage(property_uri, limit, max_sample, include_class_prefixes)`

Analyze how a property is actually used in the graph by sampling real triples. Infers domain and range from actual data rather than ontology annotations (which are often missing or inaccurate).

Purpose: Differentiate between linguistically similar properties by seeing which entity types they actually connect, and verify that a candidate property relates the expected types before committing it to the query.

INPUT SCHEMA

property_uri string REQ

The full absolute URI of the property to analyze. Do not use prefixed/abbreviated URIs.

limit number OPT

Number of example usage patterns to return (default: 10).

max_sample number OPT

Maximum number of triples to sample for type analysis (default: 5000). Reduce if the endpoint times out.

include_class_prefixes string[] OPT

Optional list of URI prefixes to restrict the domain/range types shown. Useful to filter out noise when the knowledge graph spans many namespaces.

OUTPUT FORMAT

TABLE WITH COLUMNS:

`Domain` `Domain Label` `Range` `Range Label` `Example Subject` `Subject Label` `Example Object` `Object Label` `Count`

Entity Relations Exploration

`explore_entity_relations(entity_uri, num_ents, limit)`

Explore the relationships of a specific entity by listing all its incoming and outgoing connections.

Purpose: Useful when search alone has been inconclusive for resolving a relation mentioned in the question, as it reveals the full local graph structure around the entity.

INPUT SCHEMA

entity_uri string REQ

The full absolute URI of the entity to explore. Do not use prefixed/abbreviated URIs.

num_ents number OPT

Number of sample neighbors to show per property (default: 3). Keep low to avoid large outputs.

limit number OPT

Total number of results to return from query.

OUTPUT FORMAT

2 TABLES WITH COLUMNS:

Attributes (literals)

`property` `direction` `example_neighbors` `example_neighbors_labels`

Relations (URIs)

`property` `direction` `example_neighbors` `example_neighbors_labels`

Figure 7: KGVoyager tool specification: exploration tools.

EXECUTION

Execute Query

`execute_query(query)`

Executes a SPARQL query against the Knowledge Graph and returns the results.

Purpose: Serves a dual role: running the final query to obtain the answer, and providing iterative feedback throughout reasoning. Use it to test intermediate query drafts and diagnose errors in partial queries.
Use LIMIT to prevent fetching too much data for verification queries. After verification, the final query you return should be consistent with the question on whether LIMIT is needed.

Output Format: Table with the query result columns, or an error message if the query fails.

INPUT SCHEMA

query string REQ

The valid SPARQL query string to execute.

OUTPUT FORMAT

Table with the query result columns, or an error message if the query fails.

FINISH

Answer Question

`answer(kg, sparql, answer)`

Provide your final SPARQL query and answer to the user question based on the query results. This function will stop the generation process.

INPUT SCHEMA

kg string REQ

The knowledge graph on which the final SPARQL query needs to be executed

sparql string REQ

The final SPARQL query

answer string REQ

The answer to the question based on the SPARQL query results

OUTPUT FORMAT

Stops the generation process and returns the final answer.

Figure 8: KGVoyager tool specification: execution tools.