Toward Effective and Reliable LLM Agents via Dynamic Ontology

Xiaohui Zhang[1], **Zequn Sun**[1], **Chengyuan Yang**[1], **Yuanning Cui**[2], **Lingbing Guo**[3], **Wei Hu**[1,4]

[1]State Key Laboratory for Novel Software Technology, Nanjing University, China
[2]School of Computer Science, Nanjing University of Information Science and Technology, China
[3]School of Intelligence Science and Technology, Nanjing University, China
[4]National Institute of Healthcare Data Science, Nanjing University, China
{xhzhang, cyyang}.nju@gmail.com, {sunzq, lbguo, whu}@nju.edu.cn, yncui@nuist.edu.cn

Abstract

Large language model (LLM) agents rely heavily on knowledge encoded in model parameters or presented as unstructured context. In domain-specific tasks, this leaves important semantic connections implicit. This often results in incomplete evidence use and brittle multi-step decisions. Ontologies offer a way to externalize domain concepts and relations as machine-interpretable structures, but constructing task-usable ontologies traditionally requires substantial effort from domain experts and is difficult to scale. Automatic construction is also challenging: an ontology that appears semantically plausible may not contain the relational structures needed for actual decision making. We present OaK, an ontology-as-a-kernel framework that dynamically constructs and refines task-oriented ontologies for LLM agents. Given task requirements and training data, OaK constructs an ontology and its knowledge graph, generates task-adaptation functions for graph reasoning, and uses judge feedback to iteratively refine both. By making relevant concepts and relations explicit, the ontology grounds knowledge retrieval and multi-step decision making. We evaluate OaK on TravelPlanner, CRMArenaPro, and ToolQA. Results show that OaK improves standard LLM agents, strengthens evidence grounding, and boosts the reliability of multi-step reasoning.

1 Introduction

Large language models (LLMs) have demonstrated strong capabilities in understanding and generating natural language, as well as in knowledge-intensive reasoning. LLMs address diverse tasks through instructions and in-context examples. LLM agents extend these capabilities from response generation to goal-directed task execution by coupling an LLM with external components such as retrieval systems (Zhu et al. 2025), tools and memory (Zhang et al. 2026b; Yang et al. 2026). A typical agent runs a loop that repeatedly interprets the goal and plans, then acts through tool calls or retrieval and observes the result until the task is done. Recent agent architectures further incorporate mechanisms such as reflection and procedural memory, along with modular workflow optimization (Shinn et al. 2023; Fang et al. 2026; Zhang et al. 2025; Shang et al. 2025). These developments make LLM agents a general framework for tasks that require command execution and multi-step reasoning beyond a single response.

Despite these advances, the central challenge is no longer only whether an agent can act, but whether its behavior remains controllable and trustworthy as execution becomes longer and more autonomous. At each intermediate step, the agent decides what to retrieve and which tool to call with which arguments, so an error in any of these choices can propagate to later steps. Consequently, final-answer accuracy alone reveals neither the supporting evidence nor the justification of tool calls. It also hides the influence of memory and the origin of execution failures (Wang et al. 2026). Recent studies of tool-using agents expose this gap: ToolEmu identifies realistic long-tail safety failures in high-stakes tool settings, while AgentDojo shows that untrusted tool outputs can manipulate agent behavior through prompt injection (Ruan et al. 2024; Debenedetti et al. 2024). Reflection, procedural memory, and workflow optimization can improve planning or reuse, but they generally leave the set of admissible concepts and action sequences implicit (Madaan et al. 2023; Shinn et al. 2023; Zhang et al. 2025; Fang et al. 2026). Prompt instructions and tool descriptions therefore guide an agent's behavior without providing an enforceable contract for what it may execute or how its results should be checked. Reliable agents must constrain actions during execution so that outputs connect to supporting evidence and faulty steps can be identified and fixed.

These requirements point to a missing layer between the LLM and the tools it controls: a task-oriented representation that makes both domain semantics and executable behavior explicit. Ontologies provide a natural basis because they organize concepts and relations in a machine-interpretable form (Gruber 1993; Hogan et al. 2021). However, a conventional ontology is primarily descriptive: it specifies what exists in a domain, but not necessarily what an agent may do. It also leaves open how an operation should be parameterized and which conditions must hold before its result is accepted. We therefore use ontology in an operational sense. In our setting, requirements on data representation are encoded in schema declarations, whereas requirements on computation and workflow are enforced by the control flow of functions. Such an ontology is not a passive knowledge description. It serves as a semantic and procedural contract that bounds what the agent may do and keeps execution open to inspection.

We propose OaK, a dynamic ontology-as-a-kernel framework for LLM agents. Here, dynamicity refers to task-conditioned construction. For each task, OaK automatically constructs the schema and typed reasoning functions needed to solve it. It then instantiates a corresponding knowledge

graph from task data, and refines the schema and functions with training examples and downstream task feedback before freezing the resulting kernel for inference. OaK packages the task interface into a kernel

$$\mathcal{K} = (\mathcal{S}, \mathcal{F}),$$

where $\mathcal{S}$ is a task-oriented schema that defines the domain concepts available to the agent, together with their properties and relations. The functions $\mathcal{F}$ define how these typed elements can be used through executable procedures for retrieval, filtering, traversal, projection, aggregation, and multi-step reasoning. Given $\mathcal{S}$, OaK instantiates a schema-guided knowledge graph $\mathcal{G}$ from the task data. This graph provides the evidence on which the functions operate. During inference, a ReAct agent interprets a query to select a function and bind its typed arguments, then uses the function to reason the final answer. The kernel mediates access to the available evidence and operations. During construction, OaK evaluates these executions and uses task feedback to refine $\mathcal{S}$ and $\mathcal{F}$ before the resulting kernel is applied to unseen queries.

We evaluate OaK on TravelPlanner (Xie et al. 2024), CRMArenaPro (Huang et al. 2026), and ToolQA (Zhuang et al. 2023). These datasets differ in task setting and reasoning requirements. Results and analyses show that OaK improves standard LLM agents and strengthens evidence grounding for multi-step reasoning.

Our contributions are summarized as follows:

- We introduce *ontology as a kernel* for LLM agents. It couples a task-oriented schema with typed reasoning functions and a schema-guided evidence graph.
- We develop an automated pipeline that constructs a verified schema, instantiates its knowledge graph, and compiles generic operators into executable domain functions.
- We propose judge-driven refinement, which diagnoses and repairs schema and function failures across construction rounds using official task scores and execution trajectories.
- Experiments on TravelPlanner, CRMArenaPro, and ToolQA show consistent gains across two backbones, while ablations confirm the importance of function composition, the function module, and iterative refinement.

2 Related Work

LLM for agents. LLMs serve as general-purpose reasoning and decision-making components in agents. They enable agents to interpret goals and decompose tasks, adapting their behavior from intermediate observations. ReAct established an influential paradigm that interleaves reasoning and acting within a single execution trajectory (Yao et al. 2023). Beyond reflection, MemP distills successful trajectories into reusable procedural memory, while ReCode represents planning and acting through recursively generated programs (Fang et al. 2026; Yu et al. 2025). AFlow and AgentSquare further automate agent design by searching over code-represented workflows or modular architectures composed of planning and memory components (Zhang et al. 2025; Shang et al. 2025). These approaches progressively move agent control out of the model's parameters into feedback loops and reusable procedures. They primarily improve how an agent organizes and adapts its control process, whereas OaK complements this line of work by structuring the task interface itself through an explicit domain schema and executable reasoning functions.

Reliable agents. Reliability in LLM agents extends beyond endpoint accuracy: it asks whether each step stays controlled and each decision remains grounded in evidence, and whether failures can be traced to a cause (Wang et al. 2026). Feedback-based approaches such as Self-Refine and Reflexion use model-generated critique or execution outcomes to revise responses and future decisions (Madaan et al. 2023; Shinn et al. 2023). ToolEmu and AgentDojo expose complementary risks by identifying realistic failures in high-stakes tool settings and evaluating indirect prompt injection through untrusted observations (Ruan et al. 2024; Debenedetti et al. 2024). AgentSpec addresses execution control by expressing safety requirements as structured rules, combining runtime triggers and predicates with enforcement mechanisms (Wang, Poskitt, and Sun 2025). In parallel, knowledge editing aims to correct or update the facts stored in model parameters rather than externalizing them (Zhang et al. 2026a). These approaches provide feedback, policy enforcement, or execution records. However, they typically treat task rules and allowable operations as fixed inputs or leave them distributed across prompts and implementation code. As a result, the interface between task requirements and agent execution is difficult to inspect and adapt. OaK instead makes this interface explicit as a task-specific kernel, coupling a schema with typed reasoning functions and using task scores and execution trajectories to refine both.

Graph for agents. Ontologies and knowledge graphs provide machine-interpretable structures for organizing domain concepts and their properties and relations (Gruber 1993; Hogan et al. 2021). Recent work integrates LLMs with knowledge graphs to support the structured acquisition and representation of knowledge, as well as reasoning over it (Pan et al. 2024). GraphRAG and G-Retriever further use graph structure to retrieve globally relevant or relational evidence for generation and question answering (Edge et al. 2024; He et al. 2024). These approaches primarily treat the graph as an external knowledge and retrieval layer. This can improve evidence access but does not itself specify reusable task-level computations for an agent. OaK instead constructs a task-oriented schema and a catalog of typed reasoning functions. It grounds their execution in a data-dependent knowledge graph. It jointly refines the schema and functions using downstream task feedback. In OaK, dynamicity therefore refers to constructing this task-specific schema and function catalog for the task at hand. It then instantiates the corresponding graph from task data and refines the kernel before freezing it for inference. This design makes the graph not only a source of retrieved knowledge, but also the grounded substrate on which the agent's semantic and procedural interface operates.

3 OaK Framework

3.1 Overview

OaK builds a dynamic task-oriented *ontology kernel* and uses it to mediate between an LLM agent and domain tasks.

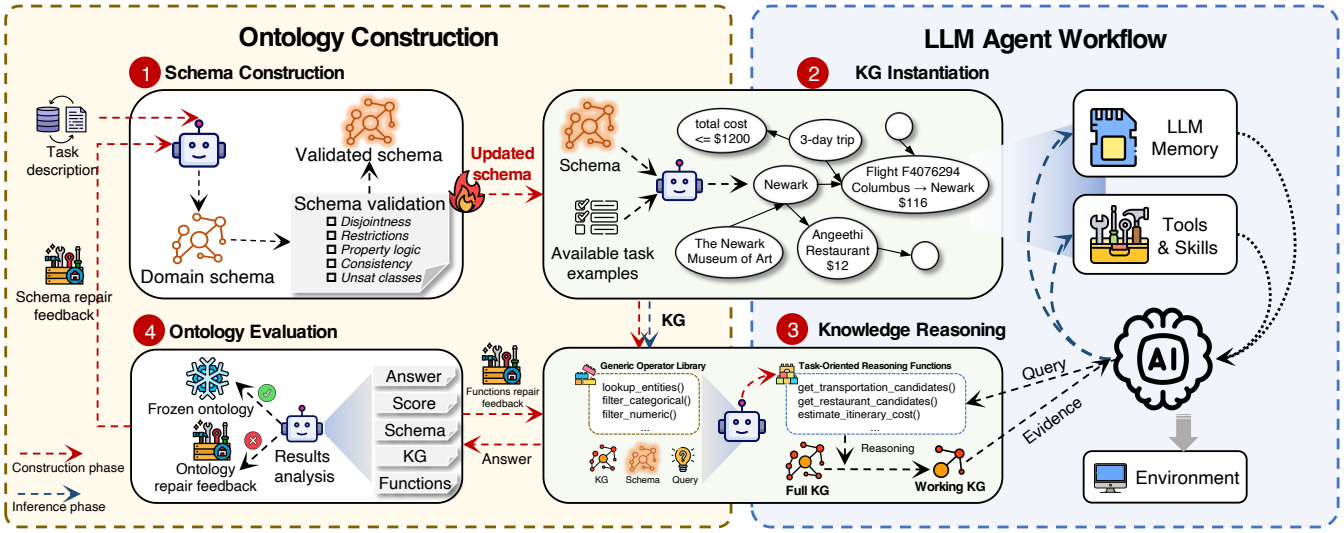

Figure 1: Overview of OaK. The construction stage (left) runs a refinement loop over the ontology kernel. The inference stage (right) freezes the kernel and lets a ReAct agent solve unseen queries by calling functions as tools.

The kernel has two main components, namely the schema $\mathcal{S}$ and the function set $\mathcal{F}$. The schema $\mathcal{S}$ bounds what can be expressed and the functions $\mathcal{F}$ bound what can be computed. Once frozen, the kernel is the only channel through which the agent reaches the data. It cannot name a concept or invoke a computation the kernel does not declare.

OaK runs in two stages. The construction stage runs on training data, where it builds the kernel and refines it in a loop. The inference stage freezes the kernel and applies it to unseen queries. Figure 1 shows the full loop.

3.2 Ontology Construction Loop

The construction stage runs a four-step loop over the kernel. At the start of each round t, OaK draws a fresh random sample $D_t \subseteq D^{\text{tr}}$ from the training set and uses it to drive that round. Each instance in D_t is a complete example (q, C_q) that pairs a query q with the reference corpus C_q needed to answer it. We write $Q_t = \{q\}$ for its queries and $C_t = \{C_q\}$ for the accompanying corpora. Resampling per round exposes the loop to varied data and keeps the schema and functions from overfitting to a fixed subset. We denote by $\mathcal{S}_t$ and $\mathcal{F}_t$ the schema and functions produced in round t.

Step 1: Schema construction. This step proceeds in three phases: requirement analysis, schema drafting, and formal verification.

Requirement analysis. An LLM reads the task description T together with the round sample D_t and produces a requirement specification:

$$R = \text{Analyze}(T, D_t).$$

R records the task scope, the key entities and relations, and the task constraints. It gives the schema a precise target rather than a free form brief.

Schema drafting. The model then drafts the schema from R together with the previous round's schema feedback:

$$\mathcal{S}_t = \text{Draft}(R,\ \psi^{\mathcal{S}}_{t-1}),$$

where $\psi^{\mathcal{S}}_{t-1}$ is the schema level repair feedback from the previous round (empty in the first round). It firstly enumerates entity types with their properties, and then declares the typed relations among them. Each entity type is assigned an identity field that serves as its primary key, and each relation names its source and target entity types.

Formal verification. A drafted schema can look semantically reasonable yet be logically flawed inside, and such a flaw would propagate into the graph. We therefore verify the draft before it is used. Our schema is lightweight—a YAML or JSON document of named entity types, properties, and typed relations that the agent reads directly, not the full axiomatic machinery of a formal ontology. This makes it easy for the model to consume, but a logical reasoner cannot check it as is. We therefore encode it as an equivalent OWL ontology and run HermiT (Glimm et al. 2014):

$$\text{HermiT}(\text{OWL}(\mathcal{S}_t)) \rightarrow \{\text{consistent},\ \text{unsatisfiable classes}\}.$$

HermiT checks five kinds of logical validity—disjointness, restriction, property-level, and global consistency, plus unsatisfiable classes. We detail them in Appendix A. A schema that fails any check is sent back to drafting with the reasoner's counterexamples, and the loop retries.

Step 2: Knowledge Graph instantiation. Under the current schema, OaK instantiates a knowledge graph from the reference corpora of the round sample. To improve extraction quality and avoid exceeding the context limit of the language-model extractor, graph construction uses a chunk–map–merge pipeline. The corpus C_t is first partitioned into token-bounded chunks $\{c_1, \ldots, c_n\}$ that fit the extractor's input budget. An LLM extractor $\Phi_{\mathcal{S}_t}$ maps each chunk c_i to a set of typed entity and relation candidates according to $\mathcal{S}_t$, and a merge operator $\bigsqcup$ reconciles these sets into one graph:

$$\mathcal{G}_t = \text{Build}(\mathcal{S}_t, C_t) = \bigsqcup_{i=1}^{n} \Phi_{\mathcal{S}_t}(c_i),$$

Because $\Phi_{\mathcal{S}_t}$ runs on each chunk independently, one real-world entity may surface as several candidates across chunks. The merge operator $\bigsqcup$ resolves these duplicates through the schema-declared primary key. Each candidate entity e is assigned a *key signature*

$$\kappa(e) = \big(\tau(e),\, \pi_{\mathrm{pk}}(e)\big),$$

where $\tau(e)$ is its entity type and $\pi_{\mathrm{pk}}(e)$ is its primary-key value. Two candidates denote the same real-world object exactly when their key signatures agree:

$$e_i \sim e_j \iff \kappa(e_i) = \kappa(e_j).$$

Since $\sim$ is an equivalence relation, $\bigsqcup$ collapses each equivalence class $[e]_\sim$ into one canonical node with a single deterministic identifier. Relations are then re-attached to these canonical endpoints. This process also collapses duplicate relations produced across chunks.

Step 3: Knowledge reasoning. This step composes the generic operators into a catalog of domain-adapted functions. A function compiles a task's recurring reasoning steps into a single typed call. This frees the agent from multi-step planning. With less reasoning done inside the model, there is less room for hallucination. An LLM-based composer assembles the catalog $\mathcal{F}_t$:

$$\mathcal{F}_t = \mathrm{Compose}\big(\mathcal{O},\, \mathcal{S}_t,\, \mathcal{G}_t,\, Q_t,\, \psi^{\mathcal{F}}_{t-1}\big),$$

where $\psi^{\mathcal{F}}_{t-1}$ is the function-level repair feedback from the previous round (empty in the first round). By reading how queries in Q_t resolve over $\mathcal{G}_t$, the composer identifies the recurring reasoning patterns the functions must cover.

Each pattern is realized as one function $f \in \mathcal{F}_t$, with typed inputs and outputs and a realization r_f over $\mathcal{O}$. The generic library $\mathcal{O}$ provides basic graph and data operations, including entity lookup, relation traversal, property projection, constraint filtering, and aggregation. Appendix B gives the complete operator signatures and descriptions. A function is grounded in $\mathcal{O}$ in one of three ways: it may *compose* several operators into a pipeline, *specialize* an operator by fixing or reinterpreting its parameters, or *adapt* an operator with lightweight pre- or post-processing. Each function is tested on $\mathcal{G}_t$ to confirm it is executable, and the validated catalog is exposed to the agent as callable tools.

With the kernel in place, a ReAct agent runs on the queries:

$$A_t = \mathrm{Agent}(Q_t, \mathcal{S}_t, \mathcal{G}_t, \mathcal{F}_t).$$

For each query in Q_t, the agent invokes the functions $\mathcal{F}_t$ as tools, binds their typed arguments according to the schema $\mathcal{S}_t$, and executes them over the graph $\mathcal{G}_t$ to retrieve the evidence needed to answer the query. The trajectory A_t collects the resulting operator traces, function outputs and final answers, which the next step reviews.

Step 4: Ontology evaluation. The final answers in A_t are first scored according to the dataset's evaluation protocol:

$$\mathbf{s}_t = \mathrm{Eval}(A_t).$$

A judge model then reviews the whole kernel together with the trajectory and its task scores (Zheng et al. 2023):

$$\psi_t = \mathrm{Judge}(\mathcal{S}_t, \mathcal{G}_t, \mathcal{F}_t, A_t, \mathbf{s}_t).$$

ψ_t is a set of repair suggestions over the round's artifacts. Let $\mathcal{U}_t = E_t \cup R_t \cup \mathcal{F}_t$ collect the entity types E_t, relation types R_t, and functions $\mathcal{F}_t$ of round t, and let $\mathrm{Act} = \{\mathsf{add}, \mathsf{delete}, \mathsf{modify}\}$. Each suggestion is a tuple

$$\sigma = (u,\, a,\, \delta,\, \rho), \qquad u \in \mathcal{U}_t,\ a \in \mathrm{Act},$$

where u is the target artifact, a the action, δ the proposed definition or patch, and ρ the diagnosed reason. The report is

$$\psi_t = \{\sigma_1, \ldots, \sigma_k\},$$

partitioned by target into $\psi_t = \psi^E_t \cup \psi^R_t \cup \psi^{\mathcal{F}}_t$, with the schema level part $\psi^{\mathcal{S}}_t = \psi^E_t \cup \psi^R_t$ feeding Step 1 and the function level part $\psi^{\mathcal{F}}_t$ feeding Step 3 of round $t{+}1$.

OaK then applies the repair and starts the next round. The loop stops when the judge finds no blocking fault or the iteration budget runs out.

3.3 Ontology-driven LLM Inference

Inference runs on the unseen test set, with no further edits to the kernel. After the loop, OaK freezes the schema $\mathcal{S}^*$ and the functions $\mathcal{F}^*$ from the final round. For a test query q, it builds an inference graph:

$$\mathcal{G}_q = \mathrm{Build}(\mathcal{S}^*, C_q),$$

where C_q is the corpus that accompanies q in test set.

A ReAct agent then solves q with the frozen kernel:

$$a = \mathrm{Agent}(q, \mathcal{S}^*, \mathcal{G}_q, \mathcal{F}^*).$$

As in construction, the agent invokes $\mathcal{F}^*$ as tools, binds their typed arguments under $\mathcal{S}^*$, and executes them over $\mathcal{G}_q$. Unlike free-form agent reasoning, inference here is routed through a task-refined kernel: $\mathcal{S}^*$ closes off concepts and relations the agent may invoke, and $\mathcal{F}^*$ restricts its computations to typed, executable compositions over schema-constrained graph instances. The agent therefore operates inside a verified semantic space: answers are meant to trace to grounded evidence rather than to model-internal inference. This can reduce reasoning errors and unsupported inferences in open-ended generation.

4 Experiments

4.1 Tasks

We choose these three benchmarks. They cover complementary agent settings, including multi-step planning, CRM workflow execution, and compositional tool use over heterogeneous corpora. This diversity lets us evaluate OaK's applicability across multiple task settings. For every dataset, we follow the official evaluation protocol and metrics.

- **TravelPlanner** (Xie et al. 2024) evaluates multi-day plans that jointly arrange transportation, meals, attractions, and accommodation. Commonsense (CS) and hard-constraint (HC) measure overall feasibility and explicit requirement compliance, respectively. Within each family, micro is the fraction of individual constraints satisfied, macro is the fraction of plans meeting all applicable constraints, and final requires both families to be fully satisfied.

LLM	Method	CS		HC		Final
		Micro	Macro	Micro	Macro	
DeepSeek-v4-flash	ReAct	81.60	19.10	44.29	36.40	15.30
	AFlow	79.31	31.70	43.60	42.70	29.10
	MemP	82.78	55.50	**63.91**	59.50	51.50
	ReCode	**86.60**	50.90	48.67	37.60	48.20
	AgentSquare	79.10	30.40	49.40	47.80	27.70
	OaK	86.11	**58.60**	59.31	**61.57**	**55.90**
GPT-4o-mini	ReAct	79.30	14.20	29.20	17.80	4.50
	AFlow	78.23	12.60	31.09	17.70	3.00
	MemP	68.43	17.40	27.29	19.50	16.30
	ReCode	**88.59**	22.00	54.50	26.50	15.00
	AgentSquare	72.06	13.50	21.11	12.10	5.40
	OaK	82.40	**30.80**	**68.00**	**48.30**	**19.70**

Table 1: TravelPlanner results on DeepSeek-v4-flash and gpt-4o-mini. All entries are percentages (%). CS = Commonsense Constraint Pass Rate; HC = Hard Constraint Pass Rate; Final = Final Pass Rate. The best value in each column is in **bold** and the second best is underlined.

- **CRMArenaPro** (Huang et al. 2026) evaluates Customer Relationship Management (CRM) tasks in synthetic B2B and B2C organizations. These represent company-oriented and individual-customer processes, respectively. Workflow, Policy, and Database use exact match, whereas Text uses exact match for discrete answers or token-level F1 for free-form answers.
- **ToolQA** (Zhuang et al. 2023) evaluates compositional tool use over heterogeneous external corpora using normalized exact match.

4.2 Implementation Details

We run every method on two LLM backbones, `DeepSeek-v4-flash` and `gpt-4o-mini`. These backbones are used throughout Step 1's requirement analysis and schema drafting, Step 2's knowledge graph instantiation, Step 3's function composition and knowledge reasoning. The ontology evaluator is held fixed across all settings and uses `claude-sonnet-4.6`. Each construction round draws a fresh sample covering 20% of the training split, and the loop runs for at most 5 iterations. The ReAct agent is capped at 20 steps per query.

4.3 Main Results

TravelPlanner. In Table 1, OaK achieves the highest final pass rate under both backbones and leads all macro-level metrics on the test sets. On the DeepSeek-v4-flash backbone, ReCode is strongest on commonsense micro scores and MemP remains competitive on hard-constraint micro scores. This indicates that satisfying individual constraints does not necessarily produce a jointly valid plan. TravelPlanner requires decisions about dates, cities, transportation, accommodation, dining, and budgets to remain compatible across multiple days. This makes cross-component coordination essential. OaK explicitly represents these entities and constraints in its schema. Its knowledge graph connects them to the available options. Its schema-adapted functions further integrate candidate retrieval, budget calculation, and constraint checking into a coherent planning procedure. This combination helps OaK preserve dependencies across the complete itinerary, explaining its stronger macro and final performance rather than merely improving isolated constraint satisfaction.

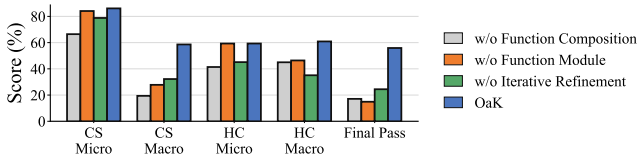

Figure 2: Ablation results on TravelPlanner.

CRMArenaPro. On CRMArenaPro, Table 2 reports the highest Avg. for OaK in both B2B and B2C organizations under both backbones, and leads in nearly all category-level comparisons. Its advantages are particularly clear on Policy and Database tasks. They require agents to combine business-rule compliance with accurate access to records. Workflow-oriented baselines do well on some Workflow tasks, and MemP shows the value of reusable procedures. However, their performance drops when the task requires selecting the correct records, fields, and relations. OaK addresses this limitation by explicitly representing CRM record types, fields, and their relations. It also grounds reusable functions for cross-table queries, rule checks, and numerical calculations in this representation. Its larger advantage with `gpt-4o-mini`, especially on Database tasks, further suggests that schema-grounded execution reduces the amount of multi-step data reasoning that must be performed by the backbone model alone.

ToolQA. For ToolQA, Table 3 reports the highest weighted-average performance for OaK under both backbones, with first-place results on most subset–backbone combinations. ToolQA requires agents to compose multiple operations over heterogeneous corpora, including field lookup, record filtering, relation traversal, and aggregation. Success therefore depends on coordinating the order of operations with the semantics of domain-specific fields and relations, rather than selecting tools independently. OaK addresses this requirement by mapping each question to typed and reusable procedures whose arguments and operations are constrained by an explicit domain schema and grounded knowledge graph. Compared with methods that optimize global workflows or procedural reuse without explicitly representing data relations, OaK more reliably connects each operation with the appropriate records and relations.

Additional robustness results over multiple random seeds are provided in Appendix C.

4.4 Ablation Study

We compare the full OaK with three variants using DeepSeek-v4-flash. We provide the ablations using gpt-4o-mini in Appendix D. *w/o Function Composition* removes the composition step and exposes the generic operators $\mathcal{O}$ directly to the agent.

LLM	Method	B2B					B2C				
		Workflow	Policy	Text	Database	Avg.	Workflow	Policy	Text	Database	Avg.
DeepSeek-v4-flash	ReAct	92.50	47.50	31.57	71.56	60.78	88.75	45.62	43.47	65.62	60.87
	AFlow	96.25	46.25	33.87	48.44	56.20	93.75	47.50	41.84	33.75	54.21
	MemP	96.25	55.62	35.74	78.12	66.44	95.00	61.25	**43.77**	69.06	67.27
	ReCode	26.25	39.38	10.13	24.06	24.96	27.50	34.38	11.71	24.06	24.41
	AgentSquare	95.00	46.88	31.10	50.94	55.98	88.75	56.25	36.37	44.06	56.36
	OaK	**97.50**	**81.88**	**39.46**	**94.69**	**78.38**	**100.00**	**71.88**	39.87	**89.06**	**75.20**
GPT-4o-mini	ReAct	27.50	27.50	7.73	17.81	20.14	15.00	33.13	13.80	23.12	21.28
	AFlow	66.25	46.25	5.61	25.62	35.93	53.75	41.25	8.80	24.38	32.04
	MemP	71.25	45.63	30.73	49.38	49.24	82.50	44.38	31.11	41.56	49.89
	ReCode	21.75	38.12	12.00	16.88	22.18	23.25	33.75	11.04	16.56	21.15
	AgentSquare	16.25	48.13	5.01	31.56	25.24	17.50	40.62	7.88	28.75	23.69
	OaK	**83.75**	**55.00**	**32.27**	**84.69**	**63.93**	**87.50**	**55.63**	**40.80**	**86.56**	**67.62**

Table 2: CRMArenaPro results on DeepSeek-v4-flash and gpt-4o-mini. All entries are percentages (%); Workflow, Policy, Text, and Database are task-category scores. Avg. is the equal-weight mean of Workflow, Policy, Text, and Database. The best value in each column is in **bold** and the second best is underlined.

LLM	Method	Flight	Coffee	Airbnb	DBLP	Yelp	Avg.
DeepSeek-v4-flash	AFlow	57.78	49.28	53.89	35.56	70.00	53.18
	AgentSquare	46.67	41.06	50.56	28.89	63.89	46.06
	ReAct	37.78	48.79	48.33	43.18	82.20	51.96
	ReCode	28.95	18.40	46.15	12.50	57.14	32.21
	MemP	55.56	48.79	54.44	35.00	82.22	55.02
	OaK	**66.67**	**64.25**	**58.33**	**44.94**	**88.76**	**64.58**
GPT-4o-mini	AFlow	30.00	38.16	52.78	30.00	57.78	41.64
	AgentSquare	10.00	40.58	42.78	16.11	35.56	29.34
	ReAct	**37.38**	63.54	**72.28**	18.92	58.18	50.45
	ReCode	17.76	62.39	52.86	12.90	42.74	38.45
	MemP	12.22	30.43	46.67	17.22	33.33	28.05
	OaK	31.10	**82.60**	54.40	**43.30**	**81.70**	**59.32**

Table 3: ToolQA results on the five subsets, using DeepSeek-v4-flash and gpt-4o-mini as backbones. All entries are percentages (%) and report exact-match accuracy. Avg. is the weighted average across subsets. The best value in each column is in **bold** and the second best is underlined.

w/o Function Module removes $\mathcal{F}$ entirely and lets the LLM reason over the full graph, retaining only minimal retrieval operations when the graph cannot be placed directly in context. *w/o Iterative Refinement* executes only the first round of the construction loop, without subsequent updates.

TravelPlanner. In Figure 2, removing any component substantially reduces the final pass rate. The largest degradation is caused by removing the function module or function composition. Although direct graph access can preserve individual hard-constraint decisions, it does not provide reusable procedures for coordinating transportation, accommodation, dining, and budget constraints across a complete plan. Without function composition, the agent must reconstruct these dependent operations for each query. This further weakens global plan consistency. The one-round variant also performs substantially worse. This shows that iterative refinement is necessary for repairing missing constraints and incomplete reasoning procedures.

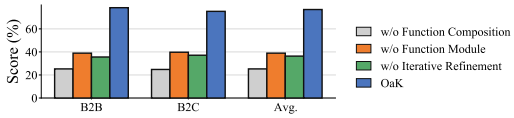

Figure 3: Ablation results on CRMArenaPro.

CRMArenaPro. In Figure 3, the full OaK model outperforms all variants on both B2B and B2C settings. Removing function composition causes the largest degradation because the agent must reconstruct cross-table queries, business-rule checks and numerical calculations from generic operators. Removing the function module also reduces performance by eliminating reusable procedures for recurring CRM operations, even when graph access is available.

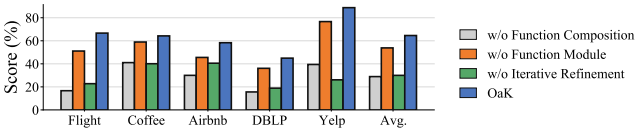

Figure 4: Ablation results on ToolQA.

ToolQA. Across all subsets in Figure 4, OaK outperforms all variants. The function-module ablation is the strongest variant but still remains clearly below the full model. This indicates that graph access alone cannot replace schema-adapted procedures. Removing function composition substantially harms performance because the agent must reconstruct multi-step sequences of lookup, filtering, relation traversal, projection, and aggregation for each question.

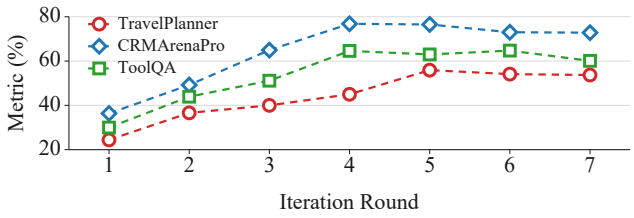

Figure 5: Loop progress on TravelPlanner, CRMArenaPro, and ToolQA using DeepSeek-v4-flash.

```
    name: Hotel
    primary_key: name
    properties:
      - name: name
      - name: city
      - name: room_type
      - name: price_per_night
+     - name: minimum_nights
+     - name: maximum_occupancy
```

Figure 6: Schema repair for the `Hotel` entity type.

4.5 Construction Loop Analysis

We record each benchmark's primary aggregate metric after every construction round under a fixed inference protocol. Across all three benchmarks, Figure 5 shows rapid improvement during the early rounds. This indicates that judge-guided updates quickly fix missing constraints and graph-mapping defects while closing function-level gaps. The curves largely plateau after rounds 4–5, suggesting that the five-round budget captures most of the benefit of iterative refinement. The small fluctuations in later rounds likely result from fine-tuning of the schema and functions, together with the fresh training sample used in each round.

4.6 Case Study

We illustrate how judge feedback coordinates schema and function level repair in TravelPlanner.

Schema repair. In an early construction round, the judge found that the `Hotel` entity exposed visible attributes such as price and room type but omitted the `minimum_nights` and `maximum_occupancy` constraints. Without these fields, the agent could select a hotel that appeared inexpensive and suitable but was invalid because it required a longer stay or could not accommodate the entire party. The judge therefore issued an *add* suggestion for the `Hotel` entity, adding both constraint fields as shown in Figure 6. The updated schema made these requirements explicit and allowed the agent to incorporate them during planning.

Function repair. The schema update alone was insufficient as the `get_accommodation_candidates` function did not yet use the newly exposed constraints. The ontology evaluator traced remaining failed plans and issued a *modify* suggestion that added filters for number of people and minimum stay, as shown in Figure 7. After the repair, a hotel was retained only when its maximum occupancy covered the party size and its minimum-night requirement fit the planned stay.

```
    def get_accommodation_candidates(
      hotels, budget, people_count,
      stay_nights, room_type):
      hotels = filter_numeric(
        hotels, "price_per_night",
        "<=", budget)
      hotels = filter_categorical(
        hotels, "room_type", room_type)
+     hotels = filter_numeric(
        hotels, "maximum_occupancy",
        ">=", people_count)
+     hotels = filter_numeric(
        hotels, "minimum_nights",
        "<=", stay_nights)
    return rank_and_select(hotels)
```

Figure 7: Function repair for `get_accommodation_candidates`.

4.7 Cost Analysis

We compare OaK with the baselines on TravelPlanner using runtime and token counts as resource proxies and final pass rate as the task-quality measure. In Figure 8, OaK achieves the highest final pass rate while using fewer input tokens than ReAct and MemP. This input reduction suggests that schema-adapted functions reduce the need to reconstruct graph operations from a large context. The benefit comes with higher runtime and output-token usage than several baselines, mainly because OaK instantiates a query-specific graph before planning. When a graph can be reused across related queries, this construction cost can be amortized. Overall, OaK trades additional execution and output cost for stronger task performance while keeping input-context cost moderate.

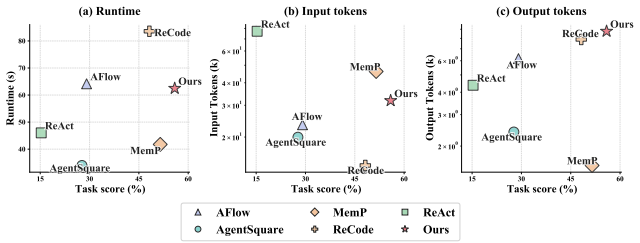

Figure 8: Resource cost versus final pass rate on TravelPlanner using DeepSeek-v4-flash.

5 Conclusion and Future Work

We presented OaK, an ontology-as-a-kernel framework that turns domain semantics into an executable interface for LLM agents. OaK constructs a task-oriented schema and instantiates a schema-guided knowledge graph. It then composes typed reasoning functions from generic operators over this graph. Official task scores and execution traces guide iterative repairs. Across TravelPlanner, CRMArenaPro and ToolQA, OaK achieves the best aggregate performance with two LLM

backbones. The ablations confirm the importance of the function module, function composition, and iterative refinement. These results suggest that making domain semantics and reasoning procedures explicit can improve both effectiveness and inspectability in LLM agents.

OaK still pays a graph-instantiation cost. Its quality also depends on the LLM-based extractor and judge, and requires a reliable task evaluator. Future work can study reusable and incrementally updated graphs to amortize construction, stronger verification or human feedback for open-ended tasks. It is also promising to include richer operator libraries and ontology representations for larger and evolving domains.

References

Debenedetti, E.; Zhang, J.; Balunovic, M.; Beurer-Kellner, L.; Fischer, M.; and Tramèr, F. 2024. AgentDojo: A Dynamic Environment to Evaluate Prompt Injection Attacks and Defenses for LLM Agents. In Globersons, A.; Mackey, L.; Belgrave, D.; Fan, A.; Paquet, U.; Tomczak, J. M.; and Zhang, C., eds., *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

Edge, D.; Trinh, H.; Cheng, N.; Bradley, J.; Chao, A.; Mody, A.; Truitt, S.; and Larson, J. 2024. From Local to Global: A Graph RAG Approach to Query-Focused Summarization. *CoRR*, abs/2404.16130.

Fang, R.; Liang, Y.; Wang, X.; Wu, J.; Qiao, S.; Xie, P.; Huang, F.; Chen, H.; and Zhang, N. 2026. Memp: Exploring Agent Procedural Memory. In Liakata, M.; Moreira, V. P.; Zhang, J.; and Jurgens, D., eds., *Findings of the Association for Computational Linguistics, ACL 2026, San Diego, California, United States, July 2-7, 2026*, 17490–17502. Association for Computational Linguistics.

Glimm, B.; Horrocks, I.; Motik, B.; Stoilos, G.; and Wang, Z. 2014. HermiT: An OWL 2 Reasoner. *J. Autom. Reason.*, 53(3): 245–269.

Gruber, T. R. 1993. A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5(2): 199–220.

He, X.; Tian, Y.; Sun, Y.; Chawla, N. V.; Laurent, T.; LeCun, Y.; Bresson, X.; and Hooi, B. 2024. G-Retriever: Retrieval-Augmented Generation for Textual Graph Understanding and Question Answering. In Globersons, A.; Mackey, L.; Belgrave, D.; Fan, A.; Paquet, U.; Tomczak, J. M.; and Zhang, C., eds., *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

Hogan, A.; Blomqvist, E.; Cochez, M.; D'amato, C.; Melo, G. D.; Gutierrez, C.; Kirrane, S.; Gayo, J. E. L.; Navigli, R.; Neumaier, S.; Ngomo, A.-C. N.; Polleres, A.; Rashid, S. M.; Rula, A.; Schmelzeisen, L.; Sequeda, J.; Staab, S.; and Zimmermann, A. 2021. Knowledge Graphs. *ACM Comput. Surv.*, 54(4).

Huang, K.; Prabhakar, A.; Thorat, O.; Agarwal, D.; Choubey, P. K.; Mao, Y.; Savarese, S.; Xiong, C.; and Wu, C. 2026. CRMArena-Pro: Holistic Assessment of LLM Agents Across Diverse Business Scenarios and Interactions. *Trans. Mach. Learn. Res.*, 2026.

Madaan, A.; Tandon, N.; Gupta, P.; Hallinan, S.; Gao, L.; Wiegreffe, S.; Alon, U.; Dziri, N.; Prabhumoye, S.; Yang, Y.; Gupta, S.; Majumder, B. P.; Hermann, K.; Welleck, S.; Yazdanbakhsh, A.; and Clark, P. 2023. Self-Refine: Iterative Refinement with Self-Feedback. In Oh, A.; Naumann, T.; Globerson, A.; Saenko, K.; Hardt, M.; and Levine, S., eds., *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Pan, S.; Luo, L.; Wang, Y.; Chen, C.; Wang, J.; and Wu, X. 2024. Unifying Large Language Models and Knowledge Graphs: A Roadmap. *IEEE Trans. Knowl. Data Eng.*, 36(7): 3580–3599.

Ruan, Y.; Dong, H.; Wang, A.; Pitis, S.; Zhou, Y.; Ba, J.; Dubois, Y.; Maddison, C. J.; and Hashimoto, T. 2024. Identifying the Risks of LM Agents with an LM-Emulated Sandbox. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Shang, Y.; Li, Y.; Zhao, K.; Ma, L.; Liu, J.; Xu, F.; and Li, Y. 2025. AgentSquare: Automatic LLM Agent Search in Modular Design Space. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Shinn, N.; Cassano, F.; Gopinath, A.; Narasimhan, K.; and Yao, S. 2023. Reflexion: language agents with verbal reinforcement learning. In Oh, A.; Naumann, T.; Globerson, A.; Saenko, K.; Hardt, M.; and Levine, S., eds., *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Wang, H.; Poskitt, C. M.; and Sun, J. 2025. AgentSpec: Customizable Runtime Enforcement for Safe and Reliable LLM Agents. *CoRR*, abs/2503.18666.

Wang, Y.; Zhang, J.; Cai, T.; Liu, Z.; Sun, Q.; Sun, Z.; Wu, Z.; Dong, M.; Zheng, M.; Yin, X.; and Zhu, Y. 2026. From Agent Traces to Trust: A Survey of Evidence Tracing and Execution Provenance in LLM Agents. *CoRR*, abs/2606.04990.

Xie, J.; Zhang, K.; Chen, J.; Zhu, T.; Lou, R.; Tian, Y.; Xiao, Y.; and Su, Y. 2024. TravelPlanner: A Benchmark for Real-World Planning with Language Agents. In Salakhutdinov, R.; Kolter, Z.; Heller, K. A.; Weller, A.; Oliver, N.; Scarlett, J.; and Berkenkamp, F., eds., *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, volume 235 of *Proceedings of Machine Learning Research*, 54590–54613. PMLR / OpenReview.net.

Yang, C.; Sun, Z.; Wei, W.; and Hu, W. 2026. Beyond Static Summarization: Proactive Memory Extraction for LLM Agents. *CoRR*, abs/2601.04463.

Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K. R.; and Cao, Y. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Yu, Z.; Zhang, J.; Su, H.; Zhao, Y.; Wu, Y.; Deng, M.; Xiang, J.; Lin, Y.; Tang, L.; Li, Y.; Luo, Y.; Liu, B.; and Wu, C. 2025. ReCode: Unify Plan and Action for Universal Granularity Control. *CoRR*, abs/2510.23564.

Zhang, J.; Xiang, J.; Yu, Z.; Teng, F.; Chen, X.; Chen, J.; Zhuge, M.; Cheng, X.; Hong, S.; Wang, J.; Zheng, B.; Liu, B.; Luo, Y.; and Wu, C. 2025. AFlow: Automating Agentic Workflow Generation. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Zhang, N.; Yao, Y.; Qin, J.; Xu, H.; Zhu, Y.; Yu, Z.; Wang, M.; Tang, Y.; Gu, J.-C.; Deng, S.; and Chen, H. 2026a. Towards principled knowledge editing methods for large language model reasoning. *Nature Machine Intelligence*.

Zhang, X.; Sun, Z.; Yang, C.; Jin, Y.; Zhang, Y.; and Hu, W. 2026b. ActMem: Bridging the Gap Between Memory Retrieval and Reasoning in LLM Agents. *CoRR*, abs/2603.00026.

Zheng, L.; Chiang, W.; Sheng, Y.; Zhuang, S.; Wu, Z.; Zhuang, Y.; Lin, Z.; Li, Z.; Li, D.; Xing, E. P.; Zhang, H.; Gonzalez, J. E.; and Stoica, I. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In Oh, A.; Naumann, T.; Globerson, A.; Saenko, K.; Hardt, M.; and Levine, S., eds., *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Zhu, R.; Liu, X.; Sun, Z.; Wang, Y.; and Hu, W. 2025. Mitigating Lost-in-Retrieval Problems in Retrieval Augmented Multi-Hop Question Answering. In Che, W.; Nabende, J.; Shutova, E.; and Pilehvar, M. T., eds., *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, 22362–22375. Association for Computational Linguistics.

Zhuang, Y.; Yu, Y.; Wang, K.; Sun, H.; and Zhang, C. 2023. ToolQA: A Dataset for LLM Question Answering with External Tools. In Oh, A.; Naumann, T.; Globerson, A.; Saenko, K.; Hardt, M.; and Levine, S., eds., *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

A Schema Consistency Checks

This appendix details the five kinds of logical validity that the HermiT reasoner verifies on the OWL ontology $\mathrm{OWL}(\mathcal{S}_t)$

Disjointness consistency. The reasoner checks that the schema does not force categories that should be mutually exclusive to overlap. For example, a schema should not imply that *Restaurant* is a subclass of *City*.

Restriction consistency. The reasoner checks that existential, universal, and cardinality constraints do not jointly create contradictions. For example, "a trip has at least one day" and "a trip can have zero days" should not be accepted together.

Property-level consistency. The reasoner checks the logical features declared on properties, including functional, inverse-functional, transitive, symmetric, asymmetric, reflexive, irreflexive, and inverse properties. It also verifies that these features remain compatible when combined with the other axioms of the schema.

Global consistency. The reasoner checks that the schema remains logically satisfiable as a whole, rather than only in isolated fragments.

Unsatisfiable classes. Finally, the reasoner detects unsatisfiable classes, that is, schema components that can never hold an instance once type definitions, relation definitions, and constraints are combined.

B Generic Operator Library

The generic library $\mathcal{O}$ contains the following nine public operators. Underscore-prefixed helpers are implementation details.

- **Runtime-slot extraction.**

```
def extract_runtime_slots(
    *,
    query: str,
    slot_specs: Sequence[RuntimeSlotSpec],
    env_path: str | Path = (
        "/ontology_research/.env"
    ),
    log_dir: str | Path = (
        "ontology_llm_logs"
    ),
    task_context: str = "",
    max_attempts: int = 3,
    max_tokens: int = 2000,
) -> RuntimeSlotExtractionResult:
```

  Maps a natural-language query to declared typed slots with an LLM, retaining only explicit or strongly implied constraints and their evidence.

- **Entity lookup.**

```
def lookup_entities(
    graph: str | Path | dict[str, Any] |
        InstantiatedGraph,
    *,
    entity_types: Sequence[str] | None =
        None,
    entity_ids: Sequence[str] | None =
        None,
    primary_key: dict[str, Any] | None =
        None,
    property_filters: dict[str, Any] |
        None = None,
    name_query: str | None = None,
    text_query: str | None = None,
    fuzzy: bool = True,
    top_k: int | None = None,
    min_score: float = 0.0,
) -> EntityLookupResult:
```

  Retrieves entities by type, identifier, primary key, exact property values, or fuzzy name and text matching, with optional ranking and truncation.

- **Relation traversal.**

```
def traverse_relations(
    graph: str | Path | dict[str, Any] |
        InstantiatedGraph,
    *,
    start_entity_ids: Sequence[str],
    relation_types: Sequence[str] | None
        = None,
    direction: TraversalDirection =
        "outgoing",
    max_hops: int = 1,
    include_starting_entities: bool =
        True,
) -> TraversalResult:
```

  Expands a bounded neighborhood from seed entities along selected relation types and directions, returning the visited entities and relations.

- **Property projection.**

```
def project_properties(
    entities: Sequence[GraphEntity] |
        EntityLookupResult |
        EntityFilterResult |
        TraversalResult,
    *,
    property_names: Sequence[str],
    missing_value: Any = None,
) -> list[ProjectedRow]:
```

  Projects selected entity properties into flat rows while preserving entity identifiers, types, and primary keys.

- **Categorical filtering.**

```
def filter_property_categorical(
    entities: Sequence[GraphEntity] |
        EntityLookupResult |
        EntityFilterResult |
        TraversalResult,
    *,
    conditions: Sequence[
    PropertyCategoricalFilterCondition
    ],
) -> EntityFilterResult:
```

  Keeps entities that satisfy exact categorical inclusion or exclusion conditions. $\mathcal{O}$ exposes this function through the alias `filter_categorical`.

- **Relation-connectivity filtering.**

```
def filter_relation_connected(
```

```
    graph: str | Path | dict[str, Any] |
        InstantiatedGraph,
    entities: Sequence[GraphEntity] |
        EntityLookupResult |
        EntityFilterResult |
        TraversalResult,
    *,
    conditions: Sequence[
       RelationFilterCondition
    ],
) -> EntityFilterResult:
```

Keeps entities connected to specified anchors through required relation types.

- **Numeric filtering.**

```
def filter_numeric(
    entities: Sequence[GraphEntity] |
        EntityLookupResult |
        EntityFilterResult |
        TraversalResult,
    *,
    conditions: Sequence[
       NumericFilterCondition
    ],
) -> EntityFilterResult:
```

Applies numeric equality, inequality, threshold, or interval constraints to entity properties.

- **Set-overlap filtering.**

```
def filter_set_overlap(
    entities: Sequence[GraphEntity] |
        EntityLookupResult |
        EntityFilterResult |
        TraversalResult,
    *,
    conditions: Sequence[
       SetOverlapFilterCondition
    ],
) -> EntityFilterResult:
```

Filters multi-valued properties by any-match, all-match, or minimum-overlap requirements against a requested set.

- **Aggregation.**

```
def aggregate_values(
    items: Sequence[GraphEntity |
        ProjectedRow | dict[str, Any]] |
        Iterable[GraphEntity |
        ProjectedRow | dict[str, Any]],
    *,
    request: AggregateRequest,
) -> AggregateResult:
```

Computes count, sum, minimum, maximum, or average statistics over entities, optionally grouped by a field.

C Multi-Seed Robustness

We report three runs of OaK with different random seeds under DeepSeek-v4-flash. Across seeds, Table 4 reports a final pass rate of 55.90 ± 2.13 for OaK. The macro metrics are also stable, with 58.60 ± 2.38 on CS Macro and 61.57 ± 1.04 on HC Macro. This suggests that the gains in Table 1 are not driven by a single favorable seed.

Seed	CS		HC		Final
	Micro	Macro	Micro	Macro	
1	83.50	55.90	58.31	62.40	54.00
2	88.77	60.40	60.46	61.90	58.20
3	86.06	59.50	59.16	60.40	55.50
Avg. $\pm$ Std.	86.11 ± 2.64	58.60 ± 2.38	59.31 ± 1.08	61.57 ± 1.04	55.90 ± 2.13

Table 4: Multi-seed robustness of OaK on TravelPlanner using DeepSeek-v4-flash. All entries are percentages (%).

D Ablation Study on GPT-4o-mini

We report the same three ablation variants as in the main-text ablation study, now using gpt-4o-mini as the backbone.

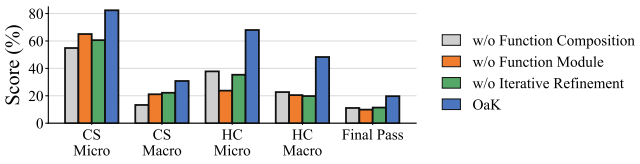

Figure 9: Ablation results on TravelPlanner using gpt-4o-mini.

TravelPlanner. Under gpt-4o-mini, Figure 9 shows that OaK achieves the highest score on all five metrics. The final pass rate drops from 19.70 to 9.90 without the function module, and the degradation is also visible on the HC metrics.

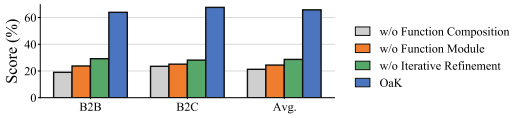

Figure 10: Ablation results on CRMArenaPro using gpt-4o-mini.

CRMArenaPro. Figure 10 shows that the full OaK model clearly outperforms all variants on both B2B and B2C. On the average score, OaK reaches 65.78, while the strongest ablation remains below 30 in both settings.

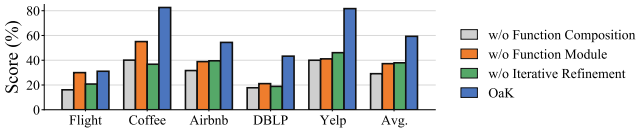

Figure 11: Ablation results on ToolQA using gpt-4o-mini.

ToolQA. In Figure 11, OaK outperforms all variants on every subset and reaches 59.32 on the weighted average. The ablations remain below 40. The clearest gaps appear on Coffee, DBLP, and Yelp.