# NAVIAGENT: BILEVEL PLANNING ON TOOL NAVIGATION GRAPH FOR LARGE-SCALE ORCHESTRATION

**Anonymous authors**Paper under double-blind review

#### **ABSTRACT**

Large language models (LLMs) have recently demonstrated the ability to act as function call agents by invoking external tools, enabling them to solve tasks beyond their static knowledge. However, existing agents typically call tools step by step at a time without a global view of task structure. As tools depend on each other, this leads to error accumulation and limited scalability, particularly when scaling to thousands of tools. To address these limitations, we propose NaviAgent, a novel bilevel architecture that decouples task planning from tool execution through graph-based modeling of the tool ecosystem. At the taskplanning level, the LLM-based agent decides whether to respond directly, clarify user intent, invoke a toolchain, or execute tool outputs, ensuring broad coverage of interaction scenarios independent of inter-tool complexity. At the execution level, a continuously evolving Tool World Navigation Model (TWNM) encodes structural and behavioral relations among tools, guiding the agent to generate scalable and robust invocation sequences. By incorporating feedback from real tool interactions, NaviAgent supports closed-loop optimization of planning and execution, moving beyond tool calling toward adaptive navigation of large-scale tool ecosystems. Experiments show that NaviAgent achieves the best task success rates across models and tasks, and integrating TWMN further boosts performance by up to 17 points on complex tasks, underscoring its key role in toolchain orchestration.

### 1 Introduction

Large language models (LLMs) are increasingly deployed as function call agents, moving beyond single utilities toward complex multi-stage workflows (Shen et al., 2023; Yang et al., 2023; Qu et al., 2025). However, real-world environments contain thousands of heterogeneous tools that are continually updated, while tasks demand long sequences of coordinated invocations. Agents built around fixed tool descriptions or rigid workflows fail to adapt, making API drift, continual updates, and unseen tool compositions key challenges for function call agents.

Existing approaches attempt to mitigate brittleness but remain incomplete. Some embed tool knowledge directly into model parameters (Wang et al., 2024), which reduces context demands but requires costly retraining when APIs change. Others derive static graphs from invocation logs (Liu et al., 2024b), yet sparse traces and missing parameter relations hinder generalization. Policy-adaptation methods adjust individual tools with feedback (Chen et al., 2024), while clustering-based planners enable substitutions (Liu et al., 2024c). Taken together, existing methods can be broadly categorized into two camps: either structured but static, failing to evolve with the ecosystem, or adaptive but unstructured, lacking the representations needed to capture composability and complementarity.

Underlying these challenges is the complexity of the tool ecosystem: it spans thousands of heterogeneous tools, exhibits interdependencies such as parameter flows and functional complementarity, and evolves continually through addition, update, and deprecation. Such properties reveal why step-by-step invocation without global awareness cannot achieve reliable tool composition. The difficulty is compounded by the fact that API documentation is written for humans and often misaligns with how models interpret and use individual tools (Qu et al., 2024), while flat catalogues provide little information on how tools compose, substitute, or adapt as the ecosystem changes. What is needed is a structured representation learned from execution traces that makes these dependencies explicit and continually adapts with feedback.

We propose **NaviAgent**, a bilevel planning framework that decouples high-level task reasoning from low-level execution. At the planning level, NaviAgent defines a four-dimensional decision space (direct response, intent clarification, toolchain retrieval, tool execution) covering core tool invocation scenarios, allowing the agent to operate without reasoning over complex inter-tool connections. At the execution level, it constructs the Tool World Navigation Model (TWNM), which encodes both structural and behavioral dependencies learned from execution traces. By coupling these graph-based representations with navigation strategies, TWNM enables retrieval, substitution, and multi-tool composition as the ecosystem evolves. Execution feedback continually updates both TWNM and the decision policy, forming a closed loop for robust adaptation to changing APIs.

Our main contributions are as follows: i) **NaviAgent Architecture.** The first bilevel agent framework that decouples high-level task planning from low-level tool execution, enabling scalable task composition across thousands of tools while preserving efficiency. ii) **Tool World Navigation Model.** A unified model that captures inter-tool structures and behavioral dependencies from execution traces, and supports navigation and flexible search in large-scale tool ecosystems. iii) **Closed-loop Evolution.** A feedback mechanism where execution traces continuously refine TWNM and decision strategies, driving the co-evolution of representation and decision-making.

# 2 RELATED WORK

**Single-Tool Invocation.** Early research focused on enhancing LLMs' single-tool invocation capabilities. TALM (Parisi et al., 2022) established foundational paradigms through predefined template chains, while EasyTool (Yuan et al., 2024) introduced structured tool descriptions to reduce semantic parsing overhead. For long-context scenarios, tool documentation compression techniques preserved critical semantics via summarization, enabling low-resource tool usage (Xu et al., 2024). Toolformer (Schick et al., 2023) innovatively embedded tool invocation APIs in pre-training, allowing self-supervised learning of usage patterns from unlabeled data. In multimodal settings, GPT4Tools (Yang et al., 2023) improved visual tool generalization (e.g. object detection) by aligning vision-language instructions with tool descriptions.

**Multi-Tool Orchestration.** As tool libraries expanded, HuggingGPT (Shen et al., 2023) proposed a four-stage pipeline (plan, select, execute, respond) for standardized multi-tool workflows, while Chameleon (Lu et al., 2023) integrated heterogeneous tools (13+ types) via modular composition. Similarly,  $\alpha$ -UMI (Shen et al., 2024) decomposes the tool-use process into planning, invocation, and summarization, but uniquely assigns each stage to a dedicated lightweight LLM, enabling modular updates and improved performance, especially for smaller models. For small toolkits, TRICE (Qiao et al., 2023) optimized single tool policies via execution feedback, and ToolFactory (Ni et al., 2025) automated tool adaptation through domain-guided code synthesis. However, these approaches struggled with dynamic collaboration. For large-scale toolkits, Confucius (Gao et al., 2024) addressed combinatorial explosion via hierarchical tool classification, and ToolVerifier (Mekala et al., 2024) improved selection robustness through self-verification mechanisms.

Dynamic Planning & Adaptation. Static frameworks faltered under open-domain task complexity, prompting dynamic decision mechanisms. ReAct (Yao et al., 2023b) pioneered the decoupling of reasoning from tool calls through chain-of-thought planning. Building on this, Reflexion (Shinn et al., 2023) enhanced error recovery by introducing iterative self-reflection, significantly improving fault tolerance in complex tasks. For long-horizon tasks, path search techniques became pivotal: Tree-of-Thoughts (ToT) (Yao et al., 2023a) formalized tool invocation as searchable reasoning trees with dynamic branching, while ToolLLM (Qin et al., 2023) optimized search efficiency through functional hierarchy-guided DFS. ToolChain (Zhuang et al., 2023) further advanced this by employing heuristic cost estimation to prioritize high-success-rate branches. Yet, these methods assumed static tool relationships, failing to adapt to API drift or cross-domain tasks. ControlLLM (Liu et al., 2024d) built static dependency graphs for task decomposition, whereas ToolNet (Liu et al., 2024b) dynamically updated tool relations from historical calls, both limited by sparse multi-hop interaction data. This gap motivates our TWNM that jointly models structural dependencies and behavioral adaptations to capture evolving tool relations, aligning with findings that graph learning enhance LLM planning (Wu et al., 2024; Besta et al., 2024).

# 3 METHODOLOGY

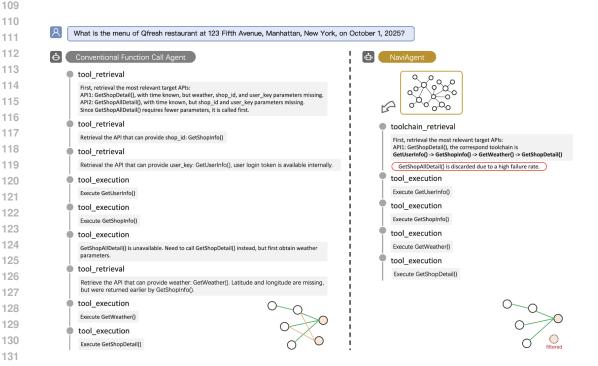


Figure 1: Conventional function call agents vs. NaviAgent.

A key challenge in function call agents is that API calls usually have interdependent parameters and strict invocation orders. NaviAgent addresses this challenge by retrieving the entire toolchain before execution, rather than calling APIs step by step. This global reasoning allows the agent to plan an end-to-end path from the user query to the target API once and execute it directly. As illustrated in Figure 1, NaviAgent avoids repeated retrievals and automatically discards unreliable APIs, leading to more efficient and robust execution.

#### 3.1 A FOUR-DIMENSIONAL DECISION AGENT

## 3.1.1 DEFINITION

The architecture achieves end-to-end decision-making through LLMs, formally modeled as a quintuple  $(\mathcal{H}, \mathcal{O}, \mathcal{G}, \mathcal{A}, F)$  where  $\mathcal{H} = \{(o_{t-i}, a_{t-i})\}_{i=1}^n$  represents historical states (containing state sequence  $\{o_i\}$  and action sequence  $\{a_i\}$ ),  $\mathcal{O}$  denotes the observation,  $\mathcal{G}$  represents the tool dependency graph,  $\mathcal{A} = \{Direct\ Response,\ Intent\ Clarification,\ ToolChain\ Retrieval,\ Tool\ Execution\}$  defines the four dimensional decision space, where each action corresponds to directly answering the user, requesting clarification, retrieving candidate tool dependency subgraph via graph pruning, or execute selected toolchains, respectively.  $F: \mathcal{H} \times \mathcal{O} \times \mathcal{G} \to \mathcal{A}$  specifies the decision function. At each time step t, the agent constructs its decision context as follows. The historical context  $\mathcal{H}_t$  is defined as

$$\mathcal{H}_t = \langle (o_{t-3}, a_{t-3}), \dots, (o_{t-1}, a_{t-1}) \rangle \tag{1}$$

where a sliding window maintains the most recent three observation-action pairs, capturing the agent's recent decision trajectory. The pruned tool dependency subgraph  $\mathcal{G}'_{t-1}=(V,E,W)$  is computed from the agent's state at the previous time step t-1, where V is the node set, E is the edge set, and W denotes the edge weights indicating dependency strengths. The subgraph is serialized into a tree-structured textual format, ensuring a simplified yet sufficient representation for selected toolchains. The overall decision function is then formulated as

$$a_t = F(\mathcal{H}_t, \mathcal{O}_t, \mathcal{G}'_{t-1}) \tag{2}$$

<sup>&</sup>lt;sup>1</sup>Our experiments demonstrate that utilizing the most recent three observation-action pairs achieves the best balance between accuracy and efficiency.

where  $\mathcal{O}_t$  is the current observation, and  $a_t \in \mathcal{A}$  is the action selected at time t.

#### 3.1.2 MODEL TRAINING

For supervised fine-tuning, we adopt the standard language modeling objective, computing the loss exclusively over the response or action generation segments. During training, the LLM-based agent receives as input the most recent historical state-action pairs  $\mathcal{H}_t$ , the current observation  $\mathcal{O}_t$ , and the pruned tool dependency subgraph  $\mathcal{G}_{\text{sub}}$ . The model is trained to maximize the likelihood of the ground-truth action  $a_t^*$  at step t, which is derived from high-quality, curated datasets (see Appendix E.2 for details):

$$\mathcal{L}_{SFT} = -\frac{1}{N} \sum_{i=1}^{N} \log p_{\theta}(a_t^* \mid \mathcal{H}_t, \mathcal{O}_t, \mathcal{G}_{sub})$$
 (3)

where N is the number of training samples and  $p_{\theta}$  denotes the agent's predicted probability over the action space.

# 3.2 TOOL WORLD NAVIGATION MODEL

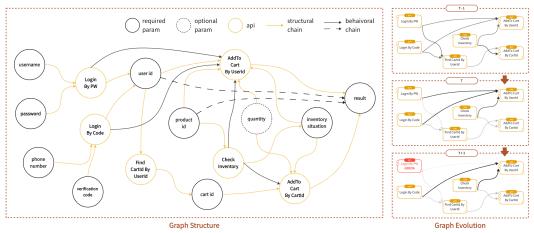


Figure 2: Tool dependency graph and its temporal evolution in TWNM. The left part shows the overall dependency relations, while the right part illustrates the pruning and evolution of executable subgraphs across time steps.

# 3.2.1 GRAPH CONSTRUCTION AND REPRESENTATION

While tool standardization frameworks (e.g., Anthropic's MCP) help normalize basic API metadata, challenges remain due to inconsistent parameter naming and undocumented tool dependencies. In our framework, each tool consists of one or more APIs. We address these issues by applying semantic similarity clustering to unify functionally equivalent parameters (see details in Appendix A).

**Definition.** We construct a directed weighted graph  $\mathcal{G}=(V,E,W)$  with API and parameter nodes. Edges include structural chains, defined by API schemas (e.g., parameter-to-API and API-to-parameter connections), as well as behavioral chains, derived from historical usage data (e.g., API-to-API and parameter-to-parameter dependencies), as illustrated in Figure 2 (left). Each edge is assigned a statistical weight  $\tilde{w}_{ij}$  reflecting empirical invocation patterns.

$$\tilde{w}_{ij} = \frac{N(v_i \to v_j)}{N(v_j)} \tag{4}$$

where  $N(v_i \to v_j)$  counts the number of successful invocations from  $v_i$  to  $v_j$ , and  $N(v_j)$  is the total number of invocations involving  $v_j$ .

We formulate tool dependency discovery as a link prediction problem Hamilton et al. (2017); Zhang & Chen (2018); Zhou et al. (2020); Wu et al. (2024). To model this, we employ a Heterogeneous Graph Transformer (HGT) that integrates node-level feature fusion, type-specific encoding, and

relation-aware message passing. Each node is initialized with both semantic (BGE-based) and structural features (including invocation statistics and degree information), and projected into a unified embedding space. We stack two multi-head HGT layers to aggregate information from the 2-hop neighborhood. Notably, the attention mechanism incorporates a statistical edge weight  $\tilde{w}_{uv}$  to reflect empirical call patterns:

$$\alpha_{uv}^{(k,r)} = softmax_{u \in \mathcal{N}_r(v)} \left( \frac{(\mathbf{W}_Q^{(k,r)} \mathbf{h}_u')^\top (\mathbf{W}_K^{(k,r)} \mathbf{h}_v')}{\sqrt{d_k}} + \mathbf{b}_r^{(k)} + \tilde{w}_{uv} \right)$$
(5)

where  $\mathcal{N}_r(v)$  denotes the set of neighbors of node v under relation r, and  $\mathbf{h}'_u$ ,  $\mathbf{h}'_v$  are the type-specific encoded representations of nodes u and v (see Appendix B.1 for details).  $\mathbf{W}_Q^{(k,r)}$  and  $\mathbf{W}_K^{(k,r)}$  are the query and key projection matrices for head k and relation r,  $\mathbf{b}_r^{(k)}$  is an edge-type-specific bias, and  $d_k = d/8$  is the dimension per head. Then the concatenated head outputs are projected to obtain the final node embeddings, which are then used for link prediction.

# 3.2.2 GRAPH TRAINING OBJECTIVE

The graph model is trained with a hybrid loss that combines cross-entropy and adaptive margin objectives, both leveraging edge weights  $\tilde{w}_{uv}$  to capture graded dependencies.

# Cross-entropy.

$$\mathcal{L}_{CE} = -\frac{1}{|\mathcal{E}|} \sum_{(u,v)\in\mathcal{E}} \left[ \tilde{w}_{uv} \log p_{uv} + (1 - \tilde{w}_{uv}) \log(1 - p_{uv}) \right]$$
 (6)

where  $p_{uv}$  is the predicted link probability,  $\tilde{w}_{uv}$  is the statistical edge weight serving as a soft label, and  $\mathcal{E}$  denotes the set of all edges in the graph.

**Adaptive margin.** It assigns larger separation to higher-weight edges(i.e.,  $\tilde{w}_{uv} \to 1$ ), focusing learning on critical dependencies. For each positive edge  $(u,v)^+ \in \mathcal{E}^+$ , k negative edges  $\{(u_i,v)\}_{i=1}^k$  are sampled to construct positive and negative pairs for the margin loss.

$$m_{uv} = m_0 \left( 1 + \sigma(\tilde{w}_{uv}) \right) \tag{7}$$

$$\mathcal{L}_{\text{Margin}} = \frac{1}{|\mathcal{E}^+|} \sum_{(u,v)^+ \in \mathcal{E}^+} \frac{1}{k} \sum_{j=1}^k \left[ m_{uv} - s(u,v)^+ + s(u_j,v)^- \right]_+$$
(8)

where  $m_0$  is a base margin,  $\sigma(\cdot)$  denotes the sigmoid function,  $\tilde{w}_{uv}$  is the statistical edge weight, s(u,v) measures the embedding similarity,  $(u,v)^+$  represents a positive edge,  $(u_j,v)^-$  denotes a negative sample, and  $[\cdot]_+$  is the hinge function, and  $\mathcal{E}^+$  is the set of positive edges.

The final training objective is a weighted sum of the two losses:

$$\mu_t = \mu_0 \cdot \gamma^t, \ \gamma \in (0, 1) \tag{9}$$

$$\mathcal{L} = \mu_t \cdot \mathcal{L}_{CE} + (1 - \mu_t) \cdot \mathcal{L}_{Margin}$$
 (10)

where  $\mu_t$  is the weight for the cross-entropy loss at epoch t,  $\mu_0$  is the initial weight, and  $\gamma$  is a decay factor controlling the rate at which the contribution of the cross-entropy loss decreases over training. This curriculum strategy first emphasizes accuracy, then discrimination, yielding accurate predictions and structured embeddings.

#### 3.2.3 GRAPH SEARCH

At inference time, the predicted link probabilities  $p_{uv}$  are used as edge weights  $w_{uv}$  in the tool graph, forming the basis for weighted-graph search and toolchain planning. We adopt two representative search strategies adapted to this setting: an Alpha-Beta Pruning method that eliminates weak toolchains using dynamic thresholds, and a heuristic search that evaluates candidate toolchains with a composite fitness balancing connectivity, depth, and cumulative weights. Complete algorithms and parameter details are provided in Appendix B.2.

# 3.2.4 Graph Evolution

The tool world is inherently dynamic, evolving as new tools are introduced, obsolete ones are deprecated, and usage patterns shift. As shown at the right of Figure 2: At time T-1, selectable paths are relatively uniform (similar line shades), indicating multiple equally viable routes. By T, the TWNM has learned to prefer a more optimal path (the top, darker line). At T+1, when the upper-left API becomes unavailable, the TWNM adapts by recomputing and selecting the lower route as the new optimal path. This demonstrates the TWNM's ability to learn from feedback and flexibly adjust its planning in response to runtime changes. To systematically support such adaptability, we design a graph evolution framework with three key mechanisms:

**Incremental Node Integration.** To accommodate newly introduced tools, we incrementally add new nodes via semantic similarity clustering, initializing their parameters (e.g.,  $N_{succ}(v) = 0$ ,  $N_{fail}(v) = 0$  for successful and failed invocation counts) and the statistical weights of associated edges (e.g.,  $\tilde{w}_{uv} = 0$ ) to ensure consistency with existing graph features.

**Targeted Subgraph Pruning.** Obsolete or rarely used tools are selectively pruned based on a weighted combination of failure rates and invocation frequencies:

$$Prune(v) \propto \lambda \cdot \sigma(f_{fail}(v)) + (1 - \lambda) \cdot \sigma(f_{freq}(v)^{-1})$$
(11)

where  $\lambda \in [0,1]$  controls the trade-off between failure rates and invocation frequencies, and  $f_{fail}$  and  $f_{freq}$  denote failure rates and invocation frequencies, respectively.

**Edge Attribute Propagation.** Long-term stability and short-term adaptation are balanced by updating the statistical edge weights  $\tilde{w}_{uv}$  through a combination of historical trends and recent invocation success rates:

$$\tilde{w}_{uv}^{(t)} = \eta \cdot \underbrace{\tilde{w}_{uv}^{(t-1)}}_{\text{long-term weight}} + (1 - \eta) \cdot \frac{N_{succ}^{\text{recent } \tau \text{ days}}(u \to v)}{N_{succ}^{\text{recent } \tau \text{ days}}(v)}$$

$$\underbrace{N_{succ}^{\text{recent } \tau \text{ days}}(v)}_{\text{recent success rate}}$$
(12)

where  $\eta \in [0,1]$  balances long-term memory and recent observations, and  $N_{succ}^{\mathrm{recent}}$   $\tau^{\mathrm{days}}$  denotes successful invocations within a sliding window of  $\tau$  days. These dynamically updated statistical edge weights  $\tilde{w}_{uv}$  are subsequently used as soft labels for supervising model training, as described in Section 3.2.2.

# 3.3 DYNAMIC EXECUTION & PATH RECOMBINATION

Robust and adaptive toolchain orchestration is achieved through a bilevel dynamic planning framework, in which the agent manages action selection and the TWNM is responsible for toolchain planning.

**NaviAgent Workflow.** When a user query arrives, NaviAgent decides whether it can respond directly, clarify the user's intent, or rely on external tools. For more complex queries, NaviAgent decomposes the task into sub-tasks and categorizes them into two types: those that can be answered or clarified immediately, and those that demand toolchain retrieval. Unlike traditional agents that fetch tools sequentially, NaviAgent searches the existing tool dependency graph for a task-relevant subgraph and selects a feasible execution path for subsequent execution. More detailed cases can be found in the Appendix C.

**Path Recombination.** During execution, if a *tool execution* action fails due to an an API is unavailable or malfunctioning, the agent switches from execution to *toolchain retrieval* and invokes its TWNM module. TWNM searches the current tool dependency graph to recombine nodes and identify an alternative toolchain, which the agent then executes. This adaptive loop can be repeated until completion or infeasibility, enabling dynamic path recombination that improves robustness and task success in complex tool environments.

# 4 EXPERIMENTS

#### 4.1 EXPERIMENTAL SETTINGS

**Datasets.** Our experiments are based on two public API benchmarks: API-Bank Li et al. (2023) and ToolBench Qin et al. (2023). As real-time API execution is currently unavailable, evaluation tasks are constructed in a simulated environment based on the extensive API lists and conversational trajectories provided by these datasets. Tasks are categorized into three levels of complexity: **Easy** (at most one API call or directly answerable), **Medium** (two API calls), and **Hard** (three or more APIs). Details of task generation are provided in Appendix D. For model fine-tuning, Qwen2.5-14B is trained on 3,500+ examples sampled from our generated task set, with strict separation between fine-tuning and evaluation data to prevent leakage.

Baselines and Models. The evaluation considers frameworks for real-world tool invocation, where managing large tool sets and enabling autonomous planning are critical. We select representative baselines in three major categories: (i) ReAct-based single-agent frameworks, where ReAct Yao et al. (2023b) serves as the foundational approach alternating reasoning and tool use; (ii) enhanced single-agent frameworks, where ToolLLM Qin et al. (2023) incorporates DFSDT-based planning with a dynamic backtracking mechanism; and (iii) multi-agent frameworks, where  $\alpha$ -UMI Shen et al. (2024) organizes modular tool-use stages via lightweight LLMs. Experiments are conducted across multiple foundation models, including open-source models (Qwen2.5-14B Yang et al. (2024), Qwen2.5-32B Tahmid & Sarker (2024), DeepSeek-R1-Distill-Qwen-32B(DeepSeek-R1-32B) Guo et al. (2025)) and closed-source models (DeepSeek-V3 Liu et al. (2024a), GPT-4o Hurst et al. (2024)), as well as a fine-tuned lightweight model (Qwen2.5-14B).

**Metrics.** Our evaluation framework considers three metrics: task success rate (TSR), execution steps (Steps), and task completion rate (TCR). TSR and Steps are the primary indicators, with TSR measuring output quality by evaluating whether the system's response fully satisfies the user's request (via LLM-based comparison with the ground truth), and Steps reflecting execution efficiency as the total number of LLM calls required to solve a task, counted only for successfully completed tasks. TCR serves as a supplementary metric, indicating whether the system produces a final output without prematurely terminating. Tasks are considered incomplete if they exceed the maximum allowed attempts, encounter parsing errors, or fail due to input token limits. Both TCR and TSR are reported as percentages over all evaluation tasks. All experiments details of training and inference setup provided in Appendix E.2.

#### 4.2 RESULTS

In this section, we present the main results on ToolBench, comparing NaviAgent with strong baselines across various model sizes and task difficulties.

Model	Method	Easy			Medium			Hard			All		
1,10de1		TCR	TSR	Steps	TCR	TSR	Steps	TCR	TSR	Steps	TCR	TSR	Steps
Qwen2.5-14B	ReAct	32.4	26.3	3.52	24.5	16.8	3.67	24.8	20.0	3.64	27.1	20.6	3.61
	ToolLLM	56.1	30.4	4.01	53.8	19.7	4.02	38.1	11.4	4.06	51.0	21.3	4.03
	α-UMI	77.7	39.2	5.53	77.9	25.0	5.88	67.6	13.3	6.07	75.5	26.9	5.74
	<b>Dynamic+H</b>	64.2	<b>50.3</b>	4.18	60.1	32.3	4.38	61.1	22.4	4.68	61.6	35.8	4.38
Qwen2.5-32B	ReAct	33.1	25.0	3.50	35.6	24.5	3.60	30.5	19.0	3.95	33.6	23.4	3.63
	ToolLLM	40.5	31.8	3.67	48.6	30.3	3.85	49.5	23.8	4.10	46.2	29.3	3.83
	α-UMI	78.4	49.3	5.66	78.8	26.0	6.02	77.1	22.9	6.58	78.3	32.8	5.94
	<b>Dynamic+H</b>	88.1	<b>61.1</b>	4.29	81.7	41.5	4.60	79.4	<b>30.8</b>	5.31	83.2	<b>45.4</b>	4.66
Deepseek-V3	ReAct	46.6	36.5	3.52	58.7	38.5	3.50	48.6	23.8	3.74	52.5	34.5	3.54
	ToolLLM	56.2	47.4	3.80	58.8	30.0	3.92	29.7	24.8	3.90	51.3	34.4	3.86
	α-UMI	80.8	59.7	5.95	89.4	32.9	5.95	73.0	29.5	6.64	82.9	40.7	6.06
	<b>Dynamic+H</b>	97.9	<b>71.8</b>	4.40	96.3	<b>48.5</b>	4.45	97.0	44.9	5.19	97.0	55.2	4.60

Table 1: **Comparison of Baseline Frameworks on ToolBench.** TCR and TSR are reported as percentages (%), and lower Steps indicates higher efficiency. The best results are marked in **bold** and the second-best results are marked with underline.

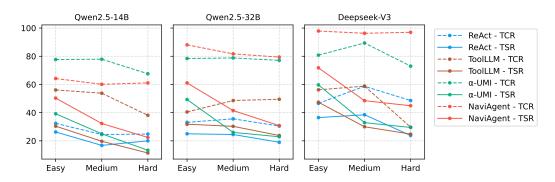
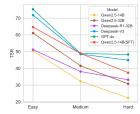


Figure 3: Evaluation of Frameworks on ToolBench Across Task Complexity.

**Overall Performance and Efficiency.** As shown in Table 1 and Figure 3, NaviAgent consistently achieves the highest TSR across all foundation models and task complexities, with absolute values of 35.8% on Qwen2.5-14B, 45.4% on Qwen2.5-32B, and 55.2% on Deepseek-V3. Compared to the average performance of the baselines on all tasks, NaviAgent achieves substantial gains of 12.9, 16.9, and 18.7 percentage points on Qwen2.5-14B, Qwen2.5-32B, and Deepseek-V3, respectively. Meanwhile, its execution steps remain close to those of the most efficient baseline, with differences typically within one step, thereby maintaining a strong balance between solution quality and execution efficiency. Consistent performance is also observed in our real-world API tests, with detailed results provided in Appendix F.1.

Relative Improvement and Robustness. NaviAgent achieves an average TSR improvement of over 10 percentage points compared to  $\alpha$ -UMI, the strongest among the three baselines, across all difficulty levels, with the most significant gain of 15.4 percentage points on Deepseek-V3 for Hard tasks. We further observe that the relative drop in TSR from Easy to Hard tasks is substantially smaller for NaviAgent than for most baselines, particularly on larger foundation models. For example, on Deepseek-V3, NaviAgent's TSR decreases by only 37.5% from Easy to Hard, while ToolLLM and  $\alpha$ -UMI experience drops of 47.7% and 50.6%, respectively.

**Adaptability through Fine-tuning.** Notably, with supervised fine-tuning, the smaller Qwen2.5-14B model achieves performance comparable to the larger 32B model (TCR 81.2% vs 83.2%, TSR 51.3% vs 45.4%, see Figure 4 and Table 2, D+N(Heur) row), indicating that fine-tuning can effectively close the gap between model sizes.



# 4.3 ABLATION STUDY

Figure 4: Effect of SFT on TSR.

To further validate the effectiveness of each component in our framework, we conduct two sets of ablation studies.

**Effect of base Components.** We analyze the NaviAgent (Base) configuration, focusing on its four-dimensional decision space in successful ToolBench cases with the Deepseek-V3 model. Specifically, we categorize the proportion of cases resolved via **Clarification** (*intent clarification* to seek additional details from the user), **Re-retrieval** (recovering from initial *toolchain retrieval* failures by invoking alternative APIs), and **Normal** (tasks completed successfully in a single attempt without clarification or re-retrieval). Results are summarized in Figure 5, demonstrating that the four-dimensional decision space of the agent enables robust error recovery and flexible intent handling, contributing to overall performance gains.

**Effect of TWNM Components.** Table 2 shows clear gains from each design choice. Compared with the Base (agent only), NaviAgent (DynamicH) improves TSR by +11.8 points on average, confirming the value of graph-based planning with search. Dynamic graphs further outperform static ones on hard tasks (e.g., +5.1 on Qwen2.5-32B, +2.0 on GPT-4o), and replacing Alpha-Beta with heuristic search brings the best results, adding 2–3 points on all tasks and about 8 points on hard

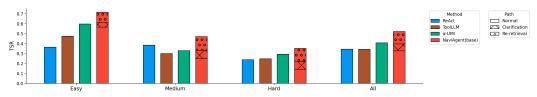


Figure 5: Comparison of TSR Distribution Between NaviAgent(base) and Baselines.

Model	Method		Easy		Medium			Hard			All		
Widdel	1/10tilou	TCR	TSR	Steps	TCR	TSR	Steps	TCR	TSR	Steps	TCR	TSR	Steps
	Base	46.4	36.0	5.38	50.5	22.9	5.39	62.0	16.3	5.76	51.8	25.6	5.47
Qwen2.5-14B	Static+A	57.3	43.7	4.37	61.2	29.0	4.54	53.0	14.0	4.59	58.1	30.3	4.50
Qwell2.3-14D	Dynamic+A	58.8	<u>48.0</u>	4.31	61.5	<u>31.7</u>	4.49	53.3	16.2	4.61	58.8	<u>33.4</u>	4.46
	Dynamic+H	64.2	50.3	4.18	60.1	32.3	4.38	61.1	22.4	4.68	61.6	35.8	4.38
	Base	77.7	47.7	5.42	75.8	32.7	6.00	86.9	19.0	7.04	78.9	34.4	6.05
Owen2.5-32B	Static+A	82.8	50.7	4.47	83.3	40.6	5.07	79.7	26.3	5.30	82.3	40.6	4.93
Qwell2.3-32B	Dynamic+A	83.1	<u>51.4</u>	4.41	85.1	<u>41.3</u>	5.03	80.0	31.4	5.37	83.3	<u>42.3</u>	4.91
	Dynamic+H	88.1	61.1	4.29	81.7	41.5	4.60	79.4	<u>30.8</u>	5.31	83.2	45.4	4.66
	Base	89.5	32.2	6.16	85.0	25.8	6.64	88.6	19.7	6.65	87.3	26.5	6.49
Deepseek-R1-32B	Static+A	92.3	45.8	5.14	92.5	35.8	5.39	91.5	20.8	5.99	92.2	35.6	5.45
Deepseek-K1-32B	Dynamic+A	92.6	51.4	5.06	93.3	38.0	5.33	91.4	<u>21.9</u>	5.93	92.6	<u>38.6</u>	5.38
	Dynamic+H	93.5	<u>51.2</u>	4.82	92.4	38.1	5.23	87.8	33.2	5.46	91.7	41.2	5.15
	Base	93.7	66.3	5.26	93.8	39.7	6.00	94.7	31.1	6.22	94.0	46.3	5.81
Doorgook V2	Static+A	92.9	70.5	4.31	95.8	47.4	4.66	93.4	31.1	5.05	94.3	51.1	4.64
Deepseek-V3	Dynamic+A	93.2	<u>71.6</u>	4.36	95.7	50.5	4.68	93.3	<u>33.3</u>	4.97	94.4	<u>53.4</u>	4.64
	Dynamic+H	97.9	71.8	4.40	96.3	<u>48.5</u>	4.45	97.0	44.9	5.19	97.0	55.2	4.60
	Base	92.0	62.7	5.07	91.0	35.2	5.67	94.5	27.8	6.26	92.1	42.3	5.61
GPT-40	Static+A	99.5	72.1	4.21	98.3	43.6	5.35	97.8	37.9	5.85	98.6	51.5	5.10
Gr 1-40	Dynamic+A	99.9	76.4	4.18	99.5	<u>45.3</u>	5.40	98.1	<u>41.4</u>	5.92	99.3	<u>54.4</u>	5.13
	Dynamic+H	99.6	<u>75.3</u>	4.01	94.5	48.9	4.71	98.9	48.3	5.12	97.1	57.2	4.58
	Base	70.9	49.1	5.94	72.8	42.1	5.94	71.0	24.5	6.99	71.8	40.3	6.18
Owan2 5 14D(CET)	Static+A	84.6	61.4	4.50	78.1	38.6	4.69	77.8	35.6	5.65	80.1	45.3	4.85
Qwen2.5-14B(SFT)	Dynamic+A	85.8	64.9	4.58	78.4	39.9	4.75	78.1	39.0	5.59	80.7	<u>47.7</u>	4.89
	Dynamic+H	82.7	<u>64.6</u>	4.59	81.4	48.9	4.67	78.5	<u>37.4</u>	5.74	81.2	51.3	4.89

Table 2: **Impact of Naviagent Variants on ToolBench.** Base retains only the core agent; StaticA augments with a static graph and Alpha-Beta pruning; DynamicA augments with a dynamic graph and Alpha-Beta pruning; DynamicH augments with a dynamic graph and heuristic pruning, which corresponds to our proposed NaviAgent. Metrics are reported as in Table 1. We also evaluate runtime, with detailed results reported in Appendix F.2. cases for large models such as Deepseek-V3 and GPT-40, highlighting that dynamic graph planning and efficient heuristic search are crucial for unlocking the reasoning and compositional potential of frontier models. Consistent results are also observed on API-Bank (see Table 7). Additional statistics on tool graph structure and link prediction are provided in Table 8.

# 5 CONCLUSION

We presented NaviAgent, a bilevel planning framework that separates high-level decision making from low-level execution over a tool world model, achieving robust gains on ToolBench and API-Bank. It scales to thousands of tools with competitive efficiency and excels in complex, multi-tool tasks and larger models. Remaining challenges include handling heterogeneous tool interfaces and dynamic conditions, which may be tackled via unified protocols and adaptive graph construction. Beyond tool reasoning, NaviAgent points to broader applications: by abstracting tools as agents, its evolving graph and decision space can naturally extend to multi-agent collaboration. This perspective underscores both the challenges of building adaptive, robust systems and the opportunities for advancing toward more collaborative AI ecosystems.

# REFERENCES

- Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 17682–17690, 2024.
- Guoxin Chen, Zhong Zhang, Xin Cong, Fangda Guo, Yesai Wu, Yankai Lin, Wenzheng Feng, and Yasheng Wang. Learning evolving tools for large language models. *arXiv preprint* arXiv:2410.06617, 2024.
- Shen Gao, Zhengliang Shi, Minghang Zhu, Bowen Fang, Xin Xin, Pengjie Ren, Zhumin Chen, Jun Ma, and Zhaochun Ren. Confucius: Iterative tool learning from introspection feedback by easy-to-difficult curriculum. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 18030–18038, 2024.
- Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in Ilms via reinforcement learning. *arXiv* preprint arXiv:2501.12948, 2025.
- Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.
- Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.
- Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.
- Donald E Knuth and Ronald W Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6 (4):293–326, 1975.
- Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. Api-bank: A comprehensive benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244*, 2023.
- Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024a.
- Xukun Liu, Zhiyuan Peng, Xiaoyuan Yi, Xing Xie, Lirong Xiang, Yuchen Liu, and Dongkuan Xu. Toolnet: Connecting large language models with massive tools via tool graph. *arXiv preprint arXiv:2403.00839*, 2024b.
- Yanming Liu, Xinyue Peng, Jiannan Cao, Yuwei Zhang, Xuhong Zhang, Sheng Cheng, Xun Wang, Jianwei Yin, and Tianyu Du. Tool-planner: Task planning with clusters across multiple tools. *arXiv* preprint arXiv:2406.03807, 2024c.
- Zhaoyang Liu, Zeqiang Lai, Zhangwei Gao, Erfei Cui, Ziheng Li, Xizhou Zhu, Lewei Lu, Qifeng Chen, Yu Qiao, Jifeng Dai, et al. Controlllm: Augment language models with tools by searching on graphs. In *European Conference on Computer Vision*, pp. 89–105. Springer, 2024d.
- Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. Chameleon: Plug-and-play compositional reasoning with large language models. *Advances in Neural Information Processing Systems*, 36:43447–43478, 2023.
- Dheeraj Mekala, Jason Weston, Jack Lanchantin, Roberta Raileanu, Maria Lomeli, Jingbo Shang, and Jane Dwivedi-Yu. Toolverifier: Generalization to new tools via self-verification. *arXiv preprint arXiv:2402.14158*, 2024.
  - Xinyi Ni, Qiuyang Wang, Yukun Zhang, and Pengyu Hong. Toolfactory: Automating tool generation by leveraging llm to understand rest api documentations. *arXiv preprint arXiv:2501.16945*, 2025.

- Aaron Parisi, Yao Zhao, and Noah Fiedel. Talm: Tool augmented language models. *arXiv preprint arXiv:2205.12255*, 2022.
- Shuofei Qiao, Honghao Gui, Chengfei Lv, Qianghuai Jia, Huajun Chen, and Ningyu Zhang. Making
   language models better tool learners with execution feedback. arXiv preprint arXiv:2305.13068,
   2023.
  - Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*, 2023.
  - Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. From exploration to mastery: Enabling llms to master tools via self-driven interactions. *arXiv preprint arXiv:2410.08197*, 2024.
  - Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. Tool learning with large language models: A survey. *Frontiers of Computer Science*, 19(8):198343, 2025.
  - Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 3505–3506, 2020.
  - Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.
  - Jonathan Shapiro. Genetic algorithms in machine learning. In *Advanced course on artificial intelligence*, pp. 146–168. Springer, 1999.
  - Weizhou Shen, Chenliang Li, Hongzhan Chen, Ming Yan, Xiaojun Quan, Hehong Chen, Ji Zhang, and Fei Huang. Small llms are weak tool learners: A multi-llm agent. *arXiv preprint arXiv:2401.07324*, 2024.
  - Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36:38154–38180, 2023.
  - Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.
  - Saad Tahmid and Sourav Sarker. Qwen2. 5-32b: Leveraging self-consistent tool-integrated reasoning for bengali mathematical olympiad problem solving. *arXiv preprint arXiv:2411.05934*, 2024.
  - Nima Tajbakhsh, Jae Y Shin, Suryakanth R Gurudu, R Todd Hurst, Christopher B Kendall, Michael B Gotway, and Jianming Liang. Convolutional neural networks for medical image analysis: Full training or fine tuning? *IEEE transactions on medical imaging*, 35(5):1299–1312, 2016.
  - Renxi Wang, Xudong Han, Lei Ji, Shu Wang, Timothy Baldwin, and Haonan Li. Toolgen: Unified tool retrieval and calling via generation, 2024a. *URL https://arxiv. org/abs/2410.03439*, 2024.
  - Xixi Wu, Yifei Shen, Caihua Shan, Kaitao Song, Siwei Wang, Bohang Zhang, Jiarui Feng, Hong Cheng, Wei Chen, Yun Xiong, et al. Can graph learning improve planning in llm-based agents? In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.
  - Yang Xu, Yunlong Feng, Honglin Mu, Yutai Hou, Yitong Li, Xinghao Wang, Wanjun Zhong, Zhongyang Li, Dandan Tu, Qingfu Zhu, et al. Concise and precise context compression for tool-using language models. *arXiv* preprint arXiv:2407.02043, 2024.

- An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.
  - Rui Yang, Lin Song, Yanwei Li, Sijie Zhao, Yixiao Ge, Xiu Li, and Ying Shan. Gpt4tools: Teaching large language model to use tools via self-instruction. *Advances in Neural Information Processing Systems*, 36:71995–72007, 2023.
  - Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023a.
  - Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023b.
  - Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Yongliang Shen, Ren Kan, Dongsheng Li, and Deqing Yang. Easytool: Enhancing llm-based agents with concise tool instruction. *arXiv* preprint *arXiv*:2401.06201, 2024.
  - Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. *Advances in neural information processing systems*, 31, 2018.
  - Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. AI open, 1:57–81, 2020.
  - Yuchen Zhuang, Xiang Chen, Tong Yu, Saayan Mitra, Victor Bursztyn, Ryan A Rossi, Somdeb Sarkhel, and Chao Zhang. Toolchain\*: Efficient action space navigation in large language models with a\* search. *arXiv preprint arXiv:2310.13227*, 2023.

# A GRAPH CONSTRUCTION

Original API	Original Parameter	Parameter Description	Standardized Parameter	Cluster ID
get_locations	name	Name of the city.	city_name	1
get_hospital_list	city	The city where the hospital is located.	city_name	1
get_hospital_list	name	Name of the hospital.	hospital_name	2
find_cheapest_prescription	city_name	The name of the city where the user wants to search for the medication.	city_name	1

Table 3: Standardization of API Parameter

#### B DETAILS OF GRAPH METHOD

#### B.1 HGT NETWORK

This section provides a detailed description of the feature construction, network architecture, and link prediction head used in our heterogeneous graph transformer (HGT) for tool dependency modeling, supplementing the main text.

**Feature Fusion.** Each node v is initialized by its semantic and structural features:

$$\mathbf{h}_v = BGE(x_v) \oplus \sigma(n_v^{succ}) \oplus \sigma(r_v^{succ}) \oplus \sigma(deg_v^{in}) \oplus \sigma(deg_v^{out})$$
(13)

where  $BGE(x_v)$  encodes the node description  $d_v$  using BGE-Large-en-V1.5,  $n_v^{succ}$  and  $n_v^{fail}$  are the counts of successful and failed invocations for node v (computed from historical invocation logs),  $r_v^{succ} = n_v^{succ}/(n_v^{succ} + n_v^{fail})$  denotes the successful ratio, and  $deg_v^{in}$  and  $deg_v^{out}$  are the in-degree and out-degree of node v, respectively.

**Node Encoder.** To project heterogeneous nodes into a unified embedding space, we apply type-specific linear transformations, followed by non-linear activation and normalization:

$$\mathbf{h}_{v}' = LayerNorm\left(LeakyReLU\left(\mathbf{W}_{\tau(v)}\mathbf{h}_{v} + \mathbf{b}_{\tau(v)}\right)\right)$$
(14)

where  $\mathbf{W}_{\tau(v)}$  and  $\mathbf{b}_{\tau(v)}$  are the learnable weight matrix and bias for node type  $\tau(v) \in \{api, param\}$ , respectively.

**WeightedHGTConv Layer.** We stack two multi-head heterogeneous graph transformer (HGT) layers (each with 8 attention heads) to aggregate information from the 2-hop neighborhood. For a center node v and its neighbor  $u \in N_r(v)$  under edge type r, the attention coefficient at head k is computed as:

$$\alpha_{uv}^{(k,r)} = softmax_{u \in \mathcal{N}_r(v)} \left( \frac{(\mathbf{W}_Q^{(k,r)} \mathbf{h}_u')^\top (\mathbf{W}_K^{(k,r)} \mathbf{h}_v')}{\sqrt{d_k}} + \mathbf{b}_r^{(k)} + \tilde{w}_{uv} \right)$$
(15)

where  $\mathbf{W}_Q^{(k,r)}$  and  $\mathbf{W}_K^{(k,r)}$  are the query and key projection matrices for head k and relation r,  $\mathbf{b}_r^{(k)}$  is an edge-type-specific bias,  $\tilde{w}_{uv}$  is the statistical edge weight from node u to v (see Eq. 4, where  $\tilde{w}_{ij}$  is defined for nodes  $v_i$  and  $v_j$ ), and  $d_k = d/8$  is the dimension per head. The normalization  $softmax_{u \in \mathcal{N}_r(v)}$  is performed over all neighbors u of v under relation r. The output embedding for node v:

$$\mathbf{h}_{v}^{"} = LayerNorm\left(\mathbf{h}_{v}^{'} + LeakyReLU\left(\mathbf{W}_{o} \cdot Concat\left[\sum_{r \in R} \sum_{u \in \mathcal{N}_{r}(v)} \alpha_{uv}^{(k,r)} \mathbf{W}_{V}^{(k,r)} \mathbf{h}_{u'}\right]_{k=1}^{8}\right)\right)$$
(16)

where  $\mathbf{W}_{V}^{(k,r)}$  is the value projection for head k and relation r,  $\mathbf{W}_{o} \in \mathbb{R}^{8d_{k} \times d}$  is the output projection, and  $Concat[\cdot]_{k=1}^{8}$  denotes concatenation of outputs from all heads.

**Link Prediction.** Given the final node embeddings, the link probability between node u and node v is computed as:

$$p_{uv} = \sigma \left( \mathbf{W}_p \cdot Concat(\mathbf{h}''_u, \mathbf{h}''_v) + \mathbf{b} \right) \tag{17}$$

where  $\mathbf{W}_p$  and  $\mathbf{b}$  are learnable parameters, and  $\sigma(\cdot)$  denotes the sigmoid function.

This completes the detailed description of our HGT-based network architecture.

#### B.2 GRAPH SEARCH ALGORITHM

This section provides detailed descriptions of the Alpha-Beta pruning and hybrid heuristic search algorithms, including all parameter settings, dynamic thresholding strategies, and algorithmic pseudocode.

Alpha-Beta Pruning. This algorithm Knuth & Moore (1975) is adapted for backward search over the tool dependency graph  $\mathcal{G}=(V,E,W)$ , parameterized by a quintuple  $(\alpha,\beta,\mathcal{H},\mathcal{D},\mathcal{C})$ , where  $\alpha\in\mathbb{R}^+$  (initialized as  $\alpha_0=0.4$ ) is the lower-bound threshold for acceptable path scores, and  $\beta\in\mathbb{R}^+$  (with  $\beta_0=0.9$ ) is the upper-bound for candidate evaluation. The dynamic threshold function  $\mathcal{H}(d)=\max(0.3,0.5\times0.9^d)$  applies exponential decay to balance search depth d and semantic relevance. The depth attenuation factor  $\mathcal{D}(d)=1/(1+\sqrt{d})$  penalizes longer paths. The connectivity constraint  $\mathcal{C}(u,v_t)=\mathrm{PathLength}(u,v_t)\leq 5$  ensures that generated subgraphs remain compact, where  $v_t$  denotes the target node (either an API node or a parameter node). The parametric scoring function is defined as:

$$S_{uv} = \frac{w_{uv} + \mathbb{I}(u \to v_t^{\text{api}}) w_{u \to v_t^{\text{api}}} + \mathbb{I}(u \to v_t^{\text{param}}) w_{u \to v_t^{\text{param}}}}{3} \times \mathcal{D}(d)$$
(18)

where  $w_{uv}$  is the direct edge weight from node u to its predecessor v (see Section 3.2.2),  $w_{u \to v_t^{\rm api}}$  and  $w_{u \to v_t^{\rm param}}$  denote the edge weights from u to the target API node  $v_t^{\rm api}$  and target parameter node  $v_t^{\rm param}$ , respectively, included only if the corresponding indicator function  $\mathbb{I}(\cdot)$  is active.

During reverse depth-first search, we apply two pruning rules: Alpha-pruning is triggered at parameter nodes when  $S_{uv} < \mathcal{H}(d)$  and  $S_{uv} < \alpha$ , while Beta-pruning is triggered at API nodes when  $S_{uv} > \beta$ . To further improve efficiency, the pruning thresholds are dynamically adjusted via  $\alpha' = \max(\alpha, S_{uv} \times 0.85)$  and  $\beta' = \min(\beta, S_{uv} \times 1.15)$ , reducing the search time complexity from  $O(b^k)$  to  $O((\sqrt{b})^k)$  Knuth & Moore (1975), where b is the branching factor and k is the maximum search depth. See Algorithm 1 for details.

Heuristic Graph Search with Dynamic Pruning. Our hybrid heuristic search algorithm combines simulated annealing Kirkpatrick et al. (1983) and genetic algorithm strategies Shapiro (1999). It is parameterized by a sextuple  $(\mathcal{T}_0, \eta, \mathcal{P}, d_{\max}, \mathcal{M}_{\theta}, \mathcal{F}_{\omega})$  (see Algorithm 2), where  $\mathcal{T}_0 = 200$  is the initial temperature that determines the probability of accepting suboptimal solutions and balances exploration and exploitation,  $\eta = 0.7$  is the cooling rate that controls the annealing schedule  $\mathcal{T}_{k+1} = \eta^{1+k/5} \mathcal{T}_k$ ,  $\mathcal{P} = 20$  is the population size,  $d_{\max} = 4$  is the maximum search depth, and  $\mathcal{M}_{\theta}$  is a temperature-sensitive mutation operator with adaptive intensity  $\theta = \lfloor \mathcal{T}/100 \rfloor$ . Candidate solutions are evaluated using a composite fitness function:

$$\mathcal{F}_{\omega} = 0.35\mathcal{C}_c + 0.15\log(1+\rho_p) + 0.3\mathcal{D}_c + 0.15\mathcal{W}_n + 0.05\mathcal{C}_p \tag{19}$$

where  $\mathcal{C}_c$  (node compactness) measures the closeness centrality of API nodes,  $\rho_p$  (parameter density) is the ratio of parameter nodes within the subgraph to promote concise yet informative solutions,  $\mathcal{D}_c = 0.2e^{-d/10} + 0.8e^{-n/8}$  (depth penalty) penalizes overly deep or complex dependency structures, with d as the average depth and n as the total node count,  $\mathcal{W}_n$  (weight quantification) encourages solutions with higher cumulative edge weights, and  $\mathcal{C}_p$  (path complexity) evaluates structural simplicity, favoring solutions with less intricate connectivity.

We parallelize the subgraph search for different target APIs in Algorithm 2. This approach processes the population evolution tasks independently and concurrently, thereby eliminating the computational bottleneck of the original algorithm's serial loops.

```
Algorithm 1 Alpha-Beta Backward Pruning
   Input G, v_{target}, \alpha_{init} = 0.4, \beta_{init} = 0.9, d_{max} = 5
  Output G_{sub}
  Initialize queue Q with v_{target}, V_{visited} = \{v_t\}
   while Q not empty do
       v = Q.pop()
       for p \in predecessors(v) do
            if p not in V_{visited} then
                s = Score(p \rightarrow u, p \rightarrow v_{target}, p \rightarrow v_{target\_param})
                d = current_depth(p)
                \mathcal{H}(d) = \max(0.3, 0.5 \times 0.9^d)
                if p \in V_{param} then
                     if s < \mathcal{H}(d) \land s < \alpha then
                         continue
                     end if
                     if s > \beta then
                         break
                     end if
                end if
                \alpha = max(\alpha, s \times 0.85)
                \beta = max(\beta, s \times 1.15)
                V_{visited}.add(p)
                Q.append(p)
            end if
       end for
  end while
   V_{sub} = \{v | v \in V_{visited} \land PathLength(u, v_{target}) \le 5\}
  return G_{sub} = (V_{sub}, E)
```

Figure 6: Alpha-Beta Backward Pruning

# C CASES

756

757

758

759

760

761

762

763

764

765

766

767 768

769

770

771

772

773

774

775

776

777

778

779

781

782

783 784

785 786

787 788

789

790

791

792

793

794

795

796 797 798

799

800 801

802

803 804

805

806

807

808

The following three cases exemplify the bilevel planning mechanism through four core actions: 1) *Direct Response*: resolves user queries using pre-trained knowledge. 2) *Intent Clarification*: initiates interactive dialogue to disambiguate vague requests. 3) *ToolChain Retrieval*: works with the TWNM to construct a pruned tool dependency subgraph, which is then returned as an executable toolchain. 4) *Tool Execution*: executes the required APIs based on the dependency subgraph, with parameter validation and state monitoring. This design achieves centralized decision control through the agent's orchestration authority while enabling dynamic resource optimization via the TWNM's graph-based toolchain generation, ensuring both efficiency and robustness of the our framework in complex task environments.

# C.1 CASE 1

# QUERY

• Could you provide me with information about gastroenteritis? Additionally, please help me log my health data from March 5, 2025.

FIRST ROUND: INTENT ANALYSIS

1. Gastroenteritis Inquiry
 ACTION: Direct Response
 CONTEXT:
 {
 "analysis": "1. Common sense judgment: Gastroenteritis is a

```
Algorithm 2 Hybrid Heuristic Pruning Algorithm
811
        Require: Dependency graph G, target API set A, initial temperature \mathcal{T}_0 = 200, cooling rate \eta = 0.7,
812
           population size \mathcal{P}=20, maximum search depth d_{max}=4
813
        Ensure: Optimized dependency subgraph G^*
814
           Initialize optimized subgraph set S \leftarrow \emptyset
815
         2: for each target API a \in A do
816
               Set temperature \mathcal{T} \leftarrow \mathcal{T}_0
817
               Generate initial population Pop (size \mathcal{P}) for API a
818
               Set iteration count k \leftarrow 0
         6:
               while T > 1 and k \le 10 do
819
                  Evaluate fitness \mathcal{F}_{\omega} for each chromosome in Pop
820
         8:
                  Select elite chromosomes (top 60% based on fitness)
821
                  Generate offspring via crossover operation
822
                  Apply temperature-sensitive mutation \mathcal{M}_{\theta} with intensity \theta = |\mathcal{T}/100|
        10:
823
                  Update population Pop with offspring
824
                  Update temperature: \mathcal{T} \leftarrow \eta^{1+k/5} \mathcal{T}
        12:
825
                  Increment iteration count k \leftarrow k+1
826
        14:
               end while
827
               Select best chromosome from Pop based on \mathcal{F}_{\omega}
828
        16:
               Decode chromosome and build API-specific subgraph G_a
829
               Add subgraph G_a to optimized subgraph set S
830
        18: end for
           Merge all subgraphs in S into final optimized subgraph G^*
831
        20: return G^*
832
833
834
                                Figure 7: Hybrid Heuristic Pruning Algorithm
835
                medical condition within general knowledge. 2. No API
836
                required for definition explanation. 3. Parameter
837
                verification not applicable",
838
                "action": "direct_answer",
839
                "intent": "1. Provide medical definition of gastroenteritis
840
                2. User's explicit request for condition explanation",
841
                "target_api": "",
842
                "params": {},
843
                "recall description": "",
                "answer": "Gastroenteritis is inflammation of the stomach
844
                and intestines,
845
                typically resulting from bacterial toxins or viral
                infection. Common
847
                symptoms include diarrhea, vomiting, and abdominal cramps."
848
                }
849
850
851
             2. Health Data Logging
852
                ACTION: TooChain Retrieval
853
                CONTEXT:
854
855
                "analysis": "1. Health data recording requires specialized
856
                API 2. No matching API in candidate list 3. Critical
857
                parameters provided:
858
                user_id=12345, blood_pressure=130/80, heart_rate=90,
859
                timestamp=2025-03-05",
860
                "action": "retrieve_api",
                "intent": "1. Store user health metrics 2. Provided
861
                parameters: blood pressure, heart rate, user ID, timestamp",
862
```

"target\_api": "",
"params": {},

 CONTEXT:

"action": "call\_api",

```
"recall_description": "HealthDataAPI(description: Records
user health
metrics, input: user_id:string/unique identifier,
blood_pressure:string/systolic & diastolic,
heart_rate:integer/bpm,
timestamp:datetime/measurement time; output:
record_id:string/new entry identifier)",
"answer": "I need to retrieve a health data recording API to
complete this action. Should I proceed with API retrieval?"
}
```

Then, the BGE model retrieves two candidate APIs (api-RecordHealthData, api-QueryRegistration) based on the recall description above, after which the TWNM is invoked to acquire the corresponding call chains for these APIs (see Figure 8).

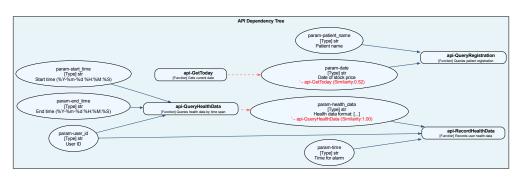


Figure 8: Pruned Tool Dependency Subgraph of Case1

# SECOND ROUND: ITERATIVE API CHAIN EXECUTION

In this round, the agent selects and executes the optimal API chain, with api-RecordHealthData designated as the target API.

```
ACTION: Tool Execution CONTEXT:
```

```
896
897
898
         "action": "call_api",
899
         'api_name': 'api-QueryHealthData'
         "params": {
900
           "user id": "12345",
901
           "start_time": "2025-03-05",
902
           "end time": "2025-03-05"
903
904
      }
905
906
907
      TOOL RESPONSE:
908
      {{'name':'blood_pressure','value':'130/80'}
909
      {'name':'heart_rate','value':'90'}}
910
911
      THIRD ROUND: ITERATIVE API CHAIN EXECUTION
912
913
914
      ACTION: Tool Execution
```

```
918
         'api_name': 'api-RecordHealthData'
919
         "params": {
           "user_id": "12345",
921
           "time": "2025-03-05 14:30:00",
922
           "health_data": "[{'name':'blood_pressure','value':'130/80'},
                             {'name':'heart_rate','value':'90'}]"
923
924
      }
925
926
927
      TOOL RESPONSE:
928
929
      Finished
930
931
      FOURTH ROUND: API OUTPUT VERBALIZATION
932
      ACTION: Direct Response
933
      CONTEXT:
934
935
936
         "analysis": "1. API parameters verified and complete
937
                       2. Successful data storage confirmation",
938
         "action": "direct_answer",
939
         "answer": "Your health data has been successfully recorded..."
940
941
      LAST ROUND: RESPONSE SYNTHESIS
942
943
      ACTION: Direct Response
944
      CONTEXT:
945
946
      1. Inflammation of gastrointestinal tract
947
           Common causes: Viral/bacterial infections
948
           Key symptoms: Diarrhea, vomiting, abdominal cramps
          Management: Hydration and rest
949
950
      2. Health Data Record Confirmation:
951
           All parameters validated successfully through api-
952
           RecordHealthData. No data integrity issues detected.
953
954
      C.2 CASE 2
955
956
      OUERY
957
           • Is Beijing in China? What is the current weather forecast for the location at latitude 40.7128
958
             and longitude -74.0060 in metric units?
959
960
      FIRST ROUND: INTENT ANALYSIS
961
962
           1. Common Sense
963
             ACTION: Direct Response
             CONTEXT:
964
965
             {'dependency_rank': 1,
966
               'intent': 'Is Beijing in China?',
967
               'analysis': '1. The specific problem addressed by this
968
               action is to determine if Beijing is located within China.
969
               2. The user input directly asks about the
               geographical location of Beijing. 3. This is a common
970
               sense question and does not require an API call. 4. No
               parameters are needed for this decision.',
```

```
972
              'action': 'direct_answer',
973
              'answer': 'Yes, Beijing is in China.'}
974
975
          2. Get Weather
976
            ACTION: TooChain Retrieval
977
            CONTEXT:
978
979
            {'dependency rank': 2,
980
            'intent': 'What is the current weather forecast for the
            location at latitude 40.7128 and longitude -74.0060 in
981
            metric units?',
982
            'analysis': '1. The specific problem addressed by this
983
            action is to retrieve the current weather forecast for a
984
            specific location using latitude and longitude coordinates.
985
            2. This is not a common sense question and requires an
986
            external API call. 3. The API matching phase will prioritize
987
            APIs that can provide weather forecasts based on geographic
988
            coordinates. 4. Required parameters are latitude and
989
            longitude, which are provided in the user input.',
990
            'action': 'retrieve_api',
991
            'recall description': 'WeatherAPI (description: Retrieve
            current weather conditions by geographic coordinates, input:
992
            latitude:float/latitude coordinate;
993
            longitude:float/longitude coordinate; output:
994
            temperature:float/current temperature;
995
            humidity:float/current humidity;
996
            wind_speed:float/current wind speed)'}
```

Then, the BGE model retrieves three candidate APIs (api-current\_weather\_data\_of\_a\_location\_for\_weather, api-by\_geographic\_coordinates\_for\_rapidweather, api-current\_for\_foreca\_weather) based on the recall description above, after which the TWNM is invoked to acquire the corresponding call chains for these APIs (see Figure 9).

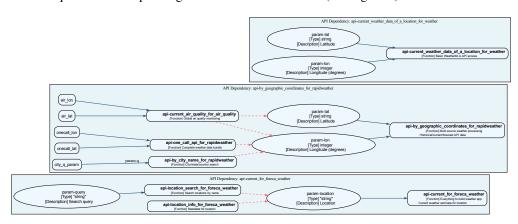


Figure 9: Pruned Tool Dependency Subgraph of Case2

SECOND ROUND: ITERATIVE API CHAIN EXECUTION

In this round, the agent selects and executes the optimal API chain, with api-by\_geographic \_coordinates\_for\_rapidweather designated as the target API.

```
ACTION: Tool Execution CONTEXT:
```

```
"action": "call_api",
1027
        'api_name': 'api-by_geographic_coordinates_for_rapidweather'
        'params': {'param-lat': '40.7128', 'param-lon': '-74.0060'}
1029
1030
1031
      TOOL RESPONSE:
1032
1033
      {temperature:11.0°C, feels:8.0°C, wind speed:5.1 m/s}
1034
1035
      THIRD ROUND: API OUTPUT VERBALIZATION
1036
1037
      ACTION: Direct Response
      CONTEXT:
1038
1039
1040
        'analysis': "1. The specific problem addressed by this
1041
        action is to retrieve the current weather forecast for a
1042
        specific location using the provided latitude and
1043
        longitude. 2. The user input directly provides the
1044
        necessary parameters (latitude and longitude) and
        specifies the need for metric units.
1046
        3. The target API 'api-
        current_weather_data_of_a_location_for_weather' has
1047
        already been successfully called and its result is
1048
        available. 4. The required parameters (lat and lon) are
        complete and the API call has been confirmed
1050
        successful.",
1051
        'action': 'direct_answer',
1052
        'answer': 'The current weather in New York is clear sky
1053
        with a temperature of 11.0°C, feels like 8.0°C, and the
1054
        wind speed is 5.1 m/s from the west-southwest.' }
1055
1056
      LAST ROUND: RESPONSE SYNTHESIS
1057
      ACTION: Direct Response
1058
      CONTEXT:
1059
      Beijing is in China. The current weather in New York is
1061
      clear sky with a temperature of 11.0°C, feels ike 8.0°C, and
1062
      the wind speed is 5.1 m/s from the west-southwest.
1063
1064
      C.3 CASE 3
1065
1066
      OUERY
           • Could you tell me what the weather will be like tomorrow?
1068
1069
      FIRST ROUND: INTENT ANALYSIS
1070
          1. Intent Clarification
1071
            ACTION: Intent Clarification
1072
            CONTEXT:
1073
1074
1075
            "analysis": "1. Weather data needs to be queried in real
1076
            time -> not common sense\n2. Required parameter (location)
1077
            is missing",
             "action": "clarify_intent",
1078
             "recall_description": "",
             "answer": "Which city do you want to query tomorrow's
```

SECOND ROUND: INTENT ANALYSIS

#### 1. Get Weather

# ACTION: **TooChain Retrieval** CONTEXT:

```
{'dependency_rank': 1,
 'intent': '1. Query the weather forecast for tomorrow in
 Beijing 2. Extract location: Beijing and time: tomorrow
 from user input',
 'analysis': '1. The specific problem addressed by this
 action is to retrieve
 the weather forecast for tomorrow in Beijing. 2. The user
 input directly provides the location (Beijing) and the
 time frame (tomorrow). 3. Since no candidate APIs are
 available, the system needs to retrieve an appropriate API
 for weather forecasting. 4. There are no parameters
 provided by the user that can be used directly with an
 API, so the system must retrieve an API that can
 accept location and time as parameters.',
 'action': 'retrieve_api',
 'recall_description': 'WeatherForecastAPI (description:
 Retrieve weather forecast for a given location and date,
 input: location:string/city name;
 date:date/forecast date; output: weather:string/weather
 condition, temperature:float/forecast temperature)'}
```

Then, the BGE model retrieves three candidate APIs (api-getweatherforecast\_for\_apjoy\_weather\_forecast, api-weather\_report\_for\_the\_weather\_api, api-location\_info\_for\_foreca\_weather) based on the recall description above, after which the TWNM is invoked to acquire the corresponding call chains for these APIs (see Figure 10).

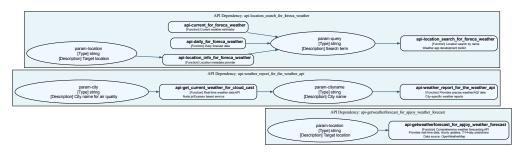


Figure 10: Pruned Tool Dependency Subgraph of Case3

THIRD ROUND: ITERATIVE API CHAIN EXECUTION

In this round, the agent selects and executes the optimal API chain, with apigetweatherforecast\_for\_apjoy\_weather\_forecast designated as the target API.

ACTION: **Tool Execution** CONTEXT:

```
1134
1135
        "action": "call_api",
1136
        'api_name': 'api-getweatherforecast_for_apjoy_weather_forecast'
1137
        'params': {'location': 'Beijing'}
1138
1139
      TOOL RESPONSE:
1140
1141
      {temperature:25°C, humidity:60%}
1142
1143
      LAST ROUND: API OUTPUT VERBALIZATION
1144
1145
      ACTION: Direct Response
      CONTEXT:
1146
1147
        The weather forecast for tomorrow in Beijing is partly
1148
        cloudy with a temperature of 25°C and humidity of 60%.
1149
1150
         DATA GENERATION
1151
      D
1152
      Problem and API Dependency Chain Generation Prompts: Automatically construct a chain of
1153
      calls with strict parameter matching based on a randomful tree structure. This process involves
1154
      tracing initial parameters from the leaf nodes and validating results at the root node to ensure that the
1155
      generated problem aligns with the authentic API dependency logic of real-world scenarios.
1156
1157
      1. Core Requirements:
1158
          - Generate a natural-language question where:
1159
            · Must explicitly contain initial parameters for leaf-
1160
            node APIs
1161
            · Implicitly requires chained API calls from leaf to
1162
            root node
1163
            • Root node API's output directly resolves the user's
            problem
1164
1165
      2. Dependency Chain Rules:
1166
          - Build parameter-passing paths where:
1167
            · Parent API outputs must exactly match child API inputs (same
1168
              parameter names & data types)
1169
            · Root node API must be called last in the chain
1170
            · The output of every leaf-node API must be utilized in
1171
            downstream
1172
              APIs or final results.
1173
            · All input values must originate from either:
1174
               Explicitly stated in the question context
               Generated by previous API outputs (no synthetic values)
1175
1176
      3. Parameter Constraints:
1177
          - Enforce strict value inheritance:
1178
            · Path/query parameters must use verbatim values from:
1179
              - User's question text
1180
              - Preceding API response.data fields
1181
            • Prohibit value transformation/format conversion
1182
          - Root API output must contain realistic values matching
1183
         its schema
1184
      4. Validation Requirements:
1185
          - Reject generation if:
1186
            · Missing parameter dependency between APIs
1187
```

• Input sources can't be traced to question/prior responses

```
1188
            · Output fields don't fulfill next API's input requirements
1189
1190
      5. Response Structure:
1191
1192
        "query": "<Real-world scenario requiring sequential API
        calls>",
1193
        "answer": "<Solution derived from root API output>",
1194
        "call_chains": [
1195
1196
             "api name": "<Leaf-node API>",
1197
             "input": {
1198
               "<param>": "<value explicitly stated in user query
1199
               or previous API output>"
1201
             "output": {
               "status": "success"
               "data": {"<field>": "<output used by next API>"}
1203
             }
          },
             "api_name": "<Root-node API>",
1207
             "input": {
1208
               "<param>": "<value from previous API output>"
1209
1210
             "output": {
1211
               "status": "success",
1212
               "data": {"<field>": "<realistic resolution to
1213
               query>"}
1214
1215
1216
        ]
1217
      The API dependency tree structure is as follows:
1218
1219
```

### **E** IMPLEMENTATION DETAILS

#### E.1 DATASET

This table 4 system displays the sample counts of the ToolBench and API-Bank datasets, as well as the distribution of their difficulty levels.

Dataset	Easy	Medium	Hard	Total
API-Bank	57	176	211	444
ToolBench	148	208	105	461

Table 4: Dataset Samples and Difficulty Distribution

# E.2 TRAINING

We fine-tuneTajbakhsh et al. (2016) our model using Qwen2.5-14B model with full parameter tuning. The model is trained with a maximum sequence length of 8192. We utilize a learning rate of 2e-5 and employ the AdamW optimizer with a cosine learning rate scheduler. The training process includes 10 epochs with a per-device batch size of 1 for both training and evaluation. Gradient checkpointing is enabled to reduce memory usage, and gradient accumulation is set to 4 steps to effectively manage smaller batch sizes. We apply a weight decay of 0.01 and set the maximum gradient norm to 1 for stable training. A warmup ratio of 0.03 is used to gradually increase the learning rate at the beginning of training. The training is executed on 8 Ascend 910B 64G GPUs within 10 hours. The DeepSpeedRasley et al. (2020) library is leveraged for efficient distributed training.

```
1242
      E.3 INFERENCE
1243
1244
      E.3.1 NAVIAGENT INFERENCE PROMPTS
1245
1246
      Inference prompts are based on intent decomposition and dependency prioritization to achieve
1247
      automatic parameter completion and error handling. They generate standardized JSON responses
      through hierarchical decision-making.
1248
1249
1250
      You are an intelligent API coordination system. Respond
      strictly according to the following rules:
1251
1252
      # Decision Architecture
1253
      1. **Intent Analysis**
1254
         - Decompose compound requests into independent ordered
1255
         sub-intents
1256
           · Sequential dependencies first, Must execute in
1257
           declared order
1258
            · Parallelizable sub-intents last
1259
            • Dependency_rank numbering for ordered execution
         - Validate parallel execution eligibility:
1260

    No overlapping data requirements

            · No sequential dependencies
1262
            · Distinct parameter sets
1263
1264
      2. **Atomic Action Formation**
1265
           · For each validated sub-intent:
              - Create self-contained decision unit, action must
1267
             implement full
1268
             Decision Logic Flow
1269
              - Maintain state separation between parallel processes
1270
              - Focus analysis scope per sub-intent
1271
              - Each action's analysis focuses only on its own
             intent
1272
             - Each action analysis only solves one intent
1273
              - Must execute each action in declared order
1274
1275
      # Decision Logic Flow
1276
      1. **Common Sense Judgment Phase**
1277
         - Input question -> Knowledge base matching
1278
          Belongs to common sense -> action=direct_answer
1279
          Requires external data -> Proceed to Phase 2
1280
1281
      2. **API Matching Phase**
         1. If candidate_apis is empty -> action=retrieve_api
         2. Match intent with API list:
           API prioritization:
1284
                 - Complete parameters from user input
1285
                 - Minimal missing parameters
1286
                 - Shortest dependency chain
1287
           API matching success:
1288
               - Validate Observation in user input to confirm
1289
               target API success:
1290
                  -> If successful -> action=direct_answer
1291
                  -> No explicit success indication:
1292
                   a) Complete parameters -> action=call_api
                   (execute based on 3.1 dependency resolution)
1293

    If Rule 3.1c applies -> action=direct_answer

1294
                   b) Missing parameters -> Proceed to Phase 3
1295
```

API matching failed -> action=retrieve\_api

```
1296
1297
      3. **Parameter Completion Phase**
         - Check required parameter set:
1299
           All parameters ready -> action=call_api
           The target API does not require parameters -> action=call_api
1301
           Missing parameters exist:
             a) Can be completed via dependent APIs -> Execute
1302
             Rule 3.1
1303
             b) Use Retrieval APIs resolve parameter deficiencies
1304
             in API
1305
                dependencies -> action=retrieve_api
1306
             c) Requires human confirmation -> action=clarify_intent
1307
1308
      # Technical Rule
1309
      ## 3.1 Dependency Resolution Rules
1310
          a) Check required parameters of target API, first call
1311
          dependent APIs.
1312
          b) For each missing parameter, select APIs from
          dependencies not marked
1313
             as failed.
1314
          c) If an input parameter of an API is unavailable, use
1315
          retrieve_api to
1316
             call another API that generates it from known parameters.
1317
             -> action=retrieve_api
1318
          d) Success propagation: Completed dependency chain
1319
             -> action=direct_answer
1320
1321
      ## 3.2 Known Failure Handling
1322
          a) Failed APIs are recorded in failed_apis
          b) Prioritize non-failed candidate APIs
1323
1324
      # Response Specification (Mandatory JSON Format)
1325
1326
        "dependency_rank": 1,
1327
        "intent": "1. <precisely describe the specific problem
1328
        addressed by the current action>
1329
                    2. <extract data segments directly related to
1330
                    the subtask from user input>",
1331
        "analysis":
1332
              "<Four-level reasoning:
1333
              1. Explicitly state the specific decision-making sub-
1334
              intent
                addressed by this action
1335
              2.Common sense judgment basis
              3.API matching logic (if applicable)
1337
              4.Parameter completeness verification>",
1338
        "action": "call_api|direct_answer|retrieve_api|clarify_intent",
1339
        "target_api": "API name (mandatory for call_api)",
1340
        "params": { "parameter": "value (mandatory for call api) "},
1341
        "recall_description":
1342
              "When action=retrieve_api: Use 'APIName(description:
1343
              API functionality, input: param:type/description;
1344
              output:
1345
              param:type/description)' format with only core
1346
              parameters (e.g.,
              'StockAPI (description: Query stock price by symbol,
1347
              input: symbol:string/stock symbol; output:
1348
              price:float/current price)')",
1349
        "answer": "When actionin[direct_answer,clarify_intent]:
```

```
1350
        Natural language response (interrogative sentences
1351
        required) "
1352
1353
1354
      # Response Specification
      Added constraint:
1355
      - JSON array items MUST be sorted by dependency_rank in
1356
      ascending order
1357
      - Sibling sub-intents should have consecutive ranks
1358
1359
      # Termination Conditions
1360
      [OR] Generate final answer
1361
      [OR] Target API must be executed successfully, as shown in
1362
      the status
1363
1364
      # Enforcement Constraints
      1. Parameter names must strictly match API documentation
      2. The 'answer' field for clarify_intent must contain
1366
      question words
1367
      3. Prioritize calling parent node APIs
1368
      4. When action in [retrieve_api]:
1369
          - The recall_description field serves exclusively as an
1370
          API retrieval identifier from predefined repositories.
1371
          - parameter descriptions must distinguish between input
1372
          and output parameters, retaining only essential
1373
          parameters
1374
          - Each recall_description can only recall one
1375
          api, multiple APIs require
1376
            multiple actions.
      5. APIs absent from Candidate APIs MUST NOT be invented
1377
      6. When action=call_api is permitted only when candidate
1378
      APIs exist and the target_api is present in the candidate
1379
      APIs.
1380
      7. The "action" field must be strictly limited to one of the
1381
      following four predefined operation types: call_api,
      direct_answer, retrieve_api or clarify_intent.
1383
      8. Use retrieve_api only when:
1384
          - Required parameters unavailable in call_api action
1385
      9. Use call_api only when:
1386
          - The target_api is not in the list of successfully
1387
          executed APIs
1388
      # Candidate API Information:
1389
1390
1391
1392
      E.3.2 INPUT GENERATION PROMPTS
1393
1394
      Input generation prompts: Integrate current queries with observational data to formulate the final
1395
      input, ensuring informational completeness.
1396
1397
1398
      User input: {user_input} \nPlease generate the final response
1399
      based on the following data:
1400
      {observation} :
1401
          Requirements:
          1. Integrate all available data
1402
          2. Indicate data limitations (if any failed APIs exist)
1403
```

3. Use natural and fluent English

```
1404
      E.3.3 API SIMULATOR PROMPTS
1405
1406
      API simulator prompts are based on historical data reuse (Case1) and intelligent simulation gen-
      eration (Case2/3). They achieve automated emulation of API chains through standardized JSON
1407
      responses. The priority strategy is as follows: historical matching > structural cloning > contextual
1408
      simulation.
1409
1410
      Act as an API Chain Simulator to generate responses based on
1411
      historical call chains.
1412
      Follow these rules strictly:
1413
1414
      Operation Rules:
      1. Request Processing Logic
1415
          - CASE 1: Existing API + Identical Inputs
1416
             · Return historical outputs verbatim
1417
             • Set {"status": "success", "type": "success"}
1418
          - CASE 2: New API
1419
             · Create mock data matching input format using:
1420
               - Similar outputs from call chain (priority)
1421
               - Simulated values (fallback)
1422
             Set {"status": "success", "type": "mock"}
          - CASE 3: Error
1424
            • If not correct
1425
             • Set {"status": "success", "type": "error"}
1426
      2. Response Requirements:
1427
          · Strictly use JSON format only
1428
          · Never explain parameter sources or chain structure
1429
          • Never ask follow-up questions
1430
          · Maintain consistent parameter naming conventions
1431
1432
      3. Output Format (JSON):
1433
1434
         "status": "<success>", // Always 'success' per operation
1435
         completion
1436
         "data": <output_parameters>,
         "type": "<success/mock/error>"
1437
1438
1439
      Implementation Notes:
1440
       1. Priority Order:
1441
          History Match > Structural Clone > Contextual Moc
1442
1443
      API call chain is as follows:
1445
      E.3.4 SIMULATED USER RESPONSE AGENT PROMPTS
1446
      Simulated user response agent prompts: Utilize a parameter extractor as the user response to
1447
      agent, serving as a simulated responder for follow-up questions by the agent. Strictly adhere to the
1448
       parameter records of the API call chain to return only the queried and existent original parameter
1449
      values. Automatically filter out uninvoked or null parameters to ensure that the responses include
1450
      only the actual request information from the existing chain of calls.
1451
1452
      As an API chain parameter extractor, directly return exact
1453
      parameter values from the given API workflows without any
1454
      modification.
1455
```

Always return raw parameter values from the latest API

## Mandatory Protocols

1. Parameter Extraction Priority

1456

```
1458
          call
1459
          Return empty string for blank parameters (e.g. param-
1460
          cuisines_1 -> "")
1461
1462
      2. Response Requirements
1463
          Merge multiple parameters in single response
           Example: "patient_id:[value] cuisine:[value]"
1464
          Strictly avoid explanations or disclaimers
1465
          Never reveal API structure or workflow logic
1466
1467
      ## Critical Examples
1468
      User: What's the patient ID and dietary preferences?
1469
      API Context: [param-patient_id_10:'P123' ...]
1470
      Response: patient_id:P123''
1471
1472
      User: Current trial phase and calories limit?
1473
      API Context: [param-trial_phase_1:'Phase 2' param-
      calories_max_1:'2000'...]
1474
      Response: phase:Phase 2 calories_max:2000
1475
1476
      User: How to activate international roaming?
1477
      API Context: Relevant records
1478
      Response: I don't Know international roaming activation
1479
      information.
1480
1481
      ## Execution Context
1482
      Current API call chain:
1483
1484
      E.4 EVALUATION
1485
1486
      E.4.1 EVALUATION PROMPTS
1487
      Evaluation prompts in GPT-4.1 are designed to assess the correctness of the answer generation
1488
      process, logical consistency, and accuracy of responses by analyzing the anticipated pathways and
1489
      the decision-making pathways of the agent.
1490
1491
      As an expert in response quality evaluation, you need to
1492
      perform the following steps:
1493
      I. Core Information Comparison Requirements
1494
      1. Reference Path Analysis
1495
      - Understand the simulated nature of reference API call
1496
      paths.
1497
      - Be aware of potential discrepancies: API names/parameter
1498
      formats may differ from actual implementations.
1499
      2. Actual Path Verification
1500
      - Compare each actual call path with the reference path.
1501
      - Focus on logical coherence rather than exact matching.
1502
1503
      II. Error Detection Standards
1504
      1. Call Process Errors
1505
       Parameter Anomalies:
1506
        * Includes fictitious or illegal parameters.
1507
       Execution Errors:
1508
        * Returns error codes (e.g., 5xx) or invalid responses.
1509
      2. Information Integrity Errors
1510
      Deviation in Answers:
1511
        * Fails to address the core user query accurately.
```

```
1512
       Missing Key Information:
1513
        * Lacks necessary data items or explanation steps.
1514
1515
      III. Correctness Determination Rules
1516
      1. Process Compliance
      - Call sequence should be logically consistent.
1517
1518
      2. Answer Completeness
1519
      - Covers all core elements of the user's question.
1520
      - Output provides a sufficient amount of information.
1521
1522
      IV. Quality Rating System
1523
      [1] High-Quality Standard:
1524
      * Complete logical coherence in call paths.
1525
      * Output results are accurate and effective.
1526
      * No technical errors.
1527
      [0] Deficiency Standard (if any condition is met):
1528
      * Critical API call failures.
1529
      * Returned results do not support the answer.
1530
      * Presence of unaddressed critical errors.
1531
1532
      V. Output Specifications
1533
      1. Detection Report Format:
1534
          1. Parameter Validation -> Compliant/Non-compliant
1535
          2. Path Verification -> Compliant/Non-compliant
1536
          3. Result Completeness -> Compliant/Non-compliant
1537
1538
      2. Final Conclusion Format:
      {'Quality Result': 1} or {'Quality Result': 0}
1539
1540
      VI. Input Data Interface
1541
      User Question: {question}
1542
      [AGENT Answer Start]
1543
      {reference}
1544
      [AGENT Answer End]
1545
      [Reference Call Path]
1546
      {reference_chain}
1547
      [Reference Call Path End]
1548
      [Actual Call]
1549
      {agent_actual_chain}
      [Actual Call End]
1550
1551
1552
1553
1554
         EXPERIMENTS
1555
```

# F.1 EVALUATING OUR APPROACH ON REAL-WORLD APIS

1556 1557

1558 1559 1560

1561

1562

1563

1564

1565

To further validate our framework, we conducted real-world evaluations on 50 APIs from RapidAPI, covering weather, air quality, restaurants, real estate, geolocation, hotels, and sports. A total of 60 queries (20 easy, 20 medium, and 20 hard) were carefully designed to ensure comprehensive coverage across these domains. On this real-world testbed, our framework consistently outperformed the  $\alpha$ -UMI baseline in both effectiveness and efficiency. Metrics are reported as in Table 5 and Time is measured in seconds (s)

Model	Method	TCR	TSR	Steps	Time
	ReAct	31.7	21.7	3.70	16
O2 5 14D	ToolLLM	53.3	23.3	4.10	19
Qwen2.5-14B	$\alpha$ -UMI	76.7	31.7	5.80	27
	Dynamic+H	65.0	<del>36.7</del>	4.90	25
	ReAct	35.0	25.0	3.80	19
Overage 2 5 22D	ToolLLM	48.3	30.0	4.00	23
Qwen2.5-32B	$\alpha$ -UMI	78.3	41.6	6.04	32
	Dynamic+H	86.7	53.3	4.84	27
	ReAct	55.0	33.3	3.75	23
DC1- V/2	ToolLLM	53.3	35.0	4.05	25
DeepSeek-V3	$\alpha$ -UMI	85.0	48.3	6.17	39
	Dynamic+H	98.3	63.3	5.11	35

Table 5: Real-World APIs Test.

# F.2 RUNTIME EXPERIMENTS

Table 6 presents the runtime (in seconds) of NaviAgent variants across different models on ToolBench. Notably, the Dynamic+A method consistently achieves lower runtime across all models, with the most significant improvement observed in Deepseek-V3: compared to the Base method (55.8 seconds), Dynamic+A reduces the runtime by 15 seconds, corresponding to a relative improvement of approximately 26.9%. Among all methodological variants, Dynamic+H demonstrates the optimal overall performance; however, it is constrained by higher runtime induced by heuristic strategies and excessive search scale, which will be the focus of subsequent optimization efforts.

Model	Method	Easy	Medium	Hard	All
	Base	26.6	34.0	44.5	34.0
Qwen2.5-14B	Static+A	21.6	27.1	36.1	27.4
	Dynamic+A	19.8	25.3	34.4	25.6
	Dynamic+H	22.9	30.4	39.6	30.1
	Base	33.4	41.7	53.3	41.7
Qwen2.5-32B	Static+A	24.0	33.0	41.1	32.0
Qwell2.3-32B	Dynamic+A	24.0	31.0	38.6	30.5
	Dynamic+H	28.1	33.8	48.4	35.3
	Base	36.2	44.2	61.2	45.5
Deepseek-R1-32B	Static+A	27.5	33.7	46.8	34.7
	Dynamic+A	24.6	34.6	44.5	33.6
	Dynamic+H	31.0	36.7	49.7	37.8
	Base	43.6	56.6	71.3	55.8
Deepseek-V3	Static+A	34.4	44.5	55.5	43.8
Deepseek- v 3	Dynamic+A	30.3	41.6	53.3	40.6
	Dynamic+H	37.0	47.3	61.6	47.3
	Base	42.3	55.6	75.6	55.9
GPT-40	Static+A	34.9	44.1	59.8	44.7
OF 1-40	Dynamic+A	32.6	43.8	56.4	43.1
	Dynamic+H	36.9	47.1	61.5	47.1
	Base	27.0	38.0	50.1	37.2
Owen2.5-14B(SFT)	Static+A	22.3	27.1	37.8	28.0
Qweii2.3-14b(3F1)	Dynamic+A	19.9	27.9	35.9	27.2
	Dynamic+H	24.5	31.4	40.6	31.3

Table 6: Runtime(in Seconds) of NaviAgent Variants on ToolBench

# F.3 EXPERIMENTS ON API-BANK

Table 7 demonstrates that the experimental outcomes of the API-Bank dataset are consistent with those observed in the ToolBench-based experiments.

Model	Method		Easy		Medium			Hard			All		
	1,1ctilou	TCR	TSR	Steps	TCR	TSR	Steps	TCR	TSR	Steps	TCR	TSR	Steps
	Base	47.8	33.4	5.40	60.5	24.8	6.09	71.6	29.9	6.47	63.1	27.6	6.06
Owen2.5-14B	Static+A	63.4	44.8	4.88	72.4	32.8	5.38	68.4	<u>34.3</u>	5.41	67.9	34.0	5.22
Qwell2.3-14b	Dynamic+A	64.9	<u>49.1</u>	4.93	72.7	<u>36.4</u>	5.32	68.7	36.0	5.36	68.3	<u>36.7</u>	5.18
	Dynamic+H	73.1	56.1	4.71	66.6	40.4	5.27	66.4	33.5	5.63	65.7	37.9	5.26
	Base	61.6	46.6	5.26	78.6	35.0	6.84	68.9	30.7	7.38	70.4	33.4	6.78
Owen2.5-32B	Static+A	88.6	65.7	4.63	80.0	34.3	5.90	84.6	39.9	6.29	81.3	39.5	5.82
Qwell2.3-32B	Dynamic+A	89.5	<u>68.4</u>	4.57	80.1	<u>35.8</u>	5.86	85.3	<u>45.0</u>	6.36	81.8	<u>42.8</u>	5.83
	Dynamic+H	90.8	74.0	4.54	87.0	44.7	5.70	84.2	45.5	5.66	84.1	47.2	5.43
	Base	88.2	66.2	6.41	65.3	28.3	7.79	64.5	25.4	7.83	65.9	30.3	7.49
Deepseek-R1-32B	Static+A	89.2	60.7	5.77	88.1	46.1	6.83	81.2	30.5	6.82	83.0	39.2	6.56
Deepseek-K1-32B	Dynamic+A	89.5	63.2	5.73	90.9	48.3	6.75	81.5	<u>34.1</u>	6.76	84.2	42.0	6.49
	Dynamic+H	99.1	77.6	4.96	89.4	<u>46.9</u>	6.06	79.3	34.4	6.81	83.6	43.2	6.16
	Base	86.3	67.0	5.95	86.3	46.5	6.65	85.4	42.1	7.09	83.9	45.5	6.64
Deepseek-V3	Static+A	97.7	77.4	4.84	98.6	55.5	6.17	98.8	48.1	5.82	96.4	53.1	5.72
Deepseek- v 3	Dynamic+A	99.9	82.5	4.77	98.9	<u>58.5</u>	6.21	99.1	<u>51.2</u>	5.87	96.9	<u>56.3</u>	5.76
	Dynamic+H	98.8	88.9	5.00	98.0	60.0	5.74	98.6	52.3	5.88	96.2	58.0	5.60
	Base	96.8	74.9	5.23	92.4	48.3	6.15	94.5	38.6	6.19	91.8	45.4	5.93
GPT-40	Static+A	99.6	<u>76.8</u>	4.15	98.6	54.5	5.17	98.3	46.9	4.85	96.3	52.0	4.79
OI 1-40	Dynamic+A	99.9	<b>78.5</b>	4.14	98.9	<u>56.4</u>	5.14	98.6	<u>52.2</u>	4.90	96.6	<u>55.5</u>	4.80
	Dynamic+H	98.9	76.1	3.70	98.1	57.9	5.00	97.0	57.8	5.00	95.5	58.5	4.75
	Base	76.0	45.4	5.63	74.9	35.1	6.35	76.3	40.6	6.39	74.0	38.0	6.15
Qwen2.5-14B(SFT)	Static+A	94.1	60.6	4.69	88.7	41.3	5.24	87.9	41.3	5.28	86.9	42.4	5.08
Qwell2.3-14D(3F1)	Dynamic+A	94.7	<u>64.3</u>	4.67	89.8	<u>44.1</u>	5.32	88.2	<u>42.2</u>	5.34	87.5	<u>44.3</u>	5.14
	Dynamic+H	93.2	71.0	4.61	90.2	48.3	5.17	87.6	44.5	5.14	87.3	47.8	4.98

Table 7: Impact of NaviAgent Variants on API-Bank. Metrics are reported as in Table 2.

# G LINK PREDICTION EVALUATION

Dataset	APIs	Nodes	Edges	ACC	F1	AUC
ToolBench	5501	7866	24215	76.4	77.6	0.75
API-Bank	2650	6025	10255	78.4	76.1	0.71

Table 8: Tool Graph Statistics and Link Prediction Evaluation. Nodes and Edges denote the number of nodes and edges in the graph, respectively. ACC and F1 are reported as percentages (%), while AUC is reported as a value between 0 and 1.

# H USAGE OF LLM

To improve clarity and readability, we used a LLM for language polishing. All research ideas, methods, and conclusions were developed solely by the authors.