The Menu Is an Execution Prior: State-Path Tool Menus for Online Agents

Anonymous ACL submission

Abstract

An agent that solves a task by calling external tools never sees the full tool library: a retriever first selects a small menu, and the agent acts only on what that menu shows. Retrievers are trained to rank tools by query relevance, but the agent does not consume a ranked set—it executes a bounded, ordered interface, and relevance is blind to whether that interface is executable. Such a menu can fail in two independent ways that relevance conflates: it can omit a tool the task secretly depends on (a *coverage* failure), or order tools so that the first call the agent can actually run is buried or absent (an *order* failure). We recast tool recommendation as *executable menu construction* and present State-Path Tool Menu, which treats the menu as an execution prior: a *state path*—a compact ordered sequence of tool roles that first covers the hidden chain of prerequisite tools, then exposes an order whose leading prefix the agent can execute. Two coupled graph models realize this: a state-path retriever that selects tools for coverage, and a reranker that separates a tool's importance from its temporal position, so an indispensable late-stage tool stays in the menu without ever being shown as the first action. Holding the task set, execution agent, 32-tool budget, and official ToolEval protocol fixed and changing only the menu constructor, end-to-end success rises from 0.737 to 0.898, a 16.1-point gain over the official retriever. Ablations attribute this to the decomposition itself—coverage alone is insufficient and ordering alone saturates—while the full state-path menu recovers complete reference chains for 70.72% of tasks using a quarter of the candidate budget the official retriever needs to reach the same coverage.

1 Introduction

Large language models have become a general interface for reasoning, instruction following, and decision support, but their parameters alone cannot access private state, query live services, update external records, or verify environment-specific facts (OpenAI, 2024; Mialon et al., 2023; Karpas et al., 2022; Schick et al., 2023). Tools are the mechanism that lets language agents act beyond text (Mialon et al., 2023; Karpas et al., 2022; Yao et al., 2023; Schick et al., 2023; Shen et al., 2023). With tools, an agent can search private databases, call application programming interfaces, update records, retrieve documents, and compose multi-step services in an external environment (Patil et al., 2023; Shen et al., 2023; Qin et al., 2024; Trivedi et al., 2024). This capability is central to practical agents: a model that cannot access the right tool at the right moment cannot complete many real tasks, even if it understands the user request (Yao et al., 2023; Liu et al., 2025; Li et al., 2023). As tool libraries grow, tool access is mediated by a compact menu of candidate tools (Qin et al., 2024; Zheng et al., 2024; Shi et al., 2025; Wang et al., 2025). This menu becomes the bridge between language reasoning and environment state, because it determines which actions the agent can discover, attempt, and combine.

The problem is that most tool recommendation optimizes semantic relevance, while online execution depends on a path of prerequisites (Qu et al., 2024; Zheng et al., 2024; Shi et al., 2025; Wang et al., 2025; Su et al., 2025). Consider an agent asked to send a receipt for order 17 to the customer. A relevance-ranked menu naturally places SENDEMAILRECEIPT first because its name almost repeats the request, yet the call is invalid until the agent has looked up the order, created a receipt identifier, and recovered the customer's email address. The needed bridge tools, such as LOOKUPORDER, CREATERECEIPT, and GETCUSTOMEREMAIL, may be weak lexical matches. Trajectory-aware tool-use benchmarks show that such hidden prerequisites and ordering constraints are common in multi-step tool execution (Jia and

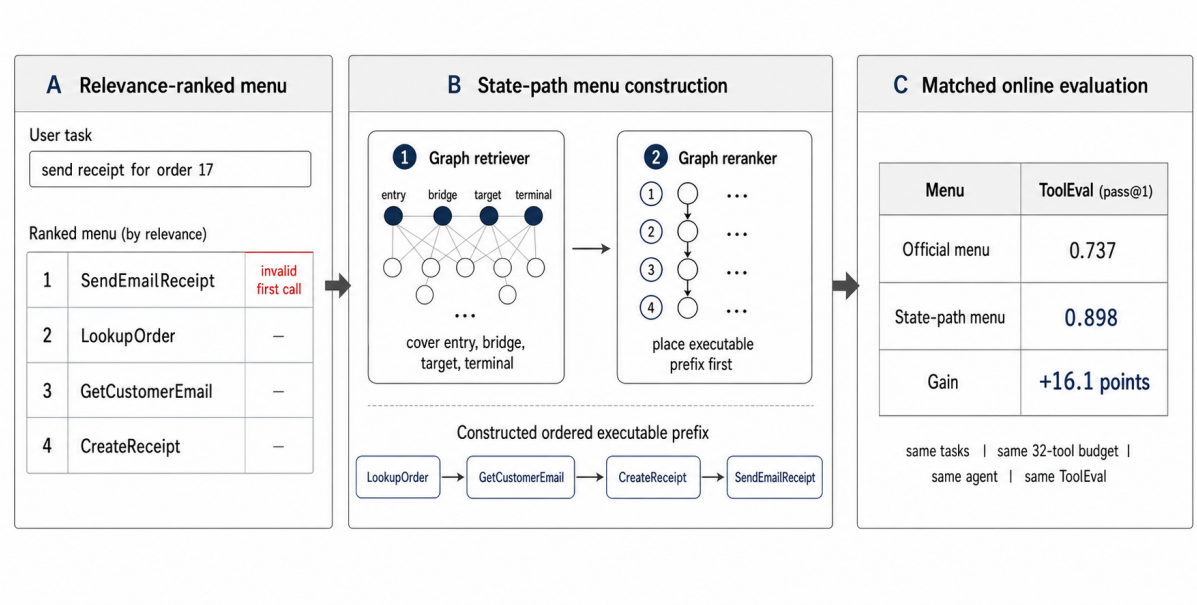

Figure 1: **Ordered tool menus as execution priors.** A relevance-ranked tool menu can contain important tools but still fail online execution: bridge tools may be missing, terminal tools may appear before their prerequisites, and the first visible action may be invalid. State-Path Tool Menu constructs a fixed-budget ordered menu as an executable state path. The retriever covers entry, bridge, target, and terminal roles, and the reranker exposes an executable prefix to the same frozen online agent. With every other factor in the matched ToolEval protocol held fixed, swapping in this menu lifts end-to-end success by 16.1 points.

Li, 2026; Yang et al., 2026; He et al., 2026; Liang et al., 2026). Thus the terminal tool can appear too early, an unnamed bridge tool can be hidden, and a wider candidate list can add distractors without giving the agent an executable prefix (Liu et al., 2024; Pezeshkpour and Hruschka, 2024; He et al., 2026).

We recast the tool menu as an *execution prior*. Because the menu is fixed before the agent acts, it behaves like a prior over executions: it bounds which action sequences are reachable and biases which one the agent attempts first, independent of how relevant any single tool looks in isolation (Figure 1). A good menu should therefore expose a compact state path: a sequence that starts from fields already observable in the request, covers hidden prerequisites, and reaches the answer-bearing action in executable order. State-Path Tool Menu implements this idea with a state-path graph retriever and a state-path graph reranker. The retriever builds a query-specific graph from task text, tool schemas, and trajectory memory to cover entry, bridge, target, and terminal roles under the fixed 32-tool budget. The reranker then builds a candidate-only graph and orders the covered set so the leading prefix is executable for the same frozen online agent.

This paper makes three contributions. First, we identify the ordered tool menu as the execution interface for online tool-use agents and formulate menu construction as building an executable state path. Second, we introduce State-Path Tool Menu, a fixed-budget menu constructor that separates hidden-path coverage and executable prefix ordering under one matched agent interface. Third, we provide a controlled online ToolEval study (Qin et al., 2024) where only the menu constructor changes: State-Path Tool Menu raises success from 0.737 to 0.898, and ablations connect the gain to hidden-prerequisite and dependency-inversion failures.

2 Problem Formulation

Let $\mathcal{T} = \{u_1, \ldots, u_N\}$ be a tool library. Each tool u_i has a document d_i containing its name, description, input fields, output fields, and usage constraints. A task is a user request q. The menu constructor must output an ordered list

$$\pi(q) = (u_{\pi_1}, \ldots, u_{\pi_K}), \quad u_{\pi_i} \in \mathcal{T}, \qquad (1)$$

where $K = 32$ in the main experiments. Here $\pi(q)$ is the menu itself, π_r is the index of the tool placed at position r, and u_{π_r} is the actual tool shown in that position. The equation fixes the object studied

by the paper: a bounded ordered interface rather than an unbounded retrieval list. The online agent sees q and $\pi(q)$, then chooses one tool call per interaction step. The agent may call only a small prefix of the menu; unused tools remain backup actions.

The goal is simple: construct a menu that makes the same fixed online agent solve more tasks. Every main comparison fixes the task set, the execution agent, the 32-tool budget, tool execution, and the official evaluation protocol. Only the menu-construction method changes. Detailed symbols and tensor definitions are listed in Appendix D.1; the main text keeps only the notation needed for the story.

Reference trajectories. Training tasks provide trajectories $\tau = (g_1, \ldots, g_{T_\tau})$. They supply supervision for which tools belong to a path, which tool is the entry action, and which tool should precede another. At test time the trajectory for the target task is hidden. The system can only use reusable transition statistics and learned path patterns from $\mathcal{M}$.

Observable state and state path. Before any tool is executed, the system reads the fields, objects, and constraints already visible in the user request. This is the observable state: for example, the task may already contain a city name and a date, while still missing a venue identifier or a user email. A tool is easy to call when its required inputs are already visible; it is a later tool when earlier calls must first produce its inputs. This state is computed from task text and tool schemas, never from the hidden reference trajectory.

A state path is the ordered sequence of tool roles that carries $s_0(q)$ to a solved state, with each role realized by a tool call. For example, a travel task may need account lookup, flight search, itinerary selection, and booking confirmation. The first two tools often produce identifiers or constraints needed by later tools. The method therefore makes two decisions: cover the tools that form the hidden path and order them into an executable menu. The first decision selects the 32-tool candidate set. The second orders the selected tools into the final menu.

3 Method

3.1 Overview

State-Path Tool Menu learns one object: an ordered tool menu that acts as an execution prior. A menu must answer two questions before the agent acts, and these two questions are the interface decomposition that drives the whole method. *Coverage:* which tools belong to the hidden state path under the 32-tool budget. *Order:* which tool should appear first, second, and so on, so the leading prefix is executable.

The retriever answers coverage by selecting a compact set of tools that jointly fill the entry, bridge, target, and terminal roles of the path. The reranker answers order by arranging those tools so that a tool needed only late in execution can stay in the menu without being exposed as the first action. Both decisions rest on relational evidence rather than isolated relevance: one tool produces a field another tool requires, one observed transition makes a bridge plausible, and one state feature decides whether a first call is even executable. We therefore parameterize both with edge-biased graph Transformers, which carry these dependencies on their edges instead of forcing each tool through an independent relevance score (Figure 2). The method follows the same two decisions end to end:

$$\begin{aligned} G_q^R &= \mathrm{RetrieveGraph}(q, \mathcal{T}, \mathcal{M}), \\ \mathcal{C}_K &= \mathrm{GreedySet}_{R_\theta}(G_q^R, K), \\ \pi_K &= \mathrm{GreedyOrder}_{O_\psi}(\mathcal{C}_K, q, \mathcal{M}). \end{aligned} \tag{2}$$

Here G_q^R is the query-specific retrieval graph, $\mathcal{C}_K$ is the retrieved 32-tool candidate set, and π_K is the final ordered menu; R_θ and O_ψ are the retrieval and ordering models learned during training. The two stages share no parameters: the first decides *what* the agent may see, the second decides *in what order*.

3.2 State-Path Graph Retriever

The retriever answers the coverage question: which 32 tools should be visible before the agent acts. It begins with high-scoring semantic anchors, expands them with trajectory neighbors from the frozen memory $\mathcal{M}$, and builds one task-conditioned graph G_q^R for the current query. This graph has two node types: learnable path-token nodes that summarize global path context, and candidate-tool nodes from the expanded frontier U_q. Tool-tool edges encode forward trajectory transitions, reverse conflicts, same-name links, same-category links, and neutral links. The graph therefore models whether a candidate is an entry action, a bridge that produces missing state, or a later

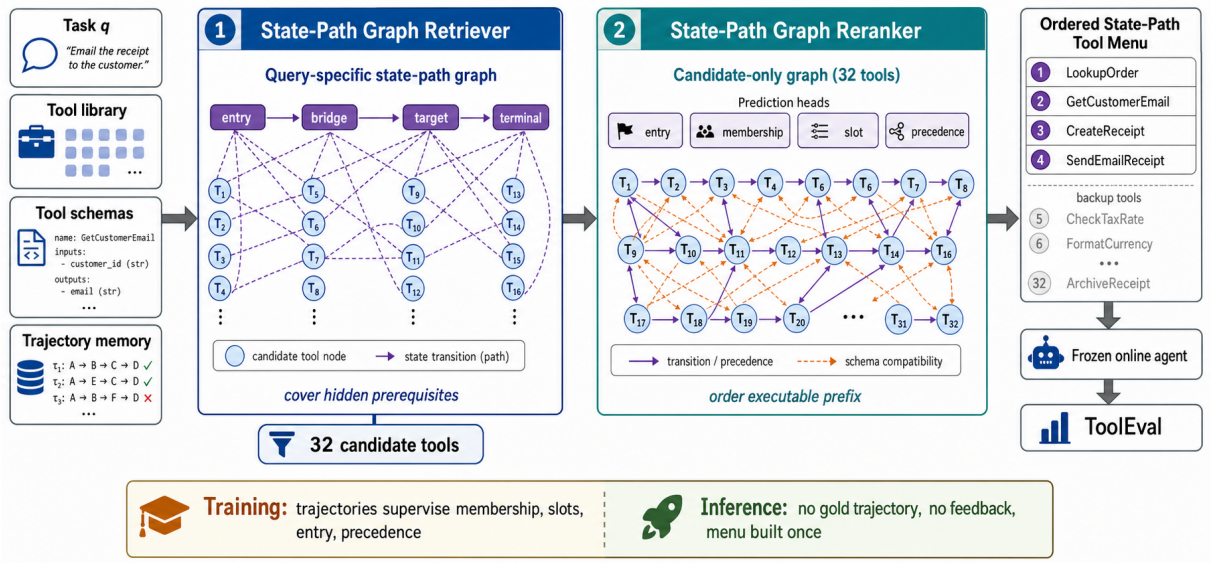

Figure 2: **Method overview.** The State-Path Graph Retriever builds one query-specific graph from the task, observable state, tool schemas, and trajectory memory, then selects a compact 32-tool set under the budget. The State-Path Graph Reranker builds a candidate-only ordering graph over those 32 tools and decodes an executable prefix for the frozen online agent. Results are reported separately in the experiment tables, keeping this figure focused on the mechanism.

answer-bearing tool. The graph is built per task:

$$G_q^R = (V_q^R, E_q^R), \qquad V_q^R = \{p_1, \ldots, p_P\} \cup U_q. \tag{3}$$

The path tokens $p_1, \ldots, p_P$ summarize latent execution roles, while U_q contains semantic anchors plus nearby tools from trajectory memory.

An edge-biased graph Transformer scores every candidate for path membership and latent path position. A coverage decoder then assembles the 32-tool set greedily: at each step it adds the remaining tool u_i with the largest marginal score

$$s_i^R = \hat{y}_i^{mem} + \lambda \, \mathrm{cov}_i - \eta_{\mathrm{red}} \, \mathrm{red}_i, \tag{4}$$

and stops once the budget $K = 32$ is full. Here $\hat{y}_i^{mem}$ is the predicted path-membership score, cov_i is the new path-role or missing-state coverage that u_i adds beyond the tools already selected, and red_i penalizes redundancy with them; λ and η_{red} weight the two terms. Equation 4 is the per-tool marginal form of the set-level objective in Eq. 26, written in the body so the reader sees exactly what the decoder optimizes: prefer tools likely to lie on the hidden path, reward those that fill still-uncovered roles, and avoid duplicates. Keeping coverage explicit is what makes a small budget span distinct roles instead of spending all 32 slots on near-duplicate high-relevance tools that leave the path's later stages uncovered. The node features and the attention form are specified in Appendix D.1.

Training supervises this retriever directly from reference trajectories, which mark three things: which tools lie on the hidden path, which tool can start it, and which tools must precede others. The retriever learns to recover these while rejecting semantically close tools that never appear in any trajectory, so it acquires compact path coverage instead of raw query relevance.

3.3 State-Path Graph Reranker

The reranker answers the order question after retrieval has fixed the candidate set. It builds a second task-conditioned graph G_q^O whose nodes are exactly the retrieved 32 tools; there are no additional state or path-token nodes in this ordering graph. The graph uses transition priors, schema compatibility, observable state compatibility, and upstream retrieval scores to decide which candidate should be visible first and which tools should remain as later backups.

The reranker predicts four signals for every candidate: a membership score (is this tool needed at all), an entry score (could it be the first executable call), a slot score (which position in the path it fits), and a precedence score (which tools it must follow). The first two together are what relevance ranking cannot separate: a tool can be indispensable for success yet belong late in the menu because its inputs do not exist at the start. The menu is then decoded

greedily from left to right, at each position placing the remaining tool that most increases a single path score

$$
\begin{aligned}
J(\pi) = & \sum_{r=1}^{|\pi|} \omega_r^{slot}(m_{\pi_r} + \phi_{\pi_r,r}) \; + \; \mu \, e_{\pi_1} \\
& + \nu \sum_{r<r'} b_{\pi_r \pi_{r'}} - \omega^{inv} \mathrm{Inv}(\pi) \\
& + \sum_{r=1}^{|\pi|} \delta^{\top} g_{\pi_r,r}.
\end{aligned} \tag{5}
$$

The first sum rewards placing a needed tool (m_{π_r}) at a position that fits its slot ($\phi_{\pi_r,r}$); $\mu \, e_{\pi_1}$ rewards a genuinely executable first action; the precedence sum and the inversion penalty Inv keep producers ahead of the tools that consume their outputs; and $g_{\pi_r,r}$ adds observable state features (which inputs are already present, which fields the tool would supply) that decide feasibility at run time. The weights $\omega_r^{slot}, \mu, \nu, \omega^{inv}, \delta$ are development-selected constants fixed before final evaluation. Because each step only re-scores the remaining candidates, decoding stays linear in the menu size; full feature and loss definitions are in Appendix D.1.

3.4 Inference Contract

Both models are trained once and then frozen, alongside a trajectory memory built from training tasks. At test time the system runs the two stages exactly once per task—cover, then order—and hands the agent a single ordered menu. It reads no gold trajectory, takes no evaluator feedback, and never rebuilds the menu mid-execution; the menu is fixed before the agent's first action. This is what makes the comparison clean: every method in our tables faces the same frozen agent and the same one-shot budget, so any gain comes from the menu itself rather than from extra test-time supervision (pseudocode in Appendix Algorithms 10 and 11).

3.5 Why Each Component Is Necessary

State features. State-derived features indicate which preconditions are already satisfied and which fields remain missing. They target first-action errors and invalid tool calls.

Graph attention. Graph attention lets a tool use evidence from related tools: directed transitions, reverse conflicts, same-family tools, and schema-compatible neighbors. It targets bridge-tool recovery and dependency inversion.

Coverage decoding. Coverage decoding protects the limited budget from duplicates. It forces the selected set to span entry, bridge, target, and terminal roles. These names are aliases for coarse path positions: early precondition setters, middle connectors, answer-bearing tools, and late finalization tools.

Separate ordering heads. Entry, membership, slot, and precedence scores answer different questions. A necessary terminal tool should receive a high membership score and a later slot score. The tool placed first is the predicted first executable action, not the globally most important tool.

4 Experiments

4.1 Evaluation Protocol

All main ToolBench experiments use the official ToolEval pass-rate protocol. For every method, we fix the task set, execution agent, 32-tool menu budget, tool-call budget, execution wrapper, decoding settings, invalid-call handling, and evaluator. Each method outputs one ordered menu before execution; the menu is held fixed while the same agent acts. The primary metric is end-to-end task success. Diagnostics such as complete-chain coverage, executable-prefix length, dependency inversion, invalid-call rate, and final-answer completion explain the mechanism behind the success score. Appendix B reports the fixed prompt, decoding settings, split boundary, and adapter contract.

4.2 Baselines

The core comparison uses one interface: every system produces a bounded ordered menu for the same online agent. We compare against official ToolBench retrieval, wider official retrieval scopes, recent paper-level tool systems, one-stage variants of our method, and the full cascade. The paper-level set includes ToolGen, Toolscaler, Tool-REX/ToolRank, SkillRouter, ToolRet, Multi-Field Tool Retrieval, and completeness-oriented retrieval (Wang et al., 2025; Su et al., 2025; Lu et al., 2026; Zheng et al., 2026; Shi et al., 2025; Tang et al., 2026; Qu et al., 2024). Appendix controls add schema sorting, transition-only ordering, sequential planning, pairwise sorting, generation-style planning, and engineered dependency ranking under the same fixed-menu interface.

Benchmark	Metric	Method	Score↑	Gap↑
ToolBench ToolEval	official success	ToolBench-IR	0.737	0.000
		COLT-Contriever	0.582	-0.155
		ToolRet-E5	0.632	-0.105
		Tool-REX Embed	0.661	-0.076
		ToolGen	0.730	-0.007
		Tool-REX Rank (4B)	0.724	-0.013
		SkillRouter	0.707	-0.030
		State-Path Tool Menu	**0.898**	**+0.161**
AppWorld	next-tool R@1	ToolBench-IR	0.103	-0.116
		COLT-Contriever	0.190	-0.029
		ToolRet-E5	0.126	-0.093
		Tool-REX Embed	0.219	0.000
		ToolRerank	0.184	-0.035
		ToolDreamer	0.197	-0.022
		State-Path Tool Menu	**0.465**	**+0.246**
TRAJECT-Bench	complete-chain@32	COLT-Contriever	0.230	-0.120
		ToolRet-E5	0.188	-0.162
		Tool-REX Embed	0.350	0.000
		State-Path Tool Menu	**0.412**	**+0.062**
UniToolCall	ordered-prefix	COLT-Contriever	0.215	-0.082
		ToolRet-E5	0.134	-0.163
		Tool-REX Embed	0.262	-0.035
		ToolGen	0.215	-0.082
		SkillRouter	0.169	-0.128
		ToolACE	0.297	0.000
		State-Path Tool Menu	**0.635**	**+0.338**

Table 1: **State-Path Tool Menu leads every completed benchmark block.** Metrics are defined as follows: official success is online task success under the frozen execution agent and official ToolEval evaluator; next-tool R@1 checks whether the first menu item matches the next executable application call; complete-chain@32 checks whether all tools in a reference trajectory appear inside the 32-tool menu; ordered-prefix checks whether the leading predicted tool-call sequence matches the gold multi-hop prefix. Gap is computed against the reference row in each benchmark block: ToolBench-IR for ToolBench, Tool-REX Embed for AppWorld and TRAJECT-Bench, and ToolACE for UniToolCall. The ToolBench block is the primary controlled online-agent comparison; the remaining blocks use each benchmark's native primary metric as robustness diagnostics.

4.3 Main Result and Evidence Map

Table 1 reports the primary completed benchmark blocks. The ToolBench block carries the main claim because it is the controlled online-agent comparison: same agent, same 32-tool budget, same execution environment, same official evaluator, and only the menu constructor changes. The other blocks use native benchmark metrics and serve as robustness diagnostics. Published values under different protocols and fixed-candidate plug-in comparisons are moved to Appendix F.2 and Appendix F.4. On the 304 ToolBench tasks, the official menu solves 224 and the state-path menu solves 273 (0.737 versus 0.898). The two menus disagree on 51 tasks: 50 are solved only by the state-path menu and 1 only by the official menu, so a paired McNemar test on these discordant pairs rejects equality at $p < 10^{-13}$, and the paired bootstrap interval on the net gain of 49 tasks stays above zero (Table 2). The remaining experiments form a controlled chain that isolates each factor (Figure 3): menu order changes online success, coverage alone is insufficient, state-path retrieval improves hidden-chain coverage, and failure analysis shows fewer hidden-prerequisite, invalid-first-call, and dependency-inversion errors.

4.4 Why More Tools Alone Is Not Enough

The first diagnostic asks whether a wider relevance list already solves hidden-chain coverage. Our 32-tool retriever reaches 70.72% complete-chain coverage, which the official retriever matches (69.08%) only after widening its menu fourfold to 128 candidates. This coverage result explains the upstream advantage of compact state paths; the end-to-end claim remains the matched online ToolEval result. The full budget curve appears in Appendix F.1.

Method or comparison	Solved	Pass rate	95% CI	Evidence
ToolBench official	224 / 304	0.737	[0.685, 0.783]	matched baseline
State-Path Tool Menu	273 / 304	**0.898**	[0.859, 0.927]	full system
State-Path Tool Menu − ToolBench official	+49 / 304	**+0.161**	[0.118, 0.204]	paired bootstrap

Table 2: **The main ToolBench gain is large relative to sampling uncertainty.** Both rows use the same task identifiers, execution agent, budget, environment, and official ToolEval protocol. The paired table contains 50 tasks solved only by State-Path Tool Menu and 1 task solved only by the official menu, so the paired bootstrap interval remains well above zero and the paired McNemar test rejects equality at $p < 10^{-13}$.

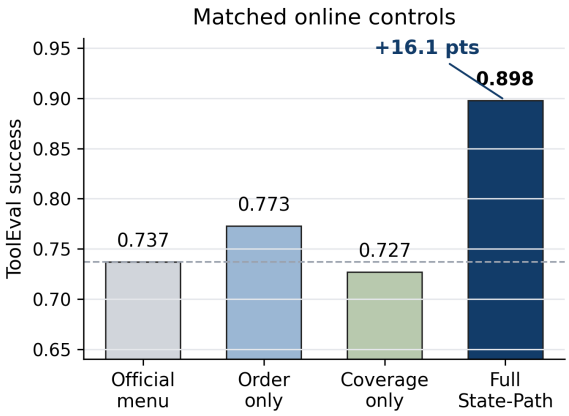

Figure 3: **Coverage and ordering must work together under the matched online protocol.** Ordering alone improves the official candidate set, and coverage alone changes the retrieved set, but neither reaches the full system. The full state-path menu combines hidden-path coverage with executable-prefix ordering and reaches 0.898 ToolEval success.

4.5 From Controlled Ablations to Failure Modes

Three single-factor controls isolate where the gain comes from, and we report them here in the main text. With the official retriever, our reranker alone raises success from 0.737 to 0.773, so changing order alone already changes the online outcome. Our retriever with the original order reaches 0.727, so improved coverage alone does not explain 0.898 either. Component and non-graph ablations then localize the removed errors: bridge-tool omission, wrong first action, dependency inversion, and distractor-heavy menus, with the full per-control tables reported in the appendix.

5 Related Work

Tool-augmented language models and benchmarks. Toolformer, MRKL-style systems, ReAct, Gorilla, HuggingGPT, ToolACE, and API-Bank establish that language models become more useful when they can call external tools, APIs, and domain services rather than rely only on parametric memory (Schick et al., 2023; Karpas et al., 2022; Yao et al., 2023; Patil et al., 2023; Shen et al., 2023; Liu et al., 2025; Li et al., 2023). ToolBench, AnyTool, AppWorld, τ-bench, Live API-Bench, Uni-ToolCall, and TRAJECT-Bench move evaluation toward large tool libraries, stateful execution, unified tool-call schemas, and multi-step trajectories (Qin et al., 2024; Du et al., 2024; Trivedi et al., 2024; Yao et al., 2024; Elder et al., 2025; Liang et al., 2026; He et al., 2026). These benchmarks make the menu-construction problem visible: before an agent can reason over tools, a small ordered interface must decide which tools the agent is allowed to see.

Tool retrieval, generation, and routing. Prior work improves tool discovery through hierarchy-aware reranking, completeness-oriented retrieval, field-aware matching, tool-document expansion, generative tool calling, structure-aware tokenization, reasoning-augmented retrieval, and skill routing (Zheng et al., 2024; Qu et al., 2024; Shi et al., 2025; Tang et al., 2026; Lu et al., 2026; Wang et al., 2025; Su et al., 2025; Sengupta et al., 2026; Zheng et al., 2026). These methods usually optimize a relevance or retrieval objective. Our focus is the online interface that follows retrieval: the agent receives a bounded ordered menu, and that menu must cover hidden prerequisites and put executable calls early. This connects tool retrieval to listwise reranking and structured prediction, but the objective is different from ranking documents by relevance because a late-stage tool can be essential while still being a poor first action.

Trajectory structure and order sensitivity. AutoTool and ToolTree use historical tool trajectories or explicit lookahead to improve planning over tool calls (Jia and Li, 2026; Yang et al., 2026). Work on long-context and option-order sensitivity shows that language models can change behavior when relevant items are buried or presented in a harmful order (Liu et al., 2024; Pezeshkpour and Hruschka, 2024). State-Path Tool Menu combines these ob-

servations in a fixed-budget setting: it learns which tools form a hidden state path, then exposes that path as an executable prefix. The controlled ToolEval experiment asks whether changing only this menu interface changes online task success.

6 Scope

The main empirical claim concerns the controlled ToolBench setting. Every main row fixes the released task set, execution prompt, decoding settings, 32-tool budget, tool-call budget, execution wrapper, and official ToolEval evaluator; the row-specific object is the ordered menu. This isolates the interface studied by the paper and makes the 0.737-to-0.898 gain interpretable as a menu-construction effect.

The cross-benchmark rows have a supporting role. They use each benchmark's native metric because AppWorld, TRAJECT-Bench, and UniTool-Call ask different upstream questions: next executable call, complete-chain coverage, and ordered-prefix recovery. Rows adapted from published systems are reported under the shared fixed-menu interface, while original native paper values are kept in the appendix. This keeps leaderboards, adapter diagnostics, and internal controls in separate evidential roles.

7 Conclusion

We presented State-Path Tool Menu, a learned menu constructor for online tool agents. The central claim is that a tool menu is an execution prior: it shapes which action paths are reachable and which prefix the agent is likely to attempt first. This view changes the objective from ranking individually relevant tools to constructing a compact state path that covers hidden prerequisites and exposes executable calls early.

Under a matched ToolEval protocol, changing only the menu constructor raises end-to-end success from 0.737 to 0.898. The supporting diagnostics explain why: wider relevance lists recover coverage inefficiently, coverage without ordering is insufficient, and ordering the same candidate set already changes online success. The full state-path menu combines the two factors and reduces bridge omissions, invalid first calls, dependency inversions, and distractor-heavy prefixes.

The broader implication is that tool-use systems should treat the menu as a first-class execution interface. As tool libraries grow, the agent's success will depend less on whether a single terminal tool is lexically relevant and more on whether the visible menu forms a short, executable route through state.

8 Limitations

State-Path Tool Menu learns its state paths from tool documentation and training trajectories. When documentation omits input–output semantics or the trajectories do not cover a rare workflow, the learned graph can understate a valid dependency; richer schema induction and broader trajectory mining are natural extensions. We also construct the menu once, before execution begins, which matches the fixed-budget online protocol studied here but does not yet exploit mid-trajectory feedback; reconstructing the state path as observations arrive is a promising direction for interactive settings. Finally, our primary online evaluation centers on the ToolEval protocol. The AppWorld, TRAJECT-Bench, and UniToolCall rows use their native upstream metrics and should be read as robustness evidence for the state-path formulation, not as additional full online ToolEval replications. Extending full online execution to more environments would further strengthen external validity. Complete-chain coverage is also a single-reference-chain diagnostic. It explains whether a compact menu contains one known executable path, while the official online success metric remains the judge of whether the agent can solve the task. The graph Transformer ablations compare against sequence planners, pairwise sorters, generation-style planners, and engineered dependency ranking, but the system still contains strong state, schema, and trajectory features. The evidence therefore supports the combined state-path graph interface; it does not claim that the Transformer block alone accounts for the entire gain.

References

Yu Du, Fangyun Wei, and Hongyang Zhang. 2024. Anytool: Self-reflective, hierarchical agents for large-scale API calls. In *International Conference on Machine Learning*.

Benjamin Elder, Anupama Murthi, Jungkoo Kang, Ankita Rajaram Naik, Kiran Kate, Kinjal Basu, and Danish Contractor. 2025. Live API-bench: 2500+ live APIs for testing multi-step tool calling. *arXiv preprint arXiv:2506.11266*.

Pengfei He, Zhenwei Dai, Bing He, Hui Liu, Xianfeng Tang, Hanqing Lu, Juanhui Li, Jiayuan Ding, Subhabrata Mukherjee, Suhang Wang, Yue Xing, Jiliang Tang, and Benoit Dumoulin. 2026. TRAJECT-bench: A trajectory-aware benchmark for evaluating agentic tool use. In *International Conference on Learning Representations*.

Jingyi Jia and Qinbin Li. 2026. Autotool: Efficient tool selection for large language model agents. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 31265–31273.

Erez Karpas, Omri Abend, Yonatan Belinkov, Barak Lenz, Opher Lieber, Nir Ratner, Yoav Shoham, Haggai Bata, Yoav Levine, Kevin Leyton-Brown, Dor Muhlgay, Neer Rozen, Elad Schwartz, Gal Shachaf, Shai Shalev-Shwartz, and Amnon Shashua. 2022. MRKL systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning. *arXiv preprint arXiv:2205.00445*.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. API-bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3102–3116. Association for Computational Linguistics.

Yijuan Liang, Xinghao Chen, Yifan Ge, Ziyi Wu, Hao Wu, Changyu Zeng, Wei Xing, and Xiaoyu Shen. 2026. Unitoolcall: Unifying tool-use representation, data, and evaluation for LLM agents. *arXiv preprint arXiv:2604.11557*.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173.

Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, Zezhong Wang, Yuxian Wang, Wu Ning, Yutai Hou, Bin Wang, Chuhan Wu, Xinzhi Wang, Yong Liu, Yasheng Wang, and 8 others. 2025. ToolACE: Winning the points of LLM function calling. In *International Conference on Learning Representations*.

Xuan Lu, Haohang Huang, Rui Meng, Yaohui Jin, Wenjun Zeng, and Xiaoyu Shen. 2026. Tools are under-documented: Simple document expansion boosts tool retrieval. In *International Conference on Learning Representations*.

Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, Edouard Grave, Yann LeCun, and Thomas Scialom. 2023. Augmented language models: A survey. *Transactions on Machine Learning Research*.

OpenAI. 2024. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large language model connected with massive APIs. *arXiv preprint arXiv:2305.15334*.

Pouya Pezeshkpour and Estevam Hruschka. 2024. Large language models sensitivity to the order of options in multiple-choice questions. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 2006–2017. Association for Computational Linguistics.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *International Conference on Learning Representations*.

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

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*.

Saptarshi Sengupta, Zhengyu Zhou, Jun Araki, Xingbo Wang, Bingqing Wang, Suhang Wang, and Zhe Feng. 2026. ToolDreamer: Instilling LLM reasoning into tool retrievers. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5465–5482, Rabat, Morocco. Association for Computational Linguistics.

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

Zhengliang Shi, Yuhan Wang, Lingyong Yan, Pengjie Ren, Shuaiqiang Wang, Dawei Yin, and Zhaochun Ren. 2025. Retrieval models aren't tool-savvy: Benchmarking tool retrieval for large language models. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 24497–24524. Association for Computational Linguistics.

Yunyue Su, Jinshuai Zhang, Bowen Fang, Wen Ye, Jinghao Zhang, Bowen Song, Weiqiang Wang, Qiang Liu, and Liang Wang. 2025. Toolscaler: Scalable generative tool calling via structure-aware semantic tokenization. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 556–578. Association for Computational Linguistics.

Yichen Tang, Weihang Su, Yiqun Liu, and Qingyao Ai. 2026. Multi-field tool retrieval. *arXiv preprint arXiv:2602.05366*.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. 2024. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 16022–16076. Association for Computational Linguistics.

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

Shuo Yang, Caren Han, Yihao Ding, Shuhe Wang, and Eduard Hovy. 2026. Tooltree: Efficient LLM agent tool planning via dual-feedback monte carlo tree search and bidirectional pruning. In *International Conference on Learning Representations*.

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

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

YanZhao Zheng, ZhenTao Zhang, Chao Ma, YuanQiang Yu, JiHuai Zhu, Yong Wu, Tianze Xu, Baohua Dong, Hangcheng Zhu, Ruohui Huang, and Gang Yu. 2026. SkillRouter: Skill routing for LLM agents at scale. *arXiv preprint arXiv:2603.22455*.

Yuanhang Zheng, Peng Li, Wei Liu, Yang Liu, Jian Luan, and Bin Wang. 2024. ToolRerank: Adaptive and hierarchy-aware reranking for tool retrieval. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation*, pages 16263–16273. ELRA and ICCL.

Appendix

A Evaluation Set Size

The released ToolBench test files used in this study contain 304 evaluable tasks after applying the official split loader and the same filtering required by the executable tool-replay environment. We use the entire released evaluable set rather than sampling from it; all main tables therefore report full-set ToolEval success.

B Matched Evaluation Protocol

This appendix records the fixed online-execution contract behind the main comparison. The method equations specify the menu constructor; the tables below specify the shared task set, prompt, decoding settings, budget, split boundary, and adapter policy. Together they make the controlled comparison checkable without changing the main narrative: every row differs only in the ordered menu shown to the same frozen agent.

B.1 Fixed Online Execution Contract

The main ToolBench comparison is a controlled menu-construction experiment. Every row receives the same task, the same tool library, the same execution agent, the same tool execution wrapper, the same menu budget, the same maximum number of tool calls, and the same official ToolEval success evaluator. The row-specific object is only the ordered menu $\pi(q)$ supplied to the agent before execution. Table 3 lists the matched quantities that remain fixed across systems.

B.2 Task Identifiers, Splits, and Execution Parameters

The main result uses the released ToolBench evaluable set after the official split loader and executable-tool filtering. The 304 task identifiers are fixed before any menu method is scored. The split labels are used only for reporting, confidence intervals, and diagnostic breakdowns; they are never part of the agent prompt, reranker feature vector, or evaluator input. Table 4 gives the concrete distribution and runtime constants.

B.3 Agent Prompt, Tool Failures, and Evaluator Input

The agent sees the user task followed by the ordered menu. Each menu entry contains the tool name, natural-language description, input schema, and argument format. The fixed agent instruction is:

Quantity	Matched value in the main ToolBench experiment
Task set	Full released evaluable ToolBench set loaded by the official split loader.
Menu budget	32 tools, ordered once before the agent begins executing.
Tool-call budget	Fixed maximum online step budget for every method; invalid calls consume the same budget as valid calls.
Execution agent	One frozen tool-using agent prompt and one deterministic decoding configuration for all rows.
Decoding	Single deterministic sample; temperature 0; nucleus probability 1; no method-specific decoding change.
Tool environment	Same executable tool wrapper, same observation format, and same timeout and exception handling.
Evaluation	Official ToolEval binary success protocol; method identity and menu-construction details are never shown to the evaluator.
Randomness	Fixed seed for menu construction and deterministic single execution per menu in the main table.

Table 3: **Matched online-execution contract for the main result.** The experiment changes the menu-construction method and holds every downstream execution and evaluation condition fixed.

Item	Value used in all matched ToolBench rows
Released task groups	G1: 107 tasks; G2: 134 tasks; G3: 63 tasks; total: 304 tasks.
Main execution model	Qwen2.5-72B-Instruct-AWQ, frozen local deployment, identical prompt for all rows.
Execution decoding	Greedy single sample; temperature 0; nucleus probability 1; maximum 8 tool-call steps.
Menu visibility	Exactly one 32-tool ordered menu is built before execution; the menu is never rebuilt after a tool call.
Invalid calls	Missing tool name, invalid arguments, tool exception, timeout, or empty response all consume one step and return the same standardized observation format.
Random seed	20260525 for menu construction, trace indexing, and paired bootstrap resampling.
Final evaluator input	Task text, full action-observation trace, final answer, and official ToolEval wrapper fields; method name and menu-construction features are hidden.

Table 4: **Task distribution and execution constants.** These values are fixed for the official retriever row, all external matched baselines, and State-Path Tool Menu. The only variable in the main comparison is the ordered menu shown to the agent.

> You are a tool-using assistant. Use only tools in the provided menu. At each step, either call one

tool with a valid JSON argument or produce the final answer. Use earlier observations when deciding the next action. Stop when the user request is satisfied or the step budget is exhausted.

The instruction is deliberately menu-neutral: it never reveals which method built the menu. If the agent calls a missing tool, produces invalid arguments, hits a tool exception, or receives an empty tool response, the environment records a standardized observation and the agent continues until the same step budget is reached. The official ToolEval evaluator receives only the original task, the complete execution trace, tool observations, and the final answer, then returns a binary task-success label. The evaluator prompt is fixed as part of the official protocol wrapper, and the paper does not vary evaluator identity across main rows.

B.4 Baseline Adapters and Retuning Policy

All external baselines are adapted through the same interface: given task q and the shared tool library, the adapter outputs an ordered 32-tool menu before execution. Retrieval baselines keep their native ranking scores and are truncated to the same menu budget. Organizer or ranking baselines receive the same candidate budget and may reorder candidates, but cannot call tools, inspect execution observations, or observe evaluator feedback. When an external paper releases hyperparameters, we use those settings. When a paper does not define a ToolBench online-menu adapter, we instantiate the closest fixed-candidate adapter once on development tasks and freeze it before final evaluation. No baseline is retuned on final ToolBench feedback. Table 5 and Table 6 report the adapter family, candidate source, ordering rule, and retuning rule for every baseline. The main table distinguishes systems that build their own 32-tool menu from fixed-candidate organizers; Appendix F.4 is the controlled organizer-only table where the upstream candidate frontier is explicitly held fixed.

Baseline family	Adapter used in the matched protocol	Retuning rule
ToolBench official retrieval	Original retrieval order, truncated to 32 tools	no retuning
Dense retrievers	Native similarity score, truncated to 32 tools	authors' released/default settings
ToolGen-style organizer	Fixed-candidate ordering adapter under the same 32-tool menu	development-only selection
Tool-Rank-style organizer	Ranking score over the same fixed candidates	authors' released/default settings
Skill-routing organizer	Router score mapped to the shared tool library and truncated	development-only calibration
Simple dependency controls	Schema or trajectory rule produces an order over the same candidates	development-only constants

Table 5: **Adapter contract for matched baselines.** All systems are forced into the same menu-output interface; no row receives online execution feedback during menu construction.

Adapter row	Candidate source	Ordering rule
Official retriever order	released ToolBench retriever	preserve released order
COLT-Contriever	method-native dense score	descending score
ToolRet-E5	method-native dense score	descending score
Tool-REX Embed	method-native dense score	descending score
ToolGen organizer	shared fixed frontier	frozen organizer score
Tool-Rank 4B	shared fixed frontier	frozen ranker score
Tool-Rank 8B	shared fixed frontier	frozen ranker score
SkillRouter	shared fixed frontier	frozen router score

Table 6: **External adapter inventory for the matched ToolBench table.** Every listed adapter uses the same task identifiers, 32-tool budget, execution model, and official success protocol. Rows whose original papers do not define an online ToolBench menu are frozen once on development tasks and then applied unchanged to the final 304 tasks.

B.5 Development/Test Separation for Menu Construction

All menu-construction choices are fixed before the final ToolBench run. Trajectory memory $\mathcal{M}$ is built from training trajectories and tool schemas. The development set is used to select loss weights, decoder constants, hard-negative policy, and baseline-adapter constants; after that point the retriever, reranker, decoding weights, random seed, and adapter identifiers are frozen. The final released evaluable ToolBench set is used only once for reporting. Its evaluator feedback is not used to choose thresholds, loss weights, decoder constants, or baseline adapters. Reference chains from the final evaluable tasks are used only for diagnostic coverage metrics and never for candidate construction or reranker decoding.

Feature family	Definition and use
Semantic anchor sim_i	normalized task–tool document similarity; builds the initial retrieval frontier
Reciprocal rank rrank_i	rank induced by the frozen semantic score; used as a weak relevance prior
State compatibility $\chi_i(s_0)$	overlap between visible request fields and tool inputs; marks executable early tools
Missing-field score miss_i	required inputs absent from the current state; discourages invalid first calls
Schema edge κ_{ij}^{schema}	output–input overlap between tools; supports producer-before-consumer ordering
Transition prior $\rho(i,j)$	smoothed transition evidence from training trajectories; supports bridge recovery
Family feature fam_i	tool namespace/category parsed from documentation; helps duplicate control

Table 7: **Feature construction is pre-execution and task-local.** Every feature can be computed before the agent takes its first action. The final task's reference trajectory is excluded from menu construction; it is used only after evaluation for diagnostic coverage reporting.

B.6 Supplementary Checkability Note

The supplement is intentionally lightweight. It contains the frozen configuration, the fixed prompts, the per-task paired outcomes, redacted traces, and checksum manifests needed to recompute the reported counts, confidence intervals, and paired tests. Raw task text and raw tool observations are replaced by stable digests in the anonymous package, while a de-anonymized rerun uses the same task identifiers, tool environment, and trained weights. These materials are a consistency layer for the controlled experiment; the paper's claims follow from the protocol and results reported here.

B.7 Feature Construction and Neural Hyperparameters

The neural implementation consumes fixed numeric features derived before execution. No feature uses the final reference chain, evaluator decision, post-execution observation, or failure tag for the target task. Table 7 spells out the feature families used by the formulas in Appendix D.1; Table 8 gives the tensor dimensions and training constants released with the reference implementation. The table below records the frozen run contract used by the final model.

Model size and compute budget. The state-path graph retriever and reranker are lightweight edge-biased graph Transformers. With hidden width 128, two layers, and four attention heads (Table 8), each stage holds on the order of half a million trainable parameters, so the two learned stages together amount to roughly one million parameters. Both stages are cheap to train: each fits in well under one GPU-hour on a single modern data-center GPU, and the dominant compute throughout this paper is frozen language-model inference during online evaluation, since we fine-tune no language model. The execution agents are frozen open-weight instruction-tuned language models from the Qwen2.5 and Llama-3.1 families (7B to 72B); the main results use Qwen2.5-72B-Instruct-AWQ. All reported menu-construction runs use the single fixed seed recorded in Table 8 on one data-center GPU.

Artifacts, licenses, and data statement. The evaluation benchmarks (ToolBench/ToolEval, AppWorld, TRAJECT-Bench, and UniToolCall), the frozen text encoder (BGE-large-en-v1.5), and the open-weight execution models are publicly released research artifacts. We use each one under its original open research license and only for research evaluation, consistent with its stated intended use; the menu constructors we release follow the same research-only terms. The tasks are synthetic API tool-use instructions and tool schemas: they contain no personally identifying information and no offensive content, and we introduce none.

C Notation

D Full Method Details

D.1 Retriever and Reranker Equations

The appendix keeps the full tensor-level formulation that the main text intentionally omits. Before execution, the observable state is

$$s_0(q) = \big(F_0(q), O_0(q), C_0(q)\big), \tag{6}$$

where the three sets contain visible fields, objects, and constraints in the request. For a tool u_i, pre-execution compatibility is

$$\chi_i(s_0) = \frac{|I_i \cap F_0(q)|}{|I_i| + \epsilon}, \qquad \mathrm{miss}_i = 1 - \chi_i(s_0). \tag{7}$$

For any training trajectory, the pairwise precedence label is

$$y_{ij}^{prec} = \mathbb{I}[\mathrm{pos}_\tau(u_i) < \mathrm{pos}_\tau(u_j)]. \tag{8}$$

Parameter	Released value
Text encoder f	frozen BGE-large-en-v1.5 task/tool-document encoder; cached embeddings and encoder identifier are mounted by the replay manifest
Retriever frontier construction	top semantic anchors plus trajectory neighbors plus high-recall fallback; internal prepool 128 before the final 32-tool menu
Negative sampling	hard semantic negatives outside the reference path plus in-frontier non-path negatives; cutoff negatives sampled around the 32-tool boundary
Tool feature dimension / query feature dimension / edge feature dimension	64 / 64 / 16
Hidden dimension / attention heads / graph-Transformer layers	128 / 4 / 2
Retriever path tokens / supervised slots	4 / 8
Relation types	neutral, forward transition, reverse transition, same tool, same category
Training epochs / batch size	retriever 2 epochs with batch 8; reranker 4 epochs with batch 8
Dropout / optimizer	0.08 / AdamW
Learning rate / weight decay	reranker 10^{-4} / 0.01; retriever builder 8×10^{-4} / 10^{-3}
Gradient clipping / random seed	1.0 / 20260525 for the final run; retriever builder seed 17 recorded before final evaluation
Loss weights	λ_{entry}=0.55, λ_{slot}=0.55, λ_{prec}=0.75, λ_{member}=0.30, λ_{path}=1.15
Decoder budgets	32 menu tools and 8 online tool-call steps
Decoder constants	development-selected weights for entry, slot, precedence, inversion, coverage, and redundancy

Table 8: **Neural tensor contract and optimization constants.** The neural implementation instantiates the same retriever and reranker heads described in the method equations.

Symbol	Meaning
q	user task
u_i, d_i	tool and its document
$N, \mathcal{T}$	number of tools and full tool library
K	final menu budget, fixed to 32
$\pi(q)$	ordered tool menu shown to the agent
A, Succ	fixed execution agent and official success judgment
$s_0(q)$	observable pre-execution state of task q
F_0, O_0, C_0	available fields, objects, and constraints in $s_0(q)$
I_i, O_i	required input fields and produced output fields of tool u_i
$\tau = (g_1, \ldots, g_{T_\tau})$	training tool trajectory of length T_τ
$\mathcal{M}$	transition, schema, and path memory
U_q	internal candidate set before final pruning
P	number of learnable path tokens, fixed to 4
S	maximum supervised path slots for ordering
G_q^R	retrieval menu graph for task q
G_q^O	ordering menu graph for the 32 candidates
r_{ij}^R	discrete relation type from node i to node j in G_q^R
a_{ij}^O	continuous pair feature from tool i to tool j in G_q^O
$\rho(i,j), c(i \prec j)$	transition prior and its trajectory count
α	optimizer learning rate
$m_i, e_i, \phi_{i,r}, b_{ij}$	membership, entry, slot, and precedence scores

Table 9: **Core notation.** The method constructs a task-conditioned menu graph for each input task rather than a single dataset-level graph passed to the model.

These definitions are placed here so the main text can focus on the state-path story.

D.2 State-Path Graph Retriever

The retriever starts from a broad semantic frontier and then expands it with path neighbors. Let $f(\cdot)$ be the text encoder used for task and tool documents:

$$\mathrm{sim}_i = f(q)^\top f(d_i). \tag{9}$$

Here $f(q)$ and $f(d_i)$ are normalized embeddings of the task and tool document, and sim_i is their semantic anchor score. This score captures obvious query-tool relevance; later graph terms correct it when a necessary bridge tool is not named by the

user. We take high-scoring semantic anchors A_q and expand them with trajectory neighbors from $\mathcal{M}$:

$$U_q = A_q \cup \bigcup_{u_i \in A_q} \mathrm{Nbr}_{\mathcal{M}}(u_i) \cup H_q, \tag{10}$$

where H_q contains additional high-recall semantic candidates. In Eq. 10, A_q is the top semantic anchor set, $\mathrm{Nbr}_{\mathcal{M}}(u_i)$ is the set of tools observed near u_i in training trajectories, and H_q is a recall-oriented fallback set. The intuition is to start from what the task text names, then add tools that historical paths say are needed nearby. This internal set is larger than the final 32-tool budget, giving the model space to recover bridge tools that are weakly matched to the query text.

Graph construction. For each task, the retriever builds one small task-conditioned menu graph:

$$G_q^R = (V_q^R, E_q^R), \quad V_q^R = \{p_1, \ldots, p_P\} \cup U_q, \tag{11}$$

where $p_1, \ldots, p_P$ are learnable path tokens and $P = 4$ in the frozen system. The node set V_q^R contains all graph nodes for task q, and E_q^R is represented by the relation matrix defined below. These tokens do not correspond to real tools. They summarize global path context while candidate tool nodes exchange information through graph attention. Thus G_q^R contains two node types: path-token nodes and candidate-tool nodes. The tool nodes are query-specific because U_q is built from semantic anchors, trajectory neighbors, and high-recall fallback candidates for the current task.

The edge set is represented by a relation matrix over all ordered node pairs. For edges touching a path-token node, the relation type is set to the neutral type 0; the path token still attends through the normal content attention. For a pair of tool nodes (u_i, u_j), trajectory memory provides a directed transition prior:

$$\rho(i,j) = \log \frac{c(i \prec j) + 1}{c(j \prec i) + 1}, \tag{12}$$

where $c(i \prec j)$ counts trajectories in which u_i precedes u_j. The $+1$ smoothing avoids zero-count instability. Positive $\rho(i,j)$ means the training trajectories more often use u_i before u_j; negative $\rho(i,j)$ warns that the reverse order is more plausible. This is the bridge from historical execution paths to graph reasoning. The graph stores the relation type

$$r_{ij}^R \in \{0, 1, 2, 3, 4\}, \tag{13}$$

where the five values mean neutral, forward trajectory transition, reverse trajectory transition, same tool name, and same tool category. The symbol r_{ij}^R is discrete because the retriever needs a stable edge type for attention, while the continuous strength $\rho(i,j)$ remains available through features. Forward and reverse trajectory evidence take priority over name and category evidence because they directly encode execution order. The graph has no manually fixed scalar edge weight. Instead, each relation type has a learned attention bias, while transition strength $\rho(i,j)$, field compatibility, and state-frontier evidence are summarized in the node and pair features. The model therefore receives one graph per task, not one global graph over the entire dataset.

Node representation. Each candidate tool node receives

$$x_i^R = [\mathrm{sim}_i,\ \mathrm{rrank}_i,\ \mathrm{hist}_i,\ a_i^{in},\ a_i^{out},\ z_i], \tag{14}$$

where the bracket denotes feature concatenation. The scalar rrank_i is reciprocal semantic rank. The remaining quantities are computed as

$$\mathrm{hist}_i = \log\left(1 + \sum_{u_j \in A_q} c(i \prec j) + c(j \prec i)\right), \tag{15}$$

$$a_i^{in} = \sum_{u_j \in U_q} \max(0, \rho(j,i)), \tag{16}$$

$$a_i^{out} = \sum_{u_j \in U_q} \max(0, \rho(i,j)), \tag{17}$$

$$z_i = [\chi_i(s_0),\ \mathrm{miss}_i,\ \mathrm{inSchema}_i,\ \mathrm{outSchema}_i,\ \mathrm{fam}_i]. \tag{18}$$

Here hist_i measures whether a tool has historical support near the semantic anchors, a_i^{in} and a_i^{out} measure whether it tends to consume or produce path state, $\kappa_{ij}^{schema} = |O_i \cap I_j| / (|I_j| + \epsilon)$ measures whether u_i can feed u_j, $\mathrm{inSchema}_i = \max_j \kappa_{ji}^{schema}$, $\mathrm{outSchema}_i = \max_j \kappa_{ij}^{schema}$, and fam_i is the tool-family feature from the documentation. The intuition is that a bridge tool can have weak query similarity but strong outgoing schema support, so the retriever scores possible path function in addition to textual relevance.

Graph encoding. The retriever applies an edge-biased graph Transformer:

$$H^0 = \mathrm{Proj}(X^R), \tag{19}$$

$$\mathbf{q}_i^{\ell,h} = W_Q^{\ell,h} h_i^\ell, \quad \mathbf{k}_j^{\ell,h} = W_K^{\ell,h} h_j^\ell, \tag{20}$$

$$\alpha_{ij}^{\ell,h} = \mathrm{softmax}_j \left(\frac{(\mathbf{q}_i^{\ell,h})^\top \mathbf{k}_j^{\ell,h}}{\sqrt{d_h}} + \beta_{\ell,h}(r_{ij}^R) \right), \tag{21}$$

$$\tilde{h}_i^{\ell,h} = \sum_j \alpha_{ij}^{\ell,h} W_V^{\ell,h} h_j^\ell, \tag{22}$$

$$H^{\ell+1} = \mathrm{Block}_\theta \left(H^\ell, \tilde{H}^\ell \right). \tag{23}$$

In this block, X^R stacks all node features x_i^R, H^ℓ is the hidden representation at layer ℓ, h_i^ℓ is node i's vector in that layer, h indexes an attention head, and d_h is the head dimension. $W_Q^{\ell,h}$, $W_K^{\ell,h}$, and $W_V^{\ell,h}$ are learned projections; Block_θ is the standard residual attention block with feed-forward transformation, and $\tilde{H}^\ell$ stacks the messages $\tilde{h}_i^{\ell,h}$ across heads. The learned bias $\beta_{\ell,h}(r_{ij}^R)$ is where retriever graph edges enter Transformer attention. It is the layer- and head-specific weight attached to an edge type: forward-transition neighbors, reverse-transition conflicts, and same-family tools therefore contribute different amounts of information during message passing. The intuition is that two tools with the same text similarity can receive different evidence if one is a predecessor bridge and the other is a reverse-order distractor. The encoder outputs membership and path-position distributions:

$$\hat{y}_i^{mem} = \sigma(w_m^\top h_i^L), \tag{24}$$

$$\hat{y}_{i,r}^{pos} = \mathrm{softmax}_r(W_r h_i^L). \tag{25}$$

Here L is the last graph-Transformer layer, σ is the logistic function, w_m maps the final node representation to a membership logit, and W_r maps it to position logits. The membership score asks whether the tool belongs in the 32-tool menu; the position score asks where the tool belongs in the latent path prefix. For exposition we call the first position *entry*, middle positions *bridge*, the position whose output most directly matches the requested answer *target*, and the last supervised position *terminal*. The supervised label remains the ordinal path position r.

Coverage decoding. Selecting the top K tools by $\hat{y}_i^{mem}$ alone can waste budget on near-duplicates. We therefore decode a set that balances local usefulness and path coverage:

$$\mathcal{C}_K = \arg \max_{\mathcal{C} \subseteq \mathcal{T},\, |\mathcal{C}| \le K} \sum_{u_i \in \mathcal{C}} \hat{y}_i^{mem} + \lambda \mathrm{Cov}(\mathcal{C}, P_q) - \eta_{\mathrm{red}} \mathrm{Red}(\mathcal{C}), \tag{26}$$

where P_q is the pre-execution frontier read off the predicted path positions and missing schema fields:

$$S_q = \{r : \max_i \hat{y}_{i,r}^{pos} > \zeta\}, \tag{27}$$

$$F_q^{miss} = \{f : f \in I_i,\ \hat{y}_i^{mem} > \zeta,\ f \notin F_0(q)\}, \tag{28}$$

$$P_q = S_q \cup F_q^{miss}. \tag{29}$$

The set S_q contains path slots the model believes should be represented, and F_q^{miss} contains input fields that likely tools need but the current state does not yet provide. The coverage and redundancy terms are

$$\mathrm{Cov}(\mathcal{C}, P_q) = \mathrm{Cov}_{slot}(\mathcal{C}, S_q) + \mathrm{Cov}_{field}(\mathcal{C}, F_q^{miss}), \tag{30}$$

$$\mathrm{Red}(\mathcal{C}) = \mathrm{Red}_{text}(\mathcal{C}) + \mathrm{Red}_{slot}(\mathcal{C}). \tag{31}$$

Cov_{slot} rewards path positions represented by at least one selected tool, and Cov_{field} rewards tools that can produce missing fields. Red_{text} penalizes near-duplicate documentation, and Red_{slot} penalizes selecting many tools predicted for the same path position. The coefficients λ and η_{red} set the strength of coverage and redundancy terms, and the constraint $|\mathcal{C}| \le K$ enforces the 32-tool budget. Because an exact search over tool subsets is intractable, we solve Eq. 26 greedily, adding at each step the tool with the largest marginal gain until the budget is full. This decoder is the bridge between retrieval and downstream execution: it asks whether the compact menu covers a usable chain.

D.3 State-Path Graph Reranker

The reranker receives a bounded set $\mathcal{C}_K$. At this point, the main danger is order: the right tools may be visible, but the agent can still be nudged toward a wrong first action or a dependency inversion. The reranker constructs a candidate-level graph $G_q^O = (V_q^O, E_q^O)$ over the tools in $\mathcal{C}_K$.

Inputs. The node set is exactly the retrieved 32 tools:

$$V_q^O = \mathcal{C}_K. \tag{32}$$

For each candidate, the ordering node feature is

$$x_i^O = [s_i^R,\ \mathrm{rank}_i,\ \hat{y}_i^{mem},\ \hat{y}_{i,\cdot}^{pos},\ \chi_i(s_0),\ \mathrm{miss}_i,\ \mathrm{fam}_i], \tag{33}$$

where s_i^R is the upstream retriever score, rank_i is its upstream rank, and $\hat{y}_{i,\cdot}^{pos}$ is the predicted position distribution from the retriever. There are no additional path-token nodes in this ordering graph. For each ordered pair (u_i, u_j), the edge vector

$$a_{ij}^O = [\rho(i,j), \rho(j,i), \Delta_{ij}^{rank}, \kappa_{ij}^{schema}, \kappa_{ij}^{path}] \tag{34}$$

contains forward transition support, reverse-direction conflict, rank difference, input-output schema compatibility, and path support. Here $\rho(i,j)$ and $\rho(j,i)$ are the forward and reverse transition priors reused from Eq. 12, Δ_{ij}^{rank} measures how far the two tools are separated in the upstream ranking, κ_{ij}^{schema} measures whether the output fields of u_i can feed input fields of u_j, and κ_{ij}^{path} measures whether the pair appears together in historical paths. These pair features are the edge attributes used by the reranker.

Graph Transformer reranking. The reranker applies an edge-biased graph Transformer:

$$Z^0 = \mathrm{Proj}(X^O), \tag{35}$$

$$\bar{\mathbf{q}}_i^{\ell,h} = \bar{W}_Q^{\ell,h} z_i^\ell, \quad \bar{\mathbf{k}}_j^{\ell,h} = \bar{W}_K^{\ell,h} z_j^\ell, \tag{36}$$

$$\alpha_{ij}^{\ell,h} = \mathrm{softmax}_j \left(\frac{(\bar{\mathbf{q}}_i^{\ell,h})^\top \bar{\mathbf{k}}_j^{\ell,h}}{\sqrt{d_h}} + (w_e^{\ell,h})^\top a_{ij}^O \right), \tag{37}$$

$$Z^{\ell+1} = \mathrm{Block}_\psi(Z^\ell, \alpha^\ell). \tag{38}$$

In this equation, X^O stacks the 32 candidate node features, Z^ℓ is the ordering-graph hidden state at layer ℓ, and $\bar{W}_Q^{\ell,h}, \bar{W}_K^{\ell,h}$ are learned query and key projections for the reranker. The vector $w_e^{\ell,h}$ turns the continuous edge feature a_{ij}^O into an attention bias. This is how schema compatibility and transition direction enter ordering, so the first visible tools are more likely to satisfy prerequisites. It produces four outputs:

$$e_i = \mathrm{Entry}(z_i), \tag{39}$$

$$m_i = \mathrm{Member}(z_i), \tag{40}$$

$$\phi_{i,r} = \mathrm{Slot}(z_i, r), \tag{41}$$

$$b_{ij} = \mathrm{Prec}(z_i, z_j, a_{ij}^O). \tag{42}$$

Here e_i scores whether u_i is a good first tool, m_i scores whether u_i belongs to the executable chain, $\phi_{i,r}$ scores its position in the path, and b_{ij} scores whether u_i should precede u_j. These outputs are intentionally separated because the same tool can be relevant to the task yet inappropriate as the first action. The model has S supervised slot heads. For decoding positions beyond that range, we use the final slot as a learned overflow bucket:

$$\phi_{i,r} = \phi_{i,S}, \quad r > S. \tag{43}$$

This gives all backup positions a valid late-position prior while keeping supervision focused on the prefix that most affects online execution.

Reranker supervision. The entry label is the first tool in the reference chain:

$$y_i^{entry} = \mathbb{I}[u_i = g_1]. \tag{44}$$

The slot label marks the tool at each supervised path position:

$$y_{i,r}^{slot} = \mathbb{I}[u_i = g_r]. \tag{45}$$

Here g_1 and g_r are the first and r-th tools of the reference trajectory $\tau = (g_1, \ldots, g_{T_\tau})$ introduced in the problem formulation. The precedence label follows the same definition as Eq. 8. The frozen reranker uses

$$\mathcal{L}_O = \lambda_{entry}\mathcal{L}_{entry} + \lambda_{slot}\mathcal{L}_{slot} + \lambda_{prec}\mathcal{L}_{prec} + \lambda_{member}\mathcal{L}_{member} + \lambda_{path}\mathcal{L}_{path}. \tag{46}$$

The coefficients $\lambda_{entry}, \lambda_{slot}, \lambda_{prec}, \lambda_{member}, \lambda_{path}$ are validation-selected weights, reported in Table 8, that prioritize entry, slot, precedence, membership, and path-prefix supervision. Each of $\mathcal{L}_{entry}$, $\mathcal{L}_{slot}$, and $\mathcal{L}_{member}$ is the cross-entropy between the corresponding head and its label, and $\mathcal{L}_{prec}$ is the pairwise loss of Eq. 8. $\mathcal{L}_{path}$ is a listwise path-prefix loss: at each prefix position, the model is rewarded for assigning the next gold tool the highest available path score. This loss directly supports online execution because the agent is most sensitive to the first few menu positions.

Menu decoding. The final menu is decoded by maximizing the path score $J(\pi)$ of Eq. 5. There $\mathrm{Inv}(\pi)$ penalizes high-confidence dependency inversions, and $g_{\pi_r,r}$ contains observable slot-state features for placing tool π_r at position r: state

compatibility, missing fields it produces, retrieval rank, and schema compatibility with the previous selected tool. The quantities ω_r^{slot}, μ, ν, ω^{inv}, and δ are fixed development-selected decoding constants. We build π greedily from left to right, placing at each position the remaining tool that maximizes the incremental score of Eq. 5, so decoding stays linear in the menu size. Eq. 5 turns the four prediction heads into one ordered menu: a tool can be highly useful through m_i, yet still move later if its slot and precedence scores say it depends on earlier state. The top part of π is the proposed execution path; the remaining candidates are appended as backup tools in decreasing utility order. This preserves a fixed menu size while giving the agent a clear prefix to follow.

Closed-candidate path alignment. Some benchmarks, such as UniToolCall, already provide a small candidate set whose main difficulty is ordering. For this setting, we reuse the same path score in Eq. 5 over the given candidates and omit the retrieval graph. The benchmark-specific alignment details are reported in the appendix; the ToolBench main system uses the learned retriever and graph reranker above.

D.4 Full Algorithms

The algorithms use the same two decisions as the main method: first select a 32-tool set for coverage, then order that set for execution. In Algorithm 10, $\mathcal{D}_{tr}$ is the trajectory-supervised training set, $\mathcal{D}_{dev}$ is the development set used only to freeze decoder constants, $\tau^\star$ is the gold tool trajectory for a training task, U_q is the broad pre-budget frontier, G_q^R is the retriever graph, and G_q^O is the ordering graph over the retrieved candidates. H_q^R and H_q^O are the hidden graph states produced by the two graph Transformers. $\mathcal{L}_R$ trains path coverage, while $\mathcal{L}_O$ trains entry, slot, membership, and precedence predictions. Ω denotes fixed decoding constants; after selection on development tasks, Ω is not changed during final evaluation.

Algorithm 11 is the frozen test-time path. The task q and tool library $\mathcal{T}$ are visible; the gold trajectory is absent. $\mathcal{C}_K$ is the retrieved 32-tool set, π_K is the ordered menu shown to the agent, A is the frozen execution agent, $\hat{\tau}$ is the executed tool-call trace, $\hat{a}$ is the final answer, E is the fixed official evaluator, and $\hat{y}$ is the binary success label. The menu is built once before the first tool call and is never rebuilt from observations.

Input: training tasks $\mathcal{D}_{tr}$, development tasks $\mathcal{D}_{dev}$, tool library $\mathcal{T}$, budget K, path-token count P, slot count S, learning rate α.
Output: retriever R_θ, reranker O_ψ, trajectory memory $\mathcal{M}$.
$\mathcal{M} \leftarrow \mathrm{BuildMemory}(\mathcal{D}_{tr}, \mathcal{T})$
for $(q, \tau^\star) \in \mathcal{D}_{tr}$ **do**
  $U_q \leftarrow \mathrm{Frontier}(q, \mathcal{T}, \mathcal{M})$
  $G_q^R \leftarrow (V_q^R, E_q^R), V_q^R = \{p_1, \ldots, p_P\} \cup U_q$
  $H_q^R \leftarrow R_\theta(G_q^R, q, \mathcal{M})$
  $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}_R(H_q^R, \tau^\star)$
  $\mathcal{C}_K(q) \leftarrow \mathrm{GreedySet}(H_q^R, K)$
end for
$\Omega \leftarrow \mathrm{SelectDecoderConstants}(\mathcal{D}_{dev})$
for $(q, \tau^\star) \in \mathcal{D}_{tr}$ **do**
  $G_q^O \leftarrow \mathrm{OrderGraph}(q, \mathcal{C}_K(q), \mathcal{M})$
  $H_q^O \leftarrow O_\psi(G_q^O, q)$
  $\psi \leftarrow \psi - \alpha \nabla_\psi \mathcal{L}_O(H_q^O, \tau^\star, \Omega)$
end for
return $R_\theta, O_\psi, \mathcal{M}, \Omega$

Table 10: **Algorithm 1: Training State-Path Tool Menus.** The retriever learns compact path coverage, and the reranker learns executable prefix order under fixed development-selected decoding constants.

Input: task q, tool library $\mathcal{T}$, R_θ, O_ψ, memory $\mathcal{M}$, decoder constants Ω, budget K, agent A, evaluator E.
Output: ordered menu π_K, execution trace $\hat{\tau}$, pass label $\hat{y}$.
$U_q \leftarrow \mathrm{Frontier}(q, \mathcal{T}, \mathcal{M})$
$G_q^R \leftarrow (V_q^R, E_q^R), V_q^R = \{p_1, \ldots, p_P\} \cup U_q$
$H_q^R \leftarrow R_\theta(G_q^R, q, \mathcal{M})$
$\mathcal{C}_K \leftarrow \mathrm{GreedySet}(H_q^R, K)$
$G_q^O \leftarrow \mathrm{OrderGraph}(q, \mathcal{C}_K, \mathcal{M})$
$H_q^O \leftarrow O_\psi(G_q^O, q)$
$\pi_K \leftarrow \mathrm{GreedyOrder}(H_q^O, \mathcal{C}_K, \Omega)$
$\hat{\tau}, \hat{a} \leftarrow A(q, \mathrm{Menu}(\pi_K))$
$\hat{y} \leftarrow E(q, \hat{\tau}, \hat{a})$
return $\pi_K, \hat{\tau}, \hat{y}$

Table 11: **Algorithm 2: Inference and online evaluation.** The system builds one fixed-budget menu before execution; the agent then acts under the same online protocol used for all compared systems.

E Concrete State-Path Examples

Figure 4 visualizes 3 real menu-construction patterns that occur in our evaluated tasks. The first pattern is a serial information-gathering chain: the task asks for countries, team names, players, and positions in the Women's World Cup 2023. The observable state contains the event name and the requested fields; the reranker orders a country-listing tool before a player-detail tool and leaves country-code lookup as backup. The second pattern is a parallel subgoal task: a weather-and-tide query provides coordinates and dates, so the reranker spreads the menu across tide, sunrise/sunset, sea-level, and

weather tools before merging their outputs. The third pattern is a schema-defined document task: a document identifier is already present, so the reranker follows the API pipeline from metadata to fields, signatures, invites, and final aggregation.

F Supplementary Results and Robustness

F.1 Retriever Budget and Coverage Baselines

Figure 5 reports the fixed-budget retrieval evidence that motivates the 32-tool setting. It also answers the common reviewer question of whether simply enlarging the official retriever scope would solve the problem. Complete-chain coverage is computed against a known successful reference chain. It is useful for checking whether the menu has the tools needed for one validated path, but it can undercount methods that find a different valid chain and can favor methods close to the training trajectory distribution. For this reason, we use it as a diagnostic for upstream coverage, while official online ToolEval success remains the main metric.

F.2 Published Native Baseline Values

Table 12 keeps the published paper numbers separate from the matched main table. Many recent systems report different benchmarks, candidate pools, and metrics, so the table records their native values separately from the matched main comparison.

F.3 Non-Graph Neural and Planner Controls

Figure 6 tests whether the main result can be explained by a strong non-graph planner, a pairwise sorter, a generation-style tool planner, or engineered dependency features. Each row is run through the same 32-tool online execution protocol where available, or through the closest frozen fixed-menu adapter specified in Appendix B.4. These controls are stronger than schema sorting alone because they include learned sequential ordering and paper-level planner adapters. They also separate the graph encoder from feature engineering: several rows receive the same state, schema, and trajectory features but remove candidate-candidate graph attention or state-path slot decoding. The remaining gap should therefore be read as evidence for the interaction between graph attention and state-path decoding, not as a claim that features alone are unimportant.

F.4 Plug-In Transfer Under Fixed Retrievers

Table 13 isolates the downstream menu organizer under a fixed upstream retriever. Tool-Rank is closest to a reranker-only method in its original form. ToolGen and SkillRouter include broader tool-generation or routing machinery in their papers, so here they are used only through their fixed-candidate ordering adapters. This gives two matched comparisons: under the official retriever, State-Path Graph Reranker beats the stock order and every tested external organizer; under State-Path Graph Retriever, selected external organizers also benefit from the stronger upstream menu, while the full cascade remains the strongest validated row.

F.5 Official Scope and Split-Level Checks

Table 14 reports two compact checks. First, widening the official candidate scope improves raw coverage, but our compact 32-tool menu already exceeds the official 128-tool coverage and is the row used for end-to-end comparison. Second, the gain holds across the 3 released ToolBench test groups rather than depending on a single aggregate or a single task type.

F.6 Cross-Model Generalization

Table 15 records the execution-model generalization matrix. It spans two open-weight model families (Qwen2.5 and Llama-3.1) and four scales from 7B to 72B; the 72B-AWQ row uses the same model as all matched main baselines, and every row is a measurement under the same fixed-budget interface.

F.7 Cross-Dataset Generalization

ToolBench remains the primary claim dataset because it provides the official online ToolEval protocol. We also test the menu-construction interface on adjacent benchmarks that expose stateful tools or multi-step trajectories. These experiments use each benchmark's native protocol because the targets differ: AppWorld asks for the next executable application call, TRAJECT-Bench asks whether a compact menu covers the complete tool chain, and UniToolCall asks whether the predicted prefix matches a multi-hop trajectory. For every non-ToolBench benchmark, training and validation are done inside that benchmark only; no ToolBench final feedback is transferred across datasets. Table 16 reports the native metric and training scope for every benchmark. The current evidence shows

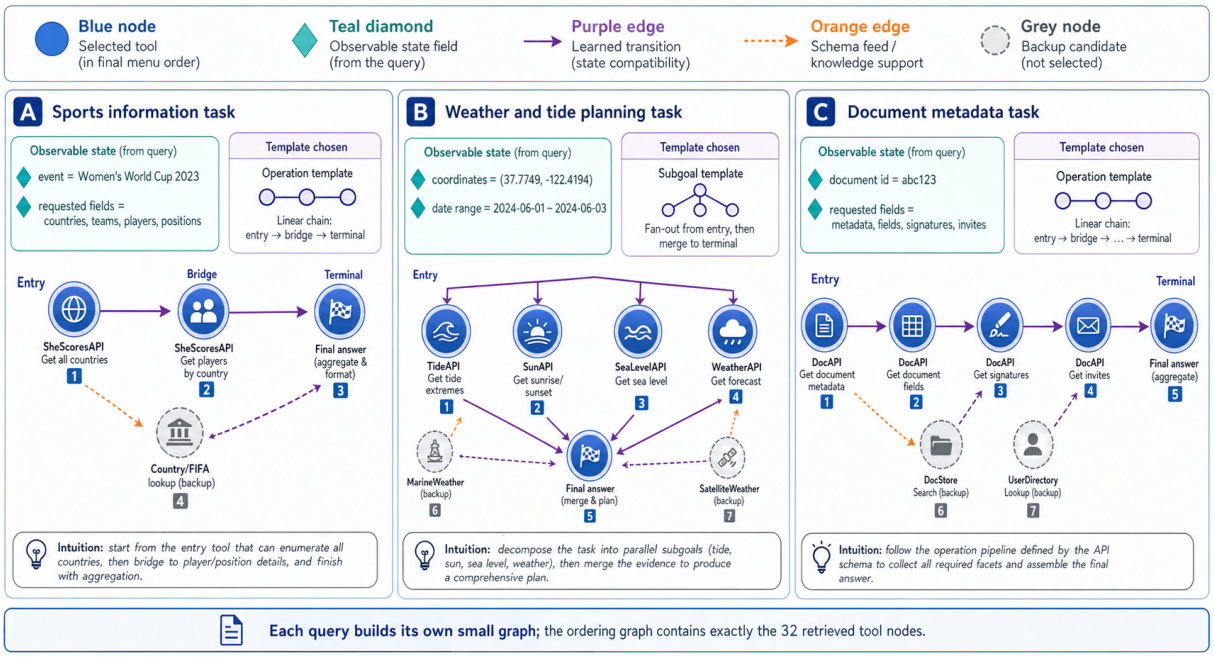

Figure 4: **Concrete state-path graph instances.** Each panel shows the observable state extracted before execution, the ordered prefix exposed by the reranker, and backup candidates retained in the 32-tool menu. Blue nodes are selected tools, teal diamonds are observable state fields, purple edges are learned transition evidence, orange edges are schema-feed evidence, and grey nodes are backup candidates. These examples illustrate the same implementation rule used in the method: the retriever graph is query-specific, and the final ordering graph contains exactly the retrieved tool nodes.

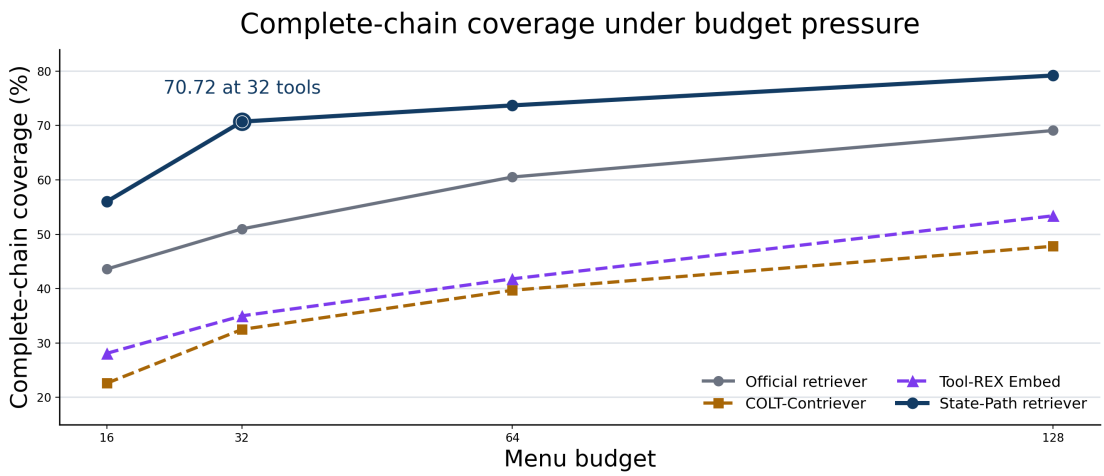

Figure 5: **State-Path Graph Retriever recovers more complete reference chains under a tight budget than dense retrieval baselines and wider official-retriever controls.** At 32 tools, State-Path Graph Retriever reaches 70.72% complete-chain coverage, while the official retriever reaches 50.99% at the same budget and 69.08% only at 128 tools. Each curve is measured on the same released ToolBench evaluation set and tool library.

the same pattern in 3 external settings: state-path learning improves compact tool menus beyond semantic retrieval and graph expansion alone.

F.8 Component-Level Evidence

Table 17 links each component to the metric it protects. Retrieval components raise compact complete-chain coverage, while reranker components raise official end-to-end success.

System family	Method	Native setting and metric	Paper value	State-Path Tool Menu and gain
ToolBench system	ToolLLaMA + official retriever	ToolBench main experiments; pass rate	0.673	**0.898**; **+0.225**
Generative tool caller	ToolGen	ToolBench ToolEval setting; official pass rate	0.730	**0.898**; **+0.168**
Tool retriever	ToolRet trained retriever	ToolBench pass-rate setting; pass rate	0.600	**0.898**; **+0.298**
Tool retrieval/ranking	Tool-REX Tool-Rank	ToolBench retrieval setting; NDCG@10 / R@10	0.564 / 0.678	**0.713 / 0.736**; **+0.149 / +0.058**
Field-aware retriever	Multi-Field Tool Retrieval	ToolBench retrieval setting; NDCG@10 / R@10	0.585 / 0.674	**0.713 / 0.736**; **+0.128 / +0.062**

Table 12: **Contextual native-metric comparison against published paper values.** This appendix table reports each paper's original metric and setting alongside our corresponding native-metric result. Protocols differ across papers, so this table is contextual; Table 1 remains the controlled comparison.

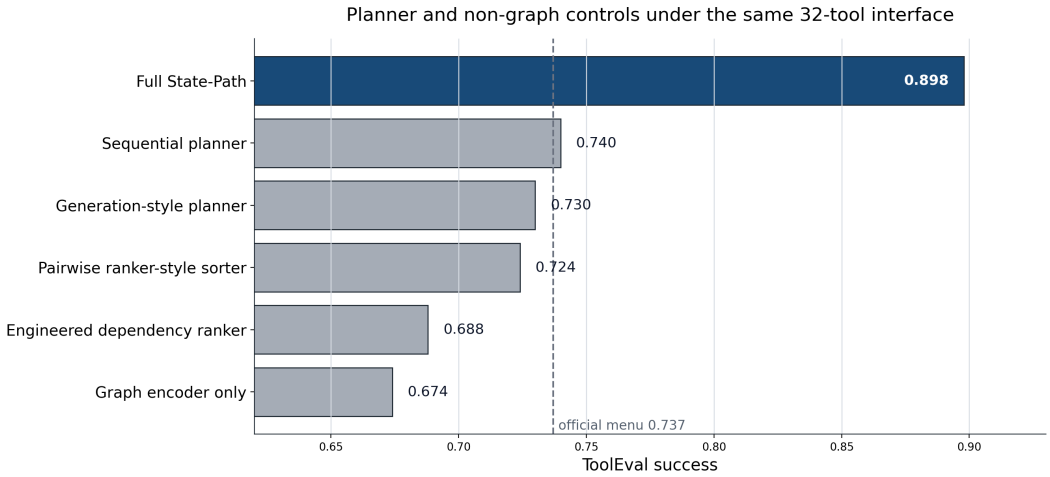

Figure 6: **Non-graph and planner controls do not close the gap.** The strongest non-graph sequential planner reaches 0.740, while the full state-path graph system reaches 0.898 under the same online success metric.

Upstream retriever	Downstream organizer	Original paper scope	Success	Gain vs upstream order
ToolBench official	stock order	retrieval order control	0.737	0.000
ToolBench official	ToolGen	generation and tool calling	0.730	-0.007
ToolBench official	Tool-REX Tool-Rank 4B	tool reranking	0.724	-0.013
ToolBench official	Tool-REX Tool-Rank 8B	tool reranking	0.717	-0.020
ToolBench official	SkillRouter	skill routing	0.707	-0.030
ToolBench official	**State-Path Graph Reranker**	**state-path menu organization**	**0.773**	**+0.036**
State-Path Graph Retriever	stock order	retrieval order control	0.727	0.000
State-Path Graph Retriever	ToolGen	generation and tool calling	0.743	+0.016
State-Path Graph Retriever	Tool-REX Tool-Rank 4B	tool reranking	0.734	+0.007
State-Path Graph Retriever	Tool-REX Tool-Rank 8B	tool reranking	0.727	+0.000
State-Path Graph Retriever	SkillRouter	skill routing	0.704	-0.023
State-Path Graph Retriever	**State-Path Graph Reranker**	**state-path menu organization**	**0.898**	**+0.171**

Table 13: **Plug-in transfer under fixed retrievers.** Rows keep the upstream candidate set fixed and vary only the organizer; official end-to-end ToolEval success is reported on all 304 tasks.

Menu source	Budget	Coverage / status	Role
Official	32	50.99; success 0.737	main baseline
Official	64	60.53; coverage only	wider list
Official	128	69.08; coverage only	wider list
State-Path Tool Menu	32	**70.72**; success **0.898**	full system

Method	G1	G2	G3	Avg.
ToolBench official	0.738	0.731	0.746	0.737
State-Path Graph Retriever + original order	0.776	0.649	0.810	0.727
State-Path Tool Menu	**0.907**	**0.881**	**0.921**	**0.898**

Table 14: **Scope and split checks support the same controlled result.** Left: the 64-tool and 128-tool official rows are coverage controls, while the 32-tool state-path row is the compact end-to-end system. Right: G1, G2, and G3 are the released ToolBench task-complexity groups; the state-path gain is present in every group.

Execution model	Access	ToolBench official	State-Path Tool Menu	Gain
Qwen2.5-7B-Instruct	local open-weight	0.333	**0.453**	**+0.120**
Qwen2.5-32B-Instruct	local open-weight	0.387	**0.520**	**+0.133**
Llama-3.1-70B-Instruct	local open-weight	0.387	**0.507**	**+0.120**
Qwen2.5-72B-Instruct-AWQ	local open-weight	0.737	**0.898**	**+0.161**

Table 15: **State-path menu gains hold across two open-weight model families and four model scales.** Each row below 72B fixes 75 sampled tasks (25 per ToolBench split), the 32-tool budget, and the same ToolEval evaluation protocol; the 72B row uses all 304 tasks. Even at 7B scale, the state-path menu already improves over the official baseline (0.333 to 0.453, $+0.120$). At 32B and 70B, the state-path reranker unlocks nested G3 tasks that consistently fail under official ordering ($+0.133$ and $+0.120$ respectively): our reranker turns 15 of 25 and 15 of 25 nested tasks into successes, compared to 8 and 5 for the official ordering. At 72B-AWQ, the same gain is confirmed on the full 304-task set ($+0.161$). The gain holds across all four scales and is largest on the strongest 72B model.

Benchmark	Metric	Evaluation split	Control	State-Path Tool Menu	Gain
ToolBench/ToolEval	official success	released set	0.737	**0.898**	**+0.161**
AppWorld	next-tool R@1	development split	0.219	**0.465**	**+0.246**
TRAJECT-Bench hard	complete-chain@32	hard split	0.350	**0.412**	**+0.062**
TRAJECT-Bench hard	complete-chain@64	hard split	0.459	**0.611**	**+0.152**
TRAJECT-Bench mixed	complete-chain@32	held-out mixed split	0.420	**0.621**	**+0.201**
UniToolCall multi-hop	ordered-prefix	hash split	0.297	**0.635**	**+0.338**

Table 16: **State-path menu construction improves the native metric on four tool-use benchmarks with distinct task structures.** Each benchmark uses its native protocol; the surrounding text explains why each metric is the benchmark's primary diagnostic.

Removed or replaced component	Main failure exposed	Diagnostic metric	Measured change
Official retrieval for our retriever	Missing bridge tools	complete-chain coverage	50.99 $\rightarrow$ 70.72
Original order for our reranker	Wrong first action and inversions	official success	0.727 $\rightarrow$ 0.773
No frontier path decoding	Fewer state-transition cues	official success	0.750 $\rightarrow$ 0.898
No path-aware slots	Prefix instability	official success	0.740 $\rightarrow$ 0.898
Document-derived order only	Static edges miss current state	official success	0.688 $\rightarrow$ 0.898
No historical path gain	Lost reusable execution fragments	complete-chain coverage	70.72 $\rightarrow$ 66.12
No sequence-memory score	Lost execution-experience prior	complete-chain coverage	70.72 $\rightarrow$ 58.55

Table 17: **Each component protects a distinct part of the state path.** Coverage-oriented rows report complete-chain coverage; ordering-oriented rows report official ToolEval success.