ToolGraph as a Q-Function for Test-Time Scaling of Tool-Use Agents

Ke Shen[1] **Ankita Bhaumik**[1] **Li Dong**[1] **Shengjie Liu**[1]
Shuzhen Li[2] **Naveed Ahmed Janvekar**[1]

[1]Amazon [2]Princeton University
{ldonga, kkshen, bhaumika, zycjlsj, njjanvek}@amazon.com
sl8808@princeton.edu

Abstract

Tool-use agents must compose long chains of interdependent calls, and the dependency structure of a particular deployed catalog—which outputs satisfy which later inputs, and in what order—is not something a general-purpose language model reliably infers from tool descriptions alone. We treat that structure as an external representation to be compiled once and reused. ToolGraph is a directed, weighted dependency graph over a deployed tool catalog: parameter matching and interventional probes independently propose edges, a probe executes a candidate chain with and without its proposed predecessor, and the resulting evidence is combined into an edge confidence. Prior work couples such a graph to a language model by retrieving relevant tools, searching it for tool paths, or scoring the call that immediately follows. We ask instead whether the same structure can supply a *multi-step* value, by composing graph paths into a goal-conditioned action value $Q(s, a, g)$. We study three estimators—analytic dynamic programming over entity sets, an inductive GraphSAGE estimator distilled from it, and an LLM scorer conditioned on graph evidence—together with a verbalizer that returns action–goal feedback to the original agent, which retains final action selection. Across two tool-use benchmarks and three backbones we compare graph-derived value ranking against graph-derived retrieval ranking over the identical graph, and against direct and reflective agents. Composition pays off only when the estimator also reads the concrete state: the LLM estimator improves on personalized-PageRank ranking of the same edges in every benchmark–backbone setting, by 8.3 points on average, while the two purely structural estimators fall below it, because a value defined over tool types cannot separate a correct tool called with a wrong argument. Graph construction is a one-time cost reused across all later requests; we report it separately rather than assume it amortized away — at our evaluation scale it is comparable to request-time cost — and sweep four knobs controlling how much graph evidence to expose against both success and tokens.

1 Introduction

Language-model agents extend text generation into interaction by invoking tools and incorporating their observations [25]. The difficulty is not usually calling a single tool correctly but *chaining*: long-horizon tasks require discovering which tool's output supplies a later tool's input, in what order, and under which preconditions. That dependency structure is a property of the deployed catalog rather than of language in general, and it is only weakly recoverable from tool descriptions read one at a time — against an expert-annotated, parameter-level API graph, models reading documentation alone classify dependency relations at only 42.3–70.0% accuracy [11] — which is why large-catalog

40th Conference on Neural Information Processing Systems (NeurIPS 2026). Workshop: From Pretrained Representations to Acting Agents (PTA).

methods add explicit tool representations, retrieval, or tool-use supervision [6, 17]. The problem is sharpest in enterprise deployments, where catalogs are large and workflows organization-specific, but it is already measurable on public long-horizon benchmarks, which is where we study it.

ToolGraph externalizes this structure: a node is a tool and its interface, a directed edge records that one tool can supply information another needs. Coupling such a graph to a frozen language model is established practice [3, 16, 11]; we take the graph and its construction pipeline as given (Section 3.2, Appendix A) and claim neither as a contribution, its schema-derived edge proposal following Liu et al. [13]. Our question is what else the same structure can do. Prior work retrieves tools with such a graph, searches it for tool paths [21], or scores the call that immediately follows [10]; none composes it into a multi-step, goal-conditioned action value — given the current state and an inferred goal, which candidate action has the greatest expected *downstream* utility? That makes the comparison between the two uses of one graph, ranked by query-conditioned personalized PageRank versus ranked by a composed value, the central experiment of this paper. It also separates two kinds of inference cost: value-guided search and verifier-based reranking both spend request-time compute per request [24, 29, 18, 4, 12, 20, 23], whereas graph construction is a one-time, catalog-level investment, so we report the two separately rather than folding them together.

We study three ways to derive Q-values: analytic dynamic programming, an inductive graph estimator, and an LLM estimator that also sees the full interaction context. A separate verbalizer presents action–goal values to the original agent without forcing numerical argmax selection. Figure 1 traces one decision: the compiled graph is queried for paths from the entities the agent currently holds toward candidate goal tools; each candidate action is scored by one of the estimators of Section 3.3; the top-k action–goal values and their supporting edge evidence are verbalized (Section 3.4); and the planner then chooses its next action. Our contributions are: (i) recasting a compiled tool-dependency graph from a retrieval structure into a source of *multi-step*, goal-conditioned action values, with the controlled retrieval-versus-value comparison over identical structure that isolates the difference; (ii) three Q-estimators spanning auditable, learned, and semantic regimes; and (iii) a four-knob study of how much graph-derived *value evidence* to expose — subgraph radius, edge-confidence threshold, per-node annotation depth, and number of scored candidates — against both task success and token cost.

The comparison returns a qualified answer, which we state up front: composing the graph's edges into a multi-step value beats ranking the same edges by relevance only when the estimator can also read the concrete state. The LLM estimator improves on personalized-PageRank ranking of the identical graph in all six benchmark–backbone settings, by 8.3 points on average, while the two purely structural estimators fall *below* it, because a value defined over tool types cannot see a correct tool called with a wrong argument.

2 Related Work

Tool graphs and tool retrieval. Large tool catalogs motivate explicit structure over tools: ToolkenGPT learns per-tool embeddings [6], and ToolLLM pairs a large API corpus with a dual-encoder retriever selecting which tools enter the prompt by query–document similarity [17] — the structure-free control against which graph-derived scores must be read. Structure-aware methods couple a dependency graph to a frozen planner in three ways. They *retrieve* with it, encoding a request-specific graph into a continuous token [3] or injecting a masked, probability-annotated tool set fitted from demonstrations [16]. They *search* it: Wu et al. [21] cast planning as selecting a connected subgraph of a task graph and find a trained GNN node scorer matches or beats greedy, adaptive and beam search while using 5–10$\times$ fewer tokens. Or they *score* the next call from it, as AutoTool does with a one-step transition statistic accumulated from an agent's own executions [10]. Reverse Chain is the prompt-only precedent for our goal formulation — pick the API that satisfies the request, then recursively select type-matched predecessors for its unfilled arguments — but its backward search happens inside one generation and leaves no reusable structure [27], while In-N-Out supplies an expert-annotated, parameter-level API graph as a one-hop oracle, leaving graph traversal for planning to future work [11]. Two differences separate us from all of these: their structure is mined from invocation histories, demonstrations, or expert labels, whereas ours needs no task data; and their graph-derived quantity is a relevance score or a one-step statistic, not a multi-step value with a goal-conditioned terminal boundary. Our retrieval arm ranks the same graph by personalized PageRank, and it is the baseline the value-based estimators must beat.

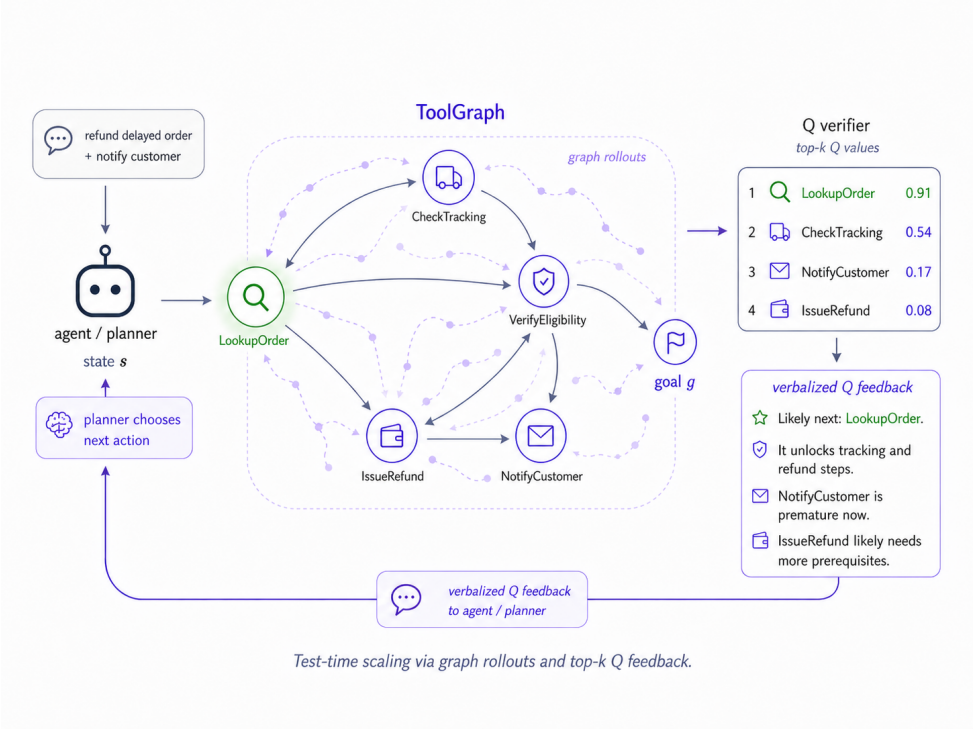

Figure 1: ToolGraph externalizes domain-specific tool dependencies and uses graph-derived Q-values to score candidate actions. The top-k values are verbalized for the original agent, which retains final action selection.

Value functions for language agents. The established route to an action value is gradient-based multi-turn RL: ArCHer trains an utterance-level critic by temporal-difference backup on online, reward-labelled rollouts [30]. Closer to our deployment shape, three lines keep the policy frozen. Hong et al. [9] fit a goal-conditioned critic offline and, at inference, verbalize its reachability estimates so a frontier model can refine its own plan without environment search — the nearest published neighbour to "compiled value, verbalized to a frozen planner." Self-Taught Lookahead reaches a value without reward labels by fine-tuning a small value model to verbalize one lookahead step [14], and Reflexion is the weight-free precursor, converting realized outcomes into verbal memory [19]. What separates us is not weight access — several leave the policy untouched — but *supervision*: each fits its value on task-specific trajectories, reward, or environment feedback, whereas ours reads none of the three. What we give up is expressiveness: our state abstraction is a set of entity names, not a learned representation of the trajectory — and Section 4.2 shows that this is what limits the structural estimators.

Verifiers, process rewards, and test-time compute. Sampling several candidates and selecting with a verifier is a standard way to spend inference compute [4], refined by step-level process reward models [12], generative verifiers [26], compute-optimal allocation studies [20], and methods that reunify reasoners with verifiers [18]. In the agentic setting, Xia et al. [22] train a process reward model on state values extracted from search trees and reuse one checkpoint to steer an unmodified policy; notably, an untrained judge of the same size scores slightly *below* greedy decoding there, which is why a no-graph pole is a necessary control rather than a formality. Our Q_{llm} estimator sits in this family, differing in that its prompt carries graph-derived evidence and a structural prior rather than the interaction alone. We include no trained agentic reward model, because every published one requires the reward-labelled rollouts our setting assumes are unavailable, and no search-based arm [24, 29, 23], because branching a tool environment requires resettable state and our pilot search runs did not complete in budget.

3 Method

3.1 Problem Formulation

Let $\mathcal{T}$ be the deployed tool catalog. At a decision point, state s contains the user request, conversation, previous tool calls, and observed results. The agent considers actions $a \in \mathcal{A}(s)$, where each action is a tool call with arguments and corresponds to a ToolGraph node. For goal g, $Q(s, a, g)$ estimates the expected future task value after taking a and continuing from the resulting state.

ToolGraph is constructed for the deployed catalog before serving requests and without updating the base model, injecting tool interfaces, domain relationships, and probe evidence that need not reside in the model parameters. During a request it supplies features and values for the candidate actions; it is not a separate planner, and the original agent remains responsible for its next action.

3.2 Constructing the ToolGraph

ToolGraph is a directed, weighted graph over tools. A node stores a tool's description and input/output schemas; an edge $A \to B$ states that running A may supply information needed by B. We store

$$v_a = (\mathrm{name}_a, \mathrm{class}_a, \mathrm{domain}_a, \mathrm{description}_a, \mathsf{In}_a, \mathsf{Out}_a),$$
$$\epsilon_{ab} = (a, b, w_{ab}, \mathsf{Match}_{ab}, \mathsf{Evidence}_{ab}).$$

$\mathsf{In}_a, \mathsf{Out}_a$ contain field names, types, and descriptions, and Match_{ab} identifies output–input correspondences.

Two proposers run independently over tool pairs and their evidence is merged per pair. *Parameter matching* proposes an edge when a field in Out_a satisfies a required field in In_b; it is cheap but applies only to tools with well-typed declared interfaces. An *interventional probe* treats each tool as a black-box service: a language model derives a candidate dependency and synthesizes a probe query for it, the query is then *executed* under the composition $A \to B$ and under a baseline that omits A, and an evaluator compares the two outcomes to score how much A contributed. This second route is what covers tools whose output semantics are not declared in a schema and must be established empirically. Discovering edges by execution has a precedent in ToolExpNet, whose LLM-synthesized trial-and-error episodes delete edges that do not hold [28]; there the execution signal resolves a binary keep-or-remove decision, here it is a with-versus-without-predecessor contrast yielding the graded confidence the estimators consume. Each retained piece of evidence contributes a confidence, and w_{ab} is their combination; $\mathsf{Evidence}_{ab}$ stores the probe query, the chain and baseline signals, and construction provenance. Edges with w_{ab} below a per-benchmark threshold θ_{prune} are discarded. Appendix A gives the fuller walkthrough, Table 3 reports the resulting graphs, and Table 4 reports what construction cost.

Two properties of the artifact matter downstream. First, the graph records which outputs can satisfy which later inputs and which orderings have empirical support — exactly the chaining structure that is hard to read off tool descriptions individually — but it does *not* contain complete task trajectories. It supplies reusable *local* transition evidence, and it is the Q-estimators of Section 3.3 that compose that evidence across multi-tool paths. Second, the evidence is catalog-level: edges are statements about tools, not about particular argument values, which bounds what any graph-derived value can discriminate — the bound that turns out to decide our main result (Section 4.2). At decision time we retrieve paths from currently available information toward candidate goal tools and expose their inputs, outputs, edge confidence, and prior call outcomes.

3.3 Graph-Derived Q Estimators

Graph construction provides domain-specific transition evidence; the remaining problem is to translate it into the value of acting now. Q_{dp} explicitly composes graph paths, Q_{emb} learns transferable graph representations, and Q_{llm} combines graph evidence with the full semantic context.

An entity-set relaxation. An *entity* is a named datum passed between tools, such as `order_id`. Let $E(s)$ be the entities available in state s, and let I_a, O_a be action a's required and produced entities. Action a is executable when $I_a \subseteq E(s)$. Writing $e = E(s)$, a successful call yields $e^a = e \cup O_a$ and a failed call leaves e unchanged. Because effects only add entities, the reachable state space

is a finite lattice: this is a probabilistic *delete-free* (monotone) relaxation of the tool-use problem, and the backup below is a probabilistic analogue of the relaxed-reachability heuristics of classical planning [2, 8]. We adopt it because it turns valuation into a small fixed-point computation instead of a search over concrete states, and we claim no novelty for the relaxation itself. What is new here is that the transition model is compiled from tool interfaces by a language model with no trajectories and no reward labels, that goals are tools rather than declared atoms, and that the resulting values are consumed by a frozen planner as injected evidence.

Goals, terminal values, and termination. We take g to be a *goal tool*: a node whose becoming executable would satisfy the request. Selecting a goal API first and then working backwards over its unfilled arguments is the strategy of Reverse Chain [27]; we solve the same backward reachability problem once over a persistent, weighted graph instead of inside each generation. Candidate goal tools $\mathcal{G}(s)$ are inferred from the request and the graph (Section 3.4); when inference returns none, the estimator abstains and returns a neutral score rather than a ranking. A state is terminal when some $g \in \mathcal{G}(s)$ is executable in it, i.e. $I_g \subseteq e$. Terminal states take value $V_g = +1$ and are not expanded further; a non-terminal state with no executable action is a dead end with $V_g = -1$. Values therefore lie in $[-1, 1]$, and goal attainment enters only through the terminal boundary: the immediate reward is identically zero, so Eq. (1) carries no separate reward term.

Analytic dynamic programming (Q_{dp}). For a non-terminal e,

$$\bar{Q}_g(e, a) = \gamma \left[p_a V_g(e^a) + (1 - p_a) V_g(e) \right], \qquad V_g(e) = \max_{a:\, I_a \subseteq e} q_g(e, a), \tag{1}$$

with $V_g(e) = +1$ on terminal states and -1 when no action is executable. The failure branch $(1 - p_a)V_g(e)$ is a self-loop in the *value recursion*, not a licence to retry an action indefinitely in an episode. We obtain a fixed point of Eq. (1) by value iteration over the reachable lattice; with $\gamma < 1$ the unshrunk backup is a contraction, while the shrinkage of Eq. (2) can inflate the modulus to $\gamma(1 + \kappa)$, so we verify convergence numerically at tolerance 10^{-6} rather than assert uniqueness from $\gamma < 1$ alone. We enumerate that lattice by breadth-first closure from the entity set the agent currently holds, stopping at terminal states and under an explicit cap on the number of enumerated states; Appendix B reports that cap and the solver's convergence settings. Because a required field that no tool produces can only come from the conversation, reachability is computed over tool-producible entities only, and conversation-supplied arguments are checked instead at the candidate level below.

Confidence shrinkage. Graph edges carry a confidence, and we damp values that rest on weak or inconsistent edges:

$$q_g(e, a) = \bar{Q}_g(e, a) - \kappa(1 - c_a)\left|\bar{Q}_g(e, a)\right|, \qquad \kappa \in [0, 1]. \tag{2}$$

This is not a lower confidence bound. Because the penalty scales $|\bar{Q}_g|$ but is subtracted, it damps optimistic values toward zero and *deepens* pessimistic ones: for $\bar{Q}_g > 0$ the factor is $1 - \kappa(1 - c_a)$, for $\bar{Q}_g < 0$ it is $1 + \kappa(1 - c_a)$. It is therefore a magnitude penalty on dispersion in the supporting evidence, not a shrinkage toward neutral, and we clamp to $[-1, 1]$ before the affine map below. Because V_g feeds back into Eq. (1), the factor applies once per backup step, so a value supported by a d-step chain is penalized more than a one-step value; this penalizes long dependency chains. We fix $\kappa = 0.3$ rather than tuning it, and we do not sweep it; since the structural estimators lose to relevance ranking on the graph regardless (Section 4.2), a κ sweep is unlikely to be where their gap lies. We also note that p_a and c_a are not independent evidence channels: both are aggregations of the *same* incoming edge weights for a — p_a their maximum, c_a their mean — so Eq. (2) acts as a dispersion penalty over that evidence rather than as a second measurement. Tools with no incoming edges receive a neutral value for both. Runtime evidence enters as inputs, not structure: a tool that has failed in this episode and never succeeded is assigned $p_a = 0$.

From values to candidate scores. Candidates are proposed actions, which may be tool calls or a message to the user. We set $Q_{\mathrm{dp}}(s, a, g) = q_g(E(s), a)$ and map $[-1, 1]$ onto $[0, 1]$ monotonically. Three cases are handled explicitly. A candidate whose tool is absent from the graph, whose required fields are neither available nor supplied by a conversation-grounded argument, or which has been marked a dead end, is scored strictly worst; this admissibility filtering is a by-product of the relaxation rather than an objective. A message-only candidate is scored $\gamma V_g(E(s))$, the value of leaving the entity state unchanged, which places it between goal-directed and inadmissible calls — a modelling

assumption, since a catalog-level graph cannot assess conversational value. A candidate proposing several calls composes them sequentially, with p the product of the per-tool values and c their minimum. This estimator is fast, auditable, and makes no model calls, but it cannot distinguish two candidates that call the same tool with different argument values, nor two states with the same fields and different histories.

Inductive graph estimator (Q_{emb}). We learn GraphSAGE [5] over schema embeddings, probe scores, degree statistics, and input/output counts. Let z_a embed the action, z_s pool successful tools, and z_g pool inferred goals:

$$Q_{\mathrm{emb}}(s, a, g) = \sigma(\mathrm{MLP}[z_a \,\|\, z_s \,\|\, z_g])\,. \tag{3}$$

We distill normalized Q_{dp} targets over reachable states. Training a graph network over a tool graph to score nodes for a frozen planner is not new—Wu et al. [21] compare exactly this family, including GraphSAGE—and the difference here is the target: our regression labels are distilled Q_{dp} values, which require no task annotation, rather than a ranking loss on gold step–tool pairs. The motivation is amortization rather than generalization to new catalogs: once trained, Q_{emb} scores a candidate with a single forward pass and no lattice enumeration, whereas Q_{dp} re-solves per decision and per goal. Neighbourhood aggregation additionally lets it assign a value to a tool the relaxation leaves unvalued, provided the tool is connected. We do not test transfer to unseen catalogs.

Language-model estimator (Q_{llm}). The LLM sees the full state, candidate arguments, goal, policy, alternatives, verbalized graph evidence, and the Q_{dp} value as a prior. It returns a 0–10 score and rationale; several scorer samples are averaged and normalized to $[0, 1]$. This lets the estimator account for concrete entity values, conversation state, and policies that catalog-level graph structure does not represent, at higher request-time cost. Note that the three estimators are nested rather than independent — Q_{emb} is distilled from Q_{dp}, and Q_{llm} is conditioned on it — so the comparison among them is a comparison of how much each adds over the one it contains, not of three independent methods. The flat retrieval control of Section 4.1 is our no-graph pole; we discuss there what that control does and does not establish about a graph-free scorer.

3.4 Top-k Retrieval and Natural-Language Q Feedback

Let $\mathcal{G}(s)$ be the candidate goals inferred from the request and graph. We cheaply recompute Q_{dp} for each goal to form

$$M[a, g] = Q_{\mathrm{dp}}(s, a, g), \qquad a \in \mathcal{A}(s),\ g \in \mathcal{G}(s). \tag{4}$$

A verbalizer converts each row and its graph evidence into action–goal feedback for the agent. It neither selects the argmax nor estimates a new value; unlike Q_{llm}, it only translates existing values. Its output is feedback plus a recommended action, which the planner may disregard, and it costs one model call per decision — included in the token accounting of Section 4.3.

Three distinct budgets appear below and we fix the notation here. The *injection width* k is the number of action–goal rows placed in the prompt, each carried with its own retrieved subgraph, whose size is controlled separately by hop depth h and edge threshold θ; the *candidate count* n is the number of actions the planner proposes per decision for the estimator to rank; and the number of Q_{llm} scorer samples averaged per candidate controls scoring fidelity. These give modest task-time compute controls without requiring online environment search.

4 Experiments

4.1 Setup

Evaluation. We use ToolComp [15] and τ^2-Bench [1] with Ministral 3 8B, Gemma 4 31B, and DeepSeek V3.2 685B. Baselines are Native, ReAct [25], and Reflexion [19], covering direct action generation, reasoning with observations, and reflection.

Two controls isolate what the graph contributes. **Flat top-k retrieval** ranks tools by embedding similarity between the request and each tool's description and injects the top k into the planner's context, with no graph, no propagation, and no scorer — the structure-free counterpart of the retrieval interface used by dual-encoder tool retrievers [17]. It is obtained from the same code path by removing

Table 1: Task success rate (%). Flat top-k retrieval and the four ToolGraph rows share the same $k = 8$ injection interface and prompt template; they differ only in how the injected tools are ranked, and only the ToolGraph rows use the graph. M, G, and D denote Ministral 3 8B, Gemma 4 31B, and DeepSeek V3.2 685B. Bold marks the best value in each column. Verifier: Gemma 4 31B for every scored arm; ToolComp answers are additionally graded by the benchmark's own answer judge, held fixed across arms.

	ToolComp			τ^2-Bench			
Method	M	G	D	M	G	D	Mean
Native	56.9	72.6	73.1	40.0	67.0	72.0	63.6
ReAct	46.7	65.0	62.4	38.0	52.0	63.0	54.5
Reflexion	50.8	69.5	72.6	49.0	69.0	75.0	64.3
Flat top-k retrieval	53.8	71.6	72.1	34.0	62.0	72.0	60.9
TG top-k PPR	54.8	74.1	69.5	37.0	66.5	74.0	62.7
TG top-k Q_{dp}	55.8	71.6	73.6	38.0	54.0	66.6	59.9
TG top-k Q_{emb}	57.6	69.5	68.4	36.0	50.4	66.0	58.0
TG top-k Q_{llm}	**62.0**	**79.7**	**78.0**	**53.0**	**73.0**	**80.0**	**71.0**

graph propagation, so the embedding, k, and prompt template are identical to the ToolGraph arms; the difference is only whether dependency structure informs the ranking. Its purpose is to separate the *interface* from the *structure*, so that any gain the ToolGraph arms show over it is attributable to the graph rather than to k tools being in the context. It is also our stand-in for deleting the graph from Q_{llm}, on the untested assumption that a scorer with no graph evidence adds little over similarity ranking; if that is wrong, the flat arm understates the no-graph condition. **TG–PPR** then ranks the same catalog by query-conditioned personalized PageRank over the weighted graph. Every ToolGraph variant shares the graph, the candidate pool, the verbalizer, and the top-k interface; only the ranking changes, although Q_{llm} also spends request-time scorer calls that the other arms do not. Flat versus TG–PPR therefore isolates dependency structure from plain relevance, and TG–PPR versus the three estimators isolates multi-step value composition from local graph relevance. Each ToolGraph is constructed once per benchmark and reused across tasks and backbones.

We select the injection width $k = 8$ on a stratified development split (20% of non-test tasks) and preserve official test partitions. Every arm draws $n = 4$ candidate actions per decision, so the estimators rank identical candidate sets, and Table 2 sweeps n. Success is the benchmark-native binary reward on the τ^2 test split of 100 tasks (airline 20, retail 40, telecom 40) and the ToolComp evaluation split of 214 tasks; construction cost is reported separately from request-time tokens. Three models are held fixed while only the planner varies: the scorer and verbalizer (Gemma 4 31B), the τ^2 user simulator (Sonnet 4.5), and the ToolComp answer judge.

4.2 Results

Table 1 is laid out so that three successive differences each isolate one step.

The interface costs success; dependency structure recovers most of it. Native to flat top-k retrieval isolates the *interface*: k tool descriptions in the context, ranked by request–description similarity, with no dependency structure. It *loses* in five of six settings ($-3.1/-1.0/-1.0$ on ToolComp, $-6.0/-5.0/0.0$ on τ^2; Mean $63.6 \rightarrow 60.9$). Since every arm already sees the full domain catalog (Appendix B), the injected block adds no callable tool; it competes for attention, and a relevance ranking that ignores dependencies sometimes promotes a tool that cannot yet be called. Flat retrieval to TG–PPR then holds the interface, embedding, and k fixed and changes only whether dependency structure informs the ranking: it helps in five of six settings ($+1.0/+2.5/-2.6$, $+3.0/+4.5/+2.0$; Mean $60.9 \rightarrow 62.7$), so the structure is real and readable by a frozen planner — though it does not fully repay the interface, TG–PPR staying 0.9 points below Native on the Mean. Together these set the bar the value arms must clear: not whether graph evidence beats nothing, but whether it beats the same interface filled by relevance.

Composing edges into a value helps only when the estimator also reads the state. TG–PPR to the three estimators is the central contrast, and the estimators see no information the retrieval arm

lacks — only a different way of combining it. Q_{dp} splits its six cells against TG–PPR three–three ($+1.0/-2.5/+4.1$, $+1.0/-12.5/-7.4$) and loses 2.8 points on the Mean only because its two losses are far larger than its three wins; Q_{emb} wins one cell of six and loses 4.7 on the Mean. Q_{llm}, which sees the same graph evidence plus the concrete state, beats TG–PPR in *every* setting ($+7.2/+5.6/+8.5$ on ToolComp, $+16.0/+6.5/+6.0$ on τ^2; Mean $62.7 \rightarrow 71.0$) and is the best arm in every column and on the Mean, ahead of the strongest baseline, Reflexion, by 6.7 points.

Two effects are at work and should not be conflated. Argument blindness is *shared* by TG–PPR and the structural estimators — both rank tool types — so it cannot explain why composition loses to relevance; it explains the ceiling both hit, and why only the state-reading scorer clears it. What separates them is the relaxation's *bias*: a monotone reachability value rewards actions that unlock many downstream types, which is not relevance to this request, and multi-step propagation compounds that preference along a path rather than damping it. Composition therefore does not merely fail to add signal, it moves the ranking — so Q_{dp} earns its place as the scorer's prior and admissibility filter, not as a ranker, and Q_{emb} inherits the bias without the auditability. One caveat: cells are point estimates without intervals, so we do not order small differences.

4.3 Ablations and Analysis

Taking Q_{llm} as Q_{best} on the strength of Table 1, we vary the *evidence budget* on τ^2 along four knobs. Three control how much of the graph the scorer sees per decision: the neighbourhood radius h of the subgraph extracted around the candidate and goal tools, the edge-weight threshold θ below which edges are pruned before verbalization, and the per-node annotation depth — edges only (N0), plus required/produced fields (N1), structural features (N2), or runtime call history (N3). The fourth, n, is the number of candidate actions sampled per decision for the scorer to rank; $n{=}1$ leaves the verbalizer's feedback as the only graph signal. The first two knobs act indirectly, so we report the resulting node and edge counts.

Rather than cross the knobs, we run four one-factor sweeps through an *anchor* configuration — fourteen configurations instead of the $4 \times 6 \times 4 \times 3$ cross. The sweeps run on the evaluation split, so the anchor is a reported operating point, not one held out from these numbers. This answers what the scaling claim needs — which knob returns more success per token — but cannot detect interaction, and it characterizes each knob only near the anchor and only for Q_{llm}, since the structural estimators read the graph directly rather than through a prompt. Cost is request-time tokens per task, $\sum_i t_i$ over candidate generation, scoring and verbalization; the one-time construction term C, amortized as C/N over N reusing tasks, is kept separate in Table 4 since it shifts every ToolGraph arm equally. We keep it separate rather than declaring it amortized away because at our scale it is not: over the 100 evaluated τ^2 tasks C/N is 88K tokens, comparable to the anchor's own 155K of request-time cost. It falls as $1/N$, so the argument is about deployments serving many more requests than a test split, not about these cells.

Every knob buys evidence at a token price, so each block is read against its own anchor: the change in success per additional thousand tokens is what that knob returns, and a block in which success falls as the budget grows is one where the extra evidence buys dilution rather than dependencies. Three things hold of the macro average across all four blocks; per backbone the peaks move, so these are aggregate tendencies rather than per-model prescriptions. First, *every knob except θ is non-monotone, with its macro peak at or near the second level* — $h{=}1$, N1, $n{=}4$: additional structure first supplies missing dependencies and then crowds the prompt, and four to six levels expose a peak without locating it precisely. Second, *the peak is also the knee*, so the defensible operating choice is never the most expensive end: $h{=}2$ and $h{=}3$ cost exactly the same 155.2K as $h{=}1$ and lose 1.3 and 2.2 points, because the τ^2 graph is already almost covered at one hop ($24.1 \rightarrow 24.3$ nodes), so further radius buys no structure and only reorders the prompt; and $n{=}8$ spends 116.6K more than $n{=}4$ to lose 1.7 points. Third, *pruning to save tokens never pays*: raising θ to 0.9 removes over half the edges to save 9.5K tokens and costs 2.5 points, declining almost monotonically in between, so the weak edges carry signal rather than noise.

Ranked by marginal return, node annotation is the cheapest win — N0 to N1 buys 0.6 points for 3.3K tokens by telling the scorer *why* two tools connect, not just *that* they do — ahead of radius (1.4 for 18.3K) and candidate count (5.3 for 87.5K), the largest absolute effect. With no candidate diversity at all the verbalizer's feedback still reaches 61.1, nine tenths of the anchor's success at under half its budget.

Table 2: Evidence-budget sweeps for Q_{llm} on τ^2-Bench. Each block varies *one* knob and holds the rest at the anchor (†: h=1, θ=0.0, N0, n=4). Success is the macro average of task pass rate (%) over three planners and the three τ^2 domains; Tok. K is an *estimated* request-time cost in thousands of tokens per task (planner, scorer and verbalizer at ≈5.5 decisions/task), computed from measured prompt sizes and per-mechanism call counts rather than per-call accounting. Blocks 1–3 and block 4 are separate run series, so the nominally identical anchor reads 65.8 in the first three and 66.4 in the fourth — a 0.6-point gap *on the macro average*; the same replicate pair differs by up to 2.5 points within a single planner column, which is the scale at which per-backbone differences should be read. Sweeps are run on the evaluation split. Verifier: Gemma 4 31B; verbalizer enabled throughout.

	nodes/edges	Succ.	Tok. K		nodes/edges	Succ.	Tok. K
1 — radius h				**2 — edge threshold θ**			
h=0	7.7/7.0	64.4	136.9	θ=0.0 †	24.1/159.5	**65.8**	155.2
h=1 †	24.1/159.5	**65.8**	155.2	θ=0.5	24.1/149.4	65.3	153.9
h=2	24.3/160.3	64.5	155.2	θ=0.6	24.1/138.4	63.6	152.7
h=3	24.3/160.3	63.6	155.2	θ=0.7	24.1/131.4	63.0	152.0
				θ=0.8	24.1/125.0	63.6	151.3
				θ=0.9	24.1/ 73.7	63.3	145.7
3 — node annotation depth				**4 — sampled candidates n**			
N0 edges only †	24.1/159.5	65.8	155.2	n=1 verbalizer only	24.1/159.5	61.1	67.7
N1 +req./prod.	24.1/159.5	**66.4**	158.5	n=4 †	24.1/159.5	**66.4**	155.2
N2 +structural	24.1/159.5	64.2	161.1	n=8	24.1/159.5	64.7	271.8
N3 +runtime	24.1/159.5	63.3	163.7				

5 Limitations

ToolGraph supplies only structure its nodes and edges represent correctly, and two blind spots are structural rather than incidental: the delete-free relaxation ignores consumed resources, conditional effects, and negative interactions between actions [7], so it overvalues plans a non-monotone model would reject; and edges are catalog-level statements about tool pairs, so no graph-derived value separates two candidates calling the same tool with different arguments — the limit that decides our main result. Probe outcomes are adjudicated by a model against synthesized queries rather than gold answers, so an edge weight is evidence of a dependency and not a proof of one; probing is harder for stateful or irreversible APIs, and amortization assumes a stable catalog and enough reuse. Our benchmarks are public, so we test only the narrower claim that composing catalog dependencies into a value beats using the same graph for retrieval, and our retrieval pole is a same-graph ablation rather than the state of the art [16]. Holding one scorer model fixed across planner backbones lets the LLM-scored arm benefit from a scorer stronger than the planner it advises, a confound the structural estimators do not have. Finally, cells are single-seed point estimates without confidence intervals, the sweeps cannot detect interaction between knobs, we do not test Q_{emb} transfer to unseen catalogs, and we cannot release code, graph artifacts, or trajectories.

6 Conclusion

We asked whether a compiled tool-dependency graph, which prior work retrieves from, searches, or reads a single next call out of, can instead be composed into a multi-step goal-conditioned action value. The answer comes back split: over one fixed graph the LLM estimator beats same-graph personalized-PageRank retrieval in every one of six settings, by 8.3 points on the mean, while the auditable fixed-point solve and its distilled variant do not. The useful reading of a compiled dependency graph is therefore neither "retrieve from it" nor "rank by its value" but the composition of the two: the structural solve supplies a cheap, auditable prior and admissibility filter, and a scorer that can still see the arguments turns it into a decision. That also says where the next gain is — lifting the entity-set relaxation toward argument-level state, rather than enlarging the graph.

References

[1] Victor Barrès, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment. In *International Conference on Machine Learning (ICML)*, 2026. arXiv:2506.07982.

[2] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1–2): 5–33, 2001. doi: 10.1016/S0004-3702(01)00108-4.

[3] Wenjie Chen, Di Yao, Wenbin Li, Xuying Meng, Chang Gong, and Jingping Bi. GTool: Graph enhanced tool planning with large language model. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2508.12725.

[4] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[5] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. arXiv:1706.02216.

[6] Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. ToolkenGPT: Augmenting frozen language models with massive tools via tool embeddings. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[7] Jörg Hoffmann. Where "ignoring delete lists" works: Local search topology in planning benchmarks. *Journal of Artificial Intelligence Research*, 24:685–758, 2005. doi: 10.1613/jair.1747.

[8] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001. doi: 10.1613/jair.855.

[9] Joey Hong, Anca Dragan, and Sergey Levine. Planning without search: Refining frontier LLMs with offline goal-conditioned RL. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

[10] Jingyi Jia and Qinbin Li. AutoTool: Efficient tool selection for large language model agents. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, volume 40, pages 31265–31273, 2026. arXiv:2511.14650.

[11] Seungkyu Lee, Nalim Kim, and Yohan Jo. In-N-Out: A parameter-level API graph dataset for tool agents. *Transactions of the Association for Computational Linguistics*, 14:1008–1027, 2026.

[12] Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2305.20050.

[13] Shengjie Liu, Li Dong, and Zhenyu Zhang. Bridging tool dependencies and domain knowledge: A graph-based framework for in-context planning. In *NeurIPS 2025 Workshop on Bridging Language, Agent, and World Models for Reasoning and Planning (LAW)*, 2025. Non-archival workshop paper. arXiv:2510.24690.

[14] Ethan Mendes and Alan Ritter. Language models can self-improve at state-value estimation for better search. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. Spotlight. arXiv:2503.02878.

[15] Vaskar Nath, Pranav Raja, Claire Yoon, and Sean Hendryx. ToolComp: A multi-tool reasoning & process supervision benchmark. *arXiv preprint arXiv:2501.01290*, 2025. Spotlight, MTI-LLM Workshop at NeurIPS 2025.

[16] Bhrij Patel, Davide Belli, Amir Jalalirad, Maximilian Arnold, Aleksandr Ermolov, and Bence Major. Dynamic tool dependency retrieval for lightweight function calling. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 33649–33672. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.1680.

[17] 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. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2307.16789.

[18] Kusha Sareen, Morgane M Moss, Alessandro Sordoni, Rishabh Agarwal, and Arian Hosseini. Putting the value back in RL: Better test-time scaling by unifying LLM reasoners with verifiers. In *Conference on Language Modeling (COLM)*, 2025. arXiv:2505.04842.

[19] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. arXiv:2303.11366.

[20] Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2408.03314.

[21] Xixi Wu, Yifei Shen, Caihua Shan, Kaitao Song, Siwei Wang, Bohang Zhang, Jiarui Feng, Hong Cheng, Wei Chen, Yun Xiong, and Dongsheng Li. Can graph learning improve planning in LLM-based agents? In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. arXiv:2405.19119.

[22] Yu Xia, Jingru Fan, Weize Chen, Siyu Yan, Xin Cong, Zhong Zhang, Yaxi Lu, Yankai Lin, Zhiyuan Liu, and Maosong Sun. AgentRM: Enhancing agent generalization with reward modeling. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 19277–19290. Association for Computational Linguistics, 2025.

[23] Shuo Yang, Soyeon Caren Han, Yihao Ding, Shuhe Wang, and Eduard Hovy. ToolTree: Efficient LLM tool planning via dual-feedback monte carlo tree search and bidirectional pruning. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2603.12740.

[24] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik R. Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

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

[26] Lunjun Zhang, Arian Hosseini, Hritik Bansal, Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. Generative verifiers: Reward modeling as next-token prediction. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2408.15240.

[27] Yinger Zhang, Hui Cai, Xierui Song, Yicheng Chen, Rui Sun, and Jing Zheng. Reverse chain: A generic-rule for LLMs to master multi-API planning. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 302–325. Association for Computational Linguistics, 2024.

[28] Zijing Zhang, Zhanpeng Chen, He Zhu, Ziyang Chen, Nan Du, and Xiaolong Li. ToolExpNet: Optimizing multi-tool selection in LLMs with similarity and dependency-aware experience networks. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 15706–15722. Association for Computational Linguistics, 2025.

[29] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *International Conference on Machine Learning (ICML)*, volume 235 of *Proceedings of Machine Learning Research*, pages 62138–62160, 2024.

[30] Yifei Zhou, Andrea Zanette, Jiayi Pan, Sergey Levine, and Aviral Kumar. ArCHer: Training language model agents via hierarchical multi-turn RL. In *International Conference on Machine Learning (ICML)*, volume 235 of *Proceedings of Machine Learning Research*, pages 62178–62209, 2024. arXiv:2402.19446.

A ToolGraph Construction

This appendix summarizes how the graphs used in our experiments were built. It is included for self-containment; the pipeline is not a contribution of this paper, and we describe it only to the depth needed to make the Q-estimators of Section 3.3 well defined.

Nodes. Every tool is normalized into a common record regardless of its source package: identifier, a coarse class distinguishing *atomic* tools with declared, well-typed interfaces from *meta* tools whose output semantics are not declared and must be established empirically, a domain grouping, a textual statement of what the tool does, and the names, types, and descriptions of its input and output fields. A node embedding is formed from the concatenated intent and output text and is reused as the input feature for Q_{emb}.

Edge proposal. For an ordered pair (a, b) we ask whether a should precede b, whether b should precede a, or whether the two are independent or substitutable and no edge should exist. Two proposers run independently. *Parameter matching* compares Out_a against the required fields of In_b and proposes an edge on a match; it is nearly free but is restricted to tools that declare usable types, so it does not reach meta tools. *Interventional probing* treats each tool as a black-box service. A language model derives a candidate dependency from one tool's interface — what information must be obtained before it can be called, or what task naturally follows from its output — checks whether the other tool could plausibly supply that piece, and synthesizes a probe query if so. The probe query is then executed under the composition and under a baseline that omits the proposed predecessor, and an evaluator compares how much of the query each outcome resolves. The two proposers overlap: many pairs are proposed by probing alone, which is why probing rather than parameter matching is the primary source of edges (Table 3).

Edge confidence and pruning. Rather than a binary decision, each edge carries a confidence in $[0, 1]$. The capability-match score acts as a prior, the probe comparison is converted into a likelihood, and the two are combined per piece of evidence; w_{ab} is the mean over the retained evidence components for that pair. An edge is instantiated only when $w_{ab} > \theta_{\mathrm{prune}}$. The threshold is set per benchmark and its values are given in Table 3; the sparser the candidate set, the lower the threshold needed to retain a usable graph. The graph is built once per benchmark, frozen, and reused by every task, backbone, and ToolGraph arm in our experiments; we perform no runtime updates to edge weights.

Provenance. Each edge stores the probe query, the chain and baseline signals, the resulting confidence components, and a graph version hash, so that a value produced by Q_{dp} can be traced back to the specific evidence supporting each edge on the path. This is what we mean by calling the structural estimator auditable.

Separation from the evaluation data. Probe queries are synthesized from tool interfaces alone. No benchmark task text, reference trajectory, reward signal, or environment feedback from the evaluation split is read at any point during construction, so edge weights cannot encode information about the tasks the graph is later evaluated on. This matters because a single graph per benchmark is reused across every task and backbone: had construction observed test tasks, that leakage would inflate every ToolGraph arm simultaneously.

B Implementation Details

Graph construction. The pipeline is described in Appendix A and the resulting graphs are characterized in Table 3. What matters for the estimators is how per-edge confidences become per-action quantities. Both p_a and c_a are aggregations over the weights of the edges *incoming* to a: p_a is their

Table 3: ToolGraph statistics per benchmark. "Param. only" / "probe only" / "both" partition the retained edges by which proposer supplied their evidence. The lower block gives the three τ^2 domains that are evaluated separately, counting only intra-domain nodes and edges; the 73 domain nodes are a subset of the 146 in the catalog graph, the remainder belonging to τ^2 domains we do not evaluate. In that block the last column is *not* θ_{prune} but $k/|\mathcal{T}|$, the fraction of the domain catalog exposed at $k = 8$, which shows where retrieval has little room to act.

	Nodes		Edges		Evidence source			
Benchmark	atomic	meta	cand.	kept	param. only	probe only	both	θ_{prune}
ToolComp	7	4	40	35	0	28	7	0.10
τ^2-Bench	102	44	1508	1303	59	1041	203	0.30
Evaluated τ^2 domains (intra-domain edges; $k/\|\mathcal{T}\|$ at $k = 8$ in the last column)								
airline	14		79					0.57
retail	16		93					0.50
telecom	43		309					0.19

Table 4: Graph construction and maintenance cost, in language-model calls and estimated tokens, broken out by construction phase. Construction is a one-time, catalog-level cost; the amortized rows divide it by the number of evaluated tasks that reuse the graph. On the larger catalog, adding a tool later costs about two orders of magnitude less than a rebuild and deleting one is near-free, which is what makes the compiled artifact reusable under catalog churn; on a catalog as small as ToolComp's eleven tools the saving is correspondingly smaller. Token counts are estimates, derived from measured prompt sizes and the max-token setting of each phase rather than from per-call accounting. Construction uses a fixed closed model (Sonnet 4.5 for edge proposal and probe synthesis, Sonnet 4 for probe evaluation), independent of the planner and verifier models used at request time.

	ToolComp	τ^2-Bench
Full construction (one-time)		
Phase 2: schema edge evaluation	21	1,266
Phase 3.1: blind task derivation	7	102
Phase 3.2: capability matching	28	4,488
Phase 3.3: information gain	65	2,400
Total LLM calls	**121**	**8,256**
Input tokens	83K	6.0M
Output tokens	37K	2.8M
Total tokens	120K	8.8M
Incremental update		
LLM calls, add one tool	14	84
Total tokens	14K	89K
LLM calls, delete one tool	7	7
Evaluated tasks reusing the graph	214	100
Amortized construction calls/task	0.6	83
Amortized construction tokens/task	0.6K	88K

maximum and c_a their mean. They are therefore two summaries of one evidence array rather than two independent measurements, which is why Eq. (2) functions as a dispersion penalty; we state this explicitly because it bounds how much additional discrimination the penalty can provide. A tool with no incoming edges receives the neutral value 0.5 for both. At decision time the retriever constructs a pool of at most 16 feasible or goal-relevant actions.

Q estimators. For Q_{dp}, we use $\gamma = 0.95$ and $\kappa = 0.3$, with a convergence tolerance of 10^{-6}, a cap of 10^4 value-iteration sweeps, and a cap of 10^5 enumerated entity-set states; the solver raises rather than silently truncating if the cap is reached. Terminal states take value $+1$ and dead ends -1. Solutions are cached per decision state and re-solved only when a candidate's successor state falls outside the enumerated lattice. Because the solver raises rather than truncating, every reported

cell is one in which the cap was not reached. Q_{emb} uses two mean-aggregation GraphSAGE layers with hidden dimension 256, 0.1 dropout, and a three-layer regression head. Node input combines a 1024-dimensional schema embedding with eight normalized structural features. We train for 200 epochs with Adam, learning rate 10^{-3}, weight decay 10^{-5}, batch size 64, and mean-squared error against normalized Q_{dp} targets. Training states and goal tools are enumerated from development-split states only, so that no test-split state contributes a distillation target.

For Q_{llm}, each candidate prompt contains the current request, conversation and tool history, candidate arguments, inferred goals, policy text, feasible alternatives, graph paths, edge evidence, and Q_{dp} prior. The scorer returns JSON with an integer in $[0, 10]$ and a short rationale. We average five samples at temperature 1.0 and normalize by 10. The same verifier model and prompt template are used for all planner backbones.

Agent and baseline configuration. The ToolGraph methods place the top-k actions, scores, goals, and verbalized evidence in the planner prompt; the planner may ignore this advice. Every arm, including the ToolGraph and flat-retrieval arms, exposes the full domain tool catalog to the planner and performs no retrieval narrowing of the callable action space, so differences between arms cannot be attributed to candidate-set pruning. The top-k block is therefore *advice within* the prompt, not a restriction of it — which is why the flat-retrieval arm can score *below* Native (Section 4.2): it adds a block of text without removing any option. Native and ReAct receive the same catalog and policy but no graph features. **Reflexion** here is the single-revision variant: it receives one feedback-and-revision opportunity, so at a single trial its post-episode mechanism is close to inert. Its self-reflection is written by the same verifier model, with the same revision budget, as the ToolGraph arms' verbalizer, so the Reflexion-versus-ToolGraph difference isolates graph-grounded feedback from ungrounded self-reflection. Candidate temperature is 0.7 for all sampling-based methods, except where a backbone's API rejects the parameter, in which case the provider default applies.

Evaluation. A single seed controls planner sampling, tie breaking, and environment randomness; task IDs, prompts, model versions, graph hashes, and generated candidates are cached so that estimator comparisons use paired inputs. Success is the benchmark-native binary reward, and the reported cells are point estimates without confidence intervals: with one seed per cell we have no within-cell variance to report, and the run-to-run spread of a single configuration in Table 2 is the only empirical handle on noise we can offer. Token counts include scoring and verbalization prompts, and graph cost is reported both in total and divided by the number of evaluated tasks to show the effect of reuse. We do not report per-decision latency or probe coverage.