GRAPHPTC: GUIDING PROGRAMMATIC TOOL CALLING WITH DYNAMIC EXECUTION GRAPHS

Anonymous authors
Paper under double-blind review

ABSTRACT

Programmatic Tool Calling (PTC) allows large language models to compose multiple tool calls with control flow and intermediate data processing within a single program block executed in a persistent runtime. Although PTC reduces model-tool interaction rounds, execution across blocks is represented as a linear history of code, standard output, and errors, leaving execution effects and dependencies implicit. Across seven benchmark settings and five backbone models, PTC underperforms Direct Tool Calling in 14 of 35 comparisons, by as much as 34.5 percentage points, indicating that programmatic execution alone does not consistently improve performance. We introduce Graph-Structured Programmatic Tool Calling (GRAPHPTC), a framework that augments PTC with a task-level dynamic execution graph incrementally constructed from runtime traces. The typed graph captures tool effects, artifacts, states, and failures within blocks and explicitly links dependencies across blocks. It links declared intents to observed effects and provides structured feedback for continuing execution, patching failed steps, replanning dependency paths, or producing an answer. Across all evaluated benchmark–backbone combinations, GRAPHPTC outperforms both baselines, with average gains of 6.8 and 6.3 percentage points over Direct Tool Calling and PTC, respectively. In the efficiency evaluations, it uses 1.7–$2.3\times$ fewer input tokens than Direct Tool Calling. [1]

1 INTRODUCTION

Tool calling enables large language models (LLMs) to interact with external environments. Conventional agents interleave reasoning with individual tool calls, using each result to select the next action (Yao et al., 2023; Schick et al., 2023; Qin et al., 2024). *Programmatic Tool Calling* (PTC) instead combines tool calls, loops, conditionals, and intermediate data processing in programs executed within a persistent runtime (Gao et al., 2023; Wang et al., 2024; Roucher et al., 2025; Anthropic, 2025). This reduces model-tool interaction rounds and supports both control flow within blocks and variable reuse across blocks. However, the programmatic interface does not provide an equally structured representation of cross-block execution.

PTC conditions subsequent programs on a chronological history of code, standard output, and errors. This history provides a limited view of execution effects and dependencies. On AppWorld (Trivedi et al., 2024), a block can summarize multiple calls and their read or write effects in one output string, while demanding tasks span more than fifteen such blocks. The agent must infer from this text whether writes succeeded, which earlier values later steps depend on, and whether differently phrased queries returned equivalent results. In our evaluation across seven settings and five backbones, PTC underperforms Direct Tool Calling in 14 comparisons, with the largest gap of 34.5 percentage points for Claude-Opus-5 on AppWorld-Normal. Programmatic execution alone therefore does not consistently improve performance.

Execution establishes relations: calls produce artifacts, later calls consume them, and writes supersede earlier state versions. Leaving these relations implicit creates three connected limitations. At the representation layer, a *linear execution context* requires repeated reconstruction of prior results and their dependencies. At the analysis layer, *implicit execution progress* disconnects intended outcomes from observed effects: a block may complete without error while retrieving an existing result

[1] The source code is available in the supplementary materials.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

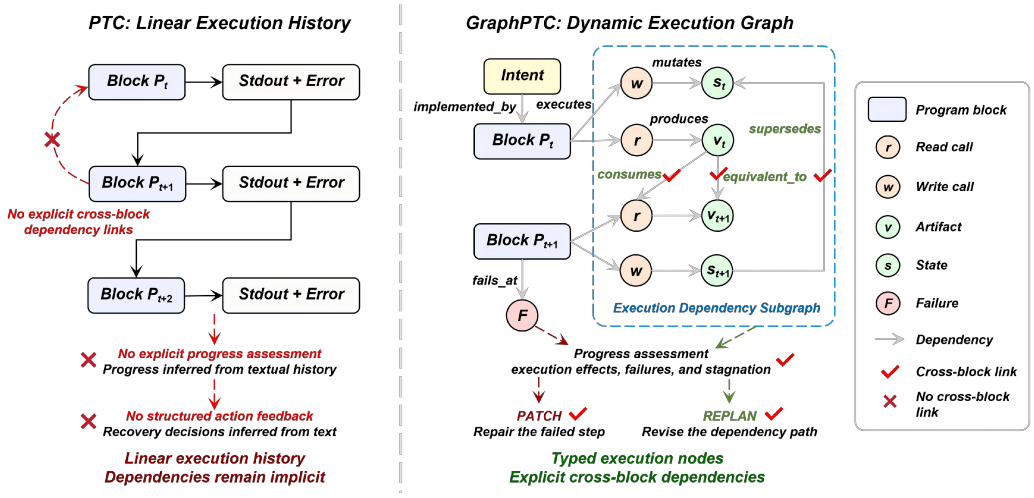


Figure 1: Comparison of execution representations in PTC and GRAPHPTC. PTC maintains a linear history of program blocks and outputs, leaving execution effects and cross-block dependencies implicit. GRAPHPTC represents tool effects, artifacts, state versions, and failures with typed nodes and edges, while linking declared intents to execution. Cross-block relations make artifact consumption, result equivalence, and state evolution explicit, supporting progress assessment and targeted repair or replanning.

and making no additional progress. At the decision layer, *unstructured decision feedback* leaves the agent to infer from scattered history whether to proceed, repair a program, or change strategy, potentially leading to repeated attempts or unproductive execution.

To address these limitations, we introduce Graph-Structured Programmatic Tool Calling (GRAPHPTC), a framework that augments PTC with a task-level dynamic execution graph incrementally constructed from runtime traces. As shown in Figure 1, each program block is represented as a typed subgraph of tool calls, read/write effects, artifacts, state versions, and localized failures, while cross-block edges link equivalent results, reused artifacts, and superseded states. The graph supports progress assessment by aligning declared intents with observed execution changes and distinguishing new results from repeated ones. It further supports action selection by providing a bounded summary of graph evidence to guide continuation, repair, replanning, or answering. Together, these mechanisms enable the agent to track completed work, identify unresolved dependencies, and decide subsequent actions from explicit execution evidence.

Our contributions are threefold.

- **Empirical and Structural Analysis.** We demonstrate the inconsistent benefits of PTC and identify its implicit representation of execution effects and dependencies as a structural limitation.
- **Graph-Guided Tool Calling.** We introduce GRAPHPTC, which represents execution within and across program blocks as a task-level dynamic execution graph. By linking declared intents to observed effects, it enables progress assessment and provides bounded feedback for continuation, targeted repair, replanning, and answering.
- **Performance and Efficiency.** GRAPHPTC outperforms both baselines across seven settings and five backbones, with lower input token usage and generally fewer interaction rounds than Direct Tool Calling in the efficiency comparisons.

2 RELATED WORK

Tool Calling and Programmatic Execution. LLMs interact with external tools through individual calls or executable programs. Early work explores tool invocation and its integration with model reasoning, including Toolformer (Schick et al., 2023), ReAct (Yao et al., 2023), ToolLLM (Qin et al., 2024), and Gorilla (Patil et al., 2024). Multi-agent frameworks such as AutoGen (Wu et al., 2023) and MetaGPT (Hong et al., 2024) further investigate coordination among agents. A related line of

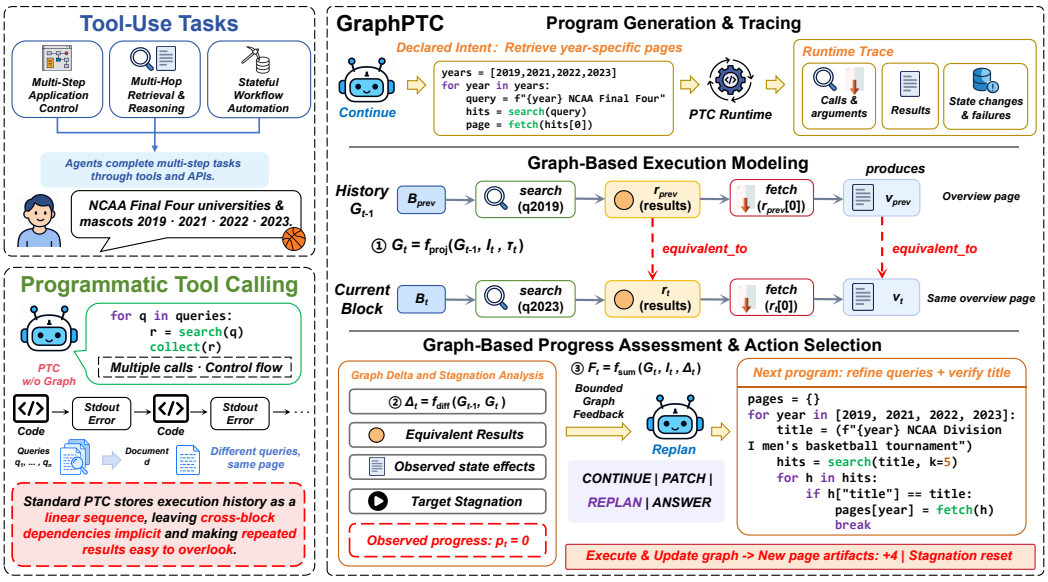


Figure 2: Overview of GRAPHPTC. The agent declares an intent and generates a program, whose execution produces a runtime trace. The trace is projected into a dynamic execution graph, and graph changes provide evidence for progress assessment and subsequent action selection through bounded feedback. In the illustrated retrieval example, cross-block equivalence links expose repeated retrieval of the same overview page. The agent then replans by refining the queries and checking result titles to retrieve year-specific pages.

work represents computation and actions as code, ranging from program-aided reasoning (Gao et al., 2023; Chen et al., 2023) to code-as-action agents that generate executable programs to invoke tools in an interpreter, such as CodeAct (Wang et al., 2024) and smolagents (Roucher et al., 2025).

Enhancement of Tool Calling. Related approaches improve tool use by enriching agent context or changing how actions are executed. In the first direction, Reflection and self-refinement methods convert prior outcomes or self-generated feedback into natural-language guidance (Shinn et al., 2023; Madaan et al., 2023), while memory systems and memory-centric agent architectures organize and retain information for subsequent use (Packer et al., 2023; Rasmussen et al., 2025; Xu et al., 2025; Sumers et al., 2024). Related provenance models in data management explicitly record how results are derived (Davidson & Freire, 2008; Herschel et al., 2017). The second direction reduces the overhead of per-call interactions by composing multiple tool calls within a single executable program. Programmatic tool calling follows this approach by executing model-generated programs that interact with multiple tools in a shared runtime (Anthropic, 2025).

GRAPHPTC retains the programmatic interface of PTC while linking declared intents to tool effects, artifact dependencies, state versions, and failures in a shared typed graph. This intent–execution connection supports the detection of repeated results and target-specific stagnation, turning execution records into bounded feedback for continuation, repair, or replanning.

3 GRAPHPTC

GRAPHPTC augments PTC with a task-level dynamic execution graph built from runtime traces. Its three stages address the limitations in Section 1.

At each nonterminal round t, the agent generates a program P_t and intent I_t. Before execution, the runtime registers I_t in $\mathcal{G}_{t-1}$, yielding $\widetilde{\mathcal{G}}_{t-1}$. Executing P_t with instrumented tools then produces a trace τ_t. The runtime updates the accumulated graph $\mathcal{G}_{t-1}$, extracts the changes, and constructs bounded feedback:

$$\mathcal{G}_t = f_{\text{proj}}(\widetilde{\mathcal{G}}_{t-1}, I_t, \tau_t), \qquad \Delta_t = f_{\text{diff}}(\mathcal{G}_{t-1}, \mathcal{G}_t), \qquad F_t = f^B_{\text{sum}}(\mathcal{G}_t, I_t, \Delta_t), \tag{1}$$

Here, f_{proj} projects the execution trace and updates the registered intent, while f_{diff} extracts the execution effects. The resulting Δ_t is then augmented with the updated target stagnation count, as described in Section 3.2, before f^B_{sum} constructs feedback within a character budget B. After execution, the runtime appends the assistant response and the program output or execution error, subject to the output limit, to the interaction history $\mathcal{H}_t$. The backbone LLM π then generates $R_{t+1} = \pi(q, \mathcal{H}_t, F_t)$, which either provides a final answer or specifies the next intent I_{t+1} and program P_{t+1}. Here, $\mathcal{H}_t$ contains the task instruction and prior interaction context; underlying tool results are not automatically included in full.

3.1 Graph-Based Execution Modeling

The typed execution graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ unifies result provenance (Herschel et al., 2017), cross-block artifact consumption, and state evolution. Nodes in $\mathcal{V}$ carry type labels and attributes, representing the task, intents, blocks, calls, artifacts, state versions, and failures. Each edge in $\mathcal{E}$ specifies a source node, a relation type, and a target node, encoding execution, data-flow, or state relations (Appendix Table 4).

Typed Trace Projection. Tool wrappers record call arguments, results, and effects, including calls executed within loops and conditionals. The projector f_{proj} links calls to the block node b_t corresponding to program P_t and records outputs, state accesses, and failures, including the edges

$$\{\, b_t \xrightarrow{\text{executes}} c \mid c \in \mathcal{C}_t \,\} \cup \{\, c(v) \xrightarrow{\text{produces}} v \mid v \in \mathcal{A}_t \,\} \cup \{\, u_\phi \xrightarrow{\text{fails_at}} \phi \mid \phi \in \Phi_t \,\}, \tag{2}$$

where $\mathcal{C}_t$, $\mathcal{A}_t$, and Φ_t denote the calls, produced artifacts, and observed failures in the current block, respectively, and $c(v)$ denotes the call that produced artifact v. The source node u_ϕ is the failing call if identifiable, and the block b_t otherwise. Each call carries a typed effect $\varepsilon(c) \in \{\text{read}, \text{write}\}$, together with success and state-change flags.

Cross-Block Dependency Linking. Using tool effect contracts, f_{proj} assigns each artifact a normalized identity key $\kappa(\cdot)$ derived from the relevant call arguments and returned content.

$$v \xrightarrow{\text{equivalent_to}} v' \quad \text{when} \quad \kappa(v) = \kappa(v'), \tag{3}$$

where v is the earlier artifact and v' the new one. This identifies equivalent results from differently phrased queries, but applies only to read-effect calls designated as deterministic by the tool contract; write-effect calls and their state-change records are excluded from equivalence detection. Existing artifacts connect through consumes when used as inputs or reuses when directly reused. Successive state versions connect through supersedes, capturing dependencies on earlier writes alongside artifact dependencies. The accumulated graph $\mathcal{G}_t$ retains these links and the current block with its calls, artifacts, states, and failures.

3.2 Graph-Based Progress Assessment

This stage links declared intents to observed effects to distinguish new results, repeated results, and target-specific stagnation.

Intent Registration and Binding. When submitting a program, the agent declares an intent

$$I_t = (a_t, z_t, e_t), \tag{4}$$

where a_t is the action type, z_t is an existing graph node targeted by the action, and e_t is the expected change. For example, a PATCH action can target a failed call. When an action advances the task without a more specific graph target, z_t is the task node. Before execution, the runtime registers an intent node, connects it to z_t through a targets edge, and records a snapshot of the current artifacts and states. After execution, f_{proj} projects the trace, links the registered intent to the executed block through an implemented_by edge, and attaches an action receipt. The receipt summarizes the execution outcome and the action realization flag, linking them to the declared intent.

Graph Delta and Stagnation Analysis. Among the produced artifacts $\mathcal{A}_t$, f_{diff} assigns those with incoming equivalent_to edges to $\mathcal{E}_t$. New artifacts and progress are defined as

$$\mathcal{N}_t = \mathcal{A}_t \setminus \mathcal{E}_t, \qquad p_t = \mathbb{1}[\mathcal{N}_t \neq \varnothing \ \vee \ \mathcal{S}_t \neq \varnothing\,], \tag{5}$$

Algorithm 1 GRAPHPTC performs graph-guided programmatic tool calling.

Require: task q, tools, execution budget, feedback character budget B
initialize graph $\mathcal{G} \leftarrow (\{v_{\text{task}}\}, \varnothing)$
initialize history $\mathcal{H} \leftarrow (q)$ and empty feedback F
initialize target counters s with default value zero
while execution budget remains **do**
  $R \leftarrow \pi(q, \mathcal{H}, F)$
  if R provides a final answer without a tool call **then**
    return the final answer in R
  end if
  extract intent $I = (a, z, e)$ and program P from R
  snapshot $\mathcal{G}^- \leftarrow \mathcal{G}$
  register I in $\mathcal{G}$ and link it to target z
  $(\tau, o) \leftarrow$ execute P with instrumented tools
  $\mathcal{G} \leftarrow f_{\text{proj}}(\mathcal{G}, I, \tau)$ {update the registered intent and project the trace, Sec. 3.1}
  $\Delta \leftarrow f_{\text{diff}}(\mathcal{G}^-, \mathcal{G})$
  compute p using Eq. 5 and update $s(z)$ using Eq. 6 {progress assessment, Sec. 3.2}
  augment Δ with the updated target counter $s(z)$
  $F \leftarrow f^B_{\text{sum}}(\mathcal{G}, I, \Delta)$, incorporating $\mathcal{O} = f_{\text{opp}}(\Delta)$ {bounded feedback, Sec. 3.3}
  append R and o to $\mathcal{H}$, excluding graph feedback
end while

where $\mathcal{S}_t$ contains state changes observed during the current round, rather than attempted write operations. The indicator p_t records new artifacts or state effects. Action realization is assessed using action-specific execution criteria, as detailed in Appendix B. Equivalent-result nodes and cache reuse alone do not count as progress. Stagnation is tracked separately for each target:

$$s_t(z) = \begin{cases} 0, & z = z_t \text{ and } p_t = 1, \\ s_{t-1}(z) + 1, & z = z_t \text{ and } p_t = 0, \\ s_{t-1}(z), & z \neq z_t. \end{cases} \tag{6}$$

Each target counter is initialized to zero when the target first becomes available. The count detects the absence of new effects even without execution errors and enables threshold-based replanning suggestions, while leaving the next action to the agent. The updated stagnation count for the current target is included in Δ_t to inform action suggestions and feedback construction.

3.3 GRAPH-BASED ACTION SELECTION

This stage presents bounded graph evidence and action opportunities for the agent to select its next action.

Bounded Feedback Construction. Given the accumulated graph, the intent, and the graph delta, f_{sum} extracts the declared action, its receipt, new and equivalent artifacts, state changes, failure context, and prior attempts associated with the target. It prunes list items, then fields, until $|F_t| \leq B$, where $|F_t|$ denotes the number of characters in the serialized graph feedback. The full graph remains in the runtime, and the program output is subject to a separate length limit.

Graph-Guided Action Selection. Recovery is formulated as an operation with an explicit graph target. The runtime maps execution signals to an opportunity set $\mathcal{O}_t = f_{\text{opp}}(\Delta_t)$: failures suggest targeted repair, and stagnation reaching the configured threshold suggests replanning. The agent selects an action

$$a_{t+1} \in \{\text{CONTINUE}, \text{PATCH}, \text{REPLAN}, \text{ANSWER}\}, \tag{7}$$

where CONTINUE advances execution to a new step, PATCH targets a failed node for repair and re-execution, REPLAN changes the dependency path, and ANSWER terminates execution. The first three actions supply a new intent and program to Eq. 1, tying action labels to executable behavior. An ANSWER response provides the final answer without issuing another programmatic tool call. Algorithm 1 summarizes the loop. Here, o denotes the program output or execution error after applying the output limit.

Table 1: Main results across seven settings and five backbones. Metrics are answer accuracy for BrowseComp-Plus, TGC for AppWorld, pass rate for Agent-Diff and AutomationBench, and judge accuracy for FanOutQA and FRAMES. **Bold** marks the best result per backbone and setting.

Method	BrowseComp-Plus (Acc)	AppWorld-Normal (TGC)	AppWorld-Challenge (TGC)	Agent-Diff (Pass)	FanOutQA (Acc)	FRAMES (Acc)	Automation-Bench (Pass)
			GPT-5.6-Sol				
Direct Tool Calling	73.49	82.10	81.80	51.34	55.81	77.31	20.83
PTC	80.00	89.90	90.60	39.73	52.58	73.42	29.17
GRAPHPTC	**80.60**	**91.70**	**93.00**	**51.49**	**60.00**	**79.00**	**31.00**
			Claude-Opus-5				
Direct Tool Calling	43.98	94.00	80.80	75.14	77.74	84.95	28.67
PTC	10.96	59.50	79.10	68.06	76.77	85.19	32.50
GRAPHPTC	**49.04**	**95.80**	**95.00**	**79.17**	**78.71**	**85.56**	**33.50**
			DeepSeek-V4-Pro				
Direct Tool Calling	4.58	81.00	65.70	70.09	57.42	81.43	21.50
PTC	3.49	88.70	80.10	72.47	58.06	80.83	17.00
GRAPHPTC	**13.37**	**89.90**	**81.50**	**73.96**	**62.26**	**81.55**	**22.83**
			Kimi-K3				
Direct Tool Calling	8.92	72.00	46.50	77.53	49.03	73.54	27.50
PTC	2.89	85.70	83.20	78.12	28.71	75.85	28.00
GRAPHPTC	**10.48**	**86.30**	**84.90**	**78.72**	**50.65**	**76.46**	**28.83**
			Qwen3.8-Max				
Direct Tool Calling	34.46	82.70	71.70	74.26	65.81	79.98	31.67
PTC	39.76	95.20	85.40	78.12	59.68	80.10	32.38
GRAPHPTC	**41.93**	**97.00**	**87.50**	**80.06**	**66.77**	**82.40**	**42.41**

4 EXPERIMENTS

4.1 BENCHMARKS AND SETUP

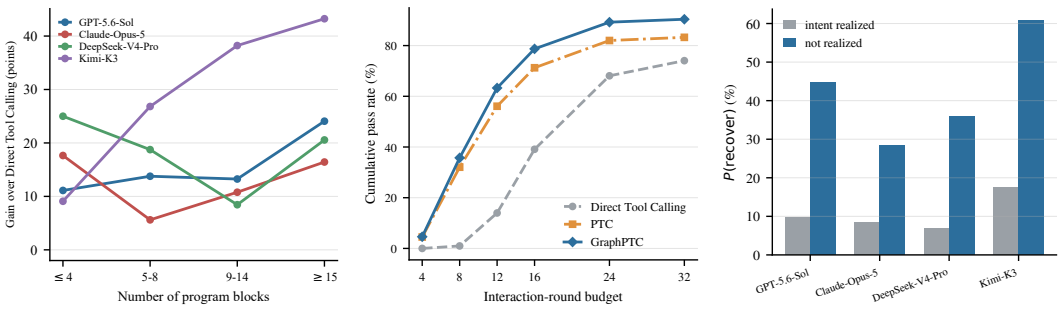
We evaluate GRAPHPTC on seven tool-use benchmark settings spanning retrieval, question answering, application control, and workflow automation. These benchmarks test complementary aspects of cross-block dependencies across different task horizons. Each provides execution-based, rule-based, or judge-based evaluation against reference outcomes. **BrowseComp-Plus** (Chen et al., 2025) evaluates evidence-seeking browsing over a fixed corpus using answer accuracy. **AppWorld** (Trivedi et al., 2024) provides an executable environment of everyday applications. We use its normal and challenge splits, evaluated using Task Goal Completion (TGC) and Scenario Goal Completion (SGC). **Agent-Diff** (Pyskło et al., 2026) evaluates stateful, multi-step tasks using an assertion-based pass rate. **FanOutQA** (Zhu et al., 2024) evaluates multi-hop, multi-document question answering using judge-based accuracy. **FRAMES** (Krishna et al., 2025) evaluates fact retrieval and reasoning using judge-based accuracy. **AutomationBench** (Shepard & Salimans, 2026) reports end-to-end pass rates for business workflows. The two AppWorld splits bring the total to seven settings, each evaluated with all five backbones.

We compare *Direct Tool Calling*, which uses native per-call function calling; *PTC*, which uses a persistent programmatic runtime without a graph; and GRAPHPTC, which adds the graph mechanisms in Section 3. All methods share the same task instructions, tools, and evaluation budgets within each benchmark. The action opportunity rules and feedback budget of GRAPHPTC remain fixed across settings and backbones. We use five backbone models: GPT-5.6-Sol, Claude-Opus-5, DeepSeek-V4-Pro, Kimi-K3, and Qwen3.8-Max, covering closed and open models with dense and mixture-of-experts architectures.

4.2 MAIN RESULTS

Table 1 shows that GRAPHPTC outperforms both baselines in every evaluated benchmark–backbone combination, with average gains of 6.3 and 6.8 percentage points over PTC and Direct Tool Calling, respectively.

Capability in Multi-Step Application Control. The AppWorld normal and challenge splits assess *multi-step application control*, requiring application calls whose write effects persist and collectively satisfy a final goal. The average gain over Direct Tool Calling increases from 9.8 to 19.1 percentage points between the normal and challenge splits. This pattern is consistent with the intended



(a) Gain by execution length (b) Pass rate vs. round budget (c) Recovery by intent realization

Figure 3: Analysis of GRAPHPTC on AppWorld across four backbones. (a) Gains over Direct Tool Calling by the number of program blocks in GRAPHPTC executions. (b) Cumulative task pass rate at each interaction-round budget, averaged equally across the four backbones. (c) Probability of selecting PATCH or REPLAN, conditioned on whether the preceding intent was realized.

role of explicit state modeling: typed state nodes connect later blocks to earlier writes, making these dependencies directly accessible. Section 5 further examines this mechanism.

Capability in Multi-Hop Retrieval and Reasoning. BrowseComp-Plus, FanOutQA, and FRAMES assess *multi-hop retrieval and reasoning*, requiring evidence from multiple tool calls. For Claude-Opus-5 on BrowseComp-Plus, GRAPHPTC raises accuracy from 10.96% with PTC to 49.04%, exceeding the Direct Tool Calling accuracy of 43.98%. Cross-block dependency links make earlier evidence accessible and identify equivalent results, helping the agent recognize redundant retrieval attempts.

Capability in Stateful Workflow Automation. Agent-Diff and AutomationBench assess *stateful workflow automation*, requiring dependent write operations to produce the intended state changes. On AutomationBench with DeepSeek-V4-Pro, PTC trails Direct Tool Calling by 4.5 percentage points, whereas GRAPHPTC exceeds it by 1.3 points. Progress assessment distinguishes state changes from repeated operations that leave the state unchanged, helping the agent identify completed steps and select subsequent actions.

5 ANALYSIS

We analyze where the performance gains of GRAPHPTC arise and examine how the execution graph relates to tool-calling behavior.

5.1 ABLATION OF THE EXECUTION GRAPH AND PROGRAMMATIC RUNTIME

Table 2 compares GRAPHPTC with two progressively simplified configurations, averaged across five backbones for each setting. Removing the execution graph together with its feedback and recovery mechanisms yields PTC, which returns success flags and raw outputs without typed execution effects; further removing the programmatic runtime yields Direct Tool Calling. The graph and its associated mechanisms jointly improve performance over PTC by 6.3 percentage points on average, with gains across all seven settings. By contrast, PTC improves over Direct Tool Calling by only 0.5 percentage points on average and underperforms it in four settings.

GRAPHPTC improves over PTC by 11.7 and 8.3 percentage points on BrowseComp-Plus and AppWorld-Normal, respectively, consistent with greater benefits from cross-block dependency tracking when later blocks reuse earlier results. The gain is smaller on FRAMES (1.9 points), where many questions require only a few independent lookups. The largest deficits of PTC relative to Direct Tool Calling occur for Claude-Opus-5 on BrowseComp-Plus and AppWorld-Normal, at 33.0 and 34.5 percentage points, respectively. Averaged across the seven settings, PTC reduces performance by 10.5 points for this backbone, whereas the other four backbones show average gains.

Table 2: Graph and runtime ablation, averaged across five backbones for each setting. The first three rows report scores (%); the final two report absolute differences in percentage points between successive configurations. Removing the execution graph also removes its associated feedback and recovery mechanisms.

Configuration	BrowseComp-Plus	AppWorld-Normal	AppWorld-Challenge	Agent-Diff	FanOutQA	FRAMES	Automation-Bench	Avg
GRAPHPTC (full)	**39.1**	**92.1**	**88.4**	**72.7**	**63.7**	**81.0**	**31.7**	**67.0**
− execution graph	27.4	83.8	83.7	67.3	55.2	79.1	27.8	60.6
− execution graph and programmatic runtime	33.1	82.4	69.3	69.7	61.2	79.4	26.0	60.2
Gain of GRAPHPTC over PTC	+11.7	+8.3	+4.7	+5.4	+8.5	+1.9	+3.9	+6.3
Gain of PTC over Direct Tool Calling	−5.7	+1.4	+14.4	−2.4	−6.0	−0.4	+1.8	+0.5

Table 3: Performance and efficiency comparison between GRAPHPTC and Direct Tool Calling, averaged across four backbones. Scores are reported using each benchmark's primary metric (%). Bold indicates higher scores or lower token and interaction-round costs.

	Score (%)		Input tokens (k)		Interaction rounds	
Benchmark	Direct Tool Calling	GRAPHPTC	Direct Tool Calling	GRAPHPTC	Direct Tool Calling	GRAPHPTC
AppWorld-Normal	82.3	**90.9**	151.4	**84.1**	16.4	**9.1**
AppWorld-Challenge	68.7	**88.6**	228.8	**133.5**	18.7	**11.7**
Agent-Diff	68.5	**70.8**	113.2	**49.0**	9.6	**6.6**
FanOutQA	60.0	**62.9**	128.5	**63.9**	8.7	**6.9**
FRAMES	79.3	**80.6**	93.0	**55.2**	**5.6**	7.1
AutomationBench	24.6	**29.0**	324.4	**171.9**	36.1	**14.3**

5.2 GRAPH-GUIDED RECOVERY BEHAVIOR

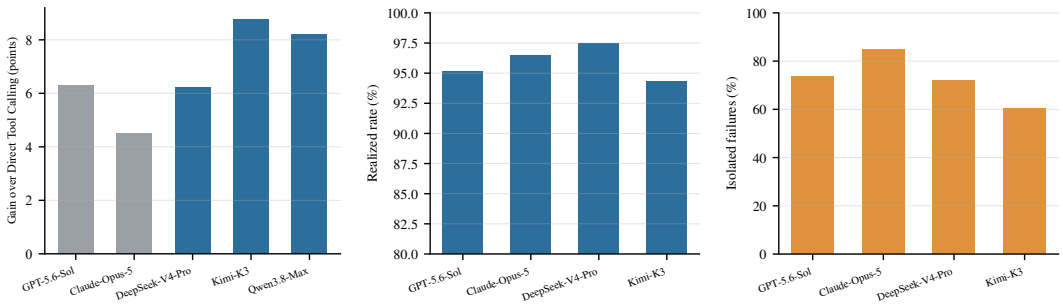
We measure the probability of selecting a recovery action conditioned on the realization flag of the preceding block, using the criteria in Appendix B. Recovery actions are approximately 3.4–5.2 times more likely after an unrealized intent (Figure 3c), suggesting that realization signals inform recovery decisions. These results are consistent with the intended role of graph-guided recovery but do not establish its causal contribution, because task selection depends on the behavior of GRAPHPTC and the methods also differ in execution representation and feedback.

5.3 PERFORMANCE ACROSS EXECUTION LENGTHS

We group AppWorld tasks by the number of program blocks in GRAPHPTC executions and measure gains over Direct Tool Calling within each group. Gains are positive across all four backbones and all groups, averaging 15.7 points for the shortest executions and 26.1 for the longest (Figure 3a). This increase is driven mainly by GPT-5.6-Sol and Kimi-K3, whose gains reach 24.1 and 43.2 points, respectively, in the longest-execution group. Claude-Opus-5 and DeepSeek-V4-Pro also improve in every group, although their gains do not increase monotonically with execution length. Longer executions require tracking more intermediate results and dependencies, making them harder to reconstruct from a linear history. The larger average gains are consistent with the explicit, addressable representation of artifacts and dependencies provided by GRAPHPTC, together with localized recovery at failed graph nodes.

5.4 INTERACTION EFFICIENCY AND TOKEN USAGE

On AutomationBench, GRAPHPTC uses approximately 54% to 65% fewer model interaction rounds than Direct Tool Calling while achieving a higher pass rate (Appendix C). This efficiency comes from executing multiple dependent calls within a single program block while recording intermediate artifacts and their reuse in the execution graph. On AppWorld, the curves averaged across four backbones show that GRAPHPTC achieves comparable cumulative task pass rates with approximately half as many interaction rounds as Direct Tool Calling, while matching or exceeding PTC at every evaluated round budget (Figure 3b). Because the execution graph remains in the runtime and only bounded graph-derived feedback is passed to the model, graph size does not directly increase the feedback budget, although other context may still grow across rounds. Across the settings reported in Table 3, GRAPHPTC achieves higher task performance with 1.7–$2.3\times$ fewer input tokens than Direct Tool Calling and fewer interaction rounds in all but the FRAMES setting.



(a) Gain by backbone (b) Intent realization rate (c) Failure isolation rate

Figure 4: Further analyses of GRAPHPTC. (a) Gains over Direct Tool Calling, averaged across seven settings for each backbone. (b) Intent realization rates and (c) proportions of block failures with no subsequent failure within three blocks, both measured on AppWorld across four backbones.

5.5 CONSISTENT GAINS ACROSS BACKBONE MODELS

GRAPHPTC improves performance across all five backbones (Figure 4a). Average gains over Direct Tool Calling are 7.7 percentage points for DeepSeek-V4-Pro, Kimi-K3, and Qwen3.8-Max, compared with 5.4 points for GPT-5.6-Sol and Claude-Opus-5. This difference may reflect how effectively each backbone tracks states and dependencies from textual history, as the execution graph makes these relations explicit. For Claude-Opus-5, although PTC underperforms Direct Tool Calling on average, GRAPHPTC outperforms both baselines in every evaluated setting.

5.6 INTENT REALIZATION AND FAILURE ISOLATION

Across the evaluated backbones, over 90% of blocks are classified as realized under the action-specific execution criteria described in Appendix B (Figure 4b). Most block failures under GRAPHPTC are not followed by another failure within the next three blocks (Figure 4c), whereas PTC often repeats the same exception on failed tasks. These patterns are consistent with explicit failure localization and targeted recovery helping the agent resume execution after failures.

5.7 PERFORMANCE ON STATE-MODIFYING WORKFLOWS

Agent-Diff annotates tasks by the operations they involve. Figure 5 groups tasks according to whether they update or delete existing state. GRAPHPTC improves over PTC by 7.2 percentage points on tasks that update or delete existing state, while showing a small decrease from 79.0% to 78.7% on tasks without such operations. By explicitly linking successive state versions, GRAPHPTC helps the agent track these changes and identify dependencies on updated state. The results therefore suggest that explicit state tracking is particularly useful for workflows in which earlier actions change the conditions for later steps.

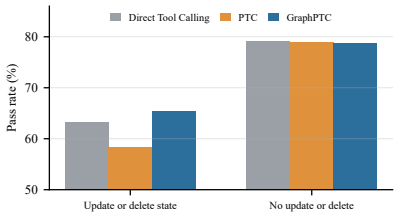


Figure 5: Agent-Diff pass rates by state-modifying operations, aggregated across four backbones.

6 CONCLUSION

In this paper, we present GRAPHPTC, which augments programmatic tool calling with a dynamic execution graph linking declared intents to observed effects and cross-block dependencies. Bounded graph feedback guides continuation, targeted repair, replanning, and answering. Across seven settings and five backbones, GRAPHPTC outperforms both baselines and reduces input token usage relative to Direct Tool Calling in efficiency evaluations, supporting explicit execution modeling for tool-use decisions.

Reproducibility Statement

We have made efforts to ensure the reproducibility of this work by providing detailed descriptions of the proposed method, experimental settings, implementation details, and evaluation protocols in the main paper and supplementary materials. The supplementary materials include the source code required to reproduce our experiments, and we will publicly release the code after the review process to further facilitate verification and future research. The content of this paper can be completely reproduced.

AI Use Statement

In this work, we used generative AI tools only for final-stage language polishing and grammar refinement of the manuscript. We have not used generative AI tools for generating research ideas, designing methods, conducting experiments, implementing code, analyzing data, interpreting results, or producing scientific claims, and other generative AI-assisted tasks requiring disclosure are not applicable to this work. We have reviewed all AI-assisted revisions to ensure their correctness and consistency with the authors' intended content. We take responsibility for the final content of this work, including all text, claims, and artifacts produced with the aid of generative AI.

References

Anthropic. Code execution with MCP: Building more efficient agents. `https://www.anthropic.com/engineering/code-execution-with-mcp`, 2025.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*, 2023.

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, Sahel Sharifymoghaddam, Yanxi Li, Haoran Hong, Xinyu Shi, Xuye Liu, Nandan Thakur, Crystina Zhang, Luyu Gao, Wenhu Chen, and Jimmy Lin. BrowseComp-Plus: A more fair and transparent evaluation benchmark of deep-research agent. *arXiv preprint arXiv:2508.06600*, 2025.

Susan B. Davidson and Juliana Freire. Provenance and scientific workflows: challenges and opportunities. In *ACM SIGMOD International Conference on Management of Data*, pp. 1345–1350, 2008.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided language models. In *International Conference on Machine Learning*, pp. 10764–10799, 2023.

Melanie Herschel, Ralf Diestelkämper, and Houssem Ben Lahmar. A survey on provenance: What for? what form? what from? *The VLDB Journal*, 26(6):881–906, 2017.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, 2024.

Satyapriya Krishna, Kalpesh Krishna, Anhad Mohananey, Steven Schwarcz, Adam Stambler, Shyam Upadhyay, and Manaal Faruqui. Fact, fetch, and reason: A unified evaluation of retrieval-augmented generation. In *Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics (NAACL)*, 2025.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, 2023.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37, 2024.

Hubert M. Pyskło, Artem Zhuravel, and Patrick D. Watson. Agent-Diff: Benchmarking llm agents on enterprise api tasks via code execution with state-diff-based evaluation. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2026.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *International Conference on Learning Representations*, 2024.

Preston Rasmussen, Pavlo Paliychuk, Travis Beauvais, Jack Ryan, and Daniel Chalef. Zep: A temporal knowledge graph architecture for agent memory. *arXiv preprint arXiv:2501.13956*, 2025.

Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. smolagents: A smol library to build great agentic systems. `https://github.com/huggingface/smolagents`, 2025.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36, 2023.

Daniel Shepard and Robin Salimans. AutomationBench. *arXiv preprint arXiv:2604.18934*, 2026.

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*, 2023.

Theodore R Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L Griffiths. Cognitive architectures for language agents. *Transactions on Machine Learning Research*, 2024.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Annual Meeting of the Association for Computational Linguistics*, 2024.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *International Conference on Machine Learning*, 2024.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. AutoGen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-MEM: Agentic memory for LLM agents. In *Advances in Neural Information Processing Systems*, 2025.

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

Andrew Zhu, Alyssa Hwang, Liam Dugan, and Chris Callison-Burch. FanOutQA: A multi-hop, multi-document question answering benchmark for large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 18–37, 2024.

A Node and Edge Types of the Execution Graph

Table 4 summarizes the node and edge types maintained in the execution graph of GraphPTC. The node types represent tasks, program blocks, tool or API calls with read or write effects, artifacts, state versions, failures, and action intents. The edge types encode execution structure, data flow, state evolution, failure attribution, cross-block equivalence and reuse, and intent binding.

Table 4: Node and edge types in the dynamic execution graph of GRAPHPTC.

Category	Types
Nodes	`TASK`, `BLOCK`, `CALL` (`read`/`write`), `ARTIFACT`, `STATE`, `FAILURE`, `INTENT`
Execution	`contains`, `executes`, `produces`, `fails_at`
Data flow	`consumes`, `reuses`, `derives`
State	`reads`, `writes`, `mutates`, `supersedes`
Cross-block	`equivalent_to`
Intent	`targets`, `implemented_by`

B IMPLEMENTATION DETAILS

GRAPHPTC maintains the execution graph incrementally, using node and edge cursors to track additions in each round. Extracting the round delta Δ_t therefore requires traversing only the nodes and edges added during the current round, without computing a general graph edit distance. During execution, tool wrappers record call arguments, results, and effects. After execution, the trace projector associates these records with the current program block. Cross-block equivalence detection uses tool effect contracts to normalize argument and result identities. Cached results from deterministic read operations may be reused, while operations with side effects are handled according to their tool contracts. To construct bounded feedback, the system first prunes list items and then reduces the included fields to satisfy the character budget B. The complete graph remains in the runtime; graph information is provided to the model through bounded feedback alongside the program output or execution error.

The runtime records an action realization flag using action-specific execution criteria. For PATCH, the flag is true when the repair block is reported as successful by the execution trace. For other nonterminal actions, it is true when the block produces new artifacts or recorded state effects, or when the number of completed goals increases. A block without a declared action receives a false flag. The declared expected change is retained in the action record, while the flag is computed from these execution signals.

C ADDITIONAL ANALYSES

Scenario Goal Completion on AppWorld. Figure 6 reports Scenario Goal Completion, the secondary AppWorld metric, which requires all tasks within a scenario to succeed. GRAPHPTC improves Scenario Goal Completion over PTC on both splits for every evaluated backbone. The gains are larger than those in Task Goal Completion, indicating that the improvements extend to the stricter requirement of completing an entire scenario.

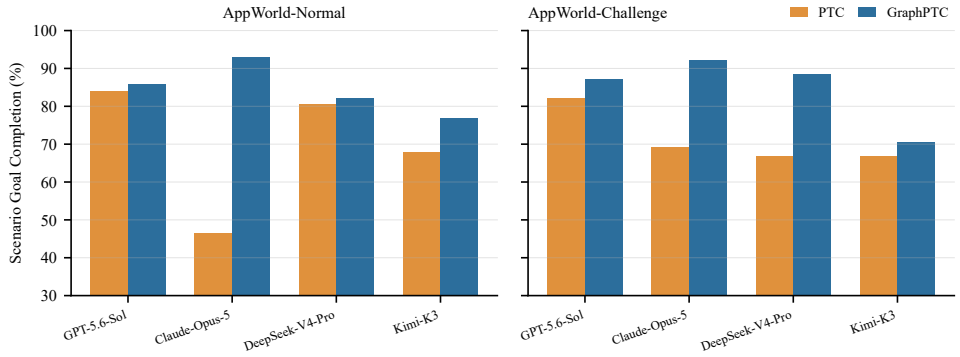


Figure 6: Scenario Goal Completion on the normal and challenge splits of AppWorld across four backbones. GRAPHPTC outperforms PTC in every evaluated combination of backbone and split.

Use of Recovery Actions. Table 5 reports the number of PATCH and REPLAN actions issued on AppWorld and the pass rate among tasks that invoke PATCH. All four evaluated backbones use

PATCH, while the frequency of REPLAN varies across backbones. Tasks that invoke PATCH achieve high pass rates, showing that successful task completion remains common even when execution requires repair. This pattern is consistent with the intended role of graph-targeted recovery in supporting continued execution after a failure.

Table 5: Recovery action usage on AppWorld, aggregated across the normal and challenge splits. Action counts report the total number of invocations. Pass rates are computed over tasks that invoke PATCH.

Backbone	PATCH calls	REPLAN calls	Pass rate on PATCH tasks (%)
GPT-5.6-Sol	516	52	100.0
Claude-Opus-5	506	20	97.8
DeepSeek-V4-Pro	586	0	94.7
Kimi-K3	972	3	99.1

Interaction Rounds on AutomationBench. Figure 7 reports the mean number of model interaction rounds per task on AutomationBench. GRAPHPTC uses approximately 54% to 65% fewer rounds than Direct Tool Calling while achieving a higher pass rate for every evaluated backbone. These results complement the AppWorld pass-rate–round curve in the main text and demonstrate efficiency gains in workflow automation as well as application control.

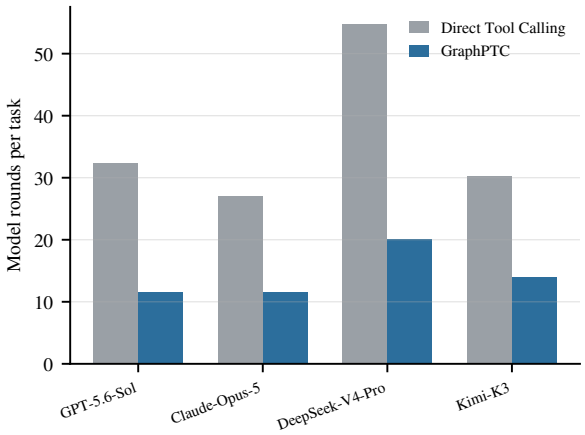


Figure 7: Mean model interaction rounds per task on AutomationBench. GRAPHPTC achieves a higher pass rate than Direct Tool Calling with fewer interaction rounds for every evaluated backbone.

Input Tokens across Benchmarks. Figure 8 reports mean input token usage per task, averaged across four backbones, for six of the seven evaluation settings. Across all reported settings, GRAPHPTC uses fewer input tokens than Direct Tool Calling, while requiring 1.1 to 1.4 times as many input tokens as PTC. Thus, the reduction in input token usage relative to Direct Tool Calling extends across multiple benchmarks, while structured graph feedback incurs additional token usage relative to PTC. The bounded feedback limits the graph information supplied at each round, allowing GRAPHPTC to retain the full execution graph in the runtime without including it in the model context.

Gain by Reasoning Type on FRAMES. Figure 9 breaks down the performance gains on FRAMES by reasoning type for GPT-5.6-Sol and Claude-Opus-5. The largest gain occurs in post-processing for GPT-5.6-Sol and temporal reasoning for Claude-Opus-5, where answers require integrating results from multiple calls. This pattern is consistent with the role of cross-block dependency linking in organizing intermediate evidence and helping the agent recognize redundant retrieval.

D CASE STUDIES

We present three case studies illustrating how the execution graph supports tool-calling decisions. Each case compares abbreviated execution traces for the same task under PTC and GRAPHPTC. Panel (a) shows the linear PTC history, while panel (b) shows the execution graph maintained by

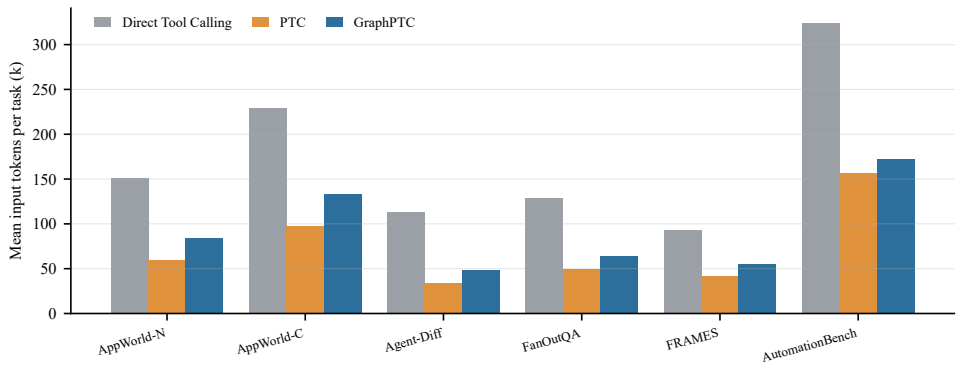


Figure 8: Mean input token usage per task, averaged across four backbones. In every reported setting, GRAPHPTC uses fewer input tokens than Direct Tool Calling while achieving the highest primary-metric score among the three methods.

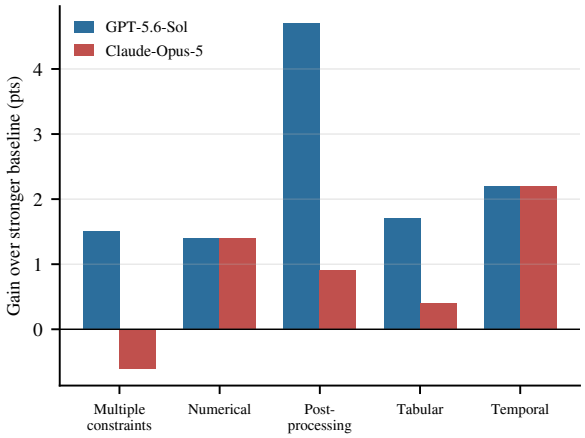
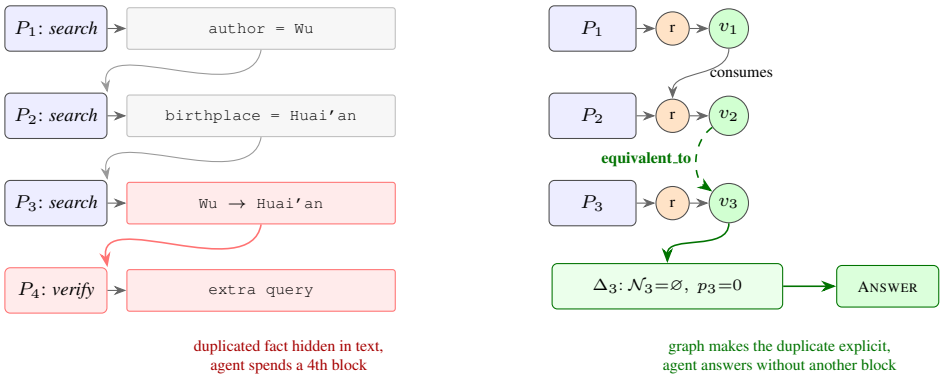


Figure 9: Accuracy gains of GRAPHPTC over the higher-scoring baseline within each reasoning category on FRAMES, reported in percentage points. The largest gain occurs in post-processing for GPT-5.6-Sol and temporal reasoning for Claude-Opus-5.

GRAPHPTC. The accompanying descriptions trace the execution sequence and identify the graph elements used to support each decision.

Case 1: Cross-Block Equivalence Identifies Redundant Retrieval. The task asks for the birthplace of the author of a specific novel. It requires a two-hop lookup: identifying the author and then using that name to retrieve the birthplace. As shown in Figure 10, both methods initially execute three program blocks. Block P_1 searches for the author, block P_2 uses the retrieved name to search for the birthplace, and block P_3 issues a reformulated query combining the two steps. This additional query may serve as a verification attempt or arise from a separate formulation of the original task. Under PTC, the three outputs appear as separate strings without an explicit representation of their equivalence. In the illustrated trace, the agent does not recognize that the third output repeats the fact returned by the second and executes a fourth block to reconcile the results. Under GRAPHPTC, the trace projector assigns each search result an identity key $\kappa(\cdot)$ based on the normalized search target and the returned entity. When the key of the third result matches that of the second, the projector adds an `equivalent_to` edge $v_2 \rightarrow v_3$. The delta Δ_3 then reports an empty set of novel artifacts and a progress indicator $p_3 = 0$. The bounded feedback identifies the third artifact as equivalent evidence. After receiving feedback from the third block, the agent selects ANSWER without executing an additional program block. The graph makes result equivalence explicit through a typed cross-block relation, supporting the decision to stop redundant retrieval.

Case 2: A Failed Write Prompts a Targeted Patch. The task is an AppWorld workflow that retrieves the latest phone bill, identifies the email address of the recipient, and sends the bill as

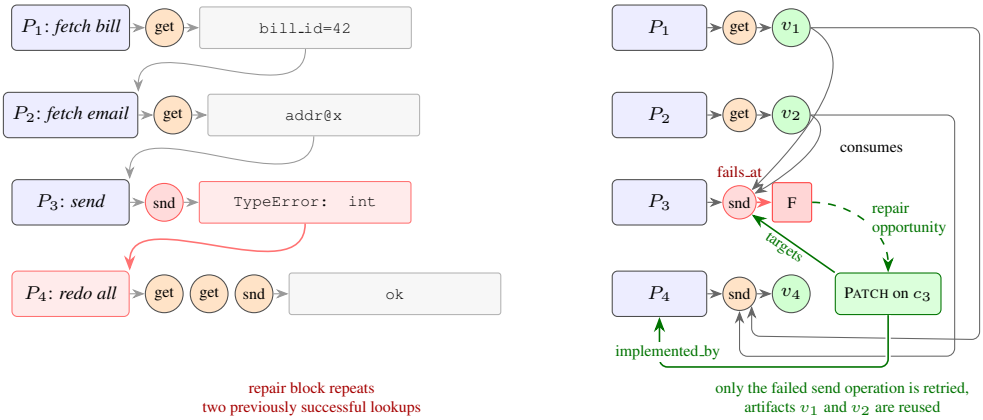


(a) PTC on a multi-hop lookup

(b) GRAPHPTC on the same lookup

Figure 10: Case 1, cross-block equivalence on a multi-hop question. (a) A linear history records three raw output strings and hides that the third and second carry the same fact under different phrasings. (b) The execution graph gives the two results one identity key $\kappa(\cdot)$ and one `equivalent_to` edge, and the delta Δ_3 indicates that the newly produced artifact adds no novel content.

an attachment. As shown in Figure 11, both methods issue three tool calls across blocks P_1, P_2, and P_3. The send call in P_3 fails with a type error involving an integer argument. Under PTC, the error appears within the lengthy textual output of the third block. The agent generates block 4, repeating the two successful lookups before issuing the corrected send call. Under GRAPHPTC, the projector records a `FAILURE` node and links the failing call c_3 to it through a `fails_at` edge. The delta Δ_3 exposes this failure as a repair opportunity. The agent declares $I_4 = (\text{PATCH}, z = c_3, e = \text{success})$, explicitly targeting the failed call, and generates a repair program in block P_4. This block issues a corrected send call c_4 that uses the existing artifacts v_1 and v_2, without repeating the two successful lookups. The `implemented_by` relation links the repair intent to P_4, while the artifact dependencies record the inputs used by c_4. Explicit failure attribution and intent-target binding focus the repair on the failed operation while preserving the results of successful calls.



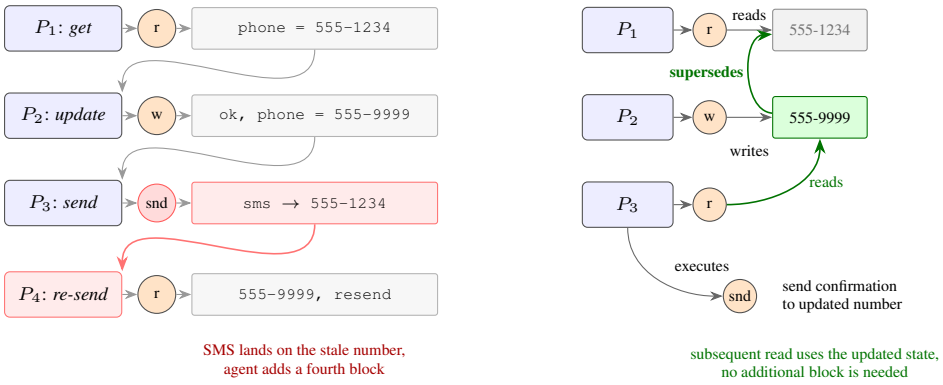
(a) PTC on a failing write

(b) GRAPHPTC on the same failure

Figure 11: Case 2: targeted repair of a failed send operation on AppWorld. (a) In the illustrated PTC execution, the repair block repeats two successful lookups before retrying the send operation. (b) GRAPHPTC associates the failure with c_3 through a `fails_at` edge. A PATCH intent targets c_3 and is implemented by block P_4, which issues the corrected call c_4 using the existing artifacts v_1 and v_2.

Case 3: A Subsequent Read Uses the Updated State. The task is an AppWorld workflow that updates the phone number of a contact and then sends a confirmation code to that contact. As shown in Figure 12, both methods begin with P_1 reading the current number and P_2 writing the new

number. Block P_3 then requires the number to send the confirmation code. Under PTC, the transcript contains both the previous value, `555-1234` from P_1, and the updated value, `555-9999` from P_2. In the illustrated trace, the agent selects the previous value and sends the SMS to the outdated number. A fourth block is then required to retrieve the contact again and resend the message. Under GRAPHPTC, the trace projector represents observable state through explicit versions. The read in P_1 is linked to the state version v_{old}, while the write in P_2 produces v_{new} and adds a `supersedes` edge $v_{\text{new}} \to v_{\text{old}}$. In the illustrated execution, the read in P_3 is linked to v_{new}, and the send operation uses the updated number. The confirmation code is therefore sent to the updated number on the first attempt, without requiring a fourth block. State versioning makes the relationship between the previous and updated values explicit, providing a structured basis for selecting the value used by the subsequent operation.



(a) PTC on a mutated state

(b) GRAPHPTC on the same task

Figure 12: Case 3: using updated state in an AppWorld workflow. (a) The PTC history contains both the previous and updated phone numbers. In the illustrated execution, the agent uses the previous number and requires an additional block to retrieve the updated value and resend the message. (b) GRAPHPTC explicitly represents the two state versions and connects them through a `supersedes` edge. The subsequent read is linked to the updated version, and the message is sent to the updated number without an additional block.