I-PTC: Interactive Programmatic Tool Calling for Stateful Tool-Augmented Agents

Huanzhi Mao* **Chengkun Cao*** **Shuo Yuan*** **Joseph E. Gonzalez**
University of California, Berkeley

Abstract

Large language models are increasingly capable of writing and executing code, suggesting a natural paradigm for tool-augmented agents: expressing tool use as programs rather than isolated calls. Programmatic tool calling (PTC) captures this idea through loops, batching, abstraction, and structured computation, but existing PTC approaches are largely stateless, limiting their ability to reuse intermediate results and adapt over time. This is problematic for real-world tasks, where tool environments and problem-solving processes are inherently stateful. We therefore present **I-PTC**, an interactive framework that executes stateful PTC: models issue code snippets that call tools, maintain reusable state, and evolve the tool interface through API corrections. To evaluate the effectiveness of I-PTC and the general PTC setting, we introduce **PTC-BENCH**, a benchmark of long-tail, production-style user requests whose difficulty comes from large initial environments, many dependent API interactions, extended planning, and recovery across follow-ups and transient failures. Experiments on PTC-BENCH show that I-PTC improves performance on complex multi-step tasks with large tool sets while substantially reducing token consumption over conventional tool-calling baselines and the original PTC baseline, highlighting stateful programmatic tool calling as a promising direction for scalable tool-augmented agents.

1 Introduction

As LLMs are asked to complete increasingly complex workflows over databases, search engines, code environments, and domain-specific APIs, the key bottleneck is no longer just tool selection, but the tool-use interface itself. Conventional tool calling emits one structured call at a time and returns each result to the prompt, which works for short tasks but becomes inefficient for long workflows: each call may require another inference step, intermediate results consume context, and operations such as looping, filtering, aggregation, branching, and joining API results must be simulated through repeated natural-language reasoning. Programmatic tool calling (PTC) [1–3] addresses this by letting agents write executable programs that orchestrate tools with loops, abstraction, conditionals, and structured computation outside the language context.

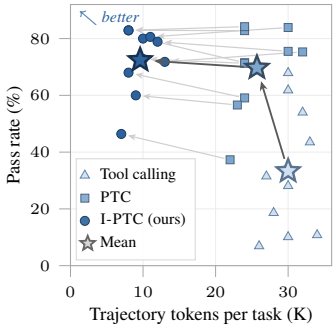


Figure 1: **Pass rate vs. trajectory tokens** for ten models on PTC-BENCH

While PTC improves complex tool use, standard execution remains brittle. One-shot programs must assume API behavior, return formats, and edge cases before

*Equal contribution.
Correspondence to Huanzhi Mao (huanzhimao@berkeley.edu).
Project code is available at https://github.com/HuanzhiMao/I-PTC.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

seeing any outputs; if these assumptions are wrong, the agent may need to restart. Splitting the workflow into smaller blocks helps exploration, but loses intermediate variables and cached results, causing redundant computation and repeated tool calls. Thus, PTC reduces context pollution but can limit observability and error recovery.

We therefore propose **I-PTC**, an interactive programmatic tool-calling framework that gives the model a persistent notebook-like execution environment [4, 5]. In I-PTC, the model can issue code snippets, call tools, inspect intermediate variables, debug execution errors, and continue from a preserved namespace. This design retains the token-efficiency advantages of PTC while recovering the observability and error-recovery benefits of conventional tool calling. By separating execution memory from language context, I-PTC allows intermediate state to persist as executable variables rather than being repeatedly serialized into the prompt.

Beyond within-task persistence, I-PTC also evolves its interaction with the tool environment. **On-demand tool inspection** lets the agent dynamically acquire detailed tool information only when needed, reducing upfront prompt overhead when operating over large tool collections. **Self-evolving APIs** allow the agent to incorporate execution feedback into persistent API corrections and usage notes, so discoveries about tool behavior can be reused across later code snippets. Together, these mechanisms extend PTC beyond static program execution: they turn tool discovery, execution feedback, and recurring tool-use patterns into reusable task state, improving efficiency and reliability across complex workflows.

To evaluate I-PTC and the broader PTC setting, we introduce **PTC-BENCH**, a benchmark of 240 realistic user queries over simulated local APIs. PTC-BENCH targets production-style workflows such as scheduling, shopping, travel planning, record analysis, personal finance review, and customer support. These tasks share a common structure: large initial environments, many dependent API calls, extended planning under constraints, and reuse of partial work across follow-ups or transient failures. Unlike existing benchmarks that focus on function-call validity, tool selection, and short-to-medium-horizon interaction, PTC-BENCH provides a complementary testbed for evaluating tool-use interfaces on long-horizon, stateful workflows.

We evaluate I-PTC against conventional tool calling and one-shot PTC across PTC-BENCH, MCP-Universe [6] and MCP-Bench [7], using both open-source and proprietary model families. Across these settings, I-PTC improves task success on most evaluated models while reducing token usage and unnecessary tool calls (Figure 1), showing that interactive stateful execution provides a more scalable interface for complex tool-augmented agents. In summary, this paper makes the following contributions:

- We propose **I-PTC**, a training-free, interactive programmatic tool-calling framework that gives agents a persistent notebook-like execution environment. I-PTC allows agents to iteratively execute code, inspect intermediate variables, debug errors, incorporate API corrections, and continue from preserved state, shifting PTC from stateless execution to stateful interaction.
- We introduce **PTC-BENCH**, a benchmark of 240 long-tail, production-style user requests over simulated APIs, characterized by large initial environments, many dependent API interactions, extended planning under user-supplied constraints, multi-turn follow-ups, and recovery from transient API failures. The benchmark is interface-agnostic: tasks are defined by the user request and a deterministic verifier over the final answer and environment state.
- We empirically evaluate traditional tool calling, standard PTC, and I-PTC across PTC-BENCH, MCP-Universe [6] and MCP-Bench [7] using both open-source and proprietary model families. Our results show that I-PTC improves task success on most evaluated models while substantially reducing token consumption and redundant tool use compared with conventional tool-calling and standard PTC baselines.

2 Related Work

Programmatic Tool Calling Recent work studies programmatic tool calling (PTC), where models write executable code to invoke tools, process results, and express control flow, rather than emitting isolated tool calls one step at a time [1]. This makes code a natural substrate for complex workflows involving repeated API calls, aggregation, filtering, and structured computation [2, 3]. Persistent Python state across turns has also been explored [8–11]. However, existing PTC approaches mostly

treat code as a one-shot orchestration mechanism: the model often commits to a full program before observing tool outputs, and failures from wrong API assumptions or return formats can require restarting the workflow. I-PTC makes PTC interactive and stateful: agents can execute partial programs, inspect intermediate outputs, reuse cached variables, and recover from errors. Unlike prior persistent-state systems, I-PTC also treats the tool interface as adaptable state, acquiring schemas on demand and writing failure-derived corrections into tool descriptions.

LLM Tool Calling and Benchmarks Tool-using LLMs have been studied through natural-language actions, structured function calls, and API-use benchmarks, including Toolformer [12], ReAct [13], Gorilla [14], ToolBench [15], BFCL, and τ-bench [16]. These works are central for evaluating tool selection, argument generation, function-call validity, conversational policy compliance, and short-to-medium-horizon interaction. In contrast, PTC-BENCH targets a complementary regime: production-style requests with large initial states, many dependent API calls, extended planning, multi-turn reuse, and recovery from transient failures. Thus, PTC-BENCH is designed to evaluate whether a tool-use interface can support long-horizon, stateful, and computation-heavy workflows, where ordinary call-observe-call interaction becomes inefficient.

Coding Agents Coding-agent systems such as SWE-agent [17], OpenHands [18], and Agentless [19] show that LLMs can use interactive development environments to inspect state, edit code, run commands, and learn from execution feedback, with SWE-bench [20] serving as a benchmark for repository-level software repair. Our work transfers this insight from software engineering to general tool use: code becomes the control layer for querying APIs, transforming data, caching intermediate results, and repairing failed strategies. I-PTC extends this setting with a persistent notebook-style environment, making programmatic tool use more adaptive than one-shot PTC and more scalable than sequential tool calling.

3 I-PTC Framework

Although PTC allows agents to express tool use as executable programs, a naive PTC interface is often stateless. Each invocation executes a code snippet in a fresh sandbox and discards the namespace afterward. This design is efficient when the generated program is correct, but brittle when the agent must explore unfamiliar APIs, inspect intermediate outputs, or recover from incorrect assumptions about return formats. In such cases, the agent must either serialize intermediate state back into the prompt or re-derive it in a later call, leading to redundant computation, repeated tool calls, and limited error recovery.

I-PTC addresses this limitation by turning one-shot program execution into an interactive, code-based tool-use interface. Tools are Python-callable functions, and by *interface* we mean the protocol around execution: what API information is exposed, how schemas are acquired, what state persists, and whether failures update future tool descriptions. I-PTC combines three distinct mechanisms: (i) *persistent execution state* within a task (Section 3.1), a notebook-like environment where agents write code incrementally, inspect intermediate results, recover from errors, and reuse variables across rounds, built on established infrastructure; (ii) *self-evolving API documentation* across tasks (Section 3.2), which persists failure-derived corrections; and (iii) *on-demand API discovery* (Section 3.3), which needs no persistent namespace. Together, these let API knowledge accumulate as reusable state instead of being repeatedly serialized into the context.

3.1 Stateful Code Execution

We implement stateful execution with Jupyter-notebook-cell semantics. Unlike the standard PTC tool, which creates a fresh sandbox for each invocation, our interactive executor lazily initializes a single dictionary namespace on the first call and reuses it for all code cells within the same task. Variables, imports, and function definitions introduced in one cell therefore remain available in later cells. The executor runs code statement by statement, so if a cell raises an exception, state produced by earlier statements in that cell is preserved. This allows the agent to recover from mid-cell failures without discarding completed work. Although persistent notebooks can suffer from hidden or inconsistent state in human use, this issue is less problematic for agents because the interaction history is linear: each cell, output, and error is appended to the trajectory in execution order. The agent can therefore condition on the full ordered history when deciding the next action, preserving

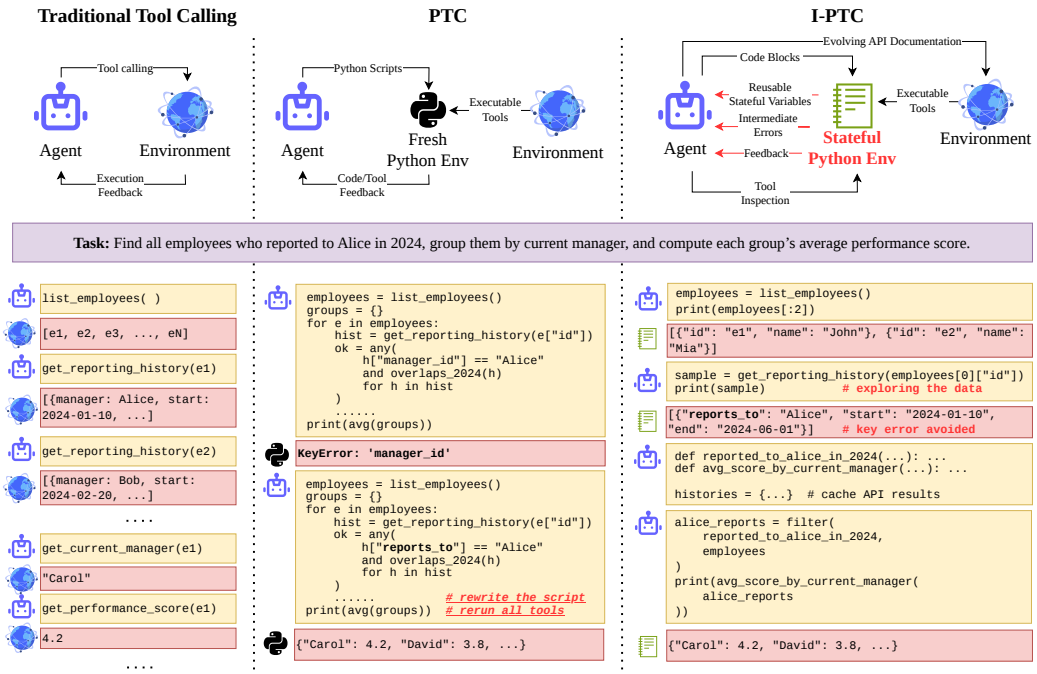


Figure 2: **Comparison of traditional tool calling, programmatic tool calling (PTC), and interactive programmatic tool calling (I-PTC).** Traditional tool calling executes tools through sequential agent-environment interactions, while PTC lets the agent generate executable code that orchestrates tool use inside a Python environment. I-PTC further extends PTC with a stateful execution environment, enabling reusable variables, intermediate-result inspection, execution feedback, tool inspection, and evolving API documentation across multiple rounds.

the benefits of notebook-style exploration without requiring the namespace itself to be serialized into the prompt.

3.2 Self-Evolving APIs

Tool-using agents typically operate over a fixed tool surface: each API method description is hand-authored once and then treated as ground truth, even when omissions or ambiguities repeatedly cause recoverable mistakes. In I-PTC, these mistakes naturally surface as fine-grained execution feedback because agents interact with tools through incremental code cells, probe APIs, inspect outputs, and debug errors across rounds. We treat these local failures as signals for improving the tool interface rather than merely as task-level errors.

We implement this idea with a lightweight API-documentation update mechanism. After every batch of B tasks, we scan accumulated execution traces for API method errors and Python tracebacks, together with the corresponding code blocks and original method descriptions. An update agent then produces a concise one- to two-sentence addendum for each affected method. These addenda are exposed to agents in subsequent batches and persist across sessions, allowing future agents using the same APIs to benefit from previously discovered corrections, constraints, and usage notes.

3.3 On-Demand Tool Inspection

In realistic tool-use settings, agents are often exposed to a broad catalog of APIs, even though only a small subset is relevant to any particular task. The full per-method schema for every available API is placed in the system context on every turn, causing agents to repeatedly pay for many irrelevant tool descriptions. At the same time, Python-based tool use often involves runtime uncertainty: APIs may return dictionaries, custom objects, nested records, or duck-typed objects whose useful fields and methods are not fully known until execution.

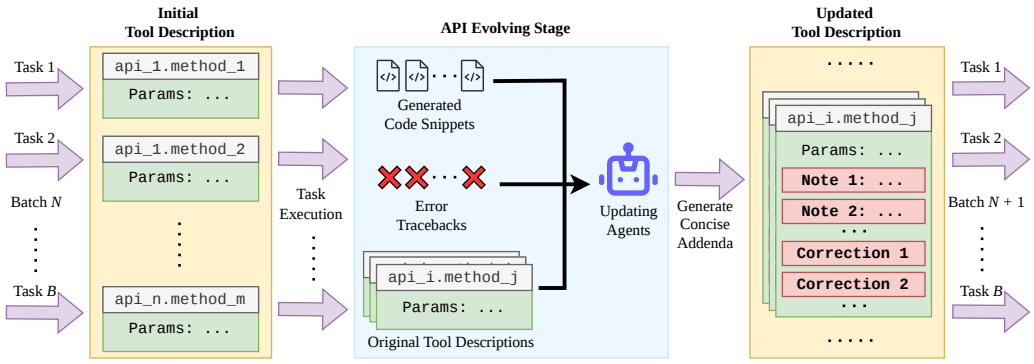


Figure 3: **Self-evolving API documentation.** After each task batch, the framework collects code snippets, API errors, tracebacks, and original tool descriptions. An updating agent converts these failure traces into concise API addenda, such as usage notes or corrections, which are appended to future tool descriptions so later agents avoid repeated mistakes.

To reduce schema overhead while improving runtime observability, we integrate on-demand tool inspection into I-PTC. Inspired by Jupyter notebook inspection syntax, I-PTC provides a hierarchical pull-based mechanism inside code cells: a line of the form `<api>??` lists the methods available on an API server, while `<api>.<method>??` returns the full description and parameter schema for a single method. The same pattern also supports exploratory programming: after calling a tool, the agent can inspect returned objects, their types, and available fields before continuing in later cells. With inspection enabled, the initial tool list contains only the code-execution tool and bare API class names. Full method schemas enter the context only when explicitly requested, so irrelevant APIs are never inspected and never incur schema cost.

4 Evaluation

To further explore PTC and evaluate the effectiveness of I-PTC, we propose PTC-BENCH, a simulated-API benchmark for evaluating tool-use interfaces on complex, stateful workflows. Each task provides a natural-language user request, a fixed initial environment state, a set of available APIs, and a deterministic verifier that checks the final answer and, when applicable, the final simulated environment state.

Benchmark design PTC-BENCH targets the workloads that emerge as users delegate increasingly involved tasks to LLM agents: scheduling, travel planning, and other workflow automation. Although each individual request is idiosyncratic and long-tail, these tasks share a recurring structural pattern. The user request cannot be resolved by selecting a small number of APIs. Instead, the agent is presented with a large initial environment, must issue many dependent API queries, retain and filter intermediate results, aggregate or optimize across records, and produce constraint-satisfying final actions; in multi-turn or failure-prone settings it must also reuse partial work across follow-ups or after errors.

Table 1: Complexity statistics for the PTC-Bench main subset.

Property	Value	Property	Value
Mean entities	244	Median state size	42 KB
Mean API calls	87.7	Median API calls	51
p90 API calls	202	$\geq$50 API calls	51%
$\geq$100 API calls	38%		

As summarized in Table 1, these properties show up quantitatively as large initial states, long tool-use trajectories, and high p90 API-call counts. The benchmark is defined entirely by these workload characteristics: tasks, initial states, and verifiers are specified at the level of the user request and the final environment, and any tool-use interface (traditional function calling, programmatic, or

Main Query

User: I have gift-card balances at several grocery delivery stores. Buy everything on my shopping list while spending as little cash out-of-pocket as possible. Not every store carries every item. Return JSON with `total_out_of_pocket` and `assignments`.

Multi-turn Query

Turn 1. Schedule a 45-minute meeting with Sam, Alice, Jordan, and Alex this week. Find all UTC slots where all four are free.

Turn 2. Alex cannot make it. From the data already gathered, which additional slots become available with only Sam, Alice, and Jordan?

Error-Recovery Query

User: Find recurring payments in my Venmo history from July 2025 through March 2026.

Injected error: `get_transaction_history` returns an internal service error. Please try again.

Figure 4: **Examples of PTC-Bench query types.** Main queries are single-shot tasks; multi-turn queries require reusing information gathered in an earlier turn; error-recovery queries inject transient tool failures that the agent must recover from. Examples are abbreviated for readability.

otherwise) can be evaluated on the same tasks. Construction, validation, and neutrality with respect to I-PTC are detailed in Appendices B, B.5, and B.6.

Tasks and Evaluation PTC-BENCH contains 240 tasks across three subsets, each targeting a different mode of complex tool use. PTC-BENCH runs on a fully simulated local backend: each manually written task is paired with a diverse initial state, and correctness is checked deterministically by task-specific Python verifiers over the final answer and environment state. Across all subsets, tasks span realistic workflow domains such as calendar scheduling, restaurant and review analysis, ride and route optimization, grocery and delivery cost optimization, multi-API daily-life planning, etc. Example user queries are shown in Figure 4. These tasks require agents to plan over multiple steps, coordinate dependent tool calls, compare alternatives, aggregate structured records, optimize under constraints, and update environment states. We organize them into the following three subsets:

- **Main queries (100).** Single-turn tasks evaluating complex end-to-end tool orchestration. Each task gives one realistic request that may require many dependent API interactions, structured computation over tool outputs, and one or more valid final actions or answers.
- **Multi-turn queries (70).** Two-turn tasks modeling user follow-ups after an initial result. The follow-up depends on data and intermediate reasoning from the first turn, testing whether agents can reuse or recompute prior work. Accuracy is evaluated on the follow-up response.
- **Error-recovery queries (70).** Single-turn tasks testing robustness to mid-execution API failures. We inject hidden deterministic errors into selected API methods, measuring whether agents can inspect partial progress, recover, and continue without unnecessarily repeating prior work.

Across all subsets, task-specific verifiers check the final response and, when relevant, the final simulated API state, including created events, orders, or other environment updates. We report both correctness and efficiency metrics, including pass rate, token usage, and API calls.

5 Experiments

We evaluate all methods on PTC-BENCH, the 240-task benchmark described in Section 4, including its main, multi-turn, and error-recovery subsets. To study whether our findings transfer beyond PTC-BENCH, we also evaluate selected methods on established MCP-based tool-use benchmarks, including MCP-Universe [6] and MCP-Bench [7]. We evaluate a broad set of proprietary and open-weight models on PTC-BENCH. Additional evaluation on provider-native code execution for Anthropic's built-in code execution tool for Claude and the corresponding GPT and Gemini runtimes can be found in Appendix C.2.

We organize evaluated methods into three interface families. **Traditional tool calling** uses the standard call-observe-call interface, where the model emits one structured tool call at a time and receives each result in the conversation context. **PTC** allows the model to write executable Python code that invokes APIs and performs local computation, but each execution starts from a fresh environment, so intermediate variables and helper functions are not preserved across calls. **I-PTC** is our proposed stateful programmatic interface, where the model executes code in a persistent notebook-like environment, with a API learning batch size of 10.

Table 2: Average accuracy and token usage across three runs on PTC-BENCH. I-PTC achieves the best or near-best accuracy for most models while consistently reducing total and trajectory token usage. Standard deviations are reported in appendix C.1.

Model	Accuracy ↑			Raw Tokens ↓			Trajectory Tokens ↓		
	Tool-only	PTC	I-PTC	Tool-only	PTC	I-PTC	Tool-only	PTC	I-PTC
GPT 5.4	18.6%	71.5%	**80.0%**	78K	60K	**48K**	28K	24K	**10K**
GPT 5.4 Mini	10.1%	59.1%	**68.0%**	123K	72K	**45K**	30K	24K	**8K**
GPT 5.4 Nano	6.9%	37.3%	**46.4%**	85K	57K	**53K**	26K	22K	**7K**
Gemini 3.1 Pro	67.9%	82.8%	**82.9%**	167K	95K	**65K**	30K	24K	**8K**
Gemini 3.1 Flash-Lite	10.8%	71.3%	**72.8%**	552K	122K	**85K**	34K	24K	**10K**
Gemini 2.5 Pro	31.5%	56.6%	**60.0%**	145K	88K	**67K**	27K	23K	**9K**
Claude Opus 4.7	61.8%	**84.2%**	82.9%	172K	184K	**85K**	30K	24K	**8K**
Claude Sonnet 4.6	54.0%	**83.9%**	80.6%	139K	218K	**112K**	32K	30K	**11K**
Qwen 3.5-397B-A17B	43.5%	**75.3%**	71.8%	232K	180K	**118K**	33K	32K	**13K**
Kimi K2.5	28.1%	75.5%	**78.9%**	130K	169K	**110K**	30K	30K	**12K**

We report pass rate as the primary correctness metric. To measure efficiency, we report two complementary token metrics. **Trajectory tokens** estimate the effective token usage under caching by tokenizing each unique piece of trajectory content once, removing repeated-prefix overhead from multi-turn chat APIs. This better reflects the distinct information the agent processes and produces during the workflow. **Raw tokens** are the provider-reported input and output tokens summed across all LLM calls without accounting for caching or repeated context. Raw tokens therefore capture the uncached execution cost, while trajectory tokens approximate the practical cost when repeated context can be reused. All results are averaged over three independent end-to-end runs on the same task sets; standard deviation of the runs can be found in appendix C.1.

5.1 Main Results on PTC-BENCH

We first compare the three main interfaces on PTC-BENCH: traditional tool calling, standard PTC, and I-PTC. Table 2 reports pass rate and token usage across model families.

PTC improves pass rate over traditional tool calling. This suggests that many PTC-Bench tasks require more than selecting the right APIs: they require composing API results through structured computation. PTC supports this directly by letting the model express loops, filtering, aggregation, and branching as executable code rather than as repeated call-observe-call steps.

I-PTC improves over standard PTC. I-PTC improves over standard PTC for most evaluated models, with the largest gains on GPT and consistent gains on Gemini. Claude is the main exception, where standard PTC remains slightly stronger, suggesting that some frontier models can already produce effective one-shot tool-use programs when API assumptions are correct. Overall, I-PTC offers a more robust interaction model for long-horizon tool use by letting agents inspect outputs, cache results, define helpers, and revise strategies across cells, making it especially useful when tool behavior is uncertain or recovery is needed.

I-PTC reduces token usage. I-PTC reduces both raw tokens and trajectory tokens across most evaluated models. For example, Claude Opus 4.7 drops from 184K to 85K raw tokens and from 24K to 8K trajectory tokens, while Claude Sonnet 4.6 drops from 218K to 112K raw tokens and from 30K to 11K trajectory tokens. These reductions suggest that stateful execution together with tool inspection avoids repeatedly reconstructing the same workflow by allowing agents to cache intermediate results, inspect tool schema only when needed, and continue from previous cells.

Stateful Execution Helps Most in Multi-Turn and Recovery Tasks. Table 3 shows that I-PTC consistently improves multi-turn and error-recovery accuracy across GPT-5.4 mini, GPT-5.4, and Gemini 3.1 Flash-Lite. These are settings where persistent state is particularly useful: I-PTC can reuse cached API results, helper functions, and intermediate computations instead of recomputing them. The Q2-only multi-turn results illustrate this benefit: second-turn API calls fall from roughly 27–30 to 4–8 across the three models. Trajectory tokens also decrease across all task subsets for all three models. Together, these results suggest that stateful execution improves efficiency broadly, with the clearest accuracy benefits on multi-turn and error-recovery tasks.

Table 3: **Breakdown by task type.** Each cell lists PTC / I-PTC to isolate the effect of stateful programmatic execution. Metrics are averaged over three runs. API calls report the mean number of underlying simulated backend invocations per task. For multi-turn tasks, parentheses give API calls in the second turn (Q2) alone.

Task Type	Pass ↑	API ↓	Raw ↓	Traj. ↓
GPT-5.4 mini				
Multi-turn	69.0% / **77.1%**	180.9 (26.9) / **90.6** (**4.1**)	83K / **51K**	24K / **9K**
Error-recovery	61.3% / **72.9%**	92.6 / **74.6**	70K / **38K**	24K / **7K**
Main	50.0% / **59.7%**	**69.4** / 81.1	66K / **43K**	23K / **7K**
GPT-5.4				
Multi-turn	78.1% / **87.1%**	103.1 (30.6) / **63.4** (**8.0**)	75K / **57K**	24K / **11K**
Error-recovery	78.0% / **85.7%**	74.6 / **66.6**	58K / **42K**	23K / **9K**
Main	62.3% / **71.0%**	71.8 / **57.1**	51K / **47K**	24K / **10K**
Gemini 3.1 Flash-Lite				
Multi-turn	77.1% / **78.6%**	250.7 (29.8) / **102.3** (**8.1**)	141K / **93K**	26K / **11K**
Error-recovery	75.6% / **82.4%**	134.3 / **107.7**	110K / **69K**	24K / **9K**
Main	**64.1%** / 62.0%	**83.9** / 119.8	116K / **90K**	23K / **9K**

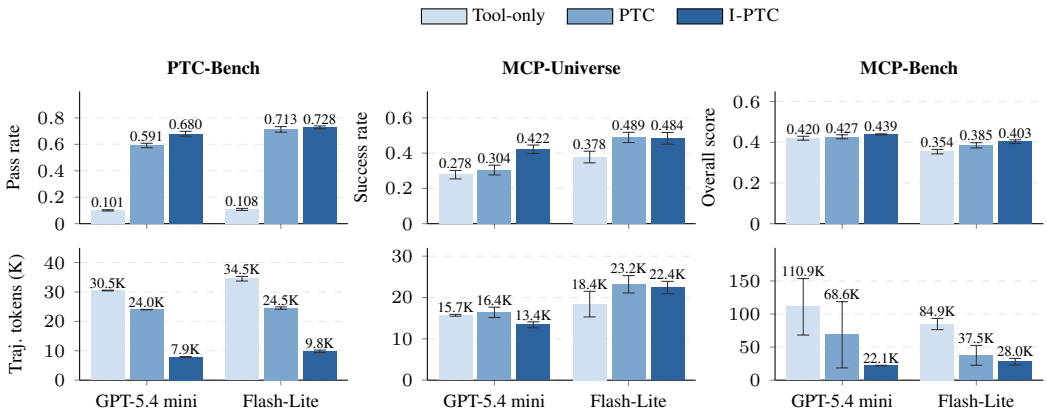


Figure 5: Generalization to existing tool-calling benchmarks. We evaluate GPT-5.4 mini and Gemini 3.1 Flash-Lite with traditional tool calling, standard PTC, and I-PTC on PTC-BENCH, MCP-Universe subset and MCP-Bench subset.

Overall, the main results show that programmatic tool calling provides a stronger interface than conventional tool calling for complex tool-use workflows, and that making this interface interactive and stateful provides additional benefits beyond stateless code execution.

5.2 Generalization to Other Tool-Calling Benchmarks

PTC-BENCH is designed to stress complex programmatic tool orchestration, but a useful interface should also transfer beyond our benchmark. We therefore evaluate GPT-5.4 mini and Gemini 3.1 Flash-Lite with traditional tool calling, standard PTC, and I-PTC on two other tool use benchmarks: MCP-Universe [6] and MCP-Bench [7]. We integrate both into the same three-mode interface used for PTC-BENCH, while preserving their original tools and evaluators.

Figure 5 shows that I-PTC transfers to MCP-based benchmarks across models: with GPT-5.4 mini, it outperforms traditional tool calling and standard PTC while using fewer tokens; with Gemini 3.1 Flash-Lite, it outperforms PTC on MCP-Bench and ties it on MCP-Universe. These results suggest that the benefits of stateful programmatic execution extend beyond PTC-BENCH to broader tool-use settings.

Table 4: **Execution overhead.** Med. cell is the median execution time of I-PTC.

(a) Mean wall time per task over PTC-BENCH tasks.

Model	Tool-only	PTC	I-PTC	Med. cell
GPT-5.4 mini	**14.0 s**	19.1 s	15.5 s	8 ms
Gemini 3.1 Pro	117.8 s	**44.6 s**	53.7 s	5 ms
Claude Sonnet 4.6	84.9 s	96.5 s	**65.6 s**	9 ms
Kimi K2.5	239.9 s	179.9 s	**147.4 s**	8 ms

(b) GPT-5.4 mini, main subset.

Condition	Total exec.	Exec. share	Peak namespace
Tool-only	0 s	0.0%	–
PTC	747.3 s	38.7%	0.53 MB
I-PTC	1.9 s	0.2%	0.44 MB

Table 5: **Component ablation** on GPT-5.4 mini and Gemini 3.1 Flash-Lite. On-demand inspection and self-evolving documentation are added to persistent cells (upper block) and to stateless execution (lower block); mean ± std over three runs.

	GPT-5.4 mini			Gemini 3.1 Flash-Lite		
Configuration	Pass ↑	Raw Tok. ↓	Traj. Tok. ↓	Pass ↑	Raw Tok. ↓	Traj. Tok. ↓
Persistent cells						
Raw I-PTC	61.3 ± 0.8	129K±5K	25.7K±0.2K	71.9 ± 2.7	130K±2K	24.1K±0.1K
Raw I-PTC + inspection	$\mathbf{68.1 \pm 1.7}$	44K±1K	**6.8K±0.1K**	72.8 ± 2.8	**84K±5K**	**9.3K±0.4K**
Raw I-PTC + documentation	62.9 ± 3.3	124K±0.5K	24.8K±0.2K	70.1 ± 0.9	128K±4K	23.9K±0.2K
Full I-PTC	68.0 ± 1.8	45K±0.4K	7.9K±0.1K	$\mathbf{72.8 \pm 1.1}$	85K±5K	9.8K±0.4K
Stateless execution						
PTC	59.1 ± 1.7	72K±2K	24.0K±0.1K	71.3 ± 2.1	122K±6K	24.5K±0.4K
PTC + inspection	65.8 ± 0.7	**40K±1K**	8.8K±0.4K	70.1 ± 0.5	86K±2K	10.2K±0.3K
PTC + documentation	57.2 ± 1.7	63K±2K	23.0K±0.2K	66.8 ± 1.7	126K±1K	24.2K±0.1K
PTC + inspection + documentation	64.7 ± 4.0	44K±3K	9.4K±0.4K	70.7 ± 0.6	115K±7K	12.2K±0.3K

5.3 Execution Overhead

Token usage alone does not capture net efficiency, so we instrumented the executor to record wall-clock time, code-execution time, and namespace size. The persistent environment adds little overhead: the median I-PTC cell runs in 5–9 ms across models (Table 4a), and in an instrumented GPT-5.4 mini run (Table 4b), I-PTC spends only 0.2% of its wall time executing code with a peak namespace of 0.44 MB, whereas stateless PTC spends 38.7% because a few one-shot programs committed to expensive brute-force strategies before observing intermediate outputs. In end-to-end wall time (Table 4a), I-PTC is faster than PTC for three of the four models and slower only for Gemini 3.1 Pro (53.7 s vs. 44.6 s per task). Since cells take milliseconds, these differences come from provider latency and the number of model rounds rather than from notebook persistence.

5.4 Ablation Study

We ablate I-PTC to identify which components drive the gains. Specifically, we ask: (1) how much benefit comes from the stateful cell interface, and whether agents reuse variables and helper functions across cells and (2) whether self-evolving API documentation reduces repeated API misuse. These ablations separate the effects of persistent execution state and learned tool-surface improvements.

Stateful cell execution and state reuse. We first isolate I-PTC's stateful execution environment. Table 5 compares persistent cells with stateless execution on GPT-5.4 mini and Gemini 3.1 Flash-Lite, with and without on-demand inspection and documentation. Persistent cells raise the mean pass rate in all eight matched configurations, by 0.6–5.7 points, although several of these gains are within run-to-run variation. On-demand inspection accounts for most of the token savings, with or without a persistent namespace. Table 6 verifies that agents use this persistence. We rerun each cell in a fresh namespace and count it as state-reusing if it fails due to missing variables, imports, or helper definitions. Overall, 61.0% of cells depend on prior states, with especially high reuse on error-recovery tasks. Thus, I-PTC does not simply split one-shot programs into smaller snippets but reuses intermediate states across the trajectory.

Table 6: **Cross-cell state reuse** (GPT-5.4, three runs). Cells that fail when rerun in a fresh namespace count as state-reusing.

Subset	Total	Reuse	Rate
Main	279.3	158.7	**56.8%**
Multi-turn	195.0	116.3	**59.7%**
Error-recovery	238.3	160.0	**67.1%**
Overall	712.7	435.0	**61.0%**

(a) Documentation-update ablation

Batch Size	API Err. ↓	Pass ↑	Raw Tok. ↓
1	82.3	80.6%	47K
5	91.3	80.6%	48K
10	94.3	80.3%	49K
20	88.0	80.8%	47K
50	148.3	80.8%	49K
100	181.3	81.8%	49K

(b) API errors vs. batch size

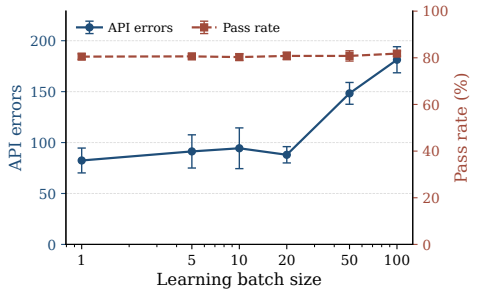


Figure 6: **Effect of self-evolving API documentation.** (a) Ablation over documentation-update batch size. (b) API errors increase as the learning batch size grows on the 100 main queries; batch size 100 corresponds to no effective learning during the run.

Self-evolving API documentation. We also evaluate whether I-PTC's self-evolving API documentation reduces repeated tool misuse. The agent accumulates lightweight corrections from API failures and applies them every B examples, where smaller B means more frequent updates and $B = 100$ means effectively no update within each task type.

Figure 6 shows that pass rate and token cost remain stable across batch sizes, but API errors rise sharply when updates are delayed. This suggests that self-evolving documentation mainly improves tool-use reliability by preventing repeated API mistakes. While our benchmark has only 240 queries, this effect should become increasingly important as tasks, tools, and repeated API interactions scale in the real world.

6 Limitations

Our work has three main limitations. First, self-evolving API documentation remains preliminary: although it helps reduce API errors, incorrect addenda may persist across tasks and affect later executions. Future work should validate updates before persistence and support versioning or rollback.

Second, I-PTC inherits risks and overheads from notebook-style persistent execution. Persistent state may introduce hidden-state errors from stale variables or unintended side effects. The measured overhead is small (Section 5.3), but I-PTC is not uniformly faster than PTC, and our multi-turn tasks have only two turns; longer sessions may amplify stale-state errors and namespace growth. Future work could improve persistent-state management through automatic cleanup, dependency tracking, stale-state detection, and selective reset.

Third, programmatic interfaces require sufficient model capability. In the Qwen3.5 family, they slightly underperform traditional tool calling at 0.8B and 2B parameters and help only from 4B upward (Appendix C.3).

7 Conclusion

We introduce I-PTC, an interactive extension of PTC that moves tool use from isolated code execution to a stateful environment. Unlike standard PTC, I-PTC lets agents incrementally write code, inspect outputs, cache API results, and recover from errors. We also introduce PTC-BENCH, a benchmark of long-tail, production-style user requests covering large initial environments, many dependent API interactions, extended planning, multi-turn follow-ups, and recovery from transient failures. Across PTC-BENCH, I-PTC improves pass rate on most evaluated models while significantly reducing token usage compared with traditional tool calling and standard PTC and redundant API calls relative to PTC. Overall, for realistic long-horizon workflows, our results suggest that future tool-augmented agents should move beyond isolated tool invocation toward persistent, stateful, and programmatic interaction environments.

Acknowledgment

Sky Computing Lab is supported by gifts from Accenture, Algorithmic SuperIntelligence Labs, Amazon, AMD, Anyscale, Broadcom, cmpnd, Google, IBM, Intel, Intesa Sanpaolo, Lambda, Lightspeed, Mirendil, NVIDIA, Samsung SDS, and VESSL.

References

[1] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents, 2024.

[2] Anthropic. Introducing advanced tool use on the claude developer platform. `https://www.anthropic.com/engineering/advanced-tool-use`, 2025. Accessed 2026-05-06.

[3] Letta. Programmatic tool calling with any llm. `https://www.letta.com/blog/programmatic-tool-calling-with-any-llm`, 2025. Accessed 2026-05-06.

[4] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damian Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87–90. IOS Press, 2016.

[5] Fernando Pérez and Brian E. Granger. Ipython: A system for interactive scientific computing. *Computing in Science & Engineering*, 9(3):21–29, 2007. doi: 10.1109/MCSE.2007.53.

[6] Ziyang Luo, Zhiqi Shen, Wenzhuo Yang, Zirui Zhao, Prathyusha Jwalapuram, Amrita Saha, Doyen Sahoo, Silvio Savarese, Caiming Xiong, and Junnan Li. Mcp-universe: Benchmarking large language models with real-world model context protocol servers, 2025. URL `https://arxiv.org/abs/2508.14704`.

[7] Zhenting Wang, Qi Chang, Hemani Patel, Shashank Biju, Cheng-En Wu, Quan Liu, Aolin Ding, Alireza Rezazadeh, Ankit Shah, Yujia Bao, and Eugene Siow. Mcp-bench: Benchmarking tool-using llm agents with complex real-world tasks via mcp servers, 2025. URL `https://arxiv.org/abs/2508.20453`.

[8] Bo Qiao, Liqun Li, Xu Zhang, Shilin He, Yu Kang, Chaoyun Zhang, Fangkai Yang, Hang Dong, Jue Zhang, Lu Wang, Minghua Ma, Pu Zhao, Si Qin, Xiaoting Qin, Chao Du, Yong Xu, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. TaskWeaver: A code-first agent framework, 2023.

[9] Maohao Ran, Zhenglin Wan, Cooper Lin, Yanting Zhang, Hongyu Xin, Hongwei Fan, Yibo Xu, Beier Luo, Yaxin Zhou, Wangbo Zhao, Lijie Yang, Lang Feng, Fuchao Yang, Jingxuan Wu, Yiqiao Huang, Chendong Ma, Yusen Huang, Dailing Jiang, Jianbo Deng, Sirui Han, Yang You, Bo An, Yike Guo, and Jun Song. CaveAgent: Transforming LLMs into stateful runtime operators, 2026.

[10] Alex L. Zhang, Tim Kraska, and Omar Khattab. Recursive language models, 2025.

[11] Ellie Y. Cheng, Logan Weber, Tian Jin, and Michael Carbin. Sharing state between prompts and programs, 2025.

[12] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023.

[13] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2022.

[14] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis, 2023.

[15] 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, 2023.

[16] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains, 2024.

[17] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024.

[18] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents, 2024.

[19] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents, 2024.

[20] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2023.

Appendix Contents

A Evaluation Setup for MCP-Universe and MCP-Bench **13**

B PTC-Bench Construction and Validation **13**

B.1 Simulated APIs 13

B.2 Task Schema 13

B.3 Verifier Design 14

B.4 Failure Injection 15

B.5 Annotation and Validation Process 15

B.6 Neutrality with Respect to I-PTC 15

C Additional Experimental Results **16**

C.1 Run-to-Run Variation in the Main Results 16

C.2 Provider-Native Code Execution 16

C.3 Model Capability and Scale 17

C.4 Backend API Calls 18

A Evaluation Setup for MCP-Universe and MCP-Bench

For MCP-Universe, we report results on the reproducible executable subset available in our setup: 90 tasks from `financial_analysis` and `web_search`. We choose these suites because they do not require private credentials, paid APIs, or specialized infrastructure, while still covering computation-heavy and retrieval-heavy tool use. For MCP-Bench, we evaluate its reproducible credential-free subset: 70 tasks from 23 free MCP servers. Tasks requiring paid or private credentials are excluded to avoid access-dependent variation unrelated to the tool-use interface. We use MCP-Bench's published Overall Score, the mean of six LLM-judged rubric dimensions normalized to $[0, 1]$, so MCP-Bench results are continuous scores rather than pass rates.

B PTC-Bench Construction and Validation

This appendix section provides additional details on the construction and validation of PTC-BENCH. Since PTC-BENCH is designed to evaluate long-horizon, production-style tool-use workflows, it naturally stresses properties such as large initial environments, dependent API calls, multi-turn refinement, and transient failures. We describe how tasks, APIs, verifiers, and failure-injection rules are constructed, and clarify why the benchmark is not tied to I-PTC-specific mechanisms.

B.1 Simulated APIs

PTC-BENCH runs on deterministic local API backends. The benchmark uses ten simulated APIs: `EnterpriseCalendarAPI`, `PersonalCalendarAPI`, `OrgChartAPI`, `YelpAPI`, `GoogleMapsAPI`, `UberAPI`, `DoorDashAPI`, `GroceryDeliveryAPI`, `SpotifyAPI`, and `VenmoAPI`. These APIs cover workflow domains such as calendar scheduling, organizational analysis, restaurant search, route estimation, ride booking, delivery optimization, playlist management, and peer-to-peer payments.

The APIs are deterministic: given the same initial state and call sequence, they return the same results. Some APIs also include state-mutating methods, such as booking a reservation, saving a business, modifying a playlist, or sending a payment. When a task requires state changes, the final backend state is passed to the verifier.

B.2 Task Schema

Each task is represented as a JSON object containing the user prompt, exposed APIs, initial simulated state, and verifier type. A simplified example is shown below.

```
{
  "id": "ptc_org_chart_largest_manager_0",
  "question": [[
    {
      "role": "user",
      "content": "Look through the org chart history and tell me
      which manager has supervised the largest number of different
      employees overall..."
    }
  ]],
  "involved_classes": ["OrgChartAPI"],
  "initial_config": {
    "OrgChartAPI": {
      "employees": [...],
      "reporting_history": [...]
    }
  },
  "verifier_type": "exact_json_match"
}
```

The field `id` uniquely identifies the task. The field `question` stores one or more dialogue turns; single-turn tasks contain one turn, while multi-turn tasks contain multiple turns. The field `involved_classes` specifies which APIs are exposed to the agent. The field `initial_config` contains the full initial state of the simulated backend. The field `verifier_type` selects the verifier used to score the final answer and, when relevant, the final backend state. Error-recovery tasks additionally include a `flakiness_config`, described below.

B.3 Verifier Design

Each task is paired with a deterministic verifier. Verifiers evaluate the model's final answer and, when applicable, the final simulated backend state. They do not inspect the agent's reasoning trace, code, variables, cell history, notebook state, or intermediate tool-call sequence. This ensures that models are evaluated by task outcome rather than by whether their internal execution process matches a particular interface.

We use two main verifier styles.

Exact JSON matching. For tasks with a fully specified output schema, the verifier extracts JSON from the model response and checks each required field against the ground truth. The extractor is tolerant of fenced code blocks and surrounding prose, but the extracted object must satisfy the requested schema and values.

```
def verify(final_answer, final_state, initial_config, ground_truth):
    answer = extract_json_from_text(final_answer)
    if answer is None:
        return VerificationResult(passed=False, score=0.0)

    checks = {
        key: answer.get(key) == expected
        for key, expected in ground_truth.items()
    }

    passed, score = score_from_checks(checks)
    return VerificationResult(
        passed=passed,
        score=score,
        checks=checks
    )
```

Recompute-from-state verification. For tasks whose expected answer depends on the simulated environment, the verifier recomputes the expected result directly from `initial_config`. For example, a restaurant-search verifier may recompute which businesses satisfy cuisine, distance, review-count, open-status, and wait-time constraints, then compare the model's answer by result count,

entity set, ordering, and per-field values. Numeric fields use explicit tolerances when appropriate, such as small tolerances for distance or rating comparisons.

This verifier design gives credit to any model that produces the correct final answer or final backend state, regardless of whether it used direct tool calling, stateless PTC, I-PTC, or provider-native code execution.

B.4 Failure Injection

The error-recovery subset uses deterministic transient failures rather than random failures. Each affected task specifies which API method fails and at which call index. For example:

```
"flakiness_config": {
  "VenmoAPI": {
    "get_transaction_history": 1
  }
}
```

This means that the first call to `get_transaction_history` raises a retryable internal error, while the next call succeeds. The failure is deterministic, recoverable, and local to the task instance. It does not persist across tasks.

From the model's perspective, a flaky API is not presented differently from a normal API. The prompt does not mention that a failure will occur, the API documentation does not reveal that the backend is flaky, and the model does not receive any special hint that a task belongs to the error-recovery subset. The only signal available to the model is the runtime error returned by the API call itself, which follows the same structured exception format as other API errors. Thus, the model must infer whether retrying is appropriate from the observed error, rather than from benchmark-specific metadata.

The purpose of this subset is to evaluate whether an agent can recover from production-style transient failures, not whether it can handle permanently broken APIs. Any agent interface with reliable retry behavior can benefit from this subset; the mechanism is not specific to I-PTC.

B.5 Annotation and Validation Process

Tasks were constructed and validated through a multi-stage process. First, we manually authored natural-language queries to reflect production-level user expectations and realistic daily workflow needs. We then constructed the corresponding simulated initial states and expected answers, ensuring that each query uniquely specified the intended result, including tie-breakers and output format when necessary.

Second, we implemented a verifier for each task with respect to its expected answer. The verifier either checks a fixed expected JSON object or recomputes the expected result from the initial simulated state. This keeps scoring independent of the agent's trace.

Third, every task was manually reviewed by two authors. Reviewers checked that the prompt was unambiguous, the initial state was internally consistent, the verifier matched the prompt, and the expected answer did not depend on unspecified assumptions such as missing tie-breakers, hidden time-zone conventions, or ambiguous default behavior. Disagreements were resolved by editing the prompt, state, or verifier until both reviewers agreed.

Fourth, tasks were executed end-to-end with strong baseline models to identify verifier bugs, impossible tasks, or tasks that were trivially solved under all interaction modes. Such tasks were revised before inclusion. For the error-recovery subset, failure locations were selected so that the injected failure occurs on a method required by a correct solution and can be recovered by retrying.

B.6 Neutrality with Respect to I-PTC

We acknowledge that PTC-BENCH is motivated by the same limitations that I-PTC addresses: long-horizon workflows, large intermediate states, repeated API calls, multi-turn reuse, and transient failures. However, the benchmark is not defined in terms of I-PTC-specific mechanisms.

First, the task prompts do not mention Python, notebooks, persistent cells, variables, or code execution. The tasks are written as natural-language workflow requests over simulated APIs.

Second, all interaction modes are evaluated on the same prompts, same initial states, same API implementations, and same verifiers. The only difference across modes is the interface through which the model interacts with the tools.

Third, verifiers do not inspect I-PTC-specific artifacts. They evaluate only the final answer, final backend state, initial configuration, and ground truth.

Fourth, the multi-turn and error-recovery subsets reward general agent capabilities rather than a specific implementation. Multi-turn tasks benefit any agent that can preserve or reconstruct useful intermediate information, including memory, caching, retrieval, or explicit recomputation. Error-recovery tasks benefit any agent that can recognize a retryable transient failure and issue a successful retry.

Therefore, PTC-BENCH should be understood as a benchmark for long-horizon, stateful, and failure-prone tool use. I-PTC is one interface designed for this setting, but the benchmark itself can evaluate any tool-use system that interacts with the same APIs and verifiers.

C Additional Experimental Results

This section reports additional experiments. Unless noted otherwise, results are averaged over three independent end-to-end runs on the same task sets, and $\pm$ denotes the standard deviation across the three run-level averages.

C.1 Run-to-Run Variation in the Main Results

Table 7 reports the standard deviation of every entry in Table 2. Each run's accuracy and per-task token averages are pooled over all 240 tasks, and the standard deviation is taken across the three run-level values.

Table 7: **Standard deviation across three runs** for each entry of Table 2. The last row averages the per-model standard deviations.

Model	Accuracy			Raw Tokens			Trajectory Tokens		
	Tool-only	PTC	I-PTC	Tool-only	PTC	I-PTC	Tool-only	PTC	I-PTC
GPT 5.4	0.2	1.8	2.5	1.6K	0.7K	0.7K	0.13K	0.31K	0.12K
GPT 5.4 Mini	0.5	1.7	1.8	2.4K	2.1K	0.4K	0.13K	0.08K	0.08K
GPT 5.4 Nano	0.2	2.2	1.0	2.6K	2.5K	4.4K	0.25K	0.12K	0.36K
Gemini 3.1 Pro	2.2	0.6	0.4	5.0K	3.3K	4.0K	0.16K	0.50K	0.35K
Gemini 3.1 Flash-Lite	0.8	2.1	1.1	6.4K	6.4K	5.0K	0.78K	0.42K	0.43K
Gemini 2.5 Pro	2.8	0.1	0.4	4.5K	2.6K	0.6K	0.23K	0.18K	0.07K
Claude Opus 4.7	7.1	1.4	1.5	26.9K	2.1K	2.5K	1.89K	0.11K	0.05K
Claude Sonnet 4.6	1.0	0.6	0.5	2.2K	1.1K	2.9K	0.16K	0.32K	0.05K
Qwen 3.5-397B-A17B	0.6	0.9	0.6	5.9K	7.3K	3.3K	0.19K	0.46K	0.24K
Kimi K2.5†	0.9	2.6	2.2	4.1K	4.1K	3.6K	0.16K	0.27K	0.24K
Mean (10 models)	1.6	1.4	1.2	6.2K	3.2K	2.7K	0.41K	0.28K	0.20K

C.2 Provider-Native Code Execution

Besides Anthropic's built-in code execution tool, we evaluate provider-native code execution for GPT and Gemini, in which model-written code calls the benchmark APIs directly. For GPT, we use Codex's code-only mode, an experimental feature at the time of our experiments (July 2026): the model can only write JavaScript, which runs in a fresh sandbox per invocation, and the benchmark APIs are exposed as MCP servers callable only from that code. At that time, Gemini offered no equivalent runtime. Gemini CLI (v0.53.0) supported tail tool calls, in which a tool's result can trigger another tool call whose result replaces it, but no interpreter used them, so we implemented a minimal Python interpreter on this mechanism through the CLI's documented hook interface. In both cases, intermediate API results stay inside the code rather than entering the model's context. We replace

the provider's system prompt with our `native_ptc` instruction, disable shell and web tools, reject API calls issued directly by the model so that the APIs are reachable only from code, keep no state between invocations, and score with the same verifiers. An audit of all traces found no access to scenario files, answer keys, or credentials. The GPT and Gemini results are single runs over all 240 tasks, since each run requires a full CLI harness; Claude results come from the runs of Table 2.

Provider-native code execution outperforms traditional tool calling on every model (Table 8). I-PTC is more accurate than it on GPT-5.4 and on all three Gemini models, by 7.5–13.8 points on Gemini, and matches it on both Claude models (82.9 ± 1.5 vs. 82.6 ± 0.5 on Opus 4.7, 80.6 ± 0.5 vs. 80.0 ± 4.1 on Sonnet 4.6). Codex exceeds the Table 2 I-PTC results on GPT-5.4 mini and nano; in a matched single run on GPT-5.4 mini with I-PTC's documentation learner disabled, the two are statistically tied (73.6% vs. 74.2%, exact McNemar $p = 1.0$).

Provider-native code execution is also more expensive (Table 9), using 2.5–$7.5\times$ more raw tokens, 2.9–$4.8\times$ more trajectory tokens, and, where measured, 3.6–$3.9\times$ more wall-clock time. The overhead comes from orchestration, not computation: each Gemini API call is a full CLI round trip (115–176 hook process launches per task, against at most 0.4 s of interpreter execution), and Codex re-sends its harness context on every model turn, whereas I-PTC runs the program inside one persistent environment.

The GPT comparison is not compute-matched. Codex runs on the Responses API with reasoning enabled and used about 2.4K reasoning tokens per task, whereas the chat-completions function-tool endpoint used for our GPT runs did not allow reasoning alongside function tools. The GPT-5.4 mini and nano differences may therefore partly reflect unequal inference budgets rather than the interface.

For completeness, Table 8 also reports a second kind of provider code execution, which we call hosted interpreters: OpenAI's `code_interpreter` tool and Gemini's `code_execution` tool. A hosted interpreter runs model-written code on the provider's servers, but that code cannot call any tools. The model must therefore still call each benchmark API itself, one tool call at a time, and can use the interpreter only to compute over results it has already received, for example by copying them into the code. This is traditional tool calling with an added code sandbox for computation, not PTC, where the code itself calls the tools and intermediate results never reach the model. Averaged over three runs, hosted interpreters stay near traditional tool calling on the GPT models and fall below it on Gemini 3.1 Pro (51.7% vs. 67.9%), helping only on Gemini 3.1 Flash-Lite (39.4% vs. 10.8%).

Table 8: **Provider-native code execution** (pass rate, %). Provider-native: Codex code-only mode (GPT) and our interpreter on Gemini CLI tail tool calls (Gemini), single runs; Anthropic's built-in code execution tool (Claude), from the runs of Table 2. Hosted interpreters: mean $\pm$ std over three runs. I-PTC: Table 2.

Model	Hosted interp.	Provider-native	I-PTC
GPT-5.4	17.6 ± 3.6	76.2	**80.0**
GPT-5.4 mini	9.6 ± 1.1	**74.2**	68.0
GPT-5.4 nano	6.7 ± 0.0	**48.3**	46.4
Gemini 3.1 Pro	51.7 ± 1.2	75.4	**82.9**
Gemini 3.1 Flash-Lite	39.4 ± 3.2	$60.8^{\dagger}$	**72.8**
Gemini 2.5 Pro	N/A*	46.2	**60.0**
Claude Opus 4.7	N/A	82.6	**82.9**
Claude Sonnet 4.6	N/A	80.0	**80.6**

*In July 2026, the Gemini API rejected the combined interpreter and function-tool configuration for Gemini 2.5 ("Tool call context circulation is not enabled"). †Run with `gemini-3.1-flash-lite`; other Flash-Lite entries use `gemini-3.1-flash-lite-preview`.

C.3 Model Capability and Scale

Programmatic interaction requires sufficient instruction following, code generation, and tracking of execution state. Table 10 evaluates the Qwen3.5 family from 0.8B to 397B total parameters on all 240 PTC-BENCH tasks, comparing traditional tool calling, one-shot PTC, and raw I-PTC. At 0.8B and 2B, both programmatic interfaces slightly underperform traditional tool calling, suggesting that these models cannot yet reliably generate and execute multi-step tool-use programs. Programmatic calling first becomes beneficial at 4B, where raw I-PTC improves pass rate from 26.9% to 32.8%, and for

Table 9: **Cost of provider-native code execution** vs. I-PTC (per task). I-PTC rows for GPT-5.4 mini and Gemini 3.1 Pro are matched single runs with the documentation learner disabled (73.6% and 82.5% pass). Claude rows come from the runs of Table 2, which did not record wall-clock time.

Runtime	Raw	Traj.	Wall
GPT-5.4 mini, Codex	332K	22.4K	55.5 s
GPT-5.4 mini, I-PTC	**44K**	**7.6K**	**15.5 s**
Gemini 3.1 Pro, Gemini CLI	174K	22.9K	211.2 s
Gemini 3.1 Pro, I-PTC	**61K**	**7.9K**	**53.7 s**
Claude Opus 4.7, built-in	246K	36.2K	–
Claude Opus 4.7, I-PTC	**85K**	**7.6K**	–
Claude Sonnet 4.6, built-in	280K	35.6K	–
Claude Sonnet 4.6, I-PTC	**112K**	**10.7K**	–

every larger model at least one programmatic interface improves substantially over tool calling. In practice, smaller models may benefit from constrained code templates or adaptive routing among tool calling, PTC, and I-PTC.

Table 10: **Qwen3.5 model-scale study** (pass rate, %, all 240 PTC-BENCH tasks). Raw I-PTC uses persistent cells without inspection or documentation updates.

Model	Active params	Tool-only	PTC	Raw I-PTC
Qwen3.5-0.8B	0.8B	$\mathbf{5.3 \pm 1.0}$	4.6 ± 0.4	4.4 ± 1.0
Qwen3.5-2B	2B	$\mathbf{10.3 \pm 1.0}$	8.9 ± 1.2	8.7 ± 1.7
Qwen3.5-4B	4B	26.9 ± 1.7	32.2 ± 0.9	$\mathbf{32.8 \pm 3.0}$
Qwen3.5-9B	9B	32.1 ± 1.0	43.5 ± 1.7	$\mathbf{46.8 \pm 1.0}$
Qwen3.5-27B	27B	62.6 ± 2.1	$\mathbf{79.4 \pm 0.2}$	78.6 ± 1.3
Qwen3.5-35B-A3B	3B	36.7 ± 1.8	66.2 ± 1.7	$\mathbf{67.9 \pm 0.8}$
Qwen3.5-122B-A10B	10B	59.6 ± 0.8	72.2 ± 0.2	$\mathbf{74.4 \pm 2.1}$
Qwen3.5-397B-A17B	17B	43.5 ± 0.6	75.3 ± 0.9	$\mathbf{77.8 \pm 1.0}$

C.4 Backend API Calls

Table 11 reports the mean number of backend API calls per task. Relative to traditional tool calling, programmatic interfaces save tokens because loops, filtering, joins, and aggregation run inside Python: one program or cell can invoke many backend APIs in a single model round without returning every intermediate result to the context. Relative to PTC, I-PTC's further token savings come mainly from on-demand inspection (Table 5), while persistent state reduces redundant backend calls: raw I-PTC makes 19–40% fewer calls than stateless PTC. Tool-only makes the fewest calls, but this reflects early stopping rather than efficiency: for these models it solves only 10–19% of tasks, versus 59–80% for the programmatic interfaces.

Table 11: **Mean backend API calls per task** on all 240 PTC-BENCH tasks.

Model	Tool-only	PTC	Raw I-PTC
GPT-5.4	30.8 ± 1.4	83.0 ± 7.4	67.1 ± 5.2
GPT-5.4 mini	45.0 ± 1.8	135.1 ± 32.3	81.6 ± 7.3
Gemini 3.1 Flash-Lite	26.2 ± 1.5	114.9 ± 8.1	84.8 ± 7.2

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction state the paper's main contributions: I-PTC, a stateful programmatic tool-calling framework, and PTC-BENCH, a benchmark for long-horizon tool-use workflows. The claims are limited to the evaluated settings and are supported by the experimental results on PTC-BENCH and the MCP-based benchmark subsets.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Limitations are discussed in Section 6.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: No theoretical results are included in this work.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The paper results are reproducible using the public code repository and dataset linked on the first page.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: Code and data are publicly available at the repository and dataset linked on the first page, with instructions to reproduce the main results.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: The paper specifies the evaluated task subsets, model families, interface variants, evaluation metrics, and benchmark-specific settings in Sections 4–5 and Appendices A and B. Since I-PTC is training-free, no optimizer or training hyperparameters are used; the main experimental hyperparameter is the API-learning batch size, which is reported in the experiments.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Error bars are included in the figures. Unless noted otherwise, results are averaged over three independent runs, and $\pm$ denotes the standard deviation across the three run-level averages (Section 5 and Appendix C). The GPT and Gemini provider-native code execution results in Appendix C.2 are single runs, as stated there.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.

- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: All experiments are run by sending API requests to model providers and executing simulated API backends locally; no model training or GPU-based optimization is performed. The main compute requirements are therefore API access, CPU execution for the local benchmark backend and verifiers, and wall-clock time proportional to the number of model calls.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research conforms with the NeurIPS Code of Ethics: it uses simulated API environments and benchmark tasks, does not involve human subjects or private personal data, and is released with reproducibility resources to support external scrutiny.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [No]

    Justification: The paper does not include a dedicated broader-impact discussion. The work is evaluated only in simulated API environments and does not involve human subjects, private user data, or real-world deployment.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: This paper does not release high-risk assets. The released assets are limited to benchmark tasks, simulated API backends, harness framework, and evaluation code.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

Justification: We use MCP-Bench and MCP-Universe as existing benchmark assets for transferability experiments. We cite the original papers and repositories and follow their released terms of use. Both assets are released under the Apache License 2.0.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: The dataset introduced in this paper is released on Hugging Face and the code in a public GitHub repository, both linked on the first page, with documentation needed to reproduce our results.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A]

Justification: This paper does not involve crowdsourcing nor research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: This paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: Yes. We studied LLMs as the primary experimental subjects in this work: all experiments evaluate different LLMs under alternative tool-use frameworks. Beyond this, LLMs are only used for writing and editing purposes that do not impact core methodology, scientific rigor, and originality of the research.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.