PROGAGENT: LEARNING TOOL ORCHESTRATION THROUGH PROGRAMMATIC TRAINING

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) are increasingly expected to orchestrate diverse tools over long-horizon tasks. Existing agents typically use Direct Tool Calling (DTC), where the model invokes tools sequentially, and tool responses accumulate in the model context, which can lead to context rot and orchestration breakdown. In contrast, Programmatic Tool Calling (PTC) invokes tools through executable code and stores intermediate results in runtime variables rather than appending them to the context. Prior work has explored PTC mainly as an inference-time optimization, but its benefits for agentic training remain unclear, and suitable verifiable training tasks are scarce. In this work, we introduce ProgAgent, a full PTC training stack, comprising a pipeline for synthesizing verifiable long-horizon tasks, a lightweight PTC runtime, and a two-stage training recipe. Using only 215 tasks instantiated at four environment scales, ProgAgent yields higher pass rates than the evaluated baselines of comparable size across four benchmarks. On LOCA-bench, ProgAgent-14B achieves a 30.7% pass rate at a 64k-token environment state, where these baselines score zero. Cross-harness experiments show that PTC-trained models outperform DTC-trained models under both evaluation interfaces, suggesting that the benefits of PTC training extend beyond the interface. We will release the model checkpoints and task synthesis pipeline.

1 INTRODUCTION

Large language models (LLMs) increasingly excel in deep research (OpenAI, 2025; Google, 2025) and software engineering (Jimenez et al., 2024; Lin, 2026). However, they struggle in long-horizon open-world tool environments (Li et al., 2025a). Agents relying on *Direct Tool Calling* (DTC) invoke tools across multiple turns, maintaining execution logic and tool results in context. As raw tool outputs accumulate in the model context, they can contribute to context rot (Lee, 2025; Chroma, 2025) and orchestration breakdown (Zeng et al., 2026).

Complex tool orchestration often requires data transformation, conditional branching, and error handling. While difficult to express through sequential tool calls, these structures are native to programming languages. *Programmatic Tool Calling* (PTC) provides an interface for applying models' coding abilities to tool orchestration (Anthropic, 2025b). Under PTC, the model generates code that invokes tools and processes intermediate results within a sandbox, storing tool returns in variables and exposing explicitly printed outputs to the model context, as shown in Figure 2.

While PTC has mainly been explored as an inference-time optimization (Anthropic, 2025b; Karten et al., 2026), we argue that its advantages make it an effective training paradigm. By representing tool orchestration as code, PTC may enable foundation models to transfer their pre-trained coding capabilities to complex tool-use tasks. PTC can also limit context growth by retaining intermediate results in runtime variables and selectively printing outputs. We discuss these advantages in §2.3.

Despite this potential, three gaps limit systematic PTC training for long-horizon tasks. *First*, publicly available tasks designed for PTC training remain scarce. Training agents for programmatic orchestration requires long-horizon tasks involving diverse tools and verifiable rewards. *Second*, open-source harnesses (Shi et al., 2026; Karten et al., 2026) embed PTC within general-purpose agent systems. Controlled studies of PTC training require a runtime that preserves execution state across turns and isolates episodes during RL rollouts. *Third*, the benefits of PTC as a training paradigm remain unclear. Existing inference-time evaluations (Anthropic, 2025b) do not establish

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.

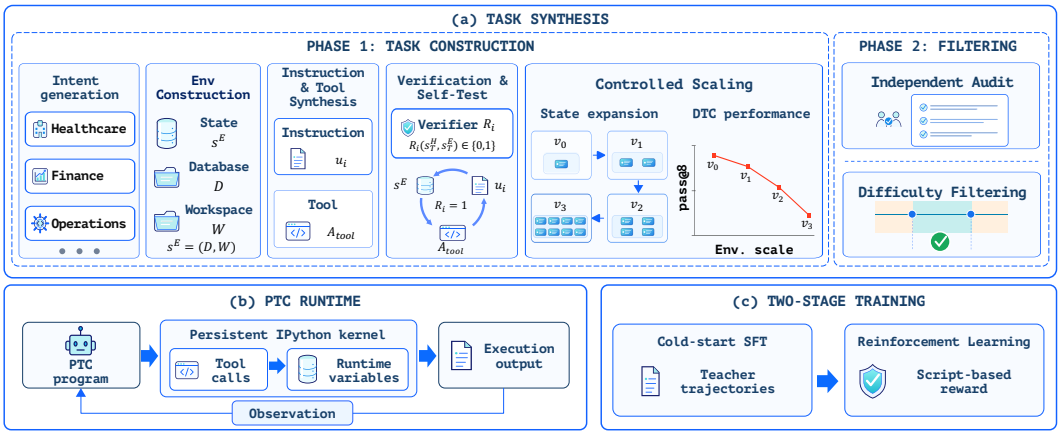

Figure 1: **Overview of ProgAgent. (a) Task synthesis.** We construct verifiable tasks, scale their environment states from v_0 to v_3 while keeping instructions, task logic, and tool schemas fixed, and apply independent auditing and difficulty filtering. The scaling plot shows DeepSeek V4 Pro's declining DTC pass@8 as environment size increases. **(b) PTC runtime.** A persistent IPython kernel executes tool-calling programs, retains intermediate results in variables across turns, and returns printed outputs to the agent. **(c) Two-stage training.** Cold-start SFT on teacher trajectories is followed by RL with binary rewards from Python task verifiers.

whether training agents to use PTC improves tool orchestration or whether the capabilities learned under PTC transfer to DTC.

In this paper, we introduce ProgAgent, a training stack for studying PTC as a training paradigm for long-horizon tool orchestration (Figure 1). We synthesize verifiable long-horizon tasks, scale their environment states to control task difficulty, and pair each task with a Python verifier that returns a binary reward (§3.1). To support reinforcement learning (RL) in these environments, we build a lightweight PTC runtime that preserves execution state across turns and isolates episodes during rollouts (§3.2). Using these environments and the runtime, we train models through cold-start supervised fine-tuning (SFT) followed by RL (§3.3).

We evaluate ProgAgent on four benchmarks using 8B and 14B models trained on 215 tasks instantiated at four environment scales (§4.2). Although the base models perform poorly under PTC, training with ProgAgent improves performance across all four benchmarks. On MCPMark (Wu et al., 2025) and Toolathlon-Verified (Li et al., 2025a), our 8B and 14B models outperform the evaluated baselines of comparable size. On LOCA-bench (Zeng et al., 2026), ProgAgent-14B achieves 30.7% at an environment-state size of 64k tokens, where the comparable-size baselines score 0%. On MCP-Atlas (Bandi et al., 2026), ProgAgent-14B performs comparably to Claude Sonnet-4.5 evaluated under DTC. PTC achieves higher pass rates than Hybrid (PTC+DTC) training, suggesting that a uniform programmatic action space may benefit tool orchestration. Cross-harness evaluations also show that PTC-trained models outperform DTC-trained models under both interfaces, suggesting that the benefits of PTC training extend to direct tool calling (§5.3).

2 Reframing Tool Use as Code

2.1 Multi-Turn Agentic Interaction as a Partially Observable Markov Decision Process

We formulate multi-turn agentic interaction with external environments as a Partially Observable Markov Decision Process (POMDP) (Åström, 1965), following AgentSkiller (Sun et al., 2026b). A POMDP is defined by a tuple

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{O}, P, R),$$

where $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{O}$ are the state, action, and observation spaces, respectively, and P is the transition function. In our setting, R is a programmatic verifier that returns a binary reward at the end of an episode. At each turn t, the agent receives an observation $o_t \in \mathcal{O}$ rather than the full state $s_t \in \mathcal{S}$

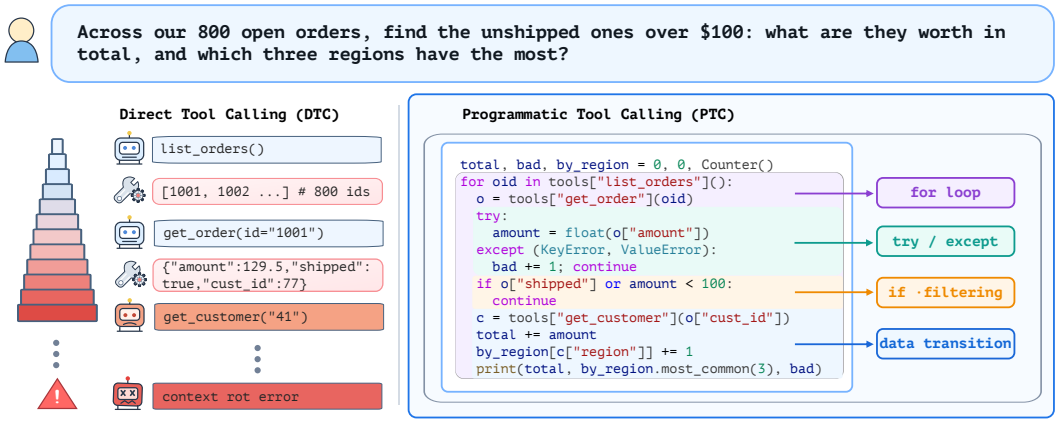

Figure 2: Direct tool calling (DTC) versus programmatic tool calling (PTC). DTC accumulates raw observations in the context, whereas PTC orchestrates tools via code and selectively prints outputs.

and takes an action $a_t \in \mathcal{A}$. The environment transitions to s_{t+1} and returns an observation o_{t+1}. In our setting, the state decomposes into an environment component and an interaction component:

$$s_t = (s_t^E, s_t^H),$$

where the environment state s_t^E contains databases, workspace files, and executable tools, and s_t^H tracks the agent's interaction history, including task instructions, returned observations, and reasoning context. The action space partitions into tool actions and natural-language responses:

$$\mathcal{A} = \mathcal{A}_{\text{tool}} \cup \mathcal{A}_{\text{resp}}.$$

Tool actions may modify s_t^E. Agent actions and observations are appended to the interaction history s_t^H. At the end of an episode of T turns, the verifier returns a reward $R(s_T^H, s_T^E) \in \{0, 1\}$.

2.2 From Direct Tool Calling to Programmatic Tool Calling

Within this framework, we distinguish between two harnesses: direct tool calling (DTC) and programmatic tool calling (PTC) (Figure 2).

Direct tool calling (DTC). As shown in Figure 2 (left), under DTC, the policy π generates an action $a_t \in \mathcal{A}_{\text{tool}}$ at each turn t, and the environment transitions as $s_t^E \xrightarrow{a_t} s_{t+1}^E$. The action a_t and its returned observation o_{t+1} are appended to the interaction context: $s_{t+1}^H = s_t^H \oplus a_t \oplus o_{t+1}$, where $\oplus$ denotes concatenation. Task control flow is expressed through a sequence of tool calls, with execution logic and tool results maintained in context. The context grows as $|s_T^H| = |s_0^H| + \sum_{t=0}^{T-1}(|a_t| + |o_{t+1}|)$. As raw tool outputs accumulate in context, they can contribute to context rot (Lee, 2025; Chroma, 2025) and orchestration breakdown (Li et al., 2025a).

Programmatic tool calling (PTC). As shown in Figure 2 (right), under PTC, the policy π generates a program $\rho_t \sim \pi(\cdot \mid s_t^H)$ that runs inside a sandbox. The program ρ_t invokes tools through calls $a_{t,1}, a_{t,2}, \ldots$ and can include control flow, data transformation, and exception handling. Its execution induces the environment transition $s_t^E \xrightarrow{\rho_t} s_{t+1}^E$. Intermediate tool results are stored in runtime variables and retained across turns. Only the program and its printed output r_{ρ_t} are appended to the interaction context: $s_{t+1}^H = s_t^H \oplus \rho_t \oplus r_{\rho_t}$. By selectively printing outputs, PTC can limit the number of tool-output tokens entering the context (§5.1).

2.3 PTC as an Effective Training Paradigm

Beyond inference efficiency, PTC offers two advantages that may make it an effective training paradigm. First, PTC may enable models to transfer their pre-trained coding capabilities to agentic tasks. During pre-training, foundation models learn to use control flow, data transformation, and

error handling in code, which are also useful for tool orchestration. With environment tools exposed as callable functions, the generated program ρ_t can use these structures to coordinate multiple interrelated tool calls within a single turn. Second, PTC provides a mechanism for context management. By retaining intermediate results in runtime variables and selectively printing outputs r_{ρ_t} to the interaction context s_t^H, PTC can limit context growth. In this work, we explore whether this paradigm enables models to achieve better agentic performance with fewer training tasks. We also investigate whether models can learn this context filtering behavior during training to mitigate context rot.

3 PROGAGENT: FROM TASK SYNTHESIS TO AGENTIC TRAINING

We build ProgAgent, a training stack with three components: a task synthesis pipeline for producing PTC-tailored, verifiable long-horizon tasks with controllable difficulty (§3.1); a lightweight PTC runtime that uses a persistent IPython kernel to preserve execution state across turns and Bubblewrap (Containers contributors, 2025) for sandbox isolation (§3.2); and a training recipe consisting of cold-start SFT followed by RL to improve long-horizon tool orchestration (§3.3).

3.1 SYNTHESIZING VERIFIABLE LONG-HORIZON AGENTIC TASKS

3.1.1 DESIGN PRINCIPLES

Long-Horizon Environment State Scaling. Existing environment synthesis pipelines (Song et al., 2026; Dong et al., 2026) primarily make tasks harder by increasing tool diversity or orchestration complexity. However, these changes do not provide direct control over environment-state size and task difficulty, limiting their ability to optimize model capabilities and behavior under context rot. Inspired by the concept of description length (Hutter, 2000; Legg & Hutter, 2007; Zeng et al., 2026), we hold the task instruction, underlying logic, and tool schemas constant while scaling the environment state at each discrete level j ($|s^{E,(j+1)}| \approx 2 \cdot |s^{E,(j)}|$). As s^E grows, the agent must retrieve and process information from a larger environment. Under DTC, the resulting tool observations are appended directly to the interaction context s^H. As the environment scales from v_0 to v_3, DeepSeek V4 Pro's (DeepSeek-AI, 2026) pass@8 on our tasks drops from 0.95 to 0.42 under DTC (Figure 1). Under PTC, its pass@8 remains above 0.90 across all four scales (Figure 7).

Programmatic Task Design. We design tasks with tool orchestration patterns that can be naturally expressed through programs. Specifically, we distribute environment data across related tables in a relational database and place task constraints in unstructured files. Solving these tasks involves interpreting these constraints and combining information from several sources. Loops and conditional branching can help coordinate tool calls during multi-hop aggregation.

Rigorous Task Verifiability. We pair each synthesized task i with a programmatic verifier $R_i(s_T^H, s_T^E) \in \{0, 1\}$. Implemented as a Python script, the verifier checks the final response and final environment state against the task's evaluation criteria, returning 1 if and only if all criteria are satisfied. We verify task solvability by requiring an oracle solution to receive full reward using only the provided tools. If the oracle fails verification, an automated repair loop revises the task environment, tools, instruction, or verifier until the task passes this self-test.

3.1.2 TASK CONSTRUCTION PIPELINE

Each task is built in five stages (Phase 1 of Figure 1). First, an LLM generates user intents from 23 curated domains detailed in Appendix B.1. A constructor agent then builds a relational database D and a file workspace W using scripts with a scaling configuration that controls environment-state size. Next, this agent synthesizes the action space $\mathcal{A}_{\text{tool}}$ as a set of single-purpose tools whose Python implementations are hidden from the task-solving agent, together with an instruction u_i that specifies the task goals and required outputs. It then generates a verifier R_i and an oracle solution, iteratively repairing the task until the oracle solution receives full reward. Finally, the agent adjusts the scaling configuration to generate four variants $\mathcal{M}_i^{(0)}, \ldots, \mathcal{M}_i^{(3)}$ (denoted v_0–v_3), increasing the environment-state size $|s^{E,(j)}|$ at each level while keeping the task instruction, underlying logic, and tool schemas fixed. We provide more details in Appendix B.2.

3.1.3 VERIFICATION-BASED FILTERING

After oracle verification, we audit each task for consistency and filter tasks by difficulty (Phase 2 of Figure 1; details in Appendix B.3). First, an independent harness agent checks that the tools, environment state, and verifier are consistent with the task instruction. Tasks with identified issues are repaired and audited again. Next, for each task and each scale, we sample $K = 8$ trajectories from DeepSeek V4 Pro under the PTC harness. We retain a task only if its pass rate lies in $(0, 0.8)$ at every scale, selecting tasks that remain challenging but solvable. Finally, each surviving task $\mathcal{M}_i$ is packaged into three core artifacts: a data generator, an environment module, and an instruction u_i.

3.2 A LIGHTWEIGHT PTC RUNTIME

Following LOCA-bench (Zeng et al., 2026), we expose the PTC runtime to the model as an MCP server. At the start of each episode, we run the data generator at a target scale j to initialize the environment state $s_0^E = (D^{(j)}, W^{(j)})$ and provide the model with the instruction u_i. The runtime uses a persistent, sandboxed IPython kernel. A tool proxy routes calls from each generated program ρ_t to the corresponding environment tools in $\mathcal{A}_{\text{tool}}$ and returns their results as native Python objects. The kernel executes ρ_t, returns its printed output as r_{ρ_t}, and preserves runtime variables across turns. The sandbox prevents the agent from accessing the verifier implementation and oracle solution (Appendix C.2). At the end of each episode, the verifier $R_i(s_T^H, s_T^E)$ evaluates the final agent response and environment state to produce a binary reward.

3.3 TRAINING WITH PROGAGENT

After filtering, the synthesis pipeline retains 215 tasks, each instantiated at the four environment scales (v_0–v_3) defined in §3.1.2. Training proceeds in two stages: cold-start SFT on teacher trajectories, followed by RL via GSPO (Zheng et al., 2025a;b). At the end of each rollout, the verifier $R_i(s_T^H, s_T^E)$ evaluates the final response and environment state to return a binary reward. We apply this training recipe across three runtime harnesses: (1) **DTC**: exposes environment tools through standard schemas for direct calls; (2) **PTC**: requires all environment tool calls to go through the Python sandbox; (3) **Hybrid**: exposes both interfaces, allowing the model to interleave direct calls and program executions. To isolate the effect of the harness, we use the same task pool and training hyperparameters across all three settings (details in §4.1 and Appendix C.3).

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Synthesis and Data Collection. We generate task intents using Gemini 3.0 Flash (Google, 2025), and the constructor and harness agents rely on Claude Opus 4.6 (Anthropic, 2026) via Claude Code. For environment difficulty filtering, we sample 8 DeepSeek V4 Pro (DeepSeek-AI, 2026) trajectories per candidate task at each scale (v_0–v_3) under PTC, retaining only tasks whose pass rate lies in $(0, 0.8)$ at every scale. The resulting 215 tasks are each instantiated at 4 scales, yielding a shared training pool of 860 task instances. Appendix B.1 shows the details of task distribution.

Training Settings & Details. We train Qwen3 (Team, 2025) (8B and 14B) across the DTC, PTC, and Hybrid settings and evaluate each model using its training harness. Models are first fine-tuned on the cold-start datasets for 3 epochs (batch size 64), then trained with GSPO (Zheng et al., 2025b) for 150 steps, sampling 16 rollouts per task for 32 tasks at each step (full details in Appendix C.3).

Benchmarks & Baselines. We evaluate our models across three agentic benchmarks: MCP-Mark (Wu et al., 2025), MCP-Atlas (Bandi et al., 2026), and Toolathlon-Verified (Li et al., 2025a). To stress-test robustness under extreme environment context growth, we further use LOCA-bench (Zeng et al., 2026) at 8k–64k environment-state sizes, a benchmark for evaluating long-horizon agents in realistic settings. Each reported pass rate averages four independent runs. Our baselines include frontier proprietary models (OpenAI, 2025; Anthropic, 2025a; Google, 2025), open-source foundation models (Team, 2025; DeepSeek-AI, 2025), and six trained baselines sum-

Table 1: **Training-data scales across baselines** (traj.: trajectories). Simulator aggregates synthesis settings; EnvScaler counts SFT + RL samples (Xu et al., 2026).

Method	TOUCAN (Xu et al., 2025)	Simulator (Li et al., 2025b)	EnvScaler (Song et al., 2026)	AWM (Wang et al., 2026)	Agent-World (Dong et al., 2026)	EnvFactory (Xu et al., 2026)	**ProgAgent** (ours)
Tasks	1.53M traj.	135k traj.	11,572	3,315	2,000+	2,575	215×4

Table 2: **Main results.** Mean pass rates (%) on four benchmarks. Best and second-best scores within each 8B/14B comparison are bolded and underlined. Evaluation protocols and sources of reported scores are detailed in §4.1 and Appendix C.4.

Model	**MCPMark**					**LOCA-bench**					**Toolathlon**	**MCP-Atlas**
	File.	Notion	Play.	Post.	**Avg.**	8k	16k	32k	64k	**Avg.**		
Frontier Proprietary Models												
GPT-5.2	60.0	42.9	40.0	66.7	52.4	72.0	70.7	60.0	52.0	63.7	41.7†	67.6
Claude Sonnet-4.5	32.5	25.0	27.0	50.0	33.6	58.7	60.0	36.0	26.7	45.3	38.9†	43.8
Gemini-3 Pro	56.7	43.8	40.0	70.2	52.7	26.7	22.7	17.3	9.3	19.0	36.4†	54.1
Open-Source Foundation Models												
Qwen3-8B	3.3	0.0	4.0	4.8	3.0	1.3	0.0	0.0	0.0	0.3	2.8	7.6
Qwen3-14B	3.3	0.0	0.0	9.5	3.2	1.3	0.0	0.0	0.0	0.3	1.9	12.4
Qwen3-235B-A22B	13.3	10.7	0.0	4.8	7.2	2.7	1.3	0.0	0.0	1.0	3.7	12.0
DeepSeek-V3.2	36.7	45.5	17.0	66.6	41.5	78.7	80.0	61.3	45.3	66.3	35.2†	56.2
Open-Source Baselines												
TOUCAN-7B	0.0	0.0	0.0	4.8	1.2	0.0	0.0	0.0	0.0	0.0	0.0	7.0
Simulator-8B	3.3	0.0	4.0	4.8	3.0	0.0	0.0	0.0	0.0	0.0	0.0	0.8
AWM-8B	3.3	0.0	4.0	4.8	3.0	0.0	0.0	0.0	0.0	0.0	1.9	8.0
EnvFactory-8B	10.0	0.0	4.0	9.5	5.9	2.7	1.3	0.0	0.0	1.0	1.9	11.8
EnvScaler-8B	10.0	0.0	4.0	9.5	5.9	1.3	0.0	0.0	0.0	0.3	2.8	13.0
Agent-World-8B	13.3	3.6	4.0	19.1	10.0	–	–	–	–	–	–	–
AWM-14B	3.3	0.0	4.0	9.5	4.2	5.3	2.7	0.0	0.0	2.0	1.9	11.0
Agent-World-14B	16.6	3.6	4.0	38.1	15.6	–	–	–	–	–	–	–
Our Method												
ProgAgent-8B	**33.3**	**14.3**	4.0	**21.4**	**18.3**	**34.7**	**24.0**	**21.3**	**17.3**	**24.3**	**9.3**	**38.6**
ProgAgent-14B	**46.7**	**25.0**	4.0	**42.9**	**29.7**	**41.3**	**40.0**	**41.3**	**30.7**	**38.3**	**17.1**	**45.6**

marized in Table 1. In Table 2, we evaluate baselines under the DTC harness. Additional evaluations of selected baselines under the PTC harness are reported in Table 7.

4.2 Main Results

As Table 2 shows, ProgAgent outperforms comparable baselines across diverse agentic benchmarks. On MCPMark, ProgAgent achieves average pass rates of 18.3% at 8B and 29.7% at 14B, respectively. On Toolathlon-Verified, ProgAgent reaches 9.3% at 8B and 17.1% at 14B, compared with at most 2.8% and 1.9% among the 8B and 14B baselines. Moreover, on MCP-Atlas, ProgAgent-14B achieves 45.6%, comparable to Claude Sonnet-4.5 (43.8%) under the DTC harness. The gap is most pronounced on LOCA-bench: the evaluated comparable-size baselines score zero at 32k and 64k, while ProgAgent mitigates context rot to maintain pass rates between 17.3% and 30.7% at 64k.

Table 3 compares each training recipe with its untrained backbone under the same harness. For untrained backbones, switching to PTC provides limited benefits across benchmarks and often yields the lowest scores among the three harnesses. Training reverses this pattern: after SFT and RL, the PTC recipe achieves the highest aggregate score on each benchmark at both model scales. ProgAgent-14B reaches 29.7% on MCPMark and 45.6% on MCP-Atlas, compared with 8.0% and 29.7% for DTC, respectively. ProgAgent-8B also outperforms the 8B DTC model on all four benchmarks. Cross-harness evaluation on LOCA-bench and MCP-Atlas (Table 6 in §5.3) further demonstrates that the gains extend beyond the inference interface to the training paradigm itself.

Table 3: **Training paradigms across benchmarks.** Final pass rates (%) with the corresponding base model's scores under the same harness shown in parentheses. Each trained model is evaluated under its training harness. The best score for each benchmark within each backbone is bolded.

Model	Paradigm	MCPMark					LOCA-bench	Toolathlon	MCP-Atlas
		File.	Notion	Play.	Post.	Avg.			
8B	DTC	6.7 (3.3)	3.6 (0.0)	4.0 (4.0)	9.5 (4.8)	6.0 (3.0)	4.0 (0.3)	2.8 (2.8)	19.0 (7.6)
	Hybrid	30.0 (13.3)	5.4 (0.0)	4.0 (0.0)	19.1 (4.8)	14.6 (4.5)	21.7 (0.3)	7.4 (0.9)	30.6 (8.4)
	PTC	33.3 (5.0)	14.3 (0.0)	4.0 (0.0)	21.4 (4.8)	**18.3** (2.5)	**24.3** (0.0)	**9.3** (0.0)	**38.6** (3.6)
14B	DTC	10.0 (3.3)	3.6 (0.0)	4.0 (0.0)	14.3 (9.5)	8.0 (3.2)	9.0 (0.3)	5.6 (1.9)	29.7 (12.4)
	Hybrid	43.3 (3.3)	16.1 (0.0)	4.0 (0.0)	38.1 (9.5)	25.4 (3.2)	31.3 (0.7)	14.7 (3.7)	38.6 (11.4)
	PTC	46.7 (6.7)	25.0 (5.0)	4.0 (0.0)	42.9 (4.8)	**29.7** (4.1)	**38.3** (0.0)	**17.1** (0.9)	**45.6** (3.8)

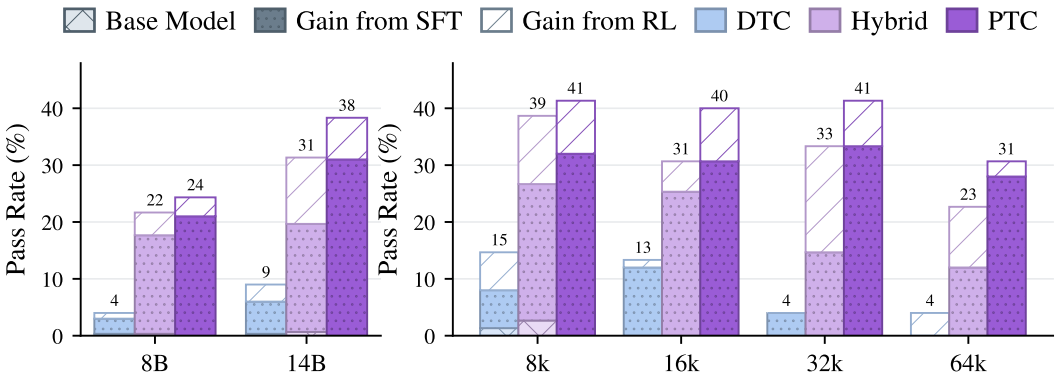

Figure 3: **Training paradigms on LOCA-bench.** Each bar decomposes the final pass rate into the untrained backbone's score and the gains from SFT and RL. Labels above the bars show rounded final pass rates; exact values are reported in Table 7. **Left:** Results for the 8B and 14B backbones, averaged over four environment-state sizes. **Right:** Results for the 14B backbone at each environment-state size, showing PTC's robustness as the environment grows.

Figure 3 compares the three training paradigms on LOCA-bench, showing gains from both SFT and RL. As the environment state scales, DTC suffers from context rot. In contrast, PTC effectively mitigates this degradation and sustains robust performance, achieving average pass rates of 24.3% (8B) and 38.3% (14B). Furthermore, despite providing a more flexible action space during both training and evaluation, the Hybrid paradigm performs worse than PTC. We hypothesize that PTC provides a unified tool interaction interface which lowers learning difficulty and enables the model to learn a more generalized representation. While a similar design is adopted in DeepSeek Harness (Shi et al., 2026), we substantiate it with empirical observations.

5 Analysis

5.1 What Does PTC Change During Training?

To understand how PTC changes agent behavior, we track the RL training dynamics of Qwen3-14B in Figure 4. DTC achieves substantially lower rewards than PTC and Hybrid on our environments. Specifically, DTC's reward increases only from 0.08 to 0.15, whereas the rewards of PTC and Hybrid start near 0.50 and reach 0.70 and 0.72, respectively. For DTC, raw tool outputs account for around 30k context tokens per episode throughout training. Programmatic agents instead express orchestration as executable code, retain intermediate state in sandbox variables, and selectively print results. During RL training, the number of tool-output tokens under PTC decreases by roughly 20%, falling below 20k per episode; both programmatic paradigms also use roughly half as many tool calls as DTC. PTC runtime errors per episode also decrease from 6.4 to 3.6 at 14B and from 7.5 to 5.0 at 8B. The 8B backbone reproduces the same trend (Appendix D.3).

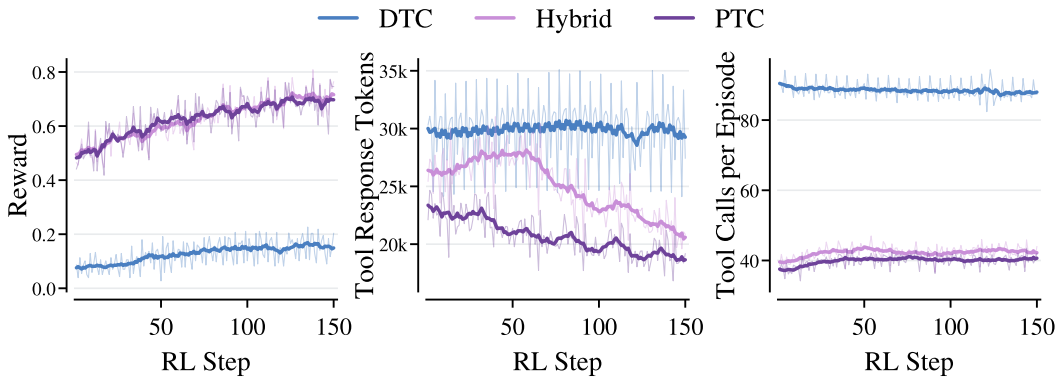

Figure 4: **RL training dynamics at 14B.** Episode reward (left), tool-output tokens entering context (middle), and tool calls per episode (right) under DTC, Hybrid, and PTC.

5.2 Does Environment State Scaling Help?

Table 4: **Environment state and task difficulty.** We present database size (KB), mean workspace file count, and DeepSeek V4 Pro's DTC pass@8 across the four environment scales.

Scale	Database Size (KB)	Workspace Files	pass@8
v_0	99	12.7	0.95
v_1	173	21.9	0.88
v_2	335	31.1	0.65
v_3	711	40.3	0.42

Environment state scaling provides a direct mechanism to control task difficulty for the DTC harness. As the environment state scales from v_0 to v_3, DeepSeek V4 Pro's pass@8 under the DTC harness drops monotonically from 0.95 to 0.42 in Table 4. As shown in Table 7, restricting training from all four scales to the initial environments v_0 reduces average pass rates on LOCA-bench from 24.3% to 16.0% at 8B and from 38.3% to 23.7% at 14B. These results show that environment state scaling improves PTC performance on LOCA-bench across both backbones.

5.3 How Does PTC Facilitate Tool Orchestration?

Table 5: **Programming structures used during RL.** We present the percentage of generated code blocks using loops and conditional execution, and the percentage of executions using error handling.

Model	Loops	Conditional execution	Error handling
8B	82.6	65.6	8.2
14B	82.1	64.0	9.5

As shown in Table 5, PTC models use a range of programming structures, including loops, conditional execution, and error handling, to orchestrate tools during RL. The Toolathlon-Verified example (Appendix E.2) shows how PTC uses these programming structures to coordinate dependent tool calls within a program. The MCP-Atlas example (Appendix E.1) illustrates another use of PTC: aggregating intermediate tool results and selectively printing a summary into the model context.

We next examine whether the benefits of PTC training extend beyond the programmatic interface. To separate training effects from harness effects, we evaluate DTC- and PTC-trained models under both DTC and PTC harnesses on LOCA-bench and MCP-Atlas (Table 6). For DTC-trained models, switching from the DTC to the PTC harness improves performance on LOCA-bench but not on MCP-Atlas. We also evaluate other baselines under the PTC harness. Their pass rates remain low (Table 7), suggesting that access to the PTC interface alone is insufficient for effective orchestration.

Table 6: **Cross-harness evaluation on LOCA-bench and MCP-Atlas.** Pass rates (%) under DTC and PTC evaluation for untrained backbones (Base) and models trained with the DTC or PTC SFT+RL recipe. LOCA-bench scores are averaged over four environment-state sizes.

Model	Eval	LOCA-bench			MCP-Atlas		
		Base	DTC-trained	PTC-trained	Base	DTC-trained	PTC-trained
8B	DTC	0.3	4.0	6.0	7.6	19.0	30.6
	PTC	0.0	10.7	24.3	3.6	11.8	38.6
14B	DTC	0.3	9.0	13.9	12.4	29.7	41.6
	PTC	0.0	20.7	38.3	3.8	29.0	45.6

Under the same PTC harness, PTC-trained models outperform DTC-trained models on both benchmarks. This advantage also holds under DTC evaluation: on MCP-Atlas, the 14B PTC-trained model outperforms its DTC-trained counterpart by 11.9 percentage points. These results suggest that PTC training improves tool orchestration beyond the gains from the PTC harness alone, with benefits that extend to direct tool calling. This advantage persists under v_0-only training (Table 7).

5.4 What Challenges Remain for PTC Agents?

Despite the overall gains in §4.2, our analysis identifies two limitations of the current PTC agents.

DTC-like behavior in complex environments: When environments return unpredictable or highly unstructured content, PTC agents may revert to DTC-like behavior. On Playwright, choosing the next tool call may require interpreting the current snapshot. For ProgAgent-14B, 40.7% of PTC calls contain a single tool invocation without control flow, error handling, or selective printing (Table 8). This behavior limits the benefits of programmatic orchestration and context management.

Silent server failures: The PTC harness of ProgAgent relies on MCP's `isError` field to raise tool exceptions; however, non-compliant servers often omit this field, bypassing the agent's exception-based recovery. Disabling this server-level error detection drops LOCA-bench pass rates from 24.3% to 19.3% at 8B and from 38.3% to 34.5% at 14B (Table 7). Without explicit exceptions, the agent may process error messages as successful tool returns.

6 Conclusion

We introduce ProgAgent, a training stack that combines verifiable tasks at multiple environment scales, a lightweight PTC runtime, and a training recipe comprising cold-start SFT followed by RL. Using 215 tasks at four scales, ProgAgent improves agentic performance across four benchmarks and outperforms the evaluated baselines of comparable size. Our analysis shows that PTC agents selectively print outputs and use programming structures to coordinate tool calls. Cross-harness evaluations show that the benefits of PTC training extend to DTC.

These findings suggest that PTC is useful not only as an inference interface but also as a training paradigm for tool orchestration. The transfer of these gains to DTC motivates further study of how programmatic training shapes tool-use behavior across interfaces. Our task synthesis pipeline and runtime provide a basis for studying this question across environment scales and training settings. We hope ProgAgent will have a positive and lasting impact on long-horizon agent research.

AI Use Statement

We used generative AI tools for language editing, synthetic task construction and auditing, and trajectory generation for filtering and training. The authors reviewed all AI-assisted content and take full responsibility for this work.

REFERENCES

Anthropic. Introducing claude sonnet 4.5. https://www.anthropic.com/news/claude-sonnet-4-5, 2025a.

Anthropic. Introducing advanced tool use on the claude developer platform, November 2025b. URL https://www.anthropic.com/engineering/advanced-tool-use.

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

Anthropic. Claude opus 4.6. https://www.anthropic.com/news/claude-opus-4-6, 2026.

Karl Johan Åström. Optimal control of markov processes with incomplete state information i. *Journal of mathematical analysis and applications*, 10:174–205, 1965.

Chaithanya Bandi, Razvan-Gabriel Dumitru, Ben Hertzberg, Divyansh Agarwal, Geobio Boo, Tejas Polakam, Sami Hassaan, Jeff Da, HiJae Kim, Vipul Gupta, Manasi Sharma, Andrew Park, Martin Dimakis, Ernesto Gabriel Hernandez Montoya, Dan Rambado, Ivan Salazar, Rafael Cruz, MohammadHossein Rezaei, Chetan Rane, Ben Levin, Daniel Yue Zhang, Brad Kenstler, and Bing Liu. Mcp-atlas: A large-scale benchmark for tool-use competency with real mcp servers, 2026. URL https://arxiv.org/abs/2602.00933.

Zhaorun Chen, Zhuokai Zhao, Kai Zhang, Bo Liu, Qi Qi, Yifan Wu, Tarun Kalluri, Xuefei Cao, Yuanhao Xiong, Haibo Tong, et al. Scaling agent learning via experience synthesis. In *International Conference on Learning Representations*, volume 2026, pp. 121394–121420, 2026.

Chroma. Context rot: How increasing input tokens impacts llm performance. Technical report, Chroma, July 2025. Technical report.

Containers contributors. Bubblewrap: Unprivileged sandboxing tool. https://github.com/containers/bubblewrap, 2025.

DeepSeek-AI. Deepseek-v3.2: Pushing the frontier of open large language models, 2025.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

Guanting Dong, Junting Lu, Junjie Huang, Wanjun Zhong, Longxiang Liu, Shijue Huang, Zhenyu Li, Yang Zhao, Xiaoshuai Song, Xiaoxi Li, et al. Agent-world: Scaling real-world environment synthesis for evolving general agent intelligence. *arXiv preprint arXiv:2604.18292*, 2026.

Google. Gemini 3: Introducing the latest gemini AI model from google. https://blog.google/products/gemini/gemini-3/, 2025.

Google. Gemini deep research overview. https://gemini.google/overview/deep-research/, 2025.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments, 2026. URL https://doi.org/10.5281/zenodo.20953922.

Marcus Hutter. A theory of universal artificial intelligence based on algorithmic complexity. *arXiv preprint cs/0004001*, 2000.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Seth Karten, Alex L. Zhang, Kevin Thomas, Sebastian Müller, Elie Bakouch, Daniel Auras, Mika Senghaas, Fares Obeid, Konstantin Dunas, Johannes Hagemann, and Sami Jaghouar. Prime Agent: A self-improving RLM harness. *arXiv preprint arXiv:2608.23552*, 2026.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Timothy B. Lee. Context rot: The emerging challenge that could hold back llm progress. Understanding AI, November 2025. URL https://www.understandingai.org/p/context-rot-the-emerging-challenge. Paid article. Accessed: 2026-01-18.

Shane Legg and Marcus Hutter. Universal intelligence: A definition of machine intelligence. *Minds and machines*, 17(4):391–444, 2007.

Junlong Li, Wenshuo Zhao, Jian Zhao, Weihao Zeng, Haoze Wu, Xiaochen Wang, Rui Ge, Yuxuan Cao, Yuzhen Huang, Wei Liu, Junteng Liu, Zhaochen Su, Yiyang Guo, Fan Zhou, Lueyang Zhang, Juan Michelini, Xingyao Wang, Xiang Yue, Shuyan Zhou, Graham Neubig, and Junxian He. The tool decathlon: Benchmarking language agents for diverse, realistic, and long-horizon task execution. 2025a. URL https://arxiv.org/abs/2510.25726.

Yuetai Li, Huseyin A Inan, Xiang Yue, Wei-Ning Chen, Lukas Wutschitz, Janardhan Kulkarni, Radha Poovendran, Robert Sim, and Saravan Rajmohan. Simulating environments with reasoning models for agent training. *arXiv preprint arXiv:2511.01824*, 2025b.

Wilson Lin. Scaling long-running autonomous coding, January 2026. URL https://cursor.com/blog/scaling-agents.

OpenAI. Deep research system card. https://cdn.openai.com/deep-research-system-card.pdf, 2025.

OpenAI. Introducing GPT-5.2. https://openai.com/index/introducing-gpt-5-2/, 2025.

Ishan Patel, Sahil Sen, Elias Lumer, and Vamse Kumar Subbiah. The bitter lesson of tool calling, 2026. URL https://arxiv.org/abs/2608.06370.

Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. Yarn: Efficient context window extension of large language models. In *International Conference on Learning Representations*, volume 2024, pp. 31932–31951, 2024.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 1279–1297, 2025.

Yifan Shi, Wei Zhang, and Tianyi Cui. A programming paradigm for spatiotemporal composability. *arXiv preprint arXiv:2608.25512*, 2026.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Xiaoshuai Song, Haofei Chang, Guanting Dong, Yutao Zhu, Ji-Rong Wen, and Zhicheng Dou. Envscaler: Scaling tool-interactive environments for llm agent via programmatic synthesis. *arXiv preprint arXiv:2601.05808*, 2026.

Shuang Sun, Huatong Song, Lisheng Huang, Jinhao Jiang, Ran Le, Zhihao Lv, Zongchao Chen, Yiwen Hu, Wenyang Luo, Wayne Xin Zhao, et al. Swe-world: Building software engineering agents in docker-free environments. *arXiv preprint arXiv:2602.03419*, 2026a.

Zexu Sun, Bokai Ji, Hengyi Cai, Shuaiqiang Wang, Lei Wang, Guangxia Li, and Xu Chen. Agentskiller: Scaling generalist agent intelligence through semantically integrated cross-domain data synthesis. *arXiv preprint arXiv:2602.09372*, 2026b.

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Qwen Team. Qwen3.5: Accelerating productivity with native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Dunwei Tu, Hongyan Hao, Hansi Yang, Yihao Chen, Yi-Kai Zhang, Zhikang Xia, Yu Yang, Yueqing Sun, Xingchen Liu, Furao Shen, et al. Scaleenv: Scaling environment synthesis from scratch for generalist interactive tool-use agent training. *arXiv preprint arXiv:2602.06820*, 2026.

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

Zhaoyang Wang, Canwen Xu, Boyi Liu, Yite Wang, Siwei Han, Zhewei Yao, Huaxiu Yao, and Yuxiong He. Agent world model: Infinity synthetic environments for agentic reinforcement learning. *arXiv preprint arXiv:2602.10090*, 2026.

Zijian Wu, Xiangyan Liu, Xinyuan Zhang, Lingjun Chen, Fanqing Meng, Lingxiao Du, Yiran Zhao, Fanshi Zhang, Yaoqi Ye, Jiawei Wang, Zirui Wang, Jinjie Ni, Yufan Yang, Arvin Xu, and Michael Qizhe Shieh. Mcpmark: A benchmark for stress-testing realistic and comprehensive mcp use, 2025. URL https://arxiv.org/abs/2509.24002.

Zikai Xiao, Jianhong Tu, Chuhang Zou, Yuxin Zuo, Zhi Li, Peng Wang, Bowen Yu, Fei Huang, Junyang Lin, and Zuozhu Liu. Webworld: A large-scale world model for web agent training. *arXiv preprint arXiv:2602.14721*, 2026.

Minrui Xu, Zilin Wang, Mengyi DENG, Zhiwei Li, Zhicheng Yang, Xiao Zhu, Yinhong Liu, Boyu Zhu, Baiyu Huang, Chao Chen, Heyuan Deng, Fei Mi, Lifeng Shang, Xingshan Zeng, and Zhijiang Guo. Envfactory: Scaling tool-use agents via executable environments synthesis and robust rl, 2026. URL https://arxiv.org/abs/2605.18703.

Zhangchen Xu, Adriana Meza Soria, Shawn Tan, Anurag Roy, Ashish Sunil Agrawal, Radha Poovendran, and Rameswar Panda. Toucan: Synthesizing 1.5m tool-agentic data from real-world mcp environments. *arXiv preprint arXiv:2510.01179*, 2025.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *Advances in Neural Information Processing Systems*, 38:113222–113244, 2026.

Weihao Zeng, Yuzhen Huang, and Junxian He. Loca-bench: Benchmarking language agents under controllable and extreme context growth. *arXiv preprint arXiv:2602.07962*, 2026.

Alex L. Zhang, Tim Kraska, and Omar Khattab. Recursive language models. *arXiv preprint arXiv:2512.24601*, 2025.

Yuze Zhao, Jintao Huang, Jinghan Hu, Xingjun Wang, Yunlin Mao, Daoze Zhang, Zeyinzi Jiang, Zhikai Wu, Baole Ai, Ang Wang, Wenmeng Zhou, and Yingda Chen. Swift: A scalable lightweight infrastructure for fine-tuning, 2024. URL https://arxiv.org/abs/2408.05517.

Chujie Zheng, Kai Dang, Bowen Yu, Mingze Li, Huiqiang Jiang, Junrong Lin, Yuqiong Liu, Hao Lin, Chencan Wu, Feng Hu, et al. Stabilizing reinforcement learning with llms: Formulation and practices. *arXiv preprint arXiv:2512.01374*, 2025a.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, Jingren Zhou, and Junyang Lin. Group sequence policy optimization. *arXiv preprint arXiv:2507.18071*, 2025b.

Yuxin Zuo, Zikai Xiao, Li Sheng, Fei Huang, Jianhong Tu, Yuxuan Liu, Tianyi Tang, Xiaomeng Hu, Yang Su, Qingfeng Lan, et al. Qwen-agentworld: Language world models for general agents. *arXiv preprint arXiv:2606.24597*, 2026.

A Related Work

Programmatic tool calling. Executable code allows an agent to compose multiple tool calls within a single action (Patel et al., 2026). CodeAct (Wang et al., 2024) uses executable code as the agent's action format and demonstrates improvements over JSON-based tool calls. However, it does not provide verifiable long-horizon tasks, which are crucial for agentic training. Recursive Language Models (Zhang et al., 2025) primarily target long-context processing, storing long inputs in REPL variables and processing selected snippets through code and recursive model calls. In MCP settings, code-execution patterns expose external tools as callable APIs and process intermediate results within a code block to reduce token use and latency (Anthropic, 2025c;b). ProgAgent studies PTC as a training paradigm for long-horizon tool orchestration. We combine verifiable tasks at controlled environment scales with a runtime that isolates episodes and preserves variables across turns, supporting both SFT and RL.

Scalable Environment Synthesis. Recent efforts to scale training environments span simulated and executable settings. Simulated environments use an LLM as a world model to generate tool returns, as in WebWorld (Xiao et al., 2026), SWE-World (Sun et al., 2026a), DreamGym (Chen et al., 2026), and Qwen-AgentWorld (Zuo et al., 2026). Executable environments implement tool behavior through programs, database backends, or finite-state machines, using graph-based task synthesis (Tu et al., 2026; Xu et al., 2026) or programmatic task synthesis (Song et al., 2026; Wang et al., 2026). Agent-World (Dong et al., 2026) combines simulated and executable environments within a self-evolving training arena. ProgAgent builds verifiable executable environments and controls task difficulty by scaling the environment state while keeping the instruction, tool interface, and verifier fixed across variants (§3.1.2). This design provides tasks at multiple scales for studying and training long-horizon tool orchestration.

B Details on Task Synthesis

B.1 Task and Tool Distributions

Figure 5 shows the distribution of the 215 tasks in §3.3 across 23 top-level domains.

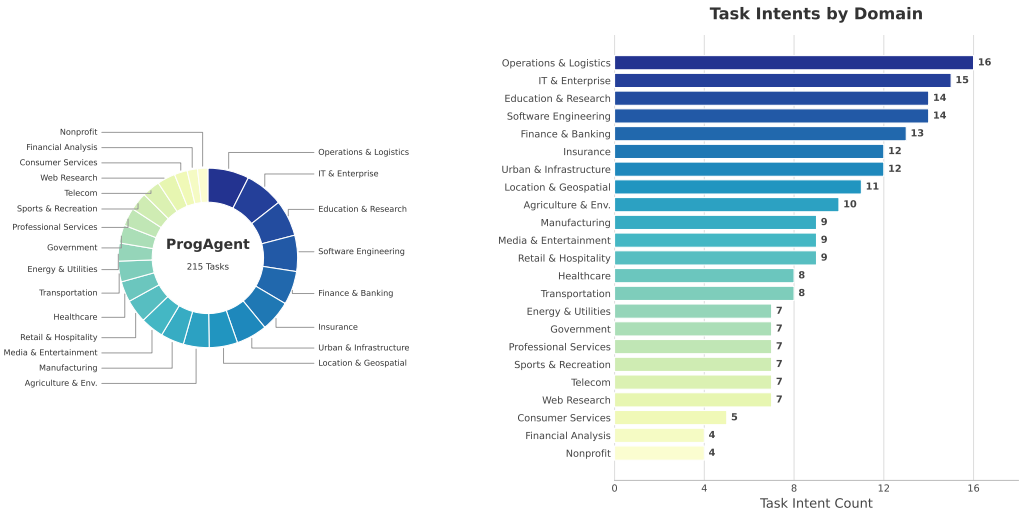

Figure 5: Distribution of 215 tasks across 23 domains.

Figure 6 shows the distribution of the number of environment tools available to the agent across the 215 tasks. For each task, the tool set remains unchanged across all four environment scales.

B.2 Task Construction Pipeline in Detail

We describe the five task construction stages summarized in §3.1.2 in detail.

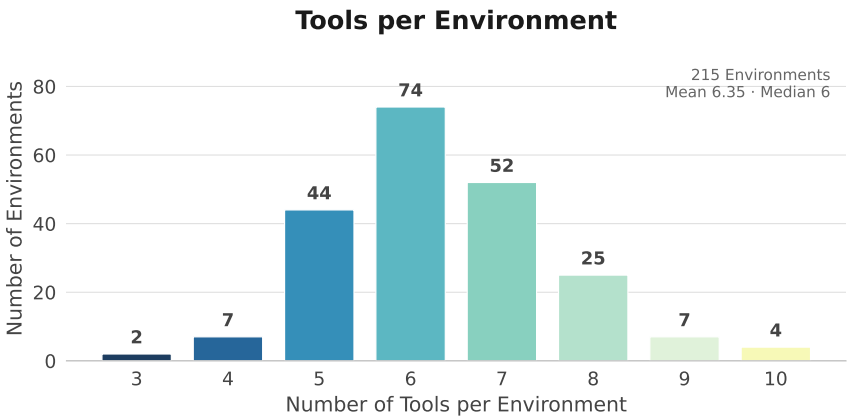

Figure 6: Distribution of environment tool counts across the 215 tasks, excluding external harness tools. Each task is counted once.

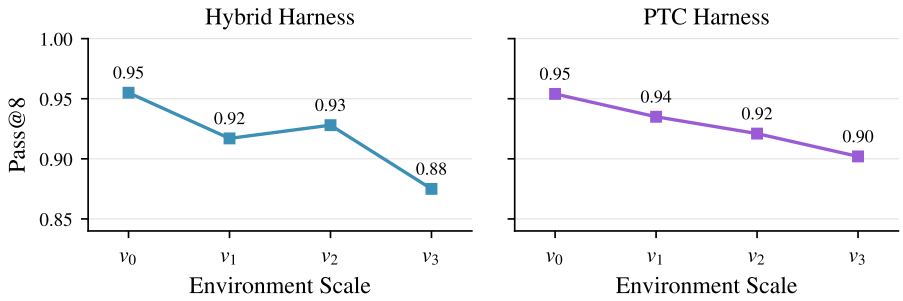

Figure 7: **Environment scaling under Hybrid and PTC.** DeepSeek V4 Pro's pass@8 on synthesized tasks at scales v_0–v_3; the DTC counterpart appears in Figure 1.

Stage I: Domain collection and intent generation. We curate 23 top-level domains and their subcategories. For each subcategory, an LLM generates user intents (prompt in §B.4). A separate LLM call filters these candidates, retaining intents that involve sufficiently rich environment states for task construction.

Stage II: Environment state construction. The constructor agent builds the initial environment state $s_0^E = (D, W)$, comprising a relational database D and a file workspace W. Solving each task requires combining records from D with task constraints specified in unstructured files in W. The constructor agent generates Python scripts to populate D and W, with environment state size controlled by the `SCALE` configuration (§B.4).

Stage III: Tool & instruction synthesis. The constructor synthesizes the action space $\mathcal{A}_{\text{tool}}$ as a set of single-purpose tools with exposed interface schemas and hidden Python implementations. Each tool call returns a bounded amount of data, for example through pagination, so retrieving large amounts of data requires multiple calls. The constructor also generates the task instruction u_i, specifying the task goals, constraints, and required outputs. The instruction omits tool names, database schemas, and verifier implementation details; tool schemas are exposed separately through the harness.

Stage IV: Verifier construction and task certification. Given the initial environment state s_0^E, $\mathcal{A}_{\text{tool}}$, and u_i, the constructor instantiates a Python verifier $R_i(s_T^H, s_T^E) \in \{0, 1\}$ that checks the final response and final environment state against the task's evaluation criteria. The verifier returns 1 if and only if all criteria are satisfied. Verification does not use an LLM judge. The criterion-level score used for SFT trajectory selection is described in §B.4. The constructor then generates a Python oracle solution restricted to the provided tools $\mathcal{A}_{\text{tool}}$. If the oracle fails verification, the constructor uses its execution trace to iteratively repair the environment data, tools, instruction, or verifier until the oracle receives a reward of 1.

Stage V: Controlled scaling. The constructor varies the `SCALE` configuration from Stage II to instantiate each task at four environment scales (v_0–v_3). This expands the database and file workspace while keeping the task instruction, underlying logic, tool schemas, and verifier fixed. The `SCALE` configuration sets database row counts to $1\times$, $2\times$, $4\times$, and $8\times$ their v_0 values, and scalable workspace file counts to $1\times$, $2\times$, $3\times$, and $4\times$, respectively.

As the environment scales from v_0 to v_3, DeepSeek V4 Pro's pass@8 decreases from 0.95 to 0.42 under DTC (Figure 1) while remaining above 0.87 under the Hybrid and PTC harnesses (Figure 7).

B.3 Verification-based Filtering in Detail

After oracle-based certification, candidate tasks may still contain inconsistencies between the instruction and verifier or fail to meet the difficulty criterion. We therefore apply an independent harness audit followed by empirical difficulty filtering, as outlined in §3.1.3.

Independent harness audit. An independent harness agent writes and runs its own solver for each candidate task at the largest scale (v_3). It then checks whether the tools, environment state, and verifier R_i are consistent with the instruction u_i. Tasks with identified issues are repaired and re-audited; tasks are discarded only during the subsequent difficulty filtering.

Empirical difficulty filtering. After the audit, we evaluate task difficulty using model rollouts. For each audited task i and each scale $j \in \{0, 1, 2, 3\}$, we sample $K = 8$ independent trajectories from DeepSeek V4 Pro under the PTC harness. We compute the pass rate $p_{i,j}$ as the fraction of trajectories for which the binary verifier R_i returns 1. We retain a task only if $p_{i,j} \in (0, 0.8)$ at every scale. This criterion removes 35 of the 250 candidate tasks, leaving 215 tasks.

For each retained task, we package each scale variant $\mathcal{M}_i^{(j)}$ into three artifacts in our native format: a *data generator* (a parameterized Python script that populates the database D and workspace W), an *environment module* (tool implementations, the verifier R_i, and their enclosing environment class), and the *task prompt* u_i. The runtime (§3.2, Appendix C.2) consumes these three artifacts. We additionally provide each task in a Harbor-compatible format (Harbor Framework Team, 2026) for use outside our training stack.

B.4 Prompts for Task Synthesis and Auditing

In Stage I, we use Gemini 3.0 Flash to generate user intents for each curated subcategory.

Intent generation prompt (Stage I)

Here are some existing task examples for reference:
{`seed_text`}

Generate {`num_topics`} brand-new task descriptions for the domain "{`category`}".

Requirements:

- Length: 20-50 words.
- Realistic and plausible in everyday business/operations.
- Involves more than 3 data entities with meaningful data volume.
- Describes an operational workflow that concludes with an actionable verifiable output.

Output a JSON array only. Each element must have:

- `"topic"`: the task topic description.
- `"folder"`: a unique short snake_case directory name.

Output only the JSON array.

For each retained intent, the constructor agent, powered by Claude Opus 4.6 through Claude Code, receives a single prompt covering Stages II–V of §3.1.2. The `step` method returns a criterion-level score $\widetilde{R}_i \in [0, 1]$, and `verify` executes the oracle solution $\pi_i^\star$ and submits its output to `step`. Difficulty filtering and RL use the binary verifier R_i, which returns 1 if and only if $\widetilde{R}_i = 1$. SFT trajectory selection uses the criterion-level score $\widetilde{R}_i$, retaining trajectories with $\widetilde{R}_i \geq 0.7$ (§C.3). We

show excerpts from the task construction and harness audit prompts, retaining the opening instructions of each section and marking omitted details with ellipses.

Task construction prompt (Stages II–V, excerpts)

You can use the conda environment `eval` to build a complex Agent task. The task topic is:

`{TOPIC}`

The Agent reads `prompt.md` as instruction and completes the task by calling tools defined in `env.py` and reading files in the `workspace`.

You are building the same task realized at multiple data scales (`v0` → `v3`); difficulty is introduced only by scale.

1. Build the Data Generation Script `data.py`

Create `v0/data.py` to initialize the database and generate workspace files.

...

2. Build the Environment `env.py`

Implement `v0/env.py`. `env.py` and `prompt.md` will be copied byte-for-byte to v1/v2/v3; only `data.py`'s `SCALE` block differs across versions. A single implementation must therefore handle every scale from v0 to v3 without special-casing.

...

3. Task Design

3.1 Task Definition (`prompt.md`)

Write the task description `v0/prompt.md` based on the existing data and tools.

...

3.2 Task Evaluation (`Task_Env.step`)

Based on `prompt.md`, implement the `Task_Env.step` function in `env.py`.

...

3.3 Task Verification (`Task_Env.verify`)

Implement the `verify` method in `Task_Env` to prove end-to-end solvability.

...

4. Replicate to v1, v2, v3

Once `v0/` passes verification at the v3 scale, produce the remaining three versions:

...

An independent harness agent, powered by Claude Opus 4.6 through Claude Code, solves each task at v_3 using its own solver and checks the directory structure, instruction–verifier alignment, and consistency across scale variants.

Harness audit prompt (excerpts)

You have just built a task at `v0/` and copied it to `v1/..v3/`. Your job is to verify the four versions hold together and fix any issues. You can use the conda environment `eval`.

Work through the sections below in order. If any check fails, fix it in `v0/` first (the single source of truth), re-propagate to `v1/..v3/`, then re-run the relevant check.

1. Structural Integrity

Confirm without opening any Python files:

...

2. Solvability Verification at v3 Scale

Verify the task is solvable on the hardest version

...

3. Env Quality

Read `v3/prompt.md` and `v3/env.py`'s `step` function side by side.

...

4. `prompt.md` Quality

Formatting:

1. No title or heading at the top of the file.

...

5. Final Clean-Up and Report

1. Remove runtime artifacts from every `v_n/`: `data.db`, `workspace/`, `__pycache__/`.

...

C Reproducibility and Implementation Details

C.1 Agent Prompts and Action Formats

During training, all three harnesses provide a terminal tool and a `claim_done` tool. Calling `claim_done` ends the episode and submits the agent's final response for verification.

DTC and Hybrid share the following system prompt during training:

Agent system prompt (DTC and Hybrid)

You are a helpful assistant. Use the provided tools to complete the task. When you have finished the task, call the 'claim_done' tool to submit your result.

PTC uses the following system prompt during training:

Agent system prompt (PTC)

You are a helpful assistant. Complete the task using the provided tools. Tools you may call directly:

- `programmatic_tool_call` — Python sandbox; the only way to invoke env tools.
- `terminal` — shell in the workspace directory.
- `claim_done` — submit your final answer when the task is complete.

DTC exposes task tools for direct calls, PTC requires calls through `programmatic_tool_call`, and Hybrid supports both. The PTC tool description is shown below; the Hybrid version differs only in the opening paragraph.

Code-execution tool description (PTC)

Run Python that calls the tools listed above as `tools["tool_name"](*args, **kwargs)`. **This is the ONLY way to invoke env tools** — they cannot be called as standalone tool calls, so every env

tool must go through this sandbox. State (variables, imports) persists across calls. Use `print()` to see output.

USE WHEN: loops, conditionals, error handling, or chaining multiple tool calls with intermediate processing.

Notes:

- Code runs in the workspace directory and file writes are restricted to it, don't write to `/tmp`; always use absolute paths for file writes; os, json, csv, sys are pre-imported.
- Tools return native Python values; the type and structure vary by tool (e.g. dict, list, or str). Always print and inspect the first result before processing many items; do not assume a result is a list and loop over it.
- Very large printed output is truncated; print summaries rather than large raw data.
- Tools may raise an exception; wrap calls in try/except to handle failures.

Usage examples:

Batch + conditional workflow:

```
print(type(tools["get_info"](id='A001')))  # inspect return type first, then loop
results = []
for item in ['A001', 'A002', 'A003']:
    info = tools["get_info"](id=item)
    if info.get('status') == 'active': # info is a dict, confirmed above
        results.append(tools["get_details"](id=item))
    else:
        print(f'Skipping {item}')
    print(f'Processed {item}')
print('Collected', len(results))
```

Error handling:

```
ok, failed = [], []
for item_id in ['A001', 'A002', 'A003']:
    try:
        r = tools["get_info"](id=item_id)
        ok.append(r)
    except Exception as e:
        failed.append((item_id, str(e)))
    print(f'{len(ok)} ok, {len(failed)} failed')
print('Failed:', failed[:3] if failed else 'none')
```

Code-execution tool description (Hybrid): opening paragraph

Run Python that calls the tools listed above as `tools["tool_name"](*args, **kwargs)`. State (variables, imports) persists across calls. Use `print()` to see output.

C.2 PTC Runtime Architecture

Kernel and tool interface. Each episode uses a persistent IPython kernel that preserves variables across turns. Tools are invoked as `tools["tool_name"](*args, **kwargs)`. We use dictionary syntax because MCP tool names may contain characters that are invalid in Python identifiers. The kernel executes each program ρ_t and returns its `stdout` as the observation r_{ρ_t}, or a traceback if execution fails. Output is capped at 50,000 characters; longer outputs retain the first 35,000 and last 15,000 characters.

Sandbox layout. The kernel runs under Bubblewrap (Containers contributors, 2025) in separate user, mount, and PID namespaces. The workspace W is the only writable host-mounted path; system directories are read-only, and temporary storage is private to the sandbox. The environment module and data generator are not mounted, preventing direct access to the verifier implementation and oracle solution. Terminal commands use the same sandbox configuration.

Episode isolation and resource limits. Each episode starts with a fresh environment and kernel. The task database is accessible only through environment tools. Each kernel and its descendant processes are collectively limited to 2 GiB of memory and 256 processes through cgroup v2, with swap disabled.

C.3 Training Hyperparameters

Both backbones, Qwen3-8B and Qwen3-14B, are used in thinking mode. All three settings share the same pool of 215 tasks, each instantiated at four scales. For cold-start SFT, we collect teacher trajectories separately under each harness and retain those with dense reward $\widetilde{R}_i \geq 0.7$ to obtain more training trajectories at a lower collection cost. The three SFT datasets contain equal numbers of trajectories and each cover all 215 tasks. We use the 0813 version of DeepSeek V4 Pro for all experiments.

Supervised fine-tuning. We apply YaRN (Peng et al., 2024) to extend the base model's native 32k context window to 128k for fine-tuning. During SFT, we convert tool calls from JSON to XML format (Team, 2026) to avoid JSON escaping. Models are trained with ms-swift (Zhao et al., 2024) for 3 epochs at a batch size of 64, a learning rate of 1×10^{-5}, and a maximum sequence length of 128k. SFT uses tensor parallelism of 2 with sequence parallelism on 2 nodes, each with 8 H200 GPUs.

Reinforcement learning. All three settings use GSPO (Zheng et al., 2025a;b) with the same task pool and verifiers. We set $\epsilon_{\text{low}} = 3 \times 10^{-4}$ and $\epsilon_{\text{high}} = 4 \times 10^{-4}$, use no KL penalty, and set the learning rate to 1×10^{-6}. We train for 150 steps, sampling 16 rollouts per task for 32 tasks at each step, at temperature 1.0.

The maximum prompt and response lengths are 4k and 85,000 tokens, respectively. Training runs on verl (Sheng et al., 2025; Yu et al., 2026) with a Megatron-LM (Shoeybi et al., 2019) backend and vLLM (Kwon et al., 2023) rollouts. RL uses tensor parallelism of 4 and context parallelism of 2 on 8 nodes, each with 8 H200 GPUs.

C.4 Evaluation Protocol

Harnesses. Our evaluations retain each benchmark's official MCP servers, verifier, system prompt, and task-completion logic. For the main results, we use the DTC harness for DTC-trained models and for baselines and untrained backbones evaluated in our own runs. Hybrid additionally exposes `programmatic_tool_call`, while PTC exposes the benchmark tool schemas, but requires calls through `programmatic_tool_call`. During evaluation, `programmatic_tool_call` has a 120-second timeout. For the main results, each trained model is evaluated under its training harness. Table 7 additionally reports cross-harness evaluations of our trained models, untrained backbones under all three harnesses, and selected trained baselines under PTC. Following Agent-World (Dong et al., 2026), we use YaRN (Peng et al., 2024) to extend the native 32k context windows of open-source models to 128k.

Benchmark subsets. We exclude the MCPMark GitHub suite because repeated account suspensions during evaluation prevented consistent comparisons across models. The reported MCPMark average is the mean over the Filesystem, Notion, Playwright, and PostgreSQL suites. LOCA-bench is run at its four environment-state sizes (§4). For MCP-Atlas, we use the full task set from the final release before version 2.0, which is compatible with our Python-based evaluation harness.

Sources of reported scores. The MCPMark scores of the frontier proprietary models, the open-source foundation models, and the trained baselines other than EnvFactory-8B are taken from Agent-World (Dong et al., 2026). For the frontier proprietary models, the LOCA-bench and MCP-Atlas scores are taken from the respective benchmark papers (Zeng et al., 2026; Bandi et al., 2026). In the Toolathlon column of Table 2, scores for GPT-5.2, Gemini-3 Pro, Claude Sonnet-4.5, and DeepSeek-V3.2 are from the original Toolathlon benchmark and marked with †. All other scores in that column are from Toolathlon-Verified. All remaining scores are from our own evaluations.

D Extended Experimental Results

D.1 Full Ablation Results

Table 7 reports the full LOCA-bench ablation results, including the results summarized in Figure 3.

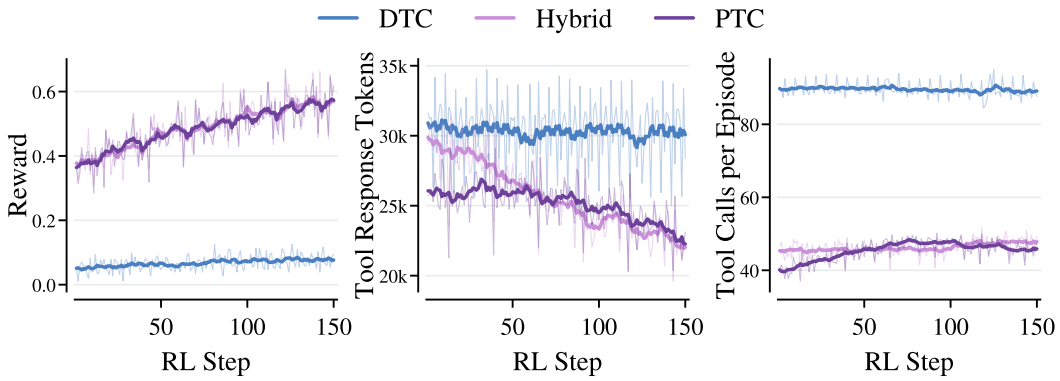

Figure 8: **RL training dynamics at 8B.** Episode reward (left), tool-output tokens entering context (middle), and tool calls per episode (right) under DTC, Hybrid, and PTC, with the same conventions as Figure 4.

Untrained base models. We evaluate the untrained 8B and 14B backbones under DTC, Hybrid, and PTC harnesses to establish performance before training. Pass rates remain near zero under all three harnesses.

Trained baselines under the PTC harness. We evaluate selected AWM (Wang et al., 2026), EnvFactory (Xu et al., 2026), and EnvScaler (Song et al., 2026) checkpoints under the PTC harness. Their pass rates remain near zero, suggesting that access to the PTC interface alone is insufficient for effective tool orchestration, consistent with §5.3.

Training paradigms, scaled environments (v_0–v_3). We report SFT and subsequent RL results for DTC, Hybrid, and PTC models trained on all four environment scales. Cross-harness evaluations help separate training effects from harness effects; additional rows evaluate DTC- and PTC-trained models under the DTC harness with the LOCA-bench memory tool.

Initial environments (v_0) only. To control for training environment difficulty, we restrict SFT and RL for DTC and PTC to the v_0 versions of all tasks. Both training paradigms cover all 215 tasks in SFT and RL, with equal numbers of SFT trajectories. We evaluate the resulting checkpoints under each harness. At 8B and 14B, PTC-trained models have higher average pass rates than DTC-trained models under the same harness after SFT and after RL.

Server-level error detection disabled. We evaluate the final PTC checkpoints with the mechanism that converts MCP server errors into tool exceptions disabled (§5.4). Average pass rates decrease at both model sizes, consistent with the importance of explicit exceptions for error recovery.

D.2 Interaction Costs on MCPMark

Table 8 reports interaction costs for the 8B and 14B PTC models across four MCPMark subsets. At both model sizes, Playwright has the highest proportion of DTC-like PTC calls (§5.4). It also has more tool-output tokens per trajectory and a higher fraction of trajectories reaching the context limit than Filesystem and PostgreSQL.

D.3 Training Dynamics at 8B

Figure 8 shows that the 8B backbone follows similar RL training trends to the 14B backbone (§5.1). Episode reward increases from approximately 0.05 to 0.08 under DTC, compared with 0.38 to 0.56 under both PTC and Hybrid. Tool-output tokens entering context remain near 30k per episode under DTC, but decrease from 26.2k to 22.5k under PTC and from 29.6k to 22.1k under Hybrid. PTC and Hybrid use roughly half as many tool calls per episode as DTC, approximately 45 versus 90.

Table 7: **Ablations on LOCA-bench.** Pass rates (%) across four environment-state sizes. SFT and +RL denote checkpoints after supervised fine-tuning and subsequent reinforcement learning. For our trained models, Setting denotes the training paradigm; evaluation uses the same harness unless otherwise specified. Rows labeled "memory tool" use the DTC harness with the LOCA-bench memory tool. Values spanning the SFT and +RL columns report a single evaluated checkpoint.

Model	Setting	8k		16k		32k		64k		Avg.	
		SFT	+RL	SFT	+RL	SFT	+RL	SFT	+RL	SFT	+RL
Untrained base models											
8B	DTC	1.3		0.0		0.0		0.0		0.3	
	Hybrid	1.3		0.0		0.0		0.0		0.3	
	PTC	0.0		0.0		0.0		0.0		0.0	
14B	DTC	1.3		0.0		0.0		0.0		0.3	
	Hybrid	2.7		0.0		0.0		0.0		0.7	
	PTC	0.0		0.0		0.0		0.0		0.0	
Trained baselines under the PTC harness											
AWM-8B	PTC harness	0.0		0.0		0.0		0.0		0.0	
AWM-14B	PTC harness	0.0		0.0		0.0		0.0		0.0	
EnvFactory-8B	PTC harness	0.0		0.0		0.0		0.0		0.0	
EnvScaler-8B	PTC harness	1.3		0.0		0.0		0.0		0.3	
Training paradigms, scaled environments (v_0–v_3)											
8B	DTC	5.3	5.3	6.7	6.7	0.0	1.3	0.0	2.7	3.0	4.0
	DTC, memory tool	6.7	4.0	1.3	4.0	0.0	1.3	0.0	1.3	2.0	2.7
	DTC, PTC harness	16.0	20.0	10.7	10.7	8.0	8.0	1.3	4.0	9.0	10.7
	Hybrid, DTC harness	10.7	13.3	6.7	8.0	6.7	4.0	4.0	4.0	7.0	7.3
	Hybrid	22.7	28.0	24.0	28.0	13.3	16.0	10.7	14.7	17.7	21.7
	PTC, DTC harness	5.3	17.3	4.0	5.3	0.0	0.0	0.0	1.3	2.3	6.0
	PTC, memory tool	10.7	16.0	4.0	13.3	0.0	6.7	0.0	1.3	3.7	9.3
	PTC	24.0	34.7	20.0	24.0	20.0	21.3	20.0	17.3	21.0	24.3
14B	DTC	8.0	14.7	12.0	13.3	4.0	4.0	0.0	4.0	6.0	9.0
	DTC, memory tool	16.0	20.0	10.7	13.3	2.7	4.0	4.0	4.0	8.3	10.3
	DTC, PTC harness	25.3	28.0	24.0	24.0	10.7	17.3	13.3	13.3	18.3	20.7
	Hybrid, DTC harness	10.7	26.7	10.7	12.0	6.7	13.3	4.0	2.7	8.0	13.7
	Hybrid	26.7	38.7	25.3	30.7	14.7	33.3	12.0	22.7	19.7	31.3
	PTC, DTC harness	12.0	23.7	5.3	18.7	4.0	8.0	2.7	5.3	6.0	13.9
	PTC, memory tool	21.3	24.0	13.3	14.7	6.7	9.3	1.3	5.3	10.7	13.3
	PTC	32.0	41.3	30.7	40.0	33.3	41.3	28.0	30.7	31.0	38.3
Initial environments (v_0) only											
8B	DTC	5.3	8.0	2.7	2.7	1.3	2.7	1.3	0.0	2.7	3.3
	DTC, PTC harness	5.3	4.0	4.0	4.0	2.7	4.0	1.3	2.7	3.3	3.7
	PTC, DTC harness	8.0	13.3	2.7	8.0	1.3	4.0	1.3	2.7	3.3	7.0
	PTC	13.3	21.3	13.3	18.7	14.7	14.7	4.0	9.3	11.3	16.0
14B	DTC	8.0	13.3	10.7	8.0	4.0	4.0	0.0	1.3	5.7	6.7
	DTC, PTC harness	16.0	20.0	17.3	20.0	6.7	14.7	1.3	10.7	10.3	16.3
	PTC, DTC harness	18.7	20.0	10.7	13.3	5.3	6.7	2.7	2.7	9.3	10.7
	PTC	24.0	25.3	26.7	26.7	21.3	26.7	13.3	16.0	21.3	23.7
Server-level error detection disabled											
8B	PTC	29.3		18.7		17.3		12.0		19.3	
14B	PTC	42.0		40.0		29.3		26.7		34.5	

Table 8: **Interaction costs of PTC on MCPMark.** Tool calls and tool-output tokens are averaged per trajectory. Context limit reports the fraction of trajectories reaching the context limit. The "DTC-like calls" column reports the percentage of PTC calls containing a single explicit tool invocation without control flow, error handling, or selective printing; any printed tool return is reproduced in full without processing.

Model	Environment	Tool calls	Tool-output tokens	Context limit (%)	DTC-like calls (%)
8B	Filesystem	50.5	21,086	16.7	6.21
	Postgres	68.3	22,404	36.9	18.00
	Notion	78.7	39,259	69.6	10.26
	Playwright	122.1	37,551	64.0	34.71
14B	Filesystem	34.9	15,925	8.8	10.10
	Postgres	58.0	22,200	8.3	14.70
	Notion	66.7	35,992	46.4	18.71
	Playwright	135.0	35,988	52.0	40.72

E QUALITATIVE EXAMPLES OF PTC

We present two concrete examples to illustrate how PTC works in practice. For each task, we show the key steps of PTC execution, with ... marking omitted output.

E.1 MCP-ATLAS: FROM CAMPAIGN RECORDS TO A STATISTICAL ANSWER

This case shows how PTC aggregates structured tool results inside the execution environment and returns a compact statistical summary. Steps 1–3 are excerpts from a single PTC execution; its output is split between Steps 2 and 3 to match the code.

Instruction

I'm trying to understand some ad data in my database. I want to see some details of campaigns that spend less than $800, specifically what was the most common type of ad, and for the least used one, what the average audience reach was.

AVAILABLE MCP SERVERS

Notion, Memory, MCP Code Executor.

1. Query campaigns spending less than $800

PTC CALL Python

```python
# db_id: advertising database identified in earlier calls.
results = tools["notion_API-post-database-query"](
    database_id=db_id,
    filter={
        "property": "SpentAmount",
        "number": {"less_than": 800}
    }
)
```

2. Aggregate the records

PTC CALL Python

```python
# Continuation of the same PTC call; results holds the query response.
from collections import Counter
ad_counts = Counter()
audience_reach_by_type = Counter()

for r in results['results']:
    ad_type = r['properties']['AdType']['select']['name']
    ad_counts[ad_type] += 1
    audience_reach_by_type[ad_type] += r['properties']['AudienceReach']['number']

print("Ad type counts:")
for ad_type, count in ad_counts.most_common():
    avg_reach = audience_reach_by_type[ad_type] / count
    print(f"  {ad_type}: {count} campaigns, avg audience reach={avg_reach:.1f}")
```

OUTPUT Execution result

```
Ad type counts:
  Social Media: 30 campaigns, avg audience reach=23512.0
  Search Engine: 28 campaigns, avg audience reach=24872.6
  Video: 20 campaigns, avg audience reach=25063.5
  Banner: 17 campaigns, avg audience reach=19380.5
```

3. Print the computed answer

PTC CALL Python

```python
print("\nMost common type:", ad_counts.most_common(1)[0][0])
print("Least common type:", ad_counts.most_common()[-1][0])
```

```
least_common_type = ad_counts.most_common()[-1][0]
print(f"\nAverage audience reach for {least_common_type}: {audience_reach_by_type[
    least_common_type] / ad_counts[least_common_type]:.1f}")
```

OUTPUT — Execution result

```
Most common type: Social Media
Least common type: Banner

Average audience reach for Banner: 19380.5
```

4. Check completeness before the final response

PTC CALL — Python

```
results = tools["notion_API-post-database-query"](
    database_id=db_id,
    filter={
        "property": "SpentAmount",
        "number": {
            "less_than": 800
        }
    },
    page_size=100
)
print(f"Total: {len(results.get('results', []))}, Has more: {results.get('has_more',
    False)}")

# Also let's verify by looking at min and max spent
spent_values = []
ad_types = []
for r in results['results']:
    spent_values.append(r['properties']['SpentAmount']['number'])
    ad_types.append(r['properties']['AdType']['select']['name'])

print(f"Min spent: {min(spent_values)}, Max spent: {max(spent_values)}")
print(f"Ad types: {set(ad_types)}")
```

OUTPUT — Execution result

```
Total: 95, Has more: False
Min spent: 509, Max spent: 798
Ad types: {'Banner', 'Social Media', 'Search Engine', 'Video'}
```

Final agent response

The analysis of campaigns spending less than $800 is complete. Here's the summary:

Ad Campaign Analysis (Spent less than $800)

Ad Type	Campaigns	Avg Audience Reach
Social Media	30	23,512.0
Search Engine	28	24,872.6
Video	20	25,063.5
Banner	**17**	**19,380.5**

Key Findings

- **Most common type of ad**: **Social Media** accounts for 30 of the 95 campaigns (31.6%), followed by Search Engine (28) and Video (20).
- **Least used type**: **Banner** ads appear in just 17 campaigns (17.9%).
- The **least used type — Banner** — has an average audience reach of **19,380.5**.

E.2 TOOLATHLON-VERIFIED: FROM AN EMAILED SCHEDULE TO PUBLISHED CANVAS COURSES

This case shows how PTC uses loops and returned identifiers to coordinate dependent course-creation and configuration calls.

Instruction

This is admin3, a registrar from XXX University. The university has sent me the new semester's course schedule via email with the subject "Course Schedule Notification." I need you to help me find the courses I'm responsible for from the attached master schedule and create corresponding courses on Canvas for the instructors. The specific requirements are as follows:

- Assign the corresponding instructor as the teacher for each course on Canvas. The instructors' emails are saved in my local workspace.
- Publish an announcement for each course that includes the instructor's name and the class time.
- If an instructor is teaching the same course multiple times, these courses should be set up for resource sharing in Canvas.
- The syllabi for each course are saved in my workspace. Please publish them for their respective courses.
- All courses I am responsible for need to be published.

AVAILABLE MCP SERVERS

Filesystem, Canvas, Terminal, Emails.

1. Select assigned courses and look up instructor IDs

PTC CALL Python

```python
# data: parsed schedule rows (course, instructor, time, administrator).
# parse_tool_result: helper defined earlier to decode the tool response.
admin3_courses = [(c, i, t) for c, i, t, a in data if a == "admin3"]
instructor_map = {}
for name in set(i for _, i, _ in admin3_courses):
    result = tools["canvas_canvas_list_account_users"](
        account_id=account_id, search_term=name, page=1, per_page=20)
    users = parse_tool_result(result)
    if users:
        u = users[0] if isinstance(users, list) else users
        instructor_map[name] = u['id']
```

2. Create courses and use returned IDs to enroll instructors

PTC CALL Python

```python
print(f"Processing {len(admin3_courses)} courses...")
created_courses = []
for idx, (course_name, instructor, class_time) in enumerate(admin3_courses):
    try:
        course = tools["canvas_canvas_create_course"](
            account_id=account_id, name=course_name,
            start_at=(date.today()).isoformat(),
            end_at=(date.today() + timedelta(days=365)).isoformat(),
            time_zone="America/Denver")
        course_id = course['id']
        tools["canvas_canvas_enroll_user"](
            course_id=course_id, user_id=instructor_map[instructor],
            role="TeacherEnrollment", enrollment_state="active")
        created_courses.append((course_name, course_id, instructor, class_time))
        print(f"[{idx+1}/{len(admin3_courses)}] Created {course_name} (ID={course_id})
     for {instructor}")
    except Exception as e:
        print(f"ERROR {idx}: {e}")
print(f"\nCreated {len(created_courses)} courses.")
```

OUTPUT Execution result

```
Processing 27 courses...
[1/27] Created English Literature-7 (ID=31) for Melissa Sanchez
[2/27] Created Business Management-7 (ID=32) for Cynthia Gomez
[3/27] Created Philosophy-7 (ID=33) for Donald Reed
[4/27] Created Creative Writing-7 (ID=34) for Carolyn Nguyen
...
Created 27 courses.
```

3. Create announcements using course IDs

PTC CALL — Python

```python
# created_courses: 27 course tuples listed earlier in this call.
for course_name, course_id, instructor, class_time in created_courses:
    message = f"Welcome to {course_name}. This course is taught by {instructor} at {
     class_time}."
    try:
        tools["canvas_canvas_create_announcement"](
            course_id=course_id, title=f"{course_name} Class Info",
            message=message, published=True)
        print(f"Announcement for {course_name} (ID={course_id}): {message[:80]}...")
    except Exception as e:
        print(f"Announcement ERROR for {course_name}: {e}")
```

OUTPUT — Execution result

```
Announcement for English Literature-7 (ID=31): Welcome to English Literature-7. This
     course is taught by Melissa Sanchez at Mon...
Announcement for Business Management-7 (ID=32): Welcome to Business Management-7. This
     course is taught by Cynthia Gomez at Wedn...
Announcement for Philosophy-7 (ID=33): Welcome to Philosophy-7. This course is taught by
      Donald Reed at Friday 1,2,3....
...
```

The following program checks course publication, teacher count, the public-syllabus flag, and announcement presence. Its success message refers to these checks; it does not verify instructor identity, syllabus or announcement content, or resource sharing.

4. Check selected course settings and summarize results

PTC CALL — Python

```python
# courses: 27 course-name/ID pairs (literal list omitted).
# instructor_map: Canvas instructor IDs to names (dictionary omitted).
print("=== FINAL COMPREHENSIVE VERIFICATION ===\n")

# Track instructor-to-course count
instructor_courses = {}
for cname, cid in courses:
    detail = tools["canvas_canvas_get_course"](course_id=cid)
    tid = detail['teachers'][0]['id'] if detail['teachers'] else None
    if tid:
        instructor_courses.setdefault(tid, []).append(cname)

# Verify
all_ok = True
failures = []

for course_name, course_id in courses:
    detail = tools["canvas_canvas_get_course"](course_id=course_id)
    state_ok = detail['workflow_state'] == 'available'
    teacher_ok = len(detail['teachers']) == 1
    # Check teacher is correct
    actual_teacher = detail['teachers'][0]['display_name'] if detail['teachers'] else '
     NONE'

    syll_ok = detail['public_syllabus']
    anns = tools["canvas_canvas_list_announcements"](course_id=course_id)
    ann_ok = len(anns) >= 1

    if not state_ok:
        failures.append(f"{course_name}: state={detail['workflow_state']}")
    if not teacher_ok:
```

```
            failures.append(f"{course_name}: teachers count={len(detail['teachers'])}")
        if not syll_ok:
            failures.append(f"{course_name}: syllabus not public")
        if not ann_ok:
            failures.append(f"{course_name}: no announcements")

        print(f"{course_name}: state={detail['workflow_state']}, teachers={len(detail['
         teachers'])} ({actual_teacher}), syll={syll_ok}, anns={len(anns)}")

if failures:
    print(f"\n× {len(failures)} failures: {failures}")
else:
    print(f"\n✓ All {len(courses)} courses verified successfully!")

# Instructor summary
print("\nInstructor-Course Mapping:")
for tid, cnames in sorted(instructor_courses.items()):
    print(f"  {tid} ({instructor_map[tid]}): {len(cnames)} courses - {cnames}")

print(f"\nTotal unique instructors: {len(instructor_courses)}")
print(f"Total courses: {len(courses)}")
```

OUTPUT Execution result

```
=== FINAL COMPREHENSIVE VERIFICATION ===

English Literature-7: state=available, teachers=1 (Melissa), syll=True, anns=1
Business Management-7: state=available, teachers=1 (Cynthia), syll=True, anns=1
Philosophy-7: state=available, teachers=1 (Donald), syll=True, anns=1
Creative Writing-7: state=available, teachers=1 (Carolyn), syll=True, anns=1
...

✓ All 27 courses verified successfully!

Instructor-Course Mapping:
  62 (Dennis Robinson): 3 courses - ['Marketing Principles-7', 'Operations Management
    -7', 'Innovation Management-7']
  63 (Donald Reed): 3 courses - ['Philosophy-7', 'Optics-7', 'Business Strategy-7']
...

Total unique instructors: 11
Total courses: 27
```