MCP-Universe RL: A Framework for Training MCP Tool-Use Agents via Reinforcement Learning

Ziyang Luo **Yan Yang** **Xiangru Jian** **Ziji Shi**
Xiaoqiang Lin **Jun Hao Liew** **Silvio Savarese** **Junnan Li**
Salesforce AI Research
mcp-universe.github.io/mcpu-rl
SalesforceAIResearch/MCP-Universe

Abstract

Reinforcement learning (RL) has become an effective way to improve the tool-use ability of large language models (LLMs), but most existing RL frameworks stop at the policy update. For every new domain, the user is left with two hard systems problems: standing up an isolated environment for each of hundreds of concurrent trajectories and connecting it to training, and scheduling the rollout so that the GPU stays busy across long, multi-turn episodes that spend much of their time stalled on slow tool calls. We present **MCP-Universe RL (MCP-U RL)**, an open-source framework that takes over both. It uses the *Model Context Protocol* (MCP) as the interface to the environment, so any tool already exposed as an MCP server plugs into training with no RL-specific integration code. It builds the two missing layers once and reuses them across domains: an environment-orchestration layer that provisions, isolates, and recycles the MCP environments over a pluggable container backend, and a rollout-orchestration layer whose staged pipeline overlaps trajectories to keep the GPU busy while episodes wait on tools. A backend-agnostic training layer then applies the update through an existing RL backend, with veRL and slime integrations. With one configuration, changing only the task specification, we train software-engineering, deep-research, and general tool-use agents on `gpt-oss-20b` and improve task reward in all three.

1 Introduction

Large language models (LLMs) are increasingly expected to act in the real world through external tools (Luo et al., 2025b), such as running code, sending emails, querying databases, or operating real applications, rather than producing text answers alone. Reinforcement learning (RL) has become an effective way to improve this behavior: by rewarding task completion, algorithms such as GRPO (Shao et al., 2024; DeepSeek-AI et al., 2025) and DAPO (Yu et al., 2025) teach an agent to plan, call tools, and recover from errors over many turns. Putting this into practice takes more than the algorithm. General RL frameworks (Sheng et al., 2025; Hu et al., 2024; THUDM, 2025; Fu et al., 2025) provide the optimization, the policy update and distributed training, but to train a tool-use agent the user still has to build two further layers for every new domain.

The first is the *environment-orchestration* layer. Agentic RL samples many trajectories per update, and each one acts on a live environment whose state it changes, editing files, writing database rows, or navigating a stateful session (Jimenez et al., 2023; de Chezelles et al., 2024). The reward is usually read back from that state, so if two trajectories shared an environment their edits would collide and the reward would be wrong; each trajectory therefore needs its own isolated environment. At the scale of RL, hundreds of these are started on a container backend, kept reachable, and recycled at once, which is a systems problem in itself. Yet the work is nearly the same from one domain to the next: a web-search environment and a code repository differ in what they run, not in how they are provisioned and recycled. The second is the *rollout-orchestration* layer that turns those environments into training signal. Each tool-use turn alternates between generating on the GPU and calling a tool that runs in the environment, and a tool call can stall for seconds. Running one trajectory at a time leaves the GPU idle through every such stall; running many at once keeps it busy but can exhaust host memory with live environments. Keeping the GPU fed without running out of memory is therefore a scheduling problem. Both layers are rebuilt per project today, which is what makes agentic RL slow to set up and hard to reproduce.

In this work, we present **MCP-Universe RL (MCP-U RL)**, an open-source framework that builds both layers once and shares them across

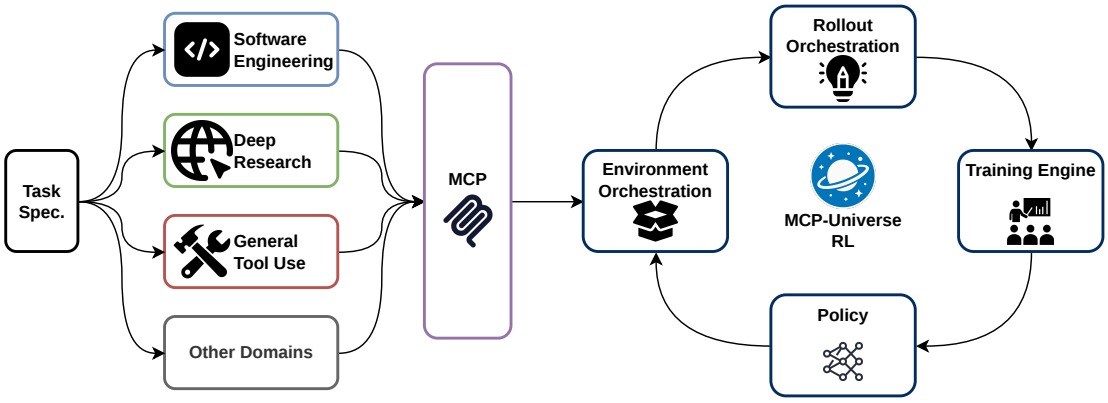

Figure 1: Overview of our MCP-Universe RL framework. Each domain is exposed as one or several MCP servers, and all domains funnel through a single MCP interface into one shared training loop, whose environment-orchestration, rollout-orchestration, and training-engine layers are reused for every domain; only the task specification changes.

domains (Figure 1). The key idea is to use the *Model Context Protocol* (MCP) (Anthropic, 2024) as a single interface to every environment: MCP standardizes how an agent calls tools, and a large ecosystem of MCP servers already exists. Once a domain's tools are exposed as an MCP server, the user only writes a task specification and presses run, with no environment-integration or rollout code. Because every domain is now reached the same way, the *environment-orchestration* layer manages the whole life cycle in one place, provisioning, isolating, and recycling environments over a container backend that can be swapped when one runtime cannot create them fast enough.

Each ready environment is then handed to the *rollout-orchestration* layer, which runs the episodes and turns them into training signal: for each trajectory it drives the agent against its environment, records the tokens for training, and scores the finished episode with the task's evaluator. Running these episodes fast is the hard part, because a tool-use episode leaves the GPU idle whenever the agent is waiting on a tool. The layer hides that wait by running the episodes as an overlapping pipeline, so that some trajectories generate on the GPU while others are acquiring an environment or waiting on a tool call. Acquiring an environment and generating draw on different resources, host memory and the GPU, so we let more trajectories generate than acquire, which keeps the GPU busy without running out of memory. A backend-agnostic *training* layer then applies the policy update through an existing RL backend, with integrations for veRL (Sheng et al., 2025) and slime (THUDM, 2025). Changing only the task specification, we train software-engineering (Jain et al., 2025), deep-research (Lu et al., 2025), and general tool-use (Wang et al., 2026) agents, and improve task reward in all three.

2 Related Work

General RL frameworks such as veRL (Sheng et al., 2025), OpenRLHF (Hu et al., 2024), slime (THUDM, 2025), and AReaL (Fu et al., 2025) supply the optimization backend but leave the environment, the reward, and the multi-turn rollout to the user. MCP-U RL integrates veRL and slime as interchangeable backends and exposes the same adapter interface for adding others, while supplying the missing environment, reward, and rollout layers. A second line of work builds the agentic rollout on top of these frameworks: SkyRL-Agent (Cao et al., 2025) runs a staged asynchronous pipeline, AgentRL (Zhang et al., 2025) a multi-turn multi-task framework, VerlTool (Jiang et al., 2025) a standardized tool API, and rLLM (Tan et al., 2025) many agent harnesses in pluggable sandboxes. All still ask the user to wire each tool or harness in behind their own interface, so a new domain means new integration code; MCP-U RL instead makes MCP the single interface, so any existing MCP server plugs into training unchanged while the environment-orchestration layer manages its life cycle, and our rollout engine exposes its scheduling as configuration.

The work closest to ours shares either our claim or our interface. Agent Lightning (Luo et al., 2025a) traces a running agent so that almost any agent framework can be optimized without code

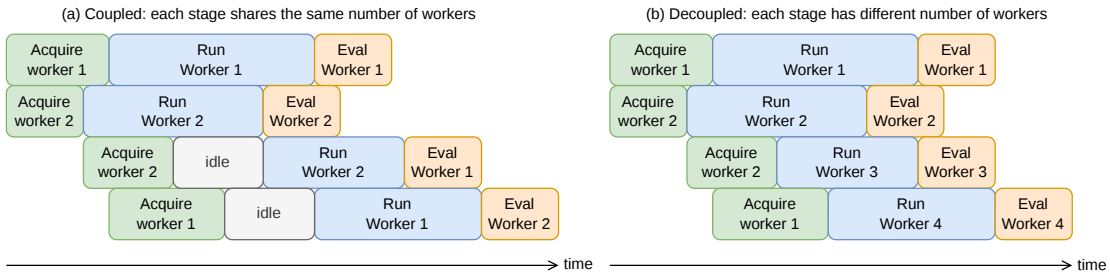

Figure 2: Per-stage concurrency in the rollout pipeline. Each row is one trajectory, flowing Acquire → Run → Eval left to right; block width is relative wall-clock. **(a)** Coupled: when every stage is given the same number of workers, a trajectory that finishes Acquire must wait for a free Run worker, leaving idle gaps. **(b)** Decoupled: giving the GPU-bound Run stage more workers than the fast Acquire stage lets trajectories flow through with less idle.

change; it makes the *agent* pluggable but still leaves the tools and their environments orchestration to build, the layer MCP-U RL supplies. A few efforts instead train agents over MCP tools directly. OpenPipe's MCP-RL (OpenPipe, 2025) trains a model to use a single MCP server's tools, released as software with a demo rather than a cross-domain study. MiroRL (MiroMind AI, 2025) targets the single deep-research domain, and MCP-Flow (Wang et al., 2025) scales MCP tool use through supervised fine-tuning rather than RL. Each of these targets a single server or domain; MCP-U RL instead trains across domains from one framework, with rewards from explicit evaluators run against the environment state.

3 MCP-Universe RL Framework

MCP-U RL has three layers, split by the resource each one uses (Figure 1). The environment-orchestration layer runs the MCP environments on the host, the rollout-orchestration layer runs the episodes between the host and the GPU, and the training-engine layer applies the policy update on the GPU. The layers pass data, not domain logic, so the only thing that changes from one domain to the next is the task specification the user writes.

3.1 Environment-Orchestration Layer

Agentic RL samples many trajectories at once, and each acts on an environment whose state it changes, editing files, writing database rows, or navigating a stateful web session. Sharing an environment would let these changes collide and corrupt the reward, which is read from the final state, so each trajectory needs its own isolated environment. Standing that up, keeping it reachable, and tearing it down is real work that existing frameworks leave to the user; we make it a layer with an interface on each side, one hiding what the tools are and one hiding how they are run. On the agent's side, the environment's tools are served behind a per-environment MCP gateway, so an environment, whether it wraps a browser, a code repository, or a database, is reached the same way: an address the rollout layer calls over MCP. On the runtime side, a *provisioner* exposes `create`, `reset`, `health_check`, and `destroy`, so no specific runtime (e.g., Docker) is hard-coded anywhere else.

Between the two interfaces, the layer manages the life cycle of each environment. It acquires an environment, runs an optional `prepares` step for per-instance setup, and holds it through both the episode and its evaluation, since the reward is often read from the live environment, for example when a hidden test suite runs in the same container the agent just edited. It then resets or destroys the environment. To avoid building one environment per trajectory, the layer keeps a pool. When a domain's tasks share a configuration, the pool is pre-warmed and released environments are reset and reused. When each task needs a distinct image, as in software engineering where every instance carries its own repository, environments are provisioned on demand instead. These two cases stress environment creation in very different ways, and no single runtime handles both well. That is why the provisioner is an interface: we ship one backend on Docker and one daemonless, and a new isolation mechanism is added simply by implementing the same interface.

3.2 Rollout-Orchestration Layer

A tool-use episode does not keep the GPU busy on its own: each turn alternates between genera-

tion on the GPU and a tool call that runs in the environment and can stall for seconds. Running trajectories one at a time idles the GPU through every tool call, so this layer overlaps many trajectories, turning each episode into a scored, tokenized trajectory while others generate. It records the trajectory at the token level, marking which tokens the agent produced and are trainable, and runs the task's evaluator for the reward. The agent type and prompt format are set by configuration, so the same code serves different models and agent styles.

The layer is a single engine organized as a three-stage pipeline whose stages overlap: acquire the environment, run the episode, and evaluate it (Figure 2), following prior agentic-RL rollout systems (Cao et al., 2025). Rather than fix one scheduling policy, we expose the engine's behavior as configuration; the control we rely on most is *per-stage concurrency*, which sizes the three stages' worker pools separately. This matters because the stages are bound by different resources. Acquiring an environment is host-memory and CPU bound, while the run stage is GPU bound (dominated by generation). We therefore give the run stage more workers than the acquire stage (at least twice as many by default), which keeps the GPU busy during other trajectories' tool calls without holding more environments in memory.

Moreover, the same engine also runs under different *placements* of the GPUs relative to the update, again by configuration. A colocated placement shares GPUs between rollout and the update and alternates; a fully asynchronous placement runs them on separate GPUs and streams trajectories through a queue so generation and the update proceed at once. Both drive the one engine, so switching between them is a configuration change rather than a reimplementation.

3.3 Training-Engine Layer

The rollout layer emits its batch in a backend-neutral form, prompt and response token ids, a trainable mask, and the per-trajectory reward, so the update itself can be left to an existing RL backend and this layer only adapts the batch to one. We provide adapters for veRL (Sheng et al., 2025) and slime (THUDM, 2025), and a new one is added by writing the adapter. The backend, the RL algorithm, and the GPU placement (Section 3.2) are all configuration and do not change the layers above.

```
{
  "instance_id": "dr_train_0001",
  "instruction": "Seek the title of a 2012
      paper published in an international
      journal focused on engineering
      science ...",
  "mcp_servers": [{"name": "serper-search"},
                  {"name": "jina-scrape"}],
  "evaluators": [{"func": "raw",
      "op": "deepresearch.llm_as_judge",
      "op_args": {"correct_answer":
          "Grains of Saws ..."}}]

  // optional, for a stateful domain (e.g. SWE):
  "dockerfile_path": "Dockerfile.SWE",
  "build_args": {"SWE_BASE_IMAGE": "..."},
  "prepares": [{"prepare_func": "swe_setup"}],
  "cleanups": [{"cleanup_func": "swe_teardown"}]
}
```

Figure 3: A task specification. The three required fields (`mcp_servers`, `instruction`, `evaluators`) define a stateless task; a stateful domain adds the optional fields below, each referring to a named registry function.

3.4 Task Specification

Once a domain's tools are available over MCP, the user drives the framework through a task specification alone (Figure 3). It is a JSON object naming the MCP servers (`mcp_servers`), the `instruction`, and one or more `evaluators` whose score becomes the reward; we adopt the format of MCP-Universe (Luo et al., 2025b). For a stateless task these three fields suffice. A stateful domain adds optional fields, a `dockerfile_path` and `build_args` for the image and `prepares`/`cleanups` hooks that seed and undo environment state, and these are the only place per-domain logic lives. Each hook and evaluator is a named function in a small registry, so extending to a new domain means writing one such function and referring to it, not changing the layers above.

4 Experiments

We evaluate MCP-U RL along its two contributions. First, that the framework can train and improve agents, using one configuration and changing only the task specification, across domains whose environments and reward checks differ (Section 4.1). Second, that the rollout layer's design choices raise throughput without changing what is trained (Section 4.2). We do not aim for state of the art on any benchmark; reaching it would take dedicated data and environment engineering, which is orthogonal to the framework.

Across all runs, the policy is the open-weight

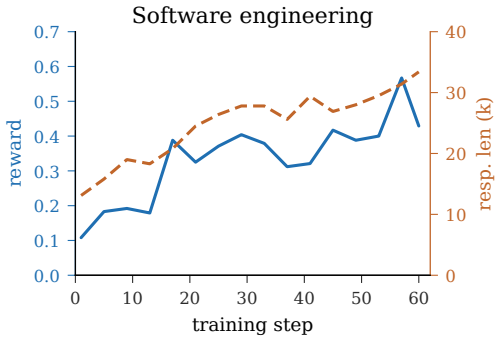

Figure 4: Software-engineering training on R2E-Gym. Left axis (solid, blue): reward; right axis (dashed, orange): average response length in thousands of tokens.

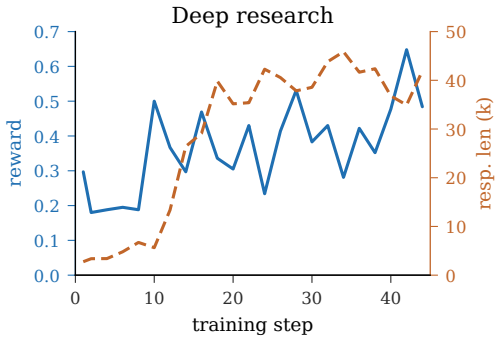

Figure 5: Deep-research training on DeepDive. Axes as in Figure 4.

`gpt-oss-20b` model (OpenAI, 2025), run as a tool-use agent. Training uses GRPO (Shao et al., 2024), with a learning rate of 2×10^{-6} and $n = 8$ trajectories sampled per task. Rewards are binary and come entirely from each task's evaluator. Full settings are in Appendix B.

4.1 Training Agents Across Domains

We train in three domains that differ substantially in their environments, their tools, their tasks, and how they are graded. The framework code and its configuration are the same across all three; only the task specification changes, which is the property we are testing.

Software engineering is a bug-fixing task: given a real bug report on a code repository, the agent must find the faulty code, edit it, and make the project's tests pass. We train on R2E-Gym (Jain et al., 2025), a collection of real GitHub issues where each task pairs an issue with a container image of the repository at the buggy commit and a hidden test suite. The environment is stateful: a `prepares` step checks out the repository, the agent inspects and edits the code over many turns, and the evaluator runs the hidden tests inside the same container for a binary reward on whether they pass. Following mini-swe-agent (SWE-agent Team, 2025), the agent is given a single shell tool, exposed over MCP, and nothing else: it reads, searches, edits, and runs tests only by issuing shell commands. This is the hardest of our three settings, as the agent has to localize the bug, change the code, and re-run the tests with no specialized editing or navigation tools, and episodes run the longest as a result. Over 60 steps of training, the success rate rises from about 0.11 to about 0.43 (peaking near 0.6), and the mean response length grows from 13k to 33k tokens as the agent learns to take more investigative turns before editing (Figure 4).

Deep research is a web-browsing task: given a hard question, the agent must gather and cross-check evidence from many web pages before it can answer, rather than retrieving a single fact. We train on DeepDive (Lu et al., 2025), a set of hard, multi-hop questions whose answers are not on any one page. The agent runs a ReAct (Yao et al., 2022) loop, alternating a reasoning step with a tool call, and reaches the open web through two stateless MCP servers: a Google Search server for queries and a Jina Scrape server for reading pages. It searches and follows links over many turns, then commits to a final answer that the evaluator scores against the reference. Unlike the software-engineering setting there is no container state to manage, only external web tools, yet the rollout and training code are unchanged. Over training the success rate rises from about 0.22 to about 0.52 (peaking at 0.65), and the mean response length climbs more than tenfold, from under 3k to around 40k tokens, as the agent learns to search and read across more pages before answering rather than replying from the first hit (Figure 5).

General tool use is a broad, multi-application setting: given an everyday instruction, such as placing an order on a shopping site, moving money between accounts, or updating a customer record, the agent must drive one or more applications through their tools to carry the task out. We train on the synthetic environments of AgentWorldModel (Wang et al., 2026), which span everyday domains including e-commerce, finance, travel, social media, and customer-management (CRM) applications. Each environment is code- and database-backed, so the agent acts on real state over many turns; each task gives an instruction whose completion changes application state, and the evaluator grades success by querying that state after the episode. We ex-

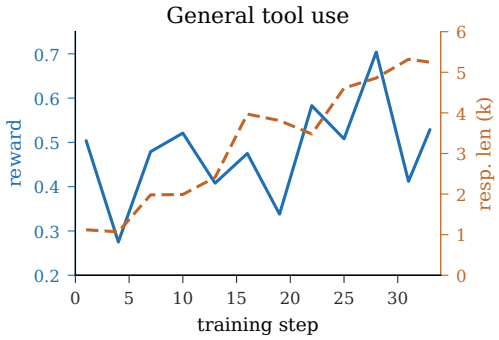

Figure 6: General-tool-use training on AgentWorldModel-1k. Axes as in Figure 4.

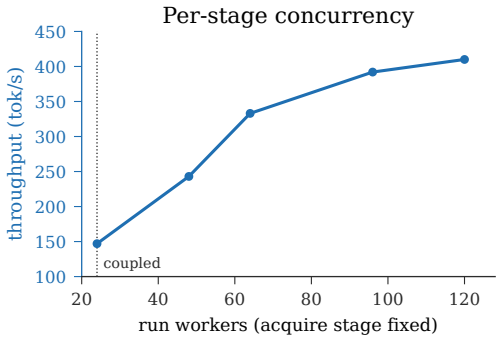

Figure 7: Per-stage concurrency. Rollout throughput as the number of run workers grows, with the acquire stage and live-environment count fixed. The dotted line marks the coupled point (run = acquire); adding run workers is the decoupled setting.

pose each environment's tools over MCP, so the toolset and the checked state differ again from the previous two domains, spanning many applications rather than one shell or the web, yet no framework code changes. The policy already starts around 0.48 here, and over training the success rate rises modestly to about 0.55 (peaking near 0.7); more strikingly, the mean response length grows fivefold, from 1k to over 5k tokens, as the agent stops answering from the surface and learns to probe each application's state and chain many tool calls before it commits (Figure 6).

4.2 Rollout Throughput

The rollout layer is one engine whose behavior is set by configuration (Section 3.2), and the configuration changes how fast trajectories are produced, not what is produced: every setting samples from the same policy and scores with the same evaluator, so the training signal is identical and only the wall-clock differs. We measure throughput on the software-engineering workload, whose long, tool-heavy episodes stress the rollout layer the most.

The control that matters most is per-stage concurrency (Section 3.2). Acquiring an environment is host-memory bound while the run stage is GPU bound (dominated by generation), so we hold the acquire stage fixed and vary the number of run workers, keeping the number of live environments (and thus the memory budget) constant throughout (Figure 7). The point where run and acquire share one limit is the *coupled* setting; there the GPU sits idle whenever that limit is spent on trajectories still acquiring an environment or waiting on a tool. Adding run workers, our *decoupled* setting, lets already-acquired trajectories keep running: rollout throughput rises from 147 to 410 tokens per second, a 2.8× gain, and then saturates as generation fills the GPU, all without holding more environments in memory.

GPU placement	Time / step (s)	Throughput (steps/h)
Colocated	1164	3.1
Fully asynchronous	**572**	**6.3**

Table 1: End-to-end training throughput under the two GPU placements, same engine and workload.

The same holds across GPU *placements*. Table 1 reports end-to-end training throughput under the two at matched concurrency: the colocated placement shares GPUs between rollout and the update so they wait on each other, while the fully asynchronous placement runs them on separate GPUs at once and is about 2× faster. Our asynchronous placement is one-step off-policy, updating on trajectories at most one step stale, so it stays close to on-policy while still overlapping rollout and update. Full settings for all rollout experiments are in Appendix C.

5 Conclusion

We presented MCP-U RL, a framework that trains tool-use agents with RL by using MCP as the interface to the environment. It builds the environment-orchestration and rollout-orchestration layers once and reuses them across domains, so a new domain is added by writing a task specification rather than RL integration code. With the same framework and only the task specification changed, we trained software-engineering, deep-research, and general-tool-use agents and improved task reward in all three; decoupling the rollout stages further raised throughput by 2.8× without altering the training signal. The framework is open source, and we hope it lowers the cost of training agents over the growing MCP ecosystem.

References

Anthropic. 2024. Introducing the Model Context Protocol. https://www.anthropic.com/news/model-context-protocol.

Shiyi Cao, Dacheng Li, Fangzhou Zhao, Shuo Yuan, Sumanth Hegde, Connor Chen, Charlie Ruan, Tyler Griggs, Shu Liu, Eric Tang, Richard Liaw, Philipp Moritz, Matei Zaharia, Joseph Gonzalez, and Ion Stoica. 2025. Skyrl-agent: Efficient rl training for multi-turn llm agent. *ArXiv*, abs/2511.16108.

Thibault Le Sellier de Chezelles, Maxime Gasse, Alexandre Lacoste, Alexandre Drouin, Massimo Caccia, L'eo Boisvert, Megh Thakkar, Tom Marty, Rim Assouel, Sahar Omidi Shayegan, Lawrence Keunho Jang, Xing Han Lù, Ori Yoran, Dehan Kong, Frank F. Xu, Siva Reddy, Quentin Cappart, Graham Neubig, Ruslan Salakhutdinov, and Nicolas Chapados. 2024. The browsergym ecosystem for web agent research. *ArXiv*, abs/2412.05467.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Jun-Mei Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiaoling Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, and 179 others. 2025. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645:633 – 638.

Wei Fu, Jiaxuan Gao, Xu Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guoyizhe Wei, Jun Mei, Jiashun Wang, Tongkai Yang, Binhang Yuan, and Yi Wu. 2025. Areal: A large-scale asynchronous reinforcement learning system for language reasoning. *ArXiv*, abs/2505.24298.

Jian Hu, Xibin Wu, Weixun Wang, Songlin Jiang, Dehao Zhang, Yu Cao, OpenLLMAI Team, Netease Fuxi, AI Lab, and Alibaba Group. 2024. Openrlhf: An easy-to-use, scalable and high-performance rlhf framework. *ArXiv*, abs/2405.11143.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. 2025. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *ArXiv*, abs/2504.07164.

Dongfu Jiang, Yi Lu, Zhuofeng Li, Zhiheng Lyu, Ping Nie, Haozhe Wang, Alex Su, Hui Chen, Kai Zou, Chao Du, Tianyu Pang, and Wenhu Chen. 2025. Verltool: Towards holistic agentic reinforcement learning with tool use. *ArXiv*, abs/2509.01055.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *ArXiv*, abs/2310.06770.

Rui Lu, Zhenyu Hou, Zihan Wang, Hanchen Zhang, Xiao Liu, Yujiang Li, Shi Feng, Jie Tang, and Yuxiao Dong. 2025. Deepdive: Advancing deep search agents with knowledge graphs and multi-turn rl. *ArXiv*, abs/2509.10446.

Xufang Luo, Yuge Zhang, Zhiyuan He, Zilong Wang, Siyun Zhao, Dongsheng Li, Luna K. Qiu, and Yuqing Yang. 2025a. Agent lightning: Train any ai agents with reinforcement learning. *ArXiv*, abs/2508.03680.

Ziyang Luo, Zhiqi Shen, Wenzhuo Yang, Zirui Zhao, Prathyusha Jwalapuram, Amrita Saha, Doyen Sahoo, Silvio Savarese, Caiming Xiong, and Junnan Li. 2025b. Mcp-universe: Benchmarking large language models with real-world model context protocol servers. *ArXiv*, abs/2508.14704.

MiroMind AI. 2025. MiroRL: An MCP-first reinforcement learning framework for deep research agents. https://github.com/MiroMindAI/MiroRL.

OpenAI. 2025. gpt-oss-120b and gpt-oss-20b model card. https://openai.com/index/introducing-gpt-oss/.

OpenPipe. 2025. MCP-RL: Teach any model to master any MCP server. https://art.openpipe.ai/features/mcp-rl.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Jun-Mei Song, Mingchuan Zhang, Y. K. Li, Yu Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *ArXiv*, abs/2402.03300.

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

SWE-agent Team. 2025. mini-swe-agent: The 100-line AI agent that solves GitHub issues. https://github.com/SWE-agent/mini-swe-agent.

Sijun Tan, Michael Luo, Colin Cai, Tarun Venkat, Kyle Montgomery, Aaron Hao, Tianhao Wu, Arnav Balyan, Manan Roongta, Chenguang Wang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. 2025. rLLM: A framework for post-training language agents. https://github.com/rllm-org/rllm. Berkeley Sky Computing Lab.

THUDM. 2025. slime: An LLM post-training framework for RL scaling. https://github.com/THUDM/slime.

Wenhao Wang, Peizhi Niu, Zhao Xu, Zhaoyu Chen, Jian Du, Yaxin Du, Xianghe Pang, Keduan Huang, Yanfeng Wang, Qiang Yan, and Siheng Chen. 2025. Mcp-flow: Facilitating llm agents to master real-world, diverse and scaling mcp tools. *ArXiv*, abs/2510.24284.

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

Hyperparameter	Value
Policy model	`gpt-oss-20b`
RL algorithm	GRPO
Learning rate	2×10^{-6}
Batch Size	16
Trajectories per task (n)	8
Reward	binary, from evaluator
Optimizer	AdamW
KL coefficient	0.0
Rollout temperature	0.7
Max seq. len	128k

Table 2: Hyperparameters shared across all three domains.

	Software eng.	Deep research	General tool use
Dataset	R2E-Gym	DeepDive	AgentWorldModel-1k
Tasks	4574	2234	3315
Agent	mini-swe-agent	ReAct	ReAct
Max turns	100	40	40
Dataset link	HF	HF	HF

Table 3: Training settings that differ by domain.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *ArXiv*, abs/2210.03629.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, and 16 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. *ArXiv*, abs/2503.14476.

Hanchen Zhang, Xiao Liu, Bowen Lv, Xueqiao Sun, Bohao Jing, Iat Long Iong, Zhenyu Hou, Zehan Qi, Hanyu Lai, Yifan Xu, Rui Lu, Hongning Wang, Jie Tang, and Yuxiao Dong. 2025. Agentrl: Scaling agentic reinforcement learning with a multi-turn, multi-task framework. *ArXiv*, abs/2510.04206.

A Availability and License

MCP-U RL is released as open source under the Apache-2.0 license. The release includes the three layers described in the paper, and runnable configurations for the three domains of Section 4.1, so every training curve reported here can be reproduced from a task specification and a single launch command. Links to the code and project page are given with the author information on the first page.

B Training Details

All runs use the open-weight `gpt-oss-20b` (OpenAI, 2025) policy trained with GRPO (Shao et al., 2024). Shared hyperparameters are in Table 2; settings that differ by domain are in Table 3.

C Rollout Experiment Details

The throughput experiments of Section 4.2 run on the software-engineering workload on 3×H200 GPUs without high-speed NVLink interconnect, so weights and trajectories move between GPUs over ordinary network transfer rather than a fast GPU fabric. For the per-stage concurrency sweep (Figure 7), the acquire stage is fixed at 24 workers and the number of run workers is varied over $\{24, 48, 64, 96, 120\}$, with the in-flight capacity (number of live environments) held fixed at 120 so the memory budget does not change. Throughput is measured as mean rollout tokens per second over 5 steps after warm-up.

For the placement comparison (Table 1), both placements run at matched concurrency (120). The colocated placement shares GPUs between rollout and the policy update and alternates between them; the fully asynchronous placement runs rollout and the update on separate GPUs and streams trajectories through a queue. The asynchronous placement is *one-step off-policy*: the update is applied to trajectories generated under a policy at most one step stale, which keeps it close to on-policy while overlapping rollout and the update. Time per step is the mean over the first 5 steps.

D Environments and MCP Servers

Each domain reaches its tools through one or more MCP servers, and the environment-orchestration layer provisions them on either the Docker or the daemonless backend depending on how environments are created (Section 3.1).

- **Software engineering.** Each task carries its own repository image, so environments are provisioned on demand on the daemonless (Apptainer) backend, built from the task's existing Dockerfile. The agent is given a single shell tool exposed over MCP; the hidden test suite runs inside the same container.
- **Deep research.** Every task is stateless and shares the same two MCP servers, a Google Search server (`serper-search`) and a page-reading server (`jina-scrape`), on the Docker backend. Because the configuration is shared, the pool is pre-warmed and environments are reset and reused across tasks.
- **General tool use.** Each AgentWorldModel environment is code- and database-backed and

exposed over MCP, provisioned on demand on the daemonless (Apptainer) backend built from a Dockerfile.

E Evaluators

Rewards come entirely from each task's evaluator, run against the environment after the episode.

- **Software engineering.** The evaluator runs the task's hidden test suite inside the container the agent edited and returns a binary reward on whether all tests pass.
- **Deep research.** The evaluator compares the agent's final answer to a reference answer with an LLM judge, returning a binary correctness reward.
- **General tool use.** Each task carries a deterministic SQL or code verifier that queries the application's database state after the episode and returns a binary reward on whether it matches the task's expected state.