TRACE: Capability-Targeted Agentic Training

Hangoo Kang [*1] **Tarun Suresh** [*1] **Jon Saad-Falcon** [1] **Azalia Mirhoseini** [1]

Abstract

Models fail to complete agentic tasks often due to missing core capabilities. However, mainstream approaches for addressing these failures often rely on fine-tuning directly on the target environments or generating synthetic data that is not targeted to the model's actual capability deficits, resulting in low sample efficiency and limited generalizability. We introduce TRACE (Turning Recurrent Agent failures into Capability-targeted training Environments), an end-to-end system for environment-specific agent self-improvement. TRACE contrasts successful and failed trajectories to automatically identify lacking capabilities, synthesizes a targeted training environment for each that rewards whether the capability was exercised, trains a LoRA adapter via RL on each synthetic environment, and then trains a mixture-of-experts (MoE) model over the capability adapters. TRACE can be effectively applied across different environments, improving over the base agent by **+15.3 points** on τ^2-Bench (customer service) and **+15 points** Pass@1 on SWE-bench Verified (software engineering), outperforming the strongest external baselines, GEPA and SWE-RL, by **+8.6 points** and **+8.4 points**, respectively. In addition, TRACE is more sample-efficient than strong finetuning baselines: using less than one-fourth the number of rollouts, TRACE outperforms the best-performing baselines, GRPO and GEPA, and achieves higher final accuracy by **+10.4** and **+8.6** points on τ^2-Bench. The code is available at `https://github.com/ScalingIntelligence/TRACE.git`.

1. Introduction

Large Language Models (LLMs) are increasingly deployed in agentic environments, such as customer-service workflows (Yao et al., 2024) and coding platforms (Jimenez et al., 2024; Xu et al., 2025), where they must complete multi-step tasks through tool use, reasoning, and interaction with an environment. When models fail in these settings, their failures are often not isolated to a single task instance. Instead, they frequently reflect missing core capabilities that are required across many different tasks. We define a *capability* as performing one or more actions in a trajectory that are necessary for successfully solving some subset of task instances in the target environment. For example, in a customer-service environment, retrieving the correct customer record is a capability necessary for tasks such as canceling a flight, changing a seat, or verifying a reservation.

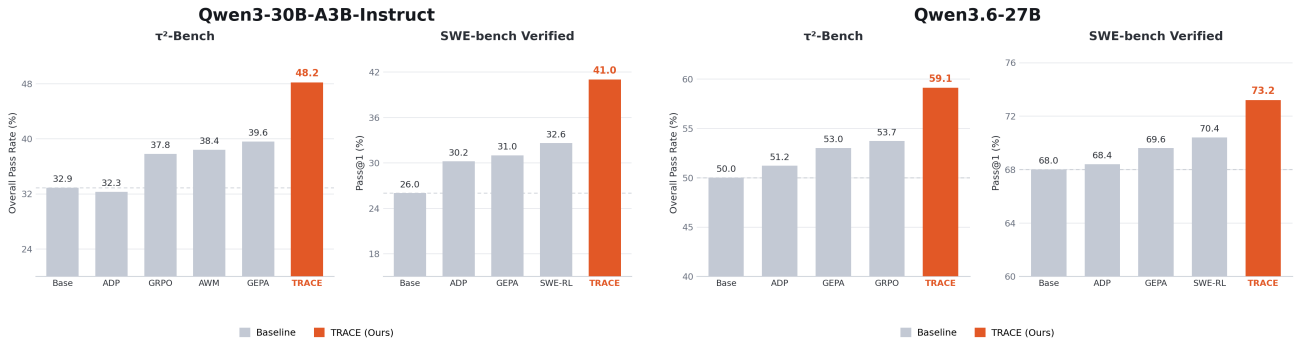
A natural approach is to finetune the LLM directly on the target environment using Reinforcement Learning (RL) or Supervised Fine-Tuning (SFT). However, direct training often provides supervision only at the level of task outcomes or task-specific trajectories, and does not explicitly reveal which underlying capabilities the agent lacks. The model must then implicitly infer shared capabilities from sparse and heterogeneous task-level feedback, making learning sample-inefficient and limiting generalization across task instances.

Another line of work scales synthetic RL environments and training data (Sullivan et al., 2025; Wang et al., 2026a; Fang et al., 2025a; Tu et al., 2026; Song et al., 2026a). While these approaches increase the amount and diversity of training data, the generated data is often not targeted to the model's actual capability deficits, so training may spend substantial compute on behaviors the model already performs well or on capabilities irrelevant to the target failures, yielding limited sample efficiency and weak environment-specific improvement.

In this work, we introduce TRACE (Turning Recurrent

[*]Equal contribution [1]Stanford University. Correspondence to: Hangoo Kang <hangook@stanford.edu>, Tarun Suresh <tsuresh@stanford.edu>.

Proceedings of the Failure Modes in Agentic AI Workshop at ICML 2026.



(a) Qwen3-30B-A3B-Instruct *(b)* Qwen3.6-27B

Figure 1. TRACE outperforms the base agent and all baselines on both τ^2-Bench (overall pass rate) and SWE-bench Verified Verified (Pass@1), across two agent backbones.

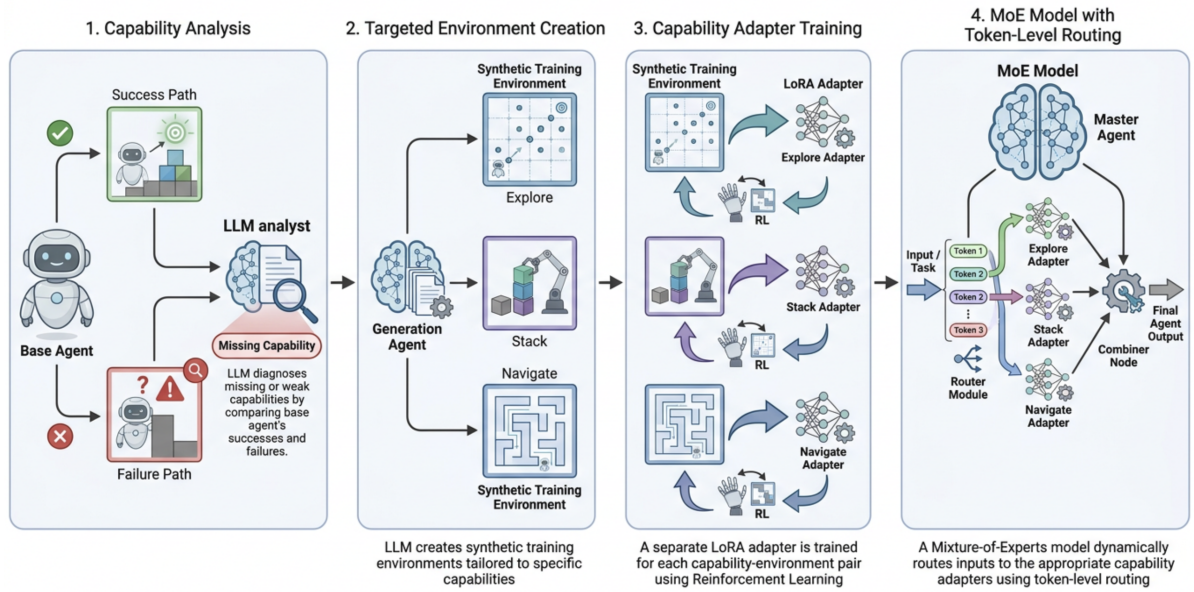


Figure 2. Overview of TRACE, an end-to-end system for automated environment-specific agent self-improvement. TRACE automatically identifies the specific capabilities that an agent lacks and synthesizes targeted training environments for each capability.

Agent failures into Capability-targeted training Environments), an end-to-end system for environment-specific agent self-improvement. Rather than training directly on the target environment or relying on untargeted synthetic data, TRACE uses the agent's own successful and failed trajectories to identify the capabilities whose absence explains recurring failures. It then synthesizes capability-targeted training environments that isolate and reward exercising each missing capability, trains a LoRA adapter via RL on each synthetic environment, and composes the resulting adapters through a Mixture-of-Experts (MoE) model.

TRACE is designed around three objectives:

1. **Identifying high-impact capability deficits.** The agent may fail for many reasons, so it is important to automatically determine which missing capabilities most clearly distinguish failed trajectories from successful ones and prioritize those that account for the largest number of failures.
2. **Synthesizing capability-targeted training environments.** For each identified deficit, the agent needs a training environment that directly isolates and rewards exercising the missing capability while preserving the target environment's interface.
3. **Learning and composing multiple capabilities.** The agent must acquire multiple identified capabilities from their respective synthetic environments and apply the appropriate capabilities to each task instance in the target environment.

Figure 2 shows an overview of TRACE, which consists of four steps. In step 1, the base agent generates rollouts in

the target environment, and an LLM-based analysis agent contrasts successful and failed trajectories to identify capabilities that distinguish the two, ranking them by failure coverage. In step 2, for each retained capability, an LLM-based generation agent uses the capability description and failed trajectories to construct a synthetic environment that isolates the missing capability while preserving the target environment's interface (tool schemas, interaction protocol, and output format); task instances are procedurally generated from random seeds, and whether the capability is exercised is verified automatically from tool arguments, state changes, or final outputs. In step 3, we train a LoRA adapter on each synthetic environment using RL, and in step 4, we train a Mixture-of-Experts (MoE) (Shazeer et al., 2017) model over the capability adapters.

Our contributions are as follows:

- We propose TRACE, an end-to-end system for environment-specific agent self-improvement. TRACE contrasts successful and failed trajectories from the target environment to automatically identify the specific capabilities the agent lacks, then synthesizes targeted training environments that teach the agent to exercise those capabilities.
- We show that TRACE can be effectively applied across different agentic environments. On τ^2-Bench (Barres et al., 2025), a customer-service benchmark, TRACE achieves a **48.2% overall pass rate**, improving over the base agent by **+15.3 points** and outperforming the strongest external baseline, GEPA, by **+8.6 points** (Table 1). On SWE-bench Verified (Jimenez et al., 2024), a software-engineering benchmark, TRACE reaches **41% Pass@1**, improving over the base agent by **+15 points** and outperforming the strongest external baseline, SWE-RL, by **+8.4 points** (Table 2). Moreover, with Qwen3.6-27B, TRACE attains **73.2% Pass@1** on SWE-bench Verified Verified, outperforming much larger frontier models, including **GPT-5.2-Codex** (72.8%), **GLM 5**, and **Claude 4.5 Sonnet**, on the public SWE-bench leaderboard[1], despite using a substantially smaller open-weight model.
- We demonstrate that TRACE is more sample-efficient than strong finetuning baselines. With Qwen3-30B-A3B-Instruct, TRACE uses only 2–4 capability-specific LoRA adapters, each updating only 5.3% of the model parameters, yet improves performance on τ^2-Bench by **+15.3 points** and Pass@1 on SWE-bench Verified by **+15 points**. In addition, using less than one-fourth the number of rollouts, TRACE outperforms the best-performing baselines, GRPO (Shao et al., 2024) and GEPA (Agrawal et al., 2026), and achieves higher final accuracy by **+10.4** and **+8.6** points on τ^2-Bench, respectively.
- We show that training on capability-targeted environments outperforms only adding capabilities to the prompt. With Qwen3-30B-A3B-Instruct, training on capability-targeted environments yields a **+8.6 point** improvement on τ^2-Bench and **+10 point** improvement in Pass@1 on SWE-bench Verified over prompt-based optimization (Figure 3a).

[1] https://www.swebench.com/

2. Related Work

LLM Agents and Interactive Environments. Autonomous LLM agents are increasingly deployed in complex, multi-turn environments, where rigorous evaluation requires mastering specific tool interfaces and interaction protocols, driving the adoption of benchmarks such as τ^2-Bench, SWE-bench (Jimenez et al., 2024), WebArena (Zhou et al., 2024), WorkArena(++) (Drouin et al., 2024; Boisvert et al., 2024), Terminal-Bench (Merrill et al., 2026), and TheAgentCompany (Xu et al., 2024).

Agentic Reinforcement Learning and Synthetic Data. Recent alignment methods utilize in-the-wild device control (Bai et al., 2024), implicit step rewards (Liu et al., 2025), or verifiable multi-turn RL (Gao et al., 2026; Zhang et al., 2025). A common capability-acquisition strategy scales training through procedurally synthesized environments (Sullivan et al., 2025) (e.g., AWM (Wang et al., 2026b), EnvScaler (Song et al., 2026a), ScaleEnv (Tu et al., 2026)) or unified public trajectories(e.g. ADP (Song et al., 2026b)). While effective for general capabilities (Yang et al., 2025; Fang et al., 2025b), these approaches overlook model-specific failures. TRACE diverges by using contrastive analysis to identify missing capabilities, synthesizing training environments to isolate and reward correct execution of each capability.

LoRA Merging and Routing. Recent work on model merging composes task-specific adaptations into a single checkpoint (Ilharco et al., 2023a; Yadav et al., 2023b; Panariello et al., 2025a), while a complementary direction instead routes over mixtures of LoRA experts (Luo et al., 2024; Cao et al., 2026; Dou et al., 2024; Wu et al., 2024; Zadouri et al., 2024; Gao et al., 2024; Zhao et al., 2025). Unlike methods where experts emerge implicitly from multi-task data, TRACE's adapters are independently trained on synthetic environments that each isolate a single, contrastively identified capability, yielding an interpretable set of experts. Such advances in adapter architectures and MoE training are orthogonal to TRACE and can be incorporated into our pipeline for further gains.

3. Method

3.1. Preliminaries

Agentic environment. We represent an agentic environment as a tuple $\mathcal{E} = (\mathcal{X}_\mathcal{E}, P_\mathcal{E}, R_\mathcal{E}, y_\mathcal{E})$, where $\mathcal{X}_\mathcal{E}$ is the set of task instances, $P_\mathcal{E}$ defines the interaction dynamics, $R_\mathcal{E}(x, \tau)$ is a trajectory-level reward and can either be discrete or continuous, and $y_\mathcal{E}(x, \tau) \in \{0, 1\}$ is a binary label indicating whether τ succeeded or not.

A task instance $x \sim \mathcal{X}_\mathcal{E}$ determines the initial observation o_1 presented to the agent. An LLM policy π_θ then selects actions conditioned on the interaction history $h_t = (o_1, a_1, \ldots, o_t)$ via $a_t \sim \pi_\theta(\cdot \mid h_t)$, where each action is a variable-length token sequence that may include natural-language reasoning and tool calls. The environment returns an observation o_{t+1} after each action, and the episode continues for at most T steps, producing a trajectory $\tau = (o_1, a_1, \ldots, o_T, a_T, o_{T+1})$. At episode end, the environment assigns reward $R_\mathcal{E}(x, \tau)$ and success label $y_\mathcal{E}(x, \tau)$. We denote by $\mathcal{D} = \{(x_i, \tau_i, r_i, y_i)\}_{i=1}^N$ a dataset of N episodes.

Synthetic environment. A synthetic verifiable environment is an agentic environment $\mathcal{E}_s = (G_{\mathcal{E}_s}, P_{\mathcal{E}_s}, R_{\mathcal{E}_s}, y_{\mathcal{E}_s})$ whose task instances are produced by an explicit generator $G_{\mathcal{E}_s}$ and whose reward and success label can be evaluated automatically from the task instance and trajectory. Concretely, given a random seed z, the generator produces a task instance $x_s = G_{\mathcal{E}_s}(z)$ together with the associated environment configuration, such as the initial state, transition logic, and evaluation criteria. Interaction with $\mathcal{E}_s$ then produces a trajectory τ_s with reward $R_{\mathcal{E}_s}(x_s, \tau_s)$ and success label $y_{\mathcal{E}_s}(x_s, \tau_s)$. Because both generation and verification are algorithmic, synthetic verifiable environments can be scaled to produce large numbers of training instances with verifiable reward signal.

3.2. Problem Formulation

Our goal is to learn an LLM policy π_θ for a target agentic environment $\mathcal{E}$ that maximizes expected return:

$$J(\pi_\theta; \mathcal{E}) \;=\; \mathbb{E}_{x \sim \mathcal{X}_\mathcal{E},\, \tau \sim p_{\pi_\theta}(\cdot \mid x, \mathcal{E})} \big[R_\mathcal{E}(x, \tau) \big],$$

Here, x is a task instance, τ is a corresponding trajectory induced by executing π_θ in $\mathcal{E}$ on x, and $R_\mathcal{E}(x, \tau)$ is the trajectory-level reward.

A central challenge is that optimization in the target environment provides limited failure attribution. The policy must infer what is necessary for success across task instances only indirectly from the training signal, making optimization inefficient. We formalize such underlying necessities as *capabilities*.

Definition 1 (Capability)**.** *A capability consists of performing one or more actions in a trajectory that are necessary for successfully solving some subset of task instances in the environment. For $X_c \subseteq X_\mathcal{E}$, let $\nu_c(x, \tau) = 1$ indicate that capability c is exercised in trajectory τ for task instance x. Then, for every $x \in X_c$ and a corresponding trajectory τ, $y_\mathcal{E}(x, \tau) = 1 \;\Rightarrow\; \nu_c(x, \tau) = 1$.*

Examples.

1. In a customer-service environment, retrieving the correct customer record is a capability, since it is necessary for tasks such as canceling a flight, changing a seat, or verifying a reservation.
2. In a coding environment, correctly locating the relevant function or file is a capability, since it is necessary for tasks such as fixing a bug, adding a feature, or updating an API call.

However, in $\mathcal{E}$, $R_\mathcal{E}(x, \tau)$ does not generally provide an explicit signal of whether $\nu_c(x, \tau) = 1$ for each capability c, making it difficult to isolate and train any single capability directly. To address this, we introduce:

Definition 2 (Capability-targeted synthetic environment)**.** *For a capability c, a capability-targeted synthetic environment is a synthetic verifiable environment $\mathcal{E}_s^c = (G_c, P_c, R_c, y_c)$ constructed to satisfy three properties:*

1. *Every task instance x_c generated by G_c ensures that exercising c is necessary for success.*
2. *P_c preserves the aspects of $P_\mathcal{E}$ that are relevant to exercising c, such as tool schemas, state representation, and policy constraints.*
3. *For any x_c and a corresponding trajectory τ_c, the reward $R_c(x_c, \tau_c)$ and success label $y_c(x_c, \tau_c)$ are automatically computable from (x_c, τ_c). Higher reward and success depend primarily on whether c is exercised in τ_c.*

Because each $\mathcal{E}_s^c$ isolates a single capability, its reward signal is generally denser and more attributable than that of $\mathcal{E}$. This decomposition reduces the problem of optimizing $J(\pi_\theta; \mathcal{E})$ to three tractable subproblems: (1) identifying the capabilities that π_θ lacks in $\mathcal{E}$, i.e., those for which $\nu_c(x, \tau) = 0$ across many task instances; (2) synthesizing a capability-targeted environment $\mathcal{E}_s^c$ for each identified capability; and (3) training π_θ on each $\mathcal{E}_s^c$ to improve performance back in $\mathcal{E}$. In the following subsections, we describe TRACE, an end-to-end self-improving system that tackles each of these subproblems.

3.3. Automated Capability-Targeted Synthetic Environment Generation Pipeline

Given a representative dataset $\mathcal{D} = \{(x_i, \tau_i, r_i, y_i)\}_{i=1}^N$ collected by rolling out π_θ in the target environment $\mathcal{E}$, TRACE

introduces an agentic pipeline that automatically: (i) identifies capabilities that distinguish failed trajectories from successful ones and (ii) constructs a capability-targeted synthetic environment for each.

Contrastive capability identification. Given $\mathcal{D}$, our goal is to identify capabilities that π_θ lacks and prioritize those responsible for a substantial fraction of failures, so that they can be targeted for training. We first split $\mathcal{D}$ into successful and failed subsets according to the success label y_i: $\mathcal{D}^+ = \{(x_i, \tau_i, r_i, y_i) \in \mathcal{D} \mid y_i = 1\}$ and $\mathcal{D}^- = \{(x_i, \tau_i, r_i, y_i) \in \mathcal{D} \mid y_i = 0\}$.

Then, to ensure capability definitions remain consistent across multiple independent analysis runs, we structure the identification process into two phases: discovery and labeling. First, in the *discovery phase*, an LLM-based analysis agent examines the tool calls, tool results, and final responses across the trajectories to induce a canonical dictionary of candidate recurring capabilities $\mathcal{C}$. Each capability $c \in \mathcal{C}$ is assigned a fixed name and natural-language description. Second, in the *labeling phase*, the analysis agent uses this fixed dictionary to systematically evaluate the dataset. For each trajectory τ_i and capability $c \in \mathcal{C}$, the agent predicts a label $\ell_c(x_i, \tau_i) \in \{\texttt{NA}, \texttt{PRESENT}, \texttt{LACKING}\}$. Here, NA means the analysis agent predicts $x_i \notin X_c$ (c is not necessary for x_i), PRESENT means the analysis agent predicts $x_i \in X_c$ and $\nu_c(x_i, \tau_i) = 1$ (c is necessary for x_i and exercised in τ_i), and LACKING means the analysis agent predicts $x_i \in X_c$ and $\nu_c(x_i, \tau_i) = 0$ (c is necessary for x_i but not exercised in τ_i). For each capability $c \in \mathcal{C}$, we estimate its error rate on successful and failed trajectories:

Let $\ell_i^c := \ell_c(x_i, \tau_i)$. Then

$$\widehat{\mathrm{ER}}^+(c) = \frac{\sum_i \mathbf{1}[\ell_i^c = \textsc{Lacking},\ y_i = 1]}{\sum_i \mathbf{1}[\ell_i^c \neq \mathrm{NA},\ y_i = 1]},$$
$$\widehat{\mathrm{ER}}^-(c) = \frac{\sum_i \mathbf{1}[\ell_i^c = \textsc{Lacking},\ y_i = 0]}{\sum_i \mathbf{1}[\ell_i^c \neq \mathrm{NA},\ y_i = 0]}.$$

We then define the contrastive gap as $\widehat{\Delta}(c) = \widehat{\mathrm{ER}}^-(c) - \widehat{\mathrm{ER}}^+(c)$. which measures how much more often c is lacking in failed trajectories than in successful ones. This contrastive criterion is more robust than analyzing failures alone: it filters out capabilities with uniformly low success rates, which often reflect task ambiguity or annotation noise rather than a trainable deficit, and it excludes capabilities that occasionally fail but do not meaningfully distinguish successful from unsuccessful trajectories.

We also compute the coverage of each capability over failed trajectories $\widehat{\mathrm{Cov}}(c) = 1/|\mathcal{D}^-| \sum_{i=1}^{N} \mathbf{1}[\ell_c(x_i, \tau_i) = \textsc{Lacking} \wedge y_i = 0]$ and retain $\mathcal{C}^* = \{c \in \mathcal{C} \mid \widehat{\Delta}(c) \geq \delta, \widehat{\mathrm{Cov}}(c) \geq \rho\}$, representing capabilities that are both strongly contrastive and cover a substantial fraction of failures.

To improve robustness, we repeat the analysis except the discovery step multiple times and retain only capabilities selected consistently across runs. In our experiments, one analysis pass takes approximately 5 minutes, which is marginal compared to training overhead. We set $\rho = 0.10$ (a retained capability must account for at least 10% of failed trajectories) and $\delta = 0.20$ (its error rate must be at least 20 percentage points higher on failed than on successful trajectories). We analyze and provide examples of the identified capabilities in Appendix J and Appendix K.

Environment synthesis. For each retained capability $c \in \mathcal{C}^*$, an LLM-based generation agent synthesizes a complete training environment $\mathcal{E}_s^c$. The agent receives the description of c together with the failure patterns identified during contrastive analysis, and produces a seeded task generator G_c, transition logic, and evaluation criteria that together define $\mathcal{E}_s^c$ while preserving the target environment's tool schemas, state representation, and policy constraints. Given a random seed z, the generator deterministically constructs a task instance $x = G_c(z)$, including synthetic user profiles, database records, and task-specific parameters, so that each seed yields a distinct scenario, preventing memorization and enabling diversity.

By construction, task success in $\mathcal{E}_s^c$ depends primarily on exercising c, which is verified automatically from tool arguments, state changes, or final outputs. For example, if c is *structured data reasoning*, the environment generates scenarios requiring the agent to search, filter, and cross-reference specific records within complex JSON databases (such as locating a specific flight routing or matching a retail item variant). For each seeded instance, the environment also generates a corresponding ground-truth solution. For that instance, the agent interacts in a multi-turn setting, using tools to query records, modify the database, and converse with a simulated user. The reward is computed by comparing the final database state against the ground truth using hash-based consistency checks and by verifying that the agent communicates the correct final result. Because each environment isolates a single capability, the reward $R_{\mathcal{E}_s^c}$ is denser and more attributable than that of the original environment, where success may depend on many capabilities at once. Examples of synthesized environments and the generation prompt are in Appendix F and Appendix L.

3.4. Acquiring Capabilities via Reinforcement Learning

Given the family of synthetic environments $\{\mathcal{E}_s^c\}_{c \in \mathcal{C}^*}$ produced by the generation pipeline (§3.3), we train a separate low-rank adapter Δ_c (Hu et al., 2021) for each capability $c \in \mathcal{C}^*$ while keeping the base policy π_θ frozen.

We optimize each adapter with GRPO, a value-free on-

policy algorithm. At each iteration, $\pi_{\theta+\Delta_c}$ generates G groups of rollouts in $\mathcal{E}_s^c$. Within each group g, K trajectories $\{\tau_{g,1}, \ldots, \tau_{g,K}\}$ are sampled from the same seed z_g, so all rollouts share an identical initial state and differ only through stochastic decoding. Let $r_{g,k} = R_c(x_g, \tau_{g,k})$ denote the reward function, where $x_g = G_c(z_g)$. GRPO normalizes rewards within each group to obtain a trajectory-level advantage: $\hat{A}_{g,k} = \frac{r_{g,k} - \bar{r}_g}{\sigma_g + \epsilon}$, where $\bar{r}_g$ and σ_g are the within-group mean and standard deviation and ϵ is a small constant for numerical stability. This normalization makes the training signal invariant to reward scale across environments. Because reward is assigned at the trajectory level, all tokens in a rollout share the same advantage; groups in which all rollouts receive identical rewards are discarded, as they carry no learning signal. The adapter is then updated via the standard clipped surrogate objective (Schulman et al., 2017); full details are provided in Appendix B.

3.5. Composing Acquired Capabilities

The target environment $\mathcal{E}$ may require different capabilities across task instances. We consequently train a Mixture-of-Experts (MoE) model over the capability adapters.

We construct the MoE by composing the capability-specific adapters $\{\Delta_c\}_{c \in \mathcal{C}^*}$ at each adapted transformer block using token-level routing (Dou et al., 2024; Wu et al., 2024; Shazeer et al., 2017). At each adapted block ℓ, we stack the LoRA factors $\{(A_c, B_c)\}_{c \in \mathcal{C}^*}$ into a shared adapter bank and introduce a lightweight linear gate g_ℓ. For capability c, let $\Delta W_c = B_c A_c$ denote its low-rank weight update. Given hidden state h at block ℓ, the gate produces capability weights $p_{\ell,c}(h) = \mathrm{softmax}(g_\ell(h))_c$. For an adapted projection with frozen base weight W, the token-dependent update is

$$\Delta W_\ell(h) = \sum_{c \in \mathcal{C}^*} p_{\ell,c}(h)\, \Delta W_c,$$

and the projection output is $(W + \Delta W_\ell(h))h$. The base policy π_θ and all adapters are frozen; only the gates $\{g_\ell\}$ are trained. We train the gates with the objective

$$\mathcal{L} = \mathcal{L}_{\mathrm{CE}} + \lambda_{\mathrm{LB}}\, \mathcal{L}_{\mathrm{LB}},$$

where $\mathcal{L}_{\mathrm{CE}}$ is the token-level cross-entropy on the agent's response tokens and $\mathcal{L}_{\mathrm{LB}} = |\mathcal{C}^*| \sum_{c \in \mathcal{C}^*} f_c\, P_c$ is a load-balancing penalty that discourages expert collapse. Here, f_c is the fraction of batch tokens for which c has the largest gate probability, P_c is the average gate probability assigned to c, and λ_{LB} controls the strength of the regularizer. At inference, each token uses the single capability with the largest gate probability. We empirically compare this MoE composition against alternative consolidation strategies (adapter merging, multi-capability GRPO, SFT, and on-policy distillation) in Appendix D.

4. Experiments

4.1. Experimental Setup

Benchmarks & Evaluation. We evaluate on τ^2-Bench (Barres et al., 2025) (50 AIRLINE and 114 RETAIL tasks) to test policy-sensitive customer-service workflows, and SWE-bench Verified (Jimenez et al., 2024) (500 instances) to test software-engineering tasks that require resolving real GitHub issues. We report *pass rate* on τ^2-Bench (broken out by domain and aggregated over both) and *Pass@1* on SWE-bench Verified; on SWE-bench Verified, all methods—including TRACE's capability discovery and environment generation—train on the disjoint SWE-smith (Yang et al., 2025) split. We additionally evaluate on ToolSandBox (Lu et al., 2025), a stateful tool-use benchmark, in Appendix C. Formal metric definitions and decoding configuration are in Appendix G and Appendix A.

Training. We evaluate two agent backbones, Qwen3-30B-A3B-Instruct-2507 (Team, 2025) and Qwen3.6-27B (Qwen Team, 2026) (Table 1), optimizing with LoRA and GRPO. On τ^2-Bench we train the 4 identified capabilities (structured data reasoning, multi-step task completion, precondition verification, and tool-calling precision), and on SWE-bench Verified we train 3 capability-targeted synthetic environments. Full training hyperparameters, hardware, and parameter counts are in Appendix A and Appendix E.

Baselines. We compare against direct GRPO training on the target environment and two general-purpose synthetic-data methods, *Agent World Model (AWM)* and *Agent Data Protocol (ADP)*; on SWE-bench Verified we additionally compare against *SWE-RL* (Wei et al., 2025), an RL method specialized for software-engineering agents. To test whether dedicated training on the identified capabilities is necessary, we also compare against the inference-time prompt-optimization method GEPA, which refines TRACE's identified capability descriptions into the agent's system prompt. All training baselines and GEPA are run under matched training budgets for fair comparison with TRACE; extended baseline descriptions are in Appendix A.

4.2. Effectiveness of Method

Tables 1 and 2 summarize the main experimental results. TRACE consistently outperforms baselines across both evaluated benchmarks, notably improving upon the base model by +15.3 points on τ^2-Bench and +15 points Pass@1 on SWE-bench Verified, while beating the strongest external baselines by +8.6 points and +8.4 points, respectively. The same pattern holds with a stronger Qwen3.6-27B backbone on τ^2-Bench (Table 1): TRACE reaches a 59.1% overall pass rate, +9.1 points over the base model and +5.4 points over the strongest baseline. On SWE-bench Verified Verified, the Qwen3.6-27B variant of TRACE reaches 73.2%

Table 1. Pass rate on τ^2-Bench with two agent backbones. TRACE outperforms all baselines under both Qwen3-30B-A3B and Qwen3.6-27B.

Model	Method	Airline (%)	Retail (%)	Overall (%)
Qwen3-30B-A3B	Base	24.0	36.8	32.9
	GRPO on Target	32.0	40.4	37.8
	ADP	28.0	34.2	32.3
	AWM	32.0	41.2	38.4
	GEPA	38.0	40.4	39.6
	Single Capability GRPO (Ours)	34.0	43.0	40.3
	TRACE (Ours)	**44.0**	**50.0**	**48.2**
Qwen3.6-27B	Base	60.0	45.6	50.0
	GRPO on Target	66.0	48.2	53.7
	ADP	56.0	49.1	51.2
	GEPA	66.0	47.4	53.0
	Single Capability GRPO (Ours)	66.0	50.8	55.4
	TRACE (Ours)	**74.0**	**52.6**	**59.1**

Table 2. Pass@1 on SWE-bench Verified (500 instances) with two agent backbones.

Model	Method	Pass@1 (%)
Qwen3-30B-A3B	Base	26.0
	ADP	30.2
	GEPA	31.0
	SWE-RL	32.6
	Single Capability GRPO (Ours)	36.6
	TRACE (Ours)	**41.0**
Qwen3.6-27B	Base	68.0
	ADP	68.4
	GEPA	69.6
	SWE-RL	70.4
	Single Capability GRPO (Ours)	71.0
	TRACE (Ours)	**73.2**

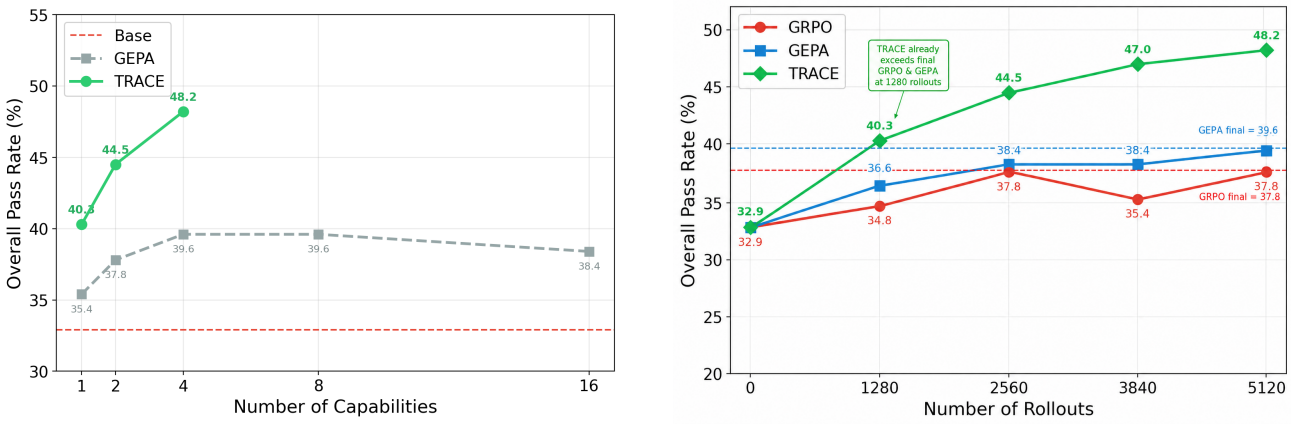
Pass@1 (Table 2), outperforming much larger frontier models, including GPT-5.2-Codex, GLM 5, and Claude 4.5 Sonnet on the public SWE-bench leaderboard.

Targeted vs. General-Purpose Training Environments. A single adapter trained on one synthesized capability environment achieves 40.3% on τ^2-Bench and 36.6% Pass@1 on SWE-bench Verified. This surpasses large-scale general-purpose methods such as AWM (38.4% on τ^2-Bench) and ADP (32.3% on τ^2-Bench, 30.2% on SWE-bench Verified). Notably, ADP's performance in the τ^2-Bench Retail domain is lower than the base model (34.2% vs. 36.8%). These results suggest that diagnosing specific missing capabilities and training on targeted synthetic environments yields higher performance than general-purpose training data.

Capability Training vs. Prompt Optimization. We evaluate the need for dedicated training versus injecting capability descriptions into the prompt optimized with GEPA. As shown in Tables 1 and 2, GEPA improves over the base model on both τ^2-Bench (overall pass rate) and SWE-bench Verified (Pass@1). However, it underperforms compared to training a single capability and TRACE with all capabilities on both benchmarks. This demonstrates that while prompting capability instructions is helpful, explicitly training the model to exercise those capabilities is necessary for stronger performance.

4.3. Method Scaling

Figure 3a shows how TRACE's overall pass rate scales with the number of trained capabilities. In contrast, GEPA, which optimizes capability descriptions in the prompt, plateaus after 4 capabilities. Figure 3b illustrates how performance on τ^2-Bench scales with the number of rollouts for TRACE, GEPA, and running GRPO on the target environment. TRACE demonstrates consistent, monotonic improvement, scaling from a base pass rate of 32.9% up to 48.2% at 5,120 rollouts. In contrast, GEPA plateaus early, reaching only 39.6%, while GRPO exhibits instability, dropping to 35.4% at 3,840 rollouts, and ultimately stalls at 37.8%. A consistent trend appears on ToolSandBox (Appendix C).



(a) Scaling the number of capabilities.

(b) Scaling the number of rollouts.

Figure 3. Method scaling on τ^2-Bench. (a) TRACE overall pass rate as the number of trained capabilities increases, compared to GEPA prompt optimization. (b) Pass rate as the number of rollouts increases for TRACE, GEPA, and GRPO on the target environment.

5. Conclusion

In this work, we propose TRACE, an end-to-end system for environment-specific agent self-improvement that automatically identifies lacking capabilities from the agent's trajectories, synthesizes targeted training environments for each capability, trains capability-specific LoRA adapters via RL, and then trains a Mixture-of-Experts (MoE) model over the capability adapters. TRACE can be effectively applied across different environments, improving over the base agent by **+15.3 points** on τ^2-Bench (customer service) and **+15 points** Pass@1 on SWE-bench Verified (software engineering), outperforming the strongest external baseline by **+8.6 points** and **+9 points**, respectively. Given the same number of rollouts, TRACE scales more efficiently than baselines, outperforming GRPO and GEPA by **+10.4** and **+8.6** points on τ^2-Bench, respectively.

Acknowledgements

We thank the Scaling Intelligence Lab and others for their constructive feedback during the composition of the paper. In particular, we would like to thank Debangshu Banerjee, Tanvir Bhathal, Alex Bloom, Andy Dimnaku, Simon Guo, Sid Jha, Hermann Kumbong, Jacky Kwok, Andrew Shi, and Shayan Talaei. We also thank Prime Intellect, Lambda Labs, Google, Modal, and IBM for providing compute resources that enabled our experiments.

References

Agrawal, L. A., Tan, S., Soylu, D., Ziems, N., Khare, R., Opsahl-Ong, K., Singhvi, A., Shandilya, H., Ryan, M. J., Jiang, M., Potts, C., Sen, K., Dimakis, A. G., Stoica, I., Klein, D., Zaharia, M., and Khattab, O. Gepa: Reflective prompt evolution can outperform reinforcement learning, 2026. URL https://arxiv.org/abs/2507.19457.

Bai, H., Zhou, Y., Cemri, M., Pan, J., Suhr, A., Levine, S., and Kumar, A. Digirl: Training in-the-wild device-control agents with autonomous reinforcement learning, 2024. URL https://arxiv.org/abs/2406.11896.

Barres, V., Dong, H., Ray, S., Si, X., and Narasimhan, K. τ^2-bench: Evaluating conversational agents in a dual-control environment, 2025. URL https://arxiv.org/abs/2506.07982.

Boisvert, L., Thakkar, M., Gasse, M., Caccia, M., Chezelles, T. L. S. D., Cappart, Q., Chapados, N., Lacoste, A., and Drouin, A. Workarena++: Towards compositional planning and reasoning-based common knowledge work tasks, 2024. URL https://arxiv.org/abs/2407.05291.

Cao, J., Fan, Z., Wang, Z., Lin, T., Zhao, Z., Yan, R., Zhang, W., Shao, F., Wang, H., Xiao, J., and Tang, S. Comol: Efficient mixture of lora experts via dynamic core space merging, 2026. URL https://arxiv.org/abs/2603.00573.

Dou, S., Zhou, E., Liu, Y., Gao, S., Zhao, J., Shen, W., Zhou, Y., Xi, Z., Wang, X., Fan, X., Pu, S., Zhu, J., Zheng, R., Gui, T., Zhang, Q., and Huang, X. Loramoe: Alleviating world knowledge forgetting in large language models via moe-style plugin, 2024. URL https://arxiv.org/abs/2312.09979.

Drouin, A., Gasse, M., Caccia, M., Laradji, I. H., Verme, M. D., Marty, T., Boisvert, L., Thakkar, M., Cappart, Q., Vazquez, D., Chapados, N., and Lacoste, A. Workarena:

How capable are web agents at solving common knowledge work tasks?, 2024. URL https://arxiv.org/abs/2403.07718.

Fang, R., Cai, S., Li, B., Wu, J., Li, G., Yin, W., Wang, X., Wang, X., Su, L., Zhang, Z., Wu, S., Tao, Z., Jiang, Y., Xie, P., Huang, F., and Zhou, J. Towards general agentic intelligence via environment scaling, 2025a. URL https://arxiv.org/abs/2509.13311.

Fang, R., Cai, S., Li, B., Wu, J., Li, G., Yin, W., Wang, X., Wang, X., Su, L., Zhang, Z., Wu, S., Tao, Z., Jiang, Y., Xie, P., Huang, F., and Zhou, J. Towards general agentic intelligence via environment scaling, 2025b. URL https://arxiv.org/abs/2509.13311.

Gao, C., Chen, K., Rao, J., Sun, B., Liu, R., Peng, D., Zhang, Y., Guo, X., Yang, J., and Subrahmanian, V. Higher layers need more lora experts, 2024. URL https://arxiv.org/abs/2402.08562.

Gao, J., Chen, J., He, C., Xu, S., Jin, D., and Wu, Y. From self-evolving synthetic data to verifiable-reward rl: Post-training multi-turn interactive tool-using agents, 2026. URL https://arxiv.org/abs/2601.22607.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Ilharco, G., Ribeiro, M. T., Wortsman, M., Gururangan, S., Schmidt, L., Hajishirzi, H., and Farhadi, A. Editing models with task arithmetic. In *International Conference on Learning Representations*, 2023a.

Ilharco, G., Ribeiro, M. T., Wortsman, M., Gururangan, S., Schmidt, L., Hajishirzi, H., and Farhadi, A. Editing models with task arithmetic. In *International Conference on Learning Representations*, 2023b.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

Liu, X., Wang, K., Wu, Y., Huang, F., Li, Y., Zhang, J., and Jiao, J. Agentic reinforcement learning with implicit step rewards, 2025. URL https://arxiv.org/abs/2509.19199.

Lu, J., Holleis, T., Zhang, Y., Aumayer, B., Nan, F., Bai, F., Ma, S., Ma, S., Li, M., Yin, G., Wang, Z., and Pang, R. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities, 2025. URL https://arxiv.org/abs/2408.04682.

Lu, K. and Lab, T. M. On-policy distillation. *Thinking Machines Lab: Connectionism*, 2025. doi: 10.64434/tml.20251026. https://thinkingmachines.ai/blog/on-policy-distillation.

Luo, T., Lei, J., Lei, F., Liu, W., He, S., Zhao, J., and Liu, K. Moelora: Contrastive learning guided mixture of experts on parameter-efficient fine-tuning for large language models, 2024. URL https://arxiv.org/abs/2402.12851.

Merrill, M. A., Shaw, A. G., Carlini, N., Li, B., Raj, H., Bercovich, I., Shi, L., Shin, J. Y., Walshe, T., Buchanan, E. K., Shen, J., Ye, G., Lin, H., Poulos, J., Wang, M., Nezhurina, M., Jitsev, J., Lu, D., Mastromichalakis, O. M., Xu, Z., Chen, Z., Liu, Y., Zhang, R., Chen, L. L., Kashyap, A., Uslu, J.-L., Li, J., Wu, J., Yan, M., Bian, S., Sharma, V., Sun, K., Dillmann, S., Anand, A., Lanpouthakoun, A., Koopah, B., Hu, C., Guha, E., Dreiman, G. H. S., Zhu, J., Krauth, K., Zhong, L., Muennighoff, N., Amanfu, R., Tan, S., Pimpalgaonkar, S., Aggarwal, T., Lin, X., Lan, X., Zhao, X., Liang, Y., Wang, Y., Wang, Z., Zhou, C., Heineman, D., Liu, H., Trivedi, H., Yang, J., Lin, J., Shetty, M., Yang, M., Omi, N., Raoof, N., Li, S., Zhuo, T. Y., Lin, W., Dai, Y., Wang, Y., Chai, W., Zhou, S., Wahdany, D., She, Z., Hu, J., Dong, Z., Zhu, Y., Cui, S., Saiyed, A., Kolbeinsson, A., Hu, J., Rytting, C. M., Marten, R., Wang, Y., Dimakis, A., Konwinski, A., and Schmidt, L. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

Panariello, A., Marczak, D., Magistri, S., Porrello, A., Twardowski, B., Bagdanov, A. D., Calderara, S., and van de Weijer, J. Accurate and efficient low-rank model merging in core space. In *Advances in Neural Information Processing Systems*, 2025a.

Panariello, A., Marczak, D., Magistri, S., Porrello, A., Twardowski, B., Bagdanov, A. D., Calderara, S., and van de Weijer, J. Accurate and efficient low-rank model merging in core space. In *Advances in Neural Information Processing Systems*, 2025b.

Qwen Team. Qwen3.6-27B: Flagship-level coding in a 27B dense model, April 2026. URL https://qwen.ai/blog?id=qwen3.6-27b.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y. K., Wu, Y., and Guo,

D. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, 2017. URL https://arxiv.org/abs/1701.06538.

Song, X., Chang, H., Dong, G., Zhu, Y., Dou, Z., and Wen, J.-R. Envscaler: Scaling tool-interactive environments for llm agent via programmatic synthesis, 2026a. URL https://arxiv.org/abs/2601.05808.

Song, Y., Ramaneti, K., Sheikh, Z., Chen, Z., Gou, B., Xie, T., Xu, Y., Zhang, D., Gandhi, A., Yang, F., Liu, J., Ou, T., Yuan, Z., Xu, F., Zhou, S., Wang, X., Yue, X., Yu, T., Sun, H., Su, Y., and Neubig, G. Agent data protocol: Unifying datasets for diverse, effective fine-tuning of llm agents. In *International Conference on Learning Representations*, 2026b.

Sullivan, M., Hartmann, M., and Koller, A. Procedural environment generation for tool-use agents, 2025. URL https://arxiv.org/abs/2506.11045.

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

Tu, D., Hao, H., Yang, H., Chen, Y., Zhang, Y.-K., Xia, Z., Yang, Y., Sun, Y., Liu, X., Shen, F., Gu, Q., Su, H., and Cai, X. Scaleenv: Scaling environment synthesis from scratch for generalist interactive tool-use agent training, 2026. URL https://arxiv.org/abs/2602.06820.

Wang, Z., Xu, C., Liu, B., Wang, Y., Han, S., Yao, Z., Yao, H., and He, Y. Agent world model: Infinity synthetic environments for agentic reinforcement learning. *arXiv preprint arXiv:2602.10090*, 2026a.

Wang, Z., Xu, C., Liu, B., Wang, Y., Han, S., Yao, Z., Yao, H., and He, Y. Agent world model: Infinity synthetic environments for agentic reinforcement learning, 2026b. URL https://arxiv.org/abs/2602.10090.

Wei, Y., Duchenne, O., Copet, J., Carbonneaux, Q., Zhang, L., Fried, D., Synnaeve, G., Singh, R., and Wang, S. I. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution, 2025. URL https://arxiv.org/abs/2502.18449.

Wu, X., Huang, S., and Wei, F. Mixture of lora experts, 2024. URL https://arxiv.org/abs/2404.13628.

Xu, F. F., Song, Y., Li, B., Tang, Y., Jain, K., Bao, M., Wang, Z. Z., Zhou, X., Guo, Z., Cao, M., Yang, M., Lu, H. Y., Martin, A., Su, Z., Maben, L., Mehta, R., Chi, W., Jang, L., Xie, Y., Zhou, S., and Neubig, G. Theagentcompany: Benchmarking llm agents on consequential real world tasks, 2024. URL https://arxiv.org/abs/2412.14161.

Xu, F. F., Song, Y., Li, B., Tang, Y., Jain, K., Bao, M., Wang, Z. Z., Zhou, X., Guo, Z., Cao, M., Yang, M., Lu, H. Y., Martin, A., Su, Z., Maben, L., Mehta, R., Chi, W., Jang, L., Xie, Y., Zhou, S., and Neubig, G. Theagentcompany: Benchmarking llm agents on consequential real world tasks, 2025. URL https://arxiv.org/abs/2412.14161.

Yadav, P., Tam, D., Choshen, L., Raffel, C., and Bansal, M. TIES-merging: Resolving interference when merging models. In *Advances in Neural Information Processing Systems*, 2023a.

Yadav, P., Tam, D., Choshen, L., Raffel, C., and Bansal, M. Ties-merging: Resolving interference when merging models. In *Advances in Neural Information Processing Systems*, 2023b.

Yang, J., Lieret, K., Jimenez, C. E., Wettig, A., Khandpur, K., Zhang, Y., Hui, B., Press, O., Schmidt, L., and Yang, D. Swe-smith: Scaling data for software engineering agents, 2025. URL https://arxiv.org/abs/2504.21798.

Yao, S., Shinn, N., Razavi, P., and Narasimhan, K. τ-bench: A benchmark for tool-agent-user interaction in real-world domains, 2024. URL https://arxiv.org/abs/2406.12045.

Zadouri, T., Üstün, A., Ahmadian, A., Ermiş, B., Locatelli, A., and Hooker, S. Pushing mixture of experts to the limit: Extremely parameter efficient moe for instruction tuning, 2024. URL https://arxiv.org/abs/2309.05444.

Zhang, H., Liu, X., Lv, B., Sun, X., Jing, B., Iong, I. L., Hou, Z., Qi, Z., Lai, H., Xu, Y., Lu, R., Wang, H., Tang, J., and Dong, Y. Agentrl: Scaling agentic reinforcement learning with a multi-turn, multi-task framework, 2025. URL https://arxiv.org/abs/2510.04206.

Zhao, Z., Zhou, Y., Zhu, D., Shen, T., Wang, X., Su, J., Kuang, K., Wei, Z., Wu, F., and Cheng, Y. Each rank could be an expert: Single-ranked mixture of experts lora for multi-task learning, 2025. URL https://arxiv.org/abs/2501.15103.

Zhou, S., Xu, F. F., Zhu, H., Zhou, X., Lo, R., Sridhar, A., Cheng, X., Ou, T., Bisk, Y., Fried, D., Alon, U., and Neubig, G. Webarena: A realistic web environment for building autonomous agents, 2024. URL https://arxiv.org/abs/2307.13854.

A. Experimental Setup Details

Metrics. On τ^2-Bench, *pass rate* is the fraction of tasks satisfying the benchmark's binary success criterion (correct task execution and correct user communication), broken out by the AIRLINE and RETAIL domains and aggregated over the union of both. On SWE-bench Verified, *Pass@1* is the fraction of instances whose generated patch resolves the issue and passes the held-out tests. Formal definitions are in Appendix G.

Decoding. Unless specified otherwise, we evaluate all models with greedy decoding (temperature 0.0) for deterministic head-to-head comparison, a maximum context length of 32,000 tokens, and a maximum of 50 interaction steps per episode.

Training hyperparameters. We optimize with AdamW (lr $= 10^{-5}$) for up to 40 iterations per capability, using a sampling temperature of 1.0 for rollouts. Training utilizes gradient checkpointing and distributed data parallelism across 4–8 A100-80GB GPUs. MoE training uses a load balancing loss coefficient $\lambda_{\text{LB}} = 0.05$. As detailed in Appendix E, each LoRA adapter introduces $\sim$1.6 B trainable parameters (5.3% of the backbone), and the trained MoE gate adds $491{,}760$ parameters in total.

Baselines. *AWM* performs RL in benchmark-independent synthetic environments for broad tool-use training, and *ADP* performs supervised fine-tuning on diverse public agent trajectories in a unified representation. For GEPA, we provide the capabilities identified by TRACE and allow it to refine their natural-language descriptions, which are then added to the agent's system prompt and optimized for the agent's pass rate in the target environment.

B. GRPO Details

Given the family of synthetic environments $\{\mathcal{E}_s^c\}_{c\in\mathcal{C}^*}$ produced by the environment generation pipeline (§3.3), we train a separate low-rank adapter (Hu et al., 2021) Δ_c for each capability $c \in \mathcal{C}^*$ while keeping the base model π_θ frozen.

GRPO training. We train with Group Relative Policy Optimization, an on-policy reinforcement learning algorithm that avoids learning an explicit value function. For each capability c, the adapted policy $\pi_{\theta+\Delta_c}$ interacts with the corresponding synthetic environment $\mathcal{E}_s^c$. At each training iteration, the policy generates G groups of rollouts. Within each group g, K trajectories $\{\tau_{g,1}, \ldots, \tau_{g,K}\}$ are sampled from the same environment seed z_g, so that all K rollouts begin from an identical initial state and differ only through stochastic decoding. Let $r_{g,k} = R_{\mathcal{E}_s^c}(x_g, \tau_{g,k})$ denote the reward assigned by the programmatic verifier of $\mathcal{E}_s^c$, where $x_g = G_{\mathcal{E}_s^c}(z_g)$. GRPO computes a group-relative normalized advantage for each trajectory,

$$\hat{A}_{g,k} = \frac{r_{g,k} - \bar{r}_g}{\sigma_g + \epsilon}, \qquad \bar{r}_g = \frac{1}{K}\sum_{j=1}^{K} r_{g,j}, \qquad \sigma_g = \sqrt{\frac{1}{K}\sum_{j=1}^{K}(r_{g,j} - \bar{r}_g)^2}, \tag{1}$$

where $\bar{r}_g$ and σ_g are the within-group mean and standard deviation. This normalization makes the training signal invariant to reward scale across capabilities and synthetic environments. Because reward is assigned at the trajectory level, all action tokens in a rollout receive the same advantage. Groups in which all K rollouts receive identical rewards are discarded, since they provide no learning signal.

The adapter parameters are updated using the clipped GRPO surrogate objective

$$\mathcal{L}_{\text{GRPO}} = -\frac{1}{|\mathcal{B}|}\sum_{(g,k)\in\mathcal{B}} \frac{1}{T_{g,k}}\sum_{t=1}^{T_{g,k}} \min\Big(\rho_t \hat{A}_{g,k}, \text{clip}(\rho_t, 1-\epsilon, 1+\epsilon)\hat{A}_{g,k}\Big), \tag{2}$$

where

$$\rho_t = \frac{\pi_{\theta+\Delta_c}(a_t \mid x, a_{<t})}{\pi^{\text{old}}_{\theta+\Delta_c}(a_t \mid x, a_{<t})}$$

is the per-token importance ratio between the current and rollout policies, a_t denotes the t-th token in the trajectory, $T_{g,k}$ is the number of tokens in trajectory $\tau_{g,k}$, $\mathcal{B}$ denotes the minibatch, and ϵ is the clipping threshold. As in PPO-style methods, clipping prevents excessively large policy updates when the estimated advantage is high.

On-policy rollout collection. At each iteration, we synchronize the current adapter Δ_c to a separate inference worker and collect rollouts on-policy from $\pi_{\theta+\Delta_c}$. The policy interacts with the task generator $G_{\mathcal{E}_s^c}$, receives observations, invokes tools, and continues until the episode terminates, after which the reward function $R_{\mathcal{E}_s^c}$ assigns the terminal reward. Because

rollouts are collected on-policy, the importance ratio ρ_t remains close to 1 at the beginning of each update, which improves optimization stability.

Table 3. Perfect Score (similarity = 1.0) and Mean Similarity on ToolSandBox

Model	Perfect	Mean Sim.
Base	19/129	0.411
ADP	19/129	0.422
GRPO on Target	22/129	0.519
AWM	20/129	0.504
GEPA	22/129	0.520
Single Capability GRPO (Ours)	22/129	0.514
TRACE (Ours)	**26 / 129**	**0.560**

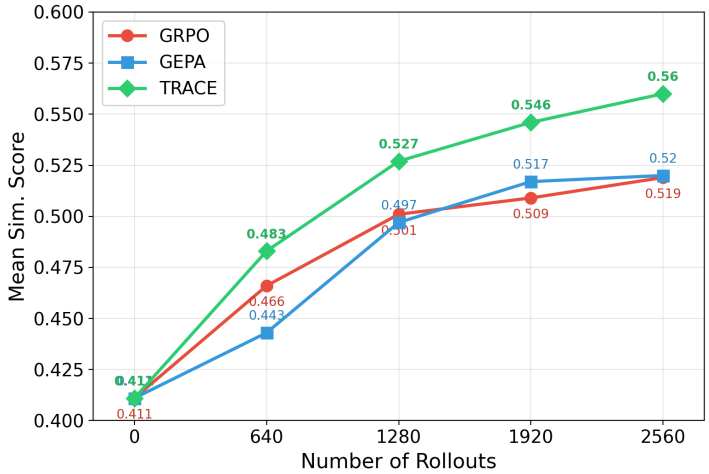


Figure 4. Mean similarity on ToolSandBox as the number of rollouts is scaled. As on τ^2-Bench, TRACE improves steadily to a 0.560 mean similarity score, outperforming both GEPA (0.520) and GRPO (0.519).

C. ToolSandbox Evaluation

We additionally evaluate TRACE on ToolSandBox (Lu et al., 2025) (129 base scenarios), a stateful, conversational tool-use benchmark. We report *mean similarity*, the average $[0, 1]$ trajectory score from the benchmark's milestone-based evaluator, and *perfect rate*, the fraction of scenarios scoring exactly 1.0 (formal definitions in Appendix G). For TRACE, the two capabilities trained are permission error recovery and datetime reasoning (Appendix K). As in the main paper, all baselines are run under matched training budgets and evaluated with greedy decoding.

Table 3 summarizes the results. TRACE achieves a mean similarity of 0.560 with 26 perfect scores, improving over the base model by $+0.149$ mean similarity and 7 additional perfect scores, and over the strongest external baseline by $+0.040$ mean similarity and 4 additional perfect scores. A single adapter trained on one synthesized capability environment already reaches 0.514, surpassing the general-purpose AWM (0.504) and ADP (0.422) baselines, while prompt optimization with GEPA reaches 0.520. These trends mirror those on τ^2-Bench and SWE-bench Verified: diagnosing and training the specific missing capabilities outperforms both general-purpose training data and prompt optimization.

D. Capability Consolidation

In Table 4, we compare TRACE MoE against alternatives to consolidate capabilities into one model. *CORE-TSV merge* (Panariello et al., 2025b; Ilharco et al., 2023b; Yadav et al., 2023a) combines independently trained, capability-specific adapters in a shared low-rank space using Core Space merging, task-vector composition, and TIES-style sign resolution. *Multi-Capability GRPO* trains a single LoRA adapter on a uniform mixture of all synthesized capability environments. *SFT Synthetic* trains a single LoRA adapter on successful trajectories collected from all the synthesized capability environments. *On-Policy Distillation* trains a LoRA teacher for each synthetic capability environment, then trains one student LoRA on a uniform mix by having the student generate on-policy rollouts and match the relevant teacher's token-level distributions (Lu & Lab, 2025). All consolidation approaches are trained under matched budgets. Only Multi-Capability GRPO (40.9%) marginally outperforms the best single adapter (40.3%), whereas MoE significantly outperforms it.

Table 4. Single Model Consolidation Approaches on τ^2-Bench

Approach	Airline (%)	Retail (%)	Overall (%)
Base	24.0	36.8	32.9
Single Capability GRPO	34.0	43.0	40.3
CORE-TSV	36.0	41.2	39.6
On Policy Distillation	28.0	42.1	37.8
SFT Synthetic	36.0	38.6	37.8
Multi Capability GRPO	38.0	42.1	40.9
TRACE	**44.0**	**50.0**	**48.2**

Table 5. Parameter counts for TRACE Model.

Component	Number of Parameters
Qwen3-30B-A3B backbone (frozen)	$\sim$30.5 B
Capability LoRA adapter (each)	$\sim$1.6 B
MoE gate (Linear $d \to \lvert\mathcal{C}^*\rvert$, one per block)	491,760

E. Parameter Counts

Table 5 breaks down the parameters of each TRACE component. The Qwen3-30B-A3B backbone ($\sim$30.5 B parameters) remains frozen throughout the entire pipeline. Each capability-specific LoRA adapter introduces $\sim$1.6 B trainable parameters (corresponding to 5.3% of the backbone) and is optimized independently on its respective synthetic environment during the RL stage (§3.4). At the composition stage (§3.5), the backbone and all $|\mathcal{C}^*|$ capability adapters are frozen, and the only trainable parameters are the MoE gates: one linear map from the block's hidden dimension d to the $|\mathcal{C}^*|$ capability logits per adapted transformer block, totaling $491{,}760$ parameters across all blocks.

Although TRACE stores all $|\mathcal{C}^*|$ adapters, inference is sparse: at each token, the MoE gate selects the top-k capabilities, so only k of the adapters are active. In our experiments we find $k{=}1$ works well, holding the per-token active parameter count to the backbone's active footprint plus a single adapter.

F. Environment Generation Prompt

System Prompt: RL Environment Designer

You are an RL environment designer. Your job is to create a GRPO-compatible training environment that teaches a language model a specific skill through procedurally generated scenarios with shaped rewards.

Target Skill

- {SKILL_DESCRIPTION}
- What the model currently does wrong: {FAILURE_PATTERN}
- What correct behavior looks like: {CORRECT_BEHAVIOR}
- Example trajectories showing the failure: {TRAJECTORY_EXAMPLES_OR_PATH}

Architecture Requirements

The environment MUST implement the `GameEnv` protocol from `game_registry.py`:

```
class GameEnv(Protocol):
    done: bool                          # Episode terminated?
    current_player: int                 # Which agent acts next (0 = agent)
    rewards: Dict[int, float]           # Terminal rewards per player
```

```
    invalid_player: Optional[int]        # Invalid move tracking

    def reset(self, seed: int) -> None: ...
    def observe(self, player_id: int) -> str: ...
    def legal_actions(self) -> List[str]: ...
    def step(self, action: Optional[str]) -> None: ...
```

And register itself via `GameSpec` in `game_registry.py`:

```
register_game(GameSpec(
    name="{GAME_NAME}",
    make_env=lambda **kwargs: YourGameEnv(**kwargs),
    extract_action=extract_action_fn,
    system_prompt="...",
    max_gen_tokens=2048,
))
```

Critical Design Constraints

1. Reward Design for GRPO

GRPO learns from WITHIN-GROUP reward variance. If all rollouts in a group get the same reward, the gradient is zero and the iteration is wasted. Your reward function MUST create variance:

- **DO**: Use continuous/multi-level rewards (0.0, 0.2, 0.5, 0.8, 1.0) so that partial success is distinguishable from total failure
- **DO**: Design scenarios where the base model succeeds ∼30-60% of the time. Too easy (>80%) = no negative signal. Too hard (<15%) = no positive signal.
- **DO**: Use multiplicative reward components for behaviors that must co-occur (e.g., `tool_correct * communicated_result`), with an additive floor to preserve gradient signal
- **DON'T**: Use pure binary rewards (0 or 1) unless the base model success rate is ∼40-60%

Recommended reward structure:

```
# Multiplicative component: all must succeed for full reward
mult = component_a * component_b * component_c

# Additive floor: preserves gradient even when one component is zero
add = w_a * component_a + w_b * component_b + w_c * component_c

# Combined: multiplicative dominates but additive prevents zero-gradient
reward = alpha * mult + (1 - alpha) * add
```

2. Tool and Format Fidelity

If the environment targets a specific benchmark (e.g., tau2-bench, ToolSandbox):

- Use the exact same tool schemas (function names, parameter names, return types)
- Error messages and response formats should be consistent within the environment
- The system prompt should set up the skill context clearly, but not replicate the exact benchmark's prompt
- Data (database contents, user profiles, etc.) should be procedurally generated

  The goal is to train the SKILL (e.g., multi-step reasoning, clarification, tool chaining), not to overfit to a particular benchmark's surface format.

This eliminates distribution shift between training and evaluation.

4. Procedural Generation

All scenarios MUST be procedurally generated from a seed:

- `reset(seed)` must produce a deterministic scenario for that seed
- Different seeds must produce meaningfully different scenarios (not just parameter swaps)
- The scenario space should be large enough that memorization is impossible
- Include randomized: database contents, user requests, constraint combinations, number of distractors

5. Skill Isolation
Each environment should primarily stress ONE skill. If you identify multiple skills:

- Create separate scenario generators per skill
- Assign probability weights (e.g., 40% skill_a, 35% skill_b, 25% skill_c)
- Reward functions can differ per skill, but the interface is the same

6. Multi-Skill Environments
When combining multiple skills in one environment (like toolsandbox_multiturn):

- Sample skill per scenario at reset time based on probability weights
- Each skill gets its own scenario generators and reward logic
- Track which skill each scenario targets (for analysis)
- Aim for 1-2 skills max per environment — beyond that, create separate envs

Deliverables

1. **{game_name}_game.py**: The environment file implementing `GameEnv`
2. **Registration in game_registry.py**:
   `GameSpec` entry with proper `extract_action`, `system_prompt`, etc.
3. **Reward verification**: Run 100+ rollouts using the vLLM server and report:
   - Mean reward (target: 0.3-0.6 for base model)
   - Reward standard deviation (target: >0.2)
   - Per-skill breakdown if multi-skill
   - % of groups with reward variance (target: >60%)

Verification Steps
Before finalizing, verify using the vLLM server at {VLLM_URL}:

```
# 1. Run 100 rollouts and check reward distribution
rewards = []
for seed in range(100):
    env = YourGameEnv()
    env.reset(seed)
    # ... play episode with vLLM ...
    rewards.append(env.rewards[0])

print(f"Mean: {np.mean(rewards):.3f}")
print(f"Std:  {np.std(rewards):.3f}")
print(f"Distribution: {Counter([round(r, 1) for r in rewards])}")

# 2. Check GRPO compatibility (group variance)
```

```
informative_groups = 0
for group_seed in range(20):
    group_rewards = []
    for _ in range(8):  # group_size
        env = YourGameEnv()
        env.reset(group_seed)
        # ... play with temperature=1.0 ...
        group_rewards.append(env.rewards[0])
    if len(set(round(r, 2) for r in group_rewards)) > 1:
        informative_groups += 1

print(f"Informative groups: {informative_groups}/20 ({informative_groups/20*100:.0f
    }%)")
```

Reference Files

Read these files to understand the architecture:

- `game_registry.py` — `GameEnv` protocol, `GameSpec`, registration
- `train_grpo_optimized.py` — How the training loop interfaces with games

G. Metrics Detail

The overall pass rate for τ^2-Bench is defined as:

$$\text{PassRate}_{\text{overall}} = \frac{S_{\text{Airline}} + S_{\text{Retail}}}{N_{\text{Airline}} + N_{\text{Retail}}},$$

where S_{Airline} and S_{Retail} denote the numbers of solved tasks in the two domains, and N_{Airline} and N_{Retail} denote the corresponding numbers of benchmark tasks.

The Pass@1 for SWE-bench Verified is defined as:

$$\text{Pass@1} = \frac{1}{N_{\text{SWE}}} \sum_{i=1}^{N_{\text{SWE}}} \mathbf{1}[\text{patch}_i \text{ resolves issue } i]\,,$$

where the indicator is 1 if and only if the single patch generated for instance i applies cleanly and passes all of that instance's held-out tests, and $N_{\text{SWE}} = 500$ is the number of SWE-bench Verified instances.

The mean similarity and the perfect score for ToolSandBox is defined as:

$$\text{MeanSimilarity} = \frac{1}{N_{\text{TS}}} \sum_{i=1}^{N_{\text{TS}}} \text{Sim}_i, \tag{3}$$

$$\text{PerfectRate} = \frac{1}{N_{\text{TS}}} \sum_{i=1}^{N_{\text{TS}}} \mathbf{1}[\text{Sim}_i = 1]\,. \tag{4}$$

where Sim_i denotes the final ToolSandbox similarity score for trajectory τ_i, and N_{TS} is the number of evaluated ToolSandbox scenarios.

H. Results with Stochastic Decoding

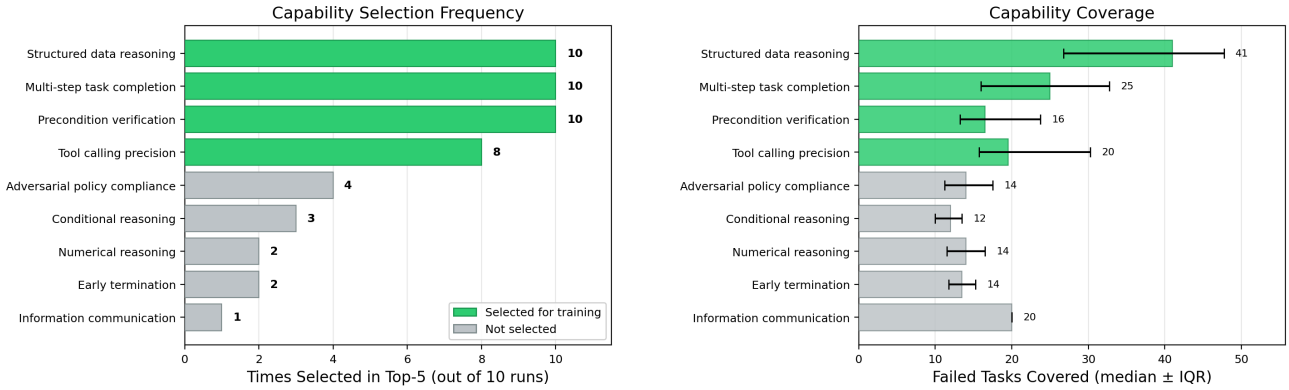
The main paper uses greedy decoding (temperature $T = 0$, seed 42) for deterministic reproducibility, which yields a single sample per task. We re-ran all baselines and TRACE at temperature 0.7 with 3 seeds and report the mean $\pm$ 95% confidence interval (CI) in Tables 6 and 7. As shown in the tables, TRACE consistently outperforms all baselines on both benchmarks, with non-overlapping CIs.

Table 6. Pass rate on τ^2-Bench under stochastic decoding (temperature 0.7, 3 seeds). We report mean $\pm$ 95% CI.

Approach	Airline (%)	Retail (%)	Overall (%)
Base	21.3 ± 2.9	36.3 ± 2.5	32.1 ± 1.8
GRPO	30.7 ± 5.7	38.9 ± 3.3	36.4 ± 3.2
GEPA	36.7 ± 2.9	38.6 ± 2.2	38.0 ± 1.7
TRACE	$\mathbf{44.0 \pm 5.0}$	$\mathbf{47.1 \pm 2.5}$	$\mathbf{46.1 \pm 3.2}$

Table 7. Pass rate and mean similarity on ToolSandBox under stochastic decoding (temperature 0.7, 3 seeds). We report mean $\pm$ 95% CI.

Model	Pass Rate (%)	Mean Sim.
Base	14.2 ± 2.2	0.40 ± 0.03
GRPO	16.3 ± 1.9	0.51 ± 0.02
GEPA	16.8 ± 1.1	0.52 ± 0.01
TRACE	$\mathbf{21.2 \pm 2.2}$	$\mathbf{0.56 \pm 0.02}$



(a) Capability selection frequency across 10 independent runs. The analysis agent consistently recovers the same top four deficits (green) for targeted training.

(b) Median task failure coverage across runs (error bars denote IQR). Coverage is not mutually exclusive, as a single failed task may lack multiple capabilities.

Figure 5. Stability and coverage of identified capabilities. Repeated contrastive analysis consistently identifies a small, stable set of capability deficits (a) that account for a heavily concentrated proportion of benchmark failures (b). This skewed distribution demonstrates that a few high-impact capabilities drive most errors, validating our use of targeted synthetic environments.

I. Cross-Environment Transfer

We furthermore evaluate cross-environment transfer in Table 8. A model trained with TRACE or GRPO on τ^2-Bench is evaluated on a second tool-calling benchmark, ToolSandBox, without re-training. The TRACE-trained agent still transfers more effectively than the GRPO-trained one ($+1$ perfect score and $+0.052$ mean similarity), indicating that TRACE's targeted training yields capabilities that carry over to a new environment rather than merely overfitting to the training benchmark.

J. Extended Capability Analysis

We analyze the capabilities identified by repeated contrastive analysis across 10 independent runs (Figure 5a). The pipeline consistently converges on a small set of lacking capabilities: structured data reasoning, multi-step task completion, and precondition verification are recovered in all 10 runs, while tool-calling precision appears in 8 of 10.

Beyond selection frequency, failure coverage is heavily concentrated within these specific capabilities (Figure 5b). Structured data reasoning accounts for the largest share of failed benchmark tasks, followed closely by multi-step task completion. This skewed distribution empirically validates our approach of explicitly isolating and targeting the lacking capabilities responsible for a significant proportion of failures, which maximizes data efficiency.

Table 8. Cross-environment transfer evaluated on ToolSandBox. Agents trained with GRPO or TRACE on τ^2-Bench are evaluated on ToolSandBox without re-training; the bottom block reports in-domain ToolSandBox training for reference. We report perfect score (similarity = 1.0) out of 129 scenarios and mean similarity.

Method	Trained on	Perfect	Mean Sim.
Base	—	19/129	0.411
GRPO	τ^2-Bench	19/129	0.430
TRACE	τ^2-Bench	**20/129**	**0.482**
GRPO	ToolSandBox	22/129	0.519
TRACE	ToolSandBox	**26/129**	**0.552**

While the analysis agent consistently selects these top capabilities, it frequently discards competing categories. Candidates such as conditional reasoning, numerical reasoning, early termination, and information communication appeared much less frequently across the 10 independent runs. This long-tail distribution confirms that the target benchmark's failure modes are concentrated rather than uniform.

Additionally, we note that the coverage counts reported in Figure 5(b) are not mutually exclusive. A single failed benchmark trajectory often involves multiple missing capabilities—for example, an agent might fail to verify a precondition before executing a multi-step task. The contrastive analysis successfully disentangles these overlapping failures, allowing the framework to synthesize isolated micro-environments for each distinct deficit.

K. Capability Details

K.1. Training Capabilities and Trajectory Examples

We identify four primary capability gaps through contrastive analysis of failed baseline trajectories on τ^2-Bench. Each capability targets a distinct failure mode.

C1: Structured Data Reasoning. The agent fails to parse or cross-reference structured records returned by tools. *Example:* A user requests an economy flight departing after 11 AM. The search tool returns flights with nested price arrays per cabin class. The base model misreads which price corresponds to economy, computes the wrong total, and the booking fails with repeated payment errors.

C2: Tool Calling Precision. The agent identifies the correct tool but passes wrong arguments. *Example:* A user requests a refund to their original payment method (credit card). The agent retrieves the order (showing `credit_card_3892` as payment) and the user profile (containing both `credit_card_3892` and `gift_card_1234`). When calling `return_delivered_order_items`, it passes `gift_card_1234` instead of the correct `credit_card_3892`.

C3: Multi-Step Task Completion. The agent completes the first sub-task of a compound request then stops. *Example:* A user asks to cancel two reservations and modify a third. The agent successfully cancels the first reservation, generates a closing statement ("If you need any further assistance. . . "), and never attempts the remaining two operations—entering a sycophancy loop with the user simulator until timeout.

C4: Precondition Verification. The agent executes state-changing actions without checking policy eligibility. *Example:* A user requests cancellation of a basic economy flight booked 14 days ago with no insurance. The policy requires at least one of: booked within 24 hours, airline-cancelled flight, business class, or covered insurance. None apply, but the agent calls `cancel_reservation` without checking any condition. The API does not enforce policy—the agent must independently verify eligibility.

K.2. ToolSandbox Training Capabilities and Trajectory Examples

We identify two primary capability gaps through analysis of base model trajectories on ToolSandbox:

C1: Permission Error Recovery. When a tool call returns a `PermissionError` due to a device setting conflict, the base model stops and reports the error to the user instead of resolving the underlying issue. *Example:* The user asks "Turn on wifi." The agent calls `set_wifi_status(on=True)`, which returns `PermissionError: Wifi cannot`

be turned on in low battery mode. The base model halts—the conversation ends with only 2 messages. The correct behavior is to diagnose the blocker by calling get_low_battery_mode_status (returns True), disable it with set_low_battery_mode_status(on=False), retry the original set_wifi_status(on=True), and communicate success to the user.

C2: Datetime Reasoning. The base model skips the timestamp_to_datetime_info tool and instead attempts to mentally decode Unix timestamps, consistently hallucinating the wrong date. *Example 1:* The user asks "Remind me to buy chocolate milk tomorrow 5PM." The agent calls get_current_timestamp (returns 1774511873), then directly calls datetime_info_to_timestamp(year=2026, month=3, day=25, ...)—guessing the current date from the raw timestamp and getting the day wrong (March 25 instead of March 26), setting the reminder in the past. The correct behavior is to first call timestamp_to_datetime_info(1774511873) which returns the exact date {year: 2026, month: 3, day: 26, ...}, then compute tomorrow as March 27. *Example 2:* The user asks "How many days till Christmas Day?" The base model calls search_holiday("Christmas Day", year=2023)—defaulting to a training-data year—and computes -823 days (past). The correct behavior is to first determine the current year via get_current_timestamp followed by timestamp_to_datetime_info, then search for the holiday in the correct year (2026), yielding 273 days.

L. Synthetic Environment Example

L.1. Structured Data Reasoning (τ^2-Bench)

Algorithm: Structured Data Reasoning Game (tau2-bench Aligned)

```
# ====================================================================
# 1. SCENARIO & DATABASE GENERATION
# ====================================================================

FUNCTION GenerateSDRScenario(seed, domain_choice):
    Initialize RNG(seed)
    domain = domain_choice OR RandomChoice(["Airline", "Retail"])

    # Select task type based on empirical failure weights
    task_type = WeightedChoice(SCENARIO_WEIGHTS[domain])

    # SYNTHETIC DB SYNTHESIS
    IF domain == "Airline":
        db = CreateFlightsDB()          # JSON: numbers, prices, seats
        user = CreateUser(reservations) # JSON: status, payments
    ELSE IF domain == "Retail":
        db = CreateProductsDB()        # JSON: 10+ variants, stock
        user = CreateUser(orders)      # JSON: history, tracking

    # CONSTRUCT GOAL & CONSTRAINTS
    # Gap 3: Conditional Fallback | Gap 6: Cross-Entity Match
    # Gap 7: Progressive Disclosure

    RETURN SDRScenario(initial_msg, user_sys_prompt, db, expected_tool, expected_answer)


# ====================================================================
# 2. MULTI-TURN ENVIRONMENT LOOP
# ====================================================================

CLASS StructuredDataGame:
    METHOD Reset(seed):
        scenario = GenerateSDRScenario(seed)
        env_db = Copy(scenario.db)
        user_sim = InitializeLLMUser(scenario.user_sys_prompt)
        RETURN scenario.initial_message

    METHOD Step(agent_action):
        IF agent_action IS "tool_call":
            result = ExecuteTool(agent_action.name, agent_action.args, env_db)
            RETURN tool_output(result)

        ELSE IF agent_action IS "respond_to_user":
            user_response = user_sim.GenerateResponse(agent_action.text)

            IF "###STOP###" IN user_response:
                TerminateGame(EvaluateReward())
            ELSE:
                RETURN user_response


# ====================================================================
# 3. REWARD & VALIDATION LOGIC
# ====================================================================

FUNCTION EvaluateReward(conversation, scenario, final_db):
    # STEP 1: VERIFY COMMUNICATION (RELIABLE REPORTING)
    comm_pass = All(val IN agent_messages FOR val IN scenario.communicate_info)

    # STEP 2: VERIFY STATE MUTATION (DATABASE INTEGRITY)
    IF scenario.expects_mutation:
        # Prevents rewards for "hallucinated" success messages
        db_match = (Hash(final_db) == Hash(ReplayGoldAction(scenario.db)))

        IF db_match AND comm_pass: RETURN 1.0  # Success
        IF db_match:               RETURN 0.3  # Correct action, bad info
        RETURN 0.0

    # STEP 3: INFORMATION RETRIEVAL TASKS
    RETURN 1.0 IF comm_pass ELSE 0.0
```

L.2. Error Recovery (ToolSandBox)

Algorithm: TEC Game v2 (Tool-use & Error Recovery)

```
# ======================================================================
# 1. SCENARIO GENERATION & SKILL TARGETS
# ======================================================================

FUNCTION GenerateTECScenario(seed):
    Initialize RNG(seed)
    roll = RandomValue(0, 1)

    # Skill A: Tool-then-Communicate (40%)
    IF roll < 0.40:
        RETURN Scenario(skill="communicate", verify_keywords=[...])

    # Skill B: Error Recovery Chain (40%)
    ELSE IF roll < 0.80:
        # PermissionError triggered by system settings (e.g. Low Battery)
        RETURN Scenario(skill="recovery", blocker="low_battery")

    # Combined: Recovery + Communication (20%)
    ELSE:
        RETURN Scenario(skill="combined")

# ======================================================================
# 2. MULTI-TURN ENVIRONMENT LOOP
# ======================================================================

CLASS TECGame:
    METHOD Step(agent_action):
        IF agent_action IS "tool_call":
            # Execute tool against synthetic ToolSandbox DB
            result = ExecuteTool(agent_action.name, agent_action.args)

            # If low_battery is True, certain tools return PermissionError
            # Agent must diagnose and call 'set_low_battery_mode_status'
            RETURN tool_result_to_agent(result)

        ELSE:
            # Final text response to user
            reward = EvaluateTEC(agent_action.text, self.db, self.scenario)
            TerminateGame(reward)

# ======================================================================
# 3. REWARD EVALUATION
# ======================================================================

FUNCTION EvaluateTEC(agent_response, final_db, scenario):
    # ACTION SCORE: Was the blocked service successfully enabled?
    IF scenario.skill IN ("recovery", "combined"):
        action_score = 1.0 IF TargetServiceIsActive(final_db) ELSE 0.0
    ELSE:
        action_score = 1.0 IF ToolWasCalled(scenario.target) ELSE 0.0

    # COMM SCORE: Did the agent provide the result to the user?
    # Matches keywords from tool output (e.g. phone numbers, coordinates)
    comm_score = KeywordMatchRatio(agent_response, scenario.keywords)

    # Weighted Reward: Combines execution and reporting
    RETURN (0.6 * action_score) + (0.4 * comm_score)
```