Successful Rollouts are Not Equal: Discovering Influential Behaviors for Training Coding Agents

Yiqing Xie[1], **Gaokai Zhang**[1], **Xingyao Wang**[2], **Dan Roth**[3,4], **Daniel Fried**[1], **Carolyn Rosé**[1]
[1]Carnegie Mellon University [2]University of Illinois Urbana-Champaign
[3]University of Pennsylvania [4]Oracle AI
{yiqingxi, gaokaiz, dfried, cp3a}@andrew.cmu.edu
https://catradar-coding-agent.github.io/

arXiv:submit/8219974 [cs.SE] 13 Oct 2026

Abstract

Most coding agent training methods judge the quality of demonstrations based on their final outcome, such as execution results or rubric scores. However, we observe that the effectiveness of training also substantially depends on *how* the demonstrations achieve the results. Previous work has manually made hypotheses about which demonstration behaviors influence the performance. To find such influential behaviors for **C**oding **A**gent **T**raining more efficiently, we present **CATRADAR**, an analysis method to automatically discover candidate behaviors by defining, extracting, and comparing the behaviors of different trained agents. **CATRADAR** enables us to find behaviors that are influential to at least one model, including specific repo-exploration strategies, the planning and verification patterns, and the verbosity of the reasoning text. Our insights also translate to performance gains by various methods, including data selection, agent step synthesis, and behavior-conditioned rollout. Results show that our agent step synthesis method improves Qwen3-8B's resolved rate from 11.6% to 17.6% on SWE-Bench Verified.

1 Introduction

Agents have been trained to be effective in diverse real-world coding applications (Mündler et al., 2024; Zhao et al., 2025). Existing research improves coding agent post-training by constructing scalable environments (Pan et al., 2025; Yang et al., 2025b), designing training methods (Wei et al., 2025; Song et al., 2026), and synthesizing training signals (Wang et al., 2026; Sun et al., 2026). Most of them still derive training signals based on the final outcome of the demonstrations, which is either obtained by execution in sandbox environments or simulated with manually-designed rubrics (Wang et al., 2026) or LLM judges (Sun et al., 2026).

However, we observe that even when we only use successful rollouts as demonstrations in training, the exact **behaviors** reflected in the demonstrations also substantially affect the performance of the trained model. We conduct a preliminary study under the setting of distilling a student model from the teacher's successful rollouts. As shown in Figure 2, on the same set of training instances and for a fixed student model, distilling from different teachers leads to substantially different performance. Although they all successfully solve the task, the teacher models may apply different problem-solving strategies, use different tools, and generate concrete actions with different patterns.

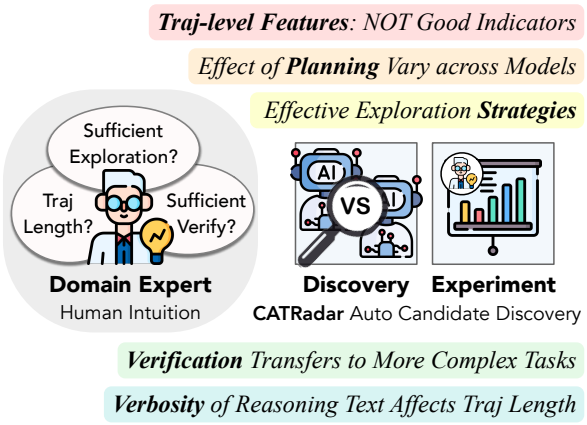


Figure 1: To find demonstration behaviors that affect the effectiveness of training, (*left*) prior works rely on human experts to manually make hypotheses, while (*right*) **CATRADAR** narrows down the search space by automatically discovering candidate behaviors by extracting and comparing the behaviors of different trained agents.

We thus would like to answer the question: *what are the demonstration behaviors that affect coding agent training performance?* Existing research presents manually defined rubrics on agent behaviors and empirically examines their impact on training (Yang et al., 2026; Wang et al., 2026; Xie et al., 2026). However, the space of coding agent behaviors is vast and open-ended, ranging from high-level problem-solving strategies to low-level tool-use patterns. The influence of a behavior could also

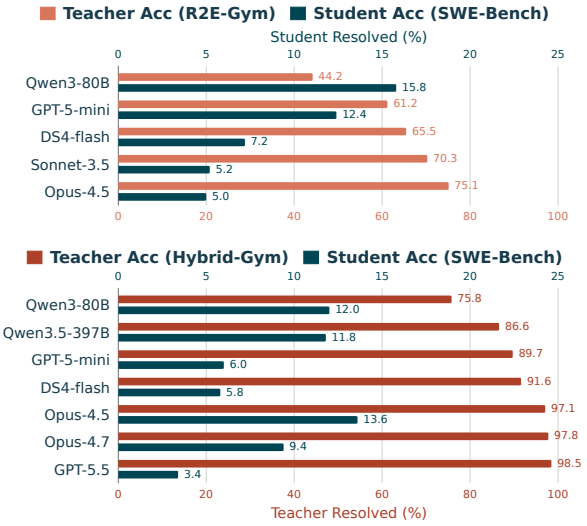


Figure 2: Across different training sets, on the same sets of task instances, successful demonstrations generated by different teacher models lead to substantially different performance, and higher teacher performance does not necessarily transfer to better student performance.

vary across training configurations and base models. Manually enumerating behavioral hypotheses lacks scalability and human expertise may be limited to specific experimental settings.

In this paper, we present **CATRADAR**, an automatic analysis method to efficiently discover candidate behaviors in training demonstrations that could potentially affect **C**oding **A**gent **T**raining performance. We leverage an LLM agent to automatically define, extract, and compare the behaviors of coding agents trained on different data sources that have different performances. We only keep candidate behaviors that are well-defined and can distinguish the two agents we compare. After **CATRADAR** narrows down the search space of behaviors, we examine the impact of those candidate behaviors through controlled interventions on the presence of each behavior in the training data.

We apply **CATRADAR** to 8 teacher models and 3 student models for two training tasks: R2EGym (Jain et al., 2025) and Hybrid-Gym (Xie et al., 2026). Guided by **CATRADAR**, we examine the impact of 10 behaviors of coding agent demonstrations and 7 of them are shown to positively affect at least one model's performance. As shown in Figure 1, such influential behaviors include specific repo-exploration strategies, planning and verification patterns, and the verbosity of reasoning text.

Furthermore, we demonstrate that finding such influential behaviors can lead to actual performance gains by various methods, including data selection, agent step synthesis, and behavior-conditioned rollout. For instance, we demonstrate that data selection based on the repo-exploration strategies we discover improves all three student models on SWE-Bench Verified (Jimenez et al., 2024) (e.g., improving the resolved rate of Qwen3-8B from 4.8% to 11.6%). We also discover that simply adding 4 LLM-synthesized planning steps to each demonstration improves the resolved rates of all three student models (e.g., improving Qwen3-8B's resolved rate from 11.6% to 17.6%).

2 Analysis Method: Discovering Influential Behaviors of Demonstrations

Figure 2 shows that coding agents trained on demonstrations produced by different models can achieve different performances, but the success rates of the demonstrations cannot explain the performance gap. We hence hypothesize that there exist a set of properties of **how** the final results are achieved that account for the performance gap, which we denote as **influential behaviors**. Identifying such behaviors is challenging because the space of coding agent behaviors is vast, spanning from high-level problem-solving strategies to low-level implementation details. In this section, we first present **CATRADAR** (§2.1) to automatically narrow down the space of candidate behaviors by discovering the different behaviors of coding agents with different performances. We then conduct controlled data intervention experiments (§2.2) to empirically examine the impact of the candidate behaviors we discovered. We released the code of the **CATRADAR** framework and all the data intervention experiments we designed.

2.1 CATRADAR: Automatic Candidate Behavior Discovery

Figure 3 shows the overview of **CATRADAR**.

Rollout and Discover Distinguishing Behaviors. We run two coding agents on the same set of task instances and provide their trajectories, paired by instance, to a discovery agent (e.g., Claude-Opus-4.7). We prompt the discovery agent to discover different *behaviors* between the two agents, where each behavior is defined by a judge metric that specifies a rule for computing either a numerical or a categorical value for each trajectory. Table 1 presents examples of behaviors and their associated judge metrics. We only retain the behaviors with a **well-defined** judge metric that can either be com-

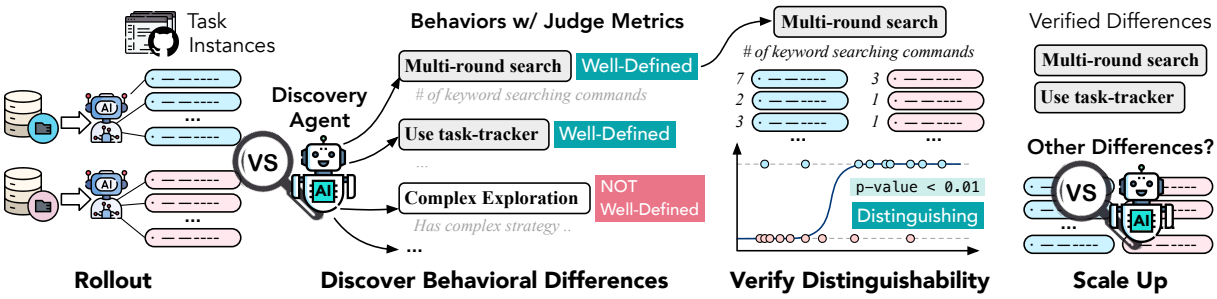


Figure 3: Overview of the **CATRADAR** automatic behavior discovery framework (§2.1). We run two trained coding agents on the same task instances, use a discovery agent to propose behavioral differences with a well-defined judge metric, and verify the distinguishability of each behavior by fitting a logistic regression.

puted by a deterministic script or scored by an LLM judge with high consistency. Both are generated by the discovery agent and manually examined. We check consistency by applying the LLM judge to the same set of examples for multiple times.

Verify Distinguishability. Given a candidate behavior and its associated judge metric, we compute the metric score for each trajectory based on its definition. Then we treat each trajectory as one observation and fit a separate logistic regression, with agent identity (i.e., the agent that produces the trajectory) as the dependent variable and the judge metric score as the predictor, where we encode categorical values with indicator variables. We label $N = 50$ trajectories for each agent and consider the behavior to be **distinguishing** if the Wald test rejects the null hypothesis with p-value< 0.01.

Scale Up. We add all well-defined behaviors that pass the distinguishability test to the candidate behavior list and repeat the discovery-and-verification procedure for at most $k = 5$ iterations per task set and agent pair. In each subsequent iteration, we provide the discovery agent with all existing behaviors and prompt it to identify additional differences. We repeat this procedure across multiple training tasks and agent pairs. We manually review and prioritize the retained behaviors for the data intervention experiments (§2.2) to examine their impact on the training performance.

2.2 Data Intervention Methods

This section aims to isolate and measure the impact of each candidate behavior discovered in §2.1 on the training performance. We introduce two complementary intervention methods that control the presence of a target behavior while minimizing changes to other properties of the demonstrations. For both methods, we consider a behavior to be **influential** if the presence or absence of the behavior leads to a substantial performance change.

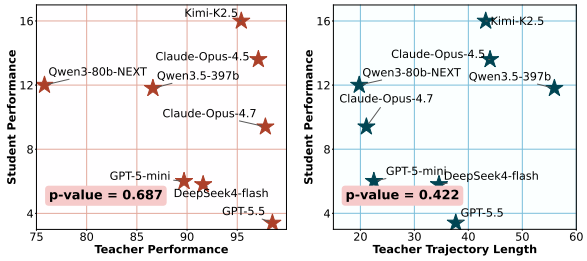
Trajectory Post-Editing. This method adds, modifies, or removes one or more trajectory steps to control the presence of a target behavior, which can be performed by deterministic rules or an LLM. One example is to remove the "verify after edit" behavior (as defined in Table 1) from a trajectory by identifying the final file-editing action by pattern matching and removing all subsequent steps.

One major challenge is to only conduct edits that preserve *trajectory consistency*: every retained step must remain supported by its preceding context. Here we provide a non-exhaustive list of classes of edits that preserve consistency: (1) Interchanging two sequences of steps that produce the same intermediate results that are crucial for resolving the task (e.g., two search sequences that both locate the target file); (2) Removing steps whose outputs and state effects are not required by subsequent actions, such as unsuccessful searches whose results do not appear in subsequent actions; and (3) Modifying steps that neither change the environment state nor provide information to subsequent actions, including steps that contained only chain-of-thought reasoning text and actions that are not executed due to wrong syntax.

Behavior-Conditioned Trajectory Generation. For behaviors that are difficult to manipulate through local edits, we generate new demonstrations for the same task instances by prompting the agent to exhibit the target behavior. To limit differences unrelated to the target behavior, we retain an original-generated trajectory pair only if both trajectories successfully solve the task and reach the same set of essential intermediate results, which are pre-defined based on the task. For example, in the issue-solving task, we require the original and newly generated trajectories to identify the exact same set of target files and pass all the unit tests.

Behavior	Judge Metric	LEGO	Hybrid	Others	Ours
Multi-round search	Run keyword searching commands for ≥ 2 times				✓
Has fine-grained keywords	Search for fine-grained keywords (e.g., specific filenames) instead of only broad keywords (e.g., `def` or `class`)				✓
Limit target files before read	Limit the candidates to ≤ 30 files before reading any file				✓
Use task-tracker	Use the `<function=task_tracker>` tool for explicit reasoning				✓
Verify after edit	Has verification steps after the final file-editing action	✓			✓
Use think	Use the `<function=think>` tool for explicit reasoning				✓
Reasoning text verbosity	The length of the text part outside tool calls		✓		✓
Stuck in loop	Try different commands after failing for 3 times	U			U
Exploration length	The number of actions before the first file-editing action (≥ 16: high; 6–15: medium; ≤ 5: low)				U
Trajectory length	Has ≤ 7 agent actions: short; ≥ 8 agent actions: long	U	✓	✓	U

Table 1: Some of the behaviors discovered and examined by our framework. We present the full list of candidate behaviors in §A.4. We use "✓" to indicate that the behavior is shown to be relevant to the final performance for some student models in related works and use "U" to indicate that the behavior is tested but not shown to be relevant. "LEGO" (Yang et al., 2026), "Hybrid" (Xie et al., 2026), and "Others" refer to existing works.



(a) Teacher performance (b) Trajectory length

Figure 4: Student performance (Qwen2.5Coder-7B) is not significantly correlated with trajectory-level features such as teacher success rate or trajectory length.

3 Analysis Results: What Behaviors Affect Coding Agent Training?

This section presents the analysis results of the **CATRADAR** method. As a proof of concept, we focus on the distillation setting. In principle, **CATRADAR** can also be applied to other settings such as self-training or reinforcement learning.

Experimental Setup. We conduct analysis on both in-domain training: the R2E-Gym issue-solving dataset (Jain et al., 2025), and transfer learning: the Hybrid-Gym function localization dataset (Xie et al., 2026), with SWE-Bench Verified (Jimenez et al., 2024) as the evaluation dataset. We adopt the distillation setting and apply **CATRADAR** to 8 teacher models and 3 student models, as described in §A.2. We use the OpenHands agent harness (Wang et al., 2025) for all the experiments. Since the impact of one behavior may depend on other factors such as other teacher behaviors or the student's capabilities, when we conduct data intervention experiments and compare the performance with the original data (§2.2), we experiment on multiple training tasks, teacher models, and student models (if applicable).

3.1 CATRADAR Discovers Diverse Candidates

Table 1 summarizes the candidate behaviors discovered by **CATRADAR** and examined in our experiments, along with their definitions. Compared with the behaviors that are manually hypothesized and examined in prior work, **CATRADAR** discovers a broader range of behaviors of diverse granularity, spanning trajectory-level features, problem-solving strategies, and tool-use patterns. Our experiments show that 7 of the 10 behaviors affect student performance in at least one combination of student model, teacher model, and training dataset. Some of these findings are also consistent with prior work. For instance, Terminal-LEGO (Yang et al., 2026) shows that verification steps in training improve performance. **CATRADAR** also discovers "verify after edit" as a potentially important behavior.

3.2 Trajectory-Level Features are Not Good Indicators of Training Effectiveness

Longer trajectories are commonly viewed as more useful training examples because they provide the model more information (Zhu et al., 2025). Under the setting of distillation, it is also a common belief that strong models often produce stronger students (Pan et al., 2025).

Our results challenge both assumptions. As shown in Figure 4 and Figure 10, none of the trajectory-level features we examine is significantly correlated with student performance. Our hypothesis is that longer trajectories may reflect overly complicated problem-solving strategies, which may not

[Case 1] Multi-Search: ✓ Target-Read: ✗ Suff-Explore: ✓	
`grep -r "def " --include='*.py' .`	→ **1771** files
Search for keywords "model"	→ **105** files
Read `core/optimizers/function_marker.py`	→ ...
[Case 2] Multi-Search: ✗ Target-Read: ✓ Suff-Explore: ✓	
`grep -r "node" --include='*.py' .`	→ **6** files
Read `core/optimizers/function_marker.py`	→ ...
[Case 3] Multi-Search: ✓ Target-Read: ✓ Suff-Explore: ✓	
`grep -r "model" --include='*.py' .`	→ **105** files
`grep -r "node" --include='*.py' .`	→ **6** files
Read `core/optimizers/function_marker.py`	→ ...

Table 2: A case study showing that demonstrations with different repo-exploration strategies can all be labeled as "sufficient exploration" by the LLM judge.

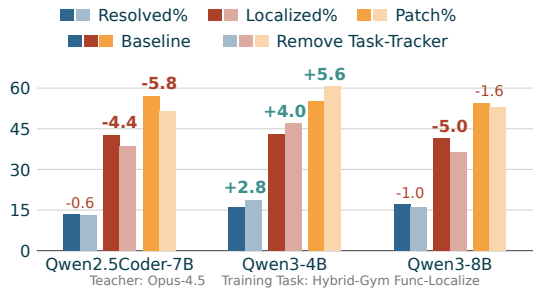


Figure 5: Planning steps in demonstrations have different effects on different models. We **highlight** significant performance changes after removing all task-tracker actions from demonstrations (with p-value < 0.05).

be suitable for the student model. Similarly, a stronger teacher model's strategies may rely on its sophisticated context understanding abilities, which are difficult to transfer to the student model, making the learned strategies less effective.

3.3 CATRADAR Discovers Effective Problem-Solving Strategies

Prior work shows that demonstrations should include a non-trivial repository-exploration stage and leverage an LLM judge to check whether it conducts sufficient exploration (Xie et al., 2026). However, it does not specify *how* the exploration is conducted. Instead, **CATRADAR** goes a step further by identifying two concrete exploration strategies: **Multi-Search**, which searches for multiple keywords iteratively, and **Target-Read**, which only reads a file after narrowing down the target to a few files. As shown in Table 2, the two strategies we discovered capture more fine-grained information than the high-level rubric used in prior works.

We think these two strategies are potentially useful for training for the following reasons: by having multiple search actions in the demonstrations, the agent is trained to try different keywords if the previous ones are too broad or fail to match any files. "Target-Read" teaches the model not to blindly read files with low relevance, which improves the efficiency of localizing the target file. We will show in §4.1 and §4.3 that demonstrations containing these specific strategies improve training performance.

3.4 Planning Steps have Different Effects on Different Models

CATRADAR discovers that the planning pattern also vary across different demonstrations. For instance, OpenHands has a "task-tracker" tool that creates and keeps a to-do list for the task, which is frequently used by some agents but not others. Intuitively, the task-tracker tool assists high-level planning and plays a similar role as chain-of-thought reasoning (Wei et al., 2023), which conducts explicit reasoning to guide the agent's decision-making process.

To examine whether the usage of task-tracker affects the performance, we directly remove all the task-tracker actions from the demonstrations and train the model. We additionally query an LLM to rewrite the text that explicitly refer to the removed steps to maintain the consistency of the trajectory.

However, results in Figure 5 show that having more planning steps in training does not always improve performance. On the demonstrations generated by Claude-Opus-4.5, the presence of task-tracker steps improves the performance of Qwen2.5Coder-7B and Qwen3-8B, while degrading the performance of Qwen3-4B. One possible explanation is that the 4B model has lower capacity and could benefit more from simpler and more straightforward demonstrations, while the larger models can better exploit complex reasoning patterns. We will further show in §4.2 that inserting LLM-synthesized task-tracker steps improves both the 7B and 8B models.

3.5 Verification Abilities Transfer from Simple Tasks to Complex Tasks

Verification is an important stage for coding agent tasks, where the agent examines its solution after it is generated. It can be complex for some tasks (e.g., issue-solving, where the agent needs to execute unit tests) but simple for others (e.g., function localization, where the agent only needs to read files and conduct reasoning). **CATRADAR** discovers that the average number of verification steps varies across different teacher models.

Teacher Model (Hybrid-Gym)	Teacher	Qwen2.5Coder-7B				Qwen3-4B				Qwen3-8B			
	# Verify	# Verify	Res%	Loc%	Patch%	# Verify	Res%	Loc%	Patch%	# Verify	Res%	Loc%	Patch%
Opus-4.5	**4.4**	**13.3**	**13.6**	**42.8**	**57.2**	**11.6**	16.0	43.0	55.0	**14.4**	**17.2**	**41.4**	**54.6**
Remove Verify	0.0	8.9	11.0 (-2.6)	39.6	53.0	7.0	**18.6** (+2.6)	**46.4**	**60.2**	7.7	15.8 (-1.4)	37.4	50.9
Opus-4.7	1.1	4.3	9.4	32.8	47.8	4.7	12.2	33.4	46.8	9.3	11.6	30.2	45.4
Add Verify	**4.2**	**11.2**	**10.8** (+1.4)	**34.0**	**54.2**	**9.9**	**13.4** (+1.2)	**38.4**	**50.8**	**15.2**	**15.0** (+3.4)	**36.2**	**50.8**

Table 3: Verification procedure in a simple training task (e.g., function localization) transfers to more complex tasks (e.g., issue-solving). However, the effect of verification steps depends on the student and teacher model. We highlight significant increases and decreases in resolved rates (with p-value < 0.05).

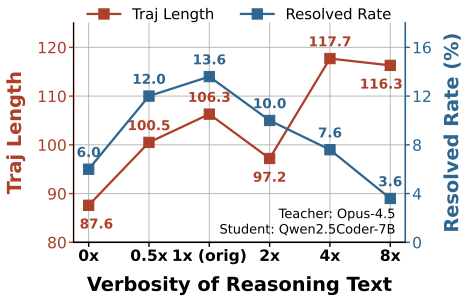


Figure 6: When the verbosity of reasoning text increases (including non-tool-call text and content of the "think" actions), the trajectory length in general increases, and the performance first increases and then decreases.

To examine the effect of verification steps in training, we design the following data intervention experiments to control the presence of verification steps in training demonstrations: (1) "Remove Verify", where we remove all the steps after the last file-editing action, and (2) "Add Verify": for demonstrations without any verification steps, we copy the verification steps from the demonstration of the same task instance generated by another agent (e.g., Claude-Opus-4.5). Other demonstrations are kept unchanged.

Surprisingly, as shown in Table 3, we observe that the verification behavior transfers across different tasks. When training the agent on function localization demonstrations with more verification steps, it also learns to perform more verification steps on the issue-solving task.

However, similar to the planning behavior, we observe that verification improves the performance of the 7B and 8B models, but does not consistently improve Qwen3-4B. This can also be explained by the hypothesis that models with lower capacity could benefit more from simpler demonstrations.

Our results make a further step beyond the finding in Terminal-LEGO (Yang et al., 2026), which shows that the inspect-act-verify pattern in training trajectories improves coding agents' performance on terminal tasks. We further show that the impact of verification also depends on other factors such as properties of the student model.

3.6 Training on Reasoning Text Increases Trajectory Length

CATRADAR discovers that the length of the text outside tool calls varies across different teacher models, which is on average 39 tokens for Claude-Opus-4.5, 98 for Qwen3-80B-NEXT, and 298 for GPT-5-mini on Hybrid-Gym.

To examine the effect of the verbosity of the reasoning text, we design an experiment that prompts an LLM to rewrite the reasoning text to reduce or increase the verbosity proportionally to its original length without adding new information. We define "reasoning text" as the combination of (1) the non-tool-call text and (2) the content of the "think" actions. We conduct post-checking with an LLM checker to ensure the rewritten text is natural and does not add new information. We also include a "0x" ablation that removes all reasoning text.

Surprisingly, Figure 6 shows that increasing the verbosity of the reasoning text increases the trajectory length. Our hypothesis is that more verbose reasoning encourages the model to articulate additional intermediate considerations, which in turn induces extra tool interactions. Similarly, we observe in Figure 11 that when we completely remove the non-tool-call text but keep the "think" actions, the trajectory length still substantially decreases.

4 Practical Utility of Our Discoveries

We apply the discoveries in §3 through three methods, ordered by increasing data-construction cost: selecting demonstrations with beneficial behaviors (§4.1), augmenting existing demonstrations with synthesized steps (§4.2), and generating new demonstrations with influential behaviors (§4.3).

4.1 Method 1: Behavior-Aware Data Selection

Data Selection. Given a fixed training-data budget, we select demonstrations that exhibit beneficial behaviors identified by **CATRADAR**. Following

Teacher Model	SWE-Bench Verified								
	Qwen2.5Coder-7B			Qwen3-4B			Qwen3-8B		
	Res%	Loc%	Patch%	Res%	Loc%	Patch%	Res%	Loc%	Patch%
***Training**: R2E-Gym Issue-Solving*									
GPT5-mini Subsets (250 examples)									
w/o Multi-Search	9.0	34.4	52.4	10.0	43.8	62.2	10.2	43.8	60.4
w/o Target-Read	8.6	36.2	52.0	13.2	48.8	62.4	10.4	42.4	56.6
Both Strategies (ours)	**11.4** (+2.8)	**49.4**	**63.6**	**14.2** (+4.2)	**58.4**	**75.4**	**13.8** (+3.6)	**48.2**	**63.2**
***Training**: Hybrid-Gym Func-Localize*									
Claude-Opus-4.5 Subsets (250 examples)									
w/o Multi-Search	2.8	13.6	31.6	11.8	**38.0**	**51.4**	4.8	15.2	29.2
w/o Target-Read	5.4	17.6	33.0	8.0	16.6	44.0	5.8	15.4	30.4
Both Strategies (ours)	**7.2** (+4.4)	**21.2**	**39.0**	**13.2** (+5.2)	35.4	50.8	**11.6** (+6.8)	**28.4**	**41.8**
Claude-Opus-4.7 Subsets (350 examples)									
w/o Multi-Search	2.8	15.8	42.8	7.2	**27.5**	**40.0**	8.6	26.8	**40.2**
w/o Target-Read	3.4	9.4	25.0	7.6	20.6	33.6	4.2	11.8	26.8
Both Strategies (ours)	**6.0** (+3.2)	**27.2**	**61.4**	**9.6** (+2.4)	25.8	36.8	**10.2** (+6.0)	**27.2**	39.4

Table 4: Data selection results with the repo-exploration strategies we discovered. Parenthesized results are absolute gains compared to the lower resolved rate of the two ablations. We highlight gains for resolved rates that are significantly better than both ablations and one of the ablations (with p-value < 0.05).

§3.3, we retain demonstrations that exhibit both the **Multi-Search** strategy (at least three keyword searches) and **Target-Read** (narrow down to $\leq$ 50 candidate files before reading). We compare against equally sized subsets lacking either strategy: "w/o Multi-Search" and "w/o Target-Read".

Results. Table 4 shows that selecting demonstrations with both strategies consistently yields higher resolved rates than either ablation across the evaluated settings. For example, training Qwen3-8B on 250 Claude-Opus-4.5 function-localization demonstrations with both strategies achieves 11.6% on SWE-Bench Verified, compared with 4.8% without Multi-Search and 5.8% without Target-Read. A possible explanation is that agents trained on the "w/o Multi-Search" ablation do not learn to recover from unsuccessful searches, and the "w/o Target-Read" ablation does not teach the strategy to progressively narrow down the candidate files. One common failure case for agents trained on "w/o Target-Read" is to repeatedly read irrelevant files until exhausting their action budget, without a more efficient strategy to locate relevant files.

4.2 Method 2: Adding Synthesized Agent Steps to Training Trajectories

Agent Step Synthesis. Another method is to augment existing demonstrations by inserting LLM-synthesized steps that instantiate a beneficial behavior (e.g., has high-level planning). Particularly, we prompt an LLM to generate a task-tracker call for each function localization demonstration that specifies a to-do list: identify the target file, locate the function, generate its docstring, and verify correctness. We insert subsequent task-tracker calls at completion points to update the corresponding items. Implementation details are provided in §A.6.

Results. As shown in Table 6, adding four synthesized task-tracker actions per demonstration improves the resolved rates of Qwen2.5Coder-7B and Qwen3-8B in all evaluated settings. For Qwen3-8B trained on Claude-Opus-4.7 demonstrations, the resolved rate increases from 11.6% to 17.6%. These steps may provide explicit planning structure that helps the student organize subsequent actions. Note that we train on the function localization task and improve issue-solving performances, suggesting that the planning behaviors transfer across tasks.

4.3 Method 3: Behavior-Conditioned Demonstration Generation

Prompt Generation. For behaviors that are difficult to incorporate through local edits, we generate new demonstrations using a behavior-conditioned teacher prompt (§2.2). We iteratively (1) ask an agent to write a prompt to elicit the target behaviors, (2) generate a small set of demonstrations, and (3) use an LLM judge to label their presence. We repeat this procedure until the generated prompt significantly increases the proportion of demonstrations exhibiting the target behaviors ($p < 0.05$), then use it to generate the training set.

Results. We apply this method to the two repo-exploration strategies we discovered in §3.3: Multi-Search and Target-Read. Table 5 shows that by generating demonstrations with the LLM-generated

Teacher Model	Teacher (Hybrid-Gym Func-Loc)			Student (SWE-Bench Verified)				
	Multi-Search%	Target-Read%	Res%	Multi-Search%	Target-Read%	Res%	Loc%	Patch%
GPT-5-mini	58.8	1.5	**89.7**	74.9	53.2	6.0	44.6	66.2
+ prompt student	58.8	1.5	**89.7**	75.6	57.6	7.2 (+1.2)	**48.2**	**67.0**
+ prompt teacher	**72.5**	**10.4**	85.9	**78.2**	**64.3**	**9.2 (+3.2)**	43.4	55.6

Table 5: Student performance (Qwen2.5Coder-7B) on SWE-Bench Verified under different teacher models and prompting strategies. We highlight significant changes in resolved rates.

Training Setting (Teacher → Student)	SWE-Bench Verified		
	Res%	Loc%	Patch%
***Training**: Hybrid-Gym Func-Localize*			
GPT-5-mini → Qwen2.5Coder-7B	6.0	**44.6**	**66.2**
Add task-tracker	**9.6 (+3.6)**	33.2	48.4
GPT-5-mini → Qwen3-8B	10.8	**56.6**	**71.6**
Add task-tracker	**13.0** (+2.2)	49.2	61.6
Opus-4.7 → Qwen2.5Coder-7B	9.4	32.8	47.8
Add task-tracker	**12.2 (+2.8)**	**38.6**	**55.2**
Opus-4.7 → Qwen3-8B	11.6	30.2	45.4
Add task-tracker	**17.6 (+6.0)**	**40.6**	**53.0**

Table 6: We can improve performance by adding synthetic task-planning steps in training trajectories. We highlight significant improvements in resolved rates.

prompt, we increase the prevalence of the strategies for both the teacher and the student models. Furthermore, training Qwen2.5Coder-7B on the resulting demonstrations significantly improves its resolved rate on SWE-Bench Verified from 6.0% to 9.2%. We also observe that directly prompting the student to follow the same strategies ("prompt student") yields a smaller gain, from 6.0% to 7.2%, suggesting that the student model has limited instruction-following capabilities to incorporate the strategies as prompted, and prompting the teacher model is more effective in this setting.

5 Related Work

Coding Agent Training. For both supervised finetuning (SFT) and reinforcement learning (RL), the quality of the final patch is typically used as the training signal, which can be defined based on the test execution results (Yang et al., 2025b; He et al., 2026; Yuan et al., 2026), similarity to the reference solution (Shen et al., 2026; Wei et al., 2025), or scores produced by a judge model (Shum et al., 2025). Another line of work also incorporate pre-defined metrics for the entire trajectories, including the structure of actions (Tao et al., 2026), trajectory-level, segment-level or step-level qualities (Han et al., 2026; Zhang et al., 2026; Wang et al., 2026), recovery attempts on failures (Liufu et al., 2026), and data distribution (Xie et al., 2025; Zheng et al., 2026). In comparison to manually-defined rubrics, our **CATRADAR** automatically discovers the demonstration behaviors that could potentially affect training effectiveness.

Analyses on Coding Agent Training. Prior work identifies demonstration properties that affect training effectiveness, including retaining failed terminal-agent trajectories (Pi et al., 2026), the pattern of inspect, act, and verify (Yang et al., 2026), combining explanary text and actions into the same step (Xie et al., 2026), and combining thinking and non-thinking traces (Ahmad et al., 2026). Particularly, Wang et al. (2026) manually define 24 rubrics to assess trajectory quality. Other analyses find that some factors are less predictive of training value, including task difficulty (Yang et al., 2025b), and trajectory length (Yang et al., 2026). However, most prior analyses start by manually making hypotheses on influential factors, which is limited by human expertise and potentially confined to particular training settings. Instead, **CATRADAR** assists such analyses by narrowing down the search space of potentially influential factors.

6 Conclusions

We present **CATRADAR**, an efficient analysis method, to discover potentially influential behaviors for training coding agents. Guided by **CATRADAR**, we conduct data intervention experiments and find influential behaviors including specific repo-exploration strategies, the planning and verification patterns, and the verbosity of reasoning text. Our insights further lead to practical improvements through data selection, agent step synthesis, and behavior-conditioned rollout, with step synthesis improving Qwen3-8B's resolved rate from 11.6% to 17.6% on SWE-Bench Verified. Beyond the supervised finetuning setting studied here, we envision **CATRADAR** supporting behavior discovery in other training settings including self-training and reinforcement learning, providing insights for why coding agent training fails or succeeds and guiding the design of the training process.

7 Limitations and Potential Risks

We identify the limitations of this work as follows:

Training and Evaluation Tasks and Models. Due to computational resource constraints, our experiments are limited to the R2E-Gym issue-solving and Hybrid-Gym function localization datasets for training, the SWE-bench Verified dataset for evaluation, and Qwen2.5Coder-7B, Qwen3-4B, and Qwen3-8B as student models. These choices limit the extent to which our findings can be generalized to other tasks, benchmarks, and model families or scales. We leave the exploration of additional tasks and models to future work.

Analysis Details. For most of the behaviors examined, we conduct data intervention experiments under only two or three training configurations, each defined by a combination of training dataset, teacher model, and student model. Although these experiments provide evidence of the effects of specific behaviors within the configurations studied, they do not establish that these effects hold consistently across a broader range of training conditions. More extensive experiments are needed to characterize how these effects depend on the training configuration and interactions among behaviors.

Analysis Settings. Our analysis is restricted to the distillation setting, in which student models are trained through supervised fine-tuning on teacher demonstrations. In principle, **CATRADAR** could also be applied to identify influential behaviors in other training settings, such as self-training and reinforcement learning (RL). For example, the presence or absence of identified behaviors could inform adjustments to reward scores for RL rollouts, enabling controlled experiments on their effects on training outcomes. The effectiveness of such extensions remains untested, and we leave their investigation to future work.

Potential Risks. We have not identified risks specific to this work beyond those generally associated with the use of large language models.

Acknowledgments

We would like to thank Andy Liu, Saujas Vaduguru, Graham Neubig, and members of CR's and DF's groups for helpful discussions and feedback. This work was supported in part by NSF grant DSES-2222762.

References

Wasi Uddin Ahmad, Nikolai Ludwig, Somshubra Majumdar, and Boris Ginsburg. 2026. Open-swe-traces: Advancing dual-mode multilingual distillation for software engineering agents. *Preprint*, arXiv:2606.16038.

Anthropic. 2024. Claude-3.5.

Anthropic. 2025. Claude-4.5.

Anthropic. 2026. Claude-4.7.

Taylor Berg-Kirkpatrick, David Burkett, and Dan Klein. 2012. An empirical investigation of statistical significance in NLP. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 995–1005, Jeju Island, Korea. Association for Computational Linguistics.

Zehui Chen, Kuikun Liu, Qiuchen Wang, Wenwei Zhang, Jiangning Liu, Dahua Lin, Kai Chen, and Feng Zhao. 2024. Agent-FLAN: Designing data and methods of effective agent tuning for large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 9354–9366, Bangkok, Thailand. Association for Computational Linguistics.

DeepSeek-AI and Xu et al. 2026. Deepseek-v4: Towards highly efficient million-token context intelligence. *Preprint*, arXiv:2606.19348.

Kanishk Gandhi, Ayush Chakravarthy, Anikait Singh, Nathan Lile, and Noah D. Goodman. 2025. Cognitive behaviors that enable self-improving reasoners, or, four habits of highly effective stars. *Preprint*, arXiv:2503.01307.

Hao Han, Jin Xie, Xuehao Ma, Weiquan Zhu, Ziyao Zhang, ZhiLiang Long, Hongkai Chen, and Qingwen Ye. 2026. SWE-TRACE: Optimizing long-horizon SWE agents through rubric process reward models and heuristic test-time scaling. *Preprint*, arXiv:2604.14820.

Zhenyu He, Qingping Yang, Wei Shen, Xiaojian Zhong, Kechi Zhang, Chenxin An, Wenlei Shi, Tianle Cai, Di He, Jiaze Chen, and Jingjing Xu. 2026. SWE-Swiss: A multi-task fine-tuning and RL recipe for high-performance issue resolution. In *Findings of the Association for Computational Linguistics: ACL 2026*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Naman Jain, Jaskirat Singh, Manish Shetty, Tianjun Zhang, Liang Zheng, Koushik Sen, and Ion Stoica.

2025. R2e-gym: Procedural environment generation and hybrid verifiers for scaling open-weights SWE agents. In *Second Conference on Language Modeling*.

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

Jiahe Jin, Abhijay Sai Paladugu, and Chenyan Xiong. 2026. Beneficial reasoning behaviors in agentic search and effective training methods to obtain them. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 28080–28097, San Diego, California, United States. Association for Computational Linguistics.

Dacheng Li, Shiyi Cao, Tyler Griggs, Shu Liu, Xiangxi Mo, Eric Tang, Sumanth Hegde, Kourosh Hakhamaneshi, Shishir G. Patil, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025. Llms can easily learn to reason from demonstrations structure, not content, is what matters! *Preprint*, arXiv:2502.07374.

Jia Liufu, Bin Hu, Linglin Jing, Terry Kong, Yuki Huang, Ashwath Aithal, Wenming Yang, and Jun Yang. 2026. FC-SWE: Failure-conditioned RL for long-horizon software engineering agents. *Preprint*, arXiv:2610.07898.

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. 2024. Swt-bench: testing and validating real-world bug-fixes with code agents. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24.

OpenAI. 2025. Gpt5.

OpenAI. 2026. Gpt5.5.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2025. Training software engineering agents and verifiers with swe-gym. In *Proceedings of the 42nd International Conference on Machine Learning*, ICML'25. JMLR.org.

Renjie Pi, Grace Lam, Mohammad Shoeybi, Pooya Jannaty, Bryan Catanzaro, and Wei Ping. 2026. On data engineering for scaling llm terminal capabilities. *Preprint*, arXiv:2602.21193.

Ethan Shen, Daniel Tormoen, Saurabh Shah, Ali Farhadi, and Tim Dettmers. 2026. SERA: Soft-verified efficient repository agents. In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, pages 111125–111142. PMLR.

KaShun Shum, Binyuan Hui, Jiawei Chen, Lei Zhang, X. W., Jiaxi Yang, Yuzhen Huang, Junyang Lin, and Junxian He. 2025. Swe-rm: Execution-free feedback for software engineering agents. *Preprint*, arXiv:2512.21919.

Huatong Song, Lisheng Huang, Shuang Sun, Jinhao Jiang, Ran Le, Daixuan Cheng, Guoxin Chen, Yiwen Hu, Zongchao Chen, Yiming Jia, Wayne Xin Zhao, Yang Song, Tao Zhang, and Ji-Rong Wen. 2026. Swe-master: Unleashing the potential of software engineering agents via post-training. *Preprint*, arXiv:2602.03411.

Shuang Sun, Huatong Song, Lisheng Huang, Jinhao Jiang, Ran Le, Zhihao Lv, Zongchao Chen, Yiwen Hu, Wenyang Luo, Wayne Xin Zhao, Yang Song, Hongteng Xu, Tao Zhang, and Ji-Rong Wen. 2026. Swe-world: Building software engineering agents in docker-free environments. *Preprint*, arXiv:2602.03419.

Chaofan Tao, Jierun Chen, Yuxin Jiang, Kaiqi Kou, Shaowei Wang, Ruoyu Wang, Xiaohui Li, Sidi Yang, Yiming Du, Jianbo Dai, Zhiming Mao, Xinyu Wang, Lifeng Shang, and Haoli Bai. 2026. Swe-lego: Pushing the limits of supervised fine-tuning for software issue resolving. *Preprint*, arXiv:2601.01426.

Qwen Team. 2026. Qwen3.5-omni technical report. *Preprint*, arXiv:2604.15804.

Xingyao Wang, Valerie Chen, Heng Ji, and Graham Neubig. 2026. A rubric-supervised critic from sparse real-world outcomes. *Preprint*, arXiv:2603.03800.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2025. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. *Preprint*, arXiv:2201.11903.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, LINGMING ZHANG, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. 2025. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Yiqing Xie, Emmy Liu, Gaokai Zhang, Nachiket Kotalwar, Shubham Gandhi, Sathwik Acharya, Xingyao Wang, Carolyn Rose, Graham Neubig, and Daniel Fried. 2026. Hybrid-gym: Training coding agents to generalize across tasks. In *Forty-third International Conference on Machine Learning*.

Yiqing Xie, Alex Xie, Divyanshu Sheth, Pengfei Liu, Daniel Fried, and Carolyn Rose. 2025. RepoST: Scalable repository-level coding environment construction with sandbox testing. In *Second Conference on Language Modeling*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025a. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

John Yang, Kilian Lieret, Carlos E Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2025b. SWE-smith: Scaling data for software engineering agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Sidi Yang, Chaofan Tao, Jierun Chen, Tiezheng Yu, Ruoyu Wang, Yuxin Jiang, Yiming Du, Wendong Xu, Jing Xiong, Taiqiang Wu, Lifeng Shang, Xiaohui Li, Ngai Wong, and Haoli Bai. 2026. What makes interaction trajectories effective for training terminal agents? *Preprint*, arXiv:2606.03461.

Danlong Yuan, Wei Wu, Zhengren Wang, Xueliang Zhao, Huishuai Zhang, and Dongyan Zhao. 2026. SWE-MiniSandbox: Container-free reinforcement learning for building software engineering agents. In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, pages 152151–152167. PMLR.

Junkai Zhang, Zihao Wang, Lin Gui, Swarnashree Mysore Sathyendra, Jaehwan Jeong, Victor Veitch, Wei Wang, Yunzhong He, Bing Liu, and Lifeng Jin. 2026. Chasing the tail: Effective rubric-based reward modeling for large language model post-training. *Preprint*, arXiv:2509.21500.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. 2025. Commit0: Library generation from scratch. In *The Thirteenth International Conference on Learning Representations*.

Dewu Zheng, Ruizhe Ye, Yanlin Wang, Yang Ye, Hongyu Zhang, Ensheng Shi, Xilin Liu, Yuchi Ma, Jianxing Yu, and Zibin Zheng. 2026. Swe-prime: Fewer trajectories, better performance. *Preprint*, arXiv:2608.27449.

Yiqi Zhu, Apurva Gandhi, and Graham Neubig. 2025. Training versatile coding agents in synthetic environments. *Preprint*, arXiv:2512.12216.

A Appendix

A.1 Additional Related Work

Analyses on the Effectiveness of General Agent Training. Outside SWE, Agent-FLAN (Chen et al., 2024) shows that agent corpora entangle format following with reasoning and redesigns them to separate the two. With regard to reasoning, the structure and behaviors of a trace matter more than its correctness: with verification, backtracking, and related behaviors, RL would be able to gain even from incorrect solutions (Gandhi et al., 2025), and long-CoT distillation tolerates wrong answers but not shuffled or deleted steps (Li et al., 2025). Furthermore, Jin et al. (2026) find the same for agentic search, where SFT on trajectories exhibiting beneficial behaviors before RL outperforms SFT on outcome-correct ones.

A.2 Experimental Setup

Datasets and Models. We conduct evaluation on the SWE-bench Verified dataset (Jimenez et al., 2024) and analysis on both in-domain training: the R2E-Gym issue-solving dataset (Jain et al., 2025), and transfer learning: the Hybrid-Gym function localization dataset (Xie et al., 2026). We experiment on 3 student models: Qwen2.5Coder-7B (Hui et al., 2024), Qwen3-4B, and Qwen3-8B (Yang et al., 2025a) and 8 teacher models: Claude-Opus-4.5 (Anthropic, 2025), Claude-Opus-4.7 (Anthropic, 2026), Claude-Sonnet-3.5 (Anthropic, 2024), GPT-5-mini (OpenAI, 2025), GPT-5.5 (OpenAI, 2026), DeepSeek4-flash (DeepSeek-AI and et al., 2026), Qwen3-80B (Yang et al., 2025a), and Qwen3.5-397B (Team, 2026).

Training Details. We fine-tune each student on the teacher demonstrations with supervised fine-tuning: we use 8 A6000 GPUs and search hyper-parameters in the range of learning-rate={5e-5, 1e-4}, num-epochs={3, 5}, and batch-size={8, 16}.

Training and Evaluation Prompts. All Hybrid-Gym teacher rollouts use the function localization prompt in Figure 7, which asks for a single function or class, except the strategy-prompted rollouts in §A.6. The students are evaluated on SWE-Bench Verified (Jimenez et al., 2024) with the issue-solving prompt in Figure 9. On R2E-Gym, whose images come with a ready virtual environment instead of conda, the Claude-Sonnet-3.5 rollouts come from the original R2EGym dataset and use the original R2E-Gym prompt (Jain et al., 2025) (Figure 8). Other models use the same prompt as our SWE-Bench Verified evaluation.

Function Localization Prompt

```
I've uploaded a python code repository in the
directory {{ workspace_dir_name }}.
Your goal is to locate the {{ module_type }} described
below and write its docstring.

<{{ module_type }}_description>
{{ description }}
</{{ module_type }}_description>

Known details:
- Target type: {{ module_type }}
- The original docstring was removed; please
restore/author it.
- Note: The target name is not provided. You must
identify the correct {{ module_type }} based on the
functionality description above.

Follow these steps:
1. EXPLORATION: Search the codebase (no file path
provided) to find where this {{ module_type }} is
defined.
2. UNDERSTANDING: Read the implementation to fully
understand its arguments, return values, and side
effects.
3. RECHECK: Verify that the {{ module_type }} you
located actually matches the given description.
Compare the implementation behavior with the
description to ensure you found the correct target.
4. GENERATION: Insert a clear, concise Python
docstring inside the definition using triple quotes.
Cover purpose, parameters, return value(s), and
important behaviors or exceptions.
5. REVIEW: Double-check formatting and accuracy;
avoid changing code outside the docstring unless
strictly necessary.
Be thorough in your exploration and reasoning. It's
fine if your thinking process is lengthy - quality and
completeness are more important than brevity.
```

Figure 7: The Hybrid-Gym function localization prompt. `module_type` is `function` or `class`.

Original R2E-Gym Prompt

```
Consider the following github issue:
<github_issue>
{{ instance.problem_statement }}
</github_issue>

Can you help me implement the necessary changes to the
repository to fix the <github_issue>?
I've already taken care of all changes to any of the
test files described in the <github_issue>. This means
you DON'T have to modify the testing logic or any of
the tests in any way!
Your task is to make the minimal changes to non-tests
files in the /testbed directory to ensure the
<github_issue> is satisfied.

IMPORTANT TIP:
Follow these steps to resolve the issue:
1. As a first step, it might be a good idea to explore
the repo to familiarize yourself with its structure.
2. Create a script ('reproduce_issue.py') to reproduce
the error and execute it to confirm the error
3. Edit the sourcecode of the repo to resolve the
issue
4. Rerun your reproduce script and confirm that the
error is fixed!
5. Think about edgecases and make sure your fix
handles them as well
```

Figure 8: The original R2E-Gym prompt (Jain et al., 2025), used by some of the R2E-Gym rollouts.

Issue-Solving Prompt

```
I have access to a python code repository in the directory {{ instance.repo_path }} . You can explore and modify files
using the available tools. Consider the following issue description:

<issue_description>
{{ instance.problem_statement }}
</issue_description>

Can you help me implement the necessary changes to the repository so that the requirements specified in the
<issue_description> are met?
I've already taken care of all changes to any of the test files described in the <issue_description>. This means you DON'T
have to modify the testing logic or any of the tests in any way!
Also the development Python environment is already set up for you (i.e., all dependencies already installed), so you don't
need to install other packages.
Your task is to make the minimal changes to non-test files in the {{ instance.repo_path }} directory to ensure the
<issue_description> is satisfied.

Follow these phases to resolve the issue:

Phase 1. READING: read the problem and reword it in clearer terms
1.1 If there are code or config snippets. Express in words any best practices or conventions in them.
1.2 Hightlight message errors, method names, variables, file names, stack traces, and technical details.
1.3 Explain the problem in clear terms.
1.4 Enumerate the steps to reproduce the problem.
1.5 Hightlight any best practices to take into account when testing and fixing the issue

Phase 2. RUNNING: install and run the tests on the repository
2.1 Activate the environment by running
./opt/miniconda3/etc/profile.d/conda.sh ; conda activate testbed
2.2 Follow the readme
2.3 Install the environment and anything needed
2.4 Iterate and figure out how to run the tests

Phase 3. EXPLORATION: find the files that are related to the problem and possible solutions
3.1 Use `grep` to search for relevant methods, classes, keywords and error messages.
3.2 Identify all files related to the problem statement.
3.3 Propose the methods and files to fix the issue and explain why.
3.4 From the possible file locations, select the most likely location to fix the issue.

Phase 4. TEST CREATION: before implementing any fix, create a script to reproduce and verify the issue.
4.1 Look at existing test files in the repository to understand the test format/structure.
4.2 Create a minimal reproduction script that reproduces the located issue.
4.3 Run the reproduction script to confirm you are reproducing the issue.
4.4 Adjust the reproduction script as necessary.

Phase 5. FIX ANALYSIS: state clearly the problem and how to fix it
5.1 State clearly what the problem is.
5.2 State clearly where the problem is located.
5.3 State clearly how the test reproduces the issue.
5.4 State clearly the best practices to take into account in the fix.
5.5 State clearly how to fix the problem.

Phase 6. FIX IMPLEMENTATION: Edit the source code to implement your chosen solution.
6.1 Make minimal, focused changes to fix the issue.

Phase 7. VERIFICATION: Test your implementation thoroughly.
7.1 Run your reproduction script to verify the fix works.
7.2 Add edge cases to your test script to ensure comprehensive coverage.
7.3 Run existing tests related to the modified code to ensure you haven't broken anything.

8.   FINAL REVIEW: Carefully re-read the problem description and compare your changes with the base commit {{
instance.base_commit }}.
8.1 Ensure you've fully addressed all requirements.
8.2 Run any tests in the repository related to:
8.2.1 The issue you are fixing
8.2.2 The files you modified
8.2.3 The functions you changed
8.3 If any tests fail, revise your implementation until all tests pass

Be thorough in your exploration, testing, and reasoning.   It's fine if your thinking process is lengthy -
quality and completeness are more important than brevity.
```

Figure 9: The issue-solving prompt used to evaluate the students on SWE-Bench Verified (first box) and its Phase 2 in our R2E-Gym rollouts (second box).

A.3 Computation of Significance

With the paired bootstrap test of Berg-Kirkpatrick et al. (2012), we test whether two students differ in resolved rate on SWE-Bench Verified. Let $\delta(x)$ be the gain in resolved rate of the better student over the other on the 500 instances x. We draw $b = 10^6$ samples $x^{(i)}$ of 500 instances from x with replacement and estimate the one-sided p-value as the fraction of samples with $\delta(x^{(i)}) \geq 2\delta(x)$; the threshold is $2\delta(x)$ rather than $\delta(x)$ because the samples are centered at $\delta(x)$ instead of at 0. We mark a change as significant if $p < 0.05$.

A.4 List of Candidate Behaviors

In Table 7, we provide a list of candidate behaviors on which we do not conduct further experiments. The reason is that we prioritize the behaviors that are more likely to be useful for training.

A.5 Detailed Analysis Results

A.5.1 Exploration Length is Not a Good Indicator of Trajectory Effectiveness

Following §3.2, we show that the exploration length of demonstrations is, however, not a good indicator of trajectory effectiveness.

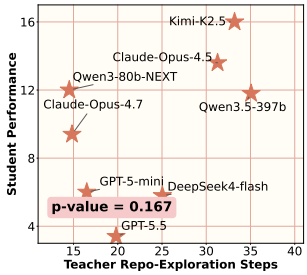


Figure 10: The exploration length of demonstrations is also not a good indicator of demonstration effectiveness.

A.5.2 Strong Teacher Does Not Always Translate to Strong Student

Some Claude-Opus-4.7 demonstrations open the target file without saying why they chose it, as is shown in Table 8. On an instance that asks for a class with "attributes [...] such as their nickname, username, and hostname", Claude-Opus-4.7 lists the repository's Python files, cut at 50 by `head`, and then opens the 9th entry, the target file, saying only "The file `sopel/tools/target.py` looks promising."; it runs no content search such as `grep` and does not narrow the listing in any form. In other words, the trajectory itself does not show any reason why it picks the target file out of the 50 searched results. In comparison, Claude-Opus-4.5 has a clearer strategy for file localization. It first names four candidate files with a reason each, and it checks where the class is used before editing it. Similarly, for function localization, on `open-mmlab__mmengine-304_415`, Claude-Opus-4.7 likewise opens the 44th entry of a 50-file listing ("The class description matches `base_model.py`."), while Claude-Opus-4.5 first lists the six requirements of the description and browses two directories.

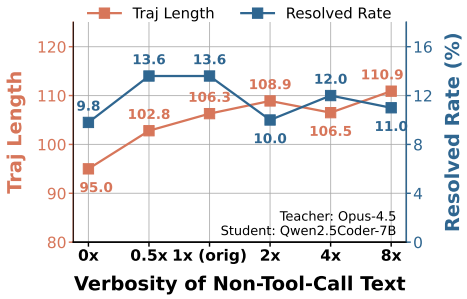


Figure 11: When we increase the verbosity of non-tool call text, the trajectory length in general increases. However, the performance does not change much.

A.5.3 Non-Tool-Call Text Increases Trajectory Length

Similar to §3.6, we observe that when we only increase the verbosity of non-tool call text, the trajectory length in general increases, but the performance doesn't change much.

A.6 Concrete Data Intervention Implementation for each Candidate Behavior

In this section, we introduce how we design experiments to control the presence or absence of each candidate behavior in training demonstrations. We manually design different experiments for different behaviors.

A.6.1 Label Demonstrations with the Presence of Repo-Exploration Strategies

We label the two strategies with rule-based statistics.

Multi-Search. A keyword search is a `terminal` command that contains `grep` or `rg`, or `find` with `-name`. A trajectory exhibits the "Multi-Search" behavior if it runs at least two keyword searches before its first file-editing action (`file_editor` with `create`, `str_replace`, or `insert`, or a `sed -i` or

Behavior	Judge Metric
Search frequency	The frequency of search ($\geq$ 15: high; 7–14: medium; $<$ 7: low)
Search-view ratio	The ratio of keyword searching to file-viewing commands is > 3
Multi-round file view	Run file-viewing actions for ≥ 3 times
Start with thinking	Use the `<function=think>` tool as the first action
Start with task-tracker	Use the `<function=task_tracker>` tool as the first action
Frequent command chain	Use command chaining on 50% or more of terminal commands
View file before edit	Has a file-viewing action before the first file-editing action
Use Git commands	Use `git log`, `git diff`, or other Git commands

Table 7: The candidate behaviors discovered by **CATRADAR** that we do not conduct further experiments on.

Claude-Opus-4.7 (5 actions)	**Claude-Opus-4.5 (16 actions)**
1 `find <repo> -type f -name "*.py" | head -50 && ls <repo>` → 50 file paths (listing cut by head)	1 `view /workspace` → directory tree
2 "The file `sopel/tools/target.py` looks promising. Let me check it." `view sopel/tools/target.py` → **target class `User`**	2 `view sopel/` → directory tree
3 `str_replace`: add the docstring to `User`	3 `think`: "[...] this sounds like a User class in an IRC context. Let me look at relevant files: 1. `sopel/tools/target.py` - the name 'target' could represent users/channels 2. `sopel/trigger.py` - might contain user-related classes 3. `sopel/bot.py` - could contain user tracking classes 4. `sopel/irc.py` - IRC-specific classes"
4 `python -c "from sopel.tools.target import User; print(User.__doc__)"`	4 `view sopel/tools/target.py` → **target class `User`**
5 `finish`	5 `think`: "I found the `User` class [...]"
	6 `grep` for the modules that import `User`
	7–8 `view sopel/bot.py`, `view sopel/coretasks.py`
	9 `think`: "Now I'm certain the `User` class [...] is the correct target."
	10 `str_replace`: add the docstring to `User`
	11–16 view the edited file, import check, run the test suite (3 commands), `finish`

Table 8: Demonstrations of Claude-Opus-4.7 and Claude-Opus-4.5 on the Hybrid-Gym instance `sopel-irc__sopel-1344_101` (both successful). `<repo>` abbreviates the repository path, and file paths are shown relative to it. Claude-Opus-4.7 opens the target file directly from a listing cut at 50 files, while Claude-Opus-4.5 takes a more comprehensive strategy: picking several candidate files with detailed reasoning, and then reading them one-by-one (action 3).

`git apply` command).

Target-Read. We take the first action that is a `file_editor view` (of a file or a directory), a `terminal` command that contains `cat`, `head`, `tail`, `less`, or `more`, or a file-editing action. Its path is the `file_editor` path, or the first argument after `cat`, `head`, `tail`, `less`, or `more` in the command. The trajectory exhibits Target-Read if an earlier keyword search lists at most 30 distinct files and its output contains this path: a listed path equals it, ends with it, or is a suffix of it, has the same file name, or the path or its file name occurs in the output text.

A.6.2 Prompt the Models to Adopt our Repo-Exploration Strategies

This section introduces the experimental settings for the "prompt teacher" and "prompt student" ablations in Table 5. **Prompt Teacher**. To instruct the teacher to follow the strategies for the Hybrid-Gym function localization task, we add the `<localization_strategy>` block in Figure 12 after the steps of the function localization prompt (Figure 7) and end step 1 with "Use the localization strategy below." GPT-5-mini produces 1,289 successful demonstrations with this prompt, which keep the strategy block in their task prompt. It produces 1,346 successful demonstrations with the original prompt (i.e., without the strategy block). We train the student on all 1,289 demonstrations

and evaluate it with the unchanged issue-solving prompt (Figure 9).

Prompt Student. As for the "prompt student" ablation, we train the student model on the original 1,346 GPT-5-mini demonstrations. In inference, we insert the `<repo_exploration_strategy>` block in Figure 12 in the prompt for the SWE-Bench Verified issue-solving task, which states the same principles for issue solving, after Phase 3 (EXPLORATION) of the issue-solving prompt at evaluation time.

A.6.3 Add Task-Tracker Steps to Training Demonstrations

Neither Claude-Opus-4.7 nor GPT-5-mini uses the task-tracker tool in its original Hybrid-Gym demonstrations. To implement the "Add Task-Tracker" ablation, we insert LLM-synthesized task-tracker plans at four points of each demonstration: (1) before the first action; (2) once the target file is found, i.e., before the last action prior to the first edit that mentions the target file (the file of the last successful edit); (3) before the first file-editing action; and (4) right after the output of the first file-editing action. If two points coincide, only the earlier point's plan is inserted. Each plan is a task-tracker action with `command=plan`, followed by the tool's usual output, "Task list has been updated with N item(s)."

We use an LLM (DeepSeek4-flash) to write the plans at temperature 0.3 without seeing the demonstration: the first plan from the task description, asked to list 4–6 tasks with the first one in progress, and each later plan from the task description and the first plan, asked to keep the task titles and update their status. For GPT-5-mini, 1,325 demonstrations receive four plan turns and 21 receive three, and every planning step is consistent with the remaining part of the trajectory. For Claude-Opus-4.7, 1,466 demonstrations receive four plan turns and one receives three. However, we cut off 354 of these turns because they change task titles in a later plan item, or generate too few or too many plans in the first task-planner tool call.

A.6.4 Add Verification Steps to Training Demonstrations

We take the verification steps from Claude-Opus-4.5, whose demonstrations contain more of them (Table 3). For each Claude-Opus-4.7 demonstration whose instance also has a successful Claude-Opus-4.5 demonstration that first edits the same file (the same path relative to the repository root), we keep the Claude-Opus-4.7 trajectory before its first file-editing action and append the Claude-Opus-4.5 trajectory from its first file-editing action on, with the workspace directory renamed. This applies to 1,389 of the 1,467 demonstrations and we keep the other 78 unchanged. In other words, we copy both the file editing steps and verification steps from Claude-Opus-4.5's trajectories.

A.6.5 Verbosity Experiments

We use DeepSeek4-flash (with thinking disabled) to rewrite the Claude-Opus-4.5 demonstrations of Hybrid-Gym, which have 30,615 assistant turns, including 3,738 `think` and 683 task-tracker actions. For every multiple $k > 0$, only the reasoning text changes: every tool call apart from the `thought` parameter of `think` calls, and every tool output, stays byte-identical, which we verify on the released datasets.

Non-tool-call text. For Figure 11, we rewrite the text before the tool call to k times its own length in Qwen3 tokens, for $k \in \{0.5, 2, 4, 8\}$, in every turn except `think` and task-tracker turns. 10,513 of these 26,194 turns have text; the others, including 4 with a malformed tool call, stay unchanged. The rewriter sees only the turn's text and its tool call and is told to keep the meaning and intent, not to invent facts, findings, file contents, or results that the text and the call do not imply, and to write as the agent at this step: longer versions elaborate on what the agent is looking for, why, and what it expects to learn, without adding steps or discoveries; shorter versions condense the same content. The versions of a turn are written as a ladder in one request: the $0.5\times$ version condenses the original, the $2\times$ version elaborates it, and the $4\times$ and $8\times$ versions each elaborate the previous one. Furthermore, a version is accepted if its length is within $\pm 25\%$ of the target; otherwise it is requested again with its measured length as feedback, for at most three rounds, after which the turn keeps its original text, as do turns whose target is below 4 or above 2,000 tokens. On average, the rewritten texts are 0.5, 1.9, 3.9, and 8.1 times as long as the originals. Out of the 10,513 turns with text, 0.8%, 1.3%, 3.0%, and 4.1% keep their original text in the first release described below. The second release has the same rounded rates except at $8\times$ (4.2%). The $0\times$ variant removes the text of all turns except the `think` and task-tracker turns.

Thinking Content. For Figure 6, we also rewrite

Strategy Block for the Teacher

```
<localization_strategy>
When searching for the target {{ module_type }}, follow these two principles
(summarized from successful trajectories):

1. MULTI-ROUND REFINEMENT. Treat localization as iterative. After each
search, look at the matches as a whole, pick the most promising signal,
and run a follow-up search refining on it (different keyword, restricted
path, different surrounding pattern). Do not commit to a single candidate
on the first search - expect 2-3 rounds of refinement before you have a
short list.

2. READ AFTER NARROWING. Do not open and read full file contents until you
have narrowed to a small handful (<=3) of plausible candidates. Opening
files too early wastes context on irrelevant code. Once you have a short
list, then read each candidate's definition and compare against the
description to pick the correct one.
</localization_strategy>
```

Strategy Block for the Student

```
<repo_exploration_strategy>
When searching the repository for the file(s)/function(s) relevant to the issue,
follow these two principles (summarized from successful trajectories):

1. MULTI-ROUND REFINEMENT. Treat localization as iterative. After each search,
look at the matches as a whole, pick the most promising signal, and run a
follow-up search refining on it (different keyword, restricted path, different
surrounding pattern). Do not commit to a single candidate on the first search -
expect 2-3 rounds of refinement before you have a short list.

2. READ AFTER NARROWING. Do not open and read full file contents until you have
narrowed to a small handful (<=3) of plausible candidates. Opening files too
early wastes context on irrelevant code. Once you have a short list, read each
candidate's definition and compare against the issue description to pick the
correct one.
</repo_exploration_strategy>
```

Figure 12: The strategy blocks added to the teacher's function localization prompt (first box; `module_type` is `function` or `class`) and to the student's issue-solving prompt (second box).

the `thought` parameter of each `think` call, and the text before the call, to the same multiple of its own length; the other turns are as in the non-tool-call setting with the same k. The rules for thoughts are the same, adapted to the agent's reasoning: keep its reasoning, conclusions, and concrete references (files, functions, line numbers), and reach no new conclusions or plan different steps. Thoughts in the trajectories are long (median 304 tokens), so every version above 600 tokens is written in several parts, each continuing the text so far, up to 8,000 tokens. The $0\times$ variant removes all `think` actions with their outputs and the text of all other turns except the task-tracker turns, which stay whole (99 of them have text).

Post-Checks. We do not check the meaning of the rewrites. Instead, we check: (1) whether the rewriting describes the rewriting task instead of the agent's reasoning (e.g., "the thought doesn't mention testing, so I won't plan that"; 4.6%, 10.0%, and 19.8% of the thoughts at $2\times$, $4\times$, and $8\times$) or ended with JSON left from the rewriter's reply (3.3% of the rewritten $2\times$ and $4\times$ texts); (2) whether the written reasoning text is natural and consistent with the remaining part of the trajectory. We keep the original reasoning text if the rewriting does not meet any of the requirements.