SWE-ROAM: Evaluating Repository-Level World Models through Active State Recovery

Anonymous ACL submission

Abstract

Large language models (LLMs) are increasingly deployed as software engineering (SWE) agents, motivating the use of an internal *world model* to anticipate how complex repositories evolve after sequential tool interactions. However, existing execution benchmarks primarily evaluate isolated code snippets in stateless settings, failing to capture the immense state complexity and strict partial observability of real-world repositories. To address this gap, we introduce SWE-ROAM, a diagnostic benchmark designed to systematically evaluate repository-level world modeling over real agent trajectories. SWE-ROAM decouples environment simulation into step-level transition prediction and trajectory-level reward auditing. By evaluating frontier LLMs under both passive reasoning (Non-Agent mode) and active state recovery (Agent mode), we uncover a fundamental dissociation: strong local execution prediction does not guarantee reliable global outcome auditing. Furthermore, our results demonstrate that active exploration substantially improves transition predictions, identifying partial observability as a primary modeling bottleneck. Finally, our online deployment evaluation reveals a critical *sim-to-real* gap, where high single-step accuracy severely overestimates multi-step reliability due to cascading errors. Ultimately, our findings establish that future SWE world models must be observability-aware, integrating active state retrieval with robust execution simulation to successfully bridge the gap. Our dataset and code are available[1].

1 Introduction

Recent advances in large language models (LLMs) have enabled software engineering (SWE) agents to resolve real-world repository issues through multi-step interaction (Jimenez et al., 2024; Chowdhury et al., 2024). Rather than addressing complex software defects in a single forward pass, these agents operate within an iterative loop: querying codebases, modifying source files, executing commands, and refining strategies based on environmental feedback (Yang et al., 2024; Xia et al., 2025; Ferrag et al., 2025). In this paradigm, progress is driven not just by the model's parametric knowledge, but by ground-truth signals from an external execution environment that reveals repository states and verifies test oracles.

However, this heavy reliance on live execution creates a severe bottleneck. Instantiating dependency-complete, containerized workspaces (*e.g.*, Docker) is computationally prohibitive, brittle, and difficult to scale (Jain et al., 2025). Furthermore, failed actions inevitably leave behind modified or corrupted states that are notoriously difficult to branch, rewind, or reuse for exploration (Yang et al., 2026). Consequently, current agents are restricted to sparse, linear interactions, lacking the ability to cheaply evaluate the vast space of alternative counterfactual paths that could yield superior repairs (Badertdinov et al., 2026).

A reliable repository-level world model would break this bottleneck (Ding et al., 2025). If an internal model could accurately forecast environmental responses to arbitrary actions, it would provide cheap, instantaneous supervision during training (Sun et al., 2026; Yang et al., 2025b; Tao et al., 2026). This capability would unlock massive counterfactual rollouts for reinforcement learning (RL) and allow agents to look ahead and plan before committing to expensive or irreversible commands (Pan et al., 2024; Luo). For example, an agent could simulate multiple candidate edits, filter out unpromising trajectories, and reserve live test execution strictly as a final authority for high-value correctness verification (Copet et al., 2025; Xia et al., 2024).

However, the extent of world-modeling capabilities that current LLMs actually possess in

[1] https://anonymous.4open.science/r/SWE-ROAM/

repository-level settings remains unclear. Crucially, software repositories diverge sharply from the traditional simulated domains (*e.g.*, games or grid-worlds) where world models are typically evaluated. In classical simulations, the current observation is often a compact, nearly complete representation of the task-relevant state, reducing the challenge primarily to learning transition dynamics (Ha and Schmidhuber, 2018; Zeng et al., 2025). In contrast, repository states are massive, distributed, and strictly *partially observed*. The information required to predict the next observation is routinely hidden within unread source files or uninspected directories. Consequently, a model's failure to predict the environment often stems from a lack of access to the requisite underlying state rather than an inability to simulate complex computational dynamics.

To systematically evaluate this capability, we introduce SWE-ROAM, a diagnostic benchmark designed to probe repository-level world modeling over real SWE-agent trajectories. SWE-ROAM instantiates a paradigm shift from conventional *passive environment simulation* (*i.e.*, predicting observations solely from text histories) to an *active state recovery* process. The benchmark comprises two core tasks: *Transition Prediction*, which requires a model to forecast the exact observation returned by the next action, and *Reward Prediction*, which evaluates trajectory-level causal auditing by judging whether a completed patch resolves the issue. To isolate passive reasoning from active state recovery, we evaluate frontier models in two distinct modes: *Non-Agent mode*, which restricts the model to static text histories, and *Agent mode*, which empowers the model to perform active, read-only exploration across the codebase before submitting its prediction.

Evaluating eight frontier LLMs on SWE-ROAM reveals several critical insights into current world-modeling capabilities. First, transition prediction is heavily bottlenecked by observability: active exploration substantially improves observation-dependent actions (*e.g.*, viewing files) but provides negligible gains for execution prediction (*e.g.*, test outcomes). Second, strong local transition simulation does not guarantee reliable trajectory-level reward auditing, demonstrating that local environment prediction and global outcome judgment are fundamentally dissociated capabilities. Finally, deploying learned transition models in an online SWE-agent loop via MODELWORLDENV exposes a severe sim-to-real gap (Kadian et al., 2020; Wagenmaker et al., 2024): while single-step accuracy is remarkably high, minor hallucinations cascade catastrophically during multi-step deployment, preventing pure neural simulators from seamlessly replacing live execution environments.

Our contributions are threefold:

- **A diagnostic benchmark for repository-level world modeling.** We introduce SWE-ROAM, decoupling step-level transition simulation from trajectory-level outcome auditing under passive and active input regimes.
- **An empirical decomposition of software modeling failures.** We demonstrate that local simulation and global auditing are independent capabilities, and that environment prediction is primarily constrained by partial observability rather than reasoning depth.
- **An online evaluation of learned software environments.** We introduce MODELWORLDENV to quantify the sim-to-real gap, revealing how cascading errors limit the viability of current world models in closed-loop agent deployment.

2 Task Formulation and Dataset Construction

We construct SWE-ROAM using real software-engineering (SWE) agent trajectories to ensure our evaluation reflects the complexity of actual repository environments. The construction process is designed to formally define two distinct evaluation tasks, isolating step-level state transitions from trajectory-level outcome auditing.

2.1 Preliminary: Repository-Level Interaction

We formulate each task instance as a repository-level issue-resolution problem:

$$\mathcal{I} = (\mathcal{R}, b, d, \mathcal{U}), \tag{1}$$

where $\mathcal{R}$ is the target repository, b is the base commit, d is the natural language issue description, and $\mathcal{U}$ denotes the hidden suite of evaluation tests used by the SWE-bench oracle. Crucially, the hidden tests $\mathcal{U}$ are used exclusively by the oracle to assign ground-truth labels and are never included in the model inputs.

A physical environment initializes a mutable workspace from $(\mathcal{R}, b)$. At step t, the workspace

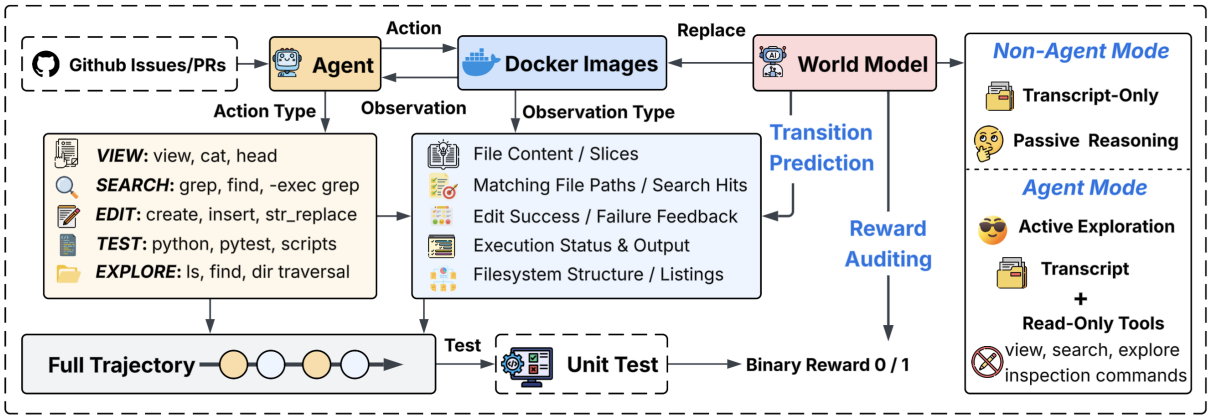

Figure 1: Task illustration of our SWE-ROAM benchmark.

state is denoted by s_t. This state is comprehensive: it encompasses the file system, newly created files, accumulated edits, and execution-side configurations. The agent issues an action $a_t \in \mathcal{A}$, and the environment updates the workspace while returning an observation o_t:

$$(o_t, s_{t+1}) = \mathcal{E}_{\mathcal{I}}(a_t, s_t). \tag{2}$$

The observation o_t may consist of file contents, search hits, edit feedback, or test execution logs. A complete interaction trajectory of length T is represented as:

$$\tau = (\mathcal{I}, (a_1, o_1), \ldots, (a_T, o_T), P_T, r)\,, \tag{3}$$

where P_T is the final submitted patch and $r \in \{0, 1\}$ is the ground-truth success label, determined by applying P_T and executing the hidden tests $\mathcal{U}$.

To prevent the benchmark from measuring whether a model can exploit the agent's private reasoning rather than modeling the repository environment, we explicitly exclude internal agent thoughts from the inputs for both tasks. Trajectories are represented strictly as objective action-observation traces. Furthermore, this formalization highlights that repository-level world modeling is inherently *stateful*. The observation o_t is not determined by a_t alone; it depends heavily on the trajectory prefix through the induced hidden state s_t. For example, earlier edits modify files, fundamentally altering the expected return values of subsequent view or test actions.

2.2 Source Trajectories

SWE-ROAM is built from trajectories generated by OpenHands (Wang et al., 2025) powered by Qwen3-30B-Coder (Yang et al., 2025a) on SWE-bench Verified instances (Chowdhury et al., 2024). These trajectories record realistic, multi-step repair attempts within Docker environments (via R2E-Gym) (Jain et al., 2025). From this corpus, we formulate two distinct evaluation tasks, each supported by a specialized dataset.

2.3 Task 1: Transition Prediction

This task evaluates a model's ability to simulate step-level environment dynamics. The model must predict the exact observation the environment would return for a target action under the hidden repository state s_t.

To construct the testbed for the task, for a given trajectory τ, let the action-observation prefix prior to step t be $h_{<t} = ((a_1, o_1), \ldots, (a_{t-1}, o_{t-1}))$. We formulate a Transition Prediction example by sampling a non-terminal step t:

$$x_t^{\text{trans}} = (d, h_{<t}, a_t)\,, \qquad y_t^{\text{trans}} = o_t. \tag{4}$$

The resulting dataset is defined as:

$$\mathcal{D}_{\text{trans}} = \{(d_i, h_{i,<t}, a_{i,t}, o_{i,t}, c_{i,t})\}_{i=1}^{N_{\text{trans}}}\,, \tag{5}$$

where $c_{i,t}$ denotes the semantic category of the target action. We randomly sample $N_{\text{trans}} = 1,000$ transition points, rigorously deduplicating them using the unique tuple $(\texttt{instance_id}, \texttt{traj_idx}, \texttt{step_idx})$ to prevent evaluation bias.

Each target action is classified into one of five categories (Table 1). We ensure that all five action categories are well-represented and report category-level metrics to avoid hiding category imbalance in the overall score. This categorization enables fine-grained capability probing: VIEW, SEARCH, and EXPLORE test the recovery of hidden states; EDIT evaluates the application of operations to the current file; and TEST demands rigorous execution-level prediction.

Category	Typical Actions	Prediction Target
VIEW	`view`, `cat`, `head`	File contents or file slices
SEARCH	`grep`, `find -exec grep`	Matching file paths or search hits
EDIT	`create`, `insert`, `str_replace`	Edit success/failure feedback
TEST	`python`, `pytest`, scripts	Execution status and output
EXPLORE	`ls`, `find`, directory traversal	Filesystem structure or listings

Table 1: Action categories for Task 1: Transition Prediction. Categories are defined by semantic roles rather than surface syntax.

2.4 Task 2: Reward Prediction

This task assesses a model's capacity for trajectory-level outcome auditing. We frame this as a binary classification problem where the model must logically audit the sequence of applied edits to determine if the final patch successfully resolves the issue.

To construct the corresponding dataset, we curate $N_{\text{reward}} = 800$ complete trajectories, approximately balanced between successful and failed repairs. Because raw records may contain terminal actions or final observations that reveal explicit success signals, we apply a strict decontamination function $\mathcal{F}$ to obtain a sanitized trajectory $\bar{\tau} = \mathcal{F}(\tau)$.

The function $\mathcal{F}$ enforces two additional constraints beyond masking agent thoughts: (1) the final `submit` action is removed; and (2) the final environment observation is truncated to a maximum of 1,000 characters. The Reward Prediction example is thus:

$$x^{\text{reward}} = (d, \bar{\tau}), \qquad y^{\text{reward}} = r. \tag{6}$$

Consequently, the model must deduce the outcome by analyzing the causal implications of the patch rather than relying on explicit completion messages.

3 Experimental Setup

We organize our evaluation around three experimental layers.

First, we evaluate the core SWE-ROAM benchmark, which consists of Transition Prediction and Reward Prediction under both Non-Agent and Agent modes (Yao et al., 2022). This main evaluation measures whether frontier LLMs can predict local environment feedback and audit complete repair trajectories.

Second, we study inference-time scaling (Muennighoff et al., 2025) by varying either the amount of active repository exploration or the depth of passive chain-of-thought prompting. This isolates whether performance gains come from external state recovery or internal reasoning.

Third, we deploy a learned transition predictor inside an online SWE-agent loop through MODEL-WORLDENV, testing whether offline world-model accuracy translates into usable multi-step environment simulation.

3.1 Passive Inputs and Active State Recovery

To disentangle execution prediction from state observability, we formalize two distinct input regimes.

Non-Agent Mode. The model operates as a passive observer over a fixed transcript. For Transition Prediction, the context is $\kappa^{\text{NA}}_{\text{trans}} = (d, h_{<t}, a_t)$; for Reward Prediction, it is $\kappa^{\text{NA}}_{\text{reward}} = (d, \bar{\tau})$.

Agent Mode. The model is granted read-only exploration over the *initial* repository state s_0. It can issue a sequence of queries $q_1, \ldots, q_K$ (subject to budget $K \leq B$). The sandbox returns $u_j = \mathcal{E}^{\text{RO}}_{\mathcal{I}}(q_j, s_0)$, forming an exploration transcript $e_{1:K} = ((q_1, u_1), \ldots, (q_K, u_K))$. The contexts are enriched to:

$$\kappa^{\text{A}}_{\text{trans}} = (d, h_{<t}, a_t, e_{1:K}), \tag{7}$$

$$\kappa^{\text{A}}_{\text{reward}} = (d, \bar{\tau}, e_{1:K}). \tag{8}$$

Because the sandbox strictly exposes s_0 rather than s_t, the model cannot trivially query the answer. It must retrieve evidence from the original codebase and mentally synthesize it with the prior edits recorded in $h_{<t}$.

To preserve sandbox integrity, observation-producing and write actions are strictly delineated. We provide three tools: `view`, `search`, and `execute_bash`. In Agent mode, `execute_bash` is restricted to non-mutating inspection commands and blocks destructive operations, network access, and workspace modifications. Write actions such as file creation, deletion (`rm`), or text replacement (`sed`), as well as network accesses, are explicitly blocked.

3.2 Evaluation Metrics

Given the structural diversity of the tasks, we design tailored metrics for step-level and trajectory-level evaluations (detailed mathematical formulations, output normalization, and specific correctness thresholds are provided in Appendix A.1).

Transition Metrics (Task 1): We evaluate step-level predictions using category-specific primary metrics rather than generic string matching. For observation-dependent actions, we measure **Content Overlap** for VIEW, **File Recall** for SEARCH, and **Set Precision** for EXPLORE. For execution-dependent actions, we apply strict binary checks: **Success Match** for EDIT and **Status Match** for TEST, based on extracted execution feedback and exit codes. In our results, the **Overall Transition Accuracy (Overall Acc)** is reported as the micro-average of predictions that meet their respective category-specific correctness thresholds (*e.g.*, 0.7 for VIEW, 1.0 for TEST), rather than a raw exact string match.

Reward Metrics (Task 2): For trajectory-level outcome auditing, we model the task as a binary classification problem. Models are prompted to output a binary success label along with a continuous confidence score. We report **Accuracy** and **AUC-ROC** against the ground-truth SWE-bench oracle labels to evaluate both absolute classification correctness and trajectory ranking capability.

3.3 Evaluated Models

We evaluate eight frontier LLMs: Gemini 3.1 Pro (Google DeepMind, 2026), Gemini 3 Flash Preview (Google DeepMind, 2025), GPT-5.4 (OpenAI, 2026), Claude Opus 4.6 (Anthropic, 2026), DeepSeek-V3.1 (DeepSeek-AI, 2025), Qwen3-480B Coder, Qwen3-30B Coder (Cao et al., 2026), and Qwen3-30B (Yang et al., 2025a). Decoding details are shown in Appenndix A.2.

3.4 Main Evaluation Protocol

We evaluate all models on two SWE-ROAM tasks under both Non-Agent and Agent modes. For Transition Prediction, we use 1,000 step-level examples from $\mathcal{D}_{\text{trans}}$ and report Overall Acc, defined as the micro-average of thresholded category-specific correctness, together with per-category metrics. For Reward Prediction, we use 800 sanitized trajectories from $\mathcal{D}_{\text{reward}}$ and report Accuracy and AUC-ROC against the SWE-bench oracle. Unless otherwise specified, Agent mode allows $B = 12$ read-only exploration calls before submission; Non-Agent mode receives the same task input without repository exploration. More details are shown in Appendix A.3.

3.5 Inference-Time Scaling and Online Deployment

We conduct two additional studies with Qwen3-30B-Coder. First, for inference-time scaling, we vary the Agent exploration budget $B \in \{1, 3, 6, 12, 20, 30\}$ and compare it with Non-Agent reasoning-depth variants (`default`, `cot_light`, `cot_heavy`). This contrasts active state recovery with passive internal reasoning.

Second, we evaluate MODELWORLDENV, where a learned Transition Model replaces part of the intermediate feedback observed by a SWE agent during rollout, while final success is always judged by the real SWE-bench oracle. We run Qwen3-30B-Coder on 100 SWE-bench Verified instances and ablate three deployment factors: replacement probability $p \in \{0.00, 0.25, 0.50, 0.75, 1.00\}$, real-environment warm-up `prefix_steps` $\in \{0, 3, 6, 10\}$, and TransitionAgent exploration `tmsteps` $\in \{0, 3, 6, 12, 20, 30\}$. Full deployment details are provided in Appendix A.5.

4 Findings

4.1 The Observability Bottleneck

Finding 1. *The primary failure mode of repository-level world models is not flawed dynamics simulation, but severe state unobservability. Active state recovery is a prerequisite for environment prediction.*

Current evaluations often assume that world models fail because they cannot accurately simulate complex execution dynamics. Our results reveal a much earlier bottleneck. As shown in Figure 2, equipping models with Agent mode substantially improves overall Transition Prediction, yet this gain is highly asymmetric. The improvements are overwhelmingly concentrated in observation-dependent actions (VIEW, SEARCH, and EXPLORE), which fundamentally require retrieving information absent from the trajectory history (*e.g.*, exact file contents or directory structures). By contrast, predicting EDIT outcomes, which relies more on action format than hidden states, sees only marginal gains. This stark contrast proves that in repository-level environments, models are not inherently poor at predicting tool outputs; rather, they are systematically starved of the underlying codebase context.

Model	Transition Prediction						Reward Prediction	
	Overall	VIEW	SEARCH	EDIT	TEST	EXPLORE	Acc	AUC
Agent Mode								
Claude Opus 4.6	79.7	76.7	70.6	91.0	84.5	70.2	**82.1**	**82.2**
GPT-5.4	89.5	96.1	96.0	91.0	83.0	79.9	55.9	54.4
Gemini 3.1 Pro	**92.9**	**96.9**	**99.5**	**92.0**	**85.5**	89.8	77.0	76.5
Gemini 3 Flash	91.5	94.7	99.3	91.5	82.0	**89.8**	80.1	80.2
DeepSeek-V3.1	83.0	94.3	88.1	89.0	67.0	75.1	66.6	66.9
Qwen3-480B Coder	82.1	89.4	93.7	87.5	67.5	69.5	57.0	55.7
Qwen3-30B Coder	74.9	73.7	86.7	83.5	65.0	68.5	69.0	69.4
Qwen3-30B	74.8	91.3	80.1	81.4	61.5	58.4	64.4	65.1
Non-Agent Mode								
Claude Opus 4.6	**79.0**	**74.0**	68.5	92.0	**89.0**	**66.8**	75.4	76.1
GPT-5.4	73.9	58.0	**74.4**	91.0	85.0	58.8	61.0	60.2
Gemini 3.1 Pro	74.2	63.3	69.3	**93.0**	82.0	65.2	**78.8**	**79.2**
Gemini 3 Flash	65.5	54.2	67.3	86.0	81.5	48.1	71.4	72.3
DeepSeek-V3.1	66.4	61.4	59.8	90.5	69.5	57.9	68.8	69.7
Qwen3-480B Coder	62.3	57.0	61.4	87.5	67.0	50.6	69.0	70.1
Qwen3-30B Coder	51.7	45.3	55.2	76.5	64.5	40.6	64.6	65.9
Qwen3-30B	50.1	49.2	54.8	70.5	66.5	33.0	60.9	62.3

Table 2: **Main results on SWE-ROAM. Transition**: Overall Acc and per-category Avg Metric (VIEW = ContentOverlap, SEARCH = FileRecall, EDIT = SuccessMatch, TEST = StatusMatch, EXPLORE = SetPrecision). **Reward**: Accuracy and AUC-ROC. Best results in each section are **bold**.

Condition	Overall	VIEW	SEARCH	EXPLORE	EDIT	TEST
Agent Mode (vary `max_steps`*)*						
`steps=1`	28.2	3.0	43.5	8.0	37.5	49.0
`steps=3`	63.6	53.5	74.5	59.0	71.5	59.5
`steps=6`	71.0	64.0	82.5	66.0	77.0	65.5
`steps=12`	**74.0**	**67.5**	81.5	72.0	80.0	**69.0**
`steps=20`	72.5	64.5	**85.0**	66.5	**81.5**	65.0
`steps=30`	73.0	65.0	84.5	**69.5**	80.0	66.0
Non-Agent Mode (vary CoT prompt variant)						
`default`	51.0	18.0	**52.5**	39.0	77.5	**68.0**
`cot_light`	50.6	16.5	52.0	45.0	76.0	63.5
`cot_heavy`	**54.5**	**19.5**	52.0	**48.0**	**88.0**	65.0

Table 3: **Inference-time compute scaling results for Qwen3-30B-Coder on Transition Prediction (1000 samples). Top**: Agent mode with varying `max_steps`. **Bottom**: Non-agent mode with varying CoT prompt variants. Per-category Avg Metric follows the same protocol as Table 2. Best result in each section is **bold**.

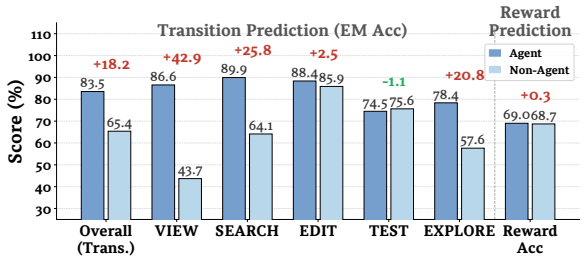

Figure 2: **Average Agent vs. Non-Agent accuracy across all models, grouped by category.** Agent mode provides large gains for observation-dependent categories (VIEW, SEARCH, EXPLORE) but negligible improvement for execution-dependent categories (TEST). Reward Prediction shows near-zero average gain (+0.3 pp).

4.2 Exploration vs. Deduction

Findings 2. *Internal deduction cannot hallucinate unobserved state. Scaling inference-time compute via CoT yields only marginal gains compared to equipping models with active external observation.*

A prevailing assumption in LLM reasoning is that deeper inference-time computation can compensate for incomplete context. We explicitly test this by contrasting *thinking harder* (scaling Chain-of-Thought depth in Non-Agent mode) with *observing more* (scaling the exploration budget in Agent mode).

The results in Table 3 strictly invalidate the assumption for repository-level environments. Deploying the heaviest CoT prompt on Qwen3-30B-Coder improves overall accuracy from 51.0% to only 54.5% (+3.5 pp). In stark contrast, Agent mode with 12 exploration steps leaps to 74.0%. The category-level breakdown explains why: no amount of sophisticated internal reasoning can deduce the contents of a source file the model has never seen. For VIEW actions, heavy CoT peaks at 19.5%, while active exploration reaches 67.5%. Active state recovery is therefore an irreplaceable mechanism; computational deduction cannot substitute for empirical observation.

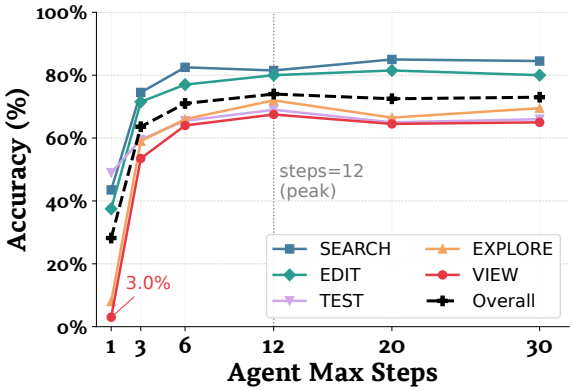

Figure 3: **Per-category accuracy vs. agent exploration steps (Qwen3-30B-Coder).**

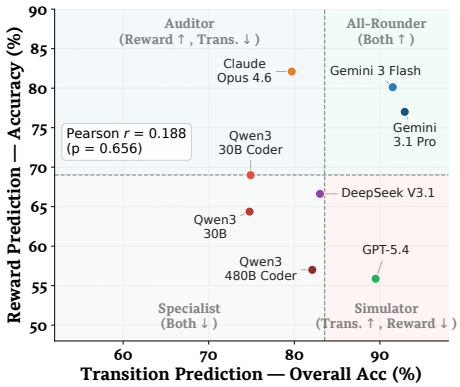

Figure 4: **Capability dissociation between Transition Prediction (x-axis) and Reward Prediction (y-axis) in Agent mode.** Dashed lines indicate mean values, dividing models into four quadrants.

Dimension	Variable	Value	Resolve Rate (%)	Notes
Baseline (100% Real Docker, p=0)				
–	–	–	**40.0**	Real Docker; upper bound
Dim1: Replace Probability (p)	*[prefix_steps=0, non-agent, n=100]*			
Dim1	replace_prob	0.00	40.0	= Baseline
		0.25	23.0	−42.5% relative drop
		0.50	5.0	Cliff: near-complete failure
		0.75	5.0	No further degradation
		1.00	1.0	Full Model World
Dim2: Real-Env Warm-up Steps	*[replace_prob=1.0, non-agent, n=100]*			
Dim2	prefix_steps	0	1.0	No warm-up
		3	1.0	
		6	1.0	
		10	**10.0**	Threshold jump (×10 over 0/3/6)
Dim3: TransitionAgent Steps	*[replace_prob=0.5, prefix_steps=0, n=100]*			
Dim3	tmsteps	0 (non-agent)	5.0	From Dim1 (p=0.5); reference
		3	20.0	×4.0 vs. non-agent
		6	15.0	Non-monotonic drop
		12	**23.0**	Global best (×4.6 vs. non-agent)
		20	15.0	Non-monotonic drop
		30	21.0	Partial recovery

Table 4: **Summary of ModelWorldEnv experiments across three dimensions.** Baseline: SWE-agent running in real Docker environment (resolve rate = 40.0%).

4.3 The Execution Simulation Frontier

Finding 3. *Active state recovery resolves the observability bottleneck but leaves execution dynamics and trajectory-level auditing largely unsolved.*

While active exploration successfully exposes static repository information, it fails to advance true computational modeling. Figure 2 demonstrates that TEST prediction (*i.e.*, forecasting the outcome of executing code) sees essentially zero improvement (-1.1 pp) in Agent mode. Predicting an execution trace requires synthesizing how trajectory edits interact with legacy dependencies to produce runtime behavior, a challenge untouched by simply reading the codebase.

This limitation extends to global evaluation. Despite Agent mode's massive pp boost to Transition Prediction, Reward Prediction gains a negligible $+0.3$ pp on average. Reward prediction requires trajectory-level causal auditing: judging whether a sequence of edits holistically constitutes a valid repair. Additional exploration often introduces noise that distracts from this holistic judgment, showing that state recovery is necessary for local prediction but insufficient for global evaluation.

4.4 Simulators vs. Auditors

Finding 4. *Local transition simulation and global outcome judgment are fundamentally dissociated capabilities. A model capable of reliably predicting the next observation may still fail to recognize whether a bug is actually fixed.*

If a world model perfectly understands how an environment responds step-by-step, it should logically understand if the final state solves the user's objective. Figure 4 proves this intuition false, revealing a near-zero correlation between Transition Prediction and Reward Prediction. Models bifurcate into distinct capability profiles: "Simulators" (*e.g.*, GPT-5.4) achieve exceptional local transition accuracy but perform near random chance on reward auditing, whereas "Auditors" (*e.g.*, Claude Opus 4.6) show weaker step-level prediction but demonstrate superior trajectory-level outcome judgment. This dissociation implies a critical takeaway for future SWE agents: training a world model purely on next-observation prediction will not automatically yield a reliable reward signal for trajectory success.

4.5 The Sim-to-Real Gap in World Models

Finding 5. *High offline transition accuracy catastrophically masks online compounding errors. The sim-to-real gap for code world models causes seemingly robust single-step predictors to collapse dur-*

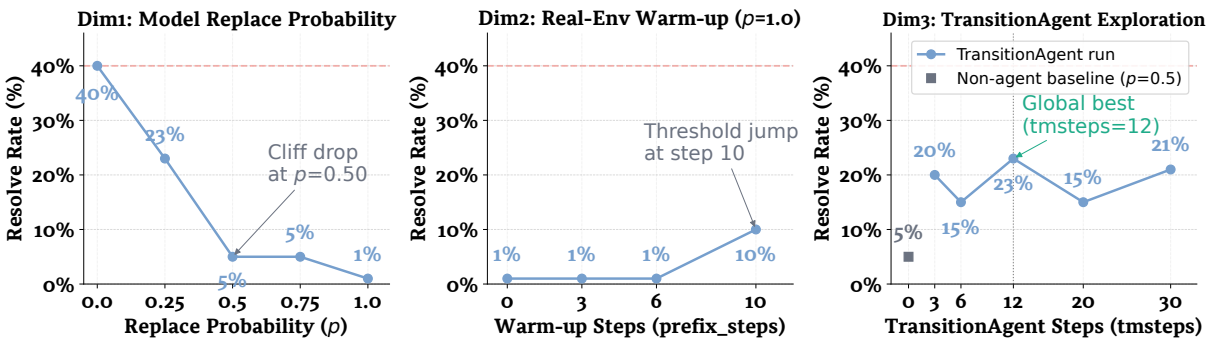

Figure 5: **ModelWorldEnv results across three dimensions.**

ing multi-step agent deployment.

Finally, we test the operational viability of substituting a real Docker environment with our learned Transition Model during actual SWE-agent execution (ModelWorldEnv). As evaluated in Table 4 and visualized in Figure 5, the results expose a severe sim-to-real gap. Routing even a small fraction of read operations to the world model drastically reduces the resolve rate compared to the real-Docker baseline, and full substitution leads to near-complete failure.

This collapse is disproportionately worse than offline step-level metrics suggest, rendering models with high single-step accuracy practically unusable over complete trajectories. The primary mechanism of failure is compounding belief-state corruption: a single hallucinated file view subtly alters the agent's understanding, leading to divergent edits that force the environment further out of distribution. Although granting the Transition Model its own active exploration layer partially mitigates this performance loss, current passive and semi-active world models remain too brittle to safely replace real execution loops.

5 Related Work

Recent applications of LLMs in software engineering have shifted from isolated code generation to repository-level issue resolution (*e.g.*, SWE-bench (Jimenez et al., 2024)). Mainstream agentic frameworks (*e.g.*, OpenHands (Wang et al., 2025), SWE-agent (Yang et al., 2024)) rely heavily on containerized environments (*e.g.*, Docker) to obtain iterative execution feedback. To overcome the prohibitive computational overhead of physical execution (Shum et al., 2025), recent studies explore *Code World Models* to internalize environment dynamics. For instance, CWM (Copet et al., 2025) trains a general-purpose neural code interpreter, while SWE-World (Sun et al., 2026) and SWE-RM (Shum et al., 2025) directly simulate execution outputs and test rewards, enabling Docker-free agent training.

Despite the rapid development of code world models, existing execution-understanding benchmarks (*e.g.*, CRUXEval (Gu et al., 2024), KernelBench (Ouyang et al., 2025)) remain restricted to static, function-level prediction within isolated environments. They systematically fail to capture the immense state complexity, strict partial observability, and cumulative state evolution inherent to real software repositories. SWE-ROAM addresses this by introducing the first diagnostic benchmark for **repository-level** world modeling. By decoupling step-level simulation from trajectory-level auditing, and contrasting passive with active state recovery, it isolates current model bottlenecks and exposes the critical offline–online gap in agent deployment.

6 Conclusion

In this paper, we introduce SWE-ROAM, a diagnostic benchmark designed to systematically evaluate the repository-level world-modeling capabilities of LLMs. By decoupling step-level environment simulation from trajectory-level outcome auditing, we uncover a fundamental dissociation in current frontier models: strong local execution prediction does not ensure reliable global outcome judgment. Furthermore, our contrast between passive reasoning and active state recovery demonstrates that software world modeling is heavily bottlenecked by partial observability. Models must actively retrieve repository context rather than passively reason over static transcripts. Finally, our online deployment evaluation reveals a critical offline–online gap, where high single-step accuracy masks catastrophic cascading errors during multi-step agent interaction.

Limitations

While SWE-ROAM provides a systematic evaluation of repository-level world modeling, our source trajectories are currently restricted to Python due to our reliance on SWE-bench. Evaluating world models on compiled languages (*e.g.*, C++, Java) introduces distinct simulation challenges, such as predicting complex build traces and linking failures, which warrants future investigation. Moreover, as a diagnostic benchmark, SWE-ROAM successfully identifies and quantifies the *sim-to-real* gap caused by cascading hallucinations, but developing concrete solutions, such as uncertainty-aware decoding or hybrid real/model execution strategies to close this gap, remains a critical open challenge for the community.

References

Anthropic. 2026. Introducing Claude Opus 4.6. https://www.anthropic.com/news/claude-opus-4-6. Published February 5, 2026; accessed May 25, 2026.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. 2026. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *Advances in Neural Information Processing Systems*, 38.

Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing, Kaixin Li, Mingze Li, Junyang Lin, and 1 others. 2026. Qwen3-coder-next technical report. *arXiv preprint arXiv:2603.00729*.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. 2024. Introducing SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/. Updated February 24, 2025. Accessed May 25, 2026.

Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, and 1 others. 2025. Cwm: An open-weights llm for research on code generation with world models. *arXiv preprint arXiv:2510.02387*.

DeepSeek-AI. 2025. DeepSeek-V3.1 Release. https://api-docs.deepseek.com/news/news250821. Published August 21, 2025; accessed May 25, 2026.

Jingtao Ding, Yunke Zhang, Yu Shang, Yuheng Zhang, Zefang Zong, Jie Feng, Yuan Yuan, Hongyuan Su, Nian Li, Nicholas Sukiennik, and 1 others. 2025. Understanding world or predicting future? a comprehensive survey of world models. *ACM Computing Surveys*, 58(3):1–38.

Mohamed Amine Ferrag, Norbert Tihanyi, and Merouane Debbah. 2025. From llm reasoning to autonomous ai agents: A comprehensive review. *arXiv preprint arXiv:2504.19678*.

Google DeepMind. 2025. Gemini 3 Flash Model Card. https://deepmind.google/models/model-cards/gemini-3-flash/. Published December 17, 2025; accessed May 25, 2026.

Google DeepMind. 2026. Gemini 3.1 Pro Model Card. https://deepmind.google/models/model-cards/gemini-3-1-pro/. Published February 19, 2026; accessed May 25, 2026.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*.

David Ha and Jürgen Schmidhuber. 2018. World models. *arXiv preprint arXiv:1803.10122*, 2(3):440.

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

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157.

Abhishek Kadian, Joanne Truong, Aaron Gokaslan, Alexander Clegg, Erik Wijmans, Stefan Lee, Manolis Savva, Sonia Chernova, and Dhruv Batra. 2020. Sim2real predictivity: Does evaluation in simulation predict real-world performance? *IEEE Robotics and Automation Letters*, 5(4):6670–6677.

Michael Luo. Deepswe: Training a fully open-sourced, state-of-the-art coding agent by scaling rl, jul 2025. *URL https://www. together. ai/blog/deepswe*.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori B Hashimoto. 2025. s1: Simple test-time scaling. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 20286–20332.

OpenAI. 2026. Introducing GPT-5.4. https://openai.com/index/introducing-gpt-5-4/. Published March 5, 2026; accessed May 25, 2026.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2024. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139*.

KaShun Shum, Binyuan Hui, Jiawei Chen, Lei Zhang, Jiaxi Yang, Yuzhen Huang, Junyang Lin, Junxian He, and 1 others. 2025. Swe-rm: Execution-free feedback for software engineering agents. *arXiv preprint arXiv:2512.21919*.

Shuang Sun, Huatong Song, Lisheng Huang, Jinhao Jiang, Ran Le, Zhihao Lv, Zongchao Chen, Yiwen Hu, Wenyang Luo, Wayne Xin Zhao, and 1 others. 2026. Swe-world: Building software engineering agents in docker-free environments. *arXiv preprint arXiv:2602.03419*.

Chaofan Tao, Jierun Chen, Yuxin Jiang, Kaiqi Kou, Shaowei Wang, Ruoyu Wang, Xiaohui Li, Sidi Yang, Yiming Du, Jianbo Dai, and 1 others. 2026. Swe-lego: Pushing the limits of supervised fine-tuning for software issue resolving. *arXiv preprint arXiv:2601.01426*.

Andrew Wagenmaker, Kevin Huang, Liyiming Ke, Kevin Jamieson, and Abhishek Gupta. 2024. Overcoming the sim-to-real gap: Leveraging simulation to learn to explore for real-world rl. *Advances in Neural Information Processing Systems*, 37:78715–78765.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2025. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pages 65882–65919.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*.

Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. 2025. Live-swe-agent: Can software engineering agents self-evolve on the fly? *arXiv preprint arXiv:2511.13646*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025a. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2026. Swe-smith: Scaling data for software engineering agents. *Advances in Neural Information Processing Systems*, 38.

Zonghan Yang, Shengjie Wang, Kelin Fu, Wenyang He, Weimin Xiong, Yibo Liu, Yibo Miao, Bofei Gao, Yejie Wang, Yingwei Ma, and 1 others. 2025b. Kimi-dev: Agentless training as skill prior for swe-agents. *arXiv preprint arXiv:2509.23045*.

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

Liang Zeng, Yongcong Li, Yuzhen Xiao, Changshi Li, Chris Yuhao Liu, Rui Yan, Tianwen Wei, Jujie He, Xuchen Song, Yang Liu, and 1 others. 2025. Skywork-swe: Unveiling data scaling laws for software engineering in llms. *arXiv preprint arXiv:2506.19290*.

A More Implementation Details

A.1 Evaluation Metrics

Transition Metrics. Given the structural diversity of observations, we define category-specific primary metrics $\mu_c(\hat{o}, o)$ and correctness thresholds θ_c. For all transition metrics, we normalize outputs by removing tool-wrapper boilerplate and repetitive whitespace. Let $W(\cdot)$ denote sets of normalized words. Let $F(\cdot)$ denote file paths extracted from search outputs, and $E(\cdot)$ denote filesystem entities extracted from exploration outputs. A prediction is correct if $\mu_c(\hat{o}, o) \geq \theta_c$.

- **VIEW (Content Overlap):** $\mu_{\text{view}} = \frac{|W(o) \cap W(\hat{o})|}{|W(o)|}$. Threshold $\theta_{\text{view}} = 0.7$.
- **SEARCH (File Recall):** $\mu_{\text{search}} = \frac{|F(o) \cap F(\hat{o})|}{|F(o)|}$. Threshold $\theta_{\text{search}} = 0.5$.
- **EXPLORE (Set Precision):** $\mu_{\text{explore}} = \frac{|E(o) \cap E(\hat{o})|}{|E(\hat{o})|}$. Threshold $\theta_{\text{explore}} = 0.5$.
- **EDIT (Success Match):** A binary check ($\mu_{\text{edit}} \in \{0, 1\}$) verifying if the predicted string reflects the applied/failed state via enhanced keyword extraction (e.g., extracting success from `edited`, `finished`, `done`, or extracting failure from specific missing-target warnings). Threshold $\theta_{\text{edit}} = 1.0$.
- **TEST (Status Match):** A strict binary check ($\mu_{\text{test}} \in \{0, 1\}$) requiring a match of the test's failure/success status derived from explicit exit codes or fallback indicators (e.g., `Traceback`, `Exception`, `FAILED`). Threshold $\theta_{\text{test}} = 1.0$.

For set-based metrics, if the denominator set is empty, we assign a score of 1 if and only if the predicted set is also empty, and 0 otherwise.

We report Overall Transition Accuracy, denoted as **Overall Acc** in our tables, as the micro-average of thresholded correctness over all 1,000 transition samples.

Reward Metrics. For Reward Prediction, the model generates a binary label $\hat{r}$ and a continuous confidence score $\hat{p}$. For the continuous score, we prompt models to output a confidence rating for SUCCESS. If the returned confidence is missing or invalid, we fall back to a deterministic score derived from the predicted label. We apply the same parsing rule to all models to ensure fairness. We report binary **Accuracy** ($\hat{r} = r$) and **AUC-ROC** (derived from $\hat{p}$ and r) to evaluate trajectory ranking capability.

A.2 Decoding Setup

All models use identical task prompts and output schemas. In Agent mode, models invoke tools through native or compatible function calling. For each example, we decode a single prediction with a greedy temperature of 0.0. Transition predictions are evaluated directly as free-form observations, while Reward predictions are parsed into the structured label and confidence score.

A.3 Main Benchmark Protocol

The main benchmark evaluation consists of two tasks. For Transition Prediction, each example is a tuple $(d, h_{<t}, a_t, o_t, c_t)$, where d is the issue description, $h_{<t}$ is the action-observation history, a_t is the target action, o_t is the ground-truth observation, and c_t is the action category. We evaluate 1,000 sampled examples across five categories: VIEW, SEARCH, EDIT, TEST, and EXPLORE. Each model is evaluated under both Non-Agent and Agent modes. The reported Overall Acc is not raw string exact match; it is the micro-average of thresholded correctness after applying the category-specific metric for each sample.

For Reward Prediction, each example is a sanitized complete trajectory $(d, \bar{\tau}, r)$, where r is the binary SWE-bench oracle label. We evaluate 800 trajectories under both modes. Models output a binary label $\hat{r}$ and a confidence score $\hat{p}$ for SUCCESS; we report Accuracy and AUC-ROC. If the returned confidence is missing or invalid, we use the same deterministic fallback rule for all models based on the predicted label.

In Agent mode, the model may use up to $B = 12$ read-only tool calls before submitting its prediction unless otherwise specified. The available tools are `view`, `search`, and `execute_bash`. The sandbox exposes the initial repository state s_0, not the hidden current state s_t induced by the trajectory prefix, so the model must combine retrieved repository evidence with the edit history. Destructive commands, workspace modifications, network access, and direct execution of the target action for answer extraction are blocked. In Non-Agent mode, the model receives only the task input and cannot inspect the repository.

A.4 Inference-Time Scaling

We perform inference-time scaling experiments on Transition Prediction using Qwen3-30B-Coder. For active exploration scaling, we vary the Agent-mode budget:

$$B \in \{1, 3, 6, 12, 20, 30\}.$$

This measures how additional repository exploration affects state recovery and transition prediction. For passive reasoning scaling, we vary the Non-Agent prompt:

`default`, `cot_light`, `cot_heavy`.

This tests whether increasing chain-of-thought depth can substitute for external repository observation. Both scaling experiments use the same 1,000 transition examples.

A.5 MODELWORLDENV Deployment

MODELWORLDENV evaluates whether an offline transition predictor can be used as an online environment surrogate during SWE-agent execution. We run a fixed Qwen3-30B-Coder SWE agent on 100 SWE-bench Verified instances. The fully real Docker environment is used as the baseline. In all settings, the final submitted patch is evaluated by the physical SWE-bench oracle; model predictions only replace intermediate feedback during the rollout.

Let $\mathcal{A}_{\text{obs}}$ denote observation-producing non-write actions, including file viewing, repository search, directory exploration, and command execution whose feedback is returned to the agent. Let $\mathcal{A}_{\text{write}}$ denote workspace-modifying edit actions such as file creation, insertion, or replacement. Write actions always execute in the real Docker environment to preserve patch validity. For actions in $\mathcal{A}_{\text{obs}}$, the observation may be intercepted and simulated by a learned Transition Model $\widehat{\mathcal{E}}_\phi$ with probability p.

During online rollout, the agent observes a mixed history

$$\tilde{h}_{<t} = ((a_1, \tilde{o}_1), \ldots, (a_{t-1}, \tilde{o}_{t-1})),$$

where previous observations may be real or model-generated. The feedback given to the agent is:

$$\tilde{o}_t = \begin{cases} \widehat{\mathcal{E}}_\phi(a_t, \tilde{h}_{<t}, e^{\phi}_{1:K}), & a_t \in \mathcal{A}_{\text{obs}}, \text{ w.p. } p, \\ \mathcal{E}_{\mathcal{I}}(a_t, s_t), & \text{otherwise.} \end{cases} \tag{9}$$

Here, $e^{\phi}_{1:K}$ denotes the exploration transcript gathered by the Transition Model itself before predicting the replaced observation. When $K = 0$, the Transition Model predicts passively without additional exploration.

We study three dimensions:

- **Replacement probability.** We vary

  $$p \in \{0.00, 0.25, 0.50, 0.75, 1.00\},$$

  with no real-environment warm-up and a non-agent transition model.

- **Real-environment warm-up.** We fix $p = 1.0$ and vary

  $$\texttt{prefix_steps} \in \{0, 3, 6, 10\},$$

  the number of initial physical steps executed before switching to model-predicted feedback.

- **TransitionAgent exploration.** We fix $p = 0.5$ and $\texttt{prefix_steps} = 0$, and vary

  $$\texttt{tmsteps} \in \{0, 3, 6, 12, 20, 30\}.$$

  The setting $\texttt{tmsteps} = 0$ corresponds to the non-agent transition model.

B More Findings

B.1 The Context Pollution Tipping Point

Finding 6. *Active exploration exhibits a "context pollution" tipping point. Unbounded tool access introduces redundant context that actively degrades the world model's prediction accuracy.*

If active state recovery is critical, it is tempting to grant models massive exploration budgets. However, Table 3 and Figure 3 show that scaling exploration is not monotonically beneficial. While initial increases in the exploration budget yield dramatic improvements in transition accuracy, performance eventually peaks and subsequently degrades before plateauing.

This non-monotonicity reveals that models struggle to filter signal from noise. Once the necessary state is recovered, further exploration pulls irrelevant files and search results into the context window, degrading the model's predictive focus. An effective code world model must therefore pair active exploration with a highly calibrated stopping policy.

C Prompt Templates

We provide the prompt templates used in SWE-ROAM. For readability, we omit few-shot demonstrations and show only the task-level instructions, input placeholders, and output format.

C.1 Transition Prediction Prompts

Transition Prediction (Non-Agent Mode):

```
You are a transition prediction expert. Given the history
      and a target tool call,
predict the exact observation that the software
      environment would return.

GUIDELINES:
1. Be realistic. Do not assume that every command succeeds.
       If the target action
   runs a test or script, consider possible Traceback,
      SyntaxError, AssertionError,
   failed tests, or nonzero exit codes.
2. Match the environment format. Your prediction should
      follow the real tool-output
   format, including execution headers, exit codes, STDOUT
      , and STDERR when applicable.
3. Track repository state. Mentally apply all previous
      edit actions in the history
   before predicting the target observation.
4. For file views, preserve file content as accurately as
      possible, including line
   numbers or line ranges when the environment would
      include them.
5. Do not output explanations outside the required fields.

INPUT:
[Problem Statement]
{problem_statement}

[History]
{trajectory_history}

[Target Action]
{target_action}

OUTPUT FORMAT:
<thought>
Briefly reason about the current repository state,
      relevant prior edits, and
the expected behavior of the target action.
</thought>
<output>
The exact predicted observation returned by the
      environment.
</output>
```

Transition Prediction (Agent Mode):

```
ROLE:
You are a world-model predictor for a software engineering
       environment.

GOAL:
Predict the exact observation that the Target Action will
      produce at its
specific point in the trajectory.

AVAILABLE TOOLS:
- view: inspect file contents or list directory contents.
- search: search for terms or symbols in the repository.
- execute_bash: run read-only inspection commands.

CRITICAL CONSTRAINTS:
1. You are in a strictly read-only sandbox.
2. You cannot modify files.
3. Destructive commands, workspace modifications, and
      network access are blocked.
4. The tools expose the initial repository state, not
      necessarily the current
   trajectory state.
5. You must not use tools to directly execute the target
      action for answer extraction.

STATE SYNTHESIS:
- Tool outputs reveal the initial repository state.
- The Target Action occurs after the History.
- If the History contains edits, the current state differs
       from the initial state.
- Combine repository evidence from tools with the edits in
       History.
- Predict the Target Action output under the synthesized
      current state.

PROCESS:
1. Inspect the problem, history, and target action.
2. Use tools to recover relevant repository evidence.
3. Mentally apply prior edits from the history.
4. Predict the environment observation.
5. Call submit with the final predicted observation.

INPUT:
[Problem Statement]
{problem_statement}

[History]
{trajectory_history}

[Target Action]
{target_action}

FINAL ANSWER:
Use the submit tool. The prediction field must contain
      only the predicted
environment observation.
```

C.2 Reward Prediction Prompts

Reward Prediction (Non-Agent Mode):

```
You are an expert software engineering evaluator. Your
      task is to determine
whether the given repair trajectory successfully solved
      the problem described
in the problem statement.

Do not rely only on superficial final signals. Analyze the
       full sequence of
actions and observations to decide whether the final patch
       likely resolves the
underlying issue according to the benchmark oracle.

EVALUATION CRITERIA:
1. Identify the core issue described in the problem
      statement.
2. Check whether the agent localized the relevant code.
3. Check whether the edits address the root cause rather
      than only a narrow case.
4. Check whether the agent performed meaningful validation.

5. Watch for silent failures, import errors, failing tests
      , tracebacks, or incomplete fixes.
6. Decide whether the final repository state should pass
      the hidden tests.

INPUT:
[Problem Statement]
{problem_statement}

[Sanitized Trajectory]
{sanitized_trajectory}

OUTPUT FORMAT:
<thought>
Briefly audit the trajectory. Explain whether the edit is
      logically sufficient
and whether the validation evidence is reliable.
</thought>
<label>
SUCCESS or FAILURE
```

```
</label>
<confidence>
A number from 0 to 1 indicating confidence that the
      trajectory is SUCCESS.
</confidence>
```

Reward Prediction (Agent Mode):

```
ROLE:
You are a skeptical software engineering auditor.

GOAL:
Determine whether the provided repair trajectory
      successfully resolves the issue.
You should look for reasons the trajectory might fail.

AVAILABLE TOOLS:
- view: inspect baseline file contents or list directories.

- search: search the baseline repository.
- execute_bash: run read-only inspection commands.

IMPORTANT CONTEXT:
1. The tools show the initial repository state before the
      trajectory edits.
2. The trajectory may contain edits that modify the
      repository.
3. You must mentally apply the trajectory edits to the
      baseline code when auditing.
4. The final submit action and internal agent thoughts
      have been removed.
5. The final observation may be truncated, so do not rely
      only on the last output.

AUDITOR CHECKLIST:
1. What is the core bug or requested behavior?
2. Did the agent inspect the relevant implementation?
3. Did the agent make a plausible and complete fix?
4. Did the agent run meaningful tests that actually
      validate the issue?
5. Are there signs of silent failure, incomplete testing,
      or unrelated changes?
6. Would the final patch likely pass the hidden benchmark
      tests?

PROCESS:
1. Read the problem and sanitized trajectory.
2. Use tools to inspect relevant baseline repository
      context.
3. Mentally merge the trajectory edits into the baseline.
4. Audit the final patch and validation evidence.
5. Call submit with SUCCESS or FAILURE and a confidence
      score.

INPUT:
[Problem Statement]
{problem_statement}

[Sanitized Trajectory]
{sanitized_trajectory}

FINAL ANSWER:
Use the submit tool with:
- label: SUCCESS or FAILURE
- confidence: a number from 0 to 1 indicating probability
      of SUCCESS
- rationale: a concise explanation of the audit decision
```

C.3 MODELWORLDENV Transition Model Prompt

MODELWORLDENV (Transition Model Prediction):

```
You are a transition model used inside an online software
      engineering agent loop.

Your task is to predict the observation that the real
      environment would return
for the current agent action.

CONTEXT:
The agent is interacting with a repository environment.
      Some previous observations
may be real environment outputs, while others may be model
      -generated. The agent
will use your predicted observation to decide its next
      action, so errors may affect
the rest of the rollout.

GOAL:
Predict the environment observation for the Current Action.


GUIDELINES:
1. Use the mixed interaction history as the agent's
      current belief state.
2. Account for prior edits and created files when
      predicting the current observation.
3. If the action is a file view or search, predict the
      relevant repository content
   or matching paths.
4. If the action executes code or tests, predict the most
      likely exit status,
   STDOUT, and STDERR.
5. Preserve the expected environment format as closely as
      possible.
6. Do not explain uncertainty outside the required output
      field.

INPUT:
[Problem Statement]
{problem_statement}

[Mixed History]
{mixed_history}

[Current Action]
{current_action}

[Optional Exploration Transcript]
{transition_model_exploration}

OUTPUT FORMAT:
<output>
The predicted observation returned by the environment.
</output>
```

D Case Studies

We present concrete input-output examples from SWE-ROAM. For readability, long histories and observations are abbreviated with ellipses. The examples illustrate how SWE-ROAM evaluates both step-level environment prediction and trajectory-level outcome auditing.

D.1 Case Study 1: Transition Prediction

This example shows a VIEW transition. The agent previously created a new reproduction script in the trajectory. The target action asks to view that file. Therefore, the model must predict the file content under the trajectory-induced state, rather than relying only on the initial repository.

Input:

```
Task: Transition Prediction
Category: VIEW
```

```
[Problem Statement]
Django admin.E108 is raised on fields accessible only via
      instance.
The issue concerns the logic in _check_list_display_item.

[History]
Step 14 Action:
str_replace_editor view /testbed/django/contrib/admin/
      checks.py --view_range [718,750]

Step 14 Observation:
Execution output of [str_replace_editor]:
Here's the result of running cat -n on /testbed/django/
      contrib/admin/checks.py:
   718    def _check_list_display_item(self, obj, item,
      label):
   719        ...
   750        ...

Step 15 Action:
str_replace_editor create /testbed/simple_test.py
<file_text>
#!/usr/bin/env python
"""Simple test to demonstrate the admin.E108 behavior."""
import os
import django
...
</file_text>

Step 15 Observation:
Execution output of [str_replace_editor]:
File created at /testbed/simple_test.py.
Here's the result of running cat -n on /testbed/
      simple_test.py:
     1  #!/usr/bin/env python
     2  """Simple test to demonstrate the admin.E108
      behavior."""
     3  import os
     4  import django
     ...

[Target Action]
str_replace_editor view /testbed/simple_test.py
```

Ground-truth Output:

```
Execution output of [str_replace_editor]:
Here's the result of running cat -n on /testbed/
      simple_test.py:
     1  #!/usr/bin/env python
     2  """Simple test to demonstrate the admin.E108
      behavior."""
     3  import os
     4  import django
     5
     6  os.environ.setdefault("DJANGO_SETTINGS_MODULE", "
      settings")
     7  django.setup()
     8
     9  from django.contrib import admin
    10  from django.core.checks import Error
    ...
```

Evaluation. For this VIEW example, the predicted observation is evaluated with content overlap:

$$\mu_{\text{VIEW}}(\hat{o}, o) = \frac{|W(\hat{o}) \cap W(o)|}{|W(o)|}.$$

The prediction is counted as correct if the overlap is at least 0.7.

D.2 Case Study 2: Transition Prediction with Execution

This example shows a TEST transition. Unlike file viewing, the model must predict whether a script succeeds or fails after earlier trajectory edits.

Input:

```
Task: Transition Prediction
Category: TEST

[Problem Statement]
Django admin.E108 is incorrectly raised for a field that
      is accessible
only through an instance.

[History]
Step 13 Action:
execute_bash python test_function_logic.py

Step 13 Observation:
Exit code: 1
[STDERR]
Traceback (most recent call last):
  ...
AssertionError: Expected no admin.E108 errors

Step 14 Action:
str_replace_editor view /testbed/django/contrib/admin/
      checks.py --view_range [718,750]

Step 14 Observation:
Execution output of [str_replace_editor]:
Here's the result of running cat -n on /testbed/django/
      contrib/admin/checks.py:
   718    def _check_list_display_item(self, obj, item,
      label):
   ...

Step 15 Action:
str_replace_editor create /testbed/simple_test.py
<file_text>
#!/usr/bin/env python
"""Simple test for the corrected admin check logic."""
...
</file_text>

Step 15 Observation:
Execution output of [str_replace_editor]:
File created at /testbed/simple_test.py.
...

[Target Action]
execute_bash python /testbed/simple_test.py
```

Ground-truth Output:

```
Exit code: 0
[STDOUT]
=== Testing Current Logic ===
Testing 'number' field...
Result: 0 errors

Testing callable field...
Result: 0 errors

Testing missing field...
Result: 1 errors

All checks completed successfully.
[STDERR]
```

Evaluation. For this TEST example, the prediction is evaluated by status match. The model does not need to reproduce every line exactly, but it must

correctly predict the execution success or failure status, including the exit-code semantics. The prediction is correct only if:

$$\mu_{\text{TEST}}(\hat{o}, o) = 1.$$

D.3 Case Study 3: Reward Prediction

This example shows a trajectory-level reward prediction task. The model receives a sanitized repair trajectory with internal thoughts removed and the final submit action deleted. It must decide whether the final patch resolves the issue.

Input:

```
Task: Reward Prediction

[Problem Statement]
Django admin.E108 is incorrectly raised on a PositionField
       that is
accessible through an instance. The admin system check
      should not report
an error for this valid list_display item.

[Sanitized Trajectory]
Step 1 Action:
str_replace_editor view /testbed/django/contrib/admin/
      checks.py

Step 1 Observation:
Execution output of [str_replace_editor]:
Here's the result of running cat -n on /testbed/django/
      contrib/admin/checks.py:
   ...
   def _check_list_display_item(...):
       ...

Step 5 Action:
execute_bash python reproduce_issue.py

Step 5 Observation:
Exit code: 1
[STDOUT]
Reproducing admin.E108 issue...
Got admin.E108 unexpectedly.
[STDERR]
Traceback ...

Step 12 Action:
str_replace_editor str_replace /testbed/django/contrib/
      admin/checks.py
<old_str>
...
</old_str>
<new_str>
...
</new_str>

Step 12 Observation:
Execution output of [str_replace_editor]:
The file /testbed/django/contrib/admin/checks.py has been
      edited.

Step 16 Action:
execute_bash python simple_test.py

Step 16 Observation:
Exit code: 0
[STDOUT]
Testing current logic...
Result: 0 errors
All tests passed.
[STDERR]

Step 17 Action:
execute_bash python manage.py check

Step 17 Observation:
Exit code: 0
[STDOUT]
System check identified no issues (0 silenced).
[STDERR]

Note:
The final submit action is removed.
Internal agent thoughts are masked.
The final observation is truncated if longer than 1000
      characters.
```

Ground-truth Output:

```
Label: SUCCESS
Confidence target: 1.0

The trajectory successfully reproduces the issue, applies
      a patch to the
relevant admin check logic, and validates the fix with
      both a targeted
script and the Django system check. The SWE-bench oracle
      labels the final
submitted patch as successful.
```

Evaluation. For Reward Prediction, the model outputs a binary label:

$$\hat{r} \in \{0, 1\}.$$

The prediction is evaluated against the SWE-bench oracle label r. We report Accuracy and AUC-ROC:

$$\text{Acc}_{\text{reward}} = \frac{1}{|\mathcal{D}_{\text{reward}}|} \sum_{(x,r)\in\mathcal{D}_{\text{reward}}} \mathbf{1}[\hat{r} = r].$$

E World Model Prediction Case Studies

We present concrete examples of model predictions on SWE-ROAM. Each example shows the target action, the ground-truth observation returned by the real environment, and the world model's predicted observation. For compactness, we truncate irrelevant whitespace and long outputs.

E.1 Active State Recovery Corrects Repository Search

This example compares Agent and Non-Agent predictions on the same SEARCH instance. The target action searches a Django compiler file for expressions related to compilation. The Agent-mode model recovers the exact matching line, while the Non-Agent model incorrectly predicts that no match exists.

Input: Target Action.

```
Instance: django__django-15382
Category: SEARCH

<function=execute_bash>
  <parameter=command>
grep -n "compile.*expression\|compile.*expr" /testbed/
      django/db/models/sql/compiler.py
  </parameter>
```

```
</function>
```

Ground Truth:

```
Exit code: 0
Execution output of [execute_bash]:
[STDOUT]

153:            sql, params = self.compile(expr)

[STDERR]
```

Agent Prediction: Correct.

```
Exit code: 0
Execution output of [execute_bash]:
[STDOUT]

153:            sql, params = self.compile(expr)

[STDERR]
```

Non-Agent Prediction: Incorrect.

```
Exit code: 1
Execution output of [execute_bash]:
[STDOUT]

[STDERR]
```

Outcome. Agent mode achieves file recall $= 1.0$, while Non-Agent mode obtains file recall $= 0.0$. This example illustrates why repository search is observability-limited: the correct answer depends on an exact repository line that is difficult to infer from the trajectory alone.

E.2 Active Exploration Recovers Filesystem State

This example shows an EXPLORE action that asks for the line count of a SymPy file. The Agent-mode model predicts the exact filesystem-derived output, while the Non-Agent model hallucinates a stale or incorrect line count.

Input: Target Action.

```
Instance: sympy__sympy-15976
Category: EXPLORE

<function=execute_bash>
  <parameter=command>
cd /testbed && wc -l sympy/printing/mathml.py
  </parameter>
</function>
```

Ground Truth:

```
Exit code: 0
Execution output of [execute_bash]:
[STDOUT]

969 sympy/printing/mathml.py

[STDERR]
```

Agent Prediction: Correct.

```
Exit code: 0
Execution output of [execute_bash]:
[STDOUT]

969 sympy/printing/mathml.py

[STDERR]
```

Non-Agent Prediction: Incorrect.

```
Exit code: 0
Execution output of [execute_bash]:
[STDOUT]
1073 /testbed/sympy/printing/mathml.py
```

Outcome. Agent mode obtains set precision $= 1.0$, while Non-Agent mode obtains set precision $= 0.0$. The failure is not caused by complex execution dynamics; the model simply lacks the observed repository state needed to predict the structural output.

E.3 Transition Prediction after Execution-Sensitive Edits

This example shows a TEST transition. The target action runs a focused SymPy test command. The Agent-mode prediction matches the successful execution status, while the Non-Agent model predicts a traceback and assertion failure.

Input: Target Action.

```
Instance: sympy__sympy-15976
Category: TEST

<function=execute_bash>
  <parameter=command>
cd /testbed && python -c "from sympy.printing.tests.
      test_conventions import test_super_sub;
      test_super_sub(); print('All tests passed')"
  </parameter>
</function>
```

Ground Truth:

```
Exit code: 0
Execution output of [execute_bash]:
[STDOUT]

All tests passed

[STDERR]
```

Agent Prediction: Correct.

```
Exit code: 0
Execution output of [execute_bash]:
[STDOUT]

All tests passed
```

Non-Agent Prediction: Incorrect.

```
Exit code: 1
Execution output of [execute_bash]:
[STDOUT]
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "/testbed/sympy/printing/tests/test_conventions.py
      ", line 30, in test_super_sub
    assert split_super_sub('x2') == ('x', [], ['2'])  #
      Issue #15594
AssertionError

[STDERR]
```

Outcome. Agent mode achieves status match $= 1.0$, while Non-Agent mode obtains status match $= 0.0$. This example shows that active repository access can sometimes help execution-related prediction when the relevant state can be recovered from the repository and trajectory.

E.4 Execution Simulation Failure under Environment Mismatch

This example shows a typical failure mode for TEST: even with a plausible command, the true environment may fail for infrastructure reasons. Here, the real environment cannot import `pytest`, but the model predicts a normal zero-exit test outcome.

Input: Target Action.

```
Instance: django__django-16560
Category: TEST

<function=execute_bash>
  <parameter=command>
cd /testbed && python -m pytest tests/validation/tests.py
      -v -k "constraint" --tb=short
  </parameter>
</function>
```

Ground Truth:

```
Exit code: Error: Exit code 1
Execution output of [execute_bash]:
Error executing command:

[STDOUT]

[STDERR]

/root/.venv/bin/python: No module named pytest
```

Agent Prediction: Incorrect.

```
Exit code: 0
Execution output of [execute_bash]:
[STDOUT]

Found 0 test(s).
No tests matched the filter.
```

Outcome: The prediction receives status match $= 0.0$. This illustrates a limitation of active state recovery: inspecting repository files does not necessarily reveal runtime environment configuration, installed dependencies, or command-execution failures.

E.5 Edit Success Requires Current-State Tracking

This example shows an EDIT prediction failure. The target action attempts a string replacement in a Django SQLite backend file. The ground truth says the replacement fails because the specified old string is not found. Both Agent and Non-Agent predictions incorrectly assume that the edit succeeds.

Input: Target Action.

```
Instance: django__django-13821
Category: EDIT

<function=str_replace_editor>
  <parameter=path>/testbed/django/db/backends/sqlite3/base.
      py</parameter>
  <parameter=command>str_replace</parameter>
  <parameter=old_str>
def check_sqlite_version():
    if Database.sqlite_version_info < (3, 8, 3):
        raise ImproperlyConfigured('SQLite 3.8.3 or later
      is required (found %s).' % Database.sqlite_version)

check_sqlite_version()
  </parameter>
  <parameter=new_str>
def check_sqlite_version():
    if Database.sqlite_version_info < (3, 9, 0):
        raise ImproperlyConfigured('SQLite 3.9.0 or later
      is required (found %s).' % Database.sqlite_version)

check_sqlite_version()
  </parameter>
</function>
```

Ground Truth:

```
Execution output of [str_replace_editor]:
ERROR: No occurrences of 'def check_sqlite_version():
    if Database.sqlite_version_info < (3, 8, 3):
        raise ImproperlyConfigured('SQLite 3.8.3 or later
      is required (found %s).' % Database.sqlite_version)

check_sqlite_version()' found in /testbed/django/db/
      backends/sqlite3/base.py for replacement.
```

Agent Prediction: Incorrect.

```
The file /testbed/django/db/backends/sqlite3/base.py has
      been edited.
Here's the result of running `cat -n` on a snippet of /
      testbed/django/db/backends/sqlite3/base.py:
    66 def check_sqlite_version():
    67     if Database.sqlite_version_info < (3, 9, 0):
    68         raise ImproperlyConfigured('SQLite 3.9.0 or
       later is required (found %s).' % Database.
      sqlite_version)
    69
    70
    71 check_sqlite_version()
```

Outcome. The prediction receives edit success match $= 0.0$. This failure shows that even when the intended modification is semantically clear, transition prediction requires precise current-state tracking: the old string may no longer exist exactly as specified.