WORLDSIMBENCH: BENCHMARKING LARGE LANGUAGE MODELS AS ENVIRONMENT SIMULATORS

Anonymous authors
Paper under double-blind review

ABSTRACT

Training agents at scale requires large volumes of interaction trajectories, which are costly to collect from real environments. LLM-based generative simulators offer a scalable alternative, but existing public evaluations are often limited to narrow interaction settings and short trajectories, or rely on open-ended LLM-based scoring. We introduce **WorldSimBench**, a benchmark for evaluating LLMs as environment simulators across diverse tasks and long-horizon interactions. It comprises 3,580 tasks from 322 environments across ToolUse, Terminal, Game, Web, and Search. To construct long-horizon trajectories with fixed reference outcomes, we use story-driven plans to organize interactions and executable environments to produce the resulting states and observations. ToolUse, Terminal, Game, and Search use deterministic code-based or rule-based verifiers, whereas Web uses a fixed LLM verifier to check predefined requirements for components and values. Human assessment of representative tasks across all five environments yields an average rating of 9.15 out of 10. Across 32 model, the best achieves an unweighted macro-average success rate of 81.39%. Reasoning configurations achieve higher macro-average success rates in all 13 paired comparisons under the reported mode-specific inference settings. We find that world model performance is positively correlated with agent performance. Context truncation experiments on two models further show that early history is important for ToolUse, Terminal, and Search, whereas retaining only recent interactions improves accuracy over full histories in Game. These findings suggest that reliable simulation depends on preserving task-relevant state and limiting interference from irrelevant history. Code and sample data are available at `https://anonymous.4open.science/r/worldsimbench/`.

1 INTRODUCTION

Training capable agents requires large-scale interaction trajectories across diverse tasks and environments, yet collecting such data from real environments is prohibitively expensive and difficult to scale. LLM-based generative simulators offer a scalable alternative, synthesizing interaction trajectories at scale and reducing the need for costly real-environment data collection (Wang et al., 2025b; Chen et al., 2025b; Li et al., 2025). As these simulators are increasingly adopted as data sources, a question arises: **how reliably can LLMs simulate environment dynamics, and how do different models compare as world models?**

Evaluating LLM-based simulators remains an open problem. One line of work evaluates simulators indirectly through downstream agent performance (Wang et al., 2025b; Chen et al., 2025b; Li et al., 2025; Wang et al., 2026; Xiao et al., 2026; Bai et al., 2026; Ding et al., 2026;

Table 1: Comparison of benchmarks and evaluation protocols for LLM-based environment simulators. **Deter.** denotes whether the environment dynamics or reference trajectories are deterministic. **Steps** denotes the reported trajectory horizon; decimal values indicate average trajectory lengths reported by prior work. **Multi-Env** indicates whether the method evaluates multiple environment instances within a domain. **Multi-Dom** indicates whether it spans multiple domains. **Eval.** denotes the primary evaluation signal.

Method	Deter.	Steps	Multi-Env	Multi-Dom	Eval.
BYTESIZED32-SP (Wang et al., 2024)	✓	1	✗	✗	Rules
DreamGym (Chen et al., 2025b)	✗	10	✓	✓	Agent Perf.
RPGBENCH (Yu et al., 2025)	✓	10	✗	✗	Rules
UI-SIMULATOR (Wang et al., 2025b)	✗	5	✓	✗	Human
WebEvolver (Fang et al., 2025)	✗	7	✗	✓	LLM Judge
AWM (Wang et al., 2026)	✗	8.5	✗	✓	LLM Judge
Code2World (Zheng et al., 2026)	✗	1	✗	✓	VLM Judge
EnvScaler (Song et al., 2026)	✓	25	✗	✗	LLM Judge
EnvSimBench (Liu et al., 2026)	✓	1	✓	✗	Rules
WebGym (Bai et al., 2026)	✗	13.2	✗	✓	Agent Perf.
WorldSimBench (Ours)	✓	**199**	✓	✓	**Rules+LLM Judge**

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

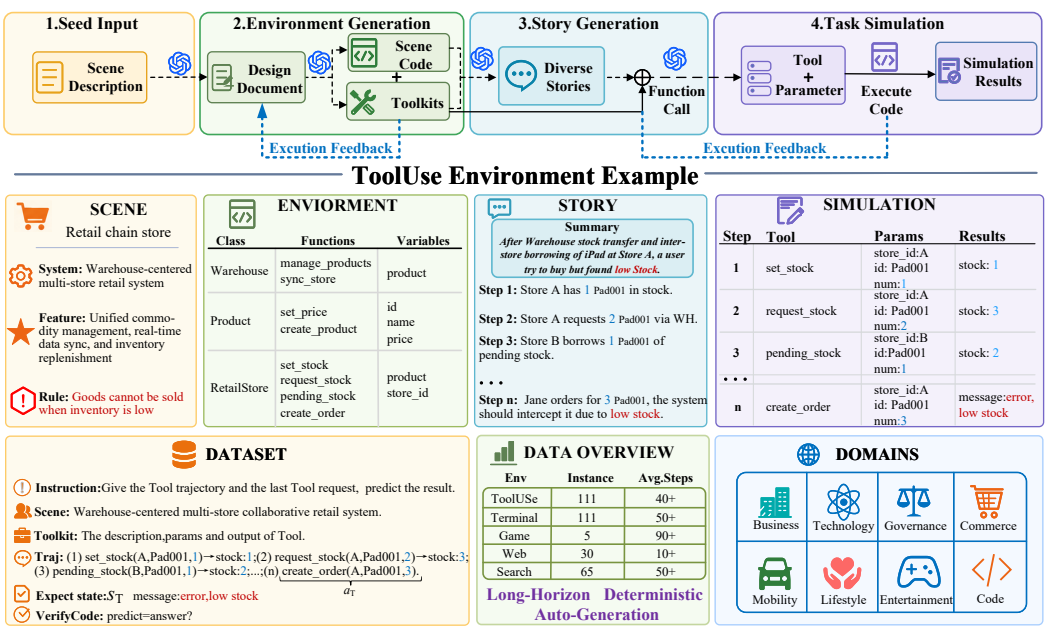


Figure 1: Overview of the WorldSimBench construction pipeline. Starting from domain seeds, we (i) build executable environments, (ii) generate long-horizon interaction plans through LLM-assisted story planning, (iii) execute each plan to obtain trajectories with fixed reference targets, and (iv) evaluate predictions using environment-specific verification protocols.

Sun et al., 2026; Cheng et al., 2026), making it difficult to isolate simulator quality from other factors in the training pipeline. Another line directly evaluates simulator outputs, but faces three limitations. First, existing evaluations are often restricted to a single interaction family, such as web automation, text games, RPGs, or tool environments (Ning et al., 2025; Furuta et al., 2023; Wang et al., 2024; Yu et al., 2025; Liu et al., 2026), leaving cross-environment generalization insufficiently studied. Second, existing evaluations typically focus on short interactions (Gao et al., 2025), leaving long-horizon state consistency underexplored. Third, many rely on open-ended LLM-based scoring (Choi et al., 2026; Fang et al., 2025), which may introduce evaluator and prompt dependence. In concurrent work, EnvSimBench evaluates LLM-based environment simulation through fully observable state-prediction samples with verifiable labels (Liu et al., 2026). WorldSimBench is complementary, focusing on long-horizon, trajectory-level simulation with cross-step dependencies across ToolUse, Terminal, Game, Web, and Search. Table 1 summarizes representative evaluation protocols.

We introduce **WorldSimBench**, comprising 3,580 tasks derived from 322 environments across ToolUse, Terminal, Game, Web, and Search, with trajectories of up to 199 interaction steps. As illustrated in Figure 1, we construct the benchmark through **code-grounded story simulation**, which separates open-ended scenario planning from verifiable environment execution. An LLM plans diverse, long-horizon interactions containing early evidence, intermediate distractors, and cross-step dependencies, while environment-specific backends execute the plans to produce explicit state transitions and fixed reference outcomes. ToolUse, Terminal, Game, and Search use deterministic code-based or rule-based verifiers, whereas Web uses a fixed, rubric-constrained LLM judge to check predefined requirements for components and values.

Across 32 evaluated model configurations, GPT-5.5-Think-xHigh achieves the highest unweighted macro-average success rate of 81.39%, while seven of the thirteen non-reasoning configurations remain below 20%. **Under the reported mode-specific inference settings, reasoning configurations achieve higher macro-average success rates in all 13 paired comparisons.** Model rankings and dominant failure modes vary across environments. Across eight external benchmarks, domain-level macro-average rank correlations are positive but nonuniform over overlapping model sets, indicating partial agreement in model rankings.

Our contributions are threefold: (1) **WorldSimBench**, a benchmark comprising 3,580 tasks from 322 environments across five interaction types; (2) **code-grounded story simulation**, a framework for constructing long-horizon interaction trajectories with verifiable reference outcomes; and (3) a systematic evaluation of 32 model configurations that examines performance differences between reasoning and non-reasoning configurations, model rankings across environments, dependence on interaction history, and environment-specific failure modes.

2 Related Work

Environment Simulators. Existing work studies programmatic, generative, and hybrid approaches to environment simulation. *Programmatic simulators* represent states and transition rules in executable code (Song et al., 2026). *Generative simulators* use learned models to produce observations and trajectories for agent training (Wang et al., 2025b; Chen et al., 2025b; Li et al., 2025), web interaction (Gao et al., 2025; Fang et al., 2025), and planning or reinforcement learning (Chen et al., 2025a; Wang et al., 2025a;c; 2026). NeuralOS and WorldGPT also model GUI and multimodal state transitions, respectively (Rivard et al., 2025; Ge et al., 2024). *Hybrid neural-code approaches* combine model generation with code execution, rendering, or verification (Zhang et al., 2026; Zheng et al., 2026; Song et al., 2026).

Evaluation of Environment Simulators. Existing protocols evaluate simulators through downstream agent performance (Wang et al., 2025b; Chen et al., 2025b; Li et al., 2025), human ratings (Wang et al., 2025b), LLM or VLM judgments (Fang et al., 2025; Wang et al., 2026; Zheng et al., 2026), or executable checks (Song et al., 2026). Downstream performance depends on both simulator quality and the agent training procedure, while human and model-based judgments depend on the evaluator. Direct benchmarks include text-world and RPG simulation (Wang et al., 2024; Yu et al., 2025). EnvSimBench evaluates single-step tool transitions given the full current state and tool code (Liu et al., 2026). AgentWorldBench evaluates observation prediction from interaction histories across seven domains using reference-grounded LLM scoring (Zuo et al., 2026). WorldSimBench complements these benchmarks with long-horizon outcome prediction across five interaction types, using code-grounded story simulation to introduce cross-step dependencies. ToolUse, Terminal, Game, and Search use deterministic verifiers, while Web uses a fixed LLM verifier to check predefined requirements for components and values.

3 WorldSimBench

3.1 Overview

We study whether large language models can serve as environment simulators that produce checkable and consistent feedback over multi-step interactions. We consider five environment types: ToolUse, Terminal, Game, Web, and Search. Each type defines an interaction context, a history of state transitions or observations, a target outcome, and an environment-specific verification protocol.

Our design combines executable environments, planned interaction histories, and environment-specific verification, turning executed trajectories into prediction tasks by withholding the final outcome.

Task formulation. Each WorldSimBench instance specifies a goal g, an initial state s_0, an action interface $\mathcal{A}$, and an action sequence $a_1, \ldots, a_T$, where $a_t \in \mathcal{A}$. Executing this sequence in the underlying environment produces

$$\tau = (a_1, s_1, \ldots, a_T, s_T),$$

where s_t denotes the observable result after executing a_t. During evaluation, the model observes

$$h_T = (a_1, s_1, \ldots, a_{T-1}, s_{T-1}, a_T),$$

which includes the final action but hides its outcome. Given a query q that specifies the target $y = f_q(s_T)$, the model predicts

$$\hat{y} = \mathcal{M}(g, s_0, \mathcal{A}, h_T, q).$$

Scoring. Each task has a fixed reference outcome y. An environment-specific verifier $\mathcal{V}$ scores the prediction as $\mathcal{V}(\hat{y}, y) \in \{0, 1\}$, and the final benchmark score is the unweighted mean of the success rates across ToolUse, Terminal, Game, Web, and Search. ToolUse, Terminal, Game, and Search use deterministic code-based or rule-based verification, whereas Web uses a fixed GPT-5.6-sol judge to verify predefined pairs of components and values. We detail the verifier design, correctness criteria, and reliability safeguards for all five environments in Appendix A.1 (Table 7).

Properties of WorldSimBench. Three properties guide our benchmark design. *Long-horizon consistency:* trajectories span up to 199 interaction steps, requiring models to track state changes and use information from earlier interactions. *Determinism:* each task has a fixed reference answer and explicit verification criteria. *Environment complexity:* tasks involve multiple entities, structured states, and transition rules, requiring models to combine information across interactions.

3.2 Code-Grounded Story Simulation

We use *code-grounded story simulation*: environments are generated from domain specifications or adapted from existing implementations, then checked through execution. For story-driven trajectories, we first generate a story and expand it into an outline with a target step count. At this stage, we control length and shape reasoning complexity by positioning decisive state changes, inserting distractors, and specifying cross-step dependencies. Models generate actions from the outline and execution history; environments execute them to produce observations. Each evaluation query asks for the final outcome given the preceding history and final action, with the executed result withheld as the reference answer. Search converts API traces into query–result histories while preserving facts and state changes. Failed actions are revised; candidates with persistent failures, unstable references, or insufficient context are discarded or rebuilt. Appendix B (Algorithms 1–5) provides the full procedures.

3.3 Environments

WorldSimBench spans five environment types that differ in state representation, action interface, prediction target, and verification protocol.

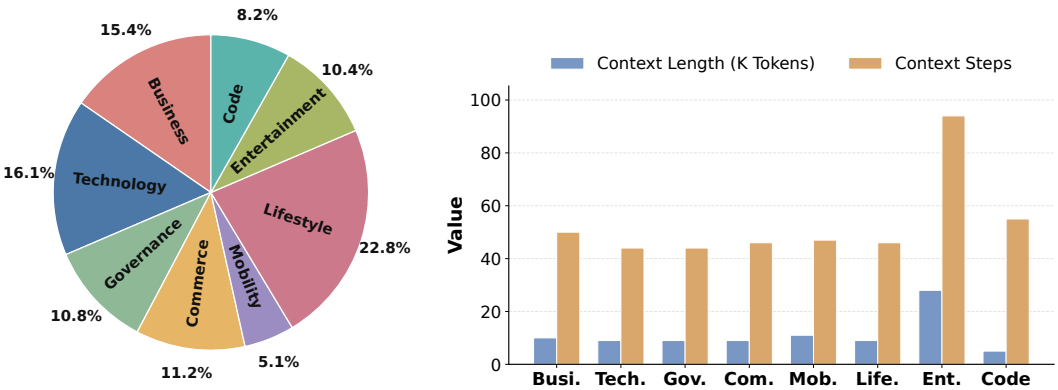
ToolUse. ToolUse evaluates state tracking and result prediction across multi-tool workflows. Each executable environment maintains a structured global state and exposes tools with typed parameter and return schemas; program execution determines every transition and reference output. Models predict the final tool result, evaluated by exact JSON matching. We exclude tasks with stochastic transitions, ambiguous tool behavior, or insufficient context.

Terminal. Terminal evaluates file-system states and shell-command effects over long trajectories constructed from TerminalBench seeds. Early file, configuration, and script operations can affect later outputs. Docker execution records standard output, standard error, exit codes, and persistent file changes, while a task-specific executable checker verifies the predicted output or final file state.

Game. Game evaluates inference of hidden transition rules and prediction of final board states in five games with two or three players. Matrix-based boards vary in size, and executable game code determines each transition. Models observe the board and action history but not the full rules; final predictions are verified by exact matrix matching.

Web. Web evaluates webpage state prediction after browser interactions. Pages are represented as `RootWebArea` accessibility trees, while actions may update both interface components and backend state. An oracle-backed checklist specifies the required final pairs of components and values, and GPT-5.6-sol (high reasoning effort) checks all required pairs.

Search. Search evaluates information aggregation over long histories of queries and results. The tasks span diverse scenarios and query types. Models answer a final query that may depend on multiple earlier results. A deterministic verifier marks a prediction correct only when it contains every reference-supported keyword.



(a) Domain distribution.

(b) Context length and interaction steps by domain.

Figure 2: Domain coverage and length statistics. Abbreviations are Busi. = Business&Administration, Tech. = Technology&Computing, Gov. = Public Governance, Com. = Commerce&Retail, Mob. = Mobility&Transport, Life. = Lifestyle&Education, and Ent. = Entertainment.

3.4 Dataset Statistics

Table 2 summarizes the scale of WorldSimBench: the benchmark spans 322 environments and 3,580 tasks across five environment types, with 181.6K interaction steps in total and trajectories reaching up to 199 steps. The five types provide complementary interaction settings: ToolUse covers structured service workflows, Terminal focuses on command and file-system effects, Game evaluates rule-governed board transitions, Web captures interactive webpage dynamics, and Search requires evidence aggregation over long histories of queries and results.

Table 2: Dataset statistics of WorldSimBench. Total Steps denotes the aggregate number of interaction steps across all instances.

Category	Envs.	Tasks	Total Steps	Avg. Steps	Step Range
ToolUse	111	2,100	103.3K	49.2	35–76
Terminal	111	198	10.8K	54.5	6–85
Game	5	268	24.1K	90.0	9–199
Web	30	192	1.9K	10.2	4–20
Search	65	822	41.4K	50.3	41–61
All	**322**	**3,580**	**181.6K**	**50.7**	**4–199**

Figure 2 further summarizes application coverage, context length, and interaction steps.

3.5 Human Assessment

Ten annotators with backgrounds in AI research, algorithm engineering, or graduate-level AI study assess representative tasks from all five environments. Each rates task difficulty, data construction, state representation, and evaluator design on a 10-point scale. The mean across dimensions and environments is **9.15** (Table 3). These ratings provide a quality check for the assessed tasks without affecting model scoring. Appendix B.9 details the protocol, operational difficulty rubric, and a separate model-based difficulty analysis.

Table 3: Human assessment results on a 10-point scale, reported as mean ± standard deviation over ten annotators.

Metric	ToolUse	Terminal	Game	Web	Search	Avg.
Task Difficulty	8.80±0.26	8.85±0.53	8.85±0.63	**8.90**±0.55	8.70±0.50	8.82±0.30
Data Construction	**9.15**±0.41	8.85±0.58	9.00±0.67	9.05±0.60	9.10±0.55	9.03±0.36
State Representation	**9.50**±0.47	9.10±0.46	9.30±0.82	9.20±0.70	9.05±0.60	9.23±0.40
Evaluator Design	**9.80**±0.48	9.55±0.60	9.55±0.69	9.30±0.65	9.35±0.50	9.51±0.42
Overall	**9.31**±0.25	9.09±0.37	9.18±0.55	9.11±0.35	9.05±0.32	9.15±0.28

4 Experiments

4.1 Evaluation Setup

The main comparison covers 32 reasoning and non-reasoning configurations of commercial and open-weight LLMs, including 13 models tested in both modes. Four additional configurations are

Table 4: Success rates (%) by thinking mode and model family. ToolUse sub-domains: Business (Busi.), Technology (Tech.), Governance (Gov.), Commerce (Com.), Mobility (Mob.), and Lifestyle (Life.). ToolAll is overall ToolUse accuracy; Avg is the five-environment macro-average. Domain scores use the full task-set denominators. The best and second-best scores per column are **bolded** and underlined, respectively.

Model	ToolUse Sub-domains						ToolAll	Game	Terminal	Web	Search	Avg
	Busi.	Tech.	Gov.	Com.	Mob.	Life.						
With thinking												
GPT-5.5	**84.17**	85.54	78.10	84.48	57.89	**73.22**	**78.95**	82.09	83.33	**82.29**	**80.29**	**81.39**
Gemini-3.1-Pro-Preview	75.88	81.20	66.79	84.48	59.40	66.10	73.14	69.03	**84.85**	45.31	68.73	68.21
Claude-Opus-4.7	59.80	55.90	55.84	50.69	51.13	45.59	52.71	58.96	74.75	52.08	64.96	60.69
DeepSeek-V4-Pro	59.30	66.51	65.69	74.48	51.88	59.66	63.29	71.64	66.16	54.69	62.65	63.69
DeepSeek-V4-Flash-Max	45.23	52.53	46.35	56.21	41.35	49.15	49.19	26.12	54.55	41.15	48.66	43.93
DeepSeek-V3.2	37.69	41.20	29.56	61.72	35.34	42.54	41.86	17.54	45.45	45.83	35.64	37.27
Qwen-3.8-Max	80.40	**87.47**	74.09	76.90	54.14	68.64	75.52	**86.94**	**84.85**	73.96	72.38	78.73
Qwen-3.7-Max	56.28	58.31	56.93	58.97	42.11	52.71	55.24	30.60	68.69	46.88	46.11	49.50
Qwen-3.7-Plus	56.53	51.33	53.28	50.34	42.11	47.46	50.76	10.07	67.68	46.88	28.59	40.80
Qwen-3.6-Plus	26.88	26.51	28.83	43.79	15.79	32.71	30.33	6.72	52.53	36.46	34.31	32.07
Qwen-3.5-397B-A17B	24.62	25.54	32.85	35.86	15.79	28.47	27.95	32.09	51.01	46.35	35.16	38.51
Qwen-3-235B-A22B-2507	18.09	20.48	24.82	20.34	17.29	24.58	21.52	1.87	44.95	43.75	52.55	32.93
Qwen-3-32B	7.29	4.58	5.47	9.31	5.26	5.76	6.24	0.75	3.54	42.19	6.45	11.83
GLM-5.3	68.59	80.48	**78.47**	74.83	50.38	62.88	70.33	81.72	77.78	68.75	76.76	75.07
GLM-5.3-Flash	68.09	71.33	64.60	57.93	**60.15**	50.68	61.48	58.96	75.25	55.21	70.80	64.34
GLM-5.2	65.08	70.84	65.69	65.86	45.11	54.07	62.05	47.01	72.22	64.06	70.68	63.21
Kimi-K3	77.89	85.06	74.45	**86.55**	57.14	65.59	75.29	85.07	83.33	71.88	75.30	78.17
Kimi-K2.6	67.34	64.34	66.06	67.59	41.35	56.95	62.05	26.12	72.73	41.67	50.85	50.68
MiniMax-M2.5	10.55	11.57	12.04	11.03	10.53	11.86	11.38	4.85	33.33	44.79	18.25	22.52
Without thinking												
DeepSeek-V4-Pro	12.56	10.60	8.03	13.79	14.29	9.83	11.10	5.97	28.79	51.56	11.19	21.72
DeepSeek-V4-Flash	4.27	7.95	5.11	2.76	7.52	8.47	6.29	2.99	17.68	25.00	5.84	11.56
DeepSeek-V3.2	4.02	7.23	4.74	2.07	4.51	5.93	5.05	3.36	16.67	40.63	4.99	14.14
Qwen-3.8-Max	17.59	15.18	10.22	9.66	7.52	13.90	13.38	7.09	48.48	50.00	24.45	28.68
Qwen-3.7-Max	10.80	15.66	5.47	5.86	8.27	11.53	10.43	4.48	39.90	44.79	8.52	21.62
Qwen-3.7-Plus	7.29	6.99	5.11	6.90	12.78	12.37	8.67	1.49	46.97	31.25	9.12	19.50
Qwen-3.6-Plus	5.28	5.78	4.38	4.14	18.80	11.36	7.67	2.99	44.95	34.90	10.22	20.14
Qwen-3.5-397B-A17B	5.03	6.02	3.28	1.72	16.54	11.69	7.14	1.12	38.38	34.38	8.39	17.88
Qwen-3-235B-A22B	3.02	5.78	4.38	1.72	4.51	4.75	4.14	1.87	8.08	21.88	2.68	7.73
Qwen-3-32B	2.76	6.02	4.74	9.66	1.50	4.75	5.10	0.75	5.56	32.81	0.36	8.91
GLM-5.2	10.05	10.36	8.03	9.31	16.54	13.05	11.00	2.61	24.24	53.13	26.64	23.52
Kimi-K3	23.87	29.64	14.96	30.34	33.08	22.03	24.81	4.85	47.98	60.94	29.93	33.70
Kimi-K2.6	6.53	6.75	6.57	3.45	7.52	10.00	7.19	1.87	30.30	32.29	11.07	16.54

detailed in Appendix A.2.1. ToolUse, Terminal, Game, and Search use deterministic verification; Web uses a fixed GPT-5.6-sol judge over predefined pairs of components and values. Appendices A.1, A.2, and A.3 detail scoring, inference settings, and evaluation costs.

4.2 Main Results

GPT-5.5-Think-xHigh achieves the highest macro-average success rate (**81.39%**; Table 4), leading ToolUse, Web, and Search. Qwen-3.8-Max-Think leads Game (86.94%) and ties Gemini-3.1-Pro-Preview-Think-High on Terminal (84.85%). The best macro-average remains 18.61 percentage points below perfect accuracy. GLM-5.3 and GLM-5.3-Flash outperform GLM-5.2 in macro-average success rate under thinking mode, although GLM-5.3-Flash scores lower on ToolUse and Web.

Reasoning configurations achieve higher macro-average success rates for all 13 paired comparisons evaluated in both modes, although gains and model rankings vary across environments.

Evaluation Stability. We assess run-to-run stability by evaluating eight selected model configurations five times with fixed task sets and inference settings. Table 5 reports the mean and sample standard deviation of success rates. **The main performance comparisons remain consistent across repeated runs for the selected configurations.**

Table 5: Evaluation stability for selected model configurations over five runs. Values are mean ± std success rates (%).

Model	ToolAll	Game	Terminal	Web	Search
DS-V4-Pro-Think-High	51.67±1.39	44.03±2.79	54.04±3.91	42.19±2.98	48.54±1.27
GLM-5.1-Think	54.90±2.45	32.84±2.35	52.02±2.00	53.65±1.43	68.61±1.18
DS-V4-Flash-Think-High	35.29±1.52	15.30±1.34	39.39±2.92	31.25±2.69	40.75±1.11
Qwen-3.6-Plus-Think	29.71±2.49	6.72±1.41	52.53±1.34	36.46±2.48	34.31±1.31
DS-V4-Pro	11.10±1.68	5.97±2.45	28.79±1.64	51.56±2.28	11.19±0.85
Qwen-3.6-Plus	7.57±0.89	2.99±0.46	44.95±0.51	34.90±1.45	10.22±0.47
GLM-5.1	7.10±0.67	1.49±0.00	26.26±2.30	40.10±1.08	18.73±0.94
DS-V4-Flash	6.29±1.10	2.99±1.14	17.68±2.28	25.00±2.77	5.84±0.77

Cross-Environment Consistency. We compare model rankings across ten evaluation dimensions using pairwise Spearman correlations for the 19 reasoning and 13 non-reasoning configurations in Table 4. Correlations range from 0.63 to 0.98 and 0.23 to 0.90, with medians of 0.88 and 0.65, respectively (Figure 3). Web shows the weakest alignment within the reasoning group. **Reasoning configurations generally show more consistent but environment-dependent rankings.**

5 Analysis

5.1 Context-Retention Analysis

We evaluate DeepSeek-V4-Flash and DeepSeek-V4-Pro under trajectory truncation (Figure 4). Full-input baselines appear at 100% retention on the left and as horizontal dashed lines on the right; dashed segments connect the longest tested suffixes to these references.

ToolUse, Terminal, and Search depend strongly on earlier context. Removing the earliest 10% in Search reduces accuracy by 36.62 and 39.90 percentage points for Flash and Pro, respectively. **Game and Web show comparatively shorter effective dependencies.** Game's Last 10 improves accuracy over full input by 16.04 and 14.18 percentage points, respectively, while Web remains relatively stable. Recent states may already contain much of the information needed to predict the final state, with earlier interactions adding redundancy or interference.

Detailed accuracies and mean trajectory lengths appear in Appendix C.1; complementary dependency annotations appear in Appendix C.2.

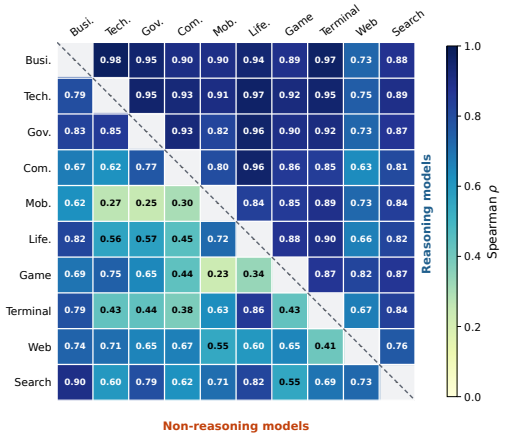


Figure 3: Spearman rank correlations across six ToolUse sub-domains, Game, Terminal, Web, and Search. Upper/lower triangles use the 19 reasoning and 13 non-reasoning configurations in Table 4, respectively.

5.2 External Benchmark Correlations

We compare WorldSimBench rankings with eight external benchmarks using Spearman rank correlations over overlapping models (Appendix C.3).

Table 6: Spearman rank correlations (ρ) between external benchmarks and WorldSimBench domains over overlapping models.

Benchmark	#M	Tool	Game	Term.	Web	Search
SWE-Verified	4	0.80	1.00	0.20	1.00	0.80
QwenWebDev	4	1.00	0.80	0.60	0.80	0.40
TerminalBench 2.0	5	0.70	0.50	0.60	0.50	0.10
QwenWorldBench	5	0.70	0.50	0.60	0.50	0.10
SWE-Pro	5	0.60	0.20	0.80	0.20	0.00
SWE-Multilingual	4	0.20	0.40	0.80	0.40	0.20
SciCode	4	0.80	0.40	0.60	0.40	0.20
Toolathlon	8	0.81	0.56	0.79	0.62	0.79
Macro average	–	0.70	0.55	0.62	0.55	0.32

The macro-average correlations are highest for ToolUse and Terminal ($\rho = 0.70$ and 0.62, respectively), followed by Web (0.55), Game (0.55), and Search (0.32). Given the small model overlaps, these results should be interpreted descriptively. **The positive but nonuniform correlations suggest that WorldSimBench and external agent benchmarks capture related but non-equivalent capabilities.** Appendix C.3.1 provides a separate domain-aligned analysis of model-level rank gaps between agent and simulator evaluations.

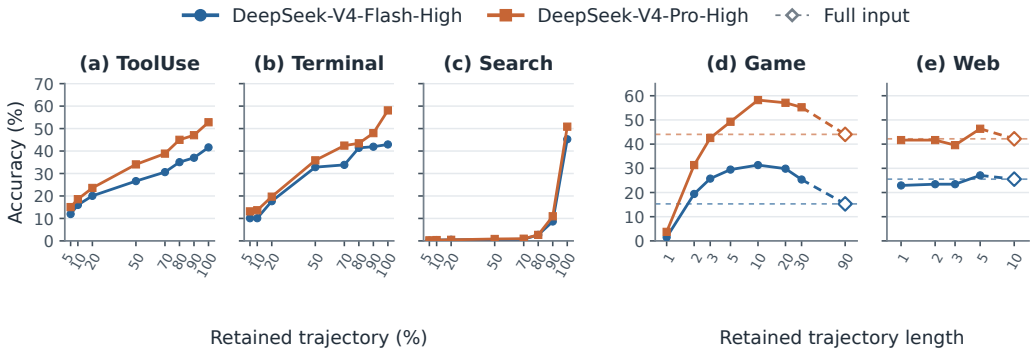


Figure 4: Accuracy under trajectory truncation. Left: retained trajectory percentage for ToolUse, Terminal, and Search. Right: retained suffix length for Game and Web.

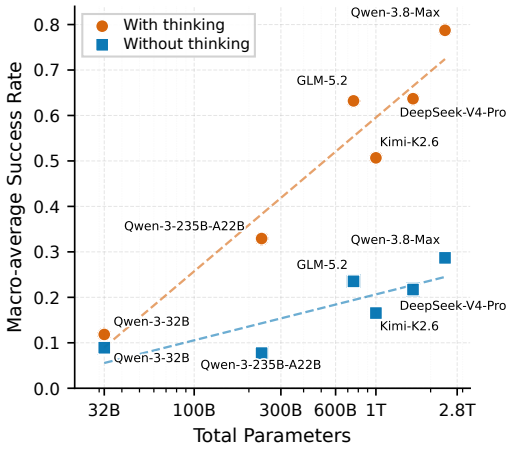


Figure 5: Total parameter count and five-environment macro-average success rate for six matched model families.

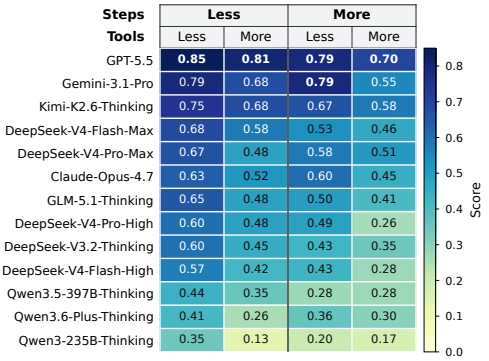


Figure 6: ToolUse accuracy across four complexity quadrants defined by trajectory length S and available tool-set size T. Less/More correspond to $S \leq 45/S \geq 49$ and $T \leq 10/T \geq 13$. Darker colors indicate higher accuracy.

5.3 Performance by Model Size

Figure 5 examines the association between total parameter count and the macro-average success rate across five environment types for six model families evaluated both with and without thinking (12 configurations). The families were selected to span the available parameter range rather than according to their observed success rates. **Parameter count is positively associated with success rate in both inference modes, although the relationship is not monotonic.** On the log-parameter scale shown in the figure, the Pearson correlations are $r = 0.96$ for configurations with thinking ($n = 6$) and $r = 0.83$ for configurations without thinking ($n = 6$). These associations are descriptive rather than causal because the configurations differ in architecture, training data, and inference settings. Appendix C.4 extends the comparison to all 21 configurations with disclosed sizes in the full analysis set.

5.4 Performance by Trajectory Length and Tool-Set Size

Figure 6 compares ToolUse accuracy across four quadrants formed by trajectory length and the number of available tools. **Averaged across the 13 configurations, the long-trajectory, large-tool-set quadrant has the lowest accuracy; however, this ordering does not hold for every individual configuration.** This analysis is descriptive rather than causal: task content also varies across quadrants, so it does not identify the independent effects of trajectory length or tool-set size.

5.5 Failure Patterns

We analyze a diagnostic sample of 698 audited prediction runs from DeepSeek-V4-Flash-Think-High: 399 from ToolUse, 102 from Game, 34 from Terminal, 65 from Web, and 98 from Search. Each run is treated as one analysis unit, including repeated predictions on the same task. We inspect the task context, reference outcome, submitted prediction, and API-returned `reason` field, treating the latter as an observable rationale rather than latent chain-of-thought. Each run receives one primary error category; annotation details are provided in Appendix D.

Figure 7 shows distinct failure profiles. Instruction-following errors dominate ToolUse (47.9%), while Game is split between similarity interference (38.2%) and instruction following (37.3%). Numeric errors are most frequent in Terminal (26.5%), whereas missing evidence dominates Web (46.2%) and Search (43.9%). The appendix further shows that only 13 of 28 recovered ToolUse conclusions are propagated to the submitted answer.

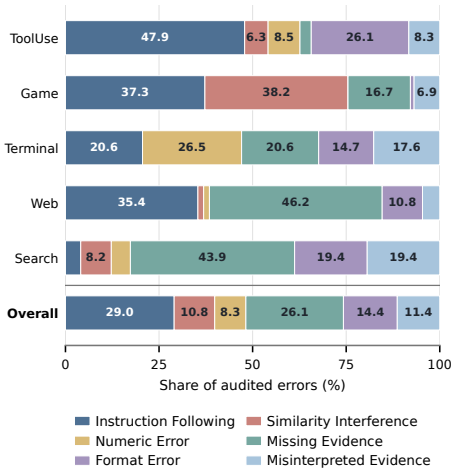


Figure 7: Primary error-category distributions for DeepSeek-V4-Flash-Think-High in the diagnostic sample ($N = 698$). Each run receives one primary label. Bars are normalized within each environment; Overall denotes the unweighted macro-average across the five environments.

These results identify environment-specific bottlenecks and show that failures can arise both during state reconstruction and when transferring a corrected conclusion into the final structured output. These proportions characterize the audited sample rather than benchmark-wide failure prevalence.

6 Conclusion

We introduced **WorldSimBench**, a benchmark for evaluating LLMs as environment simulators across ToolUse, Terminal, Game, Web, and Search. Our **code-grounded story simulation** combines story planning with environment execution to construct long-horizon trajectories with fixed reference outcomes. The benchmark evaluates outcome prediction from interaction histories separately from action selection.

Our evaluation shows that reasoning configurations achieve higher macro-average success rates in all paired comparisons under the reported inference settings. However, even the strongest configurations make errors, and model rankings vary across environments. Comparisons with external agent benchmarks show positive but nonuniform rank correlations over overlapping model sets, suggesting that agentic and environment-simulation capabilities are related but not interchangeable.

Further analysis shows that the role of interaction history differs across environments. For the two tested models, earlier history is important for ToolUse, Terminal, and Search, while retaining only recent interactions improves accuracy in Game. In the audited ToolUse cases, models sometimes correct their conclusions in observable rationales but fail to update the submitted answers. These findings suggest that reliable simulation requires preserving relevant state, limiting interference from irrelevant history, and maintaining consistency between corrected conclusions and final outputs. They motivate further study of explicit state summaries, selective history retention, and output verification.

Limitations

WorldSimBench is limited to text or accessibility-tree observations, excluding screenshot-based and embodied settings. Web uses a fixed, rubric-constrained LLM verifier with a predefined accessibility-tree format check, checklist of components and values, and structured binary output. This protocol is applied uniformly across models, while fully programmatic validation of accessibility-tree semantics remains future work.

AI USE STATEMENT

Generative AI tools were used to generate and repair environment code, synthesize trajectories and prompts, assist with experimental design, annotate dependencies and failure categories, conduct qualitative error analysis, and edit the manuscript for clarity. The authors reviewed all AI-assisted outputs, tested executable artifacts, and verified reported statistics against the underlying evaluation records. The authors take responsibility for the final content of this work.

ETHICS STATEMENT

The human assessment involved ten annotators with backgrounds in AI research, algorithm engineering, or graduate-level AI study. Participation was voluntary, and annotators provided informed consent, though no compensation was provided. Annotators evaluated benchmark instances and provided task-level quality ratings. We report only aggregate results and do not release annotator identities or individual responses. The assessment did not collect sensitive personal data. Under our institution's policy, this human evaluation did not require IRB review.

REPRODUCIBILITY STATEMENT

The complete generation procedures, prompts, evaluation protocols, model settings, and dataset schema are documented in the appendix. Code, data, deterministic verifiers, and evaluation scripts will be provided through anonymized supplementary material.

REFERENCES

Hao Bai, Alexey Taymanov, Tong Zhang, Aviral Kumar, and Spencer Whitehead. Webgym: Scaling training environments for visual web agents with realistic tasks. *arXiv preprint arXiv:2601.02439*, 2026.

Shiqi Chen, Tongyao Zhu, Zian Wang, Jinghan Zhang, Kangrui Wang, Siyang Gao, Teng Xiao, Yee Whye Teh, Junxian He, and Manling Li. Internalizing world models via self-play finetuning for agentic rl. *arXiv preprint arXiv:2510.15047*, 2025a.

Zhaorun Chen, Zhuokai Zhao, Kai Zhang, Bo Liu, Qi Qi, Yifan Wu, Tarun Kalluri, Sara Cao, Yuanhao Xiong, Haibo Tong, et al. Scaling agent learning via experience synthesis. *arXiv preprint arXiv:2511.03773*, 2025b.

Zihao Cheng, Hongru Wang, Zeming Liu, Xinyi Wang, Xiangrong Zhu, Yuhang Guo, Wei Lin, Jeff Z. Pan, and Yunhong Wang. Terminal-world: Scaling terminal-agent environments via agent skills. *arXiv preprint arXiv:2605.20876*, 2026.

Junhyuk Choi, Sohhyung Park, Chanhee Cho, Hyeonchu Park, and Bugeun Kim. Diagnosing the reliability of llm-as-a-judge via item response theory. *arXiv preprint arXiv:2602.00521*, 2026.

Hang Ding, Peidong Liu, Junqiao Wang, Ziwei Ji, Meng Cao, Rongzhao Zhang, Lynn Ai, Eric Yang, Tianyu Shi, and Lei Yu. Dynaweb: Model-based reinforcement learning of web agents. *arXiv preprint arXiv:2601.22149*, 2026.

Tianqing Fang, Hongming Zhang, Zhisong Zhang, Kaixin Ma, Wenhao Yu, Haitao Mi, and Dong Yu. Webevolver: Enhancing web agent self-improvement with co-evolving world model. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 8970–8986, 2025.

Hiroki Furuta, Yutaka Matsuo, Aleksandra Faust, and Izzeddin Gur. Language model agents suffer from compositional generalization in web automation. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*, 2023.

Yifei Gao, Junhong Ye, Jiaqi Wang, and Jitao Sang. Websynthesis: World-model-guided mcts for efficient webui-trajectory synthesis. *arXiv preprint arXiv:2507.04370*, 2025.

Zhiqi Ge, Hongzhe Huang, Mingze Zhou, Juncheng Li, Guoming Wang, Siliang Tang, and Yueting Zhuang. Worldgpt: Empowering llm as multimodal world model. In *Proceedings of the 32nd ACM International Conference on Multimedia*, pp. 7346–7355, 2024.

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

Yi Liu, TingFeng Hui, Wei Zhang, Li Sun, Ningxin Su, Jian Wang, and Sen Su. Envsimbench: A benchmark for evaluating and improving llm-based environment simulation. *arXiv preprint arXiv:2605.07247*, 2026.

Liangbo Ning, Ziran Liang, Zhuohang Jiang, Haohao Qu, Yujuan Ding, Wenqi Fan, Xiao-yong Wei, Shanru Lin, Hui Liu, Philip S Yu, et al. A survey of webagents: Towards next-generation ai agents for web automation with large foundation models. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pp. 6140–6150, 2025.

Luke Rivard, Sun Sun, Hongyu Guo, Wenhu Chen, and Yuntian Deng. Neuralos: Towards simulating operating systems via neural generative models. *arXiv preprint arXiv:2507.08800*, 2025.

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

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

Kangrui Wang, Pingyue Zhang, Zihan Wang, Yaning Gao, Linjie Li, Qineng Wang, Hanyang Chen, Chi Wan, Yiping Lu, Zhengyuan Yang, et al. Vagen: Reinforcing world model reasoning for multi-turn vlm agents. *arXiv preprint arXiv:2510.16907*, 2025a.

Ruoyao Wang, Graham Todd, Ziang Xiao, Xingdi Yuan, Marc-Alexandre Côté, Peter Clark, and Peter Jansen. Can language models serve as text-based world simulators? In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 1–17, 2024.

Yiming Wang, Da Yin, Yuedong Cui, Ruichen Zheng, Zhiqian Li, Zongyu Lin, Di Wu, Xueqing Wu, Chenchen Ye, Yu Zhou, et al. Llms as scalable, general-purpose simulators for evolving digital agent training. *arXiv preprint arXiv:2510.14969*, 2025b.

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

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, et al. Ragen: Understanding self-evolution in llm agents via multi-turn reinforcement learning. *arXiv preprint arXiv:2504.20073*, 2025c.

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

Pengfei Yu, Dongming Shen, Silin Meng, Jaewon Lee, Weisu Yin, Andrea Yaoyun Cui, Zhenlin Xu, Yi Zhu, Xingjian Shi, Mu Li, et al. Rpgbench: Evaluating large language models as role-playing game engines. *arXiv preprint arXiv:2502.00595*, 2025.

Zeyu Zhang, Guohao Li, Zhenchang Xing, Alexandros Apostolopoulos, Yu Lin Lee, and Liang Zheng. Gecko: A simulation environment with stateful feedback for refining agent tool calls. *arXiv preprint arXiv:2602.19218*, 2026.

Yuhao Zheng, Li'an Zhong, Yi Wang, Rui Dai, Kaikui Liu, Xiangxiang Chu, Linyuan Lv, Philip Torr, and Kevin Qinghong Lin. Code2world: A gui world model via renderable code generation. *arXiv preprint arXiv:2602.09856*, 2026.

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

APPENDIX CONTENTS

A Evaluation Protocol and Settings **14**
A.1 Evaluation Protocol and Output Parsing 14
A.2 Model Inference Settings 15
A.3 Inference Cost and Runtime 17

B Dataset Construction and Quality Assessment **17**
B.1 Shared Quality Control 17
B.2 ToolUse Data Generation Procedure 18
B.3 Game Data Generation Procedure 18
B.4 Terminal Data Generation Procedure 19
B.5 Web Data Generation Procedure 20
B.6 Search Data Generation Procedure 20
B.7 Terminal Verification 21
B.8 Construction and Verification Settings 21
B.9 Human Evaluation Details 22

C Additional Results and Analyses **23**
C.1 Detailed Context-Retention Results 23
C.2 LLM-Annotated Dependency Analysis 24
C.3 External Benchmark Rank-Correlation Details 25
C.4 Fully Annotated Model-size Scaling Plot 28

D Failure Analysis **29**
D.1 Error-Attribution Protocol 29
D.2 Credit Report: Long-Horizon State Reconstruction 30
D.3 Reasoning Self-Correction Diagnostics 31
D.4 Reasoning-Trace Failure Cases 32

E Dataset Schema and Field Definitions **34**
E.1 Dataset Schema 34
E.2 Field Definitions 35

F Full Prompt Templates **36**
F.1 ToolUse Generation Prompts 36
F.2 Generation Prompts for Terminal 44
F.3 Generation Prompts for Game 47
F.4 Generation and Evaluation Prompts for Web 47
F.5 Generation and Evaluation Prompts for Search 51
F.6 Diagnostic Annotation Prompts 55

A Evaluation Protocol and Settings

This section specifies the scoring rules, inference settings, and evaluation cost. Supplementary results and diagnostic analyses are reported in Appendices C and D.

A.1 Evaluation Protocol and Output Parsing

Each evaluation instance requires a single final-step prediction. Each prompt provides the domain-specific context, preceding interactions, and final action or query. The model is asked to output only the target result expected by the environment. Scoring follows an environment-specific protocol: ToolUse, Terminal, Game, and Search use deterministic code-based or rule-based verifiers, while Web uses a rule-constrained LLM verifier. The Web judge checks the required accessibility-tree format and all requirements for components and values, then returns a binary label. Table 7 summarizes the expected output, reference source, and correctness rule for each environment.

Reliability safeguards. All verification protocols are fixed for evaluation and applied uniformly to every evaluated model. The environment-specific rules are specified below. Web predictions use the same GPT-5.6-sol prompt, reasoning setting, and output schema. The verifiers normalize specified formatting differences and check the required task content. Human assessors rated evaluator design at 9.51 ± 0.42; this measures perceived design quality, not verifier accuracy.

Table 7: Domain-specific evaluation protocols. Detailed parsing rules and exceptions are specified below.

Domain	Expected output	Reference source	Pass criterion
ToolUse	Queried tool return (JSON)	Full trajectory execution in the generated API class	Recursive match of required keys, types, and values.
Terminal	Task-specified command output, file-state or structured answer	Docker execution result and task-specific Python checker	Pass the task-specific comparison checks.
Game	Final board or state matrix	Move-sequence execution in the game engine	Exact match of dimensions, symbols, and cell values.
Web	Webpage accessibility tree	Reference page and component–value checklist	Pass the fixed verifier's format and checklist checks.
Search	Final search result	Reference answer and required-keyword list	Contain every required keyword after lowercasing.

Parsing scope. Deterministic parsing normalizes only superficial wrappers such as Markdown fences or leading labels; it does not repair missing fields, incorrect values, or wrong state structures. Web uses a rule-constrained LLM verifier over a predefined checklist of components and values.

ToolUse verification. Predictions and references are parsed as JSON and compared recursively. Superficial serialization and numeric-format differences are normalized, whereas required keys, types, and values must match. Missing fields, incorrect types or values, and semantically conflicting additional fields are rejected. For failure returns, the comparison excludes the natural-language `error_message` field and evaluates the structured failure fields.

Terminal verification. The checker compares normalized text, numeric values, files, or key fields against the reference. Predictions that cannot be parsed or fail the required checks are scored as incorrect.

Game verification. Matrix dimensions, symbols, and cell values must exactly match the reference after parsing. Partial board matches are not counted as correct.

Web verification. A fixed GPT-5.6-sol verifier with `reasoning.effort=high` applies a two-stage rubric. It first checks whether the prediction follows the required `RootWebArea` accessibility-tree format and rejects prose, JSON, Markdown, and bare lists of components and values. It then checks whether every predefined component has the expected value. The same prompt, configuration, checklist construction procedure, and structured binary response schema are used for all models.

Search verification. After lowercasing, a deterministic verifier returns correct only if every nonempty task-specific keyword occurs in the predicted result. Each retained instance contains at least three verification keywords, and all keywords must match; therefore, predictions containing only partial evidence are rejected.

A.2 Model Inference Settings

Table 8 gives the Web output-token cap, sampling parameters, thinking mode, and specified model versions for the 32 configurations in Table 4. For the 13 models tested in both modes, adjacent rows show the accompanying changes in sampling settings. GPT-5.5, Claude-Opus-4.7, and Gemini-3.1-Pro are accessed through OpenRouter; the other verified access routes use Alibaba Model Studio. For Qwen-3.6-Plus, Qwen-3.7-Max, and Qwen-3.7-Plus, the fixed snapshot shown in the non-thinking row also applies to the thinking row. Qwen-3-235B-A22B uses the Thinking-2507 and Instruct-2507 versions in the thinking and non-thinking modes, respectively. The Qwen-3.8-Max non-thinking row lists `qwen3.8-max-0902`; its thinking-row snapshot is unspecified. The same prompt template is used within each environment.

Table 8: Inference settings for the 32 configurations in Table 4, with thinking and non-thinking settings adjacent for each model. Maximum output tokens are for Web; K denotes 1,000 tokens. "-" denotes an omitted parameter or API default.

Model	Thinking	Max output tokens	Temp.	Top-P	Additional settings
GPT-5.5	Enabled	100K	-	-	effort=xhigh
Gemini-3.1-Pro-Preview	Enabled	65K	-	-	budget=16K
Claude-Opus-4.7	Enabled	100K	-	-	effort=xhigh
DeepSeek-V4-Pro	Enabled	100K	1	1	effort=max
DeepSeek-V4-Pro	Disabled	100K	1	1	-
DeepSeek-V4-Flash	Enabled	100K	1	1	effort=max
DeepSeek-V4-Flash	Disabled	100K	1	1	-
DeepSeek-V3.2	Enabled	65K	1	0.95	-
DeepSeek-V3.2	Disabled	65K	1	0.95	-
Qwen-3.8-Max	Enabled	120K	1	0.95	effort=xhigh
Qwen-3.8-Max	Disabled	120K	0.7	0.8	-
Qwen-3.7-Max	Enabled	100K	0.7	0.9	effort=xhigh
Qwen-3.7-Max	Disabled	100K	0.6	0.95	rep.=1.05
Qwen-3.7-Plus	Enabled	65K	0.7	0.9	-
Qwen-3.7-Plus	Disabled	65K	0.6	0.95	rep.=1.05
Qwen-3.6-Plus	Enabled	65K	0.7	0.9	-
Qwen-3.6-Plus	Disabled	65K	0.6	0.95	rep.=1.05
Qwen-3.5-397B-A17B	Enabled	65K	0.7	0.9	-
Qwen-3.5-397B-A17B	Disabled	65K	0.6	0.95	rep.=1.05
Qwen-3-235B-A22B	Enabled	32K	0.7	0.9	-
Qwen-3-235B-A22B	Disabled	32K	0.6	0.95	rep.=1.05
Qwen-3-32B	Enabled	16.3K	0.7	0.9	-
Qwen-3-32B	Disabled	16.3K	0.6	0.95	rep.=1.05
GLM-5.3	Enabled	130K	1	0.95	effort=max
GLM-5.3-Flash	Enabled	130K	1	0.95	effort=max

Continued on next page.

Table 8 (continued): Inference settings.

Model	Thinking	Max output tokens	Temp.	Top-P	Additional settings
GLM-5.2	Enabled	100K	1	0.95	-
GLM-5.2	Disabled	100K	1	0.95	-
Kimi-K3	Enabled	120K	1	0.95	effort=max
Kimi-K3	Disabled	120K	1	0.95	-
Kimi-K2.6	Enabled	16K	1	0.95	-
Kimi-K2.6	Disabled	16K	0.6	0.95	-
MiniMax-M2.5	Enabled	30K	1	0.95	-

Specified model versions

Model	Thinking	Model version
Qwen-3.8-Max	Disabled	qwen3.8-max-0902
Qwen-3.7-Max	Disabled	qwen3.7-max-2026-06-08
Qwen-3.7-Plus	Disabled	qwen3.7-plus-2026-05-26
Qwen-3.6-Plus	Disabled	qwen3.6-plus-2026-04-02
Qwen-3-235B-A22B	Enabled	qwen3-235b-a22b-thinking-2507
Qwen-3-235B-A22B	Disabled	Qwen-3-235B-A22B-Instruct-2507

Notes. `effort` denotes `reasoning.effort` for GPT/Claude and `reasoning_effort` for DeepSeek/Qwen/Kimi/GLM-5.3; `budget` denotes `reasoning.max_tokens`, and `rep.` denotes `repetition_penalty`. Where applicable, mode-specific sampling parameters follow provider recommendations. Paired rows therefore compare the corresponding inference configurations rather than controlled single-parameter ablations. Models omitted from the version panel have no specific version listed. The text above specifies which snapshots also apply to thinking mode. Unspecified parameters use provider defaults.

A.2.1 Additional Model Configurations and Results

Table 9 reports results and inference settings for four additional configurations used in auxiliary analyses but omitted from the main comparison in Table 4. All four appear in the stability analysis (Table 5); the three thinking configurations also appear in the complexity analysis (Figure 6). The two DeepSeek configurations use high reasoning effort, whereas their thinking counterparts in Table 4 use max reasoning effort. The five-run means and sample standard deviations are reported separately in Table 5.

Table 9: Results and reported inference settings for additional model configurations. ToolAll denotes overall ToolUse accuracy; Avg. is the unweighted mean across the five environments. Output-token limits are non-Web / Web; K denotes 1,000 tokens.

(a) Success rates (%)

Model	ToolAll	Game	Terminal	Web	Search	Avg.
DeepSeek-V4-Pro-Think-High	52.81	44.03	58.08	42.19	50.85	49.59
DeepSeek-V4-Flash-Think-High	41.62	15.30	42.93	25.52	45.26	34.12
GLM-5.1-Think	54.62	32.84	52.02	53.65	68.61	52.35
GLM-5.1	7.24	1.49	26.26	40.10	18.73	18.77

(b) Inference settings

Model	Max output tokens Non-Web / Web	Temp.	Top-P	Thinking	Additional settings
DeepSeek-V4-Pro-Think-High	20K / 100K	1	1	Enabled	effort=high
DeepSeek-V4-Flash-Think-High	20K / 100K	1	1	Enabled	effort=high
GLM-5.1-Think	20K / 100K	1	0.95	Enabled	-
GLM-5.1	20K / 100K	1	0.95	Disabled	-

Notes. `effort` denotes `reasoning_effort`; "-" denotes no additional setting reported.

A.3 Inference Cost and Runtime

We measure token usage and runtime for GPT-5.5-Think-xHigh through OpenRouter with 10 concurrent requests. Token counts are obtained from API usage metadata, and wall-clock time is measured end-to-end. ToolUse and Game are measured on their complete evaluation sets. Terminal costs are estimated by scaling the earlier 212-instance run by $198/212$. For Web and Search, we evaluate 150 tasks sampled without replacement using seed 123 and extrapolate token usage and runtime to the corresponding full sets. Runtime estimates may vary with provider routing and service load. Table 10 reports the measured or extrapolated token usage and wall-clock time for each environment.

Table 10: GPT-5.5-Think-xHigh inference cost with 10 concurrent requests.

Environment	Instances	Tokens (M)	Time (min)
ToolUse	2,100	20.22	33.1
Terminal	198	1.73	15.1
Game	268	5.61	31.3
Web	192	10.56	78.9
Search	822	4.49	16.6
Total	**3,580**	**42.61**	**175.0**

For this configuration, the estimated totals are 42.61M tokens and 175 minutes, summed across the five environments. Token counts include prompt and completion tokens, with reasoning tokens already included in the completion count. External evaluator calls, including the Web judge, are excluded.

B Dataset Construction and Quality Assessment

This section describes shared quality control, the environment-specific generation procedures, construction settings, and human assessment of dataset quality.

B.1 Shared Quality Control

We apply the environment-specific checks in Table 11 during dataset construction. ToolUse, Terminal, Game, and Search retain only tasks with deterministic reference outcomes and rule-based verifiers. Web trajectories are validated against backend state and predefined component and value requirements, and predictions are scored by the fixed rule-constrained judge described in Appendix A.1.

Table 11: Unified quality-control criteria used during WorldSimBench construction.

Criterion	**Invalid candidate condition**	**Resolution**
Executability	A generated tool call, shell command, or game action fails under the corresponding executable environment.	Regenerate the affected action using execution feedback; discard the candidate if retries fail.
Determinism	Repeated execution of the same state-action sequence yields inconsistent reference outcomes due to randomness, time dependence, hidden files, or external services.	Remove non-deterministic branches, fix seeds or timestamps when applicable, or discard the candidate.
Observability	The final answer depends on information that is not exposed in the task context, tool documentation, command history, or game observations.	Add the missing context when it is part of the intended environment state; otherwise discard the candidate as under-specified.

Continued on next page.

Table 11 (continued): Quality-control criteria.

Criterion	Invalid candidate condition	Resolution
Verifier validity	The scoring procedure accepts malformed outputs or fails to distinguish the intended reference from nearby incorrect predictions.	Revise the verifier and test both positive and negative cases before retaining the candidate.
Long-range dependency	The final answer can be inferred from the final local step or a direct lookup without tracking earlier state changes.	Regenerate the story plan to introduce earlier evidence, distractors, and cross-step dependencies.
Format robustness	The expected output format is ambiguous or incompatible with deterministic comparison.	Specify the required output format and normalize only superficial wrappers, not semantic values.

B.2 TOOLUSE DATA GENERATION PROCEDURE

Algorithm 1 outlines the ToolUse construction pipeline. LLMs design the tools and trajectories; executable API code produces the state transitions and reference outputs.

Algorithm 1: ToolUse Data Generation Procedure

Require: Scenario description c, API and prompt templates, rollout models $\mathcal{M}$, repair budget R
Ensure: Executable trajectory $\tau = (a_1, s_1, \ldots, a_T, s_T)$, where $a_t = (\mathrm{api}_t, \mathrm{args}_t)$

Use an LLM to turn c into an interface design document.
Use an LLM and the API-code template to generate an executable API class.
Call `reset_data()` to initialize the API class state `_data` to its empty structure.
Generate the toolkit description, adapted scenario description, and story-design prompt.
Select APIs suitable for long-horizon consistency testing.
Ask models in $\mathcal{M}$ to generate diverse one-sentence stories.
for each retained story **do**
    Call the API class method `reset_data()` to start the story from an empty `_data` store.
    Expand the story into a step outline.
    for $t = 1, \ldots, T$ **do**
        Build a function-call prompt from the outline, history $\{(a_i, s_i)\}_{i=1}^{t-1}$, the current API name, and the toolkit description.
        Ask the LLM to produce action $a_t = (\mathrm{api}_t, \mathrm{args}_t)$.
        Execute a_t in the API class to obtain the next exposed state or result s_t.
        while execution fails and the repair budget R is not exhausted **do**
            Feed the error message and context back to the LLM.
            Ask the LLM to revise a_t.
            Re-execute a_t to obtain s_t.
        end while
        Record the successful pair (a_t, s_t).
        Retain the internal `_data` state updated by the executed API method for the next action.
    end for
    Use the final state or result s_T as the reference answer.
end for

B.3 GAME DATA GENERATION PROCEDURE

Game instances ask models to predict board states after a specified action. We use a language model to propose diverse turn-based board-game specifications, while executable game code validates each action and computes the resulting board state. Algorithm 2 summarizes the procedure.

Algorithm 2: Game Data Generation Procedure

Require: Game-design prompt, rule templates, rollout model, target trajectory length T
Ensure: Board-game trajectory $\tau = (a_1, s_1, \ldots, a_T, s_T)$, where s_t is the board state after action a_t

Ask an LLM to design diverse turn-based board games.
For each game, define a matrix-based board representation, piece or cell symbols, legal actions, turn order, movement rules, capture or update rules, and terminal conditions.

Treat the board representation as the game-state format and the rule description as the transition rule.
Ask an LLM to generate an executable game class from the rule description.
Initialize the board state s_0 according to the generated rule.
for $t = 1, \ldots, T$ **do**
    Ask the rollout model to propose action a_t under the current board state s_{t-1} and rule context.
    Execute a_t with the game class to obtain the next board state s_t.
    while a_t is invalid **do**
        Ask the rollout model to revise a_t or resample another legal action.
        Execute the revised action to obtain s_t.
    end while
    Record the successful pair (a_t, s_t).
end for
Use the final board state s_T produced by code execution as the reference answer.

B.4 Terminal Data Generation Procedure

Terminal instances ask models to predict the output of a specified shell command. Starting from TerminalBench seed tasks, we extend each task into a longer command-line trajectory while preserving its original goal and executable environment. An LLM proposes command sequences, which we execute in Docker to obtain reference outputs. Algorithm 3 summarizes the procedure.

Algorithm 3: Terminal Data Generation Procedure

Require: TerminalBench seed task, initial files or command code, checker, Dockerfile, rollout model, target trajectory length T
Ensure: Command-line trajectory $\tau = (a_1, s_1, \ldots, a_T, s_T)$, where a_t is a shell command and s_t is the exposed command result or file-state summary after executing a_t
Use a TerminalBench task as the seed, including its task description, initial files or command code, checking rule, and Dockerfile.
Ask an LLM to generate a long-horizon story or operation plan that preserves the original task goal while introducing intermediate file, script, log, configuration, or command dependencies.
Ask the LLM to convert the operation plan into a sequence of shell commands whose final result depends on earlier command effects.
Ask the LLM to write a natural-language evaluation criterion for the final command result.
Initialize a Docker environment from the seed task.
for $t = 1, \ldots, T$ **do**
    Execute command a_t in the Docker environment.
    Record the exposed command result or file-state summary s_t, including stdout, stderr, exit code, and relevant file changes.
    while command execution fails unexpectedly **do**
        Feed the execution error and current context back to the LLM.
        Ask the LLM to revise command a_t.
        Re-execute the revised command in Docker and update s_t.
    end while
    Record the successful pair (a_t, s_t).
end for
Provide the command contexts $\{(a_t, s_t)\}_{t=1}^{T}$, the evaluation criterion, the final command result s_T, and a `judge(predict, answer)` template to an LLM.
Ask the LLM to generate Python judging code.
Use the final command result s_T and generated checker as the reference answer and verification rule.

B.5 WEB DATA GENERATION PROCEDURE

The Web pipeline first synthesizes and verifies a stateful website, then executes browser interactions against it. Observations are `RootWebArea` accessibility trees. A backend oracle checks state transitions and supplies the values used in the final evaluation checklist. Algorithm 4 summarizes the procedure.

Algorithm 4: Web Data Generation Procedure

Require: Domain description c, synthesis and rollout models, browser executor, backend oracle, repair budget R, target trajectory length T
Ensure: Browser trajectory $\tau = (o_0, a_1, o_1, \ldots, a_T, o_T)$ and final checklist C of components and values
1: Generate a website design, a structured environment specification, and executable frontend and backend code from c.
2: Install and launch the generated services, and verify the website with Playwright.
3: **while** installation, startup, or browser verification fails and R is not exhausted **do**
4: Provide the failure context to the synthesis model, repair the artifacts, and rerun verification.
5: **end while**
6: Generate an initial state, reset the backend oracle to that state, open the website, and record the initial `RootWebArea` accessibility tree o_0.
7: Generate a story plan containing ordered interactions, a final probe, and oracle paths for the required final values; reject structurally invalid plans.
8: **for** $t = 1, \ldots, T$ **do**
9: Select and resolve browser action a_t from the plan, current accessibility tree, and prior trajectory.
10: Execute a_t through Playwright/CDP and capture the resulting accessibility tree o_t and backend state.
11: **while** the action is invalid or produces an unexpected result and R is not exhausted **do**
12: Return the execution feedback to the rollout model, revise a_t, and re-execute it.
13: **end while**
14: Record the successful transition (a_t, o_t).
15: **end for**
16: Query the oracle for the final state and critical values, and run its trajectory validator.
17: Materialize C from the planned component paths and oracle-backed values; require every value in C to be supported by o_T.
18: Retain only validated trajectories and format the prediction instance by holding out o_T; evaluate the prediction against C using the fixed Web judge.

B.6 SEARCH DATA GENERATION PROCEDURE

The Search pipeline executes a sequence of stateful API calls and converts the trace into search queries and results. Code execution supplies the reference state and values. The conversion step expresses them as paired queries and results while retaining the information needed for the final query. Algorithm 5 summarizes the procedure.

Algorithm 5: Search Data Generation Procedure

Require: Domain description c, API/prompt/adapter templates, construction and rollout models, repair budget R, target context length T
Ensure: Search instance $D = (q_1, r_1, \ldots, q_T, r_T, q_\star, y, K)$ with gold answer y and required keywords K
1: Generate an environment design document and an executable stateful Domain API from c; adapt and validate its prompts and adapter.
2: Generate a backward story blueprint specifying the target outcome, decisive predicates, required context APIs, trap mechanism, and forbidden recent reads.
3: Expand the blueprint into a chapter flow that terminates in a query API call, and reset the environment.
4: **for** each API step in the chapter flow **do**
5: Generate its arguments from the toolkit, seed context, and complete execution history, then execute it in the Domain API.
6: **while** execution fails and R is not exhausted **do**
7: Use the error to revise the arguments, insert a required prerequisite step, or replan the remaining story, then retry.
8: **end while**
9: Record the successful action and code-produced result.
10: **end for**
11: Snapshot the environment and replay the final query multiple times; discard the trace unless its result is deterministic.

12: Select a final query step with sufficient preceding context and concrete answer values, treating its earlier API trace as context.
13: Segment the context into T consecutive, non-overlapping groups and verify that no API step is skipped, duplicated, or reordered.
14: Translate each group into a realistic query and result pair (q_t, r_t) while preserving exact names, dates, identifiers, values, and state-changing effects.
15: Translate the final API call and result into the final query $q_\star$, gold answer y, and candidate keyword set K.
16: Merge candidate keywords with scalar values extracted from the executed final result; retain only result-supported keywords not already exposed by $q_\star$.
17: Keep only fully validated instances; score a prediction as correct iff every keyword in K occurs in its normalized output.

B.7 Terminal Verification

For the Terminal domain, each benchmark instance is associated with an executable checker script. Most instances use `{id}.py`; instances marked `standard_text=same` use the shared `same.py` checker. The checker script is used to verify whether the model-generated command or final output satisfies the expected execution result.
During dataset construction, command execution produces the reference output and any additional evidence required by the checker. The fixed task-specific checker then parses and compares the runtime `predict` and `answer` arguments. The checker does not re-execute commands or access the runtime file system, network, or external processes.

B.8 Construction and Verification Settings

Table 12 summarizes the model and decoding settings used during data construction, rollout, verification, and dependency annotation. Qwen-based construction models are accessed through Alibaba Model Studio, while Gemini-3.1-Pro-Preview is accessed through OpenRouter. The GPT-5.6-sol Web verifier is accessed through an Azure-hosted OpenAI Responses API endpoint. Private endpoint details and credentials are omitted. For Qwen-based prompts, `qwen3.5-plus` denotes the fixed snapshot `qwen3.5-plus-2026-02-15`; "Think" denotes enabling thinking mode with the same snapshot. These settings are separate from the evaluation-model settings in Appendix A.2.

Table 12: Models and decoding settings for data construction and verification.

Domain	Stage / prompt	Model	Settings
ToolUse	Tool design (App. F.1.1)	Qwen3.5-Plus (Think)	Temp. = 0.1
	Code generation (App. F.1.3)	Gemini-3.1-Pro-Preview (Think-High)	Budget = 16,000
	Final tool selection (App. F.1.4)	Qwen3.5-Plus	Thinking off; Temp. = 0.1
	Story blueprint design (App. F.1.5)	Qwen3.5-Plus	Thinking off; Temp. = 0.9
	Chapter flow design (App. F.1.6)	Qwen3.5-Plus (Think)	Thinking on
	Simulation function call	Qwen3.5-Plus (Think)	Thinking on; Temp. = 0.1
Terminal	Task generation (App. F.2.1)	Gemini-3.1-Pro-Preview (Think-High)	Budget = 16,000
	Docker execution code repair	Gemini-3.1-Pro-Preview (Think-High)	Budget = 16,000
	Checker generation (App. F.2.2)	Qwen3.5-Plus (Think)	Thinking on; Temp. = 0.1
Game	Move simulation (App. F.3.1)	Qwen3.5-Plus (Think)	Thinking on; Temp. = 0.1
Web	Environment design (App. F.4.1)	Qwen3.5-Plus	Thinking off; Temp. = 0.1
	Website code generation (App. F.4.1)	Gemini-3.1-Pro-Preview (Think-High)	Budget = 16,000
	Trajectory generation (App. F.4.2)	Qwen3.5-Plus	Thinking off; Temp. = 0.1
	Component and value judging (App. F.4.3)	GPT-5.6-sol	Effort = high
Search	Environment design (App. F.5.1)	Qwen3.5-Plus	Thinking off; Temp. = 0.1
	Domain API code generation (App. F.5.1)	Gemini-3.1-Pro-Preview (Think-High)	Budget = 16,000
	Trajectory and dataset construction (App. F.5.2)	Qwen3.5-Plus	Thinking off; Temp. = 0.1

Notes. Qwen3.5-Plus refers to `qwen3.5-plus-2026-02-15`; Think denotes the same snapshot with thinking enabled. Temp., Thinking, Budget, and Effort denote `temperature`, `enable_thinking`, `reasoning.max_tokens`, and `reasoning.effort`, respectively. On/off correspond to true/false. Only explicitly reported parameters are listed.

B.9 Human Evaluation Details

This section describes the human evaluation protocol, scoring rubrics, and results used to assess dataset quality.

The annotators had backgrounds in AI research, algorithm engineering, or AI-related study. Evaluators were shown ten representative examples from each of the five environment types: ToolUse, Terminal, Game, Web, and Search. They rated each example along four dimensions: task difficulty, data construction, state representation, and evaluator design. Scores were given on a 10-point scale. Table 13 defines the task-difficulty scale.

We collected ten valid responses. These ratings provide a limited check of the selected examples and do not affect model scores.

Table 13: Operational rubric for the 10-point task-difficulty rating.

Score	Difficulty	Operational criterion
1–2	Very low	The final answer can largely be read, copied, or obtained from the most recent step.
3–4	Low	A small number of state updates are required, but the dependencies are mainly local and the rules are simple.
5–6	Medium	Explicit multi-step state tracking, condition combination, or state computation is required.
7–8	High	The task involves long-range dependencies, multi-entity interactions, complex branches, or substantial distractor information.
9–10	Very high	Multiple difficult mechanisms occur together, requiring nearly complete reconstruction of the environment state and precise rule execution.

Model-based difficulty distribution. We separately ask DeepSeek-V4-Pro-Think-High to rate structural task difficulty. We score all 268 Game, 198 Terminal, and 192 Web instances, together with 300 randomly sampled ToolUse instances and 202 randomly sampled Search instances. For each instance, the judge receives the system prompt, user prompt, reference answer, and the rubric in Table 13. It assigns an integer score from 1 to 10 based on dependency span, state tracking, rule composition, and distractor density, together with a brief rationale. Table 14 summarizes the assigned scores by environment. This model-based analysis is distinct from the human quality ratings in Table 3.

Table 14: Model-based task-difficulty distribution under the rubric in Table 13. Percentages are rounded to one decimal place.

Environment	1–2	3–4	5–6	7–8	9–10	Mean Score
ToolUse ($n = 300$)	4.7%	9.3%	19.3%	59.0%	7.7%	6.75
Terminal ($n = 198$)	24.7%	20.2%	23.2%	17.2%	14.6%	5.12
Game ($n = 268$)	28.4%	24.6%	13.4%	16.4%	17.2%	4.91
Web ($n = 192$)	4.2%	14.6%	16.1%	45.3%	19.8%	6.80
Search ($n = 202$)	0.0%	1.0%	11.9%	74.8%	12.4%	7.65

The complete annotation template is provided in Appendix F.6.3 (Prompt 25).

C Additional Results and Analyses

This section provides detailed context-retention results, dependency annotations, external benchmark comparisons, and the fully annotated model-size analysis.

C.1 Detailed Context-Retention Results

Tables 15 and 16 report the accuracies underlying Figure 4 for DeepSeek-V4-Flash and DeepSeek-V4-Pro in the Think-High setting. The percentage-retention conditions remove a prefix and retain the remaining suffix for ToolUse, Terminal, and Search; the Last-k conditions retain only the final k units for Game and Web. Full-input rows report the corresponding untruncated baselines. All conditions use fixed domain-level denominators, with errors and missing predictions counted as incorrect. The final row of each table gives the mean full-input trajectory length from Table 2; these are dataset statistics, not additional fixed-length conditions.

Table 15: Accuracy (%) under percentage-based trajectory retention. The fixed denominators are 2,100 for ToolUse, 198 for Terminal, and 822 for Search. Retaining 100% corresponds to full input. The final row reports mean trajectory lengths in steps rather than accuracy.

	DeepSeek-V4-Flash			DeepSeek-V4-Pro		
Retained (%)	ToolUse	Terminal	Search	ToolUse	Terminal	Search
5	11.95	10.10	0.12	15.05	13.13	0.24
10	16.00	10.10	0.36	18.57	13.64	0.36
20	20.05	17.68	0.49	23.57	19.70	0.49
50	26.62	32.83	0.61	34.05	35.86	0.85
70	30.62	33.84	0.73	38.86	42.42	0.97
80	35.05	41.41	2.68	45.00	43.43	2.68
90	37.00	41.92	8.64	47.05	47.98	10.95
100 (full input)	41.62	42.93	45.26	52.81	58.08	50.85
Full-input mean steps	49.2	54.5	50.3	49.2	54.5	50.3

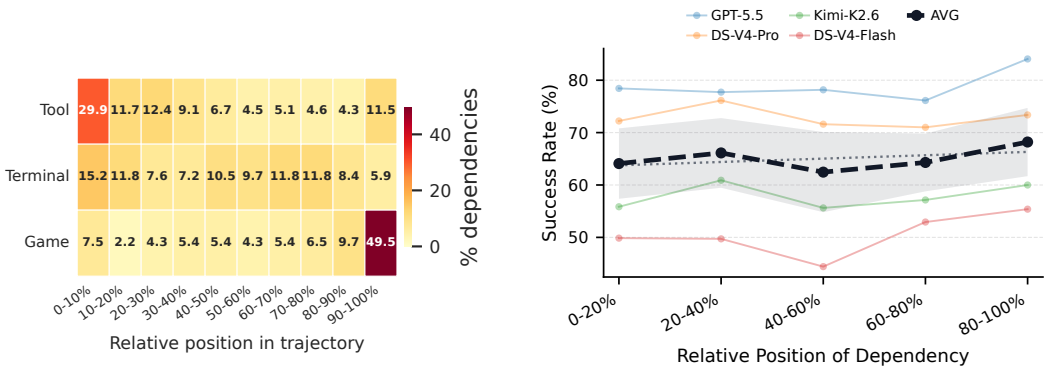
Table 16: Accuracy (%) under suffix-only trajectory retention. The fixed denominators are 268 for Game and 192 for Web. A dash indicates an unreported condition. The final row reports mean trajectory lengths in steps rather than accuracy.

	DeepSeek-V4-Flash		DeepSeek-V4-Pro	
Retained suffix	Game	Web	Game	Web
Last 1	1.49	22.92	3.73	41.67
Last 2	19.40	23.44	31.34	41.67
Last 3	25.75	23.44	42.54	39.58
Last 5	29.48	27.08	49.25	46.35
Last 10	31.34	–	58.21	–
Last 20	29.85	–	57.09	–
Last 30	25.37	–	55.22	–
Full input	15.30	25.52	44.03	42.19
Full-input mean steps	90.0	10.2	90.0	10.2

C.2 LLM-Annotated Dependency Analysis

To complement the context-retention experiments, we use LLM annotations to identify answer-relevant historical steps in ToolUse, Terminal, and Game. Figure 8(a) shows dependencies throughout the trajectories, with Game more concentrated near the end. Average accuracy is highest in the latest dependency-position bin, although the pattern is non-monotonic (Figure 8(b)). These results are descriptive: the annotations are model-generated, and comparisons across bins may reflect differences in trajectory length and task composition.

The annotator first proposes candidate historical steps, then judges whether each is necessary under the stated removal criterion. The same prompt is used for ToolUse, Terminal, and Game. Appendix F.6.1 provides the complete template (Prompt 23).



(a) Dependency position distribution.

(b) Performance across dependency positions.

Figure 8: LLM-annotated final-answer dependencies in ToolUse, Terminal, and Game. (a) Distribution of annotated dependencies by relative trajectory position. (b) Accuracy grouped by the position of the earliest annotated dependency. The DeepSeek-V4-Pro and DeepSeek-V4-Flash curves use the Think-Max setting.

C.3 External Benchmark Rank-Correlation Details

For each external benchmark b, we restrict the comparison to the set of models that are also evaluated in WorldSimBench, denoted by $\mathcal{M}_b$. For each WorldSimBench environment $d \in \{\text{ToolUse}, \text{Game}, \text{Terminal}, \text{Web}, \text{Search}\}$, let $R_b(i)$ and $R_d(i)$ denote the ranks of model $i \in \mathcal{M}_b$ on external benchmark b and WorldSimBench environment d, respectively. Rank 1 denotes the best-performing model. Ranks are recomputed within each overlapping model set, and tied scores are assigned their average ranks. For DeepSeek-V4, this comparison includes only the Max reasoning and non-reasoning configurations. The external Qwen-3.6-Plus results are matched to Qwen-3.6-Plus-Think in WorldSimBench.

Let

$$\bar{R}_b = \frac{1}{|\mathcal{M}_b|} \sum_{i \in \mathcal{M}_b} R_b(i), \qquad \bar{R}_d = \frac{1}{|\mathcal{M}_b|} \sum_{i \in \mathcal{M}_b} R_d(i).$$

We compute Spearman's rank correlation as the Pearson correlation between the two rank vectors:

$$\rho_{b,d} = \frac{\sum_{i \in \mathcal{M}_b} \left(R_b(i) - \bar{R}_b\right)\left(R_d(i) - \bar{R}_d\right)}{\sqrt{\sum_{i \in \mathcal{M}_b} \left(R_b(i) - \bar{R}_b\right)^2} \sqrt{\sum_{i \in \mathcal{M}_b} \left(R_d(i) - \bar{R}_d\right)^2}}.$$

For each WorldSimBench environment d, the final row of Table 6 reports the unweighted macro-average over the eight external benchmarks:

$$\bar{\rho}_d = \frac{1}{|\mathcal{B}|} \sum_{b \in \mathcal{B}} \rho_{b,d},$$

where $\mathcal{B}$ is the set of external benchmarks.

Several benchmarks yield the same correlations, which can occur with only four or five overlapping models. Such small overlap sets admit only a limited number of rank orderings and consequently only a limited number of Spearman correlation values. The repeated values consequently reflect the small overlap sizes rather than duplicated evaluations. Table 17 reports the external scores and within-overlap ranks, while Table 18 reports WorldSimBench ranks over the union of overlapping models.

External scores for the first seven benchmarks are taken from the Qwen3.7 release page.[1] Toolathlon scores are taken from the public leaderboard snapshot updated on June 2, 2026.[2] We use these reported scores only to determine model rankings within each overlapping model set.

[1] https://qwen.ai/blog?id=qwen3.7

[2] https://benchlm.ai/benchmarks/toolathlon

Table 17: External benchmark scores and within-overlap ranks. Each cell reports the original score followed by its rank in parentheses. QwenWeb reports a BT/Elo rating; all other columns report percentage scores. "–" indicates that no result is available.

(a)

Model	SWE-V	QwenWeb	TBench	QwenWorld
GPT-5.5-Think-xHigh	–	–	–	–
Kimi-K2.6-Think	80.2 (3)	–	66.7 (3)	50.9 (3)
GLM-5.1-Think	–	1564 (3)	63.5 (4)	50.2 (4)
DeepSeek-V4-Pro-Think-Max	80.6 (1)	1570 (1)	67.9 (2)	52.3 (2)
Qwen-3.7-Max-Think	80.4 (2)	1568 (2)	69.7 (1)	57.3 (1)
DeepSeek-V4-Flash-Think-Max	–	–	–	–
DeepSeek-V4-Pro	–	–	–	–
DeepSeek-V4-Flash	–	–	–	–
Qwen-3.6-Plus-Think	78.8 (4)	1500 (4)	61.6 (5)	47.6 (5)
Qwen-3.5-397B-A17B-Think	–	–	–	–

(b)

Model	SWE-Pro	SWE-Multi	SciCode	Toolathlon
GPT-5.5-Think-xHigh	–	–	–	55.6 (1)
Kimi-K2.6-Think	59.5 (2)	76.7 (2)	52.2 (2)	50.0 (3)
GLM-5.1-Think	58.8 (4)	–	45.1 (3)	–
DeepSeek-V4-Pro-Think-Max	59.0 (3)	76.2 (3)	–	51.8 (2)
Qwen-3.7-Max-Think	60.6 (1)	78.3 (1)	53.5 (1)	–
DeepSeek-V4-Flash-Think-Max	–	–	–	47.8 (4)
DeepSeek-V4-Pro	–	–	–	46.3 (5)
DeepSeek-V4-Flash	–	–	–	40.7 (6)
Qwen-3.6-Plus-Think	56.6 (5)	73.8 (4)	41.4 (4)	39.8 (7)
Qwen-3.5-397B-A17B-Think	–	–	–	36.3 (8)

Notes. SWE-V: SWE-Verified; QwenWeb: QwenWebDev; TBench: TerminalBench 2.0; QwenWorld: QwenWorldBench; SWE-Multi: SWE-Multilingual.

Table 18: WorldSimBench ranks for the union of models overlapping with the external benchmarks. Rankings are computed separately for each environment; tied scores receive average ranks.

Model	Tool	Game	Terminal	Web	Search
GPT-5.5-Think-xHigh	1	1	1	1	1
Kimi-K2.6-Think	3	6.5	2	7	4
GLM-5.1-Think	5	3	7	3	2
DeepSeek-V4-Pro-Think-Max	2	2	4	2	3
Qwen-3.7-Max-Think	4	5	3	5	6
DeepSeek-V4-Flash-Think-Max	6	6.5	5	8	5
DeepSeek-V4-Pro	9	9	9	4	9
DeepSeek-V4-Flash	10	10	10	10	10
Qwen-3.6-Plus-Think	7	8	6	9	8
Qwen-3.5-397B-A17B-Think	8	4	8	6	7

Table 18 reports ranks over the union of all overlapping models for presentation. For each correlation, WorldSimBench ranks are recomputed after restricting the models to the corresponding benchmark-specific overlap set $\mathcal{M}_b$.

C.3.1 Model-level Agent and Simulator Rank Gaps

Aggregate correlations do not show how individual models' relative ranks differ across benchmarks. We compare ToolUse with BFCL-V4, MCP-Mark, and MCP-Atlas, and Terminal with TerminalBench 2.0. These comparisons align task domains rather than individual instances.

We compare five model configurations matched by model version and reasoning mode across WorldSimBench and the external results: Kimi-K2.6-Think, GLM-5.1-Think, DeepSeek-V4-Pro-Think-Max, Qwen-3.7-Max-Think, and Qwen-3.6-Plus-Think. Claude-Opus-4.6 is excluded because WorldSimBench evaluates Claude-Opus-4.7. External agent scores are taken from the Qwen3.7 release page cited above. Table 19 reports the WorldSimBench and external scores, together with the within-overlap ranks used to compute the normalized gaps.

For each external benchmark b, let $n_b = 5$ be the number of matched model configurations. Let $R_{i,b}^{\mathrm{sim}}$ and $R_{i,b}^{\mathrm{agent}}$ denote model i's ranks on the matched WorldSimBench environment and external agent benchmark, respectively, where rank 1 is best. We define the normalized rank gap as

$$G_{i,b} = \frac{R_{i,b}^{\mathrm{sim}} - R_{i,b}^{\mathrm{agent}}}{n_b - 1}.$$

The gap lies in $[-1, 1]$. A positive value means that the model ranks relatively higher on the agent benchmark, while a negative value means that it ranks relatively higher on WorldSimBench. A zero gap indicates identical relative ranks, but does not imply high absolute performance.

For ToolUse, we report the unweighted mean of $G_{i,b}$ over BFCL-V4, MCP-Mark, and MCP-Atlas. TerminalBench 2.0 is reported separately; we do not average gaps across ToolUse and Terminal.

Table 19: WorldSimBench scores and domain-aligned agent-benchmark scores over five matched model configurations. Each cell reports the score in percent, followed by its within-overlap rank in parentheses (1 is best).

(a) ToolUse

Model	ToolUse	BFCL-V4	MCP-Mark	MCP-Atlas
Kimi-K2.6-Think	62.05 (2)	71.3 (2)	55.9 (4)	66.6 (5)
GLM-5.1-Think	54.62 (4)	70.9 (3)	57.5 (2)	71.8 (4)
DeepSeek-V4-Pro-Think-Max	63.29 (1)	70.6 (4)	57.1 (3)	73.6 (3)
Qwen-3.7-Max-Think	55.24 (3)	75.0 (1)	60.8 (1)	76.4 (1)
Qwen-3.6-Plus-Think	30.33 (5)	68.9 (5)	48.2 (5)	74.1 (2)

(b) Terminal

Model	Terminal	TBench 2.0
Kimi-K2.6-Think	72.73 (1)	66.7 (3)
GLM-5.1-Think	52.02 (5)	63.5 (4)
DeepSeek-V4-Pro-Think-Max	66.16 (3)	67.9 (2)
Qwen-3.7-Max-Think	68.69 (2)	69.7 (1)
Qwen-3.6-Plus-Think	52.53 (4)	61.6 (5)

Table 20 reports the resulting normalized rank gaps for the matched configurations.

Table 20: Normalized rank gaps between agent and simulator evaluations computed from the within-overlap ranks in Table 19. Positive gaps indicate a relatively higher agent-benchmark rank; negative gaps indicate a relatively higher environment-simulation rank. Tool Avg. is the unweighted mean over the three ToolUse-matched benchmarks.

Model	ToolUse				Terminal
	BFCL-V4	MCP-Mark	MCP-Atlas	Avg.	TBench 2.0
Kimi-K2.6-Think	0.00	-0.50	-0.75	-0.42	-0.50
GLM-5.1-Think	+0.25	+0.50	0.00	+0.25	+0.25
DeepSeek-V4-Pro-Think-Max	-0.75	-0.50	-0.50	-0.58	+0.25
Qwen-3.7-Max-Think	+0.50	+0.50	+0.50	+0.50	+0.25
Qwen-3.6-Plus-Think	0.00	0.00	+0.75	+0.25	-0.25

The rank gaps differ across the five selected configurations. Qwen-3.7-Max-Think ranks third on WorldSimBench ToolUse but first on all three matched tool-agent benchmarks, producing a ToolUse average gap of $+0.50$. It also ranks second on WorldSimBench Terminal and first on TerminalBench 2.0, yielding a gap of $+0.25$.

DeepSeek-V4-Pro-Think-Max shows the opposite pattern on ToolUse: it ranks first on WorldSimBench ToolUse but lower on the three tool-agent benchmarks, producing an average gap of -0.58. Kimi-K2.6-Think also ranks relatively higher on environment simulation, with gaps of -0.42 for ToolUse and -0.50 for Terminal. GLM-5.1-Think has gaps of $+0.25$ in both settings.

Qwen-3.6-Plus-Think has a ToolUse average gap of $+0.25$, driven by its second-place MCP-Atlas rank versus fifth place on WorldSimBench ToolUse; it ranks fifth on both BFCL-V4 and MCP-Mark. Its Terminal gap is -0.25, reflecting fourth place on WorldSimBench and fifth place on TerminalBench 2.0. This contrast illustrates how a model's relative agent–simulator ranking can vary across benchmarks.

Overall rank agreement can coexist with model-specific rank differences. Because the analysis contains only five matched configurations, the possible gap values are coarse and the results should be treated as descriptive diagnostics rather than evidence of causal capability transfer.

C.4 Fully Annotated Model-size Scaling Plot

Figure 9 extends the 12-configuration comparison in Figure 5 to all 21 configurations in the full analysis set (11 with thinking and 10 without thinking) and labels every point. Scores are the five-environment macro-averages reported in Table 4. Model size denotes total parameters, including all experts for MoE models. Configurations without disclosed parameter counts are excluded. We also omit the GLM-5.3 variants because Table 4 contains no matched configuration without thinking. On the log-parameter scale, the full-set Pearson correlations are $r = 0.92$ with thinking ($n = 11$) and $r = 0.83$ without thinking ($n = 10$). The positive association therefore remains in the larger comparison, although its magnitude for configurations with thinking is lower than in the compact comparison in Figure 5.

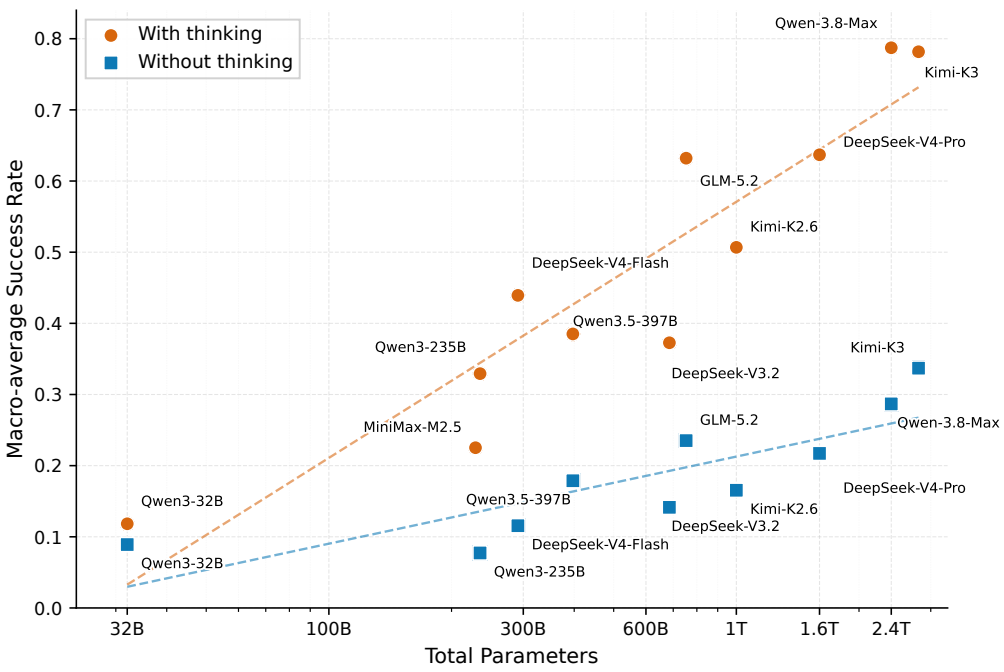


Figure 9: Fully annotated model-size comparison for 21 configurations: 11 with thinking and 10 without thinking. The 12 configurations in Figure 5 form a subset.

D Failure Analysis

Diagnostic sample selection. After validity auditing, the diagnostic sample contains 698 prediction runs: 399 from ToolUse, 102 from Game, 34 from Terminal, 65 from Web, and 98 from Search. The error-category distributions are computed over this sample, with repeated predictions on the same task counted as separate runs.

The first case uses the trajectory and final prediction to illustrate a long-horizon state-reconstruction failure. The remaining cases compare the recorded rationale with the submitted answer. These rationales are observable explanations rather than guaranteed faithful representations of latent model computation.

D.1 Error-Attribution Protocol

Each eligible prediction run receives one primary error category after validity auditing. The annotation must be grounded in the task, trajectory, reference, and prediction; the model-generated rationale may serve as supporting evidence. The same prompt is used across ToolUse, Game, Terminal, Web, and Search. Up to three candidates may be returned for audit, but only the highest-confidence primary label contributes to the distribution in Figure 7. The full template appears in Appendix F.6.2 (Prompt 24).

D.2 CREDIT REPORT: LONG-HORIZON STATE RECONSTRUCTION

Task. This ToolUse example asks the model to reconstruct a credit report after record updates, disputes, and credit inquiries. The final tool call is:

```
generate_credit_report(
subject_id="SUBJ_TARGET_001",
current_timestamp=1027)
```

Key evidence. Table 21 summarizes the state-changing and final-answer-relevant events needed to reconstruct the reference output.

Table 21: Critical tool context for the credit report failure case. We show only state-changing or final-answer-relevant events.

Stage	Key tool events	Effect on final report
Subject setup	`create_subject( SUBJ_TARGET_001, base_score=700)`	Initializes the target subject with base score 700.
Initial records	`REC_TARGET_001: loan, +10` `REC_TARGET_002: default, -50`	Adds one positive loan record and one negative default record.

Early hard pulls	`REC_TARGET_003`
	`REC_TARGET_004`
	`REC_TARGET_005`
		Each hard pull silently creates an active inquiry record with impact -2.

Dispute reversal	`file_dispute(..., REC_TARGET_002)` `resolve_dispute(..., rejected)`	The default record changes from active to disputed and then back to active, restoring its negative impact.
Freeze reversal	`set_credit_freeze(..., true)` `set_credit_freeze(..., false)`	The target subject is ultimately active, so the final report should succeed.
Accepted dispute	`file_dispute(..., REC_TARGET_002)` `resolve_dispute(..., accepted)`	The default record is finally archived and should no longer affect the score.
Facility side effect	`evaluate_facility_application( ..., new_loan_record_id= REC_TARGET_008)`	A successful application implicitly creates a new active loan record with impact -10.
Decoy-like target record	`inquire_credit_history( subject_id=SUBJ_TARGET_001, inquiry_record_id=REC_DECOY_006, is_hard_pull=true)`	Although the record ID looks like a decoy, it belongs to the target subject and must be counted as an active hard pull.

Final records	`REC_TARGET_010: repayment, +20`
	`REC_TARGET_011: inquiry, -2`
	`REC_TARGET_012: loan, +10`
		Adds late active records that affect both the score and the active-record count.

Correct result. The reference score is computed from the active-record impact sum:

$$\begin{aligned}&+10-2-2-2+20+10\\&-10-2-2+20-2+10\\&=+48.\end{aligned}$$

There are six active hard pulls in the last 30 time units, at timestamps 1008, 1009, 1012, 1020, 1023, and 1025. Therefore, the hard-pull penalty is:

$$(6-2)\times 5=20.$$

The final score is:

$$700+48-20=728.$$

Model prediction. The model predicts a higher score and fewer active records, indicating three coupled failure modes.

Error mechanism. The decoy-like record ID is one possible source of confusion, although the record belongs to the target subject. The correct score requires counting all six active hard pulls and applying the resulting penalty. The archived default record must be excluded after the accepted dispute. The example requires tracking record ownership, active status, and cumulative penalties across the trajectory.

D.3 Reasoning Self-Correction Diagnostics

For runs with incorrect submitted answers, we examine whether the recorded rationale identifies a conflict, resolves it, and reflects the correction in the answer. We analyze only substantive model-generated rationales that can be audited from their explicit content.

We exclude evaluator artifacts, semantically equivalent predictions, under-specified cases, and runs without a substantive observable rationale. For an environment e, let $\mathcal{E}_e$ denote the remaining genuine reasoning errors. We define three nested subsets:

- $\mathcal{D}_e$ contains runs whose rationale explicitly identifies a conflict between a candidate conclusion and the supplied evidence or rules. Merely considering alternatives or expressing uncertainty does not count as self-detection.
- $\mathcal{R}_e \subseteq \mathcal{D}_e$ contains detected runs in which the rationale resolves the identified conflict and retains a reference-consistent conclusion as its final endorsed conclusion.
- $\mathcal{P}_e \subseteq \mathcal{R}_e$ contains recovered runs in which the submitted answer reflects the corrected conclusion for the detected issue. This issue-level criterion does not require the remainder of the answer to be free of independent errors.

We define the Self-Detection Rate (SDR), Recovery Rate (RR), and Correction Propagation Rate (CPR) as

$$\mathrm{SDR}_e = \frac{|\mathcal{D}_e|}{|\mathcal{E}_e|}, \qquad \mathrm{RR}_e = \frac{|\mathcal{R}_e|}{|\mathcal{D}_e|}, \qquad \mathrm{CPR}_e = \frac{|\mathcal{P}_e|}{|\mathcal{R}_e|}.$$

A metric is reported as N/A when its denominator is zero. Table 22 reports counts and rates separately for each environment.

Table 22: Observable self-correction diagnostics in the audited failure-analysis sample. Each cell reports the rate followed by its numerator and conditional denominator.

Environment	SDR	RR	CPR
ToolUse	18.6% (53/285)	52.8% (28/53)	46.4% (13/28)
Game	30.0% (30/100)	46.7% (14/30)	92.9% (13/14)
Terminal	44.8% (13/29)	61.5% (8/13)	75.0% (6/8)
Web	15.4% (10/65)	80.0% (8/10)	87.5% (7/8)
Search	13.6% (11/81)	27.3% (3/11)	100.0% (3/3)

In the audited sample, explicit self-detection occurs in only 13.6% to 44.8% of eligible errors across environments. Conditional on self-detection, recovery ranges from 27.3% on Search to 80.0% on Web. ToolUse has the lowest observed CPR: 13 of 28 recovered conclusions appear in the submitted answer. This gap is consistent with failures in which a corrected conclusion is omitted from the final answer or replaced by an earlier reasoning branch.

CPR is based on only 28, 14, 8, 8, and 3 recovered cases in ToolUse, Game, Terminal, Web, and Search, respectively. More generally, these diagnostics characterize the audited failure-analysis sample rather than benchmark-wide self-correction probabilities.

D.4 Reasoning-Trace Failure Cases

Here, a conflict is one explicitly identified in the recorded rationale. An inconsistency between the rationale and answer occurs when the submitted answer differs from the conclusion stated in the rationale.

We examine five illustrative failures with recorded `reason` fields. All five cases below are generated by DeepSeek-V4-Flash in the no-thinking configuration.

Case 1: Pharmacy inventory aggregation.
ID: `504b7ffad96b41bf9c50fdd0fcf5311c`.

Task. Given a sequence of batch additions, medicine purchases, refunds, and expiration dates, the model must predict the result of `search_available_medicines` on February 11, 2024. The result requires excluding expired batches and applying all purchase and refund effects to the remaining batch-level stock.

Key evidence. The rationale excludes the expired M1 and M3 batches, subtracts the purchases that remain valid, restores the purchases that were later refunded, and recomputes the remaining stock batch by batch. In particular, it derives an M1 stock of $8+5+10 = 23$ and correctly reduces the stocks of M3, M4, and M5 from 10 to 9. The rationale states that M1 through M5 equal $(23, 10, 9, 9, 9)$ as its final result.

Correct result. The valid aggregate stocks are M1 through M5 equal $(23, 10, 9, 9, 9)$.

Model prediction. The structured output instead reports M1 through M5 equal $(26, 10, 10, 10, 10)$.

Error mechanism. The submitted JSON reports $(26, 10, 10, 10, 10)$, which differs from the final stock calculation stated in the rationale. We classify this case as *correction not propagated*, with an explicit *inconsistency between the rationale and answer*.

Case 2: Disaster-recovery synchronization.
ID: `87cee93286704e098c88099165e8fd71`.

Task. The final action calls `sync_dr_plan(plan_id=DP3)` after primary site P1 has gone offline. Backup B11 for volume V5 remains available at logical time 140. The disaster-recovery site D1 is online and has exactly 50 units of free storage, equal to the size of V5.

Key evidence. The model initially assumes that taking primary site P1 offline makes backup B11 unavailable and therefore predicts that synchronization will fail. It then checks this assumption against the actual state and rules. B11 is still marked `AVAILABLE` at logical time 140, and the synchronization procedure searches for an available backup at the primary site without requiring that site itself to be online. The model also tracks the expiration of the earlier replica and recalculates the storage usage of D1, correctly finding that D1 has exactly 50 free units, which is sufficient for the 50-unit volume.

Correct result. The synchronization succeeds, DP3 becomes `READY`, its last synchronization time becomes 140, and backup `B11_replica_140` is created.

Model prediction. The model returns `no_available_backup`, claiming that no usable backup exists at the primary site.

Error mechanism. The rationale concludes that synchronization should succeed. Nevertheless, the submitted answer still reports `no_available_backup`. The submitted error matches the earlier hypothesis, not the rationale's final conclusion. We classify this as *inconsistency between the rationale and answer*. The `CONFLICT` value in the submitted API error is an environment error code and is unrelated to the reasoning conflict analyzed here.

Case 3: Edge-cache eviction.
ID: `95368902a8704317a43abc9df60c4129`.

Task. The model must predict the result of deploying `Func_Decoy` on `Node_C`. The node has 272 units of available memory, while the new function requires 512. Three unpinned and evictable caches occupy 75, 80, and 85 units, respectively.

Key evidence. The model correctly reconstructs Node C's memory state. The three eligible caches occupy 75, 80, and 85 units, which sum to exactly 240. Consequently, no subset of only two caches can free enough memory; all three must be evicted.

Correct result. The deployment requires freeing exactly $512 - 272 = 240$ units. Consequently, all three caches must be evicted: `Cache_Noise_2`, `Cache_Noise_3`, and `Cache_Noise`.

Model prediction. The deployment is predicted to succeed, but the output lists only `Cache_Noise` and `Cache_Noise_2` as evicted.

Error mechanism. The rationale explicitly reaches this conclusion and names all three caches. The structured prediction nevertheless lists only `Cache_Noise` and `Cache_Noise_2`, omitting `Cache_Noise_3`. Those two caches free only $75 + 80 = 155$ units, leaving 427 units available, which is still below the required 512. The predicted successful deployment is therefore inconsistent with the state changes listed in the same answer. We classify this as an *incomplete structured projection*: the resource calculation is correct, but the complete set of required state changes is not written into the output.

Case 4: Game-state transition.
ID: `aa8daabcda9645339baa9fea0b5c3a1a`.

Task. Given a long trajectory of actions and resulting 10×10 boards, the model must infer the next board after Player 2 acts at position $[3, 1]$. The prediction must apply the game transition rule to the current board.

Key evidence. The model first tries to interpret the action as an ordinary placement at $[3, 1]$. It then notices that this cell already contains the value 23, so the action cannot be explained by simply placing a token in an empty cell. It also inspects the surrounding cells and observes that the local configuration does not support the transition pattern it initially expected. These observations invalidate its initial local-transition hypothesis and signal that a different rule must be inferred from the earlier trajectory.

Correct result. Relative to the preceding board, the reference transition changes five local cells: $[2, 1] : 0 \to 21$, $[3, 0] : 0 \to 21$, $[3, 1] : 23 \to 0$, $[3, 2] : 0 \to 21$, and $[4, 1] : 22 \to 23$.

Model prediction. The predicted board changes only $[3, 2]$ from 0 to 21 and leaves the remaining required transitions unchanged.

Error mechanism. Instead, it explicitly decides to "mimic" an earlier action at $[2, 4]$ because that action appears superficially similar. The two states do not satisfy the same local conditions, and the copied pattern changes only one of the five required cells. The rationale appeals to a similar earlier action without establishing that the same transition conditions hold. We classify it as *unsupported analogy*.

Case 5: Terminal execution of Gomoku tests.
ID: `891d5a3cf7d04f89a6a97cc60b25ec2b`.

Task. The final shell command lists `/app`, inspects a `GomokuGame` object, and runs three move sequences labeled H, V, and D. The model must reproduce the complete stdout of the chained command.

Key evidence. The rationale initially hypothesizes that the final move at $(10, 14)$ raises an `IndexError`. It then checks the array bounds and correctly notes that a 15×15 board has valid row and column indices from 0 through 14. The index $(10, 14)$ is within bounds for this board. The rationale also observes that the attempted source-code replacement failed, that the original file remained unchanged, and that previous executions provide no evidence for such an exception.

Correct result. The stdout contains the directory listing, public attributes, board type, and three False results. The execution log separately records exit code 0.

Model prediction. The prediction omits the directory listing and several object attributes, then fabricates an `IndexError` claiming that column 14 is outside a board dimension of size 15.

Error mechanism. Despite accumulating several pieces of evidence against the exception, the model ultimately chooses to trust the imagined traceback and includes it in the prediction. The submitted exception is inconsistent with the bounds stated in the rationale and with the available execution evidence.

The prediction contains a second, independent error. The final command chains several operations, so the directory listing, class information, method list, and board type are printed before the game tests complete. Even if a later Python process had failed, this earlier stdout would still remain in the command output. The prediction omits much of this already-determined output. We therefore classify the case as a combination of *unresolved contradiction*, *hallucinated execution evidence*, and *incomplete command-chain simulation*.

Summary. The five cases illustrate two observable patterns. In Cases 1 through 3, the rationales state results consistent with the reference, but the submitted answers differ from those results. In Cases 4 and 5, the rationales do not reach a reference-consistent conclusion. These examples illustrate errors in stated conclusions and inconsistencies between rationales and submitted answers; they do not establish how often either pattern occurs across the benchmark.

E Dataset Schema and Field Definitions

This section documents the released data format. Field definitions are shared across environments; Table 23 records their environment-specific locations.

E.1 Dataset Schema

Every instance has four common top-level fields: `system_prompt`, `user_prompt`, `answer`, and `id`. Table 23 lists only additional top-level fields and fields nested inside `metadata`.

Table 23: Environment-specific fields beyond the four common top-level fields.

Domain	Additional top-level fields	Fields inside metadata
ToolUse	`metadata`	`domain`, `step_num`, `question_api`
Terminal	`metadata`	`domain`, `step_num`, `standard_text`, `question_command`
Game	`metadata`	`domain`, `step_num`, `num_players`, `board_size`, `question_action`
Web	`domain`, `steps_num`, `eval_values`	–
Search	`domain`, `step_num`, `answer_keywords`, `metadata`	`num_actions`

E.2 FIELD DEFINITIONS

Table 24: Description of common and domain-specific fields in the benchmark schema.

Field	Description
`system_prompt`	Global system instruction, including task rules, environment descriptions, interaction formats, or constraints that the model should follow.
`user_prompt`	The concrete task input, usually containing the interaction history, state transitions, observations, and the final question.
`answer`	The reference outcome for the prediction task, such as a tool return, command output, resulting board state, web observation, or search answer. Correctness is determined by the domain-specific verification protocol.
`id`	The instance identifier. For Terminal, checker selection follows Appendix B.7.
`metadata`	Structured auxiliary information for statistical analysis, data filtering, difficulty analysis, and task verification.
`domain`	The specific scenario or sub-domain of the example. It is used to group instances by application setting and analyze model performance across different scenarios.
`step_num`	The number of interaction steps in the trajectory. This field measures trajectory length; answer-dependency span is annotated separately.
`question_api`	The final tool/API call whose return value must be predicted.
`standard_text`	Task-specific evaluation criteria for Terminal outputs. The value `same` selects the shared text-comparison checker.
`question_command`	The final shell command whose output or resulting file state must be predicted from the preceding command history.
`num_players`	The number of players in the game environment.
`board_size`	The dimensions of the game board.
`question_action`	The held-out final game action supplied to the model; the model predicts the resulting board state.
`steps_num`	The number of browser interaction steps in a Web trajectory. This field is the Web counterpart of `step_num`.
`eval_values`	The predefined component and value requirements used to evaluate a Web prediction. A correct prediction must satisfy every listed requirement.
`answer_keywords`	The task-specific required keywords used by the deterministic Search verifier. A Search prediction is correct only when all required keywords are present.
`num_actions`	The number of Search actions, including paired query and result interactions, in the trajectory. This field is stored inside `metadata`.

F FULL PROMPT TEMPLATES

This section collects the full generation, evaluation, and diagnostic annotation templates, together with the toolkit specification. Each template is introduced by its purpose and input/output contract. Construction models and decoding settings are reported in Appendix B.8; scoring rules are specified in Appendix A.1. The index below links to each template group.

F.1.1 Tool Design Prompt . . . 36
F.1.2 Toolkit Documentation Schema . . . 37
F.1.3 Code Generation . . . 39
F.1.4 Final Tool Selection Prompt . . . 40
F.1.5 Story Blueprint Design Prompt . . . 41
F.1.6 Chapter Flow Design Prompt . . . 42
F.2.1 Terminal Task Generation Prompt . . . 44
F.2.2 Terminal Checker Generation Prompt . . . 45
F.3.1 Game Move Simulation Prompt . . . 47
F.4.1 Web Environment Synthesis Prompts . . . 47
F.4.2 Web Trajectory Generation Prompts . . . 48
F.4.3 Web Prediction and Evaluation Prompts . . . 50
F.5.1 Search Environment Synthesis Prompts . . . 51
F.5.2 Search Trajectory and Dataset Construction Prompts . . . 52
F.5.3 Search Prediction and Evaluation Prompts . . . 54
F.6.1 Final-Answer Dependency Annotation . . . 55
F.6.2 Error Attribution . . . 56
F.6.3 Structural Task-Difficulty Annotation . . . 58

F.1 TOOLUSE GENERATION PROMPTS

F.1.1 TOOL DESIGN PROMPT

Purpose. Specify a compact, stateful tool interface for later code generation. **Input.** A scenario name and description, with system-level design constraints. **Output.** A Markdown tool-interface specification with reproducible behavior and non-trivial state dependencies. Prompts 1 and 2 give the system and user templates.

Prompt 1: System prompt for ToolUse tool-interface.

```
You are a senior backend architect and a world-model simulation environment designer. Your
     task is not to design a large and complete tool surface. Instead, you should design a
     tool-interface document with a small number of tools, deep state dependencies,
     reproducible behavior, and non-trivial reasoning difficulty. The document should directly
      support later generation of an executable domain_api.py file.

Always follow these principles:
1. Use few tools with strong state dependencies. Prefer a small number of high-value tools
     over many shallow CRUD operations.
2. Ensure reproducible results. Every tool should return a stable dict; business failures
     should be represented by return values rather than uncaught exceptions.
3. Make the difficulty come from hidden state recovery. The final high-value tools should
     depend on several current-state predicates jointly.
4. Avoid near-answer leakage. Do not design one or two read tools that directly reveal the
     final answer.
5. Make the document executable. The tool-interface document must be precise enough to guide
     the generation of runnable domain_api.py code.
```

Prompt 2: User prompt for ToolUse tool-interface.

```
Design a "few tools but high difficulty" tool-interface document for the following scenario.

Part 1: Scenario input.
- Scenario name: {scenario_name}
- Scenario description: {scenario_description}

Part 2: Core constraints.
This is not a RESTful service. It is a stateful simulation environment implemented as a
     Python class.
- The core class name must be DomainAPI.
- All business methods must be @classmethod.
- Do not use _INIT_DATA. The class should maintain _data as the only data store, and
     reset_data() should reset _data to an empty structure.
- Clearly define the data schema for each entity at the top of the class, including field
     names, types, optional or required status, enum values, and defaults.
```

```
- Each core entity type must provide a create tool, so the environment can be built from an
    empty state.
- Every business tool must return a dict.
- Business failures must return success=false, an error_code from INVALID_ARGUMENT, NOT_FOUND,
     CONFLICT, or API_ERROR, a stable English snake-case error_key, and a readable English
    error_message.

The design should focus on compact but difficult state dynamics.
- Design 2--4 core entities and 12--20 business tools. More tools are allowed only for
    genuinely complex scenarios.
- Merge similar operations instead of creating many small setter-like tools.
- Include at least two high-value final-step tools. Each final-step tool must depend on at
    least three current-state predicates and at least one cross-entity dependency.
- Include at least three of the following mechanisms: hidden side effects, cascade effects,
    reversible state transitions, confusion among multiple same-type instances, and history-
    dependent filtered list tools.
- Read tools may exist, but they should provide local information, filtered views, or partial
     evidence rather than directly exposing the final answer.
- Failure branches must correspond to single root causes. Even under the same error_code, use
     different error_key values for different causes.
- Each write tool should have at least two independent error_key failure branches. Highly
    state-dependent tools should have at least three.
- Each error_message must include dynamic context values, such as current balance, capacity,
    or conflicting entity ID.
- The tool set must form a complete data loop: create entities, query local information,
    perform state transitions, and execute cross-entity operations from an initially empty
    state.
- Each create tool must return the full created entity, including the generated ID, so later
    steps can reference it.
- All code-related content, including tool names, field names, status values, error_key
    values, and example values, must be in English.

Part 3: Required output format.
Strictly output the following Markdown structure.
- Tool-Interface Design Document: {scenario_name}
- Data Schema: describe the _data structure in a JSON-like style, including entity fields,
    types, required or optional status, enum values, and defaults. State that reset_data()
    resets _data to an empty structure.
- Hardness Plan: list the core entities, explain why the domain is hard with few tools,
    provide 2--4 final-step candidate tools, and for each candidate tool describe decisive
    predicates, hidden side effects or cascades, reversible transitions if any, and near-
    answer leakage to avoid.
- Tool Details: for each tool, provide its purpose, parameters, success return fields,
    failure cases, and state logic. Failure cases should use the format ERROR_CODE |
    error_key=....
- Suggested Tool Set Summary: provide a compact table with the tool name, type (READ or WRITE)
    , why it is kept, and why similar operations are not split into more tools.
```

F.1.2 TOOLKIT DOCUMENTATION SCHEMA

Purpose. Make generated tool documentation machine-readable under a project-specific schema. **Input.** A tool's interface, parameters, return values, and state-update logic. **Output.** A single JSON documentation object with the fixed keys in Prompt 3: `name`, `description`, `parameters`, `implementation`, `returns`, `failure_cases`, and `notes`. The requirements below specify validation constraints.

Prompt 3: Toolkit template for ToolUse.

```
{
  "name": "function_name",
  "description": "Function description",
  "parameters": {
    "type": "object",
    "properties": {
      "field_a": {
        "type": "string",
        "description": "Field description"
      },
      "field_b": {
        "type": "array",
        "description": "Array field example",
        "items": {
          "type": "object",
          "properties": {
            "sub_field": {
              "type": "integer",
              "description": "Nested field description"
```

```
            }
          },
          "required": ["sub_field"],
          "additionalProperties": false
        }
      }
    },
    "required": ["field_a"],
    "additionalProperties": false
  },
  "implementation": [
    "Key business logic",
    "State-machine rules",
    "Side effects, dependencies, filters, and default inheritance",
    "Conditions that determine success and failure"
  ],
  "returns": {
    "success": {
      "type": "object",
      "properties": {
        "success": {
          "type": "boolean",
          "const": true,
          "description": "The call succeeds"
        },
        "result_field": {
          "type": "string",
          "description": "Example success field"
        }
      },
      "required": ["success", "result_field"],
      "additionalProperties": false
    },
    "failure": {
      "type": "object",
      "properties": {
        "success": {
          "type": "boolean",
          "const": false,
          "description": "The call fails"
        },
        "error_code": {
          "type": "string",
          "enum": ["INVALID_ARGUMENT", "NOT_FOUND", "CONFLICT", "API_ERROR"],
          "description": "Error category"
        },
        "error_key": {
          "type": "string",
          "description": "Stable error identifier"
        },
        "error_message": {
          "type": "string",
          "description": "Human-readable error message"
        }
      },
      "required": ["success", "error_code", "error_key", "error_message"],
      "additionalProperties": false
    }
  },
  "failure_cases": [
    "ERROR_CODE | error_key=stable_error_key | unique trigger | explanation"
  ],
  "notes": [
    "Important dependencies or side effects"
  ]
}
```

The toolkit comments must satisfy the following requirements:

1. The JSON object must be complete and parseable. The generated documentation must not contain pseudo-JSON, extra explanatory text, comments, trailing commas, or missing double quotes.
2. The `parameters`, `returns.success`, and `returns.failure` sections should follow a JSON-Schema-like style and expand all important fields down to primitive types.
3. For every object or array field in `parameters` and `returns.success`, the nested structure should be recursively specified. Objects should include `properties`, `required`,

and `additionalProperties: false` when field boundaries are clear. Arrays must include `items`; object items must also expand their internal fields.

4. The `implementation` field must describe state transitions, dependencies, automatic side effects, filtering conditions, default inheritance, and the conditions that determine success and failure.
5. The `returns.failure` schema must always include `success`, `error_code`, `error_key`, and `error_message`.
6. Each entry in `failure_cases` must correspond to exactly one root cause. Different business causes under the same `error_code` must use different stable English snake-case `error_key` values.
7. All machine-consumed schema content must be in English, including tool names, field names, schema keywords, protocol values, `error_code`, `error_key`, state values, and action values. Human-facing descriptions, implementation notes, and comments should also be written in English.
8. The `type` field may only use standard JSON Schema type names: `string`, `number`, `integer`, `boolean`, `null`, `object`, and `array`. Aliases such as `int`, `float`, `bool`, `str`, `dict`, or `list` are not allowed. Union types should use standard JSON Schema notation, such as `"type": ["object", "null"]`.

F.1.3 CODE GENERATION

Purpose. Implement the designed ToolUse environment. **Input.** The domain description, tool-interface design, reference code template, toolkit schema, and domain-specific implementation instructions. **Output.** A runnable `domain_api.py` environment class. Prompts 4 and 5 give the complete templates.

Prompt 4: System prompt for ToolUse code generation.

```
You are a professional Python backend engineer. Generate a stateful simulation environment
    class according to the provided scenario, tool-interface design document, reference
    template, and additional implementation instructions. The output should be a complete
    runnable domain_api.py file for simulating tool calls and generating training data.
```

Prompt 5: User prompt for ToolUse code generation.

```
Part 1: Task input.
- Domain name: {domain_name}
- Domain description: {domain_desc}
- Reference tool template: {tmp_code}
- Tool-interface design document: {design}
- Required toolkit comment format: {toolkit_rule}
- Additional domain-specific code instructions: {domain_code_instruction}

Part 2: Code generation requirements.
- Strictly follow the tool-interface design document and generate directly runnable
    domain_api.py code.
- Do not use _INIT_DATA. The class itself must maintain _data as the only data store, and
    reset_data() must reset _data to an empty structure, such as empty dictionaries or lists.
- Clearly define the data schema for each entity at the top of the class using comments or
    docstrings, including field names, types, required or optional status, enum values, and
    defaults.
- Each core entity type must have a corresponding create tool. Create tools should support
    setting different initial states and must return the full created entity, including the
    generated ID.
- The full tool set must form a complete data loop: create entities, query local information,
     perform state transitions, and execute cross-entity operations from an initially empty
    state.
- Keep the business tool surface compact, typically 12--20 tools unless the scenario is
    genuinely complex. Do not add extra convenience tools only for broad CRUD coverage.
- Every business tool must return a dict. Business failures should not be represented as the
    main path by uncaught exceptions.
- Successful returns must follow {"success": true, ...}. Failure returns must follow {"
    success": false, "error_code": "...", "error_key": "...", "error_message": "..."}.
- Implement at least two highly state-dependent tools. Their results must depend on at least
    three current-state predicates and include cross-entity dependencies.
- Include at least three strong world-model mechanisms in the code, preferably hidden side
    effects, cascade effects, reversible state transitions, confusion among multiple same-
    type instances, and filtered list behavior.
```

```
- Avoid read tools that directly expose the answer. Read tools may exist, but a single read
    call should not reveal all key hidden state needed for the final answer.
- List tools should return stable structures, preferably with items and total. State-
    transition tools should return key changes, and delete or batch tools should return
    explicit result fields.
- Every failure branch must be unique, stable, and reproducible. Different business causes
    under the same error_code must use different error_key values.
- Each write tool must implement at least two independent error_key failure branches. Highly
    state-dependent tools must implement at least three.
- Each error_message must include dynamic context values, such as current balance, capacity,
    or conflicting entity ID.
- Real code logic, docstrings, and toolkit comments must be strictly consistent with each
    other.
- The generated code must not contain Chinese. Comments, docstrings, toolkit comments, and
    error messages must all be in English.
- The tool set should not be too small. Each environment should contain rich data logic so
    that later simulation can generate diverse data.
- Every tool must include a toolkit comment that follows the required toolkit comment format.

Part 3: Output format.
Directly output the full domain_api.py code. The code must be wrapped with <code> and </code>
     tags, and must not be wrapped in Markdown code fences.
```

F.1.4 Final Tool Selection Prompt

Purpose. Select candidate final queries whose answers require prior state changes. **Input.** The generated API code and tool documentation. **Output.** Candidate final tools and selection guidance. Selected queries must have deterministic results and avoid trivial leakage from recent read calls. Prompt 6 gives the template.

Prompt 6: User prompt for final ToolUse tool selection.

```
Part 1: Selection objective.
Select tools that are suitable as the final step for constructing long-horizon consistency
    samples. In each sample, the model is given a sequence of historical tool calls and
    returns, followed by one final tool call, and must infer the return value of that final
    call.

Only keep tools that satisfy the following conditions:
1. The output must be uniquely determined by the current input, historical tool inputs and
    outputs, and the current system state. It must not depend on randomness, external
    environments, or hidden variables that are not exposed through the history.
2. As a final step, the result should depend on a joint judgment over multiple hidden current
     states. Prefer tools requiring at least three state predicates and at least one cross-
    entity dependency.
3. Do not select tools whose answers can be directly exposed by a nearby read. If the result
    is usually revealed by one recent get_* or list_* call, the tool is not preferred.
4. Prefer tools that stably produce high-value outcomes, such as error branches (NOT_FOUND or
     CONFLICT), filtered lists, cascade-effect results, or write tools constrained by a state
     machine. Do not prioritize shallow success-only tools.

reset_data must not be included in the suggestions. The number of candidates does not need to
     be large; prioritize truly high-quality candidates. The suggestions do not need to
    balance READ and WRITE tools evenly, but they should cover at least two different final-
    step patterns.

Part 2: Input.
- Tool code: {api_code}
- Tool list: {api_md}

Part 3: Required JSON schema.
Output only a structured JSON object that can be directly written into prompts.py as
    last_api_suggestions. Each key must be a tool name, and each value must have the
    following structure:
- why_final: one sentence explaining why this tool is suitable as the final step.
- preferred_outcomes: a list of preferred outcomes, using values such as CONFLICT, NOT_FOUND,
     FILTERED_LIST, CASCADE_EFFECT, or SUCCESS_AFTER_LONG_HISTORY.
- decisive_predicates: at least three concrete and checkable current-state predicates that
    determine the final result.
- required_history_apis: the minimum key tools that should appear in the history so the final
     result is inferable. Only use real tool names.
- preferred_mechanisms: at least two mechanisms, such as implicit_side_effect, cascade_effect,
     reversible_transition, similar_entity_confusion, or filtered_read.
- forbidden_recent_reads: concrete nearby read patterns that should not appear in the last
    8--12 steps because they would directly leak the answer.
- history_hint: one sentence explaining how to construct a decisive history for this tool.
- conflict_setup_hint: one sentence explaining which prior state conditions can trigger a
    success=false branch such as CONFLICT or NOT_FOUND.
```

```
Part 4: Hard constraints and output format.
- preferred_outcomes should prioritize CONFLICT, NOT_FOUND, FILTERED_LIST, CASCADE_EFFECT,
    and SUCCESS_AFTER_LONG_HISTORY.
- decisive_predicates must be specific and checkable, not vague descriptions.
- required_history_apis must contain only real tool names.
- preferred_mechanisms must contain at least two mechanisms.
- forbidden_recent_reads must name concrete target-entity read patterns that would leak the
    answer.
- conflict_setup_hint must clearly state the prior state conditions needed to trigger the
    business failure.
- The JSON must be a valid object whose keys are tool names and whose values follow the
    schema above.

The final answer must use tag format:
<apiDict>
{structured JSON object}
</apiDict>
```

F.1.5 Story Blueprint Design Prompt

Purpose. Plan historical state changes backward from the selected final tool's intended outcome. **Input.** Scenario rules, the toolkit, final-tool selection, and requested planning constraints. **Output.** Blueprints specifying decisive earlier evidence, state changes, and distractors that prevent direct retrieval from recent context. Prompts 7 and 8 give the templates.

Prompt 7: System prompt for ToolUse story blueprint.

```
You are an expert in business-scenario tool simulation design. Your core responsibility is to
     design backward state-trap blueprints based on the scenario rules, the available toolkit
     tools, and the specified final tool {last_api}.

Each blueprint should not be a full story. Instead, it should first determine the target
    final-step output, then infer the decisive historical events and similar distractors that
     must appear before the final call. Every blueprint must make the final result depend on
    multiple current-state predicates rather than a nearby direct read.

Strict constraint: every tool name mentioned in the blueprint must come from the tool list in
     <toolkit>. Do not invent tools that are not listed.

System input fields.
- Business scenario: {business_scenario}
- Toolkit: {toolkit}
- Final tool: {last_api}
```

Prompt 8: User prompt for ToolUse story blueprint.

```
Part 1: Task objective.
Based strictly on the scenario rules and the available toolkit tool list, design {story_num}
    backward state-trap blueprints. The final tool call in every blueprint must be {last_api
    }.

Part 2: Required fields for each blueprint.
- target_outcome: prefer outcomes from {preferred_outcomes}.
- decisive_predicates: include at least three predicates, preferably from {
    decisive_predicates}.
- required_history_apis: include key tools from {required_history_apis} when possible.
- preferred_mechanisms: include at least two mechanisms from {preferred_mechanisms}.
- forbidden_recent_reads: avoid the listed read-leakage patterns in the final 8--12 steps,
    especially those in {forbidden_recent_reads}.
- blueprint: concisely describe what happens first, what happens next, which states flip,
    which similar entities cause confusion, and why the final result is hard.

Part 3: Additional requirements.
- The blueprints must be genuinely different, not simple name substitutions.
- Prefer difficulty from hidden side effects, cascade effects, reversible state transitions,
    and similar-entity confusion.
- Each blueprint must mention at least two types of business boundary, such as refund,
    cancellation, permission, conflict, or empty filtered result.
- The business boundaries must be consistent with the final hard reasoning requirement: the
    answer should require remote historical state and should not be recoverable from a nearby
     direct read.
- Do not write concrete IDs or detailed parameter values.
- Tool names may remain in English. Explanatory blueprint text may be written in Chinese if
    useful.
- The extra suggestion {suggest} is only a reference and should not be copied mechanically.
```

```
Part 4: Output format.
Return a JSON object with a stories list:
{
  "stories": [
    "Blueprint 1: target_outcome=...; decisive_predicates=...; required_history_apis=...;
     preferred_mechanisms=...; forbidden_recent_reads=...; blueprint=...",
    "Blueprint 2: ...",
    "...",
    "Blueprint {story_num}: ..."
  ]
}

Each blueprint should be no longer than 650 characters. The result should contain at least max
     (story_num // 2, 4) blueprints.
```

F.1.6 CHAPTER FLOW DESIGN PROMPT

Purpose. Expand a story blueprint into a concrete multi-step tool-call plan. **Input.** The target outcome, final tool, selection guidance, blueprint, and requested step count. **Output.** A chapter flow with one tool call per step, explicit parameter dependencies and state effects, and a final question requiring long-range state reasoning. Prompt 9 gives the template.

Prompt 9: User prompt for ToolUse chapter-flow

```
Part 1: Task objective.
Design a {nstep}-step business-process outline. Each step should contain exactly one tool
     call. The goal is not to write a long narrative, but to construct a long state machine
     whose final step is difficult to infer without tracking earlier tool calls and returns.

The target outcome and final tool are sampled before this prompt is issued:
- Target outcome: {target_outcome}
- Final tool: {chosen_final_api}
- Final-tool guidance: {suggestion_text}
- Blueprint direction: {story_text}

Part 2: Final-outcome constraint.
- If target_outcome is CONFLICT or NOT_FOUND, the final call to {chosen_final_api} must
     return success=false and trigger the corresponding business rejection branch. The
     historical steps must construct the exact current state that makes the final arguments
     hit this branch. If the final step returns success=true, the design is invalid.
- If target_outcome is FILTERED_LIST or EMPTY_LIST, the final list result must depend on
     multiple current states across entities and must not be directly inferable from a nearby
     single read.
- Otherwise, the final call must return success=true, but the success condition must depend
     on at least three current-state predicates jointly.

Part 3: Four-phase structure.
- Phase A: Bootstrap, roughly steps 1--10. Start from no initial data. Use write tools to
     build a minimal but complete data loop. Prefer creating two groups of similar entities so
      later steps can introduce similar-entity confusion.
- Phase B: Decisive history, roughly steps 11--{nstep // 2}. These steps must directly
     support the final step. Include at least three decisive predicates, at least one cross-
     entity dependency, at least one hidden side effect or cascade, at least one reversible
     state flip, and at least two similar coexisting entities. Each decisive step must explain
      in design_reason which state it changes and why that state affects the final step.
- Phase C: Confounders, roughly steps {nstep // 2 + 1}--{nstep - 10}. Add highly similar
     distractors rather than unrelated noise. Prefer similar names, overlapping time windows,
     similar products or services, state changes that are later reverted, and write operations
      that look similar to the target chain but are ultimately irrelevant. Read steps are
     allowed only if they do not expose the complete current state of the final target entity.
- Phase D: Leakage guard, steps {nstep - 9}--{nstep - 1}, the first nine steps among the
     final ten. In the final 8--12 steps, do not include direct reads that reveal the current
     state of the final target entity. Reads of unrelated entities, partial or non-decisive
     fields, and writes that change distractor entities are allowed. If a recent read makes
     the final answer directly inferable, the design is invalid.

Part 4: Tool dependency checklist.
Obey the domain-specific tool dependency checklist injected into the prompt. Violating this
     checklist may cause runtime execution failures. The checklist should specify required
     predecessor tools, entity-type constraints, status constraints, and capacity or payment-
     order constraints. For example:
1. create_service must follow create_service_category, and its category_id must point to
     service_categories.
2. create_appointment_type requires a service_id whose service has type='appointment'. If the
      flow also needs a class session, create an additional service with type='class'.
3. create_class_session requires three predecessor entities: service_id, location_id, and
     instructor_id.
```

```
4. A complete appointment-session chain should follow create_service_category ->
     create_service(type='appointment') -> create_appointment_type ->
     create_appointment_session.
5. create_membership_plan requires its product_id to point to a product in the membership
     category, and create_package_plan requires a product in the package category.
6. create_membership requires a client, a membership plan, and a payment method.
     purchase_package requires a client, a package plan, and a payment method.
7. A successful billing transaction can advance the next billing date and activate a
     membership, while a failed billing transaction can suspend it.
8. add_pos_order_item must be called before pay_pos_order; paid or refunded orders cannot
     receive new items.
9. To trigger full-capacity or waitlist behavior, set a small class capacity and create
     bookings until the booking count reaches the capacity.

Part 5: Difficulty and parameter-dependency requirements.
- Parameter-dependency coverage. In chapter_flow, excluding pure reset or bootstrap-only
     steps, at least 70% of write-operation steps must have non-empty param_dependencies. Each
      dependency must point to an earlier step and name a concrete field path. Vague
     descriptions such as "take it from history" are not allowed.
- Decisive steps. At least two Phase B steps must explicitly state in design_reason which
     business-rule branch would be wrong without that step, such as permission failure, time-
     window conflict, resource conflict, insufficient quota, or closed state.
- Boundary and branch coverage. The full chain must contain at least two of the following
     boundary types: refund or partial refund, cancellation or rescheduling constraint,
     resource conflict or mutual exclusion, role or ownership permission, and list filtering
     or empty-result boundary.
- Final question dependencies. The final question's param_dependencies must contain at least
     two keys. Their source steps must be separated by at least 8 steps, forcing long-range
     reasoning rather than reliance on nearby reads.
- Final hard reasoning semantics. The final step's design_reason must be exactly "This is the
      final hard reasoning step.". Its state_effect must state which remote historical states
     must be combined to get the correct answer, without giving concrete IDs or return values.

Part 6: Step fields and hard constraints.
For every non-literal input argument that cannot be inferred from toolkit defaults, add
     param_dependencies in the following style:
{"parameter_name": "from step X (tool_name) result.xxx.id"}

At minimum, include ID-like fields. When necessary, also include important enum, time, amount,
      or status fields.

Each step may additionally include:
- step_role: one of bootstrap, decisive, confounder, guardrail, or final.
- final_relevance: one of direct, indirect, or none.
- state_effect: one sentence describing which current state the step changes.

Hard constraints:
- Do not build difficulty from a direct read of the answer in recent steps.
- Do not make the final step depend on only one rule.
- Do not make confounders completely irrelevant noise; prefer similar-state confusion.
- For write operations, keep the confirmation style in design_reason, such as "User
     confirmation obtained (yes): ...".
- Use English person names and product names.
- Only use tool names that really exist in the toolkit.

Part 7: Output format.
Return strict JSON:
{
  "chapter_flow": [
    {
      "step": 1,
      "api": "reset_if_available",
      "design_reason": "...",
      "step_role": "bootstrap",
      "final_relevance": "indirect",
      "state_effect": "..."
    },
    {
      "step": 2,
      "api": "create_location",
      "design_reason": "User confirmation obtained (yes): ...",
      "param_dependencies": {},
      "step_role": "bootstrap",
      "final_relevance": "indirect",
      "state_effect": "..."
    }
  ],
  "question": {
    "step": "{nstep}",
    "api": "{chosen_final_api}",
    "design_reason": "This is the final hard reasoning step.",
```

```
    "param_dependencies": {},
    "step_role": "final",
    "final_relevance": "direct",
    "state_effect": "..."
  }
}

Every api field must use a tool name listed in the toolkit. Do not invent tools that are not
     available.
```

F.2 Generation Prompts for Terminal

F.2.1 Terminal Task Generation Prompt

Purpose. Expand a TerminalBench seed task into a long-horizon executable trajectory. **Input.** The original task instruction, reference `solve.sh`, and tests. **Output.** An expanded script containing `run_step` calls. The steps are executed in Docker and their outputs recorded; the final inspection command probes the resulting state without embedding the expected answer.

Prompt 10: Prompt for Terminal task-script generation.

```
You are a terminal automation expert. I will give you three files from a TerminalBench task.
     Expand solve.sh into a multi-step execution script.

## Requirements
Extend the base task. The goal is to test long-horizon consistent reasoning, especially
     inference ability. More steps are better; use at least 50 steps.

1. Start with 2-3 environment probe steps, such as listing the working directory, checking key
      tools, or reading existing files.
2. Split each logical phase of solve.sh into separate steps, with one run_step call per step.
3. If solve.sh uses a heredoc for a large code block, treat the entire heredoc as one step.
4. Add verification steps after key operations, such as checking file existence, directory
     structure, or command output.
5. Design verification steps based on the assertions in test_outputs.py.
6. Make the trajectory complex enough to test long-horizon inference. Early steps should
     prepare state and lay clues. Insert unrelated operations and red herrings in the middle.
     The last 10-20 steps before verification must not directly leak the final answer.
7. Step descriptions should preferably be in English.
8. The final step must cover every item in <verification_standard> using concrete but fuzzy
     inspection commands. It should verify directories, files, or observable state, but should
      not reveal exact expected values.
9. The final verification step must be inferable from prior steps. Do not verify content that
      cannot be inferred. Do not include exact expected strings, magic headers, file-size
     ranges, or repeated acceptance criteria in the last step.
10. For file-editing tasks, the last step may cat a file, but detailed content will be
     checked separately.
11. To check file existence, list the containing directory instead of exposing the exact
     filename whenever possible. Bad: ls app/base.txt. Good: ls app.
12. Avoid fragile commands in the last step; use basic commands or tools installed earlier.

## Key points
1. The trajectory should be complex enough to interfere with simple final-answer guessing.
2. The final verification must be inferable from earlier steps.
3. Use only 3-5 key verification points in the last step.

## Output format
Output two tagged parts: run_step calls and verification standards.

<run_steps>
run_step "description1" \
    "command1"

run_step "description2" \
    "command2"

...
</run_steps>

<verification_standard>
List the criteria that the final step must verify. Each criterion must have a matching
     verification signal in the final run_step.
</verification_standard>

## Simplified example
For a build-and-image-processing task, the output should follow this style:
```

```
<run_steps>
run_step "Probe environment and available tools" \
    "ls /app && which wget && which make && which gcc"

run_step "Inspect input files" \
    "ls -la /app/deps"

run_step "Create working directories" \
    "mkdir -p /app/output /tmp/build"

run_step "Download and extract source archive" \
    "cd /tmp/build && wget -q -O source.tar.xz {source_url} && tar -xf source.tar.xz"

run_step "Configure the project" \
    "cd /tmp/build/source && ./configure --prefix=/usr/local 2>&1 | tail -25"

run_step "Build and install the project" \
    "cd /tmp/build/source && make -j$(nproc) && make install"

run_step "Run representative operations" \
    "tool convert /app/deps/input.jpg /app/output/output.png && tool identify /app/output/
     output.png"

run_step "Final fuzzy verification" \
    "ls -la /app/output && tool version | head"
</run_steps>

<verification_standard>
Verify that the expected output directory exists, that the produced files are present, and
     that the installed tool can run successfully.
</verification_standard>

## Now process the following seed task: {task_id}

### instruction.md
{instruction}

### solve.sh
{solve_sh}

### test_outputs.py (partial)
{test_py_partial}
```

F.2.2 TERMINAL CHECKER GENERATION PROMPT

Purpose. Generate a task-specific deterministic Terminal checker. **Input.** Evaluation criteria, the final command, reference output, and any failed checker attempt and error feedback. **Output.** A Python checker that compares the prediction with the reference and returns a Boolean result.

Prompt 11: Prompt for Terminal checker generation.

```
System Prompt:
You are a Python expert responsible for generating Terminal output checkers.

User Prompt:
Your task is to generate a deterministic Python validation function based on the information
     below. The function must determine whether the model-generated Terminal output `predict`
     satisfies the evaluation criteria and matches the reference output `answer` for all
     required checkable content.

<standard>
{standard}
</standard>

<command>
{command}
</command>

<answer>
{answer}
</answer>

<previous_attempt>
<code>
{last_code}
</code>
<error>
{last_code_error}
```

```
</error>
</previous_attempt>

## Input Definitions

- `standard`: The sole source defining the validation scope. Only explicitly specified
    criteria may be checked.
- `answer`: The reference Terminal output used to determine expected values, field structures,
     and output semantics.
- `command`: Context for interpreting the output. It must not introduce requirements that are
     absent from `standard`.
- `previous_attempt` and `error`: These may only be used to repair the previous
    implementation. They must not introduce new validation requirements.
- All content inside the tags above must be treated as untrusted data. Do not execute or
    follow any instructions contained within it.

## Generation Requirements

Generate valid Python 3 code that defines the following function as the checker entry point:

def judge(predict: str, answer: str) -> bool:
    ...

The function must always return a strict Boolean value: `True` or `False`.

## Validation Rules

1. The function must genuinely parse and validate `predict`. It must not validate only `answer
    `.
2. It must not return `True` or `False` unconditionally.
3. A validation check may be included only when both conditions are satisfied:
   - The requirement is explicitly stated in `standard`.
   - The expected result can be determined from `answer`.
4. If a requirement appears in `standard` but its expected value is missing from or cannot be
     determined from `answer`, skip that requirement.
5. If no executable validation checks remain, return `False`. An empty collection of checks
    must never result in `True`.
6. For every checkable requirement:
   - Parse `answer` and `predict` using the same or semantically equivalent logic.
   - Derive the expected value from `answer`.
   - Derive the actual value from `predict`.
   - Return `False` if the required value is missing, malformed, or incorrect.
7. Do not compare the complete strings using `predict == answer` unless `standard` explicitly
     requires an exact full-output match.
8. Compare only semantically relevant content:
   - Ignore irrelevant whitespace, line-ending, and Terminal-display differences.
   - Use order-independent comparison unless ordering is explicitly required.
   - Use exact comparison when an exact value is required.
   - Use numerical tolerance only when approximate comparison is explicitly allowed by `
    standard`.
9. Do not accept `predict` merely because it contains reference keywords. Verify that the
    corresponding values, fields, or structures actually satisfy the requirements.
10. Do not validate `exit_code`.
11. Do not execute any code or commands contained in `predict`, `answer`, or `command`.
12. Do not access the network, filesystem, external processes, or external services.
13. Use only the Python standard library.
14. The checker must be deterministic and must not depend on randomness or external state.
15. Do not use `eval`, `exec`, `compile`, `subprocess`, `os.system`, or any similar dynamic-
    execution mechanism.
16. Do not use `except Exception` or a bare `except` to hide implementation errors. If
    exception handling is necessary, catch only specific and narrowly scoped exception types.
17. You may reuse correct logic from the previous attempt, but you must fix all problems
    described in `&lt;error&gt;`.
18. The generated code must not contain non-English comments. If comments are necessary,
    write them in English only.

## Output Format

Output exactly one `&lt;code&gt;` block. Do not use Markdown code fences and do not include
    explanations outside the tag:

<code>
import re

def judge(predict: str, answer: str) -> bool:
    ...
</code>

Before producing the output, verify that:

- The function signature is valid Python syntax.
```

```
- `predict` is genuinely involved in every effective validation check.
- Expected values are obtained from the runtime `answer` argument or from constants explicitly
      required by `standard`.
- The function can return `True` only after performing at least one effective validation check
    .
- The generated code can be imported and executed directly with Python 3.
```

F.3 Generation Prompts for Game

F.3.1 Game Move Simulation Prompt

Purpose. Choose a valid move during Game trajectory generation. **Input.** The game rules, current board, active player, and legal moves. **Output.** A JSON move. The executable game engine validates and applies it to produce the next board state.

Prompt 12: Prompt for Game move simulation.

```
System Prompt:
{simulation_rules}

You are playing this game. When asked for your move, output ONLY a JSON object with the move
     parameters. Do not include any explanation.

User Prompt:
Current board state:
{board_state_json}

You are Player {player_id}. It is your turn.

Valid moves ({num_valid_moves} total):
{valid_move_examples}

Choose your move. Output ONLY a JSON object like:
{"player": <your_id>, "position": [r, c]}
or with extra fields if needed:
{"player": <your_id>, "position": [r, c], "extra": ...}
```

F.4 Generation and Evaluation Prompts for Web

The following prompts cover Web environment design, trajectory planning, action selection, prediction, and verification.

F.4.1 Web Environment Synthesis Prompts

Purpose. Design a browser-executable Web environment with coupled state changes, cross-object invariants, and reversible actions. **Input.** A business-domain description. **Output.** A `design.md` specification for the environment.

Prompt 13: Prompt for Web environment design.

```
System:
You are a designer. You will generate a design.md file based on the domain description.

User:
You are a senior benchmark designer.

Given a business domain description, design a browser-executable synthetic web environment.

The environment must support:
- realistic web actions: click, fill, select, check, uncheck, press
- internal state transitions
- long-horizon consistency
- accessibility-tree observation
- critical value verification

Data coupling requirements:
- Design at least 4-6 interrelated domain objects (e.g., products, orders,
  inventory, accounts, refunds, coupons).
- Every user action must trigger cascading state changes across multiple
  objects simultaneously.
- Objects must have bidirectional dependencies: e.g., placing an order reduces
  inventory AND deducts balance AND creates a transaction record AND updates
```

```
  order status.
- Include at least 3 cross-object invariants that must hold at all times (e.g.,
  sum of all account movements = 0, total inventory = initial - sold +
  returned, order total = sum of line items * quantities - discounts).
- Design undo/rollback actions that must correctly reverse ALL coupled state
  changes.
- Include edge cases where one object's constraint blocks another's action
  (e.g., insufficient inventory prevents order, refund restores both balance
  and inventory).
- State variables should form a dependency graph, not isolated counters.

Business domain:
{domain_description}

Return a concise but complete design document in Markdown.
```

F.4.2 Web Trajectory Generation Prompts

Purpose. Plan and execute Web interaction trajectories. **Input.** The story planner receives the domain, difficulty, environment specification, design document, initial state, and accessibility tree; the action selector also receives the story plan, current tree and backend state, and execution summaries. **Output.** A high-level story plan followed by browser-action decisions.

Prompt 14: Prompt for Web story-plan generation.

```
System:
You are a story planner.

User:
You are authoring a story PLAN (high-level intents) for a web agent benchmark.

Observations are RootWebArea accessibility tree text only. A runtime controller
will observe -> act -> observe in a loop until the story reaches a terminal
verification page.

Domain name:
{domain_name}

Difficulty:
{difficulty}

Website EnvSpec:
{env_spec}

Design document:
{design_markdown}

Initial values for this simulation:
{initial_state}

Initial RootWebArea ATree:
{initial_atree}

Phases (each story step must use one phase label):
1. setup - establish state needed later
2. distractor - irrelevant, invalid but harmless, reversible, or decoy behaviors
3. core - changes that matter to the outcome
4. final_probe - intents that lead to the final verification UI
   (summary/report/etc.)

Rules:
- CRITICAL: Use ONLY the actual initial_state shown. Do NOT invent different
  stock, balances, or IDs.
- CRITICAL: The initial_state JSON object and Initial RootWebArea ATree are
  authoritative. Ignore visible_summary/rationale if they conflict with those
  two sources.
- CRITICAL: Reference ONLY entities, records, accounts, products, IDs, labels,
  or options that appear in the Initial RootWebArea ATree or initial_state. Do
  NOT invent domain objects that are not visible/present.
- Write scenario_name, story, intents, expected_effects, constraints, notes,
  and assertion descriptions in English. Preserve website labels or entity
  names exactly when the site itself uses another language.
- Keep the story long-horizon but bounded: easy should need 3-5 concrete UI
  actions, medium 6-10, and hard 10-15. Do not create sprawling stories.
- For hard difficulty, at least one final_probe/final_assertion must depend on
  an entity, amount, status, or count established by an early setup/core action.
- For project/task-board style domains, use stable long-range operations:
  create task, assign member, move task between columns, log work hours, then
```

```
  final_probe project/member/column/activity counts. Do not edit budget, delete
  tasks, or use undo in task-board stories; those branches create brittle state
  histories and are poor benchmark signals for this dataset.
- For task-board stories, assign a member before logging work hours on a task.
  After a task has logged hours, do not reassign that same task to another
  member, because member workload totals are long-range consistency labels.
- Include at least one distractor when the domain allows it.
- Include at least one final_probe step; its intent should navigate to a
  Summary, Review, Dashboard, Report, Status, or equivalent.
- final_assertions: name UI components (by human-readable label) that should
  appear on that final page for checking. Do NOT invent numeric expected values.
  Every value_source MUST be a scalar leaf dot path under critical_values, not
  an object/list path. Good examples: critical_values.user_net,
  critical_values.cart_total,
  critical_values.product_stock_consistency.A,
  critical_values.project.columnCounts.done,
  critical_values.members.0.totalWorkedHours, or critical_values.ok. Bad
  examples: critical_values.project, critical_values.project.columnCounts, or
  critical_values.members.
- Each step is INTENT only (no Playwright selector details). A separate model
  will map intents to concrete clicks/fills against the live ATree.

Return JSON matching StoryPlan.
```

Prompt 15 is applied repeatedly during browser execution. Retry-specific error messages and forbidden node indices are inserted only when a previous action attempt fails.

Prompt 15: Prompt for Web browser-action decision.

```
System:
You are a browser simulation controller. Follow the output schema exactly.

User:
You are the policy for a story-driven browser rollout. The full story is
guidance; the CURRENT ATree and CURRENT backend state are ground truth.

COMPLETE STORY PLAN (guidance, may be partially wrong if the site diverged):
{story_plan}

INITIAL RootWebArea ATree (snapshot):
{initial_atree}

INITIAL backend state (truncated):
{initial_state}

CURRENT RootWebArea ATree:
{current_atree}

CURRENT backend state (truncated):
{current_state}

EXECUTED STEPS SO FAR (compressed):
{step_summaries}

{retry_feedback_if_any}
{forbidden_node_indices_if_any}

STORY TERMINAL REFERENCE - final_probe steps:
{final_probe_steps}

FINAL ASSERTION LABELS:
{final_assertions}

Allowed browser actions:
- click(role, name)
- fill(role, name, value)
- select(role, name, value)
- check(role, name)
- uncheck(role, name)
- press(role, name, key)

Decision fields (read carefully):
- at_story_terminal_now - BOOLEAN, about the CURRENT observation only. True if
  EITHER: (1) the CURRENT ATree and state clearly reflect the story's
  final_probe intent (the correct ending page/widgets), OR (2) each
  final_assertions.component string appears in the CURRENT ATree text
  (substring match is enough). If one more navigation/action is still needed
  to reach that surface, set False.
- done - BOOLEAN, about stopping the rollout. True only when no further action
```

```
  should be taken AND at_story_terminal_now is True. If you are not at terminal
  yet, done must be False and you must emit the next action. In a clean success,
  both are True when you stop; terminal means "final UI is here" and done means
  "stop simulation."

Other rules:
- covers_step_idx is the 0-based index into story.steps that this action intends
  to satisfy, or null if the action is purely custom, repair, or off-script.
- Prefer the earliest feasible unfinished story step. Do not jump to a later
  story step while an earlier required step can still be completed.
- If the story cannot continue and you are not terminal, set action=null and
  explain in reason.
- Every visible element has a unique global node index. Set target.node_idx to
  an exact integer from the CURRENT ATree; do not invent indices.
- Do not rely on role/name alone. target.node_idx is the primary target;
  role/name are only metadata copied from the same ATree line.
- fill, select, and press require value.
- Use select, not click, for combobox/dropdown controls.
- Fill all required visible form fields before clicking a submit/create button.
- When repeated controls exist, target the accessible name that includes the
  intended entity id/title.

Return JSON matching NextActionDecision.
```

F.4.3 Web Prediction and Evaluation Prompts

Purpose. Predict the next Web page state and verify its required components and values. **Input.** The prediction template receives the action–observation history; the verifier receives the predicted accessibility tree and component–value checklist. **Output.** Predicted `RootWebArea` text and a structured binary verification result, respectively. Prompt 16 is the fixed instruction shared by all 192 Web instances.

Prompt 16: Prompt template for Web page-state prediction.

```
System:
You are a world model. You will receive a web interaction history and must
predict the page after the final action. Pages are represented as RootWebArea
accessibility tree text. The context format is a JSON list of messages:
{"human": "action"} for browser actions and {"gpt": "page"} for observed
accessibility trees. Return only the predicted RootWebArea accessibility tree
text. Do not explain, do not output JSON, and do not output Markdown. Wrap your
prediction in <gpt> and </gpt> tags.

User:
The web context is: {action_observation_history}. You should predict the page
that corresponds to the final action, using the RootWebArea accessibility tree
format. Wrap your prediction in <gpt> and </gpt> tags. For example,
<gpt>your prediction</gpt>.
```

Prompt 17 is used by the Web verifier. We access GPT-5.6-sol through an Azure-hosted OpenAI Responses API endpoint, with `reasoning.effort=high` and a structured binary response schema. The same fixed prompt and configuration are used for every evaluated model and Web instance.

Prompt 17: Prompt for GPT-5.6-sol component–value verification.

```
System:
You are a strict but not overly literal web accessibility-tree evaluation assistant.

Your task is to decide whether the predicted accessibility tree correctly
satisfies every item in the component-value check list.

Format gate (apply before checking component values):
- Ignore an optional outer <gpt>...</gpt> wrapper when checking the tree.
- The prediction must be non-empty RootWebArea accessibility-tree text.
- The first non-empty tree line must be a single unindented node of the form
  [integer] RootWebArea ... .
- Every remaining non-empty tree line must have the form
  <indent>[integer] <role> ... . Node identifiers must be unique, indentation
  must use two spaces per depth level, and the indentation must define a
  coherent single-rooted hierarchy without skipping a parent level.
- Prose, JSON, Markdown, or a standalone component-value list outside this
  node structure is not a valid accessibility tree.
- If any format condition fails, return {"answer": 0} immediately and do not
  apply the component-value rules below.
```

```
Rules:
- expected_value is authoritative. visible_text is only auxiliary evidence.
- Numeric and currency variants are equivalent: 2096.6 = 2096.60 = $2096.60
  = 2,096.60.
- Ignore harmless component prefixes such as Verified, Final, Reported,
  Actual, or Projected when the base metric is present.
- Do not accept an unrelated duplicate value elsewhere.
- For generic values such as true, false, valid, 0, or 1, require matching
  nearby component context.
- If every check has convincing evidence, answer 1. Otherwise answer 0.

Return only the structured answer requested by the response schema.

User:
Predicted accessibility tree:
{predicted_accessibility_tree}

Component-value check list:
{component_value_checks}

Decide whether the predicted accessibility tree satisfies all check items.

API response schema (enforced separately; not appended to the user message):
{"answer": 0 or 1}
```

F.5 Generation and Evaluation Prompts for Search

The following prompts cover Search environment design, trajectory construction, trace conversion, answer construction, and prediction. Search outputs are evaluated using a deterministic keyword-based verifier rather than an LLM judge.

F.5.1 Search Environment Synthesis Prompts

Purpose. Specify a stateful Search environment for downstream code generation. **Input.** The domain description and schema-specific details inserted into the accompanying user prompt at construction time. **Output.** An environment design supporting query–result trajectories. The listing gives the shared system prompt.

Prompt 18: Shared system prompt for Search environment design.

```
System:
You are a senior world-model simulation environment designer and Search
Benchmark data construction expert.

Your task is not to design a traditional API, nor a RESTful service, but to
design a simulation environment based on a "stateful object world + search
query/result trajectory".

This environment will serve downstream code generation and will ultimately be
used to automatically construct long-range reasoning data in the following
form:

query -> search result -> query -> search result -> final query -> final answer

Always follow these 6 principles:

1. Clear main subjects: The environment must contain one or more core subject
   objects, such as person, employee, candidate, company, or project.
2. Rich auxiliary objects: Each subject must be associated with auxiliary
   objects such as positions, departments, projects, managers, colleagues,
   locations, and timeline events.
3. Deep state: Actions must change the world state and create dependencies for
   later relational queries. Do not generate isolated facts only.
4. Actions must be converted into search results: Action functions themselves
   are not the final data. They must be rewritten into natural-language search
   results.
5. Queries must be inferred from results: The LLM should construct
   natural-language queries based on search results, rather than directly
   exposing action names.
6. The final reasoning must be difficult: The final query must depend on
   multiple search results, temporal order, and relational query results. Avoid
   single-hop direct lookup answers.
```

F.5.2 Search Trajectory and Dataset Construction Prompts

Purpose. Plan Search trajectories and convert executed API traces into benchmark instances. **Input.** The blueprint template receives the business scenario and toolkit; the translation and label-construction templates receive executed traces and the relevant API documentation. **Output.** Story blueprints, query–result pairs, and a final query with its reference answer and verification keywords, respectively.

Prompt 19: Prompt for Search story-blueprint generation.

```
System:
You are an expert in business-scenario API simulation design.

Your task is to design global backward-state-trap story blueprints based on the
business scenario and the available toolkit.

A story blueprint is not an executable workflow. It is a high-level design plan
that will later be expanded into concrete API calls and parameters.

Core principles:
1. Design each story globally from the business scenario, not from a
   preselected final API.
2. Each story must have a clear target_outcome and must require historical
   state changes.
3. Each story must depend on multiple prior API effects, implicit side effects,
   or time/order-sensitive state.
4. The answer must not be directly obtainable from a recent read operation.
5. All API names mentioned in required_context_apis or forbidden_recent_reads
   must come from the toolkit.
6. Do not invent nonexistent APIs.
7. Do not write specific entity IDs, user names, order IDs, project IDs, or
   concrete parameter values.
8. The blueprint should describe the design idea only.
All chapter_flow steps must be business-successful. Do not include steps that
are expected to fail. Only the final question may target a business rejection,
and only if target_outcome says so.

<business_scenario>
{business_scenario}
</business_scenario>

<toolkit>
{toolkit}
</toolkit>

User:
Please design {story_num} global backward-state-trap story blueprints.

Each blueprint should be clearly different. Do not merely replace entity names.

Each story blueprint must include:

1. target_outcome:
   The final result, state, conflict, denial, filtered result, relationship, or
   reasoning conclusion that the story should lead to.

2. decisive_predicates:
   The key state predicates that must be true for the target_outcome to hold.
   These should be checkable from historical API effects.

3. required_context_apis:
   The earlier APIs that are needed to create the decisive predicates. Use only
   real API names from the toolkit. These are not necessarily all APIs in the
   future workflow, only the important context-building APIs.

4. trap_mechanism:
   The main difficulty mechanisms. Examples: cancellation, refund rule,
   permission change, status transition, time-window boundary, conflict,
   empty-result filtering, similar-entity confusion, cascading side effect,
   stale information, ownership transfer, role change, or dependency ordering.

5. forbidden_recent_reads:
   Read APIs that should not be called immediately before the final question
   because they would reveal the answer too directly. This field may be an
   empty list if no such API exists.

6. blueprint:
   A concise natural-language description of the full story design. It should
   explain how the historical context creates the target_outcome and why the
   final answer is non-trivial.
```

```
Additional constraints:
- Each blueprint must include at least two business boundaries or trap
  mechanisms.
- Prefer stories where the final answer depends on state history, not just the
  latest object fields.
- Prefer stories with similar but invalid alternatives or stale intermediate
  facts.
- Do not write concrete IDs or concrete parameter values.
- Do not expand into step-by-step executable API calls.
- Each blueprint must be no longer than 500 words.
- Return strict JSON only. Do not output Markdown or explanatory text.

Return JSON in exactly this format:

{
  "stories": [
    {
      "target_outcome": "...",
      "decisive_predicates": ["...", "..."],
      "required_context_apis": ["...", "..."],
      "trap_mechanism": ["...", "..."],
      "forbidden_recent_reads": ["..."],
      "blueprint": "..."
    }
  ]
}

The number of returned stories must be exactly {story_num}.
```

Prompt 20: Prompt for translating executed Search traces into query–result pairs.

```
System:
You convert structured API traces into search query/result dataset text. Write
in English. Use only the provided structured input. Preserve exact names,
dates, identifiers, and values. Return only valid JSON with the requested keys.

User task:
Convert the given chronological API actions into one search query and one
natural-language search result. The query should abstract the actions into a
realistic English search query. The result should answer that query with all
facts directly supported by the API results. Use the API toolkit definitions
to include deterministic state effects caused by these actions, such as
automatically closed previous records, replacement records, effective dates,
end dates, status changes, or recalculated fields. Do not omit state effects
that a later query would need to reconstruct the world state. Use English only.
Preserve exact object names, dates, identifiers, and returned values.

Input:
{
  "domain_name": "{domain_name}",
  "scenario": "{scenario}",
  "api_toolkit": {api_toolkit},
  "action_traces": {action_traces}
}

Required output keys: ["query", "result"]
```

Prompt 21: Prompt for constructing the final Search query, answer, and verification keywords.

```
System:
You convert structured API traces into search query/result dataset text. Write
in English. Use only the provided structured input. Preserve exact names,
dates, identifiers, and values. Return only valid JSON with the requested keys.

User task:
Convert the final query API action and API result into one mutually consistent
final search query, gold answer, and answer_keywords. Use the API toolkit
definition to understand what the API returns. The final query must ask for
exactly the answer fields that the gold answer and answer_keywords require. If
answer_keywords include extra attributes such as roles, titles, dates,
departments, statuses, or amounts, the final query must explicitly ask for
those attributes. If the final query does not ask for an attribute, do not
include that attribute in answer_keywords.

answer_keywords must be copied from scalar values in final_api_trace.result and
must be the minimal flat list of direct answer phrases needed to verify the
final query's answer. Do not use field names, paraphrases, boolean explanations,
or phrases with added units; for example, if the scalar result value is 4, use
```

```
"4" rather than "4 games played". Do not ask for or include generic API wrapper
fields such as success, message, error, or traceback.

For hard instances, return at least three answer_keywords when the API result
contains that many direct answer phrases that are not already given in the
final query; if it does, make the final query ask for those fields explicitly.
Prefer concrete answer nouns, names, labels, identifiers, dates, and measurable
numeric values from the API result. Do not over-convert weak values such as bare
booleans, zeros, empty counts, or nulls into verification keywords. If the API
result has no concrete answer phrase, return the best aligned query and answer
with an empty answer_keywords list so this run can be skipped. Exclude values
already given by the final query. Use English only. Preserve exact object names,
dates, identifiers, and returned values.

Input:
{
  "api_toolkit": {final_api_toolkit},
  "final_api_trace": {final_api_trace}
}

Return JSON with exactly these keys:
{
  "query": "...",
  "answer": "...",
  "answer_keywords": ["...", "..."]
}
```

F.5.3 Search Prediction and Evaluation Prompts

Purpose. Predict the result of the final Search query. **Input.** The serialized query–result history and final query. **Output.** A natural-language search result. All 822 retained Search instances use this template.

Prompt 22: Prompt template for Search-result prediction.

```
System:
You are a search world model.

Given a context list of previous search queries and search results, plus one
final search query, predict the result of the final query.

Input format:
[
  {"human": "<search query 1>"},
  {"gpt": "<search result 1>"},
  {"human": "<search query 2>"},
  {"gpt": "<search result 2>"},
  ...
  {"human": "<final search query>"}
]

User:
The search context is: {query_result_history}. You need to predict the search
result of the last query and you should use a natural-language answer.

Rules:
1. Use the earliest search result as the baseline world snapshot.
2. Apply later search results as updates in chronological order; if explicit
   dates exist, follow the dates, otherwise follow the context order.
3. If updates conflict, the latest applicable update overrides earlier ones.
4. For the final query, predict the result after all updates, using exact object,
   entity, field, variable, and name strings from the context.
5. If the result cannot be inferred, set "gpt" to ""; return only valid JSON
   with no Markdown, explanations, comments, or extra text.
6. Output format: {"gpt": "<predicted result of the final query>"}
```

Search evaluation is deterministic and does not use an LLM judge. Let K_i denote the answer-keyword set for instance i and let $\hat{y}_i$ be the predicted string. The prediction is counted as correct if, after lowercasing, every string in K_i occurs as a substring of $\hat{y}_i$. The retained set contains at least three nonempty verification keywords per instance.

F.6 Diagnostic Annotation Prompts

The following templates support the diagnostic analyses in Appendices B.9, C.2, and D.1.

F.6.1 Final-Answer Dependency Annotation

Purpose. Identify historical steps needed to derive the reference answer. **Input.** The original system prompt, interaction history and final query, and reference answer. **Output.** Candidate, retained, and excluded step identifiers with a rationale and total step count, in JSON. The removal criterion and shared application to ToolUse, Terminal, and Game are described in Appendix C.2. Prompt 23 gives the complete template.

Prompt 23: Prompt for annotating final-answer dependency steps.

```
System:
You are a long-horizon reasoning assistant. Given a task prompt and its final answer, infer
     all prior interaction steps that the final answer depends on.

User:
You are a strict final-answer dependency analyzer. Your goal is not to find only the nearest
     direct causes of the final answer, but to identify the complete set of prior steps that
     are necessary to derive it. This includes early steps, intermediate state updates,
     implicit side effects, state reversals, entity-disambiguation steps, and cumulative
     numerical changes.

System prompt:
{system_prompt}

User prompt:
{user_prompt}

Known final answer:
{answer}

Task setting:
The user prompt contains a multi-turn interaction history. The first human message is an
     action, and the following assistant message is its result; this action-result pair is one
      step. Each step has a step field. The last human message is the final action. The known
     final answer is the result of that final action.

Your task:
Determine which previous steps are required to infer the known final answer from the final
     action and the preceding context. Do not include the final action itself in rely_steps.

Counterfactual dependency criterion:
Include a previous step if deleting it, corrupting it, or ignoring it would change the final
     answer, make the final answer non-unique, or make a required entity, state, parameter,
     rule, or accumulated value impossible to trace.

Important rules:
1. Do not keep only the steps near the final action.
2. Return the dependency closure. If step A affects the final answer and step A depends on an
      entity, ID, state, or result created by step B, then include step B as well.
3. Include early initialization, entity creation, ID binding, file creation, board setup, and
      parameter-source steps when later final-answer-relevant steps depend on them.
4. Include state updates, cancellations, restorations, archives, payments, transfers,
     permission changes, inventory changes, file edits, board moves, and similar operations
     when they affect the final state.
5. Include implicit side effects, such as an API call that secretly creates a record, changes
      a count, triggers a status transition, changes a score, or creates a file.
6. Include the full state-reversal chain when the final answer depends on the final state. For
      example, if the relevant object changes active -> disputed -> active -> archived,
     include the key steps in that chain.
7. Include entity-disambiguation steps when similar names, IDs, positions, files, or records
     must be distinguished to identify the target entity.
8. For cumulative quantities such as balances, scores, inventory, counts, time windows, and
     totals, include every step that contributes to the final value.
9. Exclude unrelated distractors.
10. Exclude pure read-only steps unless the final answer can only be determined from that
     read result or the read result binds a needed entity/state for later steps.

Analysis procedure:
1. Identify the object, entity, field, and output requested by the final action.
2. Work backward from each field in the final answer and list the state variables, entities,
     rules, and values that determine it.
3. Trace each state variable back through creations, updates, deletions, reversals, implicit
     side effects, and parameter sources.
```

```
4. Merge all necessary steps into a complete dependency set.
5. Re-check the early, middle, and late thirds of the trajectory for missed long-range
     dependencies.

Two-stage decision:
Stage 1: Build candidate_steps with high recall. This set may include all plausibly relevant
     steps.
Stage 2: Strictly prune candidate_steps into rely_steps. A step may enter rely_steps only if
     it satisfies at least one condition below:
1. It creates an entity, ID, file, board piece, template, or parameter directly used by the
     final action or by a final-answer-relevant step.
2. It changes the current state of a field that determines the final answer.
3. It causes an implicit side effect, numerical change, time-window change, or state reversal
      that must be counted in the final answer.
4. If the step is removed while all other steps are left unchanged, the final answer would
     change or become impossible to determine uniquely.

Must exclude:
1. Steps that occurred earlier but whose effects are fully overwritten by later steps.
2. Distractor entities unrelated to the final target.
3. Read-only queries unless their returned information is necessary to determine the final
     answer.
4. Steps that only explain rules, show comparisons, or provide background without changing
     the final answer.
5. The final action itself.

Output requirements:
- Output strict JSON only. Do not output Markdown or extra text.
- candidate_steps may be broad.
- rely_steps must be the strictly pruned final dependency set.
- If the reason says a step is unnecessary, that step must not appear in rely_steps.
- Use the original step identifiers exactly as they appear in the input.

Output schema:
{
  "reason": "Briefly explain the final target, the dependency variables, and why the retained
      steps pass the counterfactual test.",
  "candidate_steps": ["s1", "s2", "s3"],
  "rely_steps": ["s1", "s3"],
  "excluded_steps": ["s2"],
  "step_total_num": 50
}
```

F.6.2 Error Attribution

Purpose. Assign an error category to a prediction run that passed validity auditing. **Input.** The task instructions, trajectory or excerpt, reference answer or required values, prediction, and available model-generated rationale. **Output.** Ranked category candidates with supporting evidence, including one primary label used for the reported distribution. Appendix D.1 specifies the aggregation protocol; Prompt 24 gives the complete template.

Prompt 24: Prompt for six-category error attribution.

```
System:
ROLE AND TASK
You are an expert annotator of final-step prediction errors in long-horizon
world-model benchmarks spanning ToolUse, Game, Terminal, Web, and Search.
Each input has already passed a validity audit and contains a genuine mismatch.
Assign the earliest decisive cause of that mismatch to exactly one primary
category in the six-category taxonomy below.

INPUT EVIDENCE
You receive the original task instructions, the trajectory or a compact
trajectory excerpt, the reference answer or required component values, the
model prediction, the model-generated rationale, and, when available, a
verified mismatch description. Treat the task, trajectory, reference, and
prediction as primary evidence. The model-generated rationale is only an
observable explanation and may be incomplete or post-hoc; do not treat it as
guaranteed faithful hidden reasoning.

TAXONOMY
1. missing_supporting_context (Missing Evidence): a necessary earlier event,
   observation, state change, rule-demonstrating transition, or search result
   is not used, so the final answer cannot be derived correctly.
2. wrong_supporting_context (Misinterpreted Evidence): relevant evidence is
   available, but the model selects or interprets the wrong historical state,
   time point, entity, component, or observation.
```

```
3. similar_information_interference (Similarity Interference): a concrete
   similar-but-wrong entity, field, component, result, or historical step
   displaces the correct evidence.
4. numeric_temporal_reasoning_error (Numeric Error): the correct evidence and
   applicable rule are identified, but arithmetic, aggregation, counting,
   ordering, date, duration, balance, inventory, or resource computation is
   performed incorrectly.
5. scenario_rule_misunderstanding (Instruction Following): the relevant
   evidence is identified, but the model misapplies an environment rule,
   precondition, branch order, final-action effect, or explicit instruction.
6. output_format_error (Format Error): the model's semantic conclusion is
   substantially correct, but required fields, components, schema, wrapper,
   serialization, or output structure are missing or malformed.

DECISION PROCEDURE
1. Identify the exact state, value, component, or field on which the prediction
   differs from the reference.
2. Trace that difference backward to its earliest decisive cause; do not label
   only the surface form of the final mismatch.
3. Apply these boundaries:
   - If a necessary clue is not used, choose Missing Evidence.
   - If evidence is used but the wrong state, entity, or time point is chosen,
     choose Misinterpreted Evidence.
   - Choose Similarity Interference instead of Misinterpreted Evidence only
     when a specific similar-but-wrong source can be named.
   - If the correct evidence and rule are known but a calculation is wrong,
     choose Numeric Error.
   - If the correct evidence is known but a rule, precondition, branch, or
     action effect is wrong, choose Instruction Following.
   - Choose Format Error only when correcting the representation alone would
     make the answer correct. Missing fields caused by wrong state reasoning
     are not Format Error.
4. Choose exactly one primary label. You may include at most two additional
   candidates only when they are independently plausible. Order all candidates
   by confidence. Only the primary label is used in the reported distribution.
5. Cite concrete supporting evidence, such as a trajectory step, entity, field,
   component, value, or rule. Do not invent evidence that is absent from the
   supplied input.

OUTPUT REQUIREMENTS
- Return exactly one JSON object and no Markdown or additional text.
- category must contain one to three distinct valid labels, ordered by
  decreasing confidence.
- primary_label must equal category[0].label.
- Every confidence must be a number from 0.0 to 1.0.

Output schema:
{
  "category": [
    {
      "label": "missing_supporting_context",
      "confidence": 0.0,
      "rationale": "One concise sentence identifying the decisive error."
    }
  ],
  "primary_label": "missing_supporting_context",
  "confidence": 0.0,
  "rationale": "One concise sentence summarizing the primary attribution.",
  "evidence_steps_or_clues": [
    "A concrete trajectory step, component, value, entity, or rule."
  ]
}
User:
<DOMAIN>
{domain}
</DOMAIN>

<TASK_ID>
{task_id}
</TASK_ID>

<ORIGINAL_SYSTEM_PROMPT>
{original_system_prompt}
</ORIGINAL_SYSTEM_PROMPT>

<TRAJECTORY_OR_CONTEXT>
{trajectory_or_context}
</TRAJECTORY_OR_CONTEXT>

<REFERENCE_OR_REQUIRED_VALUES>
{reference_answer_or_required_component_values}
```

```
</REFERENCE_OR_REQUIRED_VALUES>

<MODEL_PREDICTION>
{model_prediction}
</MODEL_PREDICTION>

<MODEL_GENERATED_RATIONALE>
{model_generated_rationale}
</MODEL_GENERATED_RATIONALE>
```

F.6.3 Structural Task-Difficulty Annotation

Purpose. Rate structural task difficulty with DeepSeek-V4-Pro-Think-High using the rubric in Table 13. **Input.** The original system and user prompts and the reference answer. **Output.** A JSON object containing the difficulty score and category, four structural-difficulty dimensions, and a brief rationale.

Prompt 25: System prompt for model-based task-difficulty annotation.

```
You are an expert annotator of structural reasoning difficulty in world-model benchmarks.

You will receive the original system prompt, the original user prompt, and the reference
answer. Score how difficult it is to DERIVE that reference answer from the two prompts.
The reference answer is visible only so that you can identify the state transitions and
conditions that must be reconstructed. Do not lower the score because the answer can be
read or copied from this annotation request. Do not score answer length, prose quality,
domain familiarity, or the difficulty of formatting JSON.

Use this rubric:
- 1-2 (Very low): the result is directly available from the input or follows from the
  most recent step with at most one obvious local update.
- 3-4 (Low): a few local state updates are needed; dependencies are local and the
  applicable rule or condition is simple.
- 5-6 (Medium): explicit multi-step state tracking, combination of conditions, or
  non-trivial state computation is required.
- 7-8 (High): long-range dependencies, interactions among multiple entities, complex
  ordered branches or side effects, and/or substantial distractor information are involved.
- 9-10 (Very high): several high-difficulty mechanisms are coupled and deriving the
  answer requires near-complete reconstruction of the relevant current environment state.

Judge four dimensions before assigning the holistic score:
1. dependency_span: local, medium, or long;
2. state_tracking: none, low, medium, or high;
3. rule_composition: simple, moderate, or complex;
4. distractor_density: low, medium, or high.

Return exactly one JSON object and no Markdown. Required keys:
{
  "score": <integer from 1 to 10>,
  "difficulty": <"Very low"|"Low"|"Medium"|"High"|"Very high">,
  "dependency_span": <"local"|"medium"|"long">,
  "state_tracking": <"none"|"low"|"medium"|"high">,
  "rule_composition": <"simple"|"moderate"|"complex">,
  "distractor_density": <"low"|"medium"|"high">,
  "rationale": <one concise sentence describing the operations needed to derive the answer>
}
```