CODEWORLDBENCH: HOW WELL CAN CODING AGENTS BUILD EXECUTABLE AGENT WORLDS?

Anonymous authors
Paper under double-blind review

ABSTRACT

Modern coding agents increasingly demonstrate the capability to scale and generate complex interactive agent environments. However, existing benchmarks evaluate the ability to generate repositories from scratch or repair localized bugs, leaving its capacity to construct fully functional, stateful environments largely unmeasured. Therefore, we introduce CodeWorldBench, a benchmark designed to assess the quality of code-driven world construction. We construct 118 environments spanning common workflows and professional software domains, requiring agents to generate or repair repositories that realize a stateful world, including its data, tool interfaces, and transition logic. Rather than relying on code similarity, we test behavioral equivalence: submitted and reference worlds start from the same seed, receive hidden tool calls, and are compared on their outputs and resulting states. Our evaluation reveals vulnerabilities in current systems. While models preserve already-correct behavior, they struggle to construct environments from scratch and fail at repairing broken behavior. Agents fail on state transitions and validation order rather than syntax. Because agent-built environments increasingly serve as the grounding for training subsequent models, an environment that runs but transitions incorrectly supplies a damaging false signal. Uncovering these flaws provides insights and directions for robust Recursive Self-Improvement and the self-evolution of future agents.[1]

1 INTRODUCTION

Large language models (LLMs) have moved from static text and code generation to acting through tools, MCP servers, and external environments (OpenAI, 2026a; Anthropic, 2024; OpenClaw, 2026). Acting in such settings requires modeling how actions transform environment state and encoding those dynamics in executable systems (Wang et al., 2024; Chae et al., 2025; Tang et al., 2024). We study this through code-based environments, where entities, actions, constraints, and transition rules are realized as repository-level implementations.

Existing benchmarks do not evaluate this capability. Function-level benchmarks such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) target localized synthesis under static tests. Repository-level and software-engineering benchmarks, including SWE-bench Verified (OpenAI, 2024), extend this to completion, repair, and repository construction (Liu et al., 2023; Chan et al., 2025; Zhao et al., 2025; Ding et al., 2025), but tie correctness to APIs, unit tests, or patches over existing code. Tool-use benchmarks such as τ^3-Bench (Sierra AI, 2026) evaluate agents inside already-specified environments (Mao et al., 2025; Lu et al., 2025), testing whether agents call tools correctly rather than whether they can build the state dynamics beneath those tools. A parallel line scales environment distributions for training (Huang et al., 2025; Cai et al., 2025; Song et al., 2026; Dong et al., 2026) from crawled tasks and LLM-based synthesis, so it never measures how faithful a single agent-built environment is to its specification. The open question is not whether an agent can emit a runnable repository, which current systems do routinely, but *how well does an agent-constructed environment reproduce the state dynamics its specification prescribes?*

Production deployments make that question concrete. Platforms now allow agents to interactively construct environments directly on the infrastructure used for training and evaluation. These sessions can then be checkpointed into reusable environments, unifying construction and execution on a single

[1]The code and data are available for review at `https://anonymous.4open.science/r/CodeWorldBench-299B/`.

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.

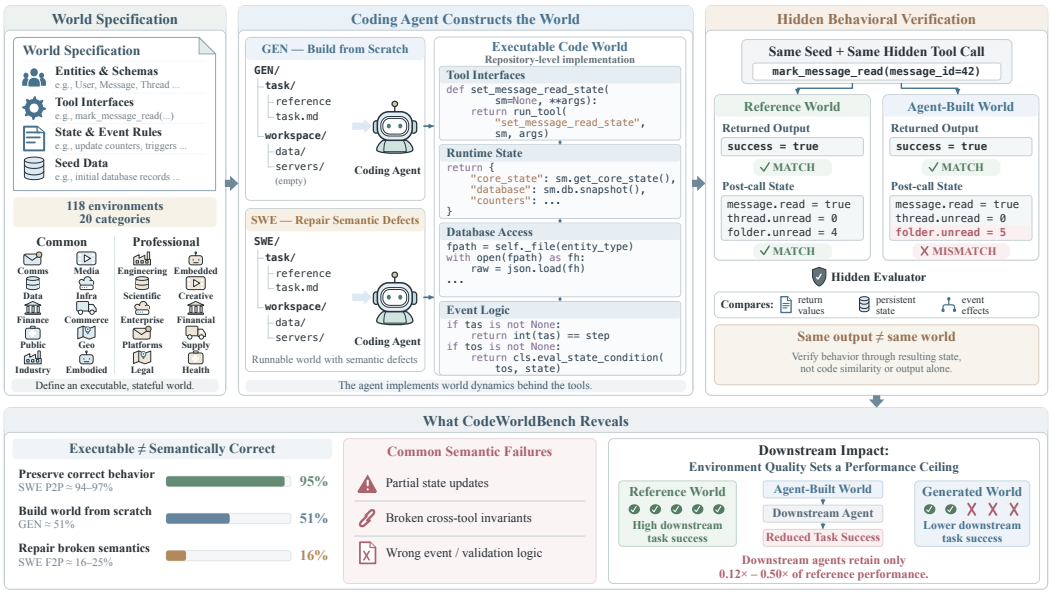


Figure 1: CodeWorldBench overview. It evaluates whether coding agents can construct and repair executable, stateful environments from world specifications. Agents either build a world from a workspace containing initial data or fix semantic defects in an existing implementation. Hidden behavioral tests apply identical tool calls to reference and agent-built worlds, comparing outputs, persistent state, and event effects. The results reveal a gap between runnable code and faithful world behavior, with semantic errors limiting downstream agent performance.

substrate (Huang et al., 2026a). The pipeline's *capacity* challenge has a production answer: a single scale unit reportedly serves millions of sandboxes per day at over five thousand creations per second. In contrast, its *fidelity* challenge remains unaddressed by public metrics. Ensuring that an agent-built environment correctly implements intended state dynamics currently relies solely on internal quality gates. This discrepancy is critical, as a world that runs but transitions incorrectly yields a misleading reward rather than a missing one.

We introduce **CodeWorldBench**, a benchmark for code-driven environment generation. Given a task specification, an agent must produce an executable repository implementing a stateful environment: seed databases, entity schemas, tool interfaces, runtime state, event logic, and hidden behavioral rules. The evaluator initializes the reference and submitted implementations from the same seed, executes hidden tool calls, and compares outputs and resulting state projections, so a repository that answers plausibly while mutating state incorrectly does not pass. Environments divide into common workflows and professional software domains, and each yields two parallel settings: GEN requires from-scratch construction of the executable world, while SWE supplies a runnable but semantically broken implementation to repair without regressing behavior that is already correct. Empirical results show that world construction and repair represent distinct capabilities. While models reliably preserve existing behavior, they achieve moderate success when building worlds and struggle severely when repairing broken ones. Trajectory analysis reveals that agents rely on superficial verification rather than rigorous test suites, allowing latent defects to pass unnoticed and ultimately causing downstream solvers to fail due to broken or missing tools.

Our contributions are as follows.

- We introduce **CodeWorldBench** to measure the behavioral *quality* of agent-built environments. It features 118 environments across generation (GEN) and repair (SWE) settings, using strict state-based comparisons rather than simple output checks.
- We reveal a critical performance gap invisible to standard output metrics: while models easily preserve existing correct behavior, they struggle significantly with from-scratch construction and repairing broken logic, primarily failing on state transitions rather than syntax.
- We demonstrate that construction quality severely impacts downstream performance: an independent agent operating within these generated environments achieves only 12% to 50% of the success rate seen in reference environments, largely due to broken or missing tools rather than task difficulty.

Table 1: Coding benchmarks, ordered by release date. *Persistent state*: hidden checks on the state actions induce. *World construction*: implementing an executable environment another agent can call.

Benchmark	Scale	Repository-level	Generation	Repair	Persistent state	World construction
HumanEval (Chen et al., 2021)	164 problems	✗	✓	✗	✗	✗
MBPP (Austin et al., 2021)	974 problems	✗	✓	✗	✗	✗
RepoBench (Liu et al., 2023)	1,669 repositories	✓	✓	✗	✗	✗
SWE-bench Verified (OpenAI, 2024)	500 issues	✓	✗	✓	✗	✗
Commit0 (Zhao et al., 2025)	54 libraries	✓	✓	✗	✗	✗
MLE-bench (Chan et al., 2025)	75 competitions	✓	✓	✗	✗	✗
NL2Repo-Bench (Ding et al., 2025)	104 repositories	✓	✓	✗	✗	✗
Terminal-Bench (Merrill et al., 2026)	89 tasks	✓	✗	✗	✗	✗
ProgramBench (Yang et al., 2026)	200 programs	✓	✓	✗	✗	✗
RepoZero (Zhang et al., 2026)	100 repositories	✓	✓	✗	✗	✗
DeepSWE (Huang et al., 2026b)	113 tasks	✓	✓	✓	✗	✗
FrontierSWE (Kondra et al., 2026)	34 tasks	✓	✓	✓	✗	✗
CodeWorldBench	**118 environments × 2 settings**	✓	✓	✓	✓	✓

2 Related Work

Benchmarks for code and agents. Code-generation evaluation has moved from function-level benchmarks such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) to repository-level settings including RepoBench (Liu et al., 2023) and SWE-bench Verified (OpenAI, 2024), to machine-learning engineering, library construction, and repository generation (Chan et al., 2025; Zhao et al., 2025; Ding et al., 2025). Terminal-Bench (Merrill et al., 2026), DeepSWE (Huang et al., 2026b), and FrontierSWE (Kondra et al., 2026) extend this to long-horizon work graded by task-specific verifiers. Closest to us are benchmarks that build a codebase from scratch and verify it by execution: ProgramBench (Yang et al., 2026) rebuilds 200 programs from a compiled binary and documentation, and RepoZero (Zhang et al., 2026) verifies from-scratch generation through an execution harness. Both evaluate the repository as a program to be tested. CodeWorldBench evaluates it as an environment, where the artifact must expose a tool surface agent can call and hidden cases check the state each call leaves. A line (Mao et al., 2025; Lu et al., 2025; Yao et al., 2025; Sierra AI, 2026) evaluates tool use and stateful interaction but supplies the environment rather than asking the agent to write it, measuring the other half of the problem. Table 1 compares these axes.

Methods for code agents and executable environments. LLM agents combine reasoning, acting, and coding interfaces (Yao et al., 2023; Yang et al., 2024; Wang et al., 2025) with tool-use learning from documentation or synthesized supervision (Schick et al., 2023; Qin et al., 2024; Patil et al., 2024), and recent work treats environments as core infrastructure for agent learning (Huang et al., 2025). One line uses LLMs as textual world models to simulate feedback and state transitions (Ren et al., 2025; Li et al., 2025b;a; Feng et al., 2025), which faces consistency and efficiency limits. Another synthesizes executable environments with sandboxes, databases, tool interfaces, and verification rules (Sullivan et al., 2025; Cai et al., 2025; Song et al., 2026; Wang et al., 2026; Tian et al., 2026; Dong et al., 2026). DSec (Huang et al., 2026a) brings this to production scale, reporting a sandbox platform where agents build environments by snapshotting an interactive session into a reusable image, so construction, RL rollouts, and evaluation share one substrate; it reports from deployment that final-output checks alone cannot establish whether a task was solved as intended. Such platforms make environments *available* at scale, and the RL infrastructure above them (Sheng et al., 2025) consumes them as black boxes. Neither layer measures whether a generated environment is semantically correct. We measure how faithfully coding agents generate verifiable worlds, evaluating what they produce by executing it.

3 CodeWorldBench

CodeWorldBench measures how well coding agents construct and repair executable, stateful code worlds from task descriptions. Hidden test cases verify behavior over both returned outputs and persistent state. We describe the task formulation in Sec. 3.1, the construction pipeline in Sec. 3.2, benchmark settings in Sec. 3.3, and static analysis in Sec. 3.4.

3.1 Task Formulation

A code world is formulated as $\mathcal{W} = (\mathcal{D}, \mathcal{S}, \mathcal{T}, \mathcal{V})$, where $\mathcal{D}$ is the seed database, $\mathcal{S}$ the runtime state space, $\mathcal{T}$ the tool set, and $\mathcal{V}$ the hidden verification rules. Tools are partitioned into read-only query tools $\mathcal{T}_Q$ and state-mutating tools $\mathcal{T}_S$. Each setting q exposes a tool subset $\mathcal{T}_q \subseteq \mathcal{T}$ and hidden test cases (t, x) with $t \in \mathcal{T}_q$ and arguments x. Both reference and submitted worlds initialize from

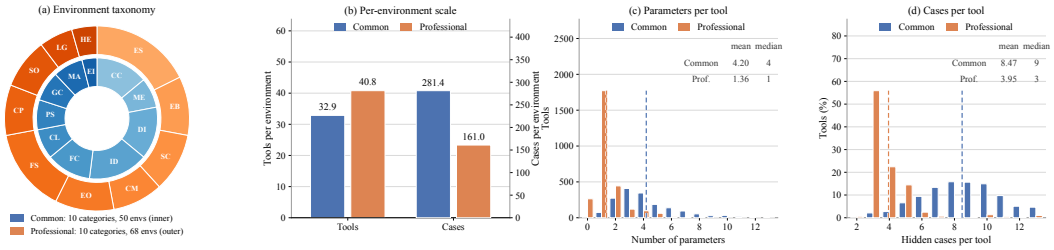


Figure 2: Static profile. (a) Category distribution (inner ring: common, outer: professional). (b)–(d) Tool-surface size, interface arity, and hidden-test depth per subset.

$s_0 = \mathrm{Init}(\mathcal{D})$, yielding $(s, y) = t(s_0, x)$ versus $(\hat{s}, \hat{y}) = \hat{t}(s_0, x)$ from agent implementation $\hat{t}$. A case passes if $\hat{y} = y$ and, for $t \in \mathcal{T}_S$, $\hat{s}$ matches s on every field designated by $\mathcal{V}$.

3.2 Code World Construction

GPT-5.5 builds reference worlds under human-authored contracts and execution-based admission.

Environment Category Selection. We organize candidate stateful domains into common and professional taxonomies, retaining environments requiring coordination across records or tools with output–state verification, while filtering out static lookups and single-function tasks. Category names and abbreviations appear in Tables 2 and 3; Appendix A.5 lists the environments in each category.

Environment Definition. Each retained category is governed by a contract $\Sigma = (\Omega, \mathcal{E}, N, \mathcal{S}, \mathcal{T}, \mathcal{L}, \mathcal{V}, \mathcal{X}, \mathcal{G})$, comprising overview Ω, entity set $\mathcal{E}$, data scale N, state schema $\mathcal{S}$, tool suite $\mathcal{T}$, tier assignment $\mathcal{L}$, validation rules $\mathcal{V}$, examples $\mathcal{X}$, and seed specification $\mathcal{G}$. For every tool, Σ details arguments, validation order, error conditions, mutations, and cross-entity side effects, finalized only when schemas, tools, and data plans align.

Seed-State Construction. Guided by Σ, we construct the seed database partitioned by entity as $\mathcal{D} = \bigcup_{e \in \mathcal{E}} \mathcal{R}_e$, where $\mathcal{R}_e = \{r_{e,1}, \ldots, r_{e,n_e}\}$ denotes instantiated records at scale N. The generator strictly enforces foreign-key, temporal, and status relationships, ensuring the seed state supports queries, pagination, conflicts, and state transitions across both success and failure branches.

Executable Environment Generation. Executable environments are generated via $\widehat{\mathcal{W}} = \Gamma(\Sigma, \mathcal{D})$. Procedure Γ establishes shared state management, tool implementations, executable tests, and behavioral notes, ensuring mutual agreement on validation logic, error branches, and event mutations. Environments adopt standard layouts for state, tools, and tests, register all tools as an MCP server, and bundle verification suites for state and events.

Quality Filtering and Final Verification. Query tools are audited for filtering, pagination, and invalid inputs; stateful tools for preconditions, idempotency, conflicts, and side effects. Two rounds of skill-guided post-generation validation iteratively repair both environment implementations and test cases using unit tests, state/event validation, and tool-list consistency checks. Worlds are admitted only after every tool passes these checks and then converted into GEN repositories; ambiguous or weakly covered worlds are repaired or discarded.

3.3 Benchmark Settings

Each environment yields two complementary settings sharing the base environment: GEN (construction) and SWE (repair). We report them side by side rather than averaged.

GEN: World Construction. The agent implements the code world from scratch in a workspace containing initial data and tool documentation, with contracts specifying entities, tool surfaces, and behavior. Hidden tests execute against reset seed states, evaluating case pass rates across three nested slices: L1 (easiest 25%), L2 (easiest 50%), and L3 (full surface; reported in main tables).

SWE: Semantic Repair. SWE evaluates semantic repair on runnable but defective repositories generated by Qwen3.6-27B under OpenHands (Wang et al., 2025). The defects arise during generation. We preserve the generated tool surfaces and seed states and withhold verification artifacts to prevent leakage (Huang et al., 2026a). We report fail-to-pass and pass-to-pass rates separately to decouple bug fixes from regression avoidance.

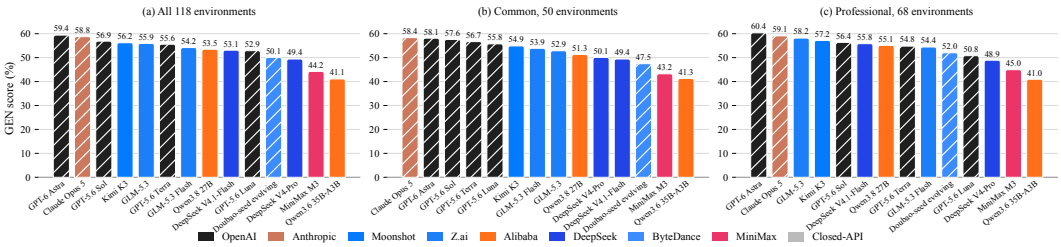


Figure 3: GEN scores macro-averaged across (a) all 118 environments, (b) the 50 common environments, and (c) the 68 professional environments. Each panel is ordered independently, bar colour is the model's creator, and hatching marks closed-source models. OpenAI models run under Codex and every other model under Claude Code.

3.4 Static Analysis of CodeWorldBench

CodeWorldBench spans 118 environments (50 common, 68 professional) with 4,417 tools (32.9 and 40.8 per env, respectively) and 25,015 hidden test cases (14,069 common, 10,946 professional; SWE splits into 9,753 pass-to-pass and 15,262 fail-to-pass). Fig. 2 summarizes surface sizes, arities, and test depths. State representations span static tables, complex configurations, and event-driven databases. By verifying both output values and persistent state, hidden tests isolate whether failures stem from basic construction or state preservation.

4 Experiments

We evaluate CodeWorldBench in the two parallel settings defined in Section 3.3. A run counts as successful only when the generated repository passes the hidden evaluator, which initializes the code world, executes hidden tool cases, and checks both tool outputs and resulting states.

4.1 Experimental Setup

Models. The closed-source set comprises GPT-5.6 Sol, GPT-5.6 Luna, GPT-5.6 Terra, and GPT-6 Astra (OpenAI, 2026b); Claude Opus 5 (Anthropic, 2026); and Doubao-seed-evolving (Volcengine, 2026). The open-source set comprises Qwen3.8-27B (Qwen Team, 2026b) and Qwen3.6-35B-A3B (Qwen Team, 2026a); MiniMax-M3 (MiniMax AI, 2026); DeepSeek-v4-pro (DeepSeek-AI, 2026a) and DeepSeek-v4.1-flash (DeepSeek-AI, 2026b); GLM-5.3-Flash (Z.ai, 2026b) and GLM-5.3 (Z.ai, 2026a); and Kimi K3 (Moonshot AI, 2026).

Scaffolds. OpenAI models use Codex. Other models are evaluated under Claude Code. OpenCode (OpenCode Team, 2026) and OpenHands (Wang et al., 2025) are retained only for Qwen3.8-27B and Qwen3.6-35B-A3B, and the Qwen native scaffold remains in the matrix for both. The model receives only the setting's task document and code-world folder.

Inference Settings. Appendix A.11 gives context and output budgets, sampling parameters, scaffold versions, and model-specific reasoning settings.

Metrics. GEN is the macro-average of per-environment case pass rates (passed cases divided by total cases), giving each environment equal weight. SWE pools case counts across environments: P2P divides still-passing cases by initially passing cases, and F2P divides newly passing cases by initially failing cases. These micro-averages give each case equal weight. Scores are percentages over the reported subset.

4.2 CodeWorldBench Main Results

Tables 2 and 3 report the common and professional subsets separately, listing closed-source models first and open-source models second, with one row per model-scaffold combination. Open-source families are ordered by ascending maximum reported total parameter count, and models within each family likewise. We use the published backbone count for DeepSeek-v4.1-flash, which excludes its additional Engram memory and auxiliary components.

Construction is the bottleneck, and preservation hides it. The two SWE columns behave nothing alike. Preserving already-correct behavior averages 96.5 on common and 94.0 on professional, with every row above 90 on common and all but one on professional, so a protocol scoring only

Table 2: Performance on the 50 common environments across ten categories: Communication Collaboration (CC), Media Entertainment (ME), Data Intelligence (DI), Infrastructure Development (ID), Financial Compliance (FC), Commerce Logistics (CL), Public Service (PS), Geospatial Construction (GC), Manufacturing Agriculture (MA), and Embodied Intelligence (EI). GEN and category scores macro-average per-environment case pass rates; SWE micro-averages cases for pass-to-pass (P2P) preservation and fail-to-pass (F2P) repair. Scores are percentages.

Provider	Model	Params. (total/active)	Scaffold	Category										Setting		
				CC	ME	DI	ID	FC	CL	PS	GC	MA	EI	GEN	SWE	
															P2P	F2P
Closed-Source Models																
OpenAI	GPT-5.6 Sol	🔒	Codex	57.97	40.78	70.09	50.51	41.69	62.92	49.41	68.73	34.38	72.16	57.59	92.36	19.09
	GPT-5.6 Luna	🔒	Codex	57.97	42.31	60.37	53.86	45.53	60.55	50.77	62.89	33.75	67.22	55.75	96.27	18.61
	GPT-5.6 Terra	🔒	Codex	55.92	39.15	60.47	56.88	43.46	60.82	52.66	71.56	33.66	71.98	56.69	96.52	22.23
	GPT-6 Astra	🔒	Codex	66.17	42.13	63.60	56.45	53.99	62.74	57.43	79.30	34.47	69.05	58.13	90.83	25.16
Anthropic	Claude Opus 5	🔒	Claude Code	67.20	39.78	61.63	55.94	47.35	57.08	63.01	77.49	34.47	72.53	58.35	98.04	22.66
ByteDance	Doubao-seed-evolving	🔒	Claude Code	53.25	36.62	45.72	48.51	38.10	39.09	36.36	47.08	33.03	66.67	47.46	96.81	16.38
Open-Source Models																
Alibaba	Qwen3.8-27B	27B	Claude Code	54.95	36.71	59.32	51.91	40.35	55.71	45.36	60.57	33.48	65.93	51.32	98.09	14.53
			OpenCode	63.66	39.60	58.76	55.05	44.19	54.79	51.94	73.54	33.48	67.40	55.42	96.88	13.39
			OpenHands	54.54	37.25	58.96	54.46	44.00	50.96	49.32	60.82	33.75	66.48	52.28	91.08	11.54
			Qwen Code	55.00	34.63	54.38	51.40	39.62	52.24	42.21	67.96	34.20	66.85	49.89	93.18	11.99
	Qwen3.6-35B-A3B	35B/3B	Claude Code	49.00	32.73	50.25	42.74	35.36	49.04	28.80	54.04	27.63	51.47	41.27	98.99	12.69
			OpenCode	49.72	29.39	42.25	35.19	32.87	31.87	28.26	56.10	27.00	54.58	39.43	99.62	12.47
			Qwen Code	24.19	31.65	37.71	39.94	31.77	44.66	17.95	52.66	29.61	59.16	36.14	99.25	11.81
MiniMax	MiniMax-M3	428B/23B	Claude Code	46.44	35.99	54.83	42.70	23.80	30.32	31.95	57.99	19.44	60.44	43.24	94.67	13.09
Z.ai	GLM-5.3-Flash	320B/18B	Claude Code	54.95	37.16	59.72	50.25	45.34	59.45	46.71	73.28	32.94	66.30	53.89	97.72	16.89
	GLM-5.3	744B/40B	Claude Code	62.02	40.78	60.27	50.34	40.54	52.15	40.59	72.68	33.48	67.58	52.87	99.45	16.65
DeepSeek	DeepSeek-v4.1-flash	552B/8B-16B	Claude Code	54.07	38.79	58.61	49.53	35.48	53.79	36.90	73.88	33.21	64.29	49.42	99.23	18.51
	DeepSeek-v4-pro	1.6T/49B	Claude Code	54.13	36.53	59.42	49.19	40.84	55.98	45.99	60.31	32.85	67.03	50.10	96.08	18.86
Moonshot	Kimi K3	2.8T/104B	Claude Code	63.45	38.07	59.57	52.50	42.00	57.26	40.23	73.63	33.48	65.57	54.90	98.77	17.42

surviving tests would call this solved. Repair does not follow: mean F2P is 16.5 and 25.2, while construction sits between them at 50.7 and 51.9 mean GEN over a 22-point spread in both pools. Nor is repair ordered by construction. The strongest professional F2P rows are Qwen3.8-27B under Qwen Code (45.3) and Claude Code (44.9), above every closed-source row, where Claude Opus 5 leads at 33.5, yet those same rows sit twelfth and seventh on GEN; this is not a family effect, since Qwen3.6-35B-A3B repairs at 8.6-10.7. Construction and repair expose distinct capabilities.

The two pools agree on the ordering, and the scaffold moves a model less than the model does. GPT-6 Astra and Claude Opus 5 lead GEN in both pools. GLM-5.3 and DeepSeek-v4.1-flash gain 5.3 and 6.4 points on professional, while GPT-5.6 Luna loses 4.9. Professional GEN spreads across scaffolds are 2.0 points for Qwen3.8-27B and 3.0 for Qwen3.6-35B-A3B (5.5 and 5.1 on common), against a 14.8-point professional gap between the two families (13.3 on common). Repair is more scaffold-sensitive: professional F2P for Qwen3.8-27B ranges from 29.2 to 45.3 across harnesses.

Category strengths differ from overall rankings. Across the 14 models under the default scaffolds in Figure 3, common category scores average 66.7 in Geospatial Construction and 32.2 in Manufacturing Agriculture; professional averages are 71.5 in Supply Operations and 28.4 in Enterprise Operations. Category rankings also differ from GEN: MiniMax-M3 leads Enterprise Operations (39.75), ahead of GPT-6 Astra (24.26), despite lower overall GEN (44.96 vs. 60.36).

5 Analysis

Figure 3 shows that CodeWorldBench separates code fluency from construction quality: closed-source systems lead overall, but both model choice and scaffold shape the outcome.

5.1 Agentic Construction Behavior

Holding the scaffold fixed, we compare Qwen-family OpenHands runs across the three GEN levels on the 50 common environments. Macro-averaged case pass rate drops from 0.4580 (L1) to 0.4370 (L2) and 0.4069 (L3) while verified surface roughly doubles at each step—four times the behavior costs about five points. Because all 50 environments score strictly between zero and one at every level, slices serve as diagnostics rather than difficulty gates. This decline is driven by stateful coupling

Table 3: Performance on the 68 professional environments across ten categories: Engineering Systems (ES), Embedded Systems (EB), Scientific Computing (SC), Creative Media (CM), Enterprise Operations (EO), Financial Services (FS), Commercial Platforms (CP), Supply Operations (SO), Legal Governance (LG), and Health Education (HE). GEN and category scores macro-average per-environment case pass rates; SWE micro-averages cases for pass-to-pass (P2P) preservation and fail-to-pass (F2P) repair. Scores are percentages.

Provider	Model	Params. (total/active)	Scaffold	Category										Setting		
				ES	EB	SC	CM	EO	FS	CP	SO	LG	HE	GEN	SWE P2P	SWE F2P
Closed-Source Models																
OpenAI	GPT-5.6 Sol	🔒	Codex	68.50	68.64	54.13	47.61	30.18	51.77	68.48	67.06	38.91	56.03	56.37	91.49	23.65
	GPT-5.6 Luna	🔒	Codex	45.22	65.17	53.40	42.14	29.45	58.29	62.43	60.56	35.18	53.99	50.81	88.03	24.43
	GPT-5.6 Terra	🔒	Codex	72.18	53.70	41.92	46.90	22.63	61.20	68.28	67.42	39.73	55.62	54.81	94.33	26.26
	GPT-6 Astra	🔒	Codex	77.17	52.68	47.68	57.03	24.26	68.15	77.03	82.05	39.70	56.44	60.36	91.19	31.90
Anthropic	Claude Opus 5	🔒	Claude Code	68.44	56.28	36.07	56.34	32.19	63.76	91.61	74.34	40.03	64.59	59.09	96.42	33.52
ByteDance	Doubao-seed-evolving	🔒	Claude Code	54.90	47.31	51.82	50.89	20.33	55.19	62.45	81.81	39.89	53.59	52.03	94.54	25.29
Open-Source Models																
Alibaba	Qwen3.8-27B	27B	Claude Code	61.67	56.36	50.44	42.88	32.84	55.14	70.26	81.66	35.71	55.84	55.12	93.92	44.86
			OpenCode	55.86	71.41	53.52	45.89	31.35	51.88	71.07	57.10	37.17	56.04	53.59	94.32	29.24
			OpenHands	66.52	81.25	52.73	30.59	31.26	44.52	68.78	57.79	38.79	56.03	53.92	96.85	39.10
			Qwen Code	60.58	62.33	38.21	49.04	28.82	46.42	74.84	67.36	49.78	56.04	53.11	90.65	45.32
	Qwen3.6-35B-A3B	35B/3B	Claude Code	34.58	46.96	46.13	30.71	15.12	56.68	47.57	57.52	33.44	32.82	40.97	91.71	10.40
			OpenCode	28.71	50.79	37.06	24.66	12.73	46.44	35.47	81.61	33.63	37.45	38.39	96.23	8.61
			Qwen Code	42.42	39.96	28.42	29.28	32.37	40.35	45.58	41.23	31.38	47.98	38.00	93.12	10.72
MiniMax	MiniMax-M3	428B/23B	Claude Code	42.79	55.18	42.32	19.68	39.75	51.04	60.13	56.36	33.07	41.06	44.96	94.15	8.88
Z.ai	GLM-5.3-Flash	320B/18B	Claude Code	60.70	61.71	52.05	55.73	31.27	62.63	70.26	57.27	35.06	30.64	54.44	95.24	21.91
	GLM-5.3	744B/40B	Claude Code	65.61	70.17	51.13	51.63	29.83	56.56	71.41	82.96	35.39	55.64	58.17	94.82	24.64
DeepSeek	DeepSeek-v4.1-flash	552B/8B-16B	Claude Code	60.46	61.50	53.85	55.42	28.92	60.94	52.28	82.44	35.71	56.04	55.83	96.22	18.72
	DeepSeek-v4-pro	1.6T/49B	Claude Code	49.02	53.49	51.01	47.27	29.96	50.27	55.40	76.14	13.49	55.84	48.92	97.23	24.57
Moonshot	Kimi K3	2.8T/104B	Claude Code	61.01	66.54	49.54	50.86	31.33	61.70	68.04	72.91	40.60	64.99	57.19	95.85	26.61

rather than prompt length: higher levels demand sharing records, maintaining derived views, and coordinating side effects across calls. Crucially, failures are semantic rather than structural. Generated repositories pass structural setup but mismanage state transitions (e.g., stale counters, shallow success envelopes, inconsistent views) because agents build first—writing managers and registries before validating transition rules. More code does not help, and the strongest run writes less.

5.2 The Scaffold Shapes Behavior More Than the Model

Parsing the deduplicated archive of 1,619 trajectories allows a controlled comparison across scaffolds. Because scaffolds define turns differently, we measure interaction via tool calls. Figure 4 holds models fixed (Qwen3.8-27B and Qwen3.6-35B-A3B) and varies the scaffold across all 118 environments.

Tool mix is a property of the harness, not the model. Role distributions (Figure 4a) cluster by scaffold. Under Claude Code, both spend 54–68% of GEN calls in shell versus 2–4% on edits (edit-to-execution ratio ≈ 0.05); Qwen Code inverts this to 1.63 and 1.71, while OpenHands lands in between. Harness determines style; leaderboards mixing scaffolds partly benchmark harnesses.

Volume varies by scaffold at fixed capability. Mean GEN calls per run (Figure 4b) span 93.3–174.2—a 1.9$\times$ variance driven solely by the harness, with individual runs hitting 500 calls. SWE requires more interaction than GEN in four of six setups (ratio ≈ 1.4). The exceptions occur under Qwen Code, where reading-heavy GEN runs prepay comprehension costs.

Errors shift from the agent's own code to the tool interface. Error taxonomy (Figure 4c) also shifts by scaffold. Under Claude Code, assertion failures in self-written probe scripts account for $\sim$60% of errors. Under OpenHands and Qwen Code, interface friction (tool misuse, missing paths) dominates. Absolute error rates stay within 2.1–7.0%. This matters because internal code failures lie on the path to semantic correctness, whereas tool misuse merely pads trace length.

5.3 Complete Trace Analysis: How Agents Spend Their Turns

Every score marks the endpoint of a trajectory. We analyze 600 traces from six families across the 50 common environments in GEN L3 and SWE: two closed-source (GPT-6 Astra, Claude Opus 5) and four open-source models (Kimi K3, DeepSeek-v4-pro, GLM-5.3, Qwen3.8-27B). We normalize

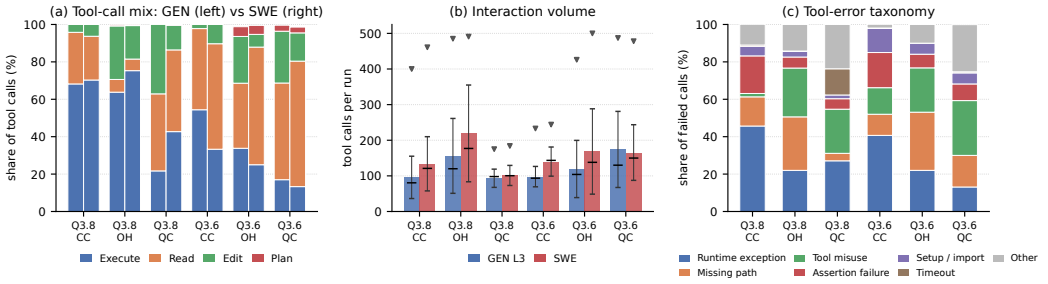


Figure 4: Scaffold view: Qwen3.8-27B (Q3.8) and Qwen3.6-35B-A3B (Q3.6) under Claude Code (CC), OpenHands (OH), and Qwen Code (QC), over 118 environments in both settings. (a) Role mix, GEN L3 left and SWE right. (b) Calls per run: bars means, whiskers one standard deviation, ticks medians, triangles the longest run. (c) Error taxonomy, pooled over both settings.

actions to *operations* and harness results to *outcomes*. Claude Opus 5 shares the open-source harness, isolating model effects from harness effects where the closed-source pair aligns.

Closed-source models act on the world; open-source models read about it. Both groups are execution-heavy but diverge elsewhere (Figure 5a). On GEN L3, closed-source models spend 86% on execution, 11% on edits, and 2% on reads; open-source models spend 23% reading and 5% editing. This reflects policy rather than budget: Claude Opus 5 reads files in $< 6\%$ of operations, while Kimi K3 spends 36.5% of GEN reading, with a 3.45-point gap in common GEN. One verifies by executing, the other by inspecting. For all Claude Code models, SWE shifts edit share upward to 7.3–14.5%, as targeted modification replaces probing.

Repair costs more than construction, and no one buys accuracy with turns. Volume inversely tracks capability (Figure 5b): Claude Opus 5 finishes GEN in 66.8 operations, while Qwen3.8-27B requires 139.2 for a lower score. SWE incurs a repair surcharge of 1.1–1.6$\times$ in operations and reported cost ($5.64–$12.60 for GEN vs. $8.20–$18.77 for SWE). Maintaining invariants requires reverse-engineering them, making SWE costlier than construction. Extra turns do not buy accuracy: across six of eight Claude Code cells, the top volume quartile scores *below* the bottom quartile (by up to 12.0 points on GEN and 9.7 on F2P). Long traces signal struggle rather than thoroughness.

Only one model habitually runs a real test suite. Ad-hoc checks are common, but structured test suites are not (Figure 5c). Appendix A.9 reports probe usage by model and setting. GPT-6 Astra runs test suites in 70% of GEN and 96% of SWE runs; other models do so in only 2.2% of runs, and never for Claude Opus 5 or GLM-5.3 on GEN. This habit drops sharply under SWE for Claude Code models. Ad-hoc scripts verify single runs but leave no reusable artifacts, failing to catch subtle transition regressions where actions return success but write invalid state.

The two groups fail at different layers. Error distributions (Figure 5d) split similarly. Open-source failures stem mainly from self-written code (80% GEN, 73% SWE vs. 10% and 14% interface errors). For closed-source models under SWE, interface issues (tool misuse, missing paths, timeouts) rise to 43% while own-code errors fall to 24%, with GPT-6 Astra timing out on 25% of failed SWE calls due to full test runs. Overall, failures remain shallow (2.6–9.2% of outcomes, error streaks averaging 1.0–2.7 calls) and weakly correlated with score: agents fail smoothly rather than catastrophically.

Three targets follow from where these failures land, none requiring a change to the underlying model: supervise state rather than output, require self-verification before submission, and treat shared invariants as an artifact rather than re-deriving them per tool. Appendix A.10 develops each.

5.4 Does Construction Quality Reach a Downstream Agent?

Generated environments serve as substrates for downstream training and evaluation, making construction quality paramount. We evaluate 387 reference-authored tasks across the 36 common GEN environments successfully built by three generators under OpenCode. Qwen3.8-27B solves all tasks via smolagents (Roucher et al., 2025) (temperature 0, $\leq$30 calls) under seven conditions: the reference environment, and each generated environment with and without a GLM-5.3-Flash-authored skill document. Table 4 reports grounded pass rate (*Grd.*), call success rate (*Call*), never-successful tools

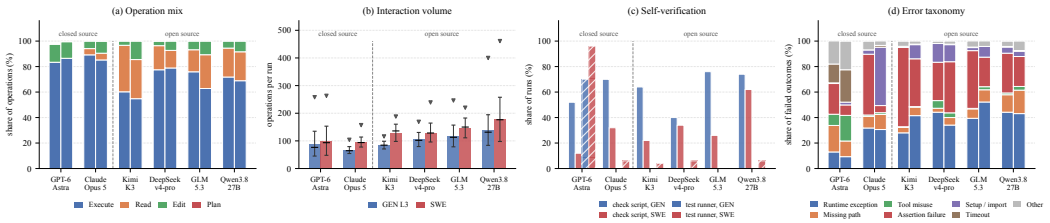


Figure 5: Complete trace view: two closed-source and four open-source models over the 50 common environments, GEN L3 (left bar) and SWE (right bar). Operations are request-side actions, outcomes harness-reported results, normalized so Codex and Claude Code are comparable. (a) Operation mix. (b) Operations per run, as in Figure 4b. (c) Runs executing a self-written check script (solid) or a test runner (hatched). (d) Error taxonomy as a share of failed outcomes.

Table 4: Downstream use of generated environments. Metrics compare *w/o* vs. *w/* skill document; + indicates an increase, while − indicates a decrease. Header arrows show the better direction. *Grd.* is the grounded pass rate, *Call* the call success rate, *Nev.* the number of never-successful tools, *NoTool* the share of failures caused by an absent tool, *Bdgt* the share of attempts that exhaust the budget, and *M-Err* the model-side error rate excluding code crashes.

(a) Frozen tasks in model's own env.

Environment	Grd. ↑		Call ↑		Nev. ↓		NoTool ↓		Bdgt ↓	
	w/o	w/	w/o	w/	w/o	w/	w/o	w/	w/o	w/
Reference	**49.6**	–	78.0	–	6	–	0	–	2.6	–
GLM-5.2	24.8	25.3^{+}	58.2	67.2^{+}	8	7^{-}	5	0^{-}	10.9	6.5^{-}
Qwen3.6-27B	18.1	19.4^{+}	49.1	55.0^{+}	15	14^{-}	16	15^{-}	12.4	5.9^{-}
Qwen3.6-35B-A3B	6.2	7.0^{+}	26.8	36.8^{+}	54	45^{-}	33	29^{-}	24.0	10.3^{-}

(b) 36 environments in terciles.

Env. Tercile	Grd. ↑		Call ↑		M-Err ↓		Bdgt ↓	
	w/o	w/	w/o	w/	w/o	w/	w/o	w/
Low	2.6	2.8^{+}	37.4	40.1^{+}	34.7	30.2^{-}	24.5	10.3^{-}
Mid	15.8	16.8^{+}	46.7	53.3^{+}	27.5	24.7^{-}	12.9	7.7^{-}
High	32.7	34.0^{+}	**44.0**	**67.7**$^{+}$	**37.5**	**21.1**$^{-}$	10.5	4.4^{-}

(*Nev.*), absent-tool failures (*NoTool*), budget exhaustion (*Bdgt*), and model-side error rate (*M-Err*, excluding code crashes). Full definitions are in Appendix A.3.

Construction quality governs downstream transfer. The reference environment reaches 49.6% grounded pass, whereas generated environments retain only one-eighth to one-half of that (24.8%, 18.1%, and 6.2%), mirroring their construction scores. Failures differ: reference failures stem almost entirely from task difficulty (all required tools exist), whereas generated environments introduce upstream-absent defects—missing tools cause up to 33% of task failures and 27.7% of call invocations in the weakest generator, which omits 332 tools and invents 203 of ∼1,250 reference tools.

Skills improve execution efficiency, capped by the environment. Skill documents provide efficiency rather than task completion: call success increases and budget exhaustion roughly halves, but pass-rate gains remain marginal ($\leq$1.3 points). Tercile stratification (Table 4b) reveals that high-quality environments benefit disproportionately (9$\times$ larger call-success gain than the bottom tercile, alongside drops in model errors). While documentation mitigates overhead, it cannot recover omitted tools—the environment sets the ceiling, while skills dictate how closely agents reach it.

6 Conclusion

We introduce CodeWorldBench to evaluate whether coding agents can construct and repair executable, stateful environments faithful to their specifications. By verifying persistent states alongside tool return values, CodeWorldBench exposes semantic errors that standard code benchmarks overlook. Our experiments reveal a capability gap: while frontier agents reliably preserve existing behavior, they struggle to synthesize correct state transitions from scratch or repair semantic defects. These defects propagate to downstream tasks, causing agents operating inside generated environments to suffer performance drops due to crashing, inconsistent, or missing tools. These findings demonstrate that building stateful environments is an unsolved challenge, highlighting the need for future coding agents to incorporate rigorous state reasoning and automated self-verification.

AI USE STATEMENT

Generative AI tools were used to improve the wording, grammar, and readability of the manuscript. They were also used as components of our experimental pipeline, including API-based model evaluation and the generation of benchmark-related artifacts. AI-generated artifacts were subsequently checked through our automated validation and execution-based verification procedures. The authors reviewed the final manuscript and experimental results and take full responsibility for the content of this paper.

ETHICS STATEMENT

This work does not involve human subjects, human annotation, or the collection of personal data. CodeWorldBench is constructed from synthetic executable environments and does not rely on private or personally identifiable information. The experiments focus on evaluating the capabilities and limitations of coding agents in controlled benchmark settings. We do not identify additional ethical concerns specific to the data or experimental procedures used in this work.

REPRODUCIBILITY STATEMENT

We provide detailed descriptions of the benchmark construction, experimental settings, evaluated models and scaffolds, evaluation metrics, and analysis procedures in Sections 3-5. Appendix A further documents the evaluation settings, task structure, benchmark artifacts, and generation and conversion procedures. We plan to release the benchmark data, reference implementations, and evaluation artifacts to facilitate reproduction and further study of the reported results.

REFERENCES

Anthropic. Introducing the model context protocol. https://www.anthropic.com/news/model-context-protocol, 2024. Accessed: 2026-05-07.

Anthropic. Introducing claude opus 5. https://www.anthropic.com/news/claude-opus-5, 2026.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Shihao Cai, Runnan Fang, Jialong Wu, Baixuan Li, Xinyu Wang, Yong Jiang, Liangcai Su, Liwen Zhang, Wenbiao Yin, Zhen Zhang, Fuli Feng, Pengjun Xie, and Xiaobin Wang. Autoforge: Automated environment synthesis for agentic reinforcement learning. *arXiv preprint arXiv:2512.22857*, 2025.

Hyungjoo Chae, Namyoung Kim, Kai Tzu-iunn Ong, Minju Gwak, Gwanwoo Song, Jihoon Kim, Sunghwan Kim, Dongha Lee, and Jinyoung Yeo. Web agents with world models: Learning and leveraging environment dynamics in web navigation. In *International Conference on Learning Representations*, 2025.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Aleksander Madry, and Lilian Weng. Mlebench: Evaluating machine learning agents on machine learning engineering. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=6s5uXNWGIh.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026a. URL https://arxiv.org/abs/2606.19348.

DeepSeek-AI. DeepSeek-V4.1-Flash: Pushing the limits of KV cache compression. https://huggingface.co/deepseek-ai/DeepSeek-V4.1-Flash, 2026b.

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, Weiran Shi, Zaiyuan Wang, Daoguang Zan, Chenchen Zhang, Xiaoxu Zhang, Qizhi Chen, Xianfu Cheng, Bo Deng, Qingshui Gu, Kai Hua, Juntao Lin, Pai Liu, Mingchen Li, Xuanguang Pan, Zifan Peng, Yujia Qin, Yong Shan, Zhewen Tan, Weihao Xie, Zihan Wang, Yishuo Yuan, Jiayu Zhang, Enduo Zhao, Yunfei Zhao, He Zhu, Chenyang Zou, Ming Ding, Jianpeng Jiao, Jiaheng Liu, Minghao Liu, Qian Liu, Chongyao Tao, Jian Yang, Tong Yang, Zhaoxiang Zhang, Xinjie Chen, Wenhao Huang, and Ge Zhang. NL2Repo-Bench: Towards long-horizon repository generation evaluation of coding agents. *arXiv preprint arXiv:2512.12730*, 2025.

Guanting Dong, Junting Lu, Junjie Huang, Wanjun Zhong, Longxiang Liu, Shijue Huang, Zhenyu Li, Yang Zhao, Xiaoshuai Song, Xiaoxi Li, et al. Agent-world: Scaling real-world environment synthesis for evolving general agent intelligence. *arXiv preprint arXiv:2604.18292*, 2026.

Jichen Feng, Yifan Zhang, Chenggong Zhang, Yifu Lu, Shilong Liu, and Mengdi Wang. Web world models. *arXiv preprint arXiv:2512.23676*, 2025.

Jialiang Huang, Hongxuan Tang, Jingchang Chen, Yuxuan Liu, Yixiao Chen, Yuan Cheng, Yi Tao, Jingli Zhou, Yupeng Chen, Haoyu Chen, Jiarui Wang, Shengkai Lin, Chuqi Zhang, Bryan Lee Teng, Lian Guo, Zhe Fu, Wenjun Gao, Yisong Wang, Liang Zhao, Zehao Wang, Ziwei Xie, Yongqiang Guo, Peixin Cong, Ziyi Gao, Shuiping Yu, Hanwei Xu, Zuofan Wu, Zhizhou Ren, Yuyang Zhou, Bowei Zhang, Zhihuan Huang, Qihao Zhu, Lei Wang, Tianle Lin, Han Yu, Jiewen Hu, Dejian Yang, Shuo Yang, Shanghao Lu, Shaoyuan Chen, Junjie Qiu, Zhangli Sha, Yinmin Zhong, Yongtong Wu, Shiyu Wang, Wei Liu, Bingzheng Xu, Longhao Chen, Qiushi Du, Yuzhen Huang, Shirong Ma, Yaohui Wang, Mingshu Chen, Tongrui Xiong, Y. C. Yan, Haowen Luo, Haofen Liang, Xiaokang Zhang, Weihao Zeng, Runxin Xu, Peiyi Wang, Jinhua Zhu, Ruoyu Zhang, Wenkai Yang, Qi Tang, Jiping Yu, Tian Ye, Ruizhe Pan, Honghui Ding, Xiaodong Liu, Lingxiao Luo, Zhihong Shao, Yuhan Wu, Jibai Lu, Wen Liu, Haoling Zhang, Jingcheng Hu, Yaoyang Ye, Chaofan Lin, Zhaochen Zhang, Jianan Tong, Hengxu Wu, Zhihao Li, Yicheng Wang, Luyao Wang, Yuzhuo Bai, Lingyue Fu, Ruifan Xu, Y. Z. Wang, Zonglin Li, Mingqi Wei, Haiyang Shen, Chengyuan Zhang, Chao Jin, Zili Zhang, R. H. Yang, Xinbo Xu, Jian Zhou, Ruidong Zhu, Yuzhe Guo, Zelun Pan, Shaoheng Nie, Erhang Li, Shuhan Lin, Zheng Liu, Anshuo Chen, Zilong Lyu, Sinuo Cao, Rui Yu, Chuhao Wang, Junyi Guo, Junxiao Song, Kaifeng Chen, Menghao Ye, Junxian Li, Di Wu, Haiyang Ma, Yilun Wang, Haoran Yang, Yizai Cai, Shichun Liu, Yiping Wang, Junbo Sun, Shicheng Xu, Xiao Bi, Ying He, Yichao Zhang, Mingxing Zhang, Liyue Zhang, Panpan Huang, and Wenfeng Liang. DeepSeek elastic compute (DSec): A sandbox infrastructure for effective agentic training at scale. *arXiv preprint arXiv:2609.22978*, 2026a. URL https://arxiv.org/abs/2609.22978.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. DeepSWE: Measuring frontier coding agents on original, long-horizon engineering tasks. *arXiv preprint arXiv:2607.07946*, 2026b.

Yuchen Huang, Sijia Li, Zhiyuan Fan, Minghao Liu, Wei Liu, and Yi R. Fung. Scaling environments for LLM agents: Fundamentals, approaches, and future directions. In *Workshop on Scaling Environments for Agents*, 2025. URL https://openreview.net/forum?id=9axZcDTiJm.

Rishyanth Kondra, Sanket Mhatre, Akshit Kumar, Evan Chu, Bilal Bakht Ahmad, Ayush Nangia, Rajan Agarwal, Arpan Dasgupta, Animesh Sinha, Bhuvanesh Sridharan, Krupa Dave, Brendan Graham, Guanyu Song, Anirudh Rahul, Wei Hern Lim, Abishek Thangamuthu, Ramneet Singh, Danna Liu, Navid Pour, Calvin Chen, and Justus Mattern. FrontierSWE v2. https://www.frontierswe.com/blog/v2, September 2026.

Yixia Li, Hongru Wang, Jiahao Qiu, Zhenfei Yin, Dongdong Zhang, Cheng Qian, Zeping Li, Pony Ma, Guanhua Chen, Heng Ji, and Mengdi Wang. From word to world: Can large language models be implicit text-based world models? *arXiv preprint arXiv:2512.18832*, 2025a.

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*, 2025b.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*, 2023.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, Zirui Wang, and Ruoming Pang. ToolSandbox: A stateful, conversational, interactive evaluation benchmark for LLM tool use capabilities. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 1160–1183. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-naacl.65. URL https://aclanthology.org/2025.findings-naacl.65/.

Huanzhi Mao, Raymond Tsao, Jingzhuo Zhou, Shishir G. Patil, and Joseph E. Gonzalez. BFCL V4: Agentic web search. https://gorilla.cs.berkeley.edu/blogs/15_bfcl_v4_web_search.html, 2025. Accessed 2026-04-30.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

MiniMax AI. MiniMax-M3 model card. https://huggingface.co/MiniMaxAI/MiniMax-M3, 2026.

Moonshot AI. Kimi K3 model card. https://huggingface.co/moonshotai/Kimi-K3, 2026.

OpenAI. Introducing SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/, 2024. Accessed 2026-04-30.

OpenAI. Introducing GPT-5.3-Codex. https://openai.com/index/introducing-gpt-5-3-codex/, 2026a.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition. https://openai.com/index/gpt-5-6/, 2026b.

OpenClaw. Openclaw: Personal ai assistant. https://github.com/openclaw/openclaw, 2026. GitHub repository. Accessed: 2026-05-07.

OpenCode Team. OpenCode. https://opencode.ai/, 2026.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive APIs. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=tBRNC6YemY.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=dHng2O0Jjr.

Qwen Team. Qwen3.6 collection. https://huggingface.co/collections/Qwen/qwen36, 2026a.

Qwen Team. Qwen3.8 collection. https://huggingface.co/collections/Qwen/qwen38, 2026b.

Jiawei Ren, Yan Zhuang, Xiaokang Ye, Lingjun Mao, Xuhong He, Jianzhi Shen, Mrinaal Dogra, Yiming Liang, Ruixuan Zhang, Tianai Yue, et al. Simworld: An open-ended realistic simulator for autonomous agents in physical and social worlds. *arXiv preprint arXiv:2512.01078*, 2025.

Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. smolagents: a smol library to build great agentic systems. https://github.com/huggingface/smolagents, 2025.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=Yacmpz84TH.

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

Sierra AI. τ^3-bench: Advancing agent evaluation to knowledge and voice. https://sierra.ai/blog/bench-advancing-agent-benchmarking-to-knowledge-and-voice, 2026.

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

Michael Sullivan, Mareike Hartmann, and Alexander Koller. Procedural environment generation for tool-use agents. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 18544–18562. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-main.936. URL https://aclanthology.org/2025.emnlp-main.936/.

Hao Tang, Darren Key, and Kevin Ellis. Worldcoder, a model-based llm agent: Building world models by writing code and interacting with the environment. In *Advances in Neural Information Processing Systems*, 2024.

Xiaoyu Tian, Haotian Wang, Shuaiting Chen, Hao Zhou, Kaichi Yu, Yudian Zhang, Jade Ouyang, Junxi Yin, Jiong Chen, Baoyan Guo, Lei Zhang, Junjie Tao, Yuansheng Song, Ming Cui, and Chengwei Liu. Astra: Automated synthesis of agentic trajectories and reinforcement arenas. *arXiv preprint arXiv:2601.21558*, 2026.

Volcengine. Doubao-seed-evolving. https://www.volcengine.com/product/doubao, 2026.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. In *International Conference on Machine Learning*, 2024.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

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.

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

John Yang, Kilian Lieret, Jeffrey Jian Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. ProgramBench: Can language models rebuild programs from scratch? *arXiv preprint arXiv:2605.03546*, 2026.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik R. Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=roNSXZpUDN.

Z.ai. GLM-5.3: Frontier coding with emergent cyber capabilities. https://z.ai/blog/glm-5.3, 2026a.

Z.ai. GLM-5.3-Flash model card. https://huggingface.co/zai-org/GLM-5.3-Flash, 2026b.

Zhaoxi Zhang, Yiming Xu, Jiahui Liang, Weikang Li, Xiaoshuai Chen, Liwei Qian, Xin Pei, Jizhou Huang, Run Sun, and Yunfang Wu. RepoZero: Can LLMs generate a code repository from scratch? *arXiv preprint arXiv:2605.07122*, 2026.

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

A Benchmark Settings and Artifacts

This appendix collects the benchmark artifacts as labelled exhibits: the static profile of the hidden suites, the tool-surface slice definition, the downstream skill-transfer protocol and its metric definitions, the evaluation settings, the environment taxonomy, the public task structures for both settings, and the prompt-template view of the skills that generate and convert environments.

A.1 Static Profile of the Hidden Suites

Section 3.4 reports the headline static axes. This subsection gives the axes that describe what a case actually checks, all measured from evaluation assets we plan to release.

Cases check state more often than output. Of the 14,069 common cases, 5,914 carry state assertions and 8,155 check the returned payload alone; on professional the split reverses, 8,105 of 10,946 carrying state assertions. Assertions are not shallow reads: a common case that checks state names 2.77 paths on average and a professional one 8.11, with the deepest naming 76 and 190. Most paths address a field inside a record rather than a top-level counter, and 2.2% to 3.0% assert that a field is *absent*, which is how a case catches a tool that writes a record it should have rejected. Counting tools rather than cases, 69.1% of common tools and 77.1% of professional ones have at least one documented state effect, so the majority of each surface cannot be verified from return values alone.

Some cases need a prepared world. A case may declare setup calls that run before the case's own call, which is how the suite reaches states the seed database does not contain: 2.2% of common cases and 24.9% of professional cases use them, the longest chaining 76 and 16 calls. The difference is a property of the domains, since professional environments start from small seed tables (mean 74 records against 10,068 on common) and must build the state they test.

Error identifiers are a large part of the contract. Exact error identifiers are compared as strings, so they form a substantial share of what an agent must reproduce. A common environment documents 69.9 distinct identifiers on average across 5.0 per tool; professional environments document 10.3 per environment and 2.4 per tool, but spend far more words per tool contract, 542 against 200. The two pools therefore load the same axis differently: common environments demand many precise rejection paths, while professional ones demand long single-tool specifications.

The defect surface is broad, not localized. Generation defects affect most of the SWE tool surface: the failing baseline cases touch 29.8 of 32.9 tools per common environment and 29.7 of 40.8 per professional one, covering most tools in both pools. This is why repair in CodeWorldBench is not the localized-patch problem of issue-based benchmarks, and why F2P and P2P must be reported separately, as an agent can satisfy one while regressing the other across a surface this wide.

A.2 Tool-Surface Slices

For every tool t, we compute a difficulty score over interface complexity, implementation complexity, state and domain coupling, behavioral constraints, and hidden-test depth,

$$d(t) = d_{\text{iface}}(t) + d_{\text{impl}}(t) + d_{\text{state}}(t) + d_{\text{beh}}(t) + d_{\text{test}}(t), \qquad d(t) \in [0, 20]. \tag{1}$$

Tools are sorted by increasing $(d(t), \max_j d_j(t), \text{source_order}(t))$ and grouped into coherent nested slices that retain query support for observing state mutations, subject to

$$|\mathcal{T}_{\text{L1}}| \le \max(1, \lfloor |\mathcal{T}|/4 \rfloor), \qquad \mathcal{T}_{\text{L1}} \subseteq \mathcal{T}_{\text{L2}}, \qquad |\mathcal{T}_{\text{L2}}| \le \max(1, \lfloor |\mathcal{T}|/2 \rfloor), \tag{2}$$

with $\mathcal{T}_{\mathrm{L3}} = \mathcal{T}$. SWE always operates on the full surface. On the common pool the constraints yield 7.8 L1 tools, 16.3 L2 tools, and 32.9 L3 tools per environment, so each slice roughly doubles the verified surface.

A.3 Downstream Skill-Transfer Protocol

Metric definitions. Table 4 reports five quantities per arm. *Grd.* is the grounded pass rate: the share of tasks whose final database state matches the oracle's on every grounded field, with no unintended row modified. *Call* is the call success rate, the share of all tool invocations that return successfully, which measures execution efficiency rather than task completion. *Nev.* counts tools that never once returned successfully across all attempts, providing a diagnostic of overall environment usability. *NoTool* is the share of failures whose cause is that a tool the task requires is missing from that environment. *Bdgt* is the share of attempts that exhaust the 30-call budget without reaching an accepting state. In Table 4b, *M-Err* is the model-side call error rate: invocations that fail because the solver called a non-existent tool, passed rejected arguments, referenced a missing entity, or violated a documented rule. Crashes *inside* generated tool code are counted separately and excluded from M-Err, distinguishing invocation patterns from runtime defects.

Grounded scoring. Scoring compares the database rows an attempt leaves behind against the oracle's, restricted to the fields the task text grounds. An ID field counts only when that ID appears in the task text; a string field requires both its value and its name to appear. Free-text and timestamp fields are excluded because they record the oracle's particular wording rather than the task's requirement. Grounded evaluation therefore verifies whether the task specification was faithfully met, rather than requiring byte-for-byte replication of the oracle trace.

The budget is not binding. Oracle traces average 2.57 steps (median 2, max 13), use 2.42 distinct tools, and every oracle step succeeds, indicating that canonical trajectories are concise. 66% of tasks need two steps and 95% finish within four, providing ample headroom within the 30-call limit. Budget exhaustion therefore primarily reflects iterative recovery behavior against failing calls under a standardized resource constraint.

Gradients hold inside each difficulty band. Stratifying tasks by oracle length demonstrates consistent relative performance across environments. On 1–2 step tasks the reference environment achieves 49.8% grounded completion against 25.7%, 19.4%, and 5.9% for the three generated variants; on 3–4 step tasks it achieves 50.5% against 26.2%, 16.5%, and 5.8%. The stable performance across bands indicates that relative fidelity remains consistent across varying task complexities. Beyond the primary 1–4 step evaluation core, extended trajectories requiring five or more oracle steps capture a complementary high-complexity boundary, serving as an exploratory benchmark for composite multi-step workflows.

Skill documents improve execution efficiency. Paired evaluation holds the tasks, solver, and budget fixed to isolate the operational contribution of environment skill documents. Across 387 evaluated tasks, providing domain skill documentation consistently supports downstream execution, driving reliable gains in tool invocation success and substantially lowering budget exhaustion rates across generators (Table 4). A granular analysis across roughly 23,700 calls demonstrates that skills primarily enhance execution efficiency by guiding the solver toward valid tool invocations and reducing trial-and-error exploration within available environments.

Evaluation cohort. Downstream evaluation is conducted on environments that satisfy standard executable criteria, comprising 36 complete environments and their corresponding task attempts. Unrunnable environments are captured separately as part of upstream environment construction failure rates to maintain a clean distinction between generation validity and downstream task execution.

A.4 Evaluation Settings

Evaluation Settings: GEN Difficulty Levels and SWE Repair

Purpose: the two CodeWorldBench settings. GEN has three nested difficulty levels; SWE is a full-surface repair setting.

GEN L1. *Public material:* a task file and a workspace with initial data and tool documentation, but no implementation. *Evaluation target:* a small coherent tool subsystem, with hidden output and state-transition checks.
GEN L2. *Public material:* a task file and a workspace with initial data and tool documentation, but no implementation. *Evaluation target:* a medium coherent tool subsystem that contains all L1 tools, checked by hidden rule cases.
GEN L3. *Public material:* a task file and a workspace with initial data and tool documentation, but no implementation. *Evaluation target:* the full original tool surface, evaluated through direct runtime calls and persistent-state assertions.
SWE (Full). *Public material:* one concise repair task file and a runnable but semantically broken starter repository. *Evaluation target:* full-surface semantic repair, with F2P checks for generation defects and P2P checks for behavior preservation.

A.5 Environment Taxonomy

Environment Taxonomy: The 50 Common Environments

Purpose: the held-out common pool by category and scenario. Environment IDs are omitted.

Communication Collaboration (7). *Communication:* Email, Instant Messaging, Message Bus. *Social Media and Marketing:* Social Media Operations, Advertising Management. *Collaboration and Productivity:* Knowledge Base, Slides Generation.
Media Entertainment (4). *Media and Content:* Video Platform, Public Relations. *Event and Entertainment:* Ticketing Management, Game Operations.
Data Intelligence (7). *Data Processing and Analysis:* Relational Database, NoSQL Database, Environmental Monitoring. *Data Storage and Processing:* Document Processing. *Academic Research:* Paper Management. *System and Business Services:* Business Registration, Hospital Directory.
Infrastructure Development (8). *Cloud and Infrastructure:* Web Search, Cloud Resource Management. *System Management and Operations:* Log Monitoring, Configuration Center. *System Tools and Simulation:* Desktop Simulation, IoT Simulation. *Software Engineering:* Project Management, API Documentation.
Financial Compliance (6). *Finance:* E-commerce, Banking Operations. *Legal and Compliance:* Contract Management, Intellectual Property. *Blockchain and Web3:* Cryptocurrency Management, NFT Marketplace.
Commerce Logistics (4). *E-commerce and Logistics:* Logistics Delivery, Supply Chain. *Real Applications:* Google Sheets ($\times 2$).
Public Service (4). *Public Services and Administration:* Vehicle Administration, Ecological Protection. *Customer Service:* Airline Customer Service, Healthcare Support.
Geospatial Construction (4). *Geography and Travel:* Shared Mobility, Traffic Monitoring. *Real Estate and Construction:* Lease Management, Construction Project.
Manufacturing Agriculture (4). *Manufacturing:* Manufacturing Execution, Industrial Control. *Agriculture and Environment:* Smart Agriculture, Veterinary Practice.
Embodied Intelligence (2). *Embodied Intelligence:* Embodied Intelligence ($\times 2$).

Environment Taxonomy: The 68 Professional Environments

The professional pool comprises 68 environments organized into ten software-domain categories.
Engineering Systems. AutoCAD, Siemens NX, Abaqus, TIA Portal, CATIA V6, SimScale, 3DEXPERIENCE Platform, Autodesk CFD, SOLIDWORKS, Studio 5000 Logix Designer, FactoryTalk View SE, Archicad.
Embedded Systems. dSPACE ConfigurationDesk, Altium Designer, Synopsys IC Compiler II, Ansys SCADE Suite, dSPACE ControlDesk, Vector CANoe, Cadence Virtuoso.
Scientific Computing. TopSpin, Xcalibur, CLC Genomics Workbench, CP2K + AiiDAlab, MassHunter, BIOVIA Materials Studio, Gaussian + GaussView.
Creative Media. Adobe Premiere Pro, Google Ad Manager (publisher side), Adobe Experience Cloud, Foundry Nuke, Innovid, The Trade Desk.
Enterprise Operations. Oracle E-Business Suite, Workday HCM, IFS Cloud, Cornerstone OnDemand, Oracle JD Edwards EnterpriseOne, Oracle Fusion Cloud HCM, SAP SuccessFactors Employee Central.

Financial Services. FactSet Workstation, Duck Creek Policy, Bloomberg Terminal, Encompass, BlackRock Aladdin, Murex MX.3, Guidewire ClaimCenter, Advent Geneva, Charles River IMS, Guidewire PolicyCenter.
Commercial Platforms. RealPage OneSite, Manhattan Associates Active Omni, Stellar MLS, AppFolio Property Manager, Blue Yonder Luminate Commerce, Datasite Diligence.
Supply Operations. Blue Yonder Warehouse Management, BestRx, Manhattan Active Warehouse Management, CargoWise One, Blue Yonder Transportation Management, PioneerRx.
Legal Governance. Aderant Elite 3E, NetDocuments, ProLaw, Tyler EnerGov.
Health Education. Centricity PACS, IntelliSpace PACS, Oracle Clinical One Data Collection.

A.6 L3 Task Structure

Public Artifact: Structure of an L3 `task.md`

Purpose: the public structure of the L3 task file, accompanied by initial data and tool documentation. Environment-specific content is replaced by placeholders.

Front matter. `env_id`, `category_id`, `category_name`, `scenario_name`, `name`, `tool_count`, `setting`: `L3`.
Task: Build The L3 Code World.
Agent Input Package. This `task.md` plus the L3 workspace containing initial data and per-tool reference contracts.
Setting Position. L3 requires the full original tool surface.
Scaffold Contract. Fill the provided workspace; implement only the declared L3 tool surface; keep the runtime importable through `mcp_servers.server.call_tool`; deliver `mcp_servers/` with `events/`, `state/`, `tools/`; keep state modules under `mcp_servers/state/`; one retained tool per `mcp_servers/tools/<tool>.py`; expose `init_database("data")` and `get_state_manager()`; require `snapshot()` to return `{core_state, database, counters}`; keep `call_tool` callable with a state manager and keyword arguments; return `content.status`, `content.data`, `content.message`; preserve exact error identifiers.
Included Query Tools. `<query_tool_1>`, `<query_tool_2>`, ...
Included Stateful Tools. `<stateful_tool_1>`, `<stateful_tool_2>`, ...
Public Repo Note. The workspace contains initial data and tool documentation; the agent supplies the executable implementation.
Entity Alias Mapping. `<singular/plural table-name aliases>`.
Code World Overview. `<domain overview and workflow summary>`.
Data Dimensions. Per entity: primary key format, fields, required values, enums, foreign keys, dependencies, reverse references.
Data Storage. `<database files, table mapping, immutable seed-data policy>`.
Data Scale Requirements. `<per-table scale and distribution requirements>`.
Sample Data. `<representative records>`.
Core State. `<derived state fields and invariants>`.
State Schema. `<snapshot schema and state assertion shape>`.
Event Module Contract. `<event triggers, payloads, ordering constraints>`.
Canonical ID Policy. `<semantic ID policy and rejected legacy ID forms>`.
Tool Classification. Query and stateful grouping for the retained L3 surface.
Tool List (up to 60 tools). Per tool: name, description, parameters, business logic, state effects, return envelope, errors.
Data Construction Guidance. Per-entity plan, field distributions, foreign-key rules, text richness constraints, visible naming realism, local opaque keys, rejection-only legacy ID policy.

A.7 SWE Task Structure

Public Artifact: Structure of an SWE `task.md`

Purpose: the complete public structure of the repair prompt. The SWE agent additionally receives the runnable but defective workspace and the same per-tool reference contracts, and no hidden test asset.

Task. Repair the existing agent environment for `<environment name>`, followed by a domain summary.
Repository Shape. The `task/` and `workspace/` sibling layout, listing the seed tables under `workspace/data/database/` and the per-tool modules under `workspace/servers/tools/`. `task/` is read-only prompt material; `workspace/` is the repository to repair.

Repair Target. Edits are confined to `workspace/servers/`; seed data, reference material, and test expectations may not be changed. Every affected tool must dispatch through `call_tool` under its documented public name, return the documented error identifier in the documented order for each rejection path, return the documented `data` fields without leaking internal records, and persist every documented state change together with the derived state that depends on it.
Repair Strategy. A seven-step order that puts wide-blast-radius defects first: entry point and dispatch, then imports and module loading, then data-path and table-name resolution, then validation order and error identifiers, then query semantics, then success payloads and durable writes, and tool-specific business rules last.
Failure Categories. The affected tools grouped by defect kind rather than by file: (A) validation, permissions, and error identifiers; (B) query semantics and filtering; (C) response data and durable state. Each group states its symptom, its repair focus, and its affected public tools. Tools appear in several groups when they carry several defects.
Completion Criteria. Every listed tool loads through `call_tool` and preserves the response envelope; each follows its reference for parameters, validation order, error identifiers, payload fields, and state effects; failed calls leave state unchanged; shared-helper edits keep other tools working.
How To Verify. A `call_tool` snippet for exercising one valid call and one documented error case, with the instruction to inspect the runtime table afterwards and confirm that an invalid call left it unchanged.
Reference Map. One row per affected public tool: public name, implementation file, reference specification file, and failure groups.
Tools, Runtime Contract, Response Shape, Implementation Notes. As in the L3 artifact, restated for a repair starting point.

Public Artifact: Structure of a Per-Tool Reference Contract

Purpose: the per-tool specification both settings expose under `task/references/tools/`. This is the unit an agent must reproduce exactly, and its error identifiers are compared as strings.

Description. One line naming the tool's effect on the world.
Parameters. Per parameter: name, type, required or optional, default, and the entity its value identifies.
Business Logic. The ordered rejection path, each entry pairing an error identifier with the exact predicate that triggers it; then derived quantities, the iteration set for reads and mutations, records copied before mutation, and the declared state effects per entity.
Returns. The exact top-level keys of the success `data` payload.
Errors. Every error identifier with its human-readable explanation.

A.8 Generation and Conversion Skill Frameworks

Generation Skill: `wm-env-gen`

Purpose: turn one catalog entry into a self-contained world specification that can later drive code generation without additional private context.

Input prompt frame. Generate only one environment prompt bundle; read the environment ID, catalog metadata, working root, and target directory; keep all generated text, schemas, examples, and user-facing content in English. Resume partial work rather than replacing it, do not modify any other environment, produce the complete bundle, and stop only once the final audit marker is valid.
Workflow. Detect progress; bootstrap from catalog metadata; write the skeleton; define entities, storage, scale, examples, core state, and state schema; specify the tool surface with arguments, business logic, state effects, return shapes, and error identifiers; write data construction guidance; generate deterministic seed data; verify prompt quality, data quality, foreign keys, record counts, English-only text, directory hygiene, and legacy-ID rejection.
Reference structure. Environment overview, data dimensions, data storage, scale requirements, sample data, core state, state schema, runtime and verification contract, checks bundle contract, test asset contract, world model deliverables, event module contract, documentation deliverables, canonical ID policy, tool classification, tool list, data construction guidance, data sample validation.
Outputs. `<ENV_ID>_P.md`, `tool_classification.json`, `generate_data.py`, `data/database/*`, `.wm_env_gen/final_audit.json`. The audit records the completion checks, making the prompt phase a gated artifact rather than an informal draft.

Generation Skill: `wm-codegen`

Purpose: convert a completed prompt bundle into a runnable MCP source repository, the executable reference world from which settings, hidden tests, and repair variants are derived.

Input prompt frame. Generate code for only one environment; read the environment ID, catalog metadata, prompt root and directory, code working root, and target code directory; consume the prompt bundle from the previous phase. English code comments, docs, tests, and user-facing text are required; reserved demo domains, placeholders, and copied cross-domain examples are rejected; caches and runtime residue are removed; partial work resumes; the run stops only after a valid final audit.
Workflow. Bootstrap from baseline runtime files; implement the state layer, database loading, canonical ID helpers, and derived core-state computation; implement each tool with its JSON case bundle, executable tests, rule documentation, and implementation notes; register tools in the server; add global docs, event profiles, verification helpers, and benchmark asset validators; run the full test and validation loop; repair inconsistencies until all required checks pass; write the final self-audit.
Runtime contract. Preserve `init_database("data")`, `get_state_manager()`, and `call_tool(...)` as stable evaluator entrypoints. Seed data stays immutable, runtime state is isolated before mutation, tools return the standard `content.status`, `content.data`, `content.message` envelope, and derived state is computed from the runtime database rather than hardcoded.
Outputs. Prompt copy, README and context files, requirements, `tool_classification.json`, `data/database/*`, `mcp_servers/`, `tests/`, `world_model/`, `checks/`, verification scripts, `.codegen_v2/final_audit.json`.

Conversion Skill: `source-env-to-bench-repo-difficulty-converter`

Purpose: convert a source CodeWorld environment into a benchmark-ready repository with GEN difficulty levels and the SWE repair setting.

Read first. Source layout, full environment contract, runtime entrypoints, state reset behavior, tool registry, source tool tests, reusable state assertions.
Preserve. Source domain, entities, runtime style, tool semantics, full source tool surface, hidden rule-test source of truth, canonical full prompt, and generator-based database construction where present.
Add. Setting-level public task files, public code-world repositories, hidden evaluator assets, difficulty metadata, SWE bug profiles.
Avoid. Exposing hidden test logic; keeping authoring-only helper context in public artifacts; publishing per-tool validation-order blocks in public task files; changing exact error identifiers.
Difficulty scoring. Score every tool from 0 to 20 on interface complexity, implementation complexity, state and domain coupling, behavioral complexity, and test complexity. Sort by score, strongest individual signal, then source order. Use the tiers as evidence, then select coherent nested L1 and L2 subsystems with model assistance; L3 remains the full tool surface.
Build GEN. L1, L2, and L3 each expose a task file and a workspace with initial data and tool documentation, preserving the world definition, runtime layout, state snapshot contract, response envelope, alias mapping, and retained tool contracts.
Build SWE. A concise symptom-driven repair task plus a runnable full-surface starter repository with generation defects, preserving the runtime shape, seed database, tool registry, and direct call-tool evaluation path while recording F2P and P2P metadata.
Migrate tests. Copy retained source rule-test bundles without truncation. Keep only cases for tools the setting retains. Store hidden cases and metadata under evaluator assets, never in public starter folders.
Validate. Setting invariants, runtime entrypoints, state snapshot compatibility, direct evaluation compatibility, exact error identifiers, hidden-test non-leakage, SWE defect profiles, public task-file completeness.

A.9 Per-Model Trace Measurements

Table 5 gives the per-model numbers behind Section 5.3 for the four models whose harness reports billed cost, so the ranges quoted in the prose can be read against their endpoints. Every row is 50 runs on the 50 common environments.

Two properties of the table are worth stating explicitly. First, the cost column is not a restatement of the operations column: Kimi K3 issues 27% more GEN operations than Claude Opus 5 for less than half the money, because per-operation price spans a factor of three across these four models. Second, test runner invocations remain infrequent across models, and probe script executions decrease from GEN to SWE across configurations. This indicates substantial potential for incorporating structured self-verification mechanisms directly into execution workflows.

Table 5: Per-model trace measurements, 50 runs per cell. *Ops* is mean operations per run and *Cost* mean billed USD per run, so *$/op* is their ratio. Role shares are over all operations, *Err* over harness-returned outcomes. *Probe* and *Runner* count runs that execute a self-written check script and a real test runner. *Streak* is the mean longest consecutive-error run. Cost is available only under Claude Code; GPT-6 Astra and Qwen3.8-27B are therefore absent here and sourced from operation counts alone in the main text.

Model	Setting	Ops	Cost	$/op	Exec	Read	Edit	Err	Streak	Probe	Runner
Claude Opus 5	GEN L3	66.8	11.90	0.178	89.2	4.9	5.9	2.6	1.0	35	0
	SWE	95.8	18.77	0.196	85.2	5.3	9.4	3.0	1.2	16	3
Kimi K3	GEN L3	84.7	5.64	0.067	60.2	36.5	3.3	5.3	1.9	32	0
	SWE	129.2	8.20	0.063	54.8	30.7	14.5	3.9	1.8	11	2
DeepSeek-v4-pro	GEN L3	105.0	9.30	0.089	77.5	18.9	3.5	7.3	2.2	20	0
	SWE	130.2	13.62	0.105	78.9	13.8	7.3	8.3	2.5	17	3
GLM-5.3	GEN L3	117.8	12.60	0.107	75.9	17.3	6.8	9.2	2.7	38	0
	SWE	146.9	13.55	0.092	62.8	26.5	10.7	5.7	2.0	13	0

A.10 Implications for Construction

Three concrete engineering directions follow directly from trace and execution analyses.

Supervise state rather than output. Stale counters, missing state values, and incorrect state assertions are invisible in a returned payload, so a training signal derived from return values cannot tell a correct update from a plausible one. Trajectories should be scored against the post-call snapshots exposed by reference implementations we plan to release. The static profile in Appendix A.1 quantifies what such a signal would cover: state assertions carry 2.77 and 8.11 paths per case on the two pools, and 2.2% to 3.0% of them require a field to be *absent*, a condition no return-value check can express.

Incorporate automated verification into generation workflows. Agents commit significant execution budgets to iterative file manipulation. Scaffolds that systematically exercise tool interfaces and verify persistent state can channel this operational effort into early verification. The trace metrics above show clear headroom for test-runner integration across models, highlighting systematic testing as a promising lever for enhancing environment robustness.

Treat shared invariants as an artifact rather than re-deriving them per tool. Systematic environment fidelity benefits strongly from consistent stateful coupling. Agents benefit from committing once to identifier policies, validation order, and derived-view update rules, subsequently implementing each tool against that shared definition. The contract density measured above underscores this design: a common environment documents 69.9 distinct error identifiers across 32.9 tools, making shared abstractions effective for ensuring uniform business semantics.

A.11 Inference Configuration

For GEN and SWE, we configure a 1M-token context window and a 64K-token output limit, except for Qwen models, which use 256K and 32K, respectively. Models whose APIs support these sampling parameters use temperature 1.0 and top-p 0.95. OpenAI models run under Codex v0.154.0; Claude Code runs use v2.1.251. Table 6 lists the configured reasoning controls for the evaluated models and the models used to construct benchmark artifacts.

Table 6: Reasoning settings used in the experiments. Named effort levels are reported as configured; thinking and adaptive denote enabled thinking modes.

Model	**Reasoning setting**
Main benchmark models	
GPT-5.6 Sol	`xhigh`
GPT-5.6 Luna	`xhigh`
GPT-5.6 Terra	`xhigh`
GPT-6 Astra	`xhigh`
Claude Opus 5	`xhigh`
Doubao-seed-evolving	`xhigh`
Qwen3.8-27B	`xhigh`
Qwen3.6-35B-A3B	Thinking enabled
MiniMax-M3	`adaptive`
DeepSeek-v4-pro	`max`
DeepSeek-v4.1-flash	`max`
GLM-5.3-Flash	`max`
GLM-5.3	`max`
Kimi K3	`max`
Additional construction models	
GPT-5.5 (reference environments)	`xhigh`
Qwen3.6-27B (generated repositories)	Thinking enabled
GLM-5.2 (downstream environment generator)	`max`

The downstream solver uses Qwen3.8-27B at temperature 0 with at most 30 tool calls per task, as described in Appendix A.3.