TwinEnv-Bench: Beyond Greenhouse Sandboxes via Counterfactual Evaluation of Reactive and Proactive Environment Adaptation in Software Agents

TraceLite AI[1,2,3,4]

[1]Institute of Information Engineering, CAS, [2]Institute of Automation, CAS

[3]Zhongguancun Academy, [4]Tsinghua University

Abstract

Prevailing benchmarks for software and terminal agents evaluate models almost exclusively inside uniform "greenhouse" containers configured with permissive system defaults, confounding algorithmic coding skill with real-world environment adaptability. We introduce TwinEnv-Bench, a controlled micro-scale benchmark of 117 scripting, systems, and data-engineering tasks that pairs identical user instructions and codebases across a standard baseline container and a target sandbox shifted along a single verified system property. Spanning a five-layer operating-system taxonomy enforced by mechanical solvability contracts and leak-proof host-isolated grading, TwinEnv-Bench contrasts *interactive-exposed* shifts, where discrepancies trigger immediate errors during sample runs, against *deployment-latent* shifts, where sample runs pass silently while production constraints reside in scheduled jobs, mount boundaries, process privileges, or resource quotas. Evaluating nine frontier models across non-refusal executed runs reveals that near-ceiling baseline competence collapses to 26.8%–67.1% single-run solve rates and 14.5%–59.5% multi-run reliability under single-variable shifts. Models that lead exposed shifts via runtime debugging and upfront shallow checking drop sharply and invert rankings under the silent constraints and production-layer shifts of the latent regime, where upfront multi-step reconnaissance is associated with a nearly sixfold higher solve rate than unprobed runs. Controlled system-prompt interventions further confirm that explicit pre-flight reconnaissance causally lifts latent solve rates by nearly 30 percentage points and doubles multi-run reliability, yet a 43.6 percentage-point cross-model *probe-to-action* gap persists across the four tested models even under near-saturated probing—indicating that checklist guidance resolves reconnaissance blindness without equalizing underlying system-level reasoning from observed host state to boundary-safe code.

Repository: https://github.com/TraceLite-AI/TwinEnv-Bench
Website: https://tracelite.ai
Correspondence: contact@tracelite.ai
Hugging Face: https://huggingface.co/spaces/TraceLite-AI/TwinEnv-Bench

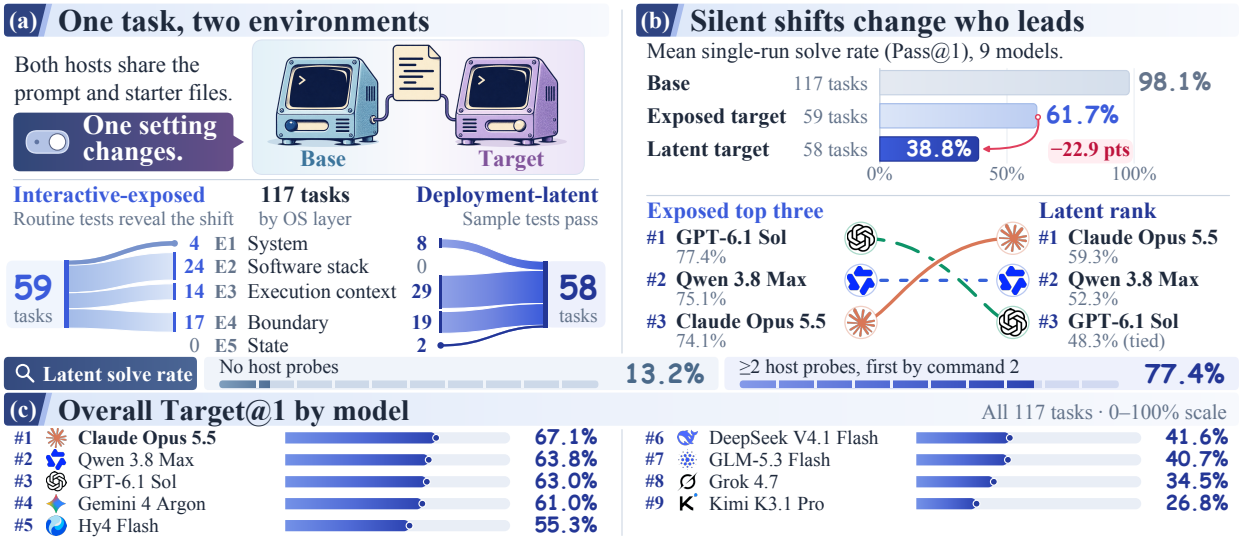


Figure 1: TwinEnv-Bench Overview.

Contents

1 Introduction **4**

1.1 The Greenhouse Illusion in Agent Evaluation 4

1.2 Counterfactual Twin Environments and Shift Exposure Regimes 4

1.3 Summary of Empirical and Causal Findings 5

1.4 Contributions 6

2 Problem Formulation, Metrics, and Benchmark Positioning **7**

2.1 Formalizing Counterfactual Environment Adaptation 7

2.2 Evaluation Metrics and Non-Refusal Scoring Protocol 8

2.3 Positioning Relative to Existing Agent Benchmarks 9

3 The E1–E5 Environment Taxonomy and Counterfactual Suite **9**

3.1 Incident-Grounded Taxonomy Construction 9

3.2 Architectural Walkthrough of Layers E1–E5 10

3.3 Dual Mechanical Verification Contracts 12

3.4 Host Discoverability and Leak-Proof Isolated Grading 12

4 Main Experimental Results **13**

4.1 Experimental Setup and Evaluated Models 14

4.2 RQ1: Collapse of the Greenhouse Illusion and Empirical Monotonicity 14

4.3 RQ2: Exposed vs. Latent Regime Shifts and Ranking Inversions 16

4.4 Cross-Layer Specialization, Bottlenecks, and Oracle Union 17

4.5 Adaptation Overhead and Cost–Reliability Pareto Frontier 19

5 Trajectory-Level Behavioral and Causal Diagnostics **21**

5.1 RQ3: Probe Depth, Upfront Timing, and Premature Blind Exit 21

5.2 Small-Budget Resampling Limits and Probing Stability 22

5.3 When Probing Is Not Enough: Probe Quality and the *probe-to-action* Gap 23

5.4 RQ4: Causal Reconnaissance Prompting and the *probe-to-action* Gap 24

6 Related Work **26**

6.1 Repository-Scale Coding Benchmarks and Agent RL Environments 26

6.2 Operating-System Sandboxes, Environment Drift, and Evaluation Hygiene 26

6.3 Partial Observability, Active Perception, and Inference-Time Reliability 27

7 Discussion, Limitations, and Conclusion **27**

7.1 Implications for Agent Post-Training and Harness Design 27

7.2 Limitations and Future Directions . . . 28
7.3 Conclusion . . . 28

A Complete E1–E5 Taxonomy Registry and Incident Grounding **33**

B Verified Sandbox Suite, Runner Security, and Model Configurations **35**
B.1 Probe-Verified Sandbox Configurations . . . 35
B.2 Four-Profile Execution Dispatcher and Zero-Leakage Audit . . . 36
B.3 Evaluated Model Snapshots and Execution Accounting . . . 36

C Fine-Grained Dimension-Group Capabilities and Unsolved Tasks **38**

D Probe Attribution Protocol and Counterfactual Trajectory Case Studies **40**
D.1 Probe Matching and Failure-Mode Attribution Protocol . . . 40
D.2 Qualitative Paired Trajectory Case Boxes . . . 40

E Causal Prompt Intervention Verbatim Prompts and Sensitivity **42**
E.1 Verbatim System Prompt Suffixes . . . 42
E.2 Pairing Accounting and Sensitivity Analyses . . . 42

F Complete 117-Task Catalog, Execution Guide, and Datasheet **43**
F.1 Complete 117-Task Counterfactual Catalog . . . 43
F.2 Quick-Start Verification Commands and Datasheet . . . 44

1 Introduction

Autonomous software engineering and terminal agents powered by large language models have progressed rapidly from single-function code synthesis to multi-step repository maintenance, interactive data engineering, and command-line administration [1–4]. On widely adopted benchmarks such as SWE-bench Verified [5] and interactive coding suites [6, 7], frontier models now resolve more than two-thirds of evaluation instances by iteratively inspecting compiler diagnostics, unit-test tracebacks, and terminal outputs. These gains have fueled growing adoption of autonomous coding agents across enterprise continuous-integration pipelines, data-processing workflows, and cloud operations.

Yet a striking gap separates benchmark leaderboards from production deployments. When high-scoring agents leave standardized evaluation harnesses and operate inside real organizational infrastructure, such as hardened enterprise build servers, air-gapped containers, unattended nightly batch schedules, or non-English regional hosts, practitioners encounter frequent, high-impact failures triggered not by algorithmic complexity, but by ordinary differences in the underlying operating environment. Scripts that pass interactive smoke tests corrupt customer records under regional character encodings or daylight-saving transitions, crash midway through production feeds under strict file-size or descriptor ceilings, fail silently under stripped background-scheduler paths, or break permission invariants across shared service accounts. Understanding why capable coding models remain brittle under routine environmental variation, and how to measure and remedy that brittleness, has become a central bottleneck for reliable agent deployment.

1.1 The Greenhouse Illusion in Agent Evaluation

We trace this discrepancy between benchmark optimism and operational fragility to a structural blind spot in prevailing evaluation methodology, which we call the *Greenhouse Illusion*. Virtually all mainstream code and terminal benchmarks instantiate tasks inside uniform, idealized "greenhouse" containers configured with convenient system defaults: standard English text encodings, unrestricted administrator privileges, familiar command-line utilities, open external network access, generous memory and file-size ceilings, uniform single-disk local storage, and directly inherited interactive shell settings [1, 8–10]. Because both post-training reinforcement-learning rollouts and benchmark leaderboards operate inside this same homogeneous container mold, models are rarely tested on whether they actually adapt to the machine in front of them.

Evaluating agents exclusively within uniform greenhouse sandboxes introduces three fundamental methodological confounds. First, it produces confounded competence attribution: when an agent passes a test suite in a greenhouse container, its score entangles genuine algorithmic problem-solving with silent reliance on convenient default settings—such as assuming a fixed text encoding, single-volume file movement, or unrestricted write privileges. Conversely, when an agent fails in a broader system benchmark [4, 11], evaluating each task in only a single static environment makes it impossible to tell whether the failure reflects hard task logic or an unfamiliar host configuration. Second, existing suites lack systematic environment coverage and verified single-variable control: real-world operational hazards span multiple layers of the computing stack, and modifying container settings without automated verification easily introduces unintended side effects or unsolvable tasks. Third, prevailing benchmarks disproportionately reward error-driven reactive patching: because environment mismatches typically surface right away as stack traces during interactive test runs, agents can succeed through a reactive write–crash–patch loop [12, 13]. In real software operations, by contrast, critical constraints frequently remain silent on small interactive sample inputs and take effect only under scheduled background execution, restricted service identities, or production-scale workloads—requiring agents to inspect their host environment proactively before finalizing code.

1.2 Counterfactual Twin Environments and Shift Exposure Regimes

To separate algorithmic coding competence from environment adaptability with causal rigor [14], we introduce TwinEnv-Bench, illustrated in Figure 1a, a counterfactual benchmark comprising 117 real-world software and data engineering tasks evaluated across 234 paired task–environment cells. For every task, TwinEnv-Bench keeps the natural-language user instruction, initial codebase, and functional grading criteria completely identical while instantiating two controlled environments: a standard baseline container with conventional greenhouse defaults, and a counterfactual target sandbox shifted along a single verified system property. Comparing an agent's outcomes across these two twin cells isolates the exact performance impact of a single environmental change on a task whose underlying logic the model can demonstrably solve.



Figure 2: **Counterfactual Tasks Across Interactive-Exposed and Deployment-Latent Regimes.** Prompt excerpts and sample rows accompany execution paths on two TwinEnv-Bench tasks. In interactive-exposed Panel a, installing the Excel reader works system-wide on the writable Base system yet fails on the read-only Target system, where the error prompts workspace bundling so Excel and CSV both import. In deployment-latent Panel b, the Target API serves the 45-shipment sample with HTTP 200 on pages 1 and 2, while page 3 returns two transient HTTP 503 errors before 200: without host probes, held-out exports stop on the first 503, whereas probing and retrying exports all 700 rows.

Every task in TwinEnv-Bench is governed by dual mechanical verification contracts that guarantee causal purity and full solvability. At the environment layer, 55 controlled sandbox configurations undergo automated pre-flight probe audits to ensure that each target sandbox alters only the intended system property without collateral drift. At the task layer, every instance passes a deterministic six-run, three-state solvability check across the paired environments: the unmodified starting codebase fails in both environments, ruling out trivial pre-solved setups; a conventional implementation relying on standard default assumptions passes in the baseline container yet fails in the shifted target sandbox, proving that environment adaptation is necessary; and an environment-aware reference implementation passes in both environments, confirming full solvability under natural, zero-spoiler user instructions. Furthermore, a leak-proof host-isolated evaluation runner keeps model credentials, reference solutions, and held-out test suites strictly outside the agent container, injecting grading payloads only after agent termination.

Built atop a five-layer operating-system taxonomy covering 15 dimension groups and 31 specific mechanisms distilled from real-world open-source agent failure reports, TwinEnv-Bench partitions its 117 tasks nearly evenly along a second orthogonal axis of shift observability, illustrated by two concrete benchmark instances in Figure 2. In the 59 *interactive-exposed* tasks exemplified by Figure 2a, the shifted environment property produces immediate error messages, warnings, or visibly malformed outputs during ordinary interactive commands or runs on small sample data, directly measuring reactive debugging and recovery. In the 58 *deployment-latent* tasks exemplified by Figure 2b, interactive runs on sample inputs succeed without error, while the shifted constraint takes effect under scheduled background execution, cross-account permissions, filesystem mount boundaries, production resource quotas, or full-scale batch traffic. Crucially, every latent task adheres to a strict *silent in sample, discoverable on host* fairness principle: no constraint is hidden in an external black box, and every active host condition can be discovered prior to submission through standard read-only system inspection commands.

1.3 Summary of Empirical and Causal Findings

We evaluate nine frontier language models across three independent trials per cell, spanning 6,318 completed trials across all 234 paired baseline and target cells. Combining multi-run leaderboard evaluation, cost telemetry, command-level trajectory parsing, and controlled prompt interventions, as previewed in Figure 1b and Figure 1c, yields four principal findings.

Collapse of the Greenhouse Illusion. Across baseline containers, all nine frontier models achieve near-ceiling single-run solve rates between 95.1% and 99.7%, confirming that prompt comprehension, domain logic, and routine command-line usage pose virtually no barrier. When a single verified system property shifts in the target sandbox, however, single-run solve rates across non-refusal executed runs fall sharply to 26.8%–67.1%, while strict three-of-three multi-run reliability drops to 14.5%–59.5%. Task-level pairing shows that 96.4% of target failures occur on tasks that the exact same model solves reliably in the baseline container, while tasks unsolved in the baseline container almost never succeed by chance in the shifted sandbox.

Exposure Regime Drops, Ranking Inversions, and Orthogonal Layer Strengths. Moving from interactive-exposed tasks to deployment-latent tasks, which combines silent sample-time observability with its associated execution-context, quota, and service-state layers, substantially reduces solve rates across every evaluated model and overturns leaderboard rankings. GPT-6.1 Sol leads interactive-exposed tasks at 77.4% through strong runtime error recovery and upfront shallow checking, yet drops to a tie for third at 48.3% when multi-hop constraints remain silent on sample data. Conversely, conditional on non-refusal executed runs, Claude Opus 5.5 pairs upfront and multi-step probing to lead deployment-latent tasks at 59.3% and rank first overall at 67.1%, while zero-refusal Qwen 3.8 Max reaches 52.3% on latent shifts and 63.8% overall—claiming first place on both leaderboards when unexecuted safety refusals are counted as failures, where Claude Opus 5.5 moves to 49.4% on latent tasks and 58.7% overall, with both deep-probing models still surpassing GPT-6.1 Sol on latent shifts. On the 19-task compound host-isolated pressure subset, which pairs silent privilege and quota boundaries with a tight 6,000-token per-turn output cap, deep-probing models likewise hold the top ranks while shallow-probing models drop sharply. Across the major E1–E4 layers, model families exhibit strong complementarity, with Gemini 4 Argon leading filesystem semantics, GPT-6.1 Sol leading software-stack and security boundaries, and Claude Opus 5.5 leading execution context on non-refusal runs, such that selecting the best-performing model on each task solves 88.0% of the benchmark.

Premature Blind Exit, Small-Budget Retry Limits, and Probing Mechanics. Command-level trajectory stratification shows that unprobed deployment-latent runs pass only 13.2% of the time, whereas trajectories executing two or more environment inspection commands within the first two steps reach a 77.4% solve rate—nearly six times the unprobed rate. Notably, failed latent runs execute fewer shell commands on average than baseline runs, diagnosing *premature blind exit*: upon observing a clean run on small sample inputs, agents terminate early without suspecting latent production constraints, which also explains why token consumption inflates far less on latent shifts than on error-surfacing exposed shifts. Because unprobed default assumptions repeat across independent trials, small-budget $K = 3$ blind resampling on latent tasks yields limited gains and still falls just short of the worst-case three-of-three floor of exposed tasks. Dissecting runs that probe consistently yet still fail further demonstrates that raw probing frequency is insufficient; upfront timing, multi-step depth, and accurate deduction from host state to code edge cases jointly govern reliability.

Causal Prompt Intervention and the Cross-Model *probe-to-action* Gap. To test whether proactive reconnaissance is a direct causal lever rather than a correlation with general coding strength, we perform a task-agnostic, three-level system-prompt intervention, progressing from the default prompt to a generic engineering reminder and a taxonomy-aligned five-part pre-flight checklist, across all deployment-latent tasks on four representative models. Explicit reconnaissance guidance causally lifts paired single-run solve rates by 29.9 percentage points from 32.0% to 61.9%, more than doubles three-run reliability, and propels DeepSeek V4.1 Flash and GPT-6.1 Sol well past every unprompted model. Yet even when structured prompting drives multi-step host inspection close to saturation across all four tested models, a 43.6 percentage-point cross-model *probe-to-action* conversion gap remains between the highest- and lowest-scoring models, indicating that checklist guidance removes reconnaissance blindness without smoothing out underlying differences in system-level reasoning from host observations to boundary-safe code.

1.4 Contributions

This work makes five primary contributions to the evaluation and understanding of autonomous software agents:

- **Counterfactual Twin-Environment Paradigm and Observability Formulation:** In Section 2, we formulate interactive environment adaptation as a partially observable sequential decision process and introduce a paired twin-cell evaluation design that holds task prompts, codebases, and grading criteria invariant across baseline and single-

variable shifted sandboxes. By crossing counterfactual isolation with a principled split between error-surfacing interactive-exposed shifts and sample-silent deployment-latent shifts, our framework disentangles baseline task coding from exposed error recovery and silent host adaptation.

- **Incident-Grounded Five-Layer System Taxonomy and Verified Sandbox Suite:** In Section 3, we distill over two hundred real-world failure reports across open-source coding agents together with operating-system specifications into a structured five-layer, 15-group, 31-mechanism taxonomy spanning storage semantics, software runtimes and toolchains, locale and scheduling contexts, security and quota boundaries, and service state. We instantiate this taxonomy in 55 reusable container configurations verified by automated probes to guarantee single-property isolation without collateral drift.
- **117-Task Controlled Micro-Scale Benchmark with Mechanical Contracts and Leak-Proof Grading:** Across Section 3.3 through Section 3.4, we construct 117 controlled micro-scale scripting, systems, and data-engineering tasks comprising 234 paired cells, each validated by a deterministic six-run, three-state solvability contract that rules out both trivial pre-solved workspaces and unsolvable black-box traps. We pair the task suite with a zero-dependency, host-isolated runner that enforces authentic account, permission, and resource limits while keeping credentials, reference patches, and held-out grading suites inaccessible during agent execution.
- **Comprehensive Multi-Model Evaluation of Reliability, Specializations, and Friction Costs:** In Section 4, we evaluate nine frontier models under both single-run solve rates and strict multi-run production reliability across paired environments. Our study contrasts near-ceiling baseline competence against severe single-variable shift drops, examines leaderboard inversions across exposed and latent regimes, maps cross-family complementarity across system layers, and characterizes the Cost–Reliability Pareto frontier alongside regime-asymmetric compute friction taxes.
- **Command-Level Behavioral Diagnostics, Causal Interventions, and Open Benchmark Release:** In Section 5, we build a syntax-aware shell trajectory analyzer across main and intervention traces to diagnose premature blind exit, test-time resampling limits, distinct model reconnaissance archetypes, and the failure modes of shallow or unconverted probing. Through a controlled three-level system-prompt intervention, we establish both the direct causal leverage of structured pre-flight reconnaissance and the persistent *probe-to-action* reasoning gap, and release all tasks, container build specifications, verification harnesses, and sanitized trajectories publicly to support reproducible research.

2 Problem Formulation, Metrics, and Benchmark Positioning

Before detailing the architectural layers of TwinEnv-Bench, we formalize single-variable counterfactual environment adaptation as a partially observable sequential decision process, define our multi-run evaluation and reliability metrics, and situate TwinEnv-Bench relative to prior benchmarks.

2.1 Formalizing Counterfactual Environment Adaptation

Interactive Execution Under Partial Environment Observability. An autonomous software or terminal agent interacts with a computing host to satisfy a natural-language task prompt p over an initial workspace W_0. Unlike standalone code generation [15, 16], multi-turn command execution unfolds inside a concrete host environment e governed by system properties spanning storage semantics, language runtimes, regional encodings and timezones, account privileges, mount boundaries, network routing, resource quotas, and background services. Crucially, this interaction is partially observable [17]: at the start of a session, the agent observes only the zero-spoiler task prompt p and initial workspace files, while the underlying host configuration remains unstated. Unless a host property triggers an immediate error during routine command execution, or the agent proactively runs read-only diagnostic commands to inspect the machine, non-default system constraints remain hidden until final verification.

Single-Variable Counterfactual Twin-Cell Design. In conventional single-environment benchmarks, each task is evaluated inside one fixed container e_i using a post-termination functional verifier V_i. When an agent fails verification, baseline algorithmic difficulty and sensitivity to the host environment remain entangled. TwinEnv-Bench resolves this identifiability problem by instantiating every task $i \in \{1, \ldots, N\}$ across $N = 117$ tasks as a paired counterfactual twin tuple:

$$\mathcal{T}_i = \left(p_i,\, W_{i,0},\, e_i^{\text{base}},\, e_i^{\text{target}},\, V_i\right), \quad \text{with} \quad \text{diff}\left(e_i^{\text{base}}, e_i^{\text{target}}\right) = \{\Delta d_i\}, \tag{1}$$

where the baseline container e_i^{base} keeps all system properties at standard greenhouse defaults, and the target sandbox e_i^{target} alters only a single verified environment dimension Δd_i while holding the task prompt p_i, initial workspace $W_{i,0}$, and functional verifier V_i identical. Letting $Y_i(e) \in \{0, 1\}$ denote the binary pass–fail outcome of running an agent inside environment e, the paired difference $Y_i(e_i^{\text{target}}) - Y_i(e_i^{\text{base}})$ under Equation 1 directly isolates the causal impact of environment shift Δd_i, unconfounded by prompt ambiguity or domain coding complexity.

Interactive-Exposed vs. Deployment-Latent Shift Regimes. Consider a competent baseline implementation that writes correct business logic for prompt p_i under standard default assumptions and validates its code solely on the workspace sample inputs in the login shell without actively inspecting the host. Across our $N = 117$ tasks, counterfactual shifts fall into two distinct observability regimes depending on how such an implementation behaves during interactive testing. In the *interactive-exposed* regime $\mathcal{T}_{\text{exp}}$ spanning 59 tasks, running ordinary build, test, or sample commands immediately surfaces a visible discrepancy between e_i^{target} and e_i^{base}, such as a stack trace, a command-flag error, or visibly malformed output, giving a reactive agent an explicit error signal from which to diagnose Δd_i and iteratively patch its code.

In contrast, in the *deployment-latent* regime $\mathcal{T}_{\text{lat}}$ spanning 58 tasks, interactive runs on sample inputs succeed cleanly with identical behavior across both environments, yet final verification fails in e_i^{target} when V_i executes the submission under its authentic deployment context—such as a scheduled background job, an unprivileged service account, a full-scale data stream, or a strict kernel resource limit. Crucially, every task in $\mathcal{T}_{\text{lat}}$ upholds *host discoverability*: the active constraint Δd_i is already present on the host from the start of the session and can be identified through standard read-only inspection commands, ensuring that an agent exercising proactive reconnaissance can resolve the shift prior to submission without guesswork.

2.2 Evaluation Metrics and Non-Refusal Scoring Protocol

For each evaluated model, task $i \in \{1, \ldots, N\}$, and environment cell $c \in \{\text{base}, \text{target}\}$, we run $K = 3$ independent stochastic trials. After excluding trials blocked prior to sandbox entry by an upstream API gateway safety refusal, let $R_{i,c} \leq 3$ denote the number of executed trials and $s_{i,c} \in \{0, \ldots, R_{i,c}\}$ denote how many of those executed trials pass verification.

Single-Run Solve Rate and Shift Drops. Over any task subset $\mathcal{S}$, the micro-averaged single-run solve rate $\text{Pass@1}_c(\mathcal{S})$ is computed as total passed runs $\sum_{i \in \mathcal{S}} s_{i,c}$ divided by total executed runs $\sum_{i \in \mathcal{S}} R_{i,c}$, expressed as a percentage. From single-run solve rates, we measure two primary performance gaps: the counterfactual shift drop $\Delta_{\text{shift}} = \text{Pass@1}_{\text{target}}(\mathcal{T}) - \text{Pass@1}_{\text{base}}(\mathcal{T})$ across paired environments, and the exposure regime drop $\Delta_{\text{regime}} = \text{Pass@1}_{\text{target}}(\mathcal{T}_{\text{lat}}) - \text{Pass@1}_{\text{target}}(\mathcal{T}_{\text{exp}})$ between silent and error-surfacing shifts.

Optimistic Retry Ceiling vs. Production Reliability. In production software operations, an agent that corrupts customer records or crashes a nightly batch job on one out of three runs cannot be safely deployed even if a retry occasionally succeeds. Following recent agent reliability conventions [18], over all tasks $\mathcal{S}^+$ with at least one executed trial, we report both the optimistic sampling ceiling Pass@3, requiring success in at least one executed repeat, and strict production reliability Pass^3, requiring consistent success across every executed repeat:

$$\text{Pass@3}_c(\mathcal{S}) = \frac{1}{|\mathcal{S}^+|} \sum_{i \in \mathcal{S}^+} \mathbb{I}\big[s_{i,c} \geq 1\big] \times 100\%, \qquad \text{Pass}^3_c(\mathcal{S}) = \frac{1}{|\mathcal{S}^+|} \sum_{i \in \mathcal{S}^+} \mathbb{I}\big[s_{i,c} = R_{i,c}\big] \times 100\%. \tag{2}$$

Their difference defines the reliability span $\text{Span}_3 = \text{Pass@3}_{\text{target}} - \text{Pass}^3_{\text{target}}$, capturing trial-to-trial instability; when analyzing scaling trends across intermediate trial budgets $k \in \{1, 2, 3\}$ on tasks with three executed runs, we compute $\text{Pass@}k$ and Pass^k via standard unbiased combinatoric estimators [15, 18].

Exclusion of Unexecuted API Refusals. To ensure that all metrics strictly measure operational environment adaptation rather than provider-level content-filter false positives on benign system-administration prompts such as inspecting account permissions or local proxy certificates, trials intercepted by an upstream API safety refusal prior to sandbox entry are excluded from the executed trial count $R_{i,c}$. Across our evaluation suite, eight of the nine evaluated models trigger zero refusals across all target runs; alongside primary metrics conditional on non-refusal executed runs, we

Table 1: Comparison of TwinEnv-Bench against representative software, operating-system, and tool-use agent benchmarks. Symbols denote full support ●, partial support ◐, and absence ○.

Benchmark	Execution setting	Live OS / shell execution	Counterfactual twin control	Probe-verified 3-state contract	Exposed vs. latent shift split	Host-isolated oracle & grading
SWE-bench Verified [5]	frozen repo container	◐	○	◐	○	◐
SWE-bench Live [8]	live-issue repo container	◐	○	◐	○	◐
OSWorld [11]	desktop GUI VM	●	○	○	○	◐
AppWorld [7]	mocked multi-app APIs	○	○	◐	○	●
τ-bench [18]	simulated tool APIs	○	○	○	○	●
Terminal-Bench [4]	single CLI container	●	○	◐	○	◐
TwinEnv-Bench (ours)	**paired twin OS sandboxes**	●	●	●	●	●

report end-to-end sensitivity treating refusals as failures in Table 4 and Section 4, with complete per-model refusal accounting in Appendix B, Table 12.

2.3 Positioning Relative to Existing Agent Benchmarks

Table 1 summarizes how TwinEnv-Bench differs from six widely adopted agent benchmarks across five methodological criteria: live operating-system and shell execution, single-variable counterfactual twin control, probe-verified three-state solvability contracts, explicit separation of exposed and latent shifts, and leak-proof host-isolated grading; broader related literature is reviewed in Section 6. Simulated API suites such as AppWorld [7] and τ-bench [18] enforce clean host-isolated state checks yet abstract away the operating system entirely, whereas repository suites such as SWE-bench Verified [5] and SWE-bench Live [8] freeze each codebase inside a static container pre-configured to pass baseline tests. Desktop and CLI benchmarks such as OSWorld [11] and Terminal-Bench [4] operate on real system utilities, yet evaluate each task in only a single static environment without counterfactual baseline controls, probe-verified single-property mutations, or a principled split between error-surfacing and sample-silent deployment constraints. TwinEnv-Bench satisfies all five criteria simultaneously, turning terminal agent evaluation from an observational leaderboard into a controlled causal diagnostic instrument.

3 The E1–E5 Environment Taxonomy and Counterfactual Suite

A credible benchmark for environment adaptation must satisfy two competing requirements: *ecological validity*, meaning every shifted property corresponds to hazards encountered in real-world software engineering and operations, and *causal purity*, meaning each task varies a single well-defined mechanism without unintended side-channel breakage or unobservable guesswork. As illustrated in Figure 3, TwinEnv-Bench achieves both properties through an incident-grounded taxonomy in Section 3.2, dual mechanical verification contracts in Section 3.3, and host discoverability paired with leak-proof isolated grading in Section 3.4.

3.1 Incident-Grounded Taxonomy Construction

Rather than inventing synthetic container puzzles, we constructed the TwinEnv-Bench taxonomy from the bottom up by auditing over two hundred real-world environment-related bug reports, pull requests, and post-mortems across leading open-source coding agents, including Anthropic's Claude Code, OpenAI's Codex CLI, OpenHands [3], and SWE-agent [2], alongside POSIX system standards, Linux kernel documentation on process isolation and resource scheduling, international timezone rules, and empirical software reproducibility studies [19].

From this empirical audit, we organize software and operating-system execution hazards into 5 hierarchical layers from E1 to E5, 15 dimension groups from E1.1 to E5.2, and 31 normalized fine-grained mechanisms. Table 2 summarizes all 15 dimension groups, their task counts and exposure splits across the 117 tasks, and typical host inspection targets; all 31 mechanisms and their corresponding open-source issue links are cataloged in Appendix A, Table 10.

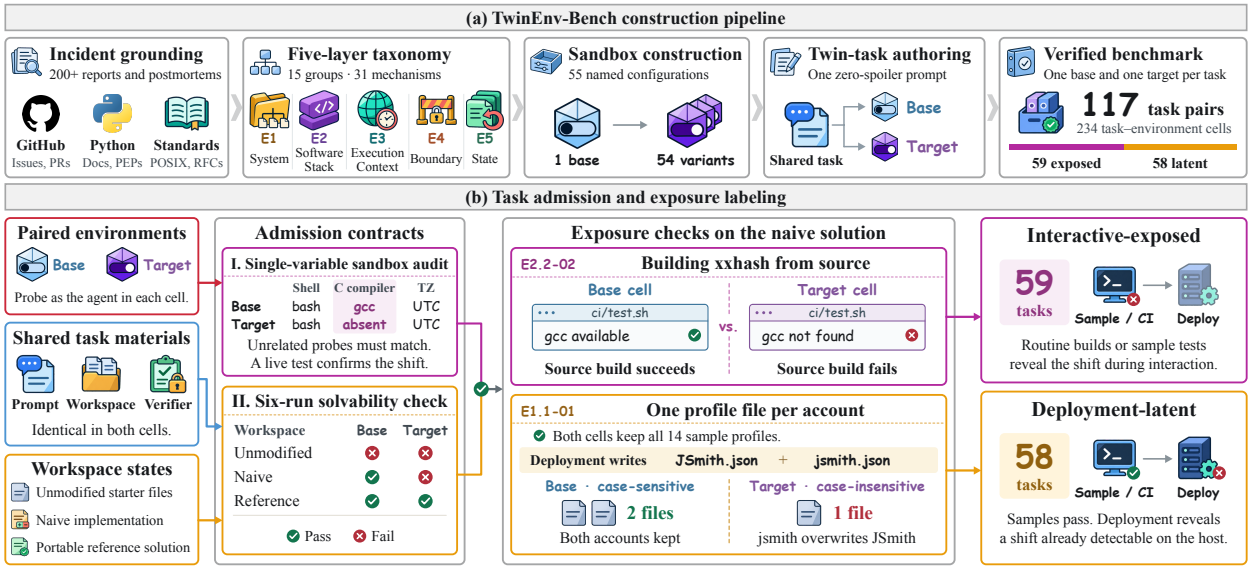


Figure 3: **Construction Pipeline, Dual Admission Contracts, and Exposure Labeling in TwinEnv-Bench.** Panel a traces the five-stage benchmark construction pipeline from over two hundred real-world engineering incident reports to 55 sandbox configurations and 117 paired tasks across 234 task–environment cells. Panel b details task admission under the single-variable sandbox audit and six-run three-state solvability matrix, followed by naive-solution exposure checks that partition the suite into 59 interactive-exposed tasks such as missing build compilers in E2.2-02 and 58 deployment-latent tasks such as case-insensitive profile overwrites in E1.1-01.

3.2 Architectural Walkthrough of Layers E1–E5

Layer E1: System Layer, 12 Tasks Across E1.1–E1.2. Layer E1 captures foundational storage and text-representation semantics governed by the underlying filesystem. Within E1.1 Filesystem Semantics, spanning 10 tasks and 5 mechanisms, greenhouse containers default to case-sensitive local directories where every input is a regular file with short ASCII names. In production environments, agents instead encounter case-insensitive volumes where renaming Logo.svg to logo.svg in a single step collides or overwrites files; decomposed Unicode filenames where accented characters stored in multi-codepoint form break exact string lookups; multi-byte UTF-8 filenames exceeding filesystem byte-length ceilings; cross-directory symbolic links; and non-rewinding named pipes that fail or block when opened and read a second time. Complementing storage semantics, E1.2 System Conventions spans 2 tasks modeling Windows-style CRLF line endings, where naive line-by-line text parsers retain stray carriage returns inside checksums or request headers.

Layer E2: Software Stack Layer, 24 Tasks Across E2.1–E2.4. Layer E2 spans userland language interpreters, package ecosystems, command-line toolchains, and headless browsers. E2.1 Language Runtime, covering 5 tasks and 2 mechanisms, evaluates adaptation across older Python 3.8 environments lacking modern type-union syntax and newer Python 3.14 releases that remove deprecated standard-library interfaces, as well as containers that provide python3 without an unversioned python command alias. E2.2 Dependencies and Package Management, covering 8 tasks and 3 mechanisms, tests slim production images where common third-party data libraries or C/C++ build compilers are absent, or where public package registries are replaced by internal mirrors and offline local wheel directories. E2.3 System Toolchain and Shell, covering 9 tasks and 2 mechanisms, replaces standard GNU utilities and bash with lightweight BusyBox or BSD toolsets and alternative default shells such as dash or zsh, exposing reliance on non-portable command flags and shell-specific syntax. Finally, E2.4 Browser Engine shifts headless browser automation across 2 tasks from Chromium to Firefox or WebKit, surfacing cross-engine layout and style differences.

Layer E3: Execution Context Layer, 43 Tasks Across E3.1–E3.2. Layer E3 constitutes the largest layer in TwinEnv-Bench, targeting inherited process settings that govern character decoding, regional formatting, timekeeping, and background scheduling. E3.1 Locale, Encoding, and Timezone, comprising 31 tasks across 3 mechanisms, covers regional

Table 2: Architecture of the TwinEnv-Bench E1–E5 five-layer environment taxonomy across 15 dimension groups, 31 fine-grained mechanisms, and 117 counterfactual tasks spanning 59 interactive-exposed and 58 deployment-latent instances, alongside typical host inspection targets.

Group	Dimension Group Name	Mechs	Tasks	Exp / Lat	Typical Host Inspection Targets
E1.1	Filesystem Semantics	5	10	3 / 7	Mount options, filename byte limits, special file types, and Unicode form
E1.2	System Conventions	1	2	1 / 1	Text line-ending format and repository newline settings
E2.1	Language Runtime	2	5	5 / 0	Interpreter version and available command aliases
E2.2	Dependencies & Packages	3	8	8 / 0	Installed libraries, package mirror settings, and build compilers
E2.3	System Toolchain & Shell	2	9	9 / 0	Default login shell and core utility implementations
E2.4	Browser Engine	1	2	2 / 0	Configured headless browser rendering engine
E3.1	Locale, Encoding & Timezone	3	31	8 / 23	System locale, character encoding, and timezone settings
E3.2	Process Context	4	12	6 / 6	User identity, default permission mask, terminal state, and scheduled jobs
E4.1	Permissions & Mounts	2	9	2 / 7	Partition mount flags, directory writability, and process privileges
E4.2	Network Boundaries	3	11	10 / 1	External connectivity, proxy settings, and DNS resolution
E4.3	System Clock Skew	1	5	5 / 0	System wall-clock time and clock-offset environment settings
E4.4	Credential Visibility	1	3	0 / 3	Runtime environment variables and credential placeholders
E4.5	Resource Limit Enforcement	1	8	0 / 8	Per-process file-size, descriptor, argument, and memory quotas
E5.1	Platform Residuals	1	1	0 / 1	Pre-existing bytecode caches and stale workspace lock files
E5.2	External Service Reliability	1	1	0 / 1	Upstream service status codes and batch retry behavior
Total	**5 Layers / 15 Dimension Groups**	**31**	**117**	**59 / 58**	**55 probe-verified sandbox configurations**

non-UTF-8 text encodings such as GBK and Big5; locale-dependent formatting and case-conversion rules, such as European comma decimals breaking numeric parsers or Turkish dotless-ı case conversion altering uppercase status keywords; and regional timezone hazards including daylight-saving transitions with missing or repeated local hours and non-integer UTC offsets such as +05:30. Complementing regional settings, E3.2 Process Context spans 12 tasks and 4 mechanisms evaluating background cron jobs that strip interactive PATH and environment variables, unprivileged service accounts encountering file-ownership restrictions on shared repositories, non-interactive standard input without an attached terminal, and restrictive default file-creation masks that leave new directories unreadable to downstream accounts.

Layer E4: Boundary Layer, 36 Tasks Across E4.1–E4.5. Layer E4 enforces filesystem permissions, mount boundaries, network perimeters, clock drift, credential scoping, and resource quotas. E4.1 Permissions and Mounts, spanning 9 tasks and 2 mechanisms, includes non-executable temporary directories, read-only application paths, cross-volume directory mounts where atomic file rename fails across devices, and restricted administrator containers stripped of discretionary file-override privileges. E4.2 Network Boundaries, spanning 11 tasks and 3 mechanisms, enforces complete offline isolation, corporate HTTP proxy routing, split-horizon internal DNS resolution, and read-only gateway proxies that block write requests. E4.3 System Clock Skew injects large wall-clock offsets or frozen system clocks across 5 tasks, testing certificate expiration handling and monotonic interval measurement. E4.4 Credential Visibility models zero-trust secret scoping across 3 tasks where credentials exist only during initialization or are replaced at runtime by

placeholder tokens. Finally, E4.5 Resource Limit Enforcement spans 8 tasks enforcing strict kernel quotas on maximum file size, command-line argument length, open file descriptors, and process memory.

Layer E5: State Layer, 2 Tasks Across E5.1–E5.2. Layer E5 captures persistent workspace artifacts and external service reliability. Under E5.1 Platform Residuals, a task workspace contains stale compiled Python bytecode stamped with future modification times, causing timestamp-based module imports to silently ignore source-code edits unless cleared. Under E5.2 External Service Reliability, an upstream HTTP service passes single-record health checks yet returns intermittent 503 overload errors under batch traffic, requiring idempotent retry logic.

3.3 Dual Mechanical Verification Contracts

To guarantee that every failure in TwinEnv-Bench reflects a genuine, solvable single-variable adaptation gap, benchmark construction enforces two automated mechanical contracts prior to admitting any task, as summarized in Figure 3b.

Contract I: Automated Single-Variable Sandbox Audit. Modifying one system setting inside a container can easily introduce unintended side effects elsewhere in the environment, such as installing a regional language pack that inadvertently alters default tool paths or sorting behavior. To rule out collateral drift across our 55 sandbox configurations cataloged in Appendix B, Table 11, every shifted sandbox e^{target} targeting system property m^* is audited automatically against the baseline container e^{base} under the exact user account and login shell seen by the agent. Letting $d_m(e)$ denote the measured state of system property m inside container e, admission requires:

$$d_{m^*}\big(e^{\text{target}}\big) \neq d_{m^*}\big(e^{\text{base}}\big), \qquad \text{and} \qquad d_m\big(e^{\text{target}}\big) = d_m\big(e^{\text{base}}\big) \quad \forall\, m \notin \mathcal{M}(m^*), \tag{3}$$

where the first condition verifies via both probe readings and a live runtime test that the intended constraint m^* actively takes effect, such as confirming that a non-executable temporary mount blocks binary execution, and the second condition ensures that every unrelated property outside m^* and its direct physical consequences $\mathcal{M}(m^*)$, such as offline isolation also preventing external DNS lookups, remains identical to the baseline container.

Contract II: Six-Run Three-State Solvability Matrix. Every task package bundles not only the initial workspace and held-out grading suite, but also two executable reference implementations: an environment-naive implementation W_i^{naive} that satisfies the functional specification under standard greenhouse defaults, and an environment-aware reference implementation W_i^{ref}. Admission into TwinEnv-Bench requires passing all six cells of the deterministic verification matrix across e_i^{base} and e_i^{target}:

$$\begin{aligned}
\text{Unmodified workspace}: &\quad V_i(W_{i,0},\, e_i^{\text{base}}) = 0 \quad \wedge \quad V_i(W_{i,0},\, e_i^{\text{target}}) = 0, \\
\text{Naive implementation}: &\quad V_i(W_i^{\text{naive}},\, e_i^{\text{base}}) = 1 \quad \wedge \quad V_i(W_i^{\text{naive}},\, e_i^{\text{target}}) = 0, \\
\text{Reference solution}: &\quad V_i(W_i^{\text{ref}},\, e_i^{\text{base}}) = 1 \quad \wedge \quad V_i(W_i^{\text{ref}},\, e_i^{\text{target}}) = 1.
\end{aligned} \tag{4}$$

Equation 4 eliminates three classic benchmark pathologies by construction: failing on the unmodified workspace rules out trivial pre-solved setups; failing in the target cell under the naive implementation proves that relying on default assumptions is insufficient under Δd_i; and passing both cells under the reference solution proves that a single portable codebase solves the task without prompt spoilers.

3.4 Host Discoverability and Leak-Proof Isolated Grading

Fair counterfactual evaluation requires a strict two-sided information boundary during agent interaction: every active host constraint must be discoverable inside the sandbox so tasks never degenerate into black-box guesswork, while all grading oracles and credentials must remain physically inaccessible outside the sandbox so agents cannot inspect expected answers.

Zero-Spoiler Instructions and Host Discoverability. All 117 task prompts follow a natural, zero-spoiler engineering style that specifies functional inputs, outputs, command-line interfaces, and business edge cases, closing with operational phrasing such as "The result will be run directly on this machine, so make sure it actually works here" without hinting at encodings, timezones, scheduled jobs, mounts, or resource quotas. In the 58 deployment-latent

tasks, workspace sample inputs are intentionally small, clean, and invoked directly in the interactive shell so naive test runs pass without error, yet every instance strictly upholds the *silent in sample, discoverable on host* principle: whenever production grading runs under a scheduled background context that differs from the login shell, the corresponding schedule configuration and wrapper scripts are world-readable inside the container; whenever production workloads or service accounts enforce resource quotas, mount boundaries, or permission masks, those limits are active from step $t = 0$ and inspectable via standard read-only commands such as locale, ulimit, or findmnt; and filesystem properties such as case-insensitive directories, symbolic links, named pipes, and stale bytecode caches reside directly in the live workspace.

Host-Isolated Execution and Post-Termination Grading. Conversely, to prevent oracle or credential leakage through mounted test files, process environments, or container image layers, TwinEnv-Bench executes all evaluations through a zero-dependency host dispatcher spanning four isolated runtime profiles detailed in Appendix B.2. All model API requests, authentication keys, conversation histories, and reference solutions remain strictly inside the host controller process, while container startup scripts initialize the workspace from a temporary staging directory and purge all staging artifacts completely before turn 1. On host-isolated tasks, every agent command runs through a dedicated privilege-dropping wrapper under the target user and group identity, default file-creation mask, and kernel quotas on file size, open descriptors, stack size, and virtual memory, verified prior to turn 1 by automated visibility checks under the agent's own account. Only after the agent invokes the completion tool, finishes its final turn, or reaches the step or timeout limit does the host controller inject the held-out test inputs and verification script to compute the binary verdict $Y_i \in \{0, 1\}$, with offline integrity across all 1,709 task files and permission bits cryptographically verified against SHA-256 manifest digests.

Table 3: **Engineering composition and per-task scale statistics across TwinEnv-Bench.** Left columns summarize taxonomic coverage, sandbox infrastructure, and language ecosystems; right columns report mean and median per-task artifact sizes across all 117 task packages, showing that environment-safe reference implementations add a median of only 7 lines over naive greenhouse solutions.

Suite Composition & Isolation		**Per-Task Artifact Scale, Mean / Median**	
Dimension	**Count**	**Artifact Metric**	**Mean / Med.**
Counterfactual tasks / twin cells	117 / 234	Zero-spoiler prompt length, words	67.8 / 68
Layers / dimension groups / mechanisms	5 / 15 / 31	Initial workspace & setup files	5.4 / 5
Exposure split: exposed / latent	59 / 58	Initial workspace & setup lines	1,046.1 / 96
Probe-verified sandboxes / profiles	55 / 4	Naive implementation length, lines	15.2 / 13
Python / Shell / polyglot tasks	81 / 51 / 14	Reference solution length, lines	24.8 / 22
SHA-256 verified package files	1,709	Adaptation delta, ref minus naive lines	+9.6 / +7
Per-task execution timeout, seconds	1,200–1,500	Held-out judge / verifier mean lines	1,712.9 / 29.2

Micro-Scale Controlled Scope and Adaptation Patch Scale. Table 3 summarizes the engineering footprint across all 117 task packages. By design, TwinEnv-Bench operates as a controlled micro-scale scripting, systems, and data-engineering benchmark rather than a multi-thousand-line repository refactoring suite: concise operational tickets averaging 67.8 words target compact multi-file Python, Shell, and polyglot Java, Node.js, C, SQL, or web workspaces paired with production-scale held-out judge feeds averaging 1,712.9 lines and deterministic verification scripts averaging 29.2 lines. Comparing paired reference implementations W_i^{ref} at 24.8 mean and 22 median lines against naive greenhouse implementations W_i^{naive} at 15.2 mean and 13 median lines shows a mean adaptation delta of just +9.6 lines and a median of +7 lines—keeping baseline code navigation minimal so outcomes isolate host diagnosis and boundary-safe adaptation rather than large-codebase search.

4 Main Experimental Results

Across Sections 4 and 5, we evaluate nine frontier language models on TwinEnv-Bench to answer four overarching research questions:

- **RQ1, Counterfactual Shift Degradation:** How severely do single-variable environment shifts degrade agent performance relative to standard greenhouse sandboxes, and do task-level failures trace cleanly to the controlled shift?
- **RQ2, Exposure Regime Inversion:** How do solve rates and model rankings change when discrepancies move from error-surfacing interactive-exposed shifts to sample-silent deployment-latent shifts?
- **RQ3, Host Reconnaissance and Premature Blind Exit:** How do upfront timing and multi-step depth of host probing govern survival on deployment-latent shifts, and why do unprobed trajectories fail?
- **RQ4, Causal Prompt Intervention and the *probe-to-action* Gap:** Does explicit reconnaissance prompting causally rescue deployment-latent adaptation, or does a substantial cross-model conversion gap from observed host state to boundary-safe code remain?

After detailing our experimental setup in Section 4.1, we address RQ1 in Section 4.2 and RQ2 in Section 4.3, followed by complementary analyses of cross-layer family specialization in Section 4.4 and cost–reliability trade-offs in Section 4.5. Section 5 then dissects command-level trajectories and controlled prompt interventions to answer RQ3 and RQ4.

4.1 Experimental Setup and Evaluated Models

Evaluated Frontier Models. Our evaluation spans nine frontier models representing diverse post-training lineages and cost tiers: Claude Opus 5.5 from Anthropic [20, 21], Qwen 3.8 Max from Alibaba [22], GPT-6.1 Sol from OpenAI [23–25], Gemini 4 Argon from Google [26], Hy4 Flash from Tencent [27, 28], DeepSeek V4.1 Flash from DeepSeek [29, 30], GLM-5.3 Flash from Zhipu AI [31], Grok 4.7 from xAI [32], and Kimi K3.1 Pro from Moonshot AI [33, 34]. Every model operates through native tool-calling for shell execution and file editing across $K = 3$ independent trials per cell under vendor-recommended high-reasoning settings; exact snapshot identifiers, endpoint protocols, and non-refusal trial counts appear in Appendix B, Table 12. Standard container and scheduled tasks allocate up to 32,768 output tokens per assistant turn, whereas the 19 host-isolated pressure tasks enforce a 6,000-token per-turn output cap alongside restrictive umask defaults, POSIX rlimits, and Linux cgroup memory quotas to prevent brute-force inline heredoc dumps.

Complete Counterfactual Grid Coverage. All 6,318 scheduled trials across 9 models, 117 tasks, 2 paired cells, and 3 repeats reached terminal verdicts with zero infrastructure gaps, comprising all 3,159 target trials with 3,115 non-refusal executed runs and all 3,159 base trials with 3,120 non-refusal executed runs, totaling 6,235 executed main-benchmark trajectories. Across the benchmark, non-refusal runs form 1,047 paired base–target task–model cells out of 1,053 total cells, with the remaining 6 cells reflecting tasks where Claude Opus 5.5 triggered upstream API safety refusal across all three trials, while eight of the nine models record zero target refusals and seven record zero refusals across both cells.

4.2 RQ1: Collapse of the Greenhouse Illusion and Empirical Monotonicity

Figure 4 and Table 4 present overall counterfactual accuracy, regime breakdowns, multi-run reliability, latent reconnaissance behavior, and per-run resource footprints across all nine frontier models.

Near-Saturated Baselines vs. Severe Target Collapses. Across all 117 baseline containers, every evaluated model achieves between 95.1% for Qwen 3.8 Max and 99.7% for Hy4 Flash in single-run Pass@1, with a nine-model mean of 98.1%. This near-ceiling baseline confirms that prompt comprehension, domain algorithms, and routine command-line syntax pose virtually no obstacle on TwinEnv-Bench tasks. Yet when a single verified environment property shifts in the target sandbox, Pass@1 conditional on non-refusal executed runs plummets by 31.3 to 72.3 percentage points, spanning 67.1% for Claude Opus 5.5 down to 26.8% for ninth-ranked Kimi K3.1 Pro and averaging just 50.4% across models, a mean drop of 47.6 percentage points. Strict three-of-three production reliability Pass^3 degrades even further to a 39.1% mean: conditional on non-refusal runs, only Claude Opus 5.5 at 59.5%, GPT-6.1 Sol at 58.1%, and Gemini 4 Argon at 51.3% solve more than half of the benchmark consistently across repeats, followed by Qwen 3.8 Max at 47.0% and Hy4 Flash at 45.3%, while all four second-tier models fall below 29%. Because Claude Opus 5.5 triggers upstream API safety refusals on 44 of 351 target trials at 12.5% whereas the other eight models incur zero target refusals, the bottom row of Table 4 also reports end-to-end refusal-as-0 sensitivity: penalizing refusals as failures moves Claude

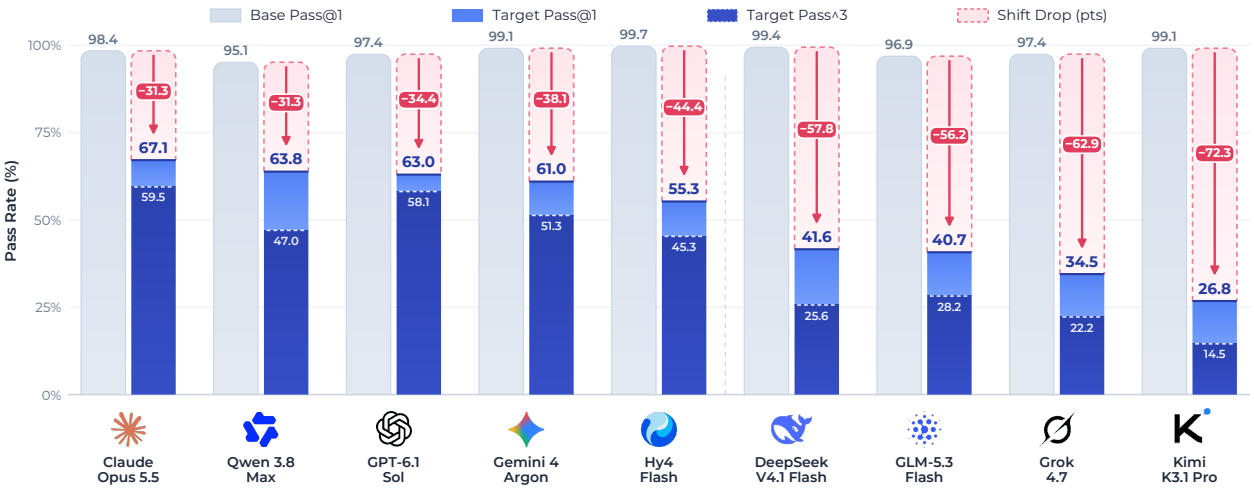


Figure 4: **Collapse of the Greenhouse Illusion across Nine Frontier Models.** Models approach saturation on the standard base container at 95.1%–99.7% Pass@1 across all 117 tasks, yet suffer 31.3–72.3 percentage-point drops conditional on non-refusal executed runs on the paired target tasks, with strict three-of-three production reliability Pass^3 falling to 14.5%–59.5%. Nearly a 14 percentage-point structural gap separates the five-model frontier tier above 55% Target Pass@1 from the remaining four models below 42%.

Table 4: **Main counterfactual leaderboard across all 117 tasks.** Outcome, reliability, and latent-reconnaissance columns report percentages across three independent trials per cell over all 234 paired baseline and target cells: Probe and Early record the share of deployment-latent target runs executing at least one host probe and probing within the first two steps. Target resource columns report mean shell commands, total tokens in thousands, wall-clock duration in seconds, and audited provider cost in USD per executed target run. Across the nine models, **bold** and underlining mark the best and second-best result in each ranked column, with smaller magnitude preferred for Δ_{shift}, Δ_{regime}, and Span; resource columns and summary rows are unranked. Main model rows report metrics conditional on non-refusal executed runs under Section 2.2; the bottom row reports end-to-end sensitivity for Claude Opus 5.5 when counting its 37 baseline and 44 target refusals as failures, while the other eight models incur zero target refusals.

	Counterfactual overall			Exposure regimes			Multi-run reliability			Latent recon.		Target resources, mean/run			
Model	**Base@1**	**Target@1**	Δ_{shift}	**Exp.**	**Lat.**	Δ_{regime}	**Pass@3**	**Pass^3**	**Span**	**Probe**	**Early**	**Cmds**	**Tok. k**	**Time s**	**Cost $**
Frontier tier Target@1 above 55% across 117 tasks															
Claude Opus 5.5	98.4	**67.1**	**−31.3**	74.1	**59.3**	−14.8	73.9	**59.5**	14.4	73.1	**69.0**	4.6	20.0	45.5	0.073
Qwen 3.8 Max	95.1	63.8	**−31.3**	75.1	52.3	−22.8	**76.9**	47.0	29.9	**82.8**	14.4	27.1	358.8	725.6	0.237
GPT-6.1 Sol	97.4	63.0	−34.4	**77.4**	48.3	−29.1	68.4	58.1	**10.3**	70.7	68.4	4.7	15.9	50.4	0.031
Gemini 4 Argon	99.1	61.0	−38.1	73.4	48.3	−25.1	70.1	51.3	18.8	58.0	42.5	10.0	137.6	178.5	0.118
Hy4 Flash	**99.7**	55.3	−44.4	69.5	40.8	−28.7	65.8	45.3	20.5	64.4	48.3	12.5	115.2	120.1	0.002
Second tier Target@1 below 42% across 117 tasks															
DeepSeek V4.1 Flash	99.4	41.6	−57.8	46.3	36.8	**−9.5**	58.1	25.6	32.5	59.8	25.3	16.2	157.8	115.3	0.005
GLM-5.3 Flash	96.9	40.7	−56.2	53.7	27.6	−26.1	52.1	28.2	23.9	40.8	10.3	15.5	158.2	197.9	0.008
Grok 4.7	97.4	34.5	−62.9	46.3	22.4	−23.9	47.0	22.2	24.8	44.3	33.9	8.3	80.0	117.9	0.088
Kimi K3.1 Pro	99.1	26.8	−72.3	39.5	13.8	−25.7	39.3	14.5	24.8	32.2	20.1	9.2	74.8	217.0	0.101
9-model mean	98.1	50.4	−47.6	61.7	38.8	−22.9	61.3	39.1	22.2	58.5	36.9	12.0	124.3	196.5	0.074
Opus 5.5, Refusal-as-0	*88.0*	*58.7*	*−29.3*	*67.8*	*49.4*	*−18.4*	*70.1*	*47.9*	*22.2*	—	—	—	—	—	—

Opus 5.5 to fourth overall at 58.7% Pass@1 and third on Pass^3 at 47.9%, shifting unconditional Pass@1 first place to Qwen 3.8 Max at 63.8% ahead of GPT-6.1 Sol at 63.0% and Gemini 4 Argon at 61.0%, and unconditional Pass^3 first place to GPT-6.1 Sol at 58.1%, even though Claude Opus 5.5 still leads at 68.1% versus 63.0% for Qwen 3.8 Max on the 92 tasks where all nine models incur zero refusals. Under both accounting rules, the leaderboard divides cleanly into the same five-model frontier tier above 55% Target Pass@1, separated by a 13.7 percentage-point gap from the four-model second tier below 42% led by DeepSeek V4.1 Flash at 41.6% and GLM-5.3 Flash at 40.7%.

Task-Level Counterfactual Attribution. To verify at the individual task–model level that target failures stem from the environmental shift Δd_i rather than random coding mistakes, we compare majority-vote pass outcomes across all 1,047 paired task–model cells, as summarized in Table 5. Exactly 524 pairs, representing 50.0% of cells, pass in both environments; 502 pairs, or 47.9%, pass in the baseline container yet fail in the shifted target sandbox; only 19 pairs, or 1.8%, fail in both; and just 2 pairs, or 0.19%, fail in baseline while passing in target due to single-run sampling variance. Consequently, 96.4% of all target task failures, 502 out of 521, occur on tasks that the exact same model solves reliably under standard defaults, and 1,045 of 1,047 pairs satisfy counterfactual empirical monotonicity at 99.8%, confirming that our three-state solvability contract from Section 3.3 holds cleanly across live frontier agents.

> **Answer to RQ1:** Single-variable environment shifts shatter near-saturated greenhouse competence at 98.1% mean Base Pass@1, cutting Target Pass@1 by 31.3–72.3 percentage points to a 50.4% mean and reducing strict three-run reliability Pass^3 to 39.1%. Across 1,047 paired task–model cells, 96.4% of target failures occur on tasks the same model solves in the baseline container and 99.8% satisfy counterfactual empirical monotonicity, isolating adaptation failure from algorithmic incompetence.

4.3 RQ2: Exposed vs. Latent Regime Shifts and Ranking Inversions

Disaggregating target performance across the 59 interactive-exposed tasks and 58 deployment-latent tasks in Figure 5 and Table 4 uncovers substantial regime drops and leaderboard inversions. Because fault observability varies naturally across system layers—E2 Software Stack accounts for 24 exposed tasks and zero latent tasks, whereas E3 Execution Context, E4.5 Resource Limits, and E5 State concentrate latent shifts—Δ_{regime} reflects the joint effect of silent sample-time observability and its associated production-layer distribution, though restricting strictly to the overlapping E1, E3, and E4 layers still shows a 29.6-point nine-model drop from 68.6% across 35 exposed tasks to 39.0% across 56 latent tasks.

Leaderboard Reversal Between Upfront Shallow Checkers and Deep Explorers. On interactive-exposed shifts, where discrepancies surface immediately through stack traces, non-zero exit codes, or malformed stdout on sample runs, GPT-6.1 Sol claims first place at 77.4%, ahead of Qwen 3.8 Max at 75.1%, Claude Opus 5.5 at 74.1% on non-refusal runs, Gemini 4 Argon at 73.4%, and Hy4 Flash at 69.5%. Moving to deployment-latent tasks cuts nine-model mean Pass@1 by 22.9 percentage points from 61.7% to 38.8% and inverts top rankings. Notably, as shown in Table 4, GPT-6.1 Sol is not a purely unprobing agent—it initiates an early check within steps 1–2 on 68.4% of latent runs, yet exemplifies *upfront shallow checking* that rarely follows up with multi-step inspection on scheduled contexts or kernel quotas, dropping 29.1 points to 48.3% into a tie for third alongside Gemini 4 Argon; even within the overlapping E1, E3, and E4 layers, GPT-6.1 Sol falls from 78.1% on exposed tasks to 48.8% on latent tasks. Meanwhile, Hy4 Flash drops 28.7 points to fifth at 40.8%, and low-probing GLM-5.3 Flash and Kimi K3.1 Pro fall by 26.1 and 25.7 points to 27.6% and 13.8%.

In contrast, models that combine upfront checks with multi-step host inspection prove substantially more resilient. Conditional on non-refusal executed runs, Claude Opus 5.5 probes within the first two steps on 69.0% of latent runs and drops by only 14.8 points from 74.1% to 59.3%, climbing from third on exposed shifts to first on latent shifts ahead of Qwen 3.8 Max at 52.3% and GPT-6.1 Sol and Gemini 4 Argon at 48.3%. Because Claude Opus 5.5 incurs 29 safety refusals across 174 scheduled latent trials at 16.7%, counting refusals as failures places Claude Opus 5.5 second on latent tasks at 49.4% behind zero-refusal Qwen 3.8 Max at 52.3%—yet both multi-step explorers still outscore exposed leader GPT-6.1 Sol at 48.3%, preserving the regime ranking inversion under both protocols. In the second tier, DeepSeek V4.1 Flash records the smallest regime drop across all evaluated models at 9.5 points, sliding from 46.3% to 36.8% and overtaking GLM-5.3 Flash for sixth place.

Compound Stress on the 19-Task Host-Isolated Pressure Subset. Table 5 contrasts the 98 standard and scheduled tasks against the 19 host-isolated pressure tasks, which form a compound stress subset combining silent RLIMIT_FSIZE and RLIMIT_NOFILE quotas, cross-UID file ownership, restrictive umask 0077 permissions, and non-seekable FIFO named pipes with a tightened 6,000-token per-turn output cap and natural-turn termination. Even under this tighter output budget, every model achieves 94.7%–100.0% Base Pass@1 on the 19 tasks, with five of the nine models scoring a perfect 100.0%. On the 98-task target subset, GPT-6.1 Sol ranks first at 72.4%, ahead of Claude Opus 5.5 at 71.3% on non-refusal runs or 64.3% under refusal-as-0, Qwen 3.8 Max at 70.1%, Gemini 4 Argon at 69.4%,



Figure 5: **Shift Exposure Regime Leaderboard Ordered by Deployment-Latent Pass@1.** Moving from the 59 error-surfacing interactive-exposed tasks to the 58 sample-silent deployment-latent tasks triggers universal 9.5–29.1 percentage-point drops across silent observability and production-layer shifts, overturning leaderboard rankings: exposed leader GPT-6.1 Sol drops 29.1 points from 77.4% into a tie for third at 48.3%, whereas conditional on non-refusal runs Claude Opus 5.5 at 59.3% and zero-refusal Qwen 3.8 Max at 52.3% claim the top two positions.

and Hy4 Flash at 62.6%. Under the compound silent-boundary and output-budget pressure of the 19-task target subset, however, GPT-6.1 Sol drops to sixth place at 14.0% behind Gemini 4 Argon and Hy4 Flash tied at 17.5%, while GLM-5.3 Flash at 3.5%, Grok 4.7 at 1.8%, and Kimi K3.1 Pro at 1.8% are nearly wiped out. Meanwhile, deep-probing models hold the top three positions, led on non-refusal runs by concise Claude Opus 5.5 at 40.5%, Qwen 3.8 Max at 31.6%, and DeepSeek V4.1 Flash at 24.6%, or under refusal-as-0 by Qwen 3.8 Max at 31.6% and Claude Opus 5.5 at 29.8%, reflecting the joint demand for multi-step boundary inspection and compact output generation.

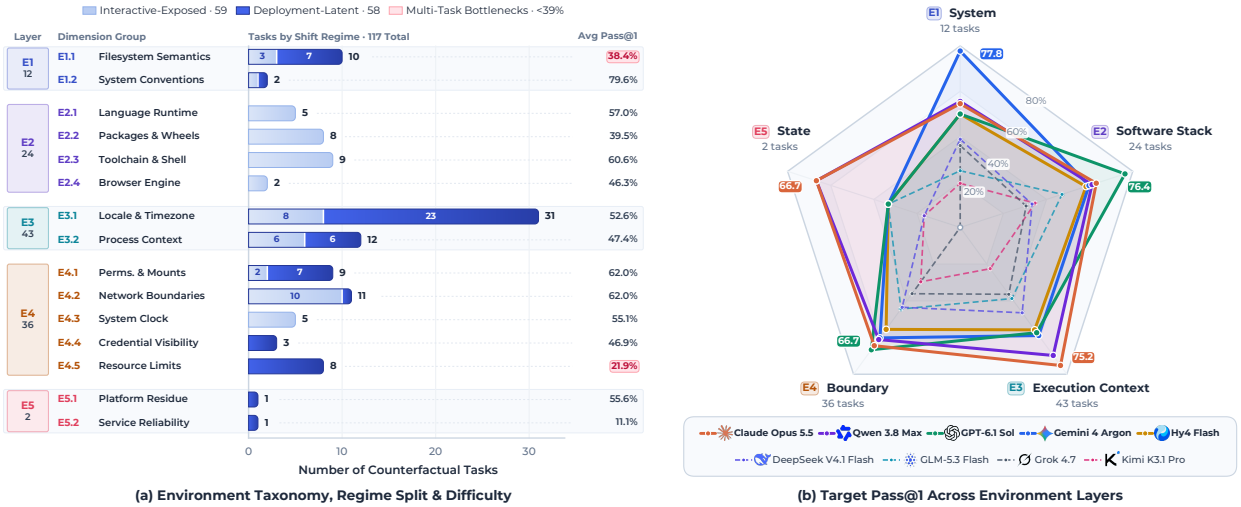
> **Answer to RQ2:** Moving from interactive-exposed shifts to deployment-latent shifts, which combines silent sample-time observability with execution-context, quota, and state-layer shifts, cuts nine-model mean Pass@1 by 22.9 percentage points from 61.7% to 38.8% and overturns leaderboard rankings. Upfront shallow checker GPT-6.1 Sol drops from 77.4% first place on exposed tasks to 48.3% on latent tasks and 14.0% under the compound silent and 6,000-token constraints of the 19-task subset, whereas multi-step explorers lead latent shifts—headed by Claude Opus 5.5 at 59.3% and Qwen 3.8 Max at 52.3% on non-refusal runs, or by Qwen 3.8 Max at 52.3% and Claude Opus 5.5 at 49.4% when refusals are counted as failures.

4.4 Cross-Layer Specialization, Bottlenecks, and Oracle Union

Figure 6 and Table 6 examine capability profiles across the five E1–E5 architectural layers.

Table 5: **Comparison between the 98 standard or scheduled tasks and the 19 host-isolated compound pressure tasks.** The 19-task subset combines silent umask, rlimit, FIFO, and cross-UID isolation boundaries with a tighter 6,000-token per-turn output budget and natural-turn termination. Columns report percentages conditional on non-refusal executed runs, where counting refusals as failures places Claude Opus 5.5 at 64.3% on the 98-task target set and 29.8% on the 19-task subset. Across all 1,047 paired base–target task–model cells, 1,045 satisfy counterfactual empirical monotonicity at 99.8%, with 524 passing both, 502 passing base and failing target to account for 96.4% of all target failures, 19 failing both, and only 2 failing base and passing target.

Model	98-Set Base@1	98-Set Target@1	19-Set Base@1	19-Set Target@1	Δ_{target}
Claude Opus 5.5	98.9	71.3	95.3	**40.5**	−30.8
Qwen 3.8 Max	94.9	70.1	96.5	31.6	−38.5
GPT-6.1 Sol	96.9	**72.4**	**100.0**	14.0	−58.4
Gemini 4 Argon	**100.0**	69.4	94.7	17.5	−51.9
Hy4 Flash	99.7	62.6	**100.0**	17.5	−45.1
DeepSeek V4.1 Flash	99.3	44.9	**100.0**	24.6	−20.3
GLM-5.3 Flash	96.6	48.0	98.2	3.5	−44.5
Grok 4.7	96.9	40.8	**100.0**	1.8	−39.0
Kimi K3.1 Pro	99.0	31.6	**100.0**	1.8	−29.8



(a) Environment Taxonomy, Regime Split & Difficulty

(b) Target Pass@1 Across Environment Layers

Figure 6: **E1–E5 Architectural Composition and Cross-Layer Capability Profiles.** Panel a breaks down all 117 tasks across 5 layers and 15 dimension groups, showing task counts for the 59 interactive-exposed and 58 deployment-latent instances alongside 9-model mean Target Pass@1 and highlighting multi-task bottleneck groups below 39%. Panel b plots the cross-layer Target Pass@1 radar illustrating complementary model strengths across system layers.

Complementary Family Specializations and the 88.0% Oracle Union. As shown in Table 6, across the four substantive E1–E4 layers spanning 12 to 43 tasks each, no single model family dominates every layer of the system stack. Gemini 4 Argon leads E1 System Layer at 77.8%, outperforming second-place Qwen 3.8 Max at 55.6% by 22.2 percentage points through strong handling of Unicode NFD/NFC filename normalization, case-folding mounts, and cross-directory symlinks. GPT-6.1 Sol leads both E2 Software Stack at 76.4%, 13.3 points ahead of Claude Opus 5.5, and E4 Boundary Layer at 66.7%, reflecting reliable adaptation across Python/Node runtime shifts, BusyBox/musl userlands, and scoped TLS or proxy credentials. Meanwhile, conditional on non-refusal runs, Claude Opus 5.5 leads E3 Execution Context at 75.2%, ahead of Qwen 3.8 Max at 69.8% and Gemini 4 Argon at 58.9%, by proactively checking cron environment stripping and POSIX locale or TZ overrides, while Hy4 Flash maintains a balanced 50.0%–58.3% profile across E1–E4 and leads all nine models on E4.1 Permissions and Mounts at 77.8%; on the two-task E5 State Layer, Claude Opus 5.5 and Qwen 3.8 Max both reach 66.7%, with full group-level scores reported in Appendix C. Because

Table 6: **Cross-layer Target Pass@1 and Oracle Union across the five system layers.** Percentage solve rates illustrate complementary model specializations alongside the nine-model Oracle Union achieved by selecting the best-performing model per task, reaching 88.0% under majority voting and 93.2% when counting any passing trial.

Model / Ensemble	E1 Sys (12)	E2 Stack (24)	E3 Ctx (43)	E4 Bnd (36)	E5 State (2)	Overall (117)
Claude Opus 5.5	54.5	63.1	**75.2**	64.4	**66.7**	**67.1**
Qwen 3.8 Max	55.6	61.1	69.8	61.1	**66.7**	63.8
GPT-6.1 Sol	50.0	**76.4**	57.4	**66.7**	33.3	63.0
Gemini 4 Argon	**77.8**	59.7	58.9	60.2	33.3	61.0
Hy4 Flash	50.0	58.3	55.8	55.6	33.3	55.3
DeepSeek V4.1 Flash	38.9	33.3	46.5	43.5	16.7	41.6
GLM-5.3 Flash	25.0	47.2	38.8	44.4	33.3	40.7
Grok 4.7	36.1	30.6	36.4	36.1	0.0	34.5
Kimi K3.1 Pro	19.4	34.7	22.5	29.6	16.7	26.8
Oracle Union, majority vote	**83.3**	**91.7**	**83.7**	**91.7**	**100.0**	**88.0**
Oracle Union, any trial	83.3	95.8	88.4	100.0	100.0	93.2

these layer strengths are genuinely complementary, a nine-model Oracle Union, which selects the best-performing model per task, solves 88.0% of the benchmark under majority voting, or 103 of 117 tasks, standing 20.9 percentage points above the highest single-model non-refusal score of 67.1% and 24.2 points above the top unconditional score of 63.8%, and reaches 93.2%, or 109 of 117 tasks, when counting any passing trial. Only 8 tasks, representing 6.8% of the suite, remain unsolved across all executed target runs out of 27 scheduled trials per task; 7 of these 8 are deployment-latent tasks involving FIFO named pipes, symlinked directory traversals, fractional UTC+05:30 cron timezones, or cross-UID file ownership, as detailed in Appendix C, Table 14.

Multi-Task Bottleneck Groups and Single-Task Service Observation. Comparing nine-model mean solve rates across multi-task dimension groups in Figure 6a and Appendix C, Figure 12, highlights two primary cross-model bottlenecks below 39%: E4.5 Resource Limit Enforcement across 8 tasks at 21.9%, where strict RLIMIT_FSIZE, RLIMIT_NOFILE, and cgroup memory ceilings alongside the 6,000-token output cap defeat monolithic in-memory pipelines, pulling Qwen 3.8 Max and Hy4 Flash down to 4.2% while concise Claude Opus 5.5 reaches 76.2% on non-refusal runs; and E1.1 Filesystem Semantics across 10 tasks at 38.4%, driven by silent overwrites on case-insensitive volumes and EXDEV cross-mount rename failures. In addition, as a single-task case observation on E5.2 External Service Reliability at 11.1% across 1 task, only Qwen 3.8 Max at 66.7% and Claude Opus 5.5 at 33.3% implement idempotent retry logic against transient HTTP 503 responses during held-out batch export, while the other seven models score zero.

4.5 Adaptation Overhead and Cost–Reliability Pareto Frontier

Operating agents outside uniform greenhouse containers also entails substantial monetary and latency costs. Combining audited provider billing logs with per-run token and wall-clock telemetry yields the Cost–Reliability Pareto analysis in Figure 7 and Table 7.

Cost–Reliability Pareto Frontier. Plotting strict production reliability Pass^3 conditional on non-refusal runs against mean USD cost per executed target run in Figure 7a traces a three-node Pareto frontier spanning three price tiers. In the sub-cent tier, Hy4 Flash achieves 45.3% Pass^3 and 55.3% Pass@1 at just \$0.002 per run across 115.2k tokens and 120.1s, strictly dominating both DeepSeek V4.1 Flash at \$0.005 with 25.6% Pass^3 and GLM-5.3 Flash at \$0.008 with 28.2% Pass^3 on both cost and reliability. At the cost-efficiency knee, GPT-6.1 Sol reaches 58.1% Pass^3 at \$0.031 per run with zero refusals, retaining 97.6% of peak non-refusal reliability at 2.4× lower cost than Claude Opus 5.5 through compact trajectories averaging 4.7 shell commands, 15.9k tokens, and 50.4s. Conditional on non-refusal executed runs, Claude Opus 5.5 reaches 59.5% Pass^3 and 67.1% Pass@1 at \$0.073 per run across 4.6 shell commands and 20.0k tokens, remaining 1.6× cheaper than Gemini 4 Argon at \$0.118 per run and 3.2× cheaper than Qwen 3.8 Max at \$0.237 per run, whose exhaustive multi-turn exploration averages 27.1 commands, 358.8k tokens, and 725.6s per run; when unexecuted safety refusals are counted as deployment failures, however, Claude Opus 5.5 drops to 47.9% Pass^3 across



Figure 7: **Cost–Reliability Pareto Frontier and Environment Friction Tax across Nine Models.** Panel a plots mean USD cost per executed target run on a logarithmic scale against Pass^3 as filled circles and Target Pass@1 as hollow squares conditional on non-refusal runs, tracing a three-node Pareto frontier from Hy4 Flash at \$0.002 and 45.3% Pass^3 through the GPT-6.1 Sol knee at \$0.031 and 58.1% Pass^3 to Claude Opus 5.5 at \$0.073 and 59.5% Pass^3; under refusal-as-0 accounting, GPT-6.1 Sol becomes the unconditional reliability peak. Panel b shows matched-pair Base → Target token and wall-clock inflation across regimes and models.

Table 7: **Audited cost, latency, and token friction across nine frontier models.** Columns report provider cost in USD per executed target run, strict three-run reliability Pass³ conditional on non-refusal runs, mean target token usage and wall-clock latency, and baseline-to-target token inflation on interactive-exposed tasks; ★ marks the three-node non-refusal Pass³ Pareto frontier, whereas counting refusals as failures places Claude Opus 5.5 at 47.9% Pass³ and leaves GPT-6.1 Sol as the unconditional reliability peak.

Model	Cost / Run	Pass³	Mean Tokens	Mean Latency	Exposed Token Friction (Base → Target)
Claude Opus 5.5★	\$0.073	59.5%	20.0k	45.5s	15.2k → 17.6k (×1.16)
Qwen 3.8 Max	\$0.237	47.0%	358.8k	725.6s	340.0k → 424.6k (×1.25)
GPT-6.1 Sol★	\$0.031	58.1%	15.9k	50.4s	13.9k → 16.6k (×1.19)
Gemini 4 Argon	\$0.118	51.3%	137.6k	178.5s	44.8k → 192.3k (×4.30)
Hy4 Flash★	\$0.002	45.3%	115.2k	120.1s	87.6k → 135.6k (×1.55)
DeepSeek V4.1 Flash	\$0.005	25.6%	157.8k	115.3s	112.0k → 185.4k (×1.66)
GLM-5.3 Flash	\$0.008	28.2%	158.2k	197.9s	112.6k → 217.8k (×1.93)
Grok 4.7	\$0.088	22.2%	80.0k	117.9s	40.0k → 81.1k (×2.03)
Kimi K3.1 Pro	\$0.101	14.5%	74.8k	217.0s	59.1k → 94.5k (×1.60)

all 117 tasks, leaving GPT-6.1 Sol at 58.1% Pass^3 as the unconditional end-to-end reliability peak alongside sub-cent anchor Hy4 Flash.

Asymmetric Compute Friction Across Exposure Regimes. Comparing matched task pairs between baseline and target sandboxes in Figure 7b reveals a revealing asymmetry between exposure regimes. On interactive-exposed shifts, iterative error-driven debugging inflates nine-model mean token usage ×1.65 from 91.8k to 152.0k, surging ×4.30 on Gemini 4 Argon from 44.8k to 192.3k, and extends wall-clock latency by 39% from 165.1s to 229.2s. On deployment-latent shifts across all 58 tasks, by contrast, token growth is muted at ×1.30, rising from 74.4k to 97.0k with a 25% latency increase from 132.3s to 164.8s—not because latent shifts are easier, but because agents see no error on sample inputs and exit early without investing compute in adaptation. We turn next to command-level trajectory diagnostics in Section 5 to test this mechanism directly and address RQ3 and RQ4.

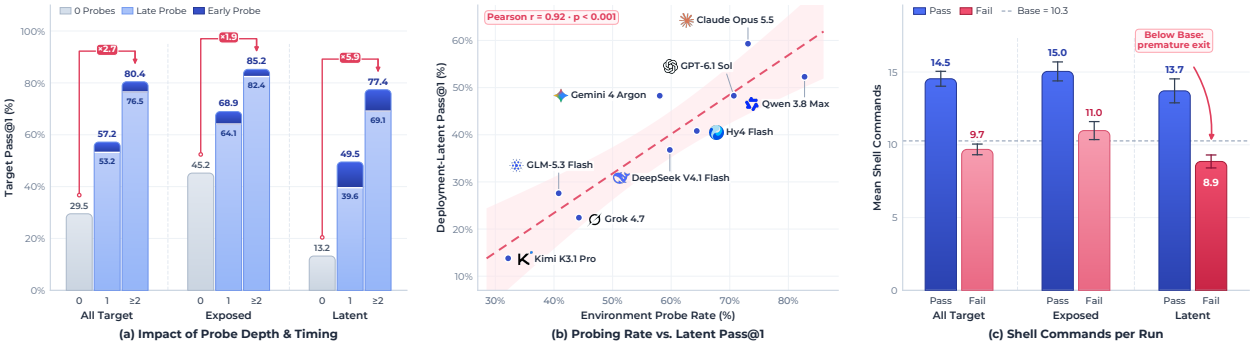


Figure 8: **Trajectory-Level Reconnaissance and Effort Diagnostics across Nine Frontier Models.** Panel a stratifies Target Pass@1 by probe depth across 0, 1, and ≥2 probes and by timing across late steps ≥ 3 versus early steps 1–2, showing a ×5.9 stratified solve-rate gap on deployment-latent tasks between unprobed runs at 13.2% and upfront multi-step runs at 77.4%. Panel b shows the cross-model correlation between host probe rate and Deployment-Latent Pass@1 at $r = 0.92$ with $p < 0.001$. Panel c reports mean shell commands per run with 95% confidence intervals, exposing premature blind exit on Latent Fail runs at 8.9 versus 10.3 on Base.

5 Trajectory-Level Behavioral and Causal Diagnostics

Aggregate leaderboard drops under RQ1 and RQ2 show that frontier agents falter outside greenhouse containers, especially under deployment-latent shifts, yet aggregate scores alone do not explain why. In this section, we analyze all 6,235 non-refusal main-benchmark trajectories and 1,384 prompt-intervention trajectories at the individual shell-command level to address RQ3 and RQ4 alongside two supporting behavioral diagnostics: first, we answer RQ3 in Section 5.1 by quantifying how probe depth and upfront timing correlate with survival while unprobed runs fail through premature blind exit; second, we examine small-budget test-time resampling limits and probing stability in Section 5.2; third, we dissect probed failure modes and the *probe-to-action* reasoning gap across model archetypes in Section 5.3; and fourth, we answer RQ4 in Section 5.4 via a controlled three-arm task-agnostic system-prompt intervention.

5.1 RQ3: Probe Depth, Upfront Timing, and Premature Blind Exit

Syntax-Aware Command-Level Probe Extraction. Measuring host reconnaissance accurately requires separating genuine system inspection commands from ordinary application code that happens to contain similar words, such as a data column named id or a table variable named df. We therefore parse every executed shell tool call with a syntax-aware command analyzer that first strips file-editing patch blocks and heredoc payloads, splits compound shell pipelines and subshells into atomic commands, and unwraps execution prefixes such as sudo, env, or timeout. An atomic command is classified as an environment probe $a_t \in \mathcal{A}_{\text{probe}}$ if and only if it performs one of four read-only host checks: invoking a system-inspection utility such as ulimit; reading system configuration or status files such as `/etc/cron.d`; printing host environment variables such as TZ; or running a read-only one-liner to query encoding, resource limits, or permissions, with full classification rules detailed in Appendix D.1. Each trajectory is annotated with total shell commands n_{bash}, probe count n_{probe}, and the one-indexed step of its first probe t_{first}, distinguishing upfront early probes in steps 1–2 where $t_{\text{first}} \in \{1, 2\}$ from late probes at step 3 or later.

Stratified Solve Rates by Probe Timing and Depth. Figure 8a reveals a sharp contrast between exposure regimes when agents omit host reconnaissance at $n_{\text{probe}} = 0$. On interactive-exposed tasks, unprobed runs still pass 45.2% of the time because sample-time error messages directly reveal the environmental mismatch without requiring dedicated inspection commands. On deployment-latent tasks, however, unprobed runs account for 41.8% of all executed trials and record just 13.2% Pass@1, succeeding almost exclusively when defensive coding habits happen to sidestep the shift. Conditioning on probe depth and timing reveals a steep observational gradient: trajectories with a single probe record 39.6% latent Pass@1 when triggered at step 3 or later and 49.5% when issued in steps 1–2, whereas trajectories executing two or more probes reach 69.1% when triggered late and 77.4% when performed upfront in steps 1–2—a 5.9-fold gap over unprobed runs that approaches the 85.2% rate of early multi-step probing on interactive-exposed

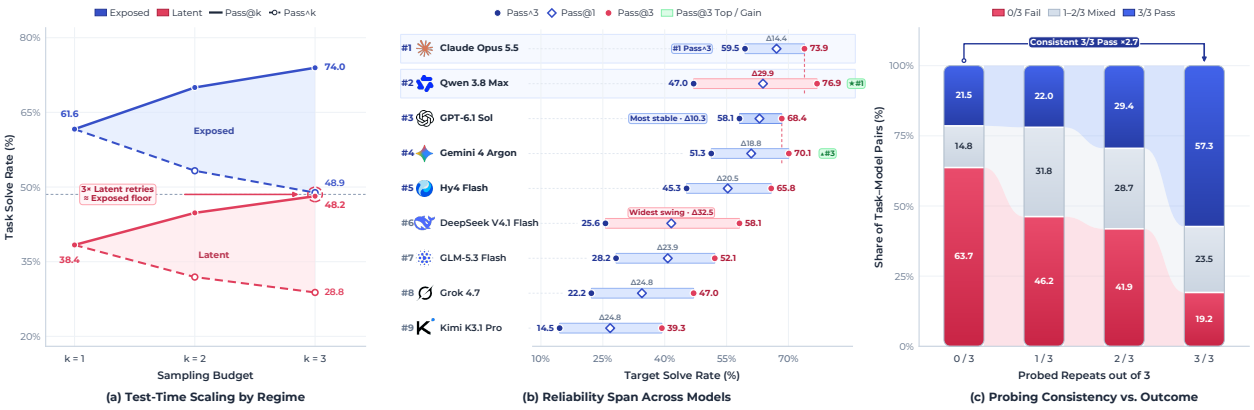


Figure 9: **Multi-Run Scaling, Reliability Spans, and Probing Consistency across Nine Models.** Panel a compares optimistic Pass@k against strict Pass^k reliability across sampling budgets $k = 1..3$, showing that three-try deployment-latent resampling at 48.2% still falls short of the strict worst-case Pass^3 floor of interactive-exposed tasks at 48.9%. Panel b displays per-model Pass^3 $\leftrightarrow$ Pass@1 $\leftrightarrow$ Pass@3 reliability spans and rank overtakes. Panel c shows task outcome stability stratified by probing consistency across three trials.

tasks. Across all nine frontier models in Figure 8b, a model's unconditional latent probe rate strongly correlates with its Deployment-Latent Pass@1, reaching Pearson $r = 0.92$ with $p = 0.0004$ and Spearman $\rho = 0.90$ with $p = 0.0011$.

Command-Budget Evidence of Premature Blind Exit. Why do unprobed latent runs fail so consistently? Figure 8c compares mean shell commands executed per run across outcome groups. Across all 117 baseline containers, agents average 10.3 shell commands to inspect the repository, edit code, and run sample checks. Successful adaptation in target sandboxes requires 42% more shell interaction, averaging 14.5 commands overall with 15.0 on exposed passes and 13.7 on latent passes. In sharp contrast, failed latent runs execute an average of only 8.9 shell commands—fewer than the 10.3 baseline average. Rather than getting stuck in long debugging loops, failing agents apply default-assuming edits, observe exit code 0 on the small sample inputs, and submit immediately without suspecting that a background cron environment, mount boundary, or RLIMIT_FSIZE quota has been violated, as illustrated in Case Box D.1 of Appendix D.

> **Answer to RQ3:** Proactive host reconnaissance strongly correlates with deployment-latent survival at Pearson $r = 0.92$ with $p < 0.001$ across all nine models: unprobed trajectories pass only 13.2% of the time, whereas trajectories executing two or more probes within steps 1–2 reach 77.4% Pass@1, nearly six times the unprobed rate. Failed latent runs execute fewer commands than baseline runs at 8.9 versus 10.3, confirming premature blind exit after clean sample runs rather than protracted debugging failure.

5.2 Small-Budget Resampling Limits and Probing Stability

Recent work on inference-time scaling [35, 36] shows that repeated stochastic sampling Pass@k substantially improves solve rates on competition math and static code synthesis. Can small-budget blind test-time resampling likewise rescue agents from latent environment shifts?

Limited Gains from Small-Budget Blind Resampling. Figure 9a tracks nine-model mean Pass@k and Pass^k across small-budget sampling $k \in \{1, 2, 3\}$. Increasing k from 1 to 3 lifts interactive-exposed Pass@k from 61.6% to 74.0%, whereas deployment-latent Pass@3 reaches 48.2%—still slightly below the 48.9% strict three-of-three floor Pass^3 of interactive-exposed tasks, while deployment-latent Pass^3 drops from 38.4% to 28.8%. Under a $K = 3$ unguided budget, when a policy lacks the habit of checking scheduled job configurations or active resource limits, independent temperature samples tend to repeat the same greenhouse assumption rather than spontaneously initiating host reconnaissance.

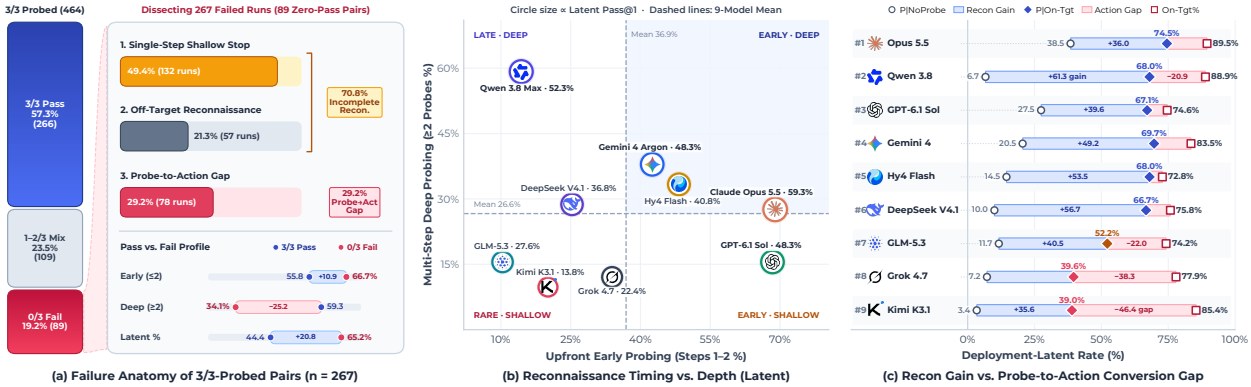


Figure 10: **Probe Quality, Reconnaissance Timing-by-Depth Quadrants, and *probe-to-action* Conversion on Deployment-Latent Tasks.** Panel a partitions the 267 failed trajectories across 89 consistently probed zero-pass pairs into 49.4% single-step shallow stops, 21.3% off-target checks, and 29.2% *probe-to-action* reasoning failures, alongside their early-probe, multi-step depth, and latent-task shifts relative to three-of-three passing pairs. Panel b positions all nine frontier models across four timing-by-depth quadrants around the nine-model mean crosshair at 36.9% early and 26.6% deep probing. Panel c contrasts unprobed survival, conditional on-target solve rate, and on-target hit precision, highlighting Kimi K3.1 Pro's 46.4-point conversion loss despite 85.4% on-target hits.

Model Reliability Spans and Probing Consistency. Comparing per-model Pass^3 ↔ Pass@1 ↔ Pass@3 intervals in Figure 9b highlights sharp differences between high-variance explorers and consistent production performers. Qwen 3.8 Max overtakes Claude Opus 5.5 on optimistic Pass@3, claiming first place at 76.9% versus 73.9%, yet drops to fourth on strict Pass^3 at 47.0% across a 29.9-point span, ahead of Hy4 Flash fifth on both metrics at 65.8% and 45.3% across a 20.5-point span; DeepSeek V4.1 Flash exhibits the widest retry swing at 32.5 points from 25.6% to 58.1%, and Gemini 4 Argon edges past GPT-6.1 Sol on Pass@3 at 70.1% versus 68.4%. Conversely, GPT-6.1 Sol records the narrowest reliability span at 10.3 points from 58.1% to 68.4%, followed on non-refusal runs by Claude Opus 5.5 with a 14.4-point span at 59.5% Pass^3 and 67.1% Pass@1, whereas scoring refusals as zero places GPT-6.1 Sol first on unconditional Pass^3 at 58.1% ahead of Claude Opus 5.5 at 47.9%. Figure 9c shows how strongly probing habits align with run-to-run stability across three-trial task–model pairs: three-of-three pass rates stand at 57.3% among the 464 consistently probed pairs versus 21.5% among the 303 unprobed pairs, a 2.7-fold difference, while zero-of-three failure rates stand at 19.2% versus 63.7%.

5.3 When Probing Is Not Enough: Probe Quality and the *probe-to-action* Gap

Two important puzzles remain in Figures 8 and 9. First, why do 19.2% of consistently probed task–model pairs, 89 of 464 pairs spanning 267 runs, still fail across all three trials? Second, why does Qwen 3.8 Max trail Claude Opus 5.5 on non-refusal Deployment-Latent Pass@1 at 52.3% versus 59.3% and on Pass^3 at 47.0% versus 59.5% despite recording the highest latent probe rate at 82.8%, while Gemini 4 Argon ties for third with a moderate 58.0% probe rate? To answer both questions, we match each run's inspection commands against the shifted mechanism of each task under the deterministic attribution protocol in Appendix D.1 to measure on-target precision and conditional solve rates across all nine models in Figure 10 and Table 8.

Failure Modes of Consistently Probed Trajectories. Inspecting the 267 failed runs across the 89 consistently probed yet zero-pass pairs in Figure 10a rules out lack of effort: 66.7% initiate probing within the first two steps, exceeding the 55.8% early-probe rate of three-of-three passing pairs, and they average 13.1 shell commands per run, well above the 10.3 baseline average. However, they execute only 1.52 probes per run, reaching two or more probes in just 34.1% of runs compared to 59.3% among consistently passing pairs, and separate into three distinct failure modes.

Nearly half of these failures, accounting for 49.4% or 132 of the 267 failed runs, stem from a single-step shallow probing illusion: the agent checks the relevant surface once in the interactive shell and stops too early—most notably querying the login shell's TZ variable and observing Asia/Tokyo at UTC+9 without inspecting `/etc/cron.d/nightly-job`, where the scheduled production job overrides TZ to fractional Asia/Kolkata at UTC+05:30, as shown in Case Box D.2.

Table 8: **Reconnaissance behavior and conditional solve rates across nine frontier models on deployment-latent tasks.** Metrics are percentages computed over N non-refusal executed runs out of 174 scheduled trials across 58 tasks and 3 repeats, tracking overall probing, upfront probing in steps 1–2, multi-step probing, on-target precision, and solve rates conditioned on probing behavior.

Model	N	Pass@1	Probe%	Early%	Deep≥2%	On-Tgt%	P|Probe	P|On-Tgt	P|NoProbe
Claude Opus 5.5	145	**59.3**	73.1	**69.0**	27.6	**89.5**	67.0	**74.5**	**38.5**
Qwen 3.8 Max	174	52.3	**82.8**	14.4	**59.2**	88.9	61.8	68.0	6.7
GPT-6.1 Sol	174	48.3	70.7	68.4	15.5	74.6	56.9	67.1	27.5
Gemini 4 Argon	174	48.3	58.0	42.5	37.9	83.5	**68.3**	69.7	20.5
Hy4 Flash	174	40.8	64.4	48.3	33.3	72.8	55.4	68.0	14.5
DeepSeek V4.1 Flash	174	36.8	59.8	25.3	28.7	75.8	54.8	66.7	10.0
GLM-5.3 Flash	174	27.6	40.8	10.3	15.5	74.2	50.7	52.2	11.7
Grok 4.7	174	22.4	44.3	33.9	12.1	77.9	41.6	39.6	7.2
Kimi K3.1 Pro	174	13.8	32.2	20.1	9.8	85.4	35.7	39.0	3.4

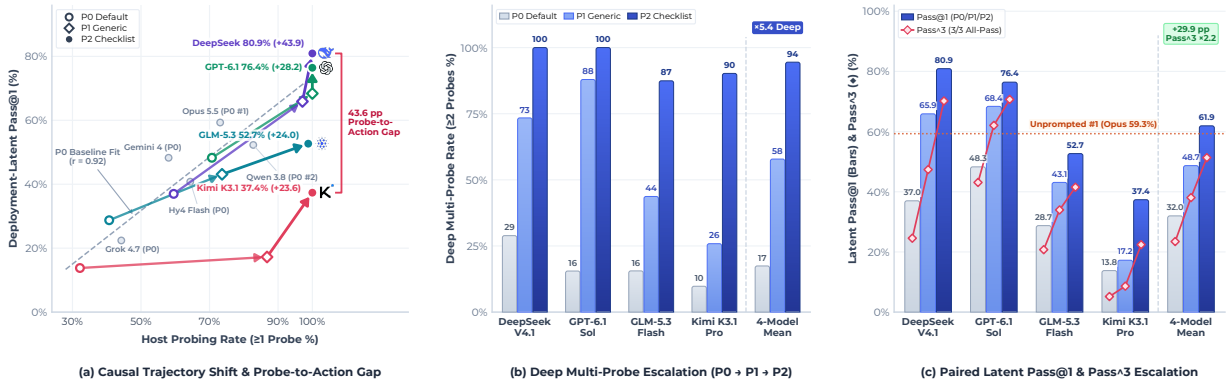
Another 21.3%, or 57 runs, reflect off-target reconnaissance, where agents issue generic identity or directory checks unrelated to the active filesystem, network, or quota constraint; together with single-step stops, 70.8% or 189 of the 267 probed failures arise from incomplete reconnaissance.

Most revealingly, the remaining 29.2% of failed runs, 78 trajectories in total, execute two or more on-target probes yet still fail to translate observed host state into boundary-safe code, exposing a *probe-to-action* reasoning gap alongside subtle granularity blind spots. In Case Box D.3, eight of the nine models fail all three trials despite explicitly reading the Europe/Berlin timezone setting: none of the eight deduce that pandas or pytz timestamp localization raises AmbiguousTimeError under default settings during the repeated autumn DST fallback hour, whereas only Qwen 3.8 Max explicitly disambiguates fold transitions in two of three runs. Similarly, in Case Box D.4, agents confirm root identity at uid=0 yet overlook dropped CAP_DAC_OVERRIDE privileges in the CapEff bitmask of `/proc/self/status`, triggering PermissionError when opening read-only 0444 foreign-owned files in place instead of writing a temporary sibling file and committing via atomic rename. Across all deployment-latent tasks in Figure 10c and Table 8, Kimi K3.1 Pro inspects the relevant axis in 85.4% of its probed runs yet solves only 39.0% of those on-target runs, compared to 74.5% for Claude Opus 5.5.

Five Reconnaissance Archetypes Across Frontier Models. Figure 10b and Table 8 show that current frontier models fall into five distinct behavioral patterns. Claude Opus 5.5 exemplifies upfront precision on its non-refusal runs, leading all models in early probing at 69.0%, on-target precision at 89.5%, and on-target solve conversion at 74.5%. Qwen 3.8 Max exemplifies exhaustive late exploration, pairing benchmark-high overall probing at 82.8% and multi-step probing at 59.2% to capture second place on non-refusal latent tasks and first place on Pass@3, though its low 14.4% early-probe share limits three-run Pass^3 consistency. Gemini 4 Argon and Hy4 Flash occupy the early-and-deep quadrant alongside Claude Opus 5.5, yet diverge in targeting precision: Gemini 4 Argon pairs a selective 58.0% probe rate and 37.9% multi-step rate with 83.5% on-target precision to achieve the highest unconditional solve rate given probing at 68.3%, whereas Hy4 Flash probes upfront at 48.3% and multi-step at 33.3% across 64.4% of runs, converting 68.0% when on target while lower 72.8% on-target precision moderates unconditional conversion to 55.4%, similar to DeepSeek V4.1 Flash at 59.8% probing and 66.7% on-target conversion. GPT-6.1 Sol exemplifies upfront shallow checking, probing early on 68.4% of runs but issuing two or more probes on only 15.5%, leaving multi-hop scheduler and privilege shifts unresolved. Finally, GLM-5.3 Flash, Grok 4.7, and Kimi K3.1 Pro exhibit rare, low-conversion probing, combining low probe rates of 32.2%–44.3% with modest conditional solve rates of 35.7%–50.7%.

5.4 RQ4: Causal Reconnaissance Prompting and the *probe-to-action* Gap

Observational correlation at $r = 0.92$ leaves open a natural confounding question: stronger models might simply happen both to probe more often and to write better code, without reconnaissance itself acting as a causal driver. Conversely, if reconnaissance is causal, can system-prompt guidance alone solve deployment-latent adaptation?



(a) Causal Trajectory Shift & Probe-to-Action Gap (b) Deep Multi-Probe Escalation (P0 → P1 → P2) (c) Paired Latent Pass@1 & Pass^3 Escalation

Figure 11: **Causal Intervention via Explicit Reconnaissance Prompting Across $n = 688$ Strictly Paired Runs.** Panel a plots causal trajectory shifts from P0 to P1 and P2 on the probe-rate versus Latent Pass@1 plane overlaid on the nine-model unprompted P0 regression line at $r = 0.92$, showing both above-trend unlocks for DeepSeek V4.1 Flash at 80.9% and GPT-6.1 Sol at 76.4% and the 43.6 percentage-point vertical *probe-to-action* gap near 100% probing. Panel b shows the ×5.4 surge in multi-step deep probing. Panel c shows paired Pass@1 and Pass^3 gains across models.

Table 9: **Causal system-prompt intervention across all 58 deployment-latent tasks.** Outcome and probing columns report percentages across P0 default, P1 generic reminder, and P2 structured checklist over $n = 688$ strictly slot-paired runs spanning 226 complete three-trial model–task pairs. The bottom row verifies robustness when counting the 7 single-turn 6,000-token output truncations as failures across $n = 695$ runs.

Model	Paired n	Pass@1 (P0→P1→P2)	Δ(P2−P0)	Pass^3 (P0→P1→P2)	Probe% (P0→P1→P2)	Deep≥2% (P0→P1→P2)
DeepSeek V4.1 Flash	173 (57)	37.0 → 65.9 → **80.9**	**+43.9**	24.6 → 47.4 → 70.2	59.5 → 97.1 → 100.0	28.9 → 73.4 → 100.0
GPT-6.1 Sol	174 (58)	48.3 → 68.4 → **76.4**	**+28.2**	43.1 → 62.1 → 70.7	70.7 → 100.0 → 100.0	15.5 → 87.9 → 100.0
GLM-5.3 Flash	167 (53)	28.7 → 43.1 → **52.7**	**+23.9**	20.8 → 34.0 → 41.5	40.7 → 73.7 → 98.8	15.6 → 43.7 → 87.4
Kimi K3.1 Pro	174 (58)	13.8 → 17.2 → **37.4**	**+23.6**	5.2 → 8.6 → 22.4	32.2 → 86.8 → 100.0	9.8 → 25.9 → 90.2
4-Model Paired Total	**688 (226)**	**32.0 → 48.7 → 61.9**	**+29.9**	**23.4 → 38.0 → 51.3**	**50.9 → 89.5 → 99.7**	**17.4 → 57.9 → 94.5**
Count-Truncation-as-0 ($n = 695$)	*695 (231)*	*31.6 → 48.6 → 61.4*	*+29.8*	*22.9 → 37.2 → 50.2*	—	—

Three-Arm Task-Agnostic Intervention Design. We perform a controlled system-prompt intervention across all 58 deployment-latent tasks with $K = 3$ trials per arm on four models spanning our behavioral archetypes: high-ceiling reasoner DeepSeek V4.1 Flash, upfront-shallow flagship GPT-6.1 Sol, mid-tier GLM-5.3 Flash, and low-conversion Kimi K3.1 Pro. Holding model snapshots, sampling temperatures, step limits, and token budgets strictly fixed, we compare three arms by appending identical task-agnostic suffixes to each runner's native system prompt while preserving termination protocols across all four runtime profiles; verbatim prompts appear in Appendix E:

- **Arm P0, Default Baseline:** Unmodified benchmark system prompt.
- **Arm P1, Generic Reminder:** Appends a concise two-sentence engineering guideline instructing the agent not to assume default container settings or rely solely on clean runs over small sample inputs, and to inspect the host environment and invocation context before finalizing code.
- **Arm P2, Taxonomy-Aligned Pre-Flight Checklist:** Appends a task-agnostic, taxonomy-aligned five-point diagnostic checklist covering filesystem mounts and case or Unicode semantics, runtime and CLI userland versions, POSIX locale, TZ, and cron schedules, effective UID, Linux capabilities, umask, and rlimit or cgroup quotas, and workspace lockfiles or service health, applied identically across all tasks without per-task hints.

Of 1,392 scheduled new trials across P1 and P2, 1,384 reached terminal verdicts after excluding 1 transient gateway drop and 7 single-turn 6,000-token output truncations, yielding $n = 688$ strictly slot-paired runs across all three arms over 226 complete three-trial task–model pairs. Figure 11 and Table 9 report the results.

Causal Leverage and Latent Capability Unlocks. Across the four-model paired cohort, progressing from P0 to P1 and P2 raises host probing from 50.9% to 89.5% and 99.7%, while expanding multi-step probing ×5.4 from 17.4% to 57.9% and 94.5% in Figure 11b. This behavioral shift lifts mean Deployment-Latent Pass@1 by 16.7 percentage points under P1, from 32.0% to 48.7%, and by 29.9 percentage points under P2 to 61.9%. Structured P2 outperforms generic P1 by 13.2 percentage points, with a 95% task-level bootstrap confidence interval [37] of [+6.7, +20.1] points and 130 P2-only wins against 39 P1-only wins under paired McNemar comparison [38]. At the same time, strict three-run reliability Pass^3 more than doubles by ×2.19 from 23.4% to 51.3%, and counting the 7 output truncations as failures across $n = 695$ runs in the bottom row of Table 9 yields virtually identical gains from 31.6% to 48.6% and 61.4%.

Crucially, removing the shallow-probing bottleneck diagnosed in Section 5.3 unlocks substantial latent capability in strong coding models, as seen in Figure 11a and Figure 11c. DeepSeek V4.1 Flash leaps 43.9 percentage points from 37.0% to 80.9% with 70.2% Pass^3, while GPT-6.1 Sol, once its multi-step probing rate rises from 15.5% to 100.0%, jumps 28.2 percentage points from 48.3% to 76.4% with 70.7% Pass^3 while averaging just 5.8 shell commands per run compared to 4.3 in P0. Both models decisively surpass unprompted non-refusal leader Claude Opus 5.5 at 59.3%, establishing upfront multi-step reconnaissance as a direct causal lever rather than a byproduct of general model scale.

The 43.6 Percentage-Point Cross-Model *probe-to-action* Gap. Equally informative is what prompt engineering leaves unresolved. On Kimi K3.1 Pro, generic reminder P1 raises probing from 32.2% to 86.8%, yet Pass@1 improves by only 3.5 percentage points from 13.8% to 17.2%, and solve conversion among probed runs drops from 35.7% to 18.5% because newly triggered checks remain superficial. Even after structured P2 drives mean multi-step probing to 94.5% and pushes overall probing to 98.8%–100.0% across all four tested models, lifting Kimi K3.1 Pro to 37.4% and GLM-5.3 Flash to 52.7%, a 43.6 percentage-point cross-model conversion spread remains between Kimi K3.1 Pro at 37.4% and DeepSeek V4.1 Flash at 80.9% along the right edge of Figure 11a. Because Kimi K3.1 Pro already reaches only 39.5% on error-surfacing interactive-exposed tasks where no reconnaissance is needed, this residual spread reflects underlying disparities in system-state deduction and boundary-safe coding that pre-flight checklists alone cannot equalize.

> **Answer to RQ4:** Task-agnostic, taxonomy-aligned pre-flight prompting causally drives multi-step probing from 17.4% to 94.5%, lifting paired Deployment-Latent Pass@1 by 29.9 percentage points from 32.0% to 61.9%, more than doubling Pass^3 reliability, and propelling DeepSeek V4.1 Flash to 80.9% and GPT-6.1 Sol to 76.4%. Even near 100% probing, however, a 43.6 percentage-point cross-model *probe-to-action* gap persists across the four tested models, indicating that checklist guidance resolves reconnaissance blindness without equalizing underlying system-level reasoning capability.

6 Related Work

6.1 Repository-Scale Coding Benchmarks and Agent RL Environments

Executable code evaluation has evolved from standalone function synthesis [15, 16, 39] to repository-scale issue resolution in SWE-bench [1, 5] and its dynamic or multilingual extensions SWE-bench Live [8] and Multi-SWE-bench [40], alongside specialized domains in MLE-bench [41] and Cybench [42]. In parallel, agent scaffolds such as SWE-agent [2], Agentless [43], and OpenHands [3], together with automated environment-setup benchmarks such as EnvBench [44] and Repo2Run [45] and reinforcement-learning rollout environments such as SWE-Gym [9] and R2E-Gym [10], have established a tight feedback loop between containerized training and repository evaluation. Yet this ecosystem shares a common structural assumption: both RL rollout environments and evaluation harnesses instantiate pre-configured greenhouse containers engineered so existing test suites run cleanly under permissive defaults. Treating the host environment as invariant infrastructure rather than a controlled experimental variable induces strong train–test coupling, rewarding rapid local test-patching while leaving environmental adaptability unmeasured. TwinEnv-Bench breaks this coupling by holding task logic invariant across paired baseline and shifted sandboxes.

6.2 Operating-System Sandboxes, Environment Drift, and Evaluation Hygiene

A second thread studies interactive agents across operating systems, web browsers, and multi-tool APIs, alongside software-engineering research on configuration drift. Early execution suites such as InterCode [6] and AgentBench [46]

introduced multi-turn shell and SQL loops; WebArena [47], OSWorld [11], Windows Agent Arena [48], and AndroidWorld [49] evaluate GUI navigation across desktop and mobile platforms; Terminal-Bench [4] curates demanding command-line workflows; and GAIA [50], AppWorld [7], τ-bench [18], MINT [51], ToolLLM [52], and ToolEmu [53] examine multi-turn tool orchestration and safety boundaries. Empirical reproducibility studies in software engineering [19] have long shown that real-world script failures frequently stem from undeclared locale, path, dependency, and permission drift rather than algorithmic bugs, yet existing OS and CLI agent benchmarks evaluate each task in a single static machine state without counterfactual controls or single-variable verification. Moreover, methodological critiques of agent evaluation [54] warn that container-resident test scripts invite subtle oracle leakage and conflate upstream API refusals with model errors—pitfalls TwinEnv-Bench eliminates via post-termination host-isolated grading and explicit non-refusal accounting.

6.3 Partial Observability, Active Perception, and Inference-Time Reliability

Acting inside an unfamiliar computing host exemplifies sequential decision-making under partial observability [17], where an agent must balance pragmatic task-completion edits against epistemic information-gathering probes, or active reconnaissance. Prevailing agentic frameworks such as ReAct [12], Reflexion [13], and Toolformer [55] incorporate environment observations primarily in a reactive loop triggered after tool errors or failed assertions. Likewise, recent work on inference-time compute scaling [30, 35, 36] focuses on repeated temperature sampling Pass@k or internal chain-of-thought search over fully specified prompts, whereas deployment-oriented studies [18] highlight steep drops under strict multi-run reliability Pass^k. Our command-level trajectory analysis and three-arm prompt intervention connect these perspectives: we show that reactive loops and small-budget $K = 3$ blind Pass@k retries yield limited gains when deployment constraints remain silent on sample inputs, that task-agnostic pre-flight reconnaissance causally doubles Pass^3 reliability, and that a 43.6 percentage-point cross-model *probe-to-action* spread persists even under near-saturated probing because prompt checklists alone do not equalize system-level state-to-code reasoning.

7 Discussion, Limitations, and Conclusion

7.1 Implications for Agent Post-Training and Harness Design

Our findings carry direct architectural lessons for both foundation-model post-training and production agent harnesses.

Breaking Greenhouse Priors in RL Post-Training. Current reinforcement-learning pipelines for coding agents [9, 10, 30] train almost exclusively inside uniform UTF-8, root, GNU Linux containers where immediate unit-test execution serves as a complete oracle. Consequently, policies learn an aggressive inductive bias: edit code rapidly against sample fixtures and terminate as soon as local tests return exit code 0. Training natively proactive agents requires *counterfactual environment randomization* during RL rollouts—exposing policies to varied locales, cron contexts, mount flags, non-root UIDs, and kernel rlimits where premature unprobed submission incurs negative reward, thereby internalizing lightweight upfront reconnaissance as an intrinsic habit rather than an external prompt crutch.

Closing the *probe-to-action* Gap via State-Conditional Synthesis. Our causal intervention in Section 5.4 shows that instructing an agent where to inspect via P2 saturates probing near 100% and unlocks strong reasoning models, lifting DeepSeek V4.1 Flash to 80.9% and GPT-6.1 Sol to 76.4%, yet leaves a wide cross-model spread down to 37.4%–52.7% and roughly a twenty-point gap below baseline ceiling even on the strongest tested models. These residual gaps motivate post-training curricula targeted at *state-to-boundary deduction*: reasoning from TZ=Europe/Berlin to DST AmbiguousTimeError disambiguation, from CapEff bitmasks in `/proc/self/status` to atomic sibling-file replacement, and from RLIMIT_FSIZE quotas to streaming bounded-chunk writers.

Adaptive vs. Exhaustive Reconnaissance in Agent Harnesses. Comparing Claude Opus 5.5 at \$0.073 per run, 20.0k tokens, and 59.5% non-refusal Pass^3, or GPT-6.1 Sol under P2 at 5.8 shell commands and 70.7% Pass^3, against unprompted Qwen 3.8 Max at \$0.237 per run, 358.8k tokens, and 47.0% Pass^3 highlights an important efficiency principle: high production reliability does not require brute-force shell transcripts. Compact, targeted pre-flight probes executed in steps 1–2 achieve both higher Pass^3 consistency and 3×–18× lower token cost than reactive late-stage exploration. Production agent harnesses can embed lightweight, read-only host-context summaries or learned selective probing routers at session startup while guarding against prompt-induced verbosity under tight per-turn output caps.

7.2 Limitations and Future Directions

While TwinEnv-Bench establishes a rigorous counterfactual foundation for evaluating environment adaptation, several boundaries motivate future extensions. First, to guarantee deterministic single-variable control and self-contained reproducibility via Docker and standard Python libraries on Linux hosts, TwinEnv-Bench focuses on controlled micro-scale POSIX scripting, systems, and data-engineering tasks across E1–E5, with a median reference implementation of 22 lines; extending counterfactual twin-cell isolation to multi-thousand-line software repositories as well as native macOS, Windows, and multi-node distributed service topologies is a natural next step.

Second, by design, every task in TwinEnv-Bench shifts a single verified mechanism Δd_i to enable unambiguous causal attribution, whereas complex production migrations occasionally couple multiple simultaneous shifts, such as running a scheduled script inside a BusyBox container behind an egress proxy under an unprivileged UID; our modular 55-sandbox suite provides direct building blocks for studying compositional k-variable interaction effects. Finally, our evaluation standardizes on a unified single-agent tool harness across $K = 3$ independent repeats per cell so models can be compared apples-to-apples; investigating how multi-agent verifier scaffolds or larger test-time sampling budgets interact with silent host constraints remains a promising avenue for future work.

7.3 Conclusion

We introduced TwinEnv-Bench, a controlled micro-scale counterfactual benchmark of 117 scripting, systems, and data-engineering tasks across 234 paired twin cells spanning a five-layer, 15-group, 31-mechanism environment taxonomy designed to evaluate agents beyond homogeneous greenhouse sandboxes. Enforced by 55 probe-verified container configurations, a deterministic six-run three-state solvability contract, and a leak-proof host-isolated runner, TwinEnv-Bench cleanly separates algorithmic coding skill from real-world operational adaptability. Evaluating nine frontier models across 6,235 non-refusal main-benchmark runs and 1,384 causal intervention runs reveals that near-saturated greenhouse competence masks 31.3–72.3 percentage-point collapses under single-variable shifts, driven by reliance on exposed error signals or upfront shallow checks alongside premature blind exit on silent deployment-latent constraints. While task-agnostic, taxonomy-aligned pre-flight reconnaissance prompting causally lifts latent solve rates by 29.9 percentage points and more than doubles multi-run reliability, a 43.6 percentage-point cross-model *probe-to-action* gap remains across the four tested models even near 100% probing, showing that checklist guidance alone does not equalize system-level state-to-code reasoning across models.

References

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

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

[3] 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 *International Conference on Learning Representations*, 2025.

[4] Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, 2026.

[5] Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, et al. Introducing SWE-bench Verified. Technical report, OpenAI, 2024.

[6] John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. InterCode: Standardizing and benchmarking interactive coding with execution feedback. In *Advances in Neural Information Processing Systems*, volume 36, pages 23826–23854, 2023.

[7] Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 16022–16076, 2024.

[8] Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, et al. SWE-bench goes live! *arXiv preprint arXiv:2505.23419*, 2025.

[9] Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. In *International Conference on Machine Learning*, pages 47717–47737. PMLR, 2025.

[10] Naman Jain, Jaskirat Singh, Manish Shetty, Tianjun Zhang, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural environment generation and hybrid verifiers for scaling open-weights SWE agents. In *Second Conference on Language Modeling*, 2025.

[11] Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems*, volume 37, pages 52040–52094, 2024.

[12] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

[13] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pages 8634–8652, 2023.

[14] Judea Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, 2nd edition, 2009.

[15] 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.

[16] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, 2025.

[17] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1–2):99–134, 1998.

[18] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *International Conference on Learning Representations*, 2025.

[19] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. A large-scale study about quality and reproducibility of Jupyter notebooks. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 507–517. IEEE, 2019.

[20] Anthropic. The Claude 3 model family: Opus, Sonnet, Haiku. Technical report, Anthropic, 2024.

[21] Anthropic. System card: Claude Opus 4 & Claude Sonnet 4. Technical report, Anthropic, 2025.

[22] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[23] Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. GPT-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

[24] Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. OpenAI o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

[25] OpenAI. GPT-5 system card. Technical report, OpenAI, 2025.

[26] Gheorghe Comanici, Eric J. Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit S. Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

[27] Xingwu Sun, Yanfeng Chen, Yiqing Huang, Ruobing Xie, Jiaqi Zhu, Kai Zhang, Shuaipeng Li, Zhen Yang, Jonny Han, Xiaobo Shu, et al. Hunyuan-Large: An open-source MoE model with 52 billion activated parameters by Tencent. *arXiv preprint arXiv:2411.02265*, 2024.

[28] Ao Liu, Botong Zhou, Can Xu, Chayse Zhou, Chenchen Zhang, Chengcheng Xu, Chenhao Wang, Decheng Wu, Dengpeng Wu, Dian Jiao, et al. Hunyuan-TurboS: Advancing large language models through Mamba-Transformer synergy and adaptive chain-of-thought. *arXiv preprint arXiv:2505.15431*, 2025.

[29] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[30] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645(8081):633–638, 2025.

[31] Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, et al. GLM-4.5: Agentic, reasoning, and coding (ARC) foundation models. *arXiv preprint arXiv:2508.06471*, 2025.

[32] xAI. Grok 4 model card. Technical report, xAI, 2025.

[33] Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, Chuning Tang, et al. Kimi k1.5: Scaling reinforcement learning with LLMs. *arXiv preprint arXiv:2501.12599*, 2025.

[34] Yifan Bai, Yiping Bao, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, et al. Kimi K2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

[35] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

[36] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. In *International Conference on Learning Representations*, 2025.

[37] Bradley Efron. Bootstrap methods: Another look at the jackknife. *The Annals of Statistics*, 7(1):1–26, 1979.

[38] Quinn McNemar. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2):153–157, 1947.

[39] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations*, 2025.

[40] Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, et al. Multi-SWE-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605*, 2025.

[41] Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, et al. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *International Conference on Learning Representations*, 2025.

[42] Andy K. Zhang, Neil Perry, Riya Dulepet, Joey Ji, Celeste Menders, Justin W. Lin, Eliot Jones, Gashon Hussein, Samantha Liu, Donovan Jasper, et al. Cybench: A framework for evaluating cybersecurity capabilities and risks of language models. In *International Conference on Learning Representations*, 2025.

[43] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):808–831, 2025.

[44] Aleksandra Eliseeva, Alexander Kovrigin, Ilia Kholkin, Egor Bogomolov, and Yaroslav Zharov. EnvBench: A benchmark for automated environment setup. *arXiv preprint arXiv:2503.14443*, 2025.

[45] Ruida Hu, Chao Peng, Xinchen Wang, Junjielong Xu, and Cuiyun Gao. Repo2Run: Automated building executable environment for code repository at scale. *arXiv preprint arXiv:2502.13681*, 2025.

[46] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. AgentBench: Evaluating LLMs as agents. In *International Conference on Learning Representations*, 2024.

[47] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. WebArena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, 2024.

[48] Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, et al. Windows Agent Arena: Evaluating multi-modal OS agents at scale. In *International Conference on Machine Learning*, pages 4874–4910. PMLR, 2025.

[49] Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, et al. AndroidWorld: A dynamic benchmarking environment for autonomous agents. In *International Conference on Learning Representations*, 2025.

[50] Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: A benchmark for general AI assistants. In *International Conference on Learning Representations*, 2024.

[51] Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. MINT: Evaluating LLMs in multi-turn interaction with tools and language feedback. In *International Conference on Learning Representations*, 2024.

[52] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *International Conference on Learning Representations*, 2024.

[53] Yangjun Ruan, Honghua Dong, Andrew Wang, Silviu Pitis, Yongchao Zhou, Jimmy Ba, Yann Dubois, Chris J. Maddison, and Tatsunori Hashimoto. Identifying the risks of LM agents with an LM-emulated sandbox. In *International Conference on Learning Representations*, 2024.

[54] Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. AI agents that matter. *Transactions on Machine Learning Research*, 2025.

[55] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36, pages 68539–68551, 2023.

[56] Timnit Gebru, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna Wallach, Hal Daumé III, and Kate Crawford. Datasheets for datasets. *Communications of the ACM*, 64(12):86–92, 2021.

A Complete E1–E5 Taxonomy Registry and Incident Grounding

Table 10 documents all 31 canonical environment mechanisms across the 5 layers E1–E5 and 15 dimension groups E1.1–E5.2 of TwinEnv-Bench. For each mechanism, we report total task count N, exact interactive-exposed and deployment-latent task counts, canonical greenhouse baseline state e^{base}, counterfactual target states e^{target}, and representative open-source agent failure issues or POSIX and runtime specification anchors that motivated its inclusion.

Reproducibility Scope. All 234 task–environment cells across the 31 software, execution-context, security-boundary, and service-state mechanisms below are designed to execute deterministically on any standard Linux host using Docker Engine and the Python 3.8+ standard library without proprietary cloud virtual machines or external infrastructure dependencies.

Table 10: **Complete registry of all 31 canonical environment mechanisms in TwinEnv-Bench.** Columns report total task count N alongside exposed and latent counts, baseline versus counterfactual target configurations, and real-world open-source agent failure issues or system specification anchors.

Mechanism ID	N (E/L)	Baseline (base)	Counterfactual Shift (target)	Incident & Specification Grounding
`E1.1.case`	2 (0/2)	sensitive	insensitive (ext4 casefold)	anthropics/claude-code#90524; POSIX case-folding
`E1.1.file_type`	1 (0/1)	regular file	FIFO named pipe (`mkfifo`)	POSIX FIFO non-seekable stream (`ESPIPE`)
`E1.1.length`	1 (0/1)	ASCII short names	UTF-8 near 255-byte `NAME_MAX`	openai/codex#36768; ext4 `ENAMETOOLONG`
`E1.1.symlink`	4 (3/1)	resolved local dir	cross-fs / relative symlink	openai/codex#43838; `O_NOFOLLOW` / `realpath`
`E1.1.unicode`	2 (0/2)	NFC byte-exact	NFD decomposed filenames	anthropics/claude-code#87822; Unicode UAX #15
`E1.2.line_ending`	2 (1/1)	LF (`\n`)	CRLF (`\r\n` / `autocrlf`)	anthropics/claude-code#56946, anthropics/claude-code#89307
`E2.1.entry_name`	1 (1/0)	`python` + `python3`	`python` alias absent	PEP 394 `python3`-only container distributions
`E2.1.version`	4 (4/0)	Python 3.12	Python 3.8 (legacy) / 3.14 (bleeding)	SWE-agent/SWE-agent#1179; stdlib deprecations
`E2.2.build_tools`	2 (2/0)	`gcc`/`make` present	C/C++ compiler toolchain absent	Slim production runtime images without `build-essential`
`E2.2.package_source`	3 (3/0)	public PyPI	internal mirror / air-gapped wheelhouse	PEP 503 simple repository & offline wheel resolution
`E2.2.preinstalled`	3 (3/0)	full scientific stack	third-party packages absent/partial	All-Hands-AI/OpenHands#12765; stdlib fallback
`E2.3.shell`	4 (4/0)	GNU `bash`	`dash` / `mksh` / `zsh` / `fish` / `pwsh`	POSIX `sh` non-bashism & `SHELL` invocation semantics
`E2.3.toolchain`	5 (5/0)	GNU coreutils/`gawk`	BusyBox / Toybox / BSD / `mawk`	anthropics/claude-code#95705, anthropics/claude-code#79862
`E2.4.browser_engine`	2 (2/0)	Chromium	Firefox / WebKit	Cross-engine CSS/DOM/Playwright rendering divergence
`E3.1.encoding`	5 (1/4)	`UTF-8`	`GBK` / `Big5` / `CP932` / `ISO-8859-9`	anthropics/claude-code#56946, openai/codex#32958
`E3.1.locale`	17 (3/14)	`C.UTF-8`	`de_DE` / `tr_TR` / `lt_LT` / `fr_FR`	openai/codex#21957; `LC_NUMERIC` comma & Turkish I/ı
`E3.1.timezone`	9 (4/5)	`UTC`	DST (`Europe/Berlin`) / `+05:30` (`Asia/Kolkata`)	openai/codex#48436; IANA tzdb DST & half-hour offsets
`E3.2.env_inherit`	5 (4/1)	full interactive env	cron minimal/stripped `PATH` & env	openai/codex#37662; `cron(8)` clean environment
`E3.2.identity`	5 (1/4)	`uid=0 (root)`	unprivileged UID / foreign file owner	`fs.protected_hardlinks`, Git `safe.directory`, non-root DAC
`E3.2.tty`	1 (1/0)	interactive PTY	`no-tty` (`stdin /dev/null`)	anthropics/claude-code#237; non-interactive hang

Mechanism ID	N (E/L)	Baseline (base)	Counterfactual Shift (target)	Incident & Specification Grounding
E3.2.umask	1 (0/1)	0022	restrictive 0077	Group/world-unreadable permissions under umask 0077
E4.1.mount	3 (0/3)	single rw ext4	/tmp noexec / cross-device EXDEV	anthropics/claude-code#26741; os.rename EXDEV
E4.1.permissions	6 (2/4)	full root + rw sys	readonly rootfs / dropped CAP_DAC_OVERRIDE	Linux capabilities(7) & immutable deployment directories
E4.2.dns	3 (3/0)	public DNS	SERVFAIL / split-horizon internal DNS	anthropics/claude-code#72706, openai/codex#16312
E4.2.egress	6 (5/1)	direct internet	offline / allowlist / HTTP proxy sidecar	openai/codex#376; corporate egress proxy routing
E4.2.http_methods	2 (2/0)	all HTTP methods	read-only proxy (GET/HEAD only; POST 403)	Method-filtering API gateway / MITM egress policy
E4.3.clock	5 (5/0)	NTP-synced clock	+365d / -365d / frozen libfaketime	TLS/token TTL validation & monotonic timer (time.monotonic)
E4.4.secret_visibility	3 (0/3)	runtime env visible	setup-only purged / placeholder token	Zero-trust build-vs-runtime secret scoping
E4.5.limit_enforcement	8 (0/8)	permissive rlimits	kernel RLIMIT_FSIZE/NOFILE/AS / hard OOM	anthropics/claude-code#85639; POSIX setrlimit(2)
E5.1.residual	1 (0/1)	clean workspace	stale future-mtime .pyc / lockfile residue	anthropics/claude-code#42269; PEP 3147 bytecode cache
E5.2.service_reliability	1 (0/1)	100% healthy upstream	intermittent 503 Service Unavailable	Idempotent exponential-backoff retry engineering

B Verified Sandbox Suite, Runner Security, and Model Configurations

B.1 Probe-Verified Sandbox Configurations

Table 11 lists all 55 named sandbox configurations in TwinEnv-Bench, comprising one canonical base profile plus 54 single-property variants. Across the 117 tasks, 45 tasks bind directly to named sandbox variants, while 72 tasks instantiate task-specific environment configurations across standard twin-container, scheduled daemon, and host-isolated execution profiles, each validated under the six-run three-state solvability matrix in Equation 4 across `--mode none`, `--mode naive`, and `--mode ref`.

Every named sandbox variant undergoes automated single-variable verification under Equation 3: the audit harness launches each container under the exact agent user identity, login shell, and environment wrapper, records all host probe readings, and asserts both that the set of properties differing from the baseline container matches the declared shift without collateral drift and that live runtime assertions confirm active enforcement.

Table 11: **Complete registry of all 55 probe-verified sandbox configurations.** The suite comprises the canonical base profile plus 54 single-property variants, listing each targeted canonical mechanism, shifted value, container realization, and automated probe-and-assertion verification mode.

Sandbox ID	Mechanism	Target Value	Container Realization & Verification Mode
base	all (E1–E5)	standard	tlsbx:py3.12 (UTF-8, root, GNU, online, synced clock)
fs-symlink-otherfs	E1.1.symlink	symlink-to-other-fs	docker=--tmpfs /mnt/othervol:rw,exec (check)
fs-casefold	E1.1.case	insensitive	docker=--privileged (check)
fs-crlf-checkout	E1.2.line_ending	crlf	setup: git config --system core.autocrlf true (check)
py-old	E2.1.version	old	image=tlsbx:py3.8 (check)
py-future	E2.1.version	future	image=tlsbx:py3.14 (check)
py-alias-missing	E2.1.entry_name	alias-missing	setup: rm -f /usr/local/bin/python /usr/local/b... (probe)
deps-absent	E2.2.preinstalled	absent	setup: pip uninstall -y -q pandas pyyaml reques... (check)
deps-partial	E2.2.preinstalled	partial	setup: pip uninstall -y -q pandas >/dev/null 2>... (check)
pkg-mirror	E2.2.package_source	internal-mirror	env: PIP_INDEX_URL=https://mirrors.a... (probe)
pkg-unreachable	E2.2.package_source	unreachable	env: PIP_INDEX_URL=http://127.0.0.1:... (probe+check)
no-build-tools	E2.2.build_tools	absent	setup: rm -f /usr/bin/gcc* /usr/bin/g++* /usr/b... (probe)
tc-busybox	E2.3.toolchain	busybox	env: PATH=/opt/tc/busybox:/... (probe)
tc-toybox	E2.3.toolchain	toybox	env: PATH=/opt/tc/toybox:/u... (probe+check)
sh-dash	E2.3.shell	sh-dash	shell=dash; env: SHELL=/bin/dash (probe)
sh-zsh	E2.3.shell	zsh	shell=zsh; env: SHELL=/usr/bin/zsh (probe)
sh-fish	E2.3.shell	fish	shell=fish; env: SHELL=/usr/bin/fish (probe)
sh-mksh	E2.3.shell	mksh	shell=mksh; env: SHELL=/bin/mksh (probe)
enc-gbk	E3.1.encoding	gbk	env: LC_ALL=zh_CN.GBK (probe)
enc-big5	E3.1.encoding	cp950	env: LC_ALL=zh_TW.BIG5 (probe)
loc-en_US	E3.1.locale	en_US	env: LC_ALL=en_US.UTF-8 (probe)
loc-de_DE	E3.1.locale	de_DE	env: LC_ALL=de_DE.UTF-8 (probe)
loc-tr_TR	E3.1.locale	tr_TR	env: LC_ALL=tr_TR.UTF-8 (probe)
loc-zh_CN	E3.1.locale	zh_CN	env: LC_ALL=zh_CN.UTF-8 (probe)
tz-fixed	E3.1.timezone	fixed-offset-zone	env: TZ=Asia/Shanghai (probe)
tz-dst	E3.1.timezone	dst-zone	env: TZ=America/New_York (probe)
tz-halfhour	E3.1.timezone	half-hour-offset-zone	env: TZ=Asia/Kolkata (probe)
id-unprivileged	E3.2.identity	unprivileged-user	user=agent (probe)
tty-pty	E3.2.tty	pty	wrapper: script -qec (probe)
env-minimal	E3.2.env_inherit	minimal	wrapper: env -i PATH=/usr/local/bin:/usr... (probe+check)
env-pure	E3.2.env_inherit	pure	wrapper: env -i (probe+check)
perm-sudo	E4.1.permissions	sudo-nopasswd	user=agent (probe)
perm-nosudo	E4.1.permissions	no-sudo	user=agent (probe)
perm-readonly-sys	E4.1.permissions	readonly-sys	docker=--read-only --tmpfs /tmp:rw,exec (check)

Sandbox ID	Mechanism	Target Value	Container Realization & Verification Mode
perm-restricted-caps	E4.1.permissions	restricted-caps	docker=--cap-drop ALL --security-opt (check)
mnt-tmp-noexec	E4.1.mount	tmp-noexec	docker=--tmpfs /tmp:rw,noexec,nosuid (probe)
mnt-ro-workspace	E4.1.mount	readonly-workspace	mount=/workspace:ro (check)
net-offline	E4.2.egress	offline	docker=--network none (probe)
net-allowlist	E4.2.egress	allowlist	docker=--cap-add NET_ADMIN; wrapper: setpriv --inh-caps=-net_admin -... (probe)
net-proxy	E4.2.egress	proxy-required	env: HTTPS_PROXY=http://tlsbx-prox..., HTTP_PROXY=http://tlsbx-prox... (probe)
dns-servfail	E4.2.dns	servfail	docker=--dns 127.0.0.2 (probe)
clock-future	E4.3.clock	future+1y	env: LD_PRELOAD=/usr/lib/x86_64-l..., FAKETIME=+365d (probe)
clock-past	E4.3.clock	past-1y	env: LD_PRELOAD=/usr/lib/x86_64-l..., FAKETIME=-365d (probe)
clock-frozen	E4.3.clock	frozen	env: LD_PRELOAD=/usr/lib/x86_64-l..., FAKETIME=2026-01-01 00:00:00 (probe)
secret-setup-only	E4.4.secret_visibility	setup-only	env: API_TOKEN= (check)
secret-placeholder	E4.4.secret_visibility	placeholder	env: API_TOKEN=__PLACEHOLDER_API... (check)
lim-overcommit	E4.5.limit_enforcement	overcommit	cgroup=--memory-reservation 4g --cpus (check)
lim-hardkill	E4.5.limit_enforcement	hard-kill	cgroup=--memory 4g --memory-swap (check)
sh-pwsh	E2.3.shell	powershell-7	shell=pwsh; env: SHELL=/usr/bin/pwsh (probe)
web-chromium	E2.4.browser_engine	chromium	image=tlsbx-web:py3.12; env: BROWSER_ENGINE=chromium (check)
web-firefox	E2.4.browser_engine	firefox	image=tlsbx-web:py3.12; env: BROWSER_ENGINE=firefox (check)
web-webkit	E2.4.browser_engine	webkit	image=tlsbx-web:py3.12; env: BROWSER_ENGINE=webkit (check)
net-mirror-only	E4.2.egress	mirror-only	docker=--cap-add NET_ADMIN; env: PIP_INDEX_URL=https://mirrors.a...; wrapper: setpriv --inh-caps=-net_admin -... (probe+check)
dns-sandbox-resolver	E4.2.dns	sandbox-resolver	docker=--dns 172.30.0.53 (check)
net-http-readonly	E4.2.http_methods	read-only	env: HTTPS_PROXY=http://tlsbx-mitm..., HTTP_PROXY=http://tlsbx-mitm... (probe+check)

B.2 Four-Profile Execution Dispatcher and Zero-Leakage Audit

The unified host evaluation dispatcher routes all 117 tasks transparently across four runtime isolation profiles according to the physical boundary required by Δd_i. First, the standard container profile covers 33 tasks where e_i^{base} and e_i^{target} differ via setup scripts, mounted loop volumes, or environment variables, keeping held-out grading inputs outside the container until agent termination. Second, the named variant sandbox profile covers 45 tasks bound directly to configurations in Table 11, including isolated bridge networks, HTTP and read-only proxy sidecars, clock-skew interposition, and cross-engine headless browser images. Third, the scheduled daemon profile covers 20 tasks where interactive shell state matches the baseline container while post-termination verification invokes the target job under background cron rules in `/etc/cron.d/nightly-job`. Fourth, the host-isolated pressure profile covers 19 tasks enforcing pre-return staging-directory removal under `/opt/stage`, zero-leakage visibility checks under the unprivileged task UID, per-command umask and kernel setrlimit caps across file size, open descriptors, stack size, and virtual memory, a 6,000-token per-turn output cap, and post-termination judge payload injection.

B.3 Evaluated Model Snapshots and Execution Accounting

Table 12 reports exact snapshot identifiers, API protocols, and executed versus refusal trial counts across all nine frontier models. All models use vendor-recommended high-reasoning configurations; Anthropic Claude Opus 5.5 is queried via the native Anthropic Messages API with high reasoning effort, while the remaining eight models are queried via OpenAI-compatible Chat Completions endpoints with native tool calling enabled.

Table 12: Evaluated frontier model snapshots, API protocols, and non-refusal executed versus API-refused trial counts across all 117 tasks. All 6,318 scheduled trials across 9 models, 234 twin cells, and 3 repeats reached terminal verdicts, comprising all 3,159 target slots with 3,115 non-refusal executed runs and all 3,159 base slots with 3,120 non-refusal executed runs.

Display Name	**Snapshot ID**	**Provider**	**API Protocol**	**Base Exec / Ref.**	**Target Exec / Ref.**
Claude Opus 5.5	`claude-opus-5-5`	Anthropic	Anthropic Messages, high effort	314 / 37	307 / 44
Qwen 3.8 Max	`qwen3.8-max`	Alibaba	OpenAI Chat Completions	349 / 2	351 / 0
GPT-6.1 Sol	`gpt-6.1-sol`	OpenAI	OpenAI Chat Completions	351 / 0	351 / 0
Gemini 4 Argon	`gemini-4-argon`	Google	OpenAI Chat Completions	351 / 0	351 / 0
Hy4 Flash	`hy4-flash`	Tencent	OpenAI Chat Completions	351 / 0	351 / 0
DeepSeek V4.1 Flash	`deepseek-v4.1-flash`	DeepSeek	OpenAI Chat Completions	351 / 0	351 / 0
GLM-5.3 Flash	`glm-5.3-flash`	Zhipu AI	OpenAI Chat Completions	351 / 0	351 / 0
Grok 4.7	`grok-4-7`	xAI	OpenAI Chat Completions	351 / 0	351 / 0
Kimi K3.1 Pro	`kimi-k3.1-pro`	Moonshot AI	OpenAI Chat Completions	351 / 0	351 / 0

C Fine-Grained Dimension-Group Capabilities and Unsolved Tasks

Figure 12 and Table 13 provide the complete 15×9 breakdown of Target Pass@1 across every canonical dimension group from E1.1 to E5.2 and all nine evaluated frontier models, complementing the five-layer summary in Section 4.4.



Figure 12: **Fine-Grained Capability Heatmap Across All 15 Environment Dimension Groups and 9 Frontier Models.** Cell values report Target Pass@1 in percent. Dark cell borders denote per-group top performers; rose summary pills on the right highlight the two multi-task bottleneck groups below a 39% 9-model macro mean, namely E4.5 Resource Limits across 8 tasks at 21.9% and E1.1 Filesystem Semantics across 10 tasks at 38.4%, while single-task E5.2 Service Reliability stands at 11.1%.

Micro-Level Cross-Family Contrasts. Inspecting Figure 12 reveals pronounced fine-grained divergences hidden by macro averages. Comparing E4.5 Resource Limit Enforcement against E4.2 Network Boundaries, Claude Opus 5.5 leads E4.5 on non-refusal runs at 76.2%, standing 30.4 percentage points ahead of second-place GPT-6.1 Sol at 45.8%, whereas Qwen 3.8 Max and Hy4 Flash drop to 4.2% on E4.5 because verbose shell inspection or staging outputs collide with tight output and resource ceilings—yet Qwen 3.8 Max leads all models on E4.2 Network Boundaries at 87.9% and reaches 66.7% on the single-task E5.2 External Service Reliability case, while Hy4 Flash claims sole first place on E4.1 Permissions and Mounts at 77.8%, 7.4 points ahead of Qwen 3.8 Max and GPT-6.1 Sol at 70.4%. On E4.4 Credential Visibility, GPT-6.1 Sol achieves 88.9% across setup-only and placeholder secret tasks, while Claude Opus 5.5 scores 0.0% due to conservative abstention around missing authentication secrets in executed runs. Meanwhile, Gemini 4 Argon leads both E1.1 Filesystem Semantics at 73.3%, 21.4 points over second-place Claude Opus 5.5, and E4.3 System Clock Skew at 80.0%, reflecting strong handling of Unicode NFD normalization and monotonic time APIs.

Diagnosis of the 8 Universally Unsolved Tasks. Table 14 catalogs the only 8 tasks, representing 6.8% of TwinEnv-Bench, on which all nine models fail across every executed trial out of 27 scheduled runs per task. Seven of the eight belong to the deployment-latent regime, spanning six host-isolated tasks and one scheduled daemon task across FIFO named pipes, symlinked inbox delivery, half-hour cron timezone overrides, stripped cron PATH directories, and cross-UID repository, hard-link, or output ownership boundaries. Several of these tasks exhibit moderate-to-high probe rates

of 60%–81%, underscoring that even when agents inspect the host, multi-constraint POSIX ownership and stream semantics remain an open frontier.

Table 13: **Numerical Target Pass@1 matrix across all 15 canonical dimension groups and 9 frontier models.** Values are percentages complementing Figure 12.

Dimension Group (N)	Opus	Qwen	GPT	Gem.	Hy4	DS	GLM	Grok	Kimi	Mean
E1.1 Filesystem Semantics (10)	51.9	46.7	40.0	**73.3**	40.0	33.3	20.0	26.7	13.3	**38.4**
E1.2 System Conventions (2)	66.7	**100.0**	**100.0**	**100.0**	**100.0**	66.7	50.0	83.3	50.0	**79.6**
E2.1 Language Runtime (5)	66.7	66.7	**86.7**	80.0	66.7	40.0	46.7	20.0	40.0	**57.0**
E2.2 Dependencies & Packages (8)	39.1	41.7	**62.5**	37.5	37.5	41.7	33.3	33.3	29.2	**39.5**
E2.3 System Toolchain & Shell (9)	79.2	74.1	**88.9**	70.4	70.4	22.2	59.3	40.7	40.7	**60.6**
E2.4 Browser Engine (2)	**83.3**	66.7	50.0	50.0	66.7	33.3	50.0	0.0	16.7	**46.3**
E3.1 Locale, Encoding & Timezone (31)	**79.5**	73.1	62.4	59.1	55.9	46.2	39.8	35.5	21.5	**52.6**
E3.2 Process Context (12)	60.0	**61.1**	44.4	58.3	55.6	47.2	36.1	38.9	25.0	**47.4**
E4.1 Permissions & Mounts (9)	65.2	70.4	70.4	66.7	**77.8**	63.0	59.3	44.4	40.7	**62.0**
E4.2 Network Boundaries (11)	58.1	**87.9**	75.8	66.7	66.7	51.5	60.6	48.5	42.4	**62.0**
E4.3 System Clock Skew (5)	69.2	73.3	60.0	**80.0**	73.3	46.7	26.7	26.7	40.0	**55.1**
E4.4 Credential Visibility (3)	0.0	66.7	**88.9**	44.4	55.6	33.3	66.7	55.6	11.1	**46.9**
E4.5 Resource Limit Enforcement (8)	**76.2**	4.2	45.8	37.5	4.2	12.5	8.3	8.3	0.0	**21.9**
E5.1 Platform Residuals (1)	**100.0**	66.7	66.7	66.7	66.7	33.3	66.7	0.0	33.3	**55.6**
E5.2 External Service Reliability (1)	33.3	**66.7**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	**11.1**

Table 14: **The 8 universally unsolved counterfactual tasks in TwinEnv-Bench.** All nine frontier models score 0 passes across all executed target runs out of 27 scheduled trials per task; 7 of the 8 belong to the deployment-latent regime.

Task ID	Mechanism	Regime	Probe%	Primary Counterfactual Failure Bottleneck
E1.1-03-stream-feed-shares	E1.1.file_type	Latent	4%	Production feed arrives via a single-read FIFO named pipe; two-pass validation and aggregation blocks indefinitely on the second open.
E1.1-08-symlinked-inbox-merge	E1.1.symlink	Latent	11%	Transfer daemon delivers report blobs via symbolic links; find -type f skips symlinks and exits 0 with a header-only output.
E2.2-03-mirror-pin-audit-names	E2.2.package_source	Exposed	70%	Internal PEP 503 package mirror returns 404 on unnormalized mixed-case or underscore package names without PyPI 301 redirects.
E3.1-29-halfhour-log-localtime	E3.1.timezone	Latent	81%	Nightly cron job overrides TZ to Asia/Kolkata at +05:30 while login shell stays Asia/Tokyo; integer-hour rounding misformats +06:00.
E3.2-05-cron-path-report	E3.2.env_inherit	Latent	65%	Scheduled cron environment strips /opt/reporting/bin from PATH, causing bare formatter invocation to fail with exit code 127.
E3.2-06-shared-checkout-relnotes	E3.2.identity	Latent	15%	Production release repository is owned by deploy UID, triggering Git safe.directory dubious-ownership rejection under the job account.
E3.2-07-nonroot-hardlink-store	E3.2.identity	Latent	60%	Unprivileged ingest UID hits EPERM under Linux fs.protected_hardlinks=1 when hard-linking scanner-owned inbox files.
E3.2-08-foreign-owned-output	E3.2.identity	Latent	19%	Prior export left a foreign-owned file in a writable outbox directory; in-place overwrite raises PermissionError without atomic rename.

D Probe Attribution Protocol and Counterfactual Trajectory Case Studies

D.1 Probe Matching and Failure-Mode Attribution Protocol

To guarantee reproducible behavioral diagnostics in Section 5 and Table 8, every executed trajectory is processed through a three-stage attribution pipeline combining syntax-aware command parsing, mechanism-aligned surface matching, and trace-audited failure partitioning.

Stage 1: Syntax-Aware Atomic Probe Extraction. For every shell tool invocation, the parser removes file-editing patch blocks and file-redirected heredoc bodies so business logic and data columns never trigger false positives, splits compound pipelines and subshells into atomic commands, and strips execution wrappers such as sudo, env, nice, time, and timeout. An atomic command is labeled an environment probe $a_t \in \mathcal{A}_{\text{probe}}$ when it invokes one of 21 read-only system utilities such as uname, locale, ulimit, umask, id, findmnt, getconf, timedatectl, or capsh; reads one of nine system configuration or procfs paths such as `/etc/cron.d`, `/etc/timezone`, `/etc/resolv.conf`, `/proc/self/limits`, or `/proc/self/status`; echoes one of eight host environment variables such as TZ, LANG, LC_ALL, PATH, or UMASK; or runs a non-redirected Python introspection one-liner querying encoding, rlimits, UID, timezone, or mount state.

Stage 2: Mechanism-Specific On-Target Matching. Each of the 31 canonical mechanisms is paired with a deterministic target-surface predicate specifying the probe families capable of revealing Δd_i: for instance, timezone shifts match date offset queries, timedatectl, TZ, `/etc/timezone`, `/etc/localtime`, runtime tzname checks, or `/etc/cron.d` schedule files; resource-quota shifts match ulimit, prlimit, `/proc/self/limits`, or getrlimit; and mount or capability shifts match findmnt, mount, df, `/proc/self/mounts`, capsh, or `/proc/self/status`. A probed trajectory with $n_{\text{probe}} \geq 1$ is classified as on-target if at least one probe satisfies the task's target-surface predicate, defining On-Tgt% and conditional solve rate P | On-Tgt in Table 8.

Stage 3: Failure Partitioning on Consistently Probed Zero-Pass Pairs. Across the 89 task–model pairs comprising 267 trajectories that probe in all three target trials yet pass zero of three trials in Section 5.3, each failed trajectory falls into one of three mutually exclusive categories verified against full execution transcripts: off-target reconnaissance at 21.3%, or 57 of 267 runs, where $n_{\text{probe}} \geq 1$ yet zero probes match the task's target-surface predicate; single-step shallow probing at 49.4%, or 132 of 267 runs, where the agent touches the target category once in the login shell with $n_{\text{on-target}} = 1$ without inspecting the secondary deployment configuration or fine-grained kernel file governing Δd_i; and the *probe-to-action* reasoning gap at 29.2%, or 78 of 267 runs, where the agent executes $n_{\text{on-target}} \geq 2$ probes surfacing the active constraint yet submits boundary-unsafe code.

D.2 Qualitative Paired Trajectory Case Boxes

Case Boxes D.1–D.4 present paired execution trace excerpts from our sanitized public trajectory release across four representative tasks illustrating each behavioral mechanism.

Case Box D.1: Premature Blind Exit · Task E4.5-05 · Layer E4.5 · Deployment-Latent

Task & Single-Variable Shift: Clean and partition national outage records into regional CSV files. The inherited script stages all cleaned rows into a single intermediate file before splitting. Baseline uses unlimited file size, whereas target caps kernel RLIMIT_FSIZE at 1 MiB, equal to 1,024 blocks under ulimit.

Unprobed Trajectory: 0/3 Fail · 6 Commands

```
$ ls -la /workspace && head -n 5 sample/outages.csv
$ cat split_outages.py
# Edits cleaning regex; keeps monolithic staging.tmp
$ python3 split_outages.py --in sample/outages.csv
Wrote 4 regional CSVs (staging.tmp: 4.8 KB, exit 0)
# Zero errors on 60-row sample -> terminates!
[Post-Run Judge] OSError: [Errno 27] File too large
while writing staging.tmp on 3.1 MB national feed.
```

Upfront Recon Trajectory: Claude Opus 5.5 · 3/3 Pass

```
$ id; ulimit -a; df -h /workspace /tmp
file size (blocks, -f) 1024 # <-- Spots 1 MiB cap!
$ cat split_outages.py
# Removes monolithic staging file; streams rows
# directly into per-region buffered writers
$ python3 test_synthetic_50k_rows.py # >2 MB total
All regional shards < 600 KB; exit 0
[Post-Run Judge] PASS: all regional CSVs exact.
```

Case Box D.2: Shallow Probing Illusion · Task E3.1-29 · Layer E3.1 · Deployment-Latent

Task & Single-Variable Shift: Add an `--errors-only` flag to `/workspace/logs/annotate.py` while preserving exact local timestamp offsets when invoked by the nightly scheduled job. Baseline sets TZ=Asia/Tokyo at +09:00 in both the login shell and `/etc/cron.d/nightly-job`; target keeps Asia/Tokyo in the login shell while overriding `/etc/cron.d/nightly-job` to TZ=Asia/Kolkata at +05:30.

1-Step Shallow Probe: GPT-6.1 / Opus · 0/3 Fail

```
$ date '+%Z %z' && echo "TZ=$TZ"
JST +0900 / TZ=Asia/Tokyo
# Stops probing! Assumes whole-hour +09:00 offset.
# Leaves inherited round(utcoffset/3600) intact,
# adding only --errors-only filter.
[Post-Run Judge] Under cron TZ=Asia/Kolkata +05:30,
round(19800/3600) formats +06:00 instead of +05:30!
```

Multi-Step Cron Recon: `--mode ref` / P2 Checklist

```
$ date '+%Z %z'; grep -rn 'TZ=' /etc/cron*
/etc/default
/etc/cron.d/nightly-job: TZ=Asia/Kolkata
# Identifies +05:30 half-hour offset in cron
context!
# Replaces round(sec/3600) with exact divmod(mins,
60)
$ TZ=Asia/Kolkata python3 annotate.py --errors-only
...
[Post-Run Judge] PASS: exact +05:30 offset in cron.
```

Case Box D.3: *probe-to-action* Reasoning Gap · Task E3.1-28 · Layer E3.1 · Interactive-Exposed

Task & Single-Variable Shift: Extend `/workspace/energy/rollup.py` to output each day's UTC peak hour from naive local smart-meter timestamps. Baseline uses monotonic TZ=UTC without daylight-saving jumps, whereas target uses TZ=Europe/Berlin, repeating 02:00–02:59 during the autumn DST fallback transition.

Reasoning Gap: 8 of 9 Models Fail 0/3 Despite Probing

```
$ date '+%Z %z' && cat /etc/timezone
CET +0100 / Europe/Berlin # <-- Probed target axis!
# Writes df['ts'].dt.tz_localize('Europe/Berlin')
# using default ambiguous='raise', overlooking
# repeated 02:00--02:59 hour in meter stream.
[Post-Run Judge] pytz.AmbiguousTimeError: Cannot
infer dst time from '2025-10-26 02:15:00'.
```

Boundary-Safe Deduction: Qwen 3.8 Max · 2/3 Pass

```
$ cat /etc/timezone && python3 -c "import time;
print(time.tzname)"
Europe/Berlin / ('CET', 'CEST')
# Deduces CET<->CEST transitions on chronological
# meter series: sets ambiguous='infer' and
# nonexistent='shift_forward'; stress-tests Oct 26.
[Post-Run Judge] PASS: spring & autumn DST exact.
```

Case Box D.4: Probe Granularity Blind Spot · Task E4.1-08 · Layer E4.1 · Deployment-Latent

Task & Single-Variable Shift: Update `/srv/sitepub/publish.py` to sync and prune a web document root while preserving the uploads directory and replacing read-only 0444 hotfix files owned by a foreign SFTP user. Baseline runs as unrestricted root with full Linux capabilities, whereas target drops CAP_DAC_OVERRIDE from root.

Coarse UID Check: DeepSeek / GLM · 0/3 Fail

```
$ id && ls -ld /var/www/html
uid=0(root) gid=0(root) # <-- Stops at uid=0!
# Assumes uid=0 bypasses 0444 mode on foreign files.
# Calls shutil.copy2(src, dst), which opens dst
# with O_WRONLY|O_TRUNC in place.
[Post-Run Judge] PermissionError: [Errno 13]
Permission denied: '/var/www/html/hotfix.css'.
```

Capability-Aware Atomic Replace: GPT-6.1 Sol · 3/3 Pass

```
$ id; grep Cap /proc/self/status
CapEff: 0000000000000000 # <-- Zero Linux caps!
# Recognizes root owns parent dir /var/www/html
# with mode 0755 so unlink/rename works even when
# O_WRONLY fails: writes temp + atomic os.replace().
[Post-Run Judge] PASS: foreign 0444 hotfixes
replaced.
```

E Causal Prompt Intervention Verbatim Prompts and Sensitivity

E.1 Verbatim System Prompt Suffixes

As detailed in Section 5.4, the 58 deployment-latent tasks span four runtime isolation profiles with two termination interfaces: the 9 standard container tasks, 10 named variant sandbox tasks, and 20 scheduled daemon tasks invoke a dedicated finish tool, whereas the 19 host-isolated tasks terminate when the assistant emits a final summary turn without tool calls. Overwriting native system prompts with a single static string would break termination semantics on host-isolated tasks. Accordingly, P1 and P2 append identical task-agnostic suffixes after a blank line across all 58 tasks while leaving P0 unmodified, with P2 organized as a taxonomy-aligned five-layer checklist without per-task hints:

Arm P1, Generic Reminder — Appended Verbatim System-Prompt Suffix

```
IMPORTANT ENGINEERING RULE: Do not assume this machine uses standard default settings, and do
not rely solely on small sample inputs passing in your interactive shell. Before editing code
or finalizing your solution, proactively inspect the host environment, execution context, and
how the program will be invoked in production so your solution handles the actual machine
configuration.
```

Arm P2, Taxonomy-Aligned Pre-Flight Checklist — Appended Verbatim System-Prompt Suffix

```
IMPORTANT PRODUCTION RECONNAISSANCE PROTOCOL: Do not assume a default container setup, and
never treat a clean run on small sample data in your interactive shell as sufficient proof of
correctness. In your initial steps-before writing code-proactively run diagnostic commands to
audit the host environment across five dimensions:
1. Filesystem & Mounts: check mount boundaries (`df -T`, `findmnt`), symlinks, case/Unicode
behavior, and path limits.
2. Runtime & Toolchain: verify exact interpreter/binary versions, shell flavor, and available
offline packages.
3. Execution & Scheduling Context: check `locale`, `TZ`/`/etc/timezone`, and inspect
`/etc/cron*`, systemd units, or wrapper scripts for non-interactive environment overrides.
4. Permissions, Boundaries & Quotas: check `id`, file ownership/modes, `umask`, Linux
capabilities (`/proc/self/status`), network/proxy variables, and kernel resource limits
(`ulimit -a`).
5. State & Services: check for stale lock/cache artifacts and test upstream service behavior.
Design and stress-test your implementation explicitly against every constraint you discover.
```

E.2 Pairing Accounting and Sensitivity Analyses

Across 4 models × 58 tasks × 3 repeats = 696 scheduled slots per arm, P0 completes 696/696 terminal runs; P1 completes 693/696 terminal runs after excluding 1 transient gateway drop without verdict on GLM-5.3 Flash task E4.1-08 repeat 1 and 2 single-turn 6,000-token output truncations on GLM-5.3 Flash; and P2 completes 691/696 terminal runs after excluding 5 single-turn 6,000-token output truncations comprising 4 on GLM-5.3 Flash and 1 on DeepSeek V4.1 Flash. All three sensitivity denominators yield virtually identical effect sizes. Under our primary specification in Table 9 using strict common terminal pairing at $n = 688$, Pass@1 rises from 32.0% to 48.7% and 61.9%, yielding $\Delta_{P2-P0} = +29.9$ points and $\Delta_{P2-P1} = +13.2$ points with a 95% task-bootstrap CI of $[+6.7, +20.1]$ points. Under count-truncation-as-failure pairing at $n = 695$, which scores all 7 truncated runs as zero, Pass@1 rises from 31.6% to 48.6% and 61.4%, yielding $\Delta_{P2-P0} = +29.8$ points and $\Delta_{P2-P1} = +12.8$ points with a 95% CI of $[+6.3, +19.6]$ points. Excluding the two case-folding tasks E1.1-01 and E1.1-02 at $n = 664$ similarly yields 32.7% to 49.4% and 63.0%, with $\Delta_{P2-P1} = +13.6$ points and a 95% CI of $[+6.9, +20.7]$ points.

F Complete 117-Task Catalog, Execution Guide, and Datasheet

F.1 Complete 117-Task Counterfactual Catalog

Table 15 enumerates all 117 counterfactual tasks in TwinEnv-Bench ordered by canonical identifier from E1.1-01 through E5.2-01, alongside each task's fine-grained mechanism, exposure regime, target sandbox or runner profile, 9-model mean Target Pass@1, and 9-model mean host probe rate.

Table 15: **Complete catalog of all 117 counterfactual tasks in TwinEnv-Bench.** Columns list canonical task ID, fine-grained mechanism, exposure regime, target sandbox or runner profile, 9-model mean Target Pass@1, and 9-model mean host probe rate.

Canonical Task ID	Mechanism	Regime	Target Config / Profile	Pass@1	Probe%
E1.1-01-casefold-profile-export	E1.1.case	Latent	base_container	29.6%	59%
E1.1-02-casefold-asset-rename	E1.1.case	Latent	base_container	11.1%	67%
E1.1-03-stream-feed-shares	E1.1.file_type	Latent	host_isolated	0.0%	4%
E1.1-04-name255-build-tag-reports	E1.1.length	Latent	host_isolated	44.4%	56%
E1.1-05-symlink-include-guard	E1.1.symlink	Exposed	base_container	63.0%	30%
E1.1-06-symlink-archive-report	E1.1.symlink	Exposed	base_container	81.5%	44%
E1.1-07-symlink-release-bundle	E1.1.symlink	Exposed	base_container	100.0%	44%
E1.1-08-symlinked-inbox-merge	E1.1.symlink	Latent	host_isolated	0.0%	11%
E1.1-09-nfd-gallery-manifest	E1.1.unicode	Latent	host_isolated	37.0%	59%
E1.1-10-nfd-shell-release-assets	E1.1.unicode	Latent	host_isolated	11.5%	85%
E1.2-01-crlf-refund-report	E1.2.line_ending	Exposed	base_container	74.1%	48%
E1.2-02-crlf-alert-hook	E1.2.line_ending	Latent	base_container	85.2%	30%
E2.1-01-noalias-lint-shebang	E2.1.entry_name	Exposed	py-alias-missing	66.7%	26%
E2.1-02-pyold-isoz-sessions	E2.1.version	Exposed	py-old	37.0%	7%
E2.1-03-pyold-fsum-reconcile	E2.1.version	Exposed	py-old	22.2%	11%
E2.1-04-py314-alias-pool	E2.1.version	Exposed	base_container	100.0%	4%
E2.1-05-py313-imghdr-sniff	E2.1.version	Exposed	base_container	58.3%	8%
E2.2-01-nocc-crcmod-verify	E2.2.build_tools	Exposed	base_container	55.6%	37%
E2.2-02-nocc-sdist-pin	E2.2.build_tools	Exposed	base_container	85.2%	81%
E2.2-03-mirror-pin-audit-names	E2.2.package_source	Exposed	pkg-mirror	0.0%	70%
E2.2-04-unreach-outdated-report	E2.2.package_source	Exposed	pkg-unreachable	29.6%	81%
E2.2-05-unreach-autoinstall-xlsx	E2.2.package_source	Exposed	pkg-unreachable	19.2%	42%
E2.2-06-depsabsent-yaml-overrides	E2.2.preinstalled	Exposed	deps-absent	14.8%	11%
E2.2-07-depspartial-bom-regions	E2.2.preinstalled	Exposed	deps-partial	11.1%	15%
E2.2-08-depspartial-p95-latency	E2.2.preinstalled	Exposed	deps-partial	100.0%	7%
E2.3-01-mksh-bundle-budget	E2.3.shell	Exposed	sh-mksh	65.4%	54%
E2.3-02-mksh-release-notes	E2.3.shell	Exposed	sh-mksh	69.2%	50%
E2.3-03-dash-echo-index	E2.3.shell	Exposed	base_container	88.5%	15%
E2.3-04-dash-subject-length	E2.3.shell	Exposed	base_container	3.7%	30%
E2.3-05-toybox-perf-slowest	E2.3.toolchain	Exposed	tc-toybox	33.3%	44%
E2.3-06-toybox-crm-normalize	E2.3.toolchain	Exposed	tc-toybox	85.2%	63%
E2.3-07-bb-awk-egress	E2.3.toolchain	Exposed	base_container	66.7%	44%
E2.3-08-bb-grep-feed	E2.3.toolchain	Exposed	base_container	92.6%	93%
E2.3-09-bb-date-sessions	E2.3.toolchain	Exposed	base_container	40.7%	22%
E2.4-01-ff-beacon-tagcheck	E2.4.browser_engine	Exposed	web-firefox	77.8%	78%
E2.4-02-wk-dateparse-incidents	E2.4.browser_engine	Exposed	web-webkit	14.8%	85%
E3.1-01-gbk-ticket-digest	E3.1.encoding	Latent	host_scheduled	63.0%	52%
E3.1-02-gbk-git-relnotes	E3.1.encoding	Latent	host_scheduled	3.7%	22%
E3.1-03-gbk-bank-batch	E3.1.encoding	Latent	host_scheduled	100.0%	85%
E3.1-04-big5-fsenc-manifest	E3.1.encoding	Latent	enc-big5	65.4%	81%
E3.1-05-big5-iconv-sepa	E3.1.encoding	Exposed	enc-big5	100.0%	100%
E3.1-06-collate-bisect-lookup	E3.1.locale	Latent	host_scheduled	81.5%	89%
E3.1-07-collate-latest-backup	E3.1.locale	Exposed	base_container	18.5%	48%
E3.1-08-numeric-mawk-hours	E3.1.locale	Latent	host_scheduled	3.7%	19%
E3.1-09-numeric-sort-topn	E3.1.locale	Latent	host_scheduled	74.1%	59%
E3.1-10-numeric-java-qty	E3.1.locale	Latent	host_scheduled	59.3%	78%
E3.1-11-tr-java-config	E3.1.locale	Latent	host_scheduled	23.1%	38%
E3.1-12-tr-bash-loglevel	E3.1.locale	Latent	host_scheduled	14.8%	26%
E3.1-13-tr-grep-authreport	E3.1.locale	Latent	host_scheduled	22.2%	41%
E3.1-14-time-strptime-accesslog	E3.1.locale	Latent	host_scheduled	77.8%	100%
E3.1-15-time-date-monthabbr	E3.1.locale	Latent	host_scheduled	51.9%	81%
E3.1-16-time-weekday-settle	E3.1.locale	Latent	host_scheduled	18.5%	33%
E3.1-17-collate-lt-shardroute	E3.1.locale	Latent	host_scheduled	59.3%	74%
E3.1-18-numeric-grouping-metrics	E3.1.locale	Latent	host_scheduled	77.8%	100%
E3.1-19-zhcn-sortm-authmerge	E3.1.locale	Exposed	loc-zh_CN	55.6%	78%
E3.1-20-zhcn-range-nslabels	E3.1.locale	Exposed	loc-zh_CN	92.6%	100%
E3.1-21-cron-locale-labels	E3.1.locale	Latent	host_isolated	29.2%	92%
E3.1-22-locale-join	E3.1.locale	Latent	host_scheduled	29.6%	56%
E3.1-23-emptytz-till-roundtrip	E3.1.timezone	Latent	base_container	74.1%	89%

Canonical Task ID	Mechanism	Regime	Target Config / Profile	Pass@1	Probe%
E3.1-24-user-tz-meeting	E3.1.timezone	Exposed	base_container	85.2%	85%
E3.1-25-yesterday-report	E3.1.timezone	Exposed	base_container	85.2%	100%
E3.1-26-dst-session-billing	E3.1.timezone	Latent	host_scheduled	25.9%	81%
E3.1-27-dst-hourly-rollup	E3.1.timezone	Latent	host_scheduled	92.6%	100%
E3.1-28-dst-meter-localize	E3.1.timezone	Exposed	base_container	7.4%	100%
E3.1-29-halfhour-log-localtime	E3.1.timezone	Latent	host_scheduled	0.0%	81%
E3.1-30-halfhour-daily-gawk	E3.1.timezone	Exposed	base_container	88.9%	100%
E3.1-31-halfhour-quiet-hours-java	E3.1.timezone	Latent	host_scheduled	40.7%	93%
E3.2-01-envmin-shell-plan	E3.2.env_inherit	Exposed	env-minimal	77.8%	81%
E3.2-02-envmin-tz-metering	E3.2.env_inherit	Exposed	env-minimal	81.5%	93%
E3.2-03-envpure-childpy-ingest	E3.2.env_inherit	Exposed	env-pure	100.0%	100%
E3.2-04-envpure-home-netcfg	E3.2.env_inherit	Exposed	env-pure	92.6%	100%
E3.2-05-cron-path-report	E3.2.env_inherit	Latent	host_isolated	0.0%	65%
E3.2-06-shared-checkout-relnotes	E3.2.identity	Latent	host_isolated	0.0%	15%
E3.2-07-nonroot-hardlink-store	E3.2.identity	Latent	host_isolated	0.0%	60%
E3.2-08-foreign-owned-output	E3.2.identity	Latent	host_isolated	0.0%	19%
E3.2-09-root-dac-pinned	E3.2.identity	Latent	base_container	79.2%	96%
E3.2-10-git-owner-ci	E3.2.identity	Exposed	base_container	25.9%	52%
E3.2-11-pty-merge-autoedit	E3.2.tty	Exposed	tty-pty	66.7%	81%
E3.2-12-umask-nested-publish	E3.2.umask	Latent	host_isolated	41.7%	58%
E4.1-01-noexec-etl-transform	E4.1.mount	Latent	base_container	92.3%	100%
E4.1-02-noexec-native-scan	E4.1.mount	Latent	base_container	100.0%	100%
E4.1-03-xdev-spool-archive	E4.1.mount	Latent	host_isolated	7.4%	41%
E4.1-04-readonly-fx-cache	E4.1.permissions	Latent	base_container	48.1%	93%
E4.1-05-readonly-rotated-logs	E4.1.permissions	Latent	base_container	100.0%	70%
E4.1-06-readonly-xlsx-import	E4.1.permissions	Exposed	base_container	88.9%	41%
E4.1-07-caps-icmp-netcheck	E4.1.permissions	Exposed	perm-restricted-caps	76.0%	100%
E4.1-08-caps-foreign-docroot-prune	E4.1.permissions	Latent	perm-restricted-caps	44.4%	100%
E4.1-09-readonly-drop-ranking	E4.1.permissions	Latent	host_isolated	3.7%	33%
E4.2-01-dns-ipcheck-online	E4.2.dns	Exposed	dns-servfail	85.2%	52%
E4.2-02-resolver-java-url-equals	E4.2.dns	Exposed	dns-sandbox-resolver	44.4%	44%
E4.2-03-resolver-linkcheck-nxdomain	E4.2.dns	Exposed	dns-sandbox-resolver	22.2%	26%
E4.2-04-allowlist-gh-archive-dep	E4.2.egress	Exposed	net-allowlist	66.7%	59%
E4.2-05-allowlist-enrich-hang	E4.2.egress	Exposed	net-allowlist	81.5%	67%
E4.2-06-proxy-java-notice	E4.2.egress	Latent	net-proxy	18.5%	26%
E4.2-07-proxy-precheck-yanked	E4.2.egress	Exposed	net-proxy	33.3%	52%
E4.2-08-mirror-uv-index	E4.2.egress	Exposed	net-mirror-only	84.6%	100%
E4.2-09-mirror-hashlock-jsonapi	E4.2.egress	Exposed	net-mirror-only	74.1%	93%
E4.2-10-readonly-git-smart-http	E4.2.http_methods	Exposed	net-http-readonly	88.9%	100%
E4.2-11-readonly-batch-post-lookup	E4.2.http_methods	Exposed	net-http-readonly	84.6%	65%
E4.3-01-future-entitlement-allowlist	E4.3.clock	Exposed	clock-future	59.3%	56%
E4.3-02-skew-token-refresh	E4.3.clock	Exposed	base_container	51.9%	30%
E4.3-03-skew-http-cache	E4.3.clock	Exposed	base_container	55.6%	30%
E4.3-04-frozen-poll-deadline	E4.3.clock	Exposed	base_container	37.0%	63%
E4.3-05-frozen-lww-flags	E4.3.clock	Exposed	base_container	72.0%	60%
E4.4-01-setuponly-anon-tickets	E4.4.secret_visibility	Latent	secret-setup-only	12.0%	92%
E4.4-02-setuponly-ratelimit-export	E4.4.secret_visibility	Latent	secret-setup-only	84.0%	72%
E4.4-03-placeholder-hmac-bulk	E4.4.secret_visibility	Latent	secret-placeholder	58.3%	88%
E4.5-01-overcommit-reach-approx	E4.5.limit_enforcement	Latent	lim-overcommit	14.8%	22%
E4.5-02-overcommit-bash-logtriage	E4.5.limit_enforcement	Latent	lim-overcommit	37.0%	37%
E4.5-03-hardkill-shard-defer	E4.5.limit_enforcement	Latent	lim-hardkill	11.1%	41%
E4.5-04-hardkill-awk-dedup-window	E4.5.limit_enforcement	Latent	lim-hardkill	37.0%	63%
E4.5-05-fsize-staging-split	E4.5.limit_enforcement	Latent	host_isolated	11.1%	26%
E4.5-06-argmax-inbox-digest	E4.5.limit_enforcement	Latent	host_isolated	8.3%	50%
E4.5-07-nofile-station-files	E4.5.limit_enforcement	Latent	host_isolated	33.3%	33%
E4.5-08-memory-cap-usage-report	E4.5.limit_enforcement	Latent	host_isolated	14.8%	22%
E5.1-01-stale-bytecode-tariff	E5.1.residual	Latent	host_isolated	55.6%	15%
E5.2-01-flaky-503-shipments	E5.2.service_reliability	Latent	host_isolated	11.1%	11%

F.2 Quick-Start Verification Commands and Datasheet

Self-Contained Verification and Execution. The standalone TwinEnv-Bench release bundles all 117 task definitions across 1,709 files with zero external runtime downloads required. Practitioners can verify SHA-256 digests, restore exact octal file modes 0644 and 0755, validate the deterministic three-state mechanical contract without LLM API keys, and launch multi-run evaluations in three commands:

```
$ python3 fetch_tasks.py                              # Verify 1,709 SHA-256 hashes & octal modes
$ bash build_images.sh && python3 run.py --task E4.5-05-fsize-staging-split --mode ref
$ python3 run.py --all --cells base,target --mode agent --model gpt-6.1-sol --repeats 3 --jobs 2
```

Datasheet Summary [56].

- **Motivation & Purpose:** Created by TraceLite AI to measure single-variable environment adaptation, proactive host reconnaissance, and multi-run operational reliability in software and terminal agents without confounding algorithmic difficulty.
- **Composition & Instances:** 117 self-contained software and data engineering tasks across 234 baseline and target twin cells with prompts, initial workspaces, naive and reference implementations, and post-termination held-out grading suites, accompanied by 6,318 sanitized main-benchmark trajectories across 12,636 cryptographically verified result and trace files alongside 1,384 causal intervention trajectories.
- **Collection & Sanitization:** Tasks are constructed from public open-source agent failure patterns and POSIX specifications; no personally identifiable information, proprietary code, or live external production credentials are included, all test tokens use synthetic sentinels such as tl-demo-secret-0001, and all archive headers are normalized to root:root.
- **Distribution & Maintenance:** Code, container build specifications, and all 117 tasks are maintained at `https://github.com/TraceLite-AI/TwinEnv-Bench`, with interactive leaderboards and trajectory exploration hosted at `https://huggingface.co/spaces/TraceLite-AI/TwinEnv-Bench`.