TERMINALWORLD: Benchmarking Agents on Real-World Terminal Tasks

Zhaoyang Chu[1] **Jiarui Hu**[2*] **Xingyu Jiang**[2*] **Pengyu Zou**[2*] **Han Li**[2] **Chao Peng**[3]
Peter O'Hearn[1] **Earl T. Barr**[1] **Mark Harman**[1] **Federica Sarro**[1] **He Ye**[1†]
[1]University College London [2]Nanjing University [3]Tencent

Abstract

We introduce TERMINALWORLD, a scalable data engine that automatically reverse-engineers high-fidelity evaluation tasks from "in-the-wild" terminal recordings. Processing 80,870 terminal recordings, the engine yields a full benchmark of 1,530 validated tasks, spanning 18 real-world categories, ranging from short everyday operations to workflows exceeding 50 steps, and covering 1,280 unique commands. From these, we curate a VERIFIED subset of 200 representative, manually reviewed tasks. Comprehensive benchmarking on TERMINALWORLD-VERIFIED across eight frontier models and six agents reveals that current systems still struggle with authentic terminal workflows, achieving a maximum pass rate of only 62.5%. Moreover, TERMINALWORLD captures real-world terminal capabilities distinct from existing expert-curated benchmarks (*e.g.*, Terminal-Bench), with only a weak correlation to their scores (Pearson $r = 0.20$). The automated engine makes TERMINALWORLD authentic and scalable *by construction*, enabling it to evaluate agents in real-world terminal environments as developer practices evolve. Data and code are available at https://github.com/EuniAI/TerminalWorld.

1 Introduction

Terminal environments serve as a primary action space for autonomous agents to complete diverse tasks in complex software systems. Powered by advances in *Large Language Models* (LLMs) for multi-step reasoning and tool use, these agents are increasingly capable of automating terminal workflows by issuing commands, composing tools, and interpreting feedback within interactive CLI sessions, exemplified by open-source frameworks (*e.g.*, SWE-agent [Yang et al., 2024], OpenHands [Wang et al., 2025]) and commercial CLI assistants (*e.g.*, Claude Code [Anthropic, 2025], Codex CLI [OpenAI, 2025], Gemini CLI [Google, 2025]). Yet, how to reliably evaluate these agents on *real-world* terminal tasks remains an open question.

The prevailing answer has been manually curated benchmarks, such as Terminal-Bench [Merrill et al., 2026] and LongCLI-Bench [Feng et al., 2026], where domain experts author tasks paired with executable environments. However, experts often tend to prioritize adversarial puzzles to artificially maximize difficulty, thereby diverging from authentic, real-world terminal workflows. Moreover, this labor-intensive process struggles to scale with evolving terminal practices and diverse emerging tools, leaving benchmarks narrowly scoped and quickly outdated. While recent automated synthesis methods [Zhu et al., 2026, Lin et al., 2026, Gandhi et al., 2026, Pi et al., 2026, Wu et al., 2026] attempt to bypass this scalability bottleneck, they are primarily designed for training and rarely undergo the rigorous validation required to guarantee true authenticity. Therefore, we argue that an **authentic, scalable** evaluation system should be established to answer the key question: *"How well do terminal agents perform on the real-world tasks that evolve alongside everyday practices?"*

*These authors contributed equally as co-second authors.
†Corresponding author: he.ye@ucl.ac.uk

Preprint.

To answer this question, we argue that naturally occurring terminal operations, if faithfully recorded, can be *reverse-engineered* into evaluation tasks that are authentic by construction. The `asciinema` platform[1] makes this feasible: developers voluntarily share terminal session recordings, each with a structured transcript capturing every command and its corresponding system response. These recordings form a self-curated, human-vetted, and continuously growing corpus of authentic developer workflows. To systematically exploit this resource, we introduce TERMINALWORLD, a data engine that operationalizes this insight: it automatically turns in-the-wild terminal recordings into executable, rigorously validated evaluation tasks.

Turning raw recordings into evaluation tasks requires addressing three practical challenges, which our data engine systematically resolves: (1) ***Recordings are noisy and lack clear intent.*** Recording transcripts often contain typos, retries, and verbose system output, without explicit statements of the developer's goal. We address this by distilling each transcript into two artifacts via an LLM (*e.g.*, Claude Sonnet 4.6 [Anthropic, 2026a]): an outcome-oriented instruction that captures the developer's underlying intent and a clean command script as the reference solution. (2) ***Recordings do not capture the underlying execution environments.*** The transcript captures commands but not the underlying system state of the author's machine. We employ an LLM agent (*e.g.*, Claude Code [Anthropic, 2025]) to reverse-engineer this environment by inferring actual requirements while eliminating hallucinated dependencies. In particular, the agent physically builds a Docker image, launches the container, and replays the reference solution, using runtime failures as feedback for targeted repair. (3) ***Recordings lack an explicit test suite.*** While recordings naturally capture the execution trajectory, they lack an explicit test suite to automatically judge whether the task goal is achieved. Relying solely on LLMs to statically generate these tests is inherently vulnerable to false negatives (where correct solutions are rejected due to brittle tests) and false positives (where tasks can be trivially bypassed or solved by flawed workflows). We resolve this by equipping the agent with a trial-based refinement loop to generate and calibrate the test suite via actual execution feedback within the reproduced Docker container.

Running this engine over 80,870 raw `asciinema` recordings yields 1,530 validated terminal tasks as the full TERMINALWORLD benchmark. The resulting tasks span 18 real-world terminal categories, range from short everyday operations to workflows exceeding 50 steps, and cover 1,280 unique commands, 91% of which are absent from Terminal-Bench. Since the pipeline is fully automated and `asciinema` keeps accumulating new uploads, TERMINALWORLD can be re-run as the platform grows, allowing it to scale with evolving developer practices. Unlike prior benchmarks, TERMINALWORLD is authentic and scalable *by construction*.

From this collection, we curate a VERIFIED[2] subset of 200 tasks, each cross-reviewed by the authors who manually execute the reference solution in the reproduced environment and audit the semantic alignment across all artifacts. While the full set of 1,530 tasks provides a representative snapshot of in-the-wild terminal usage, this VERIFIED subset serves as a rigorous and challenging testbed for benchmarking frontier models and agents on complex, real-world terminal tasks.

Through comprehensive benchmarking experiments using TERMINALWORLD-VERIFIED across eight frontier LLMs (*e.g.*, Claude Opus 4.7 [Anthropic, 2026b], GPT-5.5 [OpenAI, 2026], Gemini 3.1 Pro [Google, 2026]) and six leading terminal agents (*e.g.*, Claude Code [Anthropic, 2025], Codex CLI [OpenAI, 2025], Gemini CLI [Google, 2025]), our analysis reveals several key findings. (1) Frontier LLMs still struggle with real-world terminal tasks: even the best model solves only 62.5%, while failures expose an *efficiency paradox*, spending extra compute exploring authentic environments without making progress. (2) Agent frameworks mainly affect cost-effectiveness rather than the underlying capability ceiling, suggesting that practical agents for real-world terminal environments should reduce exploration friction rather than merely add orchestration complexity. (3) Terminal-Bench scores are only weakly predictive of agent performance on TERMINALWORLD-VERIFIED (Pearson $r = 0.20$), suggesting that existing expert-curated challenges do not fully capture the capabilities needed for real-world terminal workflows. (4) Although TERMINALWORLD tasks are grounded in real-world human recordings, agents often solve them through different valid command paths rather than mimicking the original human workflows, as reflected by an overall median command-set overlap of only 21.4%.

[1] https://asciinema.org/

[2] Here, "VERIFIED" denotes manual review of label correctness, not proof of conformance to a formal specification.

2 Related Work

Terminal Agents. Terminal agents are designed to autonomously issue commands, compose diverse tools, and execute multi-step workflows while interpreting execution feedback within interactive CLI environments. Early frameworks (*e.g.*, SWE-agent [Yang et al., 2024], OpenHands [Wang et al., 2025]) wrap shell commands behind structured tool APIs, allowing agents to resolve tasks through a constrained action schema. More recent native CLI assistants (*e.g.*, Claude Code [Anthropic, 2025], Codex CLI [OpenAI, 2025], Gemini CLI [Google, 2025]) instead expose the raw shell as the primary interface. This enables agents to invoke arbitrary commands and directly observe execution feedback, leading to tighter tool integration on real-world tasks. In parallel, a growing body of work seeks to advance terminal agents by synthesizing large-scale environments and leveraging agent trajectories for training [Zhu et al., 2026, Lin et al., 2026, Gandhi et al., 2026, Wu et al., 2026, Pi et al., 2026], improving the underlying models to drive stronger agentic performance in CLI environments. This rapid proliferation of terminal agents calls for high-quality benchmarks that faithfully assess their real-world capabilities.

Benchmarks for Terminal Agents. Terminal-oriented evaluation has progressed from narrow, single-skill tasks to end-to-end agentic assessment. Earlier benchmarks target isolated command-line abilities, such as translation of natural language into shell commands (*e.g.*, NL2Bash [Lin et al., 2018]) and single-turn command execution with interactive feedback (*e.g.*, InterCode [Yang et al., 2023]). Recent efforts assess agents inside interactive shell sandboxes, covering complex tasks that require multi-step reasoning and tool use, such as Terminal-Bench [Merrill et al., 2026] and LongCLI-Bench [Feng et al., 2026]. Despite this progress, current benchmarks rely on manual curation, which is costly to scale and gravitates toward adversarial puzzles to maximize difficulty. As a result, tasks often diverge from authentic developer workflows, and high scores may not reliably reflect an agent's competence on the routine terminal tasks encountered by practitioners. To address this, our work automates benchmark construction by reverse-engineering in-the-wild terminal recordings, making evaluation grounded in real-world authenticity and scalable as developer practices evolve.

3 TERMINALWORLD: Scalable Data Engine for Real-World Terminal Tasks

As illustrated in Figure 1, we propose TERMINALWORLD, a scalable data engine designed to automatically reverse-engineer terminal tasks[3] from real-world human recordings, which operates through four key steps: **(1) Collecting Human Recordings.** It harvests large-scale, real-world terminal recordings from the `asciinema` platform. **(2) Synthesizing Terminal Tasks.** It reverse-engineers the noisy recordings by inferring the underlying human intent to formalize a task instruction and extracting the core commands as the reference solution. **(3) Reproducing Executable Environments.** It creates and refines an isolated Docker container with the corresponding file system and dependencies, ensuring the core recording workflow can be replayed. **(4) Generating Test Suites.** It implements a trial-based refinement loop to generate and calibrate test suites within the reproduced Docker container.

3.1 Collecting Human Recordings

We collect real-world terminal operation data from `asciinema`, a public platform where practitioners share their terminal session recordings.

Data Retrieval. We systematically index large-scale publicly shared `asciinema` recordings. For each recording, we acquire its transcript text via the standard public download links provided by `asciinema`, along with metadata (*e.g.*, title and description). The transcript offers a high-fidelity log of the real-world terminal execution, capturing the ordered sequence of executed commands and standard outputs. In total, we collect 80,870 real-world terminal recordings by humans.

Data Filtering. While real-world recordings guarantee authenticity, their inherent noise requires systematic filtering. First, for privacy and safety, we exclude recordings exposing *Personally Identifiable Information* (PII), sensitive credentials, or malicious/destructive commands (*e.g.*, `rm -rf *`).

[3] In this paper, we scope *terminal tasks* to pure CLI workflows, where the agent issues shell commands and observes their `stdout/stderr` output. TUI-based interactions are outside our current evaluation scope and left to future work; see Appendix B for details.

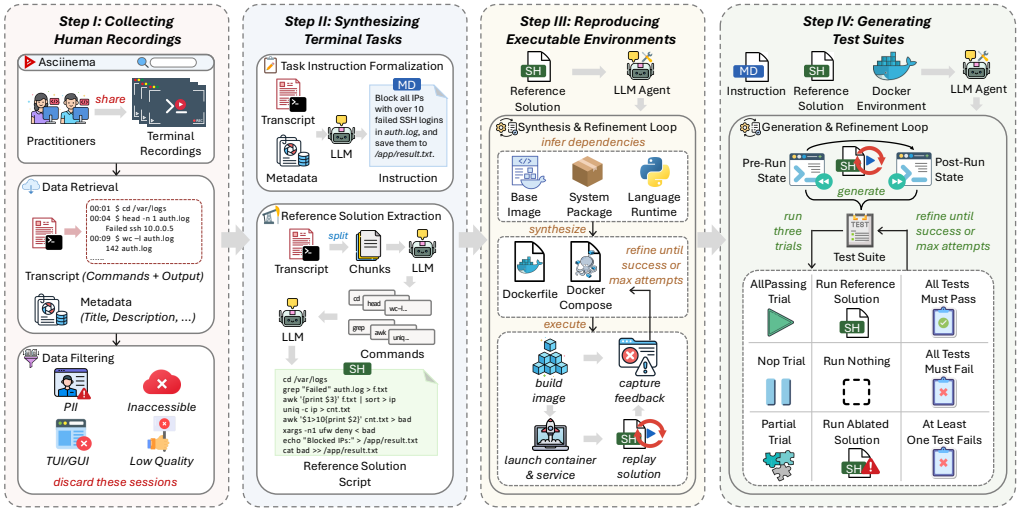


Figure 1: **An overview of the TERMINALWORLD pipeline.** Our data engine automates terminal task synthesis through four key stages: **(1) Collecting Human Recordings** harvests in-the-wild developer operations; **(2) Synthesizing Terminal Tasks** distills an outcome-oriented task instruction and a clean reference solution; **(3) Reproducing Executable Environments** creates and refines an isolated Docker container to replay the core workflow; and **(4) Generating Test Suites** utilizes a trial-based refinement loop to generate and calibrate test suites within the Docker container.

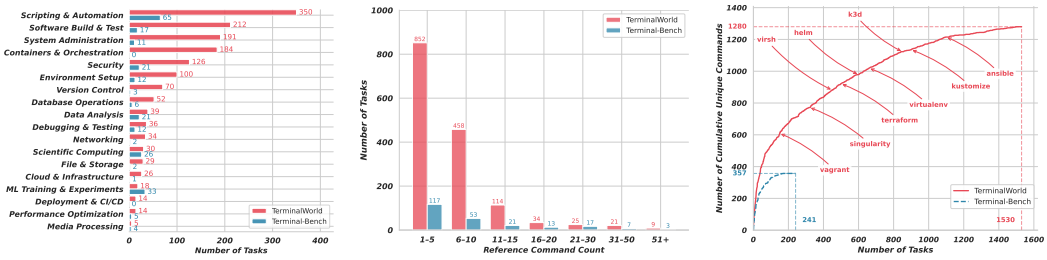
Second, we isolate pure CLI workflows by discarding recordings involving *Text User Interfaces* (TUIs, *e.g.*, `vim`, `nano`, and `emacs`) or GUI applications. Third, we remove recordings incapable of deterministic reproduction in Docker containers, such as those dependent on inaccessible URLs, Windows environments, or proprietary software. Fourth, we eliminate excessively short recordings, typically aborted or trivial sessions. Finally, we employ an LLM (*e.g.*, Claude Sonnet 4.6 [Anthropic, 2026a]) to score recording quality, filtering out opaque or purely exploratory sessions (*e.g.*, repetitive `ls` and `cat` commands). Ultimately, this filtering process yields 9,492 high-quality recordings. Further discussions on ethics, copyright compliance, and privacy mitigation of our data collection are deferred to Appendix A.

3.2 Synthesizing Terminal Tasks

Unlike video recordings that require *Vision-Language Models* (VLMs) to parse with substantial overhead and information loss, the `asciinema` transcript provides a high-fidelity text record of commands and system responses. We purify the transcript by removing typos, failed attempts, and redundant commands, distilling it into an instruction inferred from the underlying human intent and a clean command sequence as the reference solution. By reconstructing the developer's goal and workflow, TERMINALWORLD ensures that each synthesized task is authentic to real-world usage.

Task Instruction Formalization. We leverage an LLM (*e.g.*, Claude Sonnet 4.6 [Anthropic, 2026a]) to synthesize natural language instructions by distilling the developer's core intent from the transcript alongside the title and description. The instruction specifies the task goal in concise, outcome-oriented language: it must describe the expected *final state*, not the path to reach it. We explicitly prohibit procedural phrasing, specific commands, and step-by-step enumerations, preventing solution-specific hints from leaking into the task. To establish a testable contract, the instruction must explicitly specify required output paths (*e.g.*, `/app/result.txt`) and strict structural formats, while omitting arbitrary internal artifacts invented by the original developer, such as custom labels or print banners.

Reference Solution Extraction. We employ an LLM (*e.g.*, Claude Sonnet 4.6 [Anthropic, 2026a]) to extract a clean, executable bash script from the raw transcript as the ground-truth reference solution for the formalized task instruction. For long transcripts with verbose system outputs, we first split the transcript into chunks to filter execution noise and isolate valid commands. We then merge the extracted commands, remove duplicates, and assemble a coherent solution workflow. Since source recordings are mostly pre-planned showcases rather than messy debugging sessions, this process can recover clean reference solutions without being overwhelmed by excessive human trial-and-error.



(a) Task Category Distribution. (b) Task Complexity Distribution. (c) Unique Command Coverage.

Figure 2: **Statistical comparison of 1,530 TERMINALWORLD tasks and 241 unique Terminal-Bench tasks.** (a) highlights that TERMINALWORLD captures diverse real-world workflows (*e.g.*, container orchestration, CI/CD) severely underrepresented in expert-curated benchmarks. (b) shows a natural spectrum that mirrors everyday terminal usage, from brief operations to multi-step workflows. (c) reveals an extensive vocabulary of 1,280 commands (91% absent from Terminal-Bench) that can assess an agent's capability to discover and wield diverse utilities.

Consistent with the outcome-oriented instruction design, the script is constrained to redirect its final results from transient terminal outputs to explicit file paths (*e.g.*, `/app/result.txt`). This ensures the reference workflow is idempotent and its outcome is deterministically captured in the filesystem. We generate instructions and reference solutions for all 9,492 filtered recordings.

3.3 Reproducing Executable Environments

Raw in-the-wild recordings are inherently volatile, as they depend on the original developer's system state, implicit toolchains, and transient resources. Thus, a basic container for isolated command execution is insufficient; we instead reverse-engineer the dependency context required to replay the recorded terminal workflow. Without a faithful executable environment, it is difficult to assess the quality of synthesized tasks, such as task solvability and test soundness. On the other hand, by encapsulating complex dependencies into an isolated Docker sandbox, we eliminate environmental indeterminism for rigorous agent evaluation.

Environment Synthesis. We leverage an LLM agent (*e.g.*, Claude Code [Anthropic, 2025]) to synthesize a `Dockerfile` (and `docker-compose.yaml` for multi-service tasks) by inferring required dependencies from the reference solution (*e.g.*, base images, system packages, and language runtimes). When the recording includes an external repository link, the agent clones and scans the project to infer environment requirements. To guarantee the environment's authenticity, we eliminate hallucinated dependencies by explicitly prohibiting fake binaries, stubbed dependencies, and bypasses of real software installation.

Execution-Based Refinement Loop. Since static synthesis by LLMs is prone to dependency conflicts and missing hidden packages, TERMINALWORLD equips the agent with an execution-feedback loop to refine the environment. The agent builds an image from the synthesized Dockerfile, parses build logs to diagnose compilation errors or package manager failures, and iteratively repairs the Dockerfile when needed. It then runs a Docker container from the image, launching any required auxiliary services via `docker-compose.yaml` when necessary. The agent executes the reference solution script in a persistent shell session, aiming to replay the original terminal recording. The environment is considered reproduced only when the script executes successfully, as indicated by *an exit code of 0*. Otherwise, any runtime error, such as missing libraries, unconfigured environment variables, or unrecognized commands, is fed back to the agent for targeted repairs to the `Dockerfile` or `docker-compose.yaml`. Recordings that remain irreproducible within our computational budget or are reliant on inaccessible resources are discarded. Ultimately, this loop reproduces executable environments for 5,035 terminal tasks.

3.4 Generating Test Suites

The raw recordings naturally capture the execution trajectory, but they lack explicit tests to automatically judge whether the underlying goal is achieved. Thus, we introduce an automated execution-feedback loop that generates and refines a test suite within the reproduced Docker environment.

Table 1: **Performance of frontier LLMs on TERMINALWORLD under the Terminus-2 scaffold.** †Values in parentheses denote resolved rate = pass / (total - errors), where errors are tasks on which the Harbor evaluation harness failed before the agent could attempt the task (see Appendix C.1 for error details). The results suggest that frontier LLMs still struggle with real-world terminal tasks, often falling into an *efficiency paradox* where more compute and attempts fail to yield better performance (see Table 4 for full details). Overcoming this requires genuine reasoning abilities to navigate open-ended action spaces.

Model	Type	Pass Rate (%)†	Avg. Turns	Avg. Tokens (K)	Avg. Time (min)	Cost ($)	$/Pass
Claude Opus 4.7	Closed	62.5 (64.8)	16.7	261.5	3.6	63.47	0.51
Gemini 3.1 Pro	Closed	55.0 (57.0)	10.6	173.2	4.0	56.82	0.52
GPT-5.5	Closed	53.5 (56.6)	14.8	499.0	3.0	100.28	0.94
Kimi K2.6	Open	57.5 (60.2)	16.9	289.6	5.5	17.68	0.15
GLM 5.1	Open	57.0 (58.5)	15.5	189.1	7.0	18.24	0.16
Qwen3.6-Max-Preview	Open	54.0 (56.8)	12.5	172.2	6.9	21.44	0.20
DeepSeek-V4-Pro	Open	50.0 (52.1)	20.1	398.0	9.4	17.35	0.17
MiniMax M2.7	Open	49.0 (50.8)	27.5	683.6	9.3	10.95	0.11

Test Suite Generation. Based on the formalized instruction (equivalently, the *task specification*) and the reference solution in Section 3.2, test generation reduces to solving the test oracle problem [Barr et al., 2015]. Concretely, we synthesize a suite of test assertions to assess whether the expected final state is achieved. These assertions typically target persistent artifacts, such as file existence, content hashes, or structured outputs in `/app/result.txt`. However, LLM-generated tests without execution feedback are vulnerable to hallucination and misalignment, where ambiguous assertions easily diverge from the actual system state produced by the reference solution. To resolve this, we adopt an LLM agent (*e.g.*, Claude Code [Anthropic, 2025]) to capture snapshots of the pre- and post-execution state in the reproduced Docker environment, recording the true filesystem changes caused by the reference solution. Using these state deltas, the agent generates and calibrates the test suite to align with the actual final state. During generation, we explicitly instruct the agent to avoid brittle checks, such as exact string matching on transient outputs or non-deterministic execution values (*e.g.*, timestamps, temporary process IDs).

Trial-Based Refinement Loop. Once generated, the agent iteratively refines the test suite through three execution trials in fresh, isolated containers to eliminate false negatives and false positives:

- ▷ **AllPassing Trial** executes the reference solution and requires all tests to pass. It prevents false negatives, where overly rigid tests reject correct solutions, thereby ensuring task *solvability*.
- ▷ **Nop Trial** runs nothing, leaving the container in its initial state, and requires all tests to fail. It prevents false positives where tasks are solved by an empty state, thus ensuring task *non-triviality*.
- ▷ **Partial Trials** execute incomplete solutions derived by truncating or ablating the reference solution, and require at least one test to fail. This is a stringent check against nuanced false positives, ensuring that the tests can reject incomplete solutions, enhancing the *discriminability* of the test suite.

A task is admitted only if its test suite satisfies all three trials. If any trial fails, the agent diagnoses the trial outputs, applies targeted fixes to the test suite, and reruns the loop. When necessary, adjustments to the test suite are synchronized back to the instruction, ensuring that the instruction explicitly specifies the state being evaluated. A task is discarded if its test suite cannot be repaired to pass all three trials within the computational budget. Ultimately, this refinement loop distills 1,530 validated terminal tasks as the full TERMINALWORLD benchmark.

4 The TERMINALWORLD Benchmark

4.1 Dataset Statistics

Given that the terminal recordings capture naturally occurring workflows and are voluntarily uploaded by developers, the resulting TERMINALWORLD dataset is inherently self-curated and human-vetted. We obtain both of these highly desirable properties *by construction*, ensuring comprehensive coverage of the tools, configurations, and problem-solving strategies encountered in everyday terminal usage.

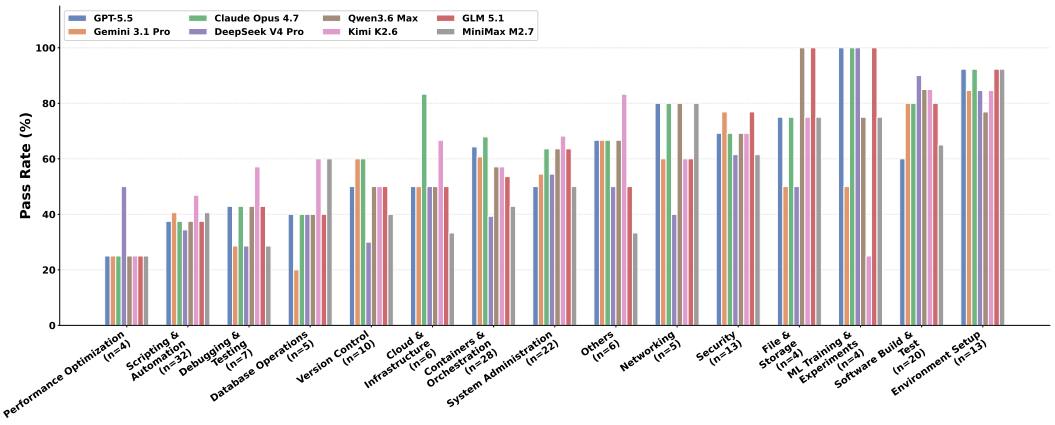


Figure 3: **Performance of frontier LLMs across terminal task categories.** The results indicate that LLMs still struggle with complex tasks involving performance optimization, scripting & automation, and debugging & testing, showing domain-specific blind spots and a lack of general tool-use ability.

We characterize the benchmark's representativeness, diversity, and complexity by comparing it against Terminal-Bench [Merrill et al., 2026], a widely used benchmark constructed by human experts.

Task Category Distribution. Figure 2a details the category distribution of the 1,530 TERMINALWORLD tasks alongside the 241 unique tasks de-duplicated across both Terminal-Bench 1.0 and 2.0. TERMINALWORLD covers 18 real-world terminal categories (see Table 3 for details), introducing tasks that are ubiquitous in modern developer routines yet severely underrepresented in Terminal-Bench, such as container orchestration, CI/CD pipelines, and cloud infrastructure management. Beyond the artificial boundaries of expert curation, TERMINALWORLD provides a comprehensive and authentic snapshot of the diverse workflows that practitioners actually execute in the wild.

Task Complexity Distribution. Figure 2b compares task complexity based on the number of commands required in the reference solutions. Derived from real-world operations, TERMINALWORLD exhibits a natural complexity spectrum spanning from a few commands to workflows exceeding 50 steps, providing a larger scale across all length intervals. The high density of short workflows accurately reflects everyday terminal usage, where developers frequently execute brief command sequences, while TERMINALWORLD still captures the long tail of complex, multi-step scenarios.

Unique Command Coverage. Figure 2c illustrates the cumulative distribution of unique commands appearing in the reference solutions across terminal tasks. TERMINALWORLD's command coverage climbs steadily with task scale, encompassing an extensive vocabulary of 1280 unique commands. Notably, 91% of these commands are absent from Terminal-Bench, spanning diverse tool categories such as environment management (*e.g.*, `vagrant`, `virtualenv`), infrastructure configuration (*e.g.*, `terraform`, `ansible`), and Kubernetes orchestration (*e.g.*, `k3d`, `kustomize`). This broad command coverage reflects the rich diversity of tools developers employ in real-world workflows, establishing TERMINALWORLD as an authentic testbed for assessing an agent's capability to discover and wield diverse utilities to solve in-the-wild terminal problems.

4.2 The VERIFIED Subset

From the full benchmark, we curate a VERIFIED subset of 200 representative tasks by carefully balancing diversity and complexity. It spans diverse real-world task categories in Figure 2a, while prioritizing tasks with longer command sequences and non-trivial domain-specific tools.

This subset undergoes manual review by four authors, each with over three years of terminal-based software development experience. Annotators build the Docker image, enter the containerized environment, and execute the reference solution step by step, repairing the `Dockerfile` or `docker-compose.yaml` if necessary. They also cross-review artifact alignment to ensure that every test constraint is stated in the instruction and that the tests accurately check the persistent artifacts produced by the reference solution. Tasks that pass this human verification form the highest-fidelity evaluation core of TERMINALWORLD. While the full set of 1,530 tasks provides a representative

Table 2: **Performance of terminal agents on TERMINALWORLD.** †Values in parentheses denote resolved rate = pass / (total - errors), where errors are tasks on which the Harbor evaluation harness failed before the agent could attempt the task (see Appendix C.2 for error details). The results suggest that agent frameworks drive cost-effectiveness rather than shifting the model's capability ceiling (see Table 5 for full details). Agent designs in real-world environments should prioritize minimizing exploration friction to discover solution paths economically rather than inflating model capabilities.

Agent	Model	Pass Rate (%)†	Avg. Turns	Avg. Tokens (K)	Avg. Time (min)	Cost ($)	$/Pass
Terminus-2	Claude Opus 4.7	62.5 (64.8)	16.7	261.5	3.6	63.47	0.51
Claude Code	Claude Opus 4.7	58.0 (60.7)	18.0	667.7	6.0	105.12	0.91
mini-SWE-agent	Claude Opus 4.7	52.0 (59.8)	17.1	206.4	3.9	56.94	0.55
OpenHands	Claude Opus 4.7	45.0 (57.3)	21.9	410.9	5.3	371.21	4.12
Terminus-2	Gemini 3.1 Pro	55.0 (57.0)	10.6	173.2	4.0	56.82	0.52
Gemini CLI	Gemini 3.1 Pro	56.0 (59.6)	41.5	694.7	6.3	85.90	0.77
Terminus-2	GPT-5.5	53.5 (56.6)	14.8	499.0	3.0	100.28	0.94
Codex CLI	GPT-5.5	48.5 (56.1)	29.3	431.4	1.6	128.80	1.33

snapshot of in-the-wild terminal usage, this VERIFIED subset offers a rigorous and challenging testbed for benchmarking frontier models and agents on complex, real-world terminal tasks.

Evaluation Setup. We use TERMINALWORLD-VERIFIED to evaluate three frontier closed-source models (*i.e.*, Claude Opus 4.7 [Anthropic, 2026b], GPT-5.5 [OpenAI, 2026], Gemini 3.1 Pro [Google, 2026]) alongside several leading open-weight counterparts (*i.e.*, DeepSeek-V4-Pro [DeepSeek, 2026], Qwen3.6-Max-Preview [Alibaba, 2026], Kimi K2.6 [Moonshot AI, 2026], GLM-5.1 [Z.ai, 2026], and MiniMax M2.7 [MiniMax, 2026]). We also assess three general-purpose terminal agents (*i.e.*, Terminus 2 [Merrill et al., 2026], mini-SWE-agent [Yang et al., 2024], OpenHands [Wang et al., 2025]) and three model-native CLI assistants (*i.e.*, Claude Code [Anthropic, 2025], Codex CLI [OpenAI, 2025], Gemini CLI [Google, 2025]). All evaluations are conducted using Terminal-Bench's standard Harbor harness [Merrill et al., 2026]. Detailed evaluation settings are provided in Appendix C.

5 Benchmarking Results on TERMINALWORLD-VERIFIED

We evaluate frontier LLMs and terminal agents on TERMINALWORLD-VERIFIED to assess model capability and the impact of agent framework choices. We further compare model performance on TERMINALWORLD-VERIFIED against Terminal-Bench 2.0 [Merrill et al., 2026] and analyze how agent behavior diverges from the original human workflows.

5.1 TERMINALWORLD-VERIFIED Challenges Large Language Models

As shown in Table 1, we assess frontier LLMs using Terminus-2 [Merrill et al., 2026], the standardized agent scaffold from Terminal-Bench, to isolate model capability from agent framework differences.

Real-World Terminal Tasks Continue to Challenge SOTA LLMs. Overall, all evaluated models achieve modest pass rates (49.0%–62.5%, avg. 54.8%), with even the best model (*i.e.*, Claude Opus 4.7) failing on over one-third of the tasks, confirming that the real-world terminal tasks in TERMINALWORLD pose a substantial challenge to frontier LLMs. Interestingly, open-weight models (*e.g.*, Kimi K2.6 and GLM 5.1) demonstrate highly competitive capabilities, rapidly closing the gap with and even outperforming closed-source counterparts (*e.g.*, Gemini 3.1 Pro and GPT-5.5). Furthermore, they incur significantly lower costs (avg. $17.13) than closed-source models (avg. $70.82), yielding a $4\times$–$8\times$ advantage in cost-effectiveness ($/Pass).

LLMs Exhibit an Efficiency Paradox in Real-World Terminal Environments. Higher resource consumption does not correlate with better outcomes. At a statistical level, task success rates show weak negative correlations with both turn count (Pearson $r = -0.49$) and token usage ($r = -0.62$). This trend is evident in specific models: GPT-5.5 and MiniMax M2.7 consume substantially more tokens and turns than most peers, yet achieve lower pass rates. Our trajectory analysis further shows that failed attempts are disproportionately expensive. Across all models, failed attempts consume

on average 3.3× more tokens and 1.4× more time than successful ones, monopolizing 63% of total evaluation costs despite representing only 43% of attempts.

This *efficiency paradox* reflects a unique challenge of TERMINALWORLD: real-world terminal tasks expose open-ended action spaces filled with nuanced dependencies and domain-specific tools. Without reliable planning and stopping criteria, agents cannot simply brute-force their way to a solution; they may keep exploring the authentic environment, spending more compute without making progress toward the correct outcome.

TERMINALWORLD Exposes Polarized Tool-Use Capabilities. As shown in Figure 3, model performance varies sharply across task domains. LLMs perform well on environment setup (avg. 87.5%) and software build & test (avg. 78.1%), but struggle with performance optimization (avg. 28.1%), scripting & automation (avg. 39.1%), and debugging & testing (avg. 39.3%). Moreover, no single model dominates across all domains: Claude Opus 4.7 performs strongly on cloud & infrastructure (83.3%) and containers & orchestration (67.9%), while Kimi K2.6 outperforms it on scripting & automation (46.9% vs. 37.5%). These results show that current LLMs still lack robust tool-use ability across diverse real-world CLI workflows, a blind spot exposed by TERMINALWORLD's broad task coverage.

5.2 Agentic Leaderboarding on TERMINALWORLD-VERIFIED

Table 2 evaluates general terminal agents and model-native CLI assistants across various LLMs. Note that the Harbor harness [Merrill et al., 2026] is natively compatible with Terminus-2. For other agents, the Harbor harness occasionally triggers infrastructural errors (e.g., container initialization timeouts or dependency conflicts) *before* the agent can even attempt the task (see Appendix C.2 for error details). Thus, we also report *pass / (total - errors)* in parentheses for fair comparison.

Agent Frameworks Drive Cost-Effectiveness More Than Capabilities. The results suggest that agent frameworks affect performance, but do not substantially change the underlying model's capability ceiling. Their larger impact is on *cost-effectiveness*. For example, with Claude Opus 4.7, pass rates remain relatively close across frameworks, yet Terminus-2 and mini-SWE-agent achieve comparable performance at roughly $60 total cost ($0.51–$0.55 per pass), using fewer turns, tokens, and time. This suggests that practical agent design for real-world terminal environments should prioritize reducing exploration friction rather than merely adding orchestration complexity, helping models find effective solution paths earlier and more economically.

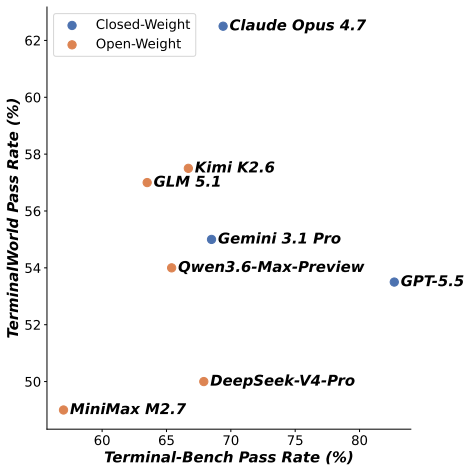


Figure 4: **Scatter of Terminal-Bench 2.0 score vs. TERMINALWORLD-VERIFIED score.** The weak score correlation (**Pearson $r = 0.20$**) highlights a disconnect: LLMs that dominate expert-curated challenges still struggle on real-world workflows, indicating that TERMINALWORLD assesses a distinct terminal capability.

5.3 TERMINALWORLD vs. Terminal-Bench

To test whether Terminal-Bench performance transfers to real-world terminal workflows, Figure 4 plots each model's Terminal-Bench 2.0 pass rate against its TERMINALWORLD-VERIFIED pass rate. We use the officially reported Terminal-Bench 2.0 scores as each model's reference performance; details are provided in Appendix C.3.

Terminal-Bench Scores Transfer Weakly to Real-World Performance on TERMINALWORLD. The results reveal a clear gap between expert-curated challenges and authentic terminal workflows. As shown in Figure 4, TERMINALWORLD-VERIFIED presents a harder real-world evaluation setting: models score 57.0%–82.7% on Terminal-Bench 2.0, but only 49.0%–62.5% on TERMINALWORLD-VERIFIED. Moreover, model rankings are notably reshuffled across the two benchmarks. GPT-5.5, despite scoring near 83% on Terminal-Bench 2.0, reaches only 53.5% on TERMINALWORLD-VERIFIED. In contrast, open-weight models such as Kimi K2.6 achieve moderate Terminal-Bench

2.0 scores (66.7%) but remain competitive on TERMINALWORLD-VERIFIED (∼57%), outperforming GPT-5.5 and Gemini 3.1 Pro. Only Claude Opus 4.7 performs strongly across both benchmarks. This rank reshuffling yields a weak correlation between Terminal-Bench 2.0 and TERMINALWORLD-VERIFIED performance (**Pearson $\mathbf{r} = 0.20$**). Thus, TERMINALWORLD captures terminal capabilities beyond Terminal-Bench, showing that expert-curated scores can overestimate competence in authentic terminal environments.

5.4 Agents vs. Humans

Since each TERMINALWORLD task is derived from a real-world human recording, we can compare agent behavior against the original human workflow. We examine whether agents follow similar command paths to humans or reach the same outcome through alternative strategies. For each successfully solved task, we compute the Jaccard similarity between the command sets in the agent trajectory and the reference solution, after stripping flags and arguments. Figure 5 shows the resulting distribution of command-set overlap across models. More detailed analysis is provided in Appendix C.4.

Agents Solve Tasks Through Alternative Command Paths, Not Mimicking Humans. The median overlap is only **21.4%**, meaning agents typically reach the correct outcome via a different set of commands than the human practitioner used. For example, on a network packet analysis task (extract HTTP Basic Auth credentials from a pcap file), the reference solution uses `ettercap` to replay and parse the capture; agents instead use `tshark` with Python to parse the pcap directly. On a disk image modification task, the reference solution manually creates device nodes with `mknod` to access partitions; agents use standard tools (`fdisk`, `mkfs.ext4`, `mount`) to achieve the same result. In both cases, the agent and reference command sets are disjoint, yet the outcome-based verifier accepts both as correct solutions. This reflects a core design choice of TERMINALWORLD: tasks specify the required end state rather than the steps to reach it, so any correct path is accepted.

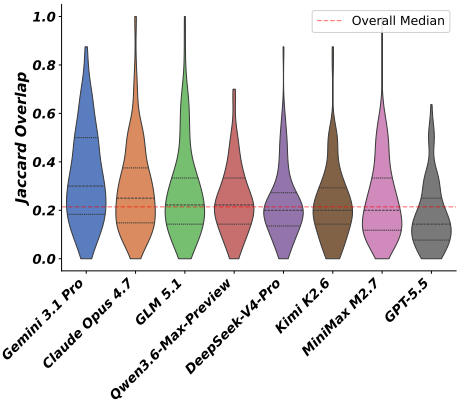


Figure 5: **Command-Set Overlap of Agents with Human Workflows.** While tasks are derived from real-world human recordings, agents often reach the correct outcome through different command paths, with **21.4%** median command-set overlap.

6 Conclusion

We presented TERMINALWORLD, a scalable data engine that reverse-engineers high-fidelity terminal evaluation tasks from in-the-wild human recordings. From 80,870 raw `asciinema` recordings, TERMINALWORLD has distilled 1,530 validated tasks, of which 200 tasks have also been human-verified (VERIFIED subset). Benchmarking frontier LLMs and terminal agents on TERMINALWORLD-VERIFIED shows that current systems still struggle with real-world terminal workflows, and that performance on expert-curated benchmarks does not fully transfer to TERMINALWORLD-VERIFIED. As developer practices evolve and new terminal recordings accumulate, TERMINALWORLD can be re-run to provide a continually updated testbed for terminal agents. We hope TERMINALWORLD helps the community move beyond static, expert-curated benchmarks toward evaluating and developing agents that are reliable in real-world, evolving terminal workflows.

Acknowledgments

We gratefully acknowledge Amazon Web Services (AWS) for supporting this research through AWS credits and Bedrock access support. These resources were used in part for large-scale task synthesis, benchmark construction, and model evaluation in TERMINALWORLD. We also thank the AWS team supporting UCL for their guidance and support.

References

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 Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang, editors, *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, and et al. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

Anthropic. Claude code overview - claude code docs. https://code.claude.com/docs/en/overview, 2025. Accessed: 2026-05-01.

OpenAI. Codex cli. https://developers.openai.com/codex/cli, 2025. Accessed: 2026-05-01.

Google. Build, debug & deploy with ai | gemini cli. https://geminicli.com/, 2025. Accessed: 2026-05-01.

Mike A Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Kumar Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Kwesi Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Jenia Jitsev, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

Yukang Feng, Jianwen Sun, Zelai Yang, Jiaxin Ai, Chuanhao Li, Zizhen Li, Fanrui Zhang, Kang He, Rui Ma, Jifan Lin, Jie Sun, Yang Xiao, Sizhuo Zhou, Wenxiao Wu, Yiming Liu, Pengfei Liu, Yu Qiao, Shenglin Zhang, and Kaipeng Zhang. Longcli-bench: A preliminary benchmark and study for long-horizon agentic programming in command-line interfaces. *CoRR*, abs/2602.14337, 2026. doi: 10.48550/ARXIV.2602.14337. URL https://doi.org/10.48550/arXiv.2602.14337.

Kaijie Zhu, Yuzhou Nie, Yijiang Li, Yiming Huang, Jialian Wu, Jiang Liu, Ximeng Sun, Zhenfei Yin, Lun Wang, Zicheng Liu, Emad Barsoum, William Yang Wang, and Wenbo Guo. Termigen: High-fidelity environment and robust trajectory synthesis for terminal agents. *CoRR*, abs/2602.07274, 2026. doi: 10.48550/ARXIV.2602.07274. URL https://doi.org/10.48550/arXiv.2602.07274.

Yusong Lin, Haiyang Wang, Shuzhe Wu, Lue Fan, Feiyang Pan, Sanyuan Zhao, and Dandan Tu. Cli-gym: Scalable CLI task generation via agentic environment inversion. *CoRR*, abs/2602.10999, 2026. doi: 10.48550/ARXIV.2602.10999. URL https://doi.org/10.48550/arXiv.2602.10999.

Kanishk Gandhi, Shivam Garg, Noah D Goodman, and Dimitris Papailiopoulos. Endless terminals: Scaling rl environments for terminal agents. *arXiv preprint arXiv:2601.16443*, 2026.

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

Siwei Wu, Yizhi Li, Yuyang Song, Wei Zhang, Yang Wang, Riza Batista-Navarro, Xian Yang, Mingjie Tang, Bryan Dai, Jian Yang, and Chenghua Lin. Large-scale terminal agentic trajectory generation from dockerized environments. *CoRR*, abs/2602.01244, 2026. doi: 10.48550/ARXIV.2602.01244. URL https://doi.org/10.48550/arXiv.2602.01244.

Anthropic. Introducing claude sonnet 4.6. https://www.anthropic.com/news/claude-sonnet-4-6, 2026a. Accessed: 2026-05-01.

Anthropic. Introducing claude opus 4.7. https://www.anthropic.com/news/claude-opus-4-7, 2026b. Accessed: 2026-05-01.

OpenAI. Introducing gpt-5.5. https://openai.com/index/introducing-gpt-5-5/, 2026. Accessed: 2026-05-01.

Google. Gemini 3.1 pro: Best for complex tasks and bringing creative concepts to life. https://deepmind.google/models/gemini/pro/, 2026. Accessed: 2026-05-01.

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. NL2Bash: A corpus and semantic parser for natural language interface to the linux operating system. In Nicoletta Calzolari, Khalid Choukri, Christopher Cieri, Thierry Declerck, Sara Goggi, Koiti Hasida, Hitoshi Isahara, Bente Maegaard, Joseph Mariani, Hélène Mazo, Asuncion Moreno, Jan Odijk, Stelios Piperidis, and Takenobu Tokunaga, editors, *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan, May 2018. European Language Resources Association (ELRA). URL https://aclanthology.org/L18-1491/.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/4b175d846fb008d540d233c188379ff9-Abstract-Datasets_and_Benchmarks.html.

Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015. doi: 10.1109/TSE.2014.2372785.

DeepSeek. Deepseek v4 preview release. https://api-docs.deepseek.com/news/news260424, 2026. Accessed: 2026-05-01.

Alibaba. Qwen3.6-max-preview: Smarter, sharper, still evolving. https://qwen.ai/blog?id=qwen3.6-max-preview, 2026. Accessed: 2026-05-01.

Moonshot AI. Kimi k2.6: From code to creation, from one to many. https://www.kimi.com/ai-models/kimi-k2-6, 2026. Accessed: 2026-05-01.

Z.ai. Glm-5.1: Towards long-horizon tasks. https://z.ai/blog/glm-5.1, 2026. Accessed: 2026-05-01.

MiniMax. Minimax m2.7: Early echoes of self-evolution. https://www.minimax.io/news/minimax-m27-en, 2026. Accessed: 2026-05-01.

A Broader Impact and Ethical Considerations

In developing the TERMINALWORLD benchmark, we adhere strictly to ethical data practices and copyright compliance, specifically addressing the challenges inherent in sourcing in-the-wild, user-generated terminal recordings.

Data Sourcing and Consent. TERMINALWORLD sources data from `asciinema`, a platform where practitioners publicly broadcast terminal sessions for education and collaboration. Our framework essentially automates this intended use: it "views" these recordings to learn from them, just as a human user would. To ensure ethical compliance at scale, we strictly retrieve publicly listed transcripts using the platform's standard download mechanisms in full accordance with `robots.txt` directives. We collect only the `.txt` transcripts and their coupled metadata (*e.g.*, titles and environmental specs), which are intentionally exposed by the platform UI and are essential for evaluating task correctness. All data is utilized strictly for non-commercial academic research.

Copyright, Rehosting, and the Right to be Forgotten. A critical ethical mandate is respecting the intellectual property of individual creators and the platform's Terms of Service. To definitively prevent the unauthorized redistribution (rehosting) of copyrighted material, TERMINALWORLD *does not* distribute the original human transcripts or native `.cast` files. Instead, our publicly released benchmark tasks contain only the synthesized evaluation artifacts (instructions, reference solutions, execution environments, and test suites) accompanied by direct hyperlinks pointing back to the original recordings on `asciinema`. This pointer-based architecture ensures robust compliance: it inherently respects the *"Right To Be Forgotten"* of recording authors. If a creator removes their recording from the host platform, our reference hyperlink naturally expires, ensuring no unauthorized local copies persist in our dataset.

Privacy and PII Mitigation. Terminal recordings inherently carry the risk of exposing sensitive system information or Personally Identifiable Information (PII). As detailed in our data filtering methodology (Section 3.1), we mitigate this risk at the source. Our automated pipeline actively filters out recordings containing exposed PII, sensitive credentials (*e.g.*, API keys, AWS tokens), or operations demonstrating malicious intent before any task synthesis occurs.

Human Quality Verification. To ensure the integrity of our evaluation core, the final VERIFIED subset of TERMINALWORLD tasks underwent rigorous manual verification. This process was conducted entirely by the paper's authors and expert collaborating researchers, ensuring that no uncompensated or under-compensated crowdsourced labor was utilized.

Platform Mutual Benefit. Rather than exploiting the host platform, TERMINALWORLD is designed to create a mutually beneficial ecosystem. By relying on direct hyperlinks rather than rehosted files, our benchmark actively drives community traffic back to `asciinema`. Furthermore, this work highlights the broader value of platforms such as `asciinema` in the LLM era: they preserve authentic human interaction traces that can support the evaluation and improvement of AI agents. This creates opportunities for platform maintainers and research communities to collaborate on responsible data access, curated recording collections, and benchmark development.

B Terminal Task Scope and Categories in TERMINALWORLD

TERMINALWORLD operationalizes a terminal task as a *pure CLI* workflow: the agent issues shell commands and observes their `stdout`/`stderr` output to decide the next action, without invoking any full-screen TUI application. During data collection (Section 3.1), all recordings involving terminal-based editors (*e.g.*, `vim`, `nano`, `emacs`) or TUI programs are discarded. Evaluating TUI-based interactions is left to future work.

Categories. TERMINALWORLD covers 18 real-world terminal task categories derived from the purposes of the source recordings. Each task receives a *single* category label reflecting its *primary goal*, *i.e.*, the final state the practitioner wants to achieve, rather than the incidental tools used along the way. For example, a task that writes a shell script to automate nightly database backups is categorised as *Database Operations*, not *Scripting & Automation*, because the script is merely the means (the *journey*), not the end goal. The full taxonomy with per-category scope boundaries and disambiguation rules is released with the benchmark. Table 3 lists all 18 categories with descriptions and task counts.

Table 3: The 18 task categories in TERMINALWORLD, grouped by practitioner goals. Representative CLI tools are shown in parentheses.

Category	Description	#
Infrastructure & Operations		
System Administration	Configure, maintain, or repair OS-level infrastructure: services, users, disk partitions, kernel parameters, and boot (`systemctl`, `fdisk`, `sysctl`).	191
Networking	Configure network services, diagnose connectivity, set up VPN tunnels, or measure network bandwidth (`dig`, `wg`, `haproxy`, `iperf3`).	34
File & Storage	Manage, backup, archive, sync, or transfer files as the primary objective (`rsync`, `rclone`, `tar`, `restic`).	29
Software Development		
Software Build & Test	Compile, test, lint, or package a code project via CLI build toolchains (`make`, `cargo`, `gradle`, `pytest`).	212
Version Control	Manage code history, resolve conflicts, or rewrite repository history as the primary goal (`git rebase`, `git bisect`, `git filter-repo`).	70
Environment Setup	Prepare a development environment: virtual envs, dependency installation, SDK version management, dotfiles (`conda`, `nvm`, `pyenv`).	100
Debugging & Testing	Diagnose a functional failure or verify correctness—crash analysis, regression bisection, log triage (`strace`, `gdb`, `valgrind`).	36
Performance Optimization	Measure, profile, or benchmark execution performance—flame graphs, latency reports, throughput benchmarks (`perf`, `hyperfine`, `py-spy`).	14
Cloud & Container Stack		
Containers & Orchestration	Build, run, or manage containers and clusters; the container layer (images, pods, registries) is the primary subject (`docker`, `kubectl`, `helm`).	184
Cloud & Infrastructure	Provision or manage cloud resources via provider CLIs or IaC tools (`aws`, `gcloud`, `az`, `terraform`).	26
Deployment & CI/CD	Deliver an application to a target environment or configure a CI/CD pipeline (`gh workflow`, `argocd`, `fly deploy`).	14
Data Processing & Automation		
Scripting & Automation	Write or run scripts, pipelines, or automation routines; includes driving stdin-consuming programs via pipe or heredoc (`bash`, `awk`, `jq`, `expect`).	350
Data Analysis	Extract insights or a transformed dataset from structured data—log analysis, CSV aggregation, statistical summaries (`pandas`, `duckdb`, `csvkit`).	39
Database Operations	Query, administer, migrate, backup, or tune a database; the database itself is the primary subject (`psql`, `mongosh`, `alembic`).	52
ML Training & Experiments	Train, fine-tune, evaluate, or run inference on a machine learning model (`torchrun`, `huggingface-cli`, `svm-train`, `vllm serve`).	18
Specialized Domains		
Security	Find vulnerabilities, exploit systems, capture CTF flags, reverse-engineer binaries, or audit security posture (`nmap`, `gdb`, `hashcat`, `radare2`).	126
Scientific Computing	Perform domain-specific computation: bioinformatics pipelines, physics simulations, formal proofs, HPC jobs (`samtools`, `gmx`, `sbatch`, `lean`).	30
Media Processing	Transform, convert, or extract content from audio, video, or image files (`ffmpeg`, `sox`, `convert`, `yt-dlp`).	5
Total		**1,530**

C Detailed Experimental Setup and Analysis

C.1 Benchmarking Large Language Models

All models in this experiment are evaluated using Terminus-2 [Merrill et al., 2026], Harbor's native agent scaffold, to ensure a fair comparison that isolates model capability from agent framework differences. For models supporting configurable reasoning effort, we default to the providers' default settings (*e.g.*, "medium" for GPT-5.5 and "high" for Claude Opus 4.7) to ensure standardized comparison. All experiments were run on a CPU-based server that executed the Harbor harness and Docker containers locally, while LLM inference was performed through the corresponding provider APIs. We did not train or fine-tune any models, and no local GPU computation was required.

Table 4: **Detailed per-model statistics for the LLM benchmarking experiment under Terminus-2.** Timeout (TO) is a subset of Fail; Err counts tasks where the harness failed before the agent could run. For Avg. Turns, Avg. Tokens, Avg. Time, and $/task, the overall average is shown first, with per-subset averages for passed / failed tasks in parentheses; averages are computed only over tasks where the agent ran. †Resolved rate = pass / (total − errors).

Model	Type	Pass Rate (%)†	Outcomes				Avg. Turns (Pass / Fail)	Avg. Tokens (K) (Pass / Fail)	Avg. Time (min) (Pass / Fail)	Cost ($)	$/Pass	$/task (Pass / Fail)
			Pass	Fail	TO	Err						
Claude Opus 4.7	Closed	62.5 (64.8)	125	68	2	7	16.7 (12.0 / 25.0)	261.5 (139 / 481)	3.6 (3.7 / 3.6)	63.47	0.51	0.23 / 0.39
Gemini 3.1 Pro	Closed	55.0 (57.0)	110	83	0	7	10.6 (9.1 / 12.5)	173.2 (84 / 289)	4.0 (3.6 / 4.4)	56.82	0.52	0.20 / 0.39
GPT-5.5	Closed	53.5 (56.6)	107	82	3	11	14.8 (7.9 / 24.1)	499.0 (76 / 1068)	3.0 (2.5 / 3.6)	100.28	0.94	0.22 / 0.93
Kimi K2.6	Open	57.5 (60.2)	115	76	4	9	16.9 (13.4 / 22.2)	289.6 (136 / 528)	5.5 (4.4 / 7.3)	17.68	0.15	0.06 / 0.15
GLM 5.1	Open	57.0 (58.5)	114	81	8	5	15.5 (15.2 / 16.1)	189.1 (131 / 273)	7.0 (6.2 / 8.1)	18.24	0.16	0.07 / 0.13
Qwen3.6-Max-Preview	Open	54.0 (56.8)	108	82	3	10	12.5 (11.6 / 13.2)	172.2 (120 / 237)	6.9 (5.9 / 7.7)	21.44	0.20	0.09 / 0.14
DeepSeek-V4-Pro	Open	50.0 (52.1)	100	92	12	8	20.1 (19.7 / 20.8)	398.0 (321 / 493)	9.4 (8.6 / 10.4)	17.35	0.17	0.08 / 0.10
MiniMax M2.7	Open	49.0 (50.8)	98	95	13	7	27.5 (23.6 / 31.7)	683.6 (378 / 1006)	9.3 (7.9 / 10.9)	10.95	0.11	0.04 / 0.08

Model Configuration. All Terminus-2 trials share a fixed set of agent-level parameters, read from the `Terminus2.__init__` defaults in the Harbor adapter (`terminus_2.py`):

- ▷ **Sampling temperature:** $T = 0.7$ for all models.
- ▷ **Reasoning effort / thinking:** `reasoning_effort = None` and `max_thinking_tokens = None`, meaning Terminus-2 does not override provider defaults for extended thinking. Each model uses its native behavior.
- ▷ **Interleaved thinking:** `False`. Reasoning content is not retained in chat history and is not sent to the model in subsequent turns.
- ▷ **Context summarization:** `enable_summarize = True`, with proactive summarization triggered when free tokens drop below `proactive_summarization_threshold = 8000`. Summarization compresses prior conversation history to keep the agent within context limits.
- ▷ **Maximum turns:** unbounded (`max_turns = None`). The agent runs until it emits a stop signal, hits a timeout, or encounters an unrecoverable error.
- ▷ **LLM backend:** LiteLLM (`llm_backend = LLMBackend.LITELLM`), which routes all API calls through a unified interface and handles retry logic at the SDK level.
- ▷ **Output parser:** `parser_name = "json"`. The agent's LLM responses are parsed in JSON format to extract shell commands, file operations, and control signals.
- ▷ **Terminal multiplexer:** `tmux` with pane dimensions 160×40 characters.

Harness Configuration. All Terminus-2 experiments share the following Harbor settings unless otherwise noted:

- ▷ **Concurrency:** $n = 4$ for all models (to manage API rate limits and Docker resource contention on a single host).
- ▷ **Docker lifecycle:** `environment.force_build = false` (reuse cached images) and `environment.delete = true` (prune containers after each trial to prevent resource accumulation).
- ▷ **Network isolation:** Harbor automatically modifies each task's `docker-compose.yaml` to inject `network_mode: none` before container startup. This prevents agents from downloading solutions or consulting external services and ensures the task environment is fully self-contained. No Docker network pool exhaustion was observed across any experiment.

Error Classification. A small fraction of task attempts are classified as *errors*, in which the evaluation harness fails before the agent can attempt the task. Since all models share the same Terminus-2 scaffold, the error sources are identical across models and fall into two categories:

- ▷ **Tmux session initialization failures.** Terminus-2 relies on a `tmux` session to multiplex shell interaction. A race condition in its adapter occasionally sends keystrokes before the session is fully initialized, causing the attempt to abort. This is a non-deterministic harness bug independent of the task or model.

Table 5: **Detailed per-agent statistics for the agent benchmarking experiment.** Timeout (TO) is a subset of Fail; Err counts tasks where the harness failed before the agent could run. For Avg. Turns, Avg. Tokens, Avg. Time, and $/task, the overall average is shown first, with per-subset averages for passed / failed tasks in parentheses; averages are computed only over tasks where the agent ran. †Resolved rate = pass / (total − errors).

Agent	Model	Pass Rate (%)†	Outcomes				Avg. Turns (Pass / Fail)	Avg. Tokens (K) (Pass / Fail)	Avg. Time (min) (Pass / Fail)	Cost ($)	$/Pass	$/task (Pass / Fail)
			Pass	Fail	TO	Err						
Terminus-2	Claude Opus 4.7	62.5 (64.8)	125	68	2	7	16.7 (12.0 / 25.0)	261.5 (139 / 481)	3.6 (3.7 / 3.6)	63.47	0.51	0.23 / 0.39
Claude Code	Claude Opus 4.7	58.0 (60.7)	116	75	1	9	18.0 (18.1 / 17.8)	667.7 (572 / 816)	6.0 (5.5 / 7.0)	105.12	0.91	0.50 / 0.63
mini-SWE-agent	Claude Opus 4.7	52.0 (59.8)	104	70	1	26	17.1 (17.2 / 16.8)	206.4 (191 / 229)	3.9 (4.0 / 3.8)	56.94	0.55	0.28 / 0.29
OpenHands	Claude Opus 4.7	45.0 (57.3)	90	67	1	43	21.9 (22.7 / 20.7)	410.9 (417 / 403)	5.3 (6.1 / 4.6)	371.21	4.12	2.19 / 2.10
Terminus-2	Gemini 3.1 Pro	55.0 (57.0)	110	83	0	7	10.6 (9.1 / 12.5)	173.2 (84 / 289)	4.0 (3.6 / 4.4)	56.82	0.52	0.20 / 0.39
Gemini CLI	Gemini 3.1 Pro	56.0 (59.6)	112	76	6	12	41.5 (40.9 / 42.3)	694.7 (673 / 726)	6.3 (4.0 / 9.7)	85.90	0.77	0.45 / 0.47
Terminus-2	GPT-5.5	53.5 (56.6)	107	82	3	11	14.8 (7.9 / 24.1)	499.0 (76 / 1068)	3.0 (2.5 / 3.6)	100.28	0.94	0.22 / 0.93
Codex CLI	GPT-5.5	48.5 (56.1)	97	76	0	27	29.3 (32.4 / 26.6)	431.4 (464 / 416)	1.6 (1.6 / 1.3)	128.80	1.33	0.75 / 0.73

▷ **Container startup timeouts.** A small number of tasks use resource-intensive Docker images that exceed the harness startup timeout, preventing the agent from entering the environment.

These two sources account for all errors observed in this experiment, with error rates ranging from 2.5% to 5.5% across models. We report both the standard pass rate (pass / total) and the resolved rate (pass / (total − errors)) to isolate task-solving capability from harness artifacts. Agent timeouts are *not* classified as errors; an agent that runs until the time limit without producing a correct answer is counted as a failure.

C.2 Benchmarking Terminal Agents

Evaluation Harness. All agents are evaluated using the Harbor harness [Merrill et al., 2026], which provisions a fresh Docker container per task, injects the agent, and collects execution traces. Terminus-2, as Harbor's native agent, is tightly integrated with the harness and requires only a working `bash` shell inside the container. All other agents (*e.g.*, Claude Code, Codex CLI, Gemini CLI, OpenHands, Mini-SWE-Agent) are third-party and must be installed *inside* each task container before execution, which requires the harness to download and configure their respective toolchains (Node.js, npm, Python runtimes, etc.).

Harness Configuration (CLI Agent Experiments). The CLI agent experiments used Harbor with the following settings, shared with the Terminus-2 experiments where not specified:

▷ **Concurrency:** $n = 4$ for all CLI agents, to limit Docker resource pressure and API rate-limit risk on a single host.

▷ **Retry strategy:** previously errored tasks were re-run after fixing agent installation scripts. Results from the retry phase were merged with non-error results from the original run, keeping the most recent result per task.

▷ **Docker isolation:** identical `network_mode: none` injection, `force_build: false`, and `delete: true` as in the LLM benchmarking experiments (Appendix C.1).

Error Classification. We classify a task attempt as an *error* when the evaluation harness fails before the agent can attempt the task. Errors are distinct from task failures, in which the agent executes commands but does not satisfy the verifier. Across our evaluation, errors fall into three categories:

▷ **Tmux session initialization failures.** Terminus-2 relies on a `tmux` session to multiplex shell interaction. A race condition in its adapter (`tmux_session.py`) occasionally sends keystrokes before the session is fully initialized, causing the attempt to abort. This affects only Terminus-2 and is independent of the task or model.

▷ **Container startup timeouts.** A small number of tasks use resource-intensive Docker images that exceed the harness startup timeout, preventing any agent from entering the environment.

▷ **Third-party agent installation failures on incompatible base images.** Approximately 10% of TERMINALWORLD tasks use legacy base images (e.g., Ubuntu 16.04/18.04, CentOS 6/7, Debian 9)

whose package repositories are end-of-life. On these images, `apt-get update` or `yum install` fails, preventing the harness from installing the agent's toolchain inside the container. As Harbor's native agent, Terminus-2 does not require in-container installation and is therefore unaffected by this failure mode. This is the primary driver of the higher error rates observed for third-party agents (4.5–21.5%) compared to Terminus-2 (3.5–5.5%).

C.3 TERMINALWORLD vs. Terminal-Bench

Table 6 summarizes the scaffold and reasoning effort used for each model on Terminal-Bench and TERMINALWORLD. For Terminal-Bench, we use each model's official self-reported score, which may reflect different scaffolds and reasoning configurations. For TERMINALWORLD, all models are evaluated under a unified Terminus-2 scaffold with default reasoning effort. Since re-running all models under identical Terminal-Bench conditions is infeasible, we adopt the best available official score as each model's reference performance.

Table 6: Evaluation configurations for the Terminal-Bench and TERMINALWORLD experiments in Figure 4, including the scaffold and reasoning-effort setting used for each model.

Model	Type	Terminal-Bench			TERMINALWORLD		
		Scaffold	Reasoning	Pass (%)	Scaffold	Reasoning	Pass (%)
Claude Opus 4.7	Closed	Terminus-2	max	69.4	Terminus-2	high (default)	62.5
Gemini 3.1 Pro	Closed	Terminus-2	high	68.5	Terminus-2	high (default)	55.0
GPT-5.5	Closed	Codex CLI	xhigh	82.7	Terminus-2	medium (default)	53.5
Kimi K2.6	Open	Terminus-2	default	66.7	Terminus-2	default	57.5
GLM 5.1	Open	Terminus-2	default	63.5	Terminus-2	default	57.0
Qwen3.6-Max-Preview	Open	Terminus-2	default	65.4	Terminus-2	default	54.0
DeepSeek-V4-Pro	Open	Self-Developed Agent	max	67.9	Terminus-2	high (default)	50.0
MiniMax M2.7	Open	Claude Code	default	57.0	Terminus-2	default	49.0

C.4 Behavior Comparison between Agents and Humans

This section extends the agent-human comparison from Section 5. Beyond command-set overlap (21.4% median Jaccard similarity), we further examine how agent success rate varies with task complexity and how agent command counts relate to reference solution length.

Agent Success Degrades with Procedural Complexity. Figure 6 examines pass rate along two dimensions: human completion time and reference command count, where each cell reports the average pass rate across all eight evaluated models on tasks within that bin. Reference command count is the clearer predictor: tasks requiring 21+ commands remain consistently hard across all time bins (25.0%–41.2%), whereas tasks with 6–10 commands achieve up to 70.6%. Human completion time, by contrast, is a noisier signal because it conflates two distinct factors: a task with few commands but long runtime (*e.g.*, compilation or model training) appears in the same time bin as a procedurally complex task, yet is fundamentally easier for an agent that only needs to issue the right command and wait. This is reflected in the data: tasks with 1–5 commands still achieve 51.0% pass rate even when recordings exceed 180 seconds, while tasks with 21+ commands that humans complete in under 60 seconds drop to 25.0%. The key driver of agent difficulty is therefore the length of the sequential command plan the agent must execute correctly, not how long the execution takes.

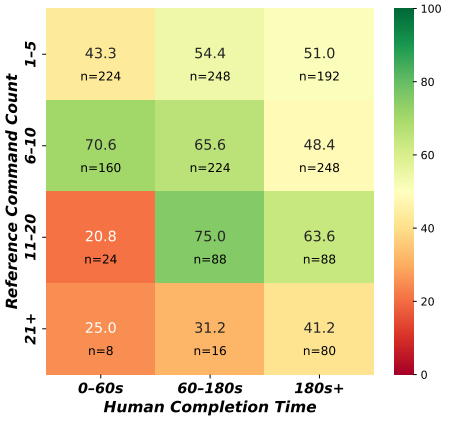


Figure 6: **Agents vs. Humans.** Tasks requiring many reference commands are consistently harder for agents, even when humans complete them quickly.

Agent Command Count Weakly Follows Reference Workflow Length. Figure 7 compares the agent

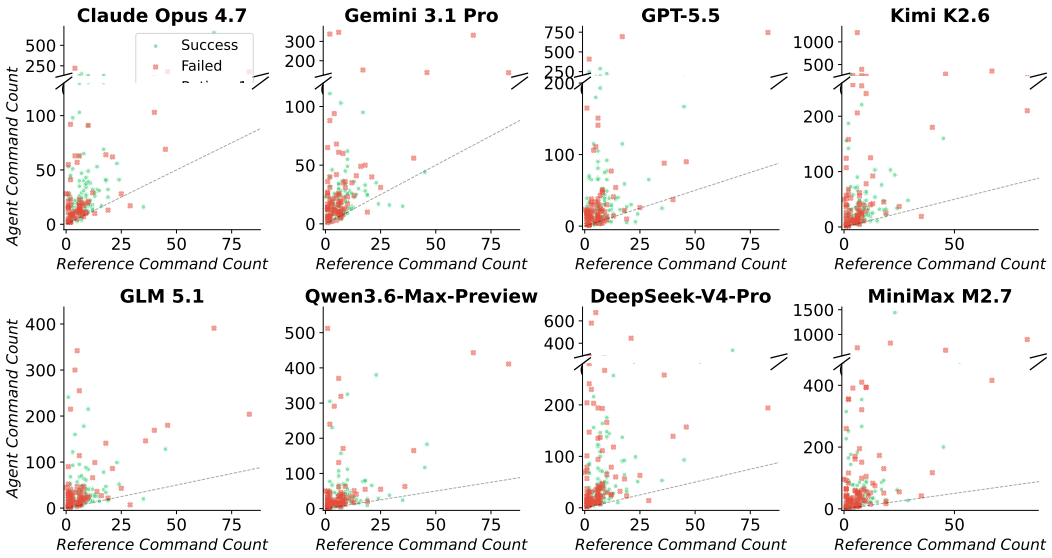


Figure 7: **Agent Command Count vs. Reference Command Count across Eight Models.** Reference command count does not tightly predict how many commands an agent will issue. Failed tasks consistently require far more commands than successful ones, reflecting unproductive exploration when the agent cannot identify the correct solution path.

command count with the reference solution command count across all eight evaluated models. Although the reference command count captures the length of the original human workflow, it does not tightly predict how many commands an agent will issue and execute. Most runs, both successful and failed, fall above the diagonal, meaning agents typically issue more commands than the reference solution. Failed runs cluster further above the diagonal than successful ones, reflecting unproductive exploration when the agent cannot find the correct solution path. Successful runs are closer to (though still generally above) the diagonal, indicating that agents can reach correct outcomes without needing to match reference solution efficiency exactly. Overall, agent command count shows little correlation with reference command count across all eight models. This gap may be explained by the nature of the recordings: source recordings are pre-planned demonstrations rather than exploratory sessions, so reference command counts are compact and clean; agents, by contrast, must explore, verify, and backtrack, accumulating far more commands before converging on a solution.