The Last Interface: Benchmarking Language Agents on Native Text-based User Interfaces

Anonymous authors
Paper under double-blind review

Abstract

Terminal-oriented agent evaluation has developed quickly, but it has converged on one interaction model: the agent issues shell commands and reads streamed output. Yet much of the software that runs on servers never offers even that prompt. Editors, file managers, process monitors, debuggers, and mail clients take over the terminal, repaint one screen of characters, and make that screen their entire interface: there is no prompt to return to, no element tree to read, and no API to call. This interface class, the native text-based user interface, has gone unmeasured. We present **TUI-Bench**, the first execution benchmark for agents that operate native TUIs. Its 492 tasks span 77 native Linux TUI tools in 19 application categories grouped into 6 supercategories; each task starts inside the running program, exposes a plain-text 32×96 character viewport with cursor coordinates, permits only keyboard actions, and is scored by deterministic verifiers that check application state, preserved data, and whether the program should still be running at the end. In a three-round evaluation of frontier models, the strongest models solve only a small fraction of scored episodes. The evaluation reveals a stable and interpretable failure pattern: agents succeed almost only where the interface has already degraded to a command line or a filter box, which suggests that they rely on pattern matching against commands rather than on continuous tracking of interface state. Whenever a task requires spanning multiple modes, holding focus, or protecting untouched state while reading, models lose control through premature exits, collateral state damage, or budget-exhausted repetition: the text on the screen is read correctly, but the interface itself is never modeled. TUI-Bench thus provides a controlled and reproducible measure of a capability gap that was previously invisible. Code and recorded trajectories are available at `https://anonymous.4open.science/r/TUI-Bench/`.

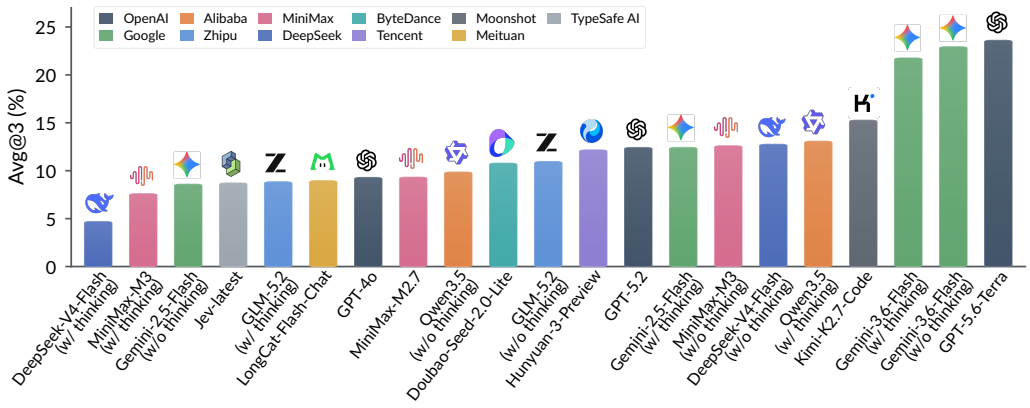


Figure 1: **TUI-Bench leaderboard.** Avg@3 of each completed configuration, ordered by mean success over the three recorded passes; colour encodes the model provider.

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

1 Introduction

Almost everyone who has administered a server has opened `vim`, edged the cursor around with scattered key presses, and then searched the screen for a way out. That experience is not a quirk of one editor. `vim`, `htop`, `tmux`, `Midnight Commander`, `Alpine`, and hundreds of other programs take over the terminal completely. They repaint a full screen of characters, treat each key press differently depending on hidden mode and focus, and never return to a prompt. Software of this kind is called a *text-based user interface* (TUI), and it remains one of the most common ways that real systems are operated (Figure 2).

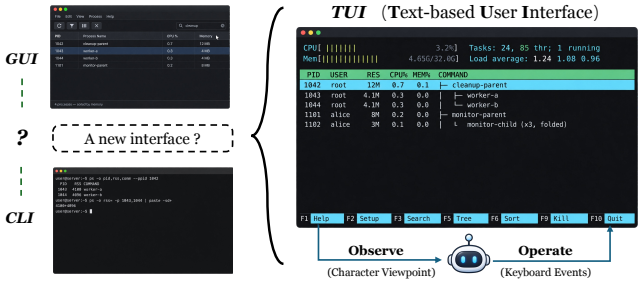


Figure 2: **A third interface paradigm.** The full-screen TUI, using `top` as the example, beside the graphical and command-line interfaces.

Software exposes itself to an agent through three interface paradigms, and benchmarks cover two of them. The first paradigm is the *graphical interface*: the windows, buttons, and menus of websites and desktop applications. It is thoroughly measured, from early controlled environments (Shi et al., 2017) and web-trajectory datasets (Deng et al., 2023) to executable arenas such as WebArena, VisualWebArena, OSWorld, and AndroidWorld (Zhou et al., 2024; Koh et al., 2024; Xie et al., 2024; Rawles et al., 2025), and that frontier has already moved on to long-horizon, cross-application desktop work (Yuan et al., 2026; Li et al., 2026a). There the agent observes rendered pixels or a serialized element tree and acts by clicking and typing. The second paradigm is the *command-line interface*, the question and answer of the shell. It is measured by Terminal-Bench and its successors (Merrill et al., 2026; Chen et al., 2026; Chu et al., 2026), in a literature now large enough to have a survey of its own (Bin et al., 2026). There the agent issues a command, reads a page of output, and the prompt returns, so every exchange is a fresh turn and a mistake costs one command. The third paradigm, the *text-based user interface*, has no benchmark at all. Inside a full-screen TUI the shell's rules do not apply: there is no prompt to return to and no corrective command to issue, the screen is a live control surface whose borders, status lines, and dialogs must be read as layout rather than as text, the same key inserts a character, selects a row, or closes a dialog depending on state the agent cannot see, and finishing the task often means leaving the program in a particular running state rather than exiting it. So an agent that is excellent at the shell has no reason to be excellent here, and nothing tells us whether it is.

We build **TUI-Bench** to measure exactly that. The benchmark contains 492 tasks on 77 native Linux TUI tools spanning 19 application categories grouped into 6 supercategories, from editors and file managers to debuggers, mail clients, packet analyzers, and kernel configuration. Each episode begins with the target application already running at an audited entry point, and the agent interacts with it through its only interface: the character grid and the keyboard, verified on state, preservation, and lifecycle at the end. We evaluate 21 model configurations, each on the full task set over three recorded passes under a common retry protocol. Twenty of them are generative configurations across 14 frontier models. The remaining one, Jev-latest, is a System One decision model, a non-generative model that returns calibrated distributions over caller-fixed answer spaces rather than text. The results expose a capability gap that is wide and structured. The best model solves only 22.9% of scored episodes, 71% of scored tasks were never solved by any model, and three of the 19 application categories have no success at all. Success concentrates where an interface offers a command line, search box, or filter list.

Our contributions are threefold:

- We identify native TUI operation as a distinct, previously unmeasured agent capability and formalize it as a partially observable decision problem (Åström, 1965; Kaelbling et al., 1998) in which interface state is hidden behind a character grid, together with a verification framework

(application state, preservation, and lifecycle) that captures failure modes invisible to file diffs and exit codes.

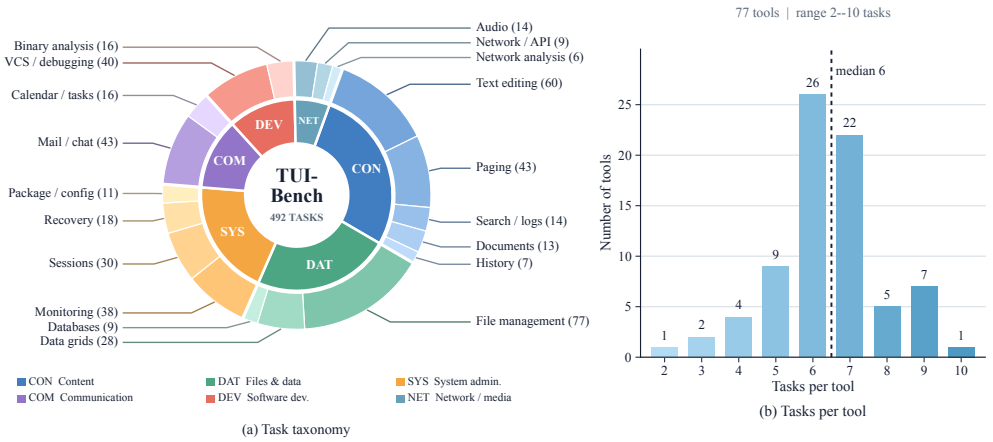
- We introduce TUI-Bench, the first execution benchmark for TUI operation: 492 tasks over 77 native Linux TUI tools in 19 application categories grouped into 6 supercategories, a character-grid observation with keyboard-only actions, and deterministic lifecycle-aware verifiers with executable release gates.
- We evaluate 21 model configurations, each on the full task set over three recorded passes, and read the trajectories. Every configuration fails most of the benchmark through the same few mechanisms, and all of them trace to one absence: models read the screen well but never build a model of the interface behind it. Success concentrates where an interface offers a place to type; premature exits, collateral state damage, and budget-exhausted repetition take the rest. TUI-Bench thereby provides a controlled and reproducible measurement of a capability gap that was previously invisible to benchmarks.

2 Related work

Agents on graphical interfaces. Visual agents progressed from visual web tasks in VisualWebArena (Koh et al., 2024) to complete operating systems in OSWorld and AndroidWorld (Xie et al., 2024; Rawles et al., 2025), where the agent observes screenshots and acts by clicking and typing. These lines establish the value of real software and end-state verification, and the frontier keeps extending, from human baselines that price agent effort against a person's (Abhyankar et al., 2026) to per-platform rebuilds such as macOS-World (Yang et al., 2025). Their central difficulty is grounding: mapping pixels to actions. A TUI removes that particular difficulty, because its surface is already text, and replaces it with a different one. The text is a transient control surface whose meaning depends on application state, and no widget hierarchy describes the available actions. That hierarchy is load-bearing in the graphical setting: where a serialized structure is available, agents lean on it heavily, and their state beliefs follow it even when the pixels they can read perfectly well disagree (Zhang et al., 2026); even with both channels in hand, reconstructing the transition a single action produced remains unreliable (Pillai et al., 2026). A native TUI offers no such channel to lean on.

Agents on the command line. Command-line settings made the agent–computer interface itself an object of study. SWE-bench and SWE-agent (Jimenez et al., 2024; Yang et al., 2024) evaluate repository repair through a shell; Terminal-Bench 2.0 (Merrill et al., 2026) grades hard command-line work in isolated containers; Long-Horizon-Terminal-Bench adds dense subtask grading (Li et al., 2026c); TerminalWorld reverse-engineers CLI tasks from tens of thousands of in-the-wild terminal recordings (Chu et al., 2026); and TUA-Bench extends command-line evaluation beyond programming (Chen et al., 2026). A parallel line synthesizes terminal environments at scale for *training* rather than evaluation (Cheng et al., 2026), and a recent survey organizes the area around the command-execution loop itself, observing that prevailing evaluations report final outcomes while exposing process quality and recovery unevenly (Bin et al., 2026). In all of these, the terminal is a place where an agent issues commands and reads output: control returns to the prompt after every exchange.

Agents in the terminal, beyond the prompt. Our object of study begins where the command line stops: the application is already open, there is no prompt to return to, and no command, script, or direct file operation can substitute for using the program. Closest to us is TUI testing (Peng et al., 2026), which exposes native TUIs to models but asks them to explore for coverage and crashes; TUI-Bench instead asks the operator question, completing a user request by reading a plain-text screen and pressing keys to a correct terminal state, which changes both task construction and grading. Character grids also appear in game environments such as NetHack (Küttler et al., 2020; Paglieri et al., 2025), but a game engine has one authored rule set, uniform across episodes, with a score defined inside the game, whereas a native TUI sits on a real filesystem, real processes, and real state that the task binds. Finally, two studies make the interface itself the independent variable: screen-only graphical agents versus skill-mediated command-line agents on matched goals and verifiers (Zhou et al., 2026), and a hybrid-interface benchmark where no single channel suffices (Li et al., 2026b); both keep the command channel available in at least one arm. TUI-Bench is the case in which it is gone, and the only thing the agent can do to the world is press a key.



(a) Task taxonomy (b) Tasks per tool

Figure 3: **TUI-Bench dataset overview.** Inner ring: 6 supercategories. Outer ring: 19 application categories. Right panel: tasks per tool.

3 TUI-Bench

TUI-Bench asks one question: when an entire interface is a grid of characters, can an agent operate the program behind it? The screen is plain text, the same alphabet the model reads fluently, yet it is also a live control surface: every keystroke lands in a hidden mode, a pane, a dialog, and the task is complete only when the machine behind the characters is left in the requested state. This section formalizes the interaction (§3.1), describes the task set and its taxonomy (§3.2), and explains the environments and verification (§3.3). Figure 3 summarizes dataset composition: 492 tasks organized into 6 supercategories over 19 application categories, distributed across 77 tools.

3.1 Formulation: interaction through a character grid

We model each task as a partially observable Markov decision process (POMDP) (Åström, 1965; Kaelbling et al., 1998) $\langle \mathcal{Q}, \mathcal{S}, \mathcal{A}, \mathcal{O}, T, r \rangle$ in the form standard for LLM agents acting under environment feedback (Song et al., 2024): instruction space $\mathcal{Q}$, hidden state space $\mathcal{S}$, action space $\mathcal{A}$, observation space $\mathcal{O}$, deterministic transition T, and terminal reward $r \in \{0, 1\}$. The hidden state s_t is everything the running application holds; the agent sees only the observation

$$o_t \;=\; \big(q,\; b_t,\; h_t,\; G_t,\; c_t\big), \qquad G_t \in \Sigma^{32\times 96}, \tag{1}$$

the instruction, remaining budgets, recent interaction records, the 32×96 character grid G_t, and the cursor position, and replies with one action

$$a_t \in \big\{\texttt{press}(k_1, \ldots, k_m),\; \texttt{type}(x),\; \texttt{wait_idle},\; \texttt{finish}\big\}, \tag{2}$$

selected by $a_t \sim \pi_\theta(\cdot \mid o_0, a_1, o_1, \ldots, a_{t-1}, o_t)$; the first three tools touch the world only through the keyboard. The episode ends at `finish`, at application exit, or when a budget is exhausted, and the reward is computed once, after the episode, from the final state s_T and the persistent artifacts D_0, D_T: a task passes only when

$$\text{state}(s_T) \;\wedge\; \text{preserve}(D_0, D_T) \;\wedge\; \text{lifecycle}(s_T, \ell) \tag{3}$$

all hold, with ℓ the declared lifecycle. The observation contents, the four-tool contract, the shared budgets, and the scoring conjuncts are specified in Appendix C.

3.2 Tasks and applications

6 supercategories, 19 application categories. The inner ring of Figure 3 organizes the 492 tasks into 6 supercategories that an operator would recognize at a glance. Content is the largest with 137 tasks, followed by Files and data with 114, System administration with 97, Communication with 59,

Software development with 56, and Network and media with 29. Each supercategory decomposes into the 19 application categories of the outer ring, led by File management with 77 tasks and Text editing with 60 and narrowing to Database TUIs with 9 and Network analysis with 6, so the two rings read as one hierarchy: a supercategory names the kind of work, and its application categories name the programs that do it. All category-level results in this paper are reported on the 19 application categories; the full distribution and the supercategory mapping are in Appendices A.2 and A.3.

Tasks per tool. The right panel of Figure 3 shows how the 492 tasks fall on the 77 tools: every tool contributes between two and ten tasks, with a median of six. The cap is deliberate. With at most ten tasks per program, no single familiar editor or file manager can dominate the score, and an agent cannot pass by excelling at one interface family alone. The complete tool-by-tool distribution is in Appendix A.4.

The 77 tools. Admission is a property of the program, not of the task: a tool enters the benchmark only if it is terminal-resident and character-rendered, keeps persistent interactive state, and is navigable by keyboard; line-oriented shells, read–eval–print loops, and one-shot command output do not qualify. The catalog that results spans modal and modeless editors, dual-pane and tree-based file managers, pagers and text browsers, Git front-ends and debuggers, process monitors, multiplexers, mail and chat clients, data grids, music players, disk utilities, and specialized interfaces such as `menuconfig` and `radare2` visual mode. Table 4 in Appendix A.4 lists all 77 tools by category.

3.3 Environments and verification

Every task runs in a fresh Docker environment with two CPUs, 4 GiB of memory, no network, and a deterministic fixture that fixes the files, processes, and configuration the task needs. A task contract binds the application command, entry point, workspace, lifecycle, and verifier to the task identifier by digest. Harbor (Harbor Framework Team, 2026) constructs and grades the environment, and the `tui-test` runtime owns the pseudo-terminal, decodes the stream into character frames, and applies keyboard events. The same runtime records every model call, action, screen, and verdict, so any episode can be audited without giving the model an extra observation channel.

Verification is deterministic and task-specific. The 26 verifier modes form seven families: structured terminal-surface checks, file and tree transforms, application state, tool-native state, session state, Git state, and structured exit or recovery output. Appendix A.6 gives the complete inventory in Table 6. Each task also carries executable release gates: the intended positive fixture passes twice, and the untouched initial state, a wrong target, and the opposite lifecycle are each rejected, for all 492 tasks. Every task was additionally audited by human experts, whose ratings reached substantial agreement (Cohen's $\kappa = 0.81$; Cohen, 1960; Landis & Koch, 1977) before the freeze. The construction, admission gates, and human audit are documented in Appendix A.6, and the complete execution contract shared by every task in Appendix A.5.

4 Experiments

4.1 Metrics and setup

We report three success numbers and five cost numbers per model, with precise definitions in Table 7 of Appendix B and the outcome accounting behind the scored denominators in Appendix B. Table rates throughout are percents of scored episodes with one decimal. Avg@3 is the mean success rate over scored episodes across the three recorded passes, where a scored episode is one whose runtime was valid. Pass@3 is the share of scored tasks solved at least once, which measures reach, and Pass^3 is the share solved in every scored round, which measures reliability; it instantiates at $k = 3$ the pass^k family introduced by Yao et al. (2025) for tool-agent settings. The cost columns price the three budgets the protocol grants every episode: *Total*, *Read*, and *Write* are mean tokens per scored episode, *Time* is mean wall-clock minutes per scored episode, and *Keys* is mean keyboard input units per scored episode, the input bill of an interface that offers nothing but keys.

A model configuration is a named model revision with its provider endpoint and reasoning setting, run through the common TUI-Bench controller. We evaluate 21 configurations: 20 generative ones covering 14 frontier models, plus Jev-latest, a System One decision model. Unlike every other row it does not generate text at all. The caller fixes an answer space, such as a choice among options, a

Table 1: Main results on TUI-Bench over three recorded passes per configuration. †Jev-latest is a System One decision model, reported as its own class and excluded from the Field row. The best value in each column is **bold** and the worst is underlined; metric definitions are in Appendix B.

	Task success (%)			*Tokens*			*Cost*	
Model	Avg@3	Pass@3	Pass^3	Total	Read	Write	Time	Keys
Jev-latest†	8.4	8.7	7.6	1.5M	1.3M	282k	6.9	273
Non-thinking models								
Gemini-2.5-Flash (w/o thinking)	8.5	8.6	8.6	670k	670k	407	9.3	267
LongCat-Flash-Chat	9.1	9.0	9.0	118k	116k	2k	**3.0**	48
GPT-4o	9.2	9.3	9.0	457k	456k	1k	8.7	241
MiniMax-M2.7	9.3	9.3	9.1	212k	201k	11k	4.5	65
Qwen3.5 (w/o thinking)	9.7	9.9	9.9	431k	430k	1k	4.6	103
Doubao-Seed-2.0-Lite	10.3	10.8	10.8	803k	800k	4k	20.8	289
Hunyuan-3-Preview	10.5	12.2	10.8	221k	218k	3k	48.5	130
GLM-5.2 (w/o thinking)	11.0	11.0	11.0	96k	95k	1k	5.5	72
MiniMax-M3 (w/o thinking)	12.4	12.6	12.6	314k	307k	7k	16.6	127
DeepSeek-V4-Flash (w/o thinking)	12.5	12.8	12.8	688k	684k	4k	15.3	357
Gemini-3.6-Flash (w/o thinking)	22.9	22.9	21.9	241k	240k	735	14.0	229
Thinking models								
DeepSeek-V4-Flash (w/ thinking)	4.8	4.7	4.7	192k	181k	11k	4.7	98
MiniMax-M3 (w/ thinking)	7.7	7.6	7.4	64k	52k	12k	3.3	38
GLM-5.2 (w/ thinking)	9.0	8.8	8.2	**37k**	**29k**	8k	3.8	**29**
GPT-5.2	12.1	12.4	12.4	490k	489k	2k	10.1	309
Gemini-2.5-Flash (w/ thinking)	12.4	12.4	12.4	529k	529k	**360**	9.0	310
Qwen3.5 (w/ thinking)	12.9	13.1	12.6	741k	705k	37k	10.7	213
Kimi-K2.7-Code	15.2	15.3	14.0	222k	200k	21k	13.6	192
Gemini-3.6-Flash (w/ thinking)	21.8	21.8	21.8	304k	303k	859	16.4	233
GPT-5.6-Terra	**22.9**	**23.6**	**23.6**	488k	471k	17k	24.8	332
Field	12.2	12.4	12.1	366k	359k	7k	12.4	184

score, or a judgment, and the model returns a calibrated distribution over that space. This contract is built for the high-frequency, low-entropy decisions inside an agent loop rather than for open-ended generation. The benchmark's four-tool action space is translated into that interface, so the comparison is kept as fair as the architecture allows; it is, however, controlled in configuration rather than in mechanism, because Jev-latest neither reads the screen as text nor emits keystrokes as tokens. Its row is therefore an architecture contrast reported as its own class, not a strictly controlled comparison against the generative field (the translation layer and the prompts it receives are specified in Appendix D.9). Every configuration was recorded over three passes of the full 492-task set; the *Field* row averages the 20 generative configurations, and Jev-latest is reported as its own class rather than averaged into the generative field.

4.2 Overall results

Table 1 is the headline, and the reading is blunt. The best model, GPT-5.6-Terra, solves 22.9% of scored episodes and only 23.6% of its scored tasks every time, while the weakest, DeepSeek-V4-Flash (w/ thinking), sits at 4.8%, so the entire evaluated field fits within a 18.2-point band. Pooling all passes of all models, 135 of 467 scored tasks were ever solved, which means 71% were never solved in any of the attempts, and only 6 tasks were solved by every model.

Reach and reliability stay close together under this protocol's retry policy (§5.4): GPT-5.6-Terra converts its full 23.6% reach into every-pass reliability, and no configuration loses more than a fifth of its solved tasks between the two. Cost separates the field far more than accuracy does, and all three budgets tell the same story at different prices. Reading dominates the token bill: the field spends 98% of its tokens re-entering the screen into the context, and Gemini-3.6-Flash reads 240k tokens per episode to write 735. Time and keys track competence from both sides, from 25m and 332 for the best model to 7m for the System One decision model, which spends the most without ever learning when to stop. The cheapest episodes belong to the models that end their own tasks, and the

analysis of Section 5 reads these bills against the failure modes behind them; the full per-dimension breakdowns are in Appendix D.

5 Analysis

Section 4.2 established that every completed configuration fails most of TUI-Bench. This section reads the trajectories to find out why, and one absence turns out to sit behind every finding: models read the screen well but never build a model of the interface behind it. We first locate where success survives, then decompose the failures into four recurring patterns, trace all four to that shared root, and close with what the survivors look like.

5.1 Where success survives

Success on a native TUI survives only where the interface still offers a place to type a command. We test this directly instead of inferring it from category rankings: each of the 467 scored tasks was labelled, before any outcome was examined, with six binary interface requirements drawn from its task specification and tool identity. The affordance effect is the largest in the benchmark: tasks whose tool exposes a command line, search box, or query editor pass at 16.95%, against 6.78% for tasks without one, a 10.17-point difference with non-overlapping tool-cluster bootstrap intervals. The burden conditions cut the other way: tasks that require a modal transition pass at 8.8% against 14.4%, and the lifecycle contract prices the field hardest of all, with episodes that must end with a live program passing at 10.4% while clean exits pass at 30.3%. Cross-view memory and explicit preservation constraints barely move success on their own, so it is the state-tracking demands rather than information volume that separate the survivors from the rest. Figure 4 Figure 4 retains selected nonzero categories, including network analysis at 0.7%, and all four zero-success categories. The complete 19-application-category ranking with the 31 of 73 judgeable tools that never pass is in Appendix D.1 and Appendix D.2.



Figure 4: **Category success.** Selected nonzero categories, including the lowest nonzero category, and all zero-success categories.

The surviving tools share one interface property, a command line or search box that accepts input directly, and that property also makes success fast: the mean pass finishes in 34 steps, with 20% inside five. The best case is extreme, Kimi-K2.7 formatting a SQL query in Harlequin with two keystrokes, a colon and the F4 key. The worst case is its mirror: asked to fold one row in htop's tree view, Gemini-2.5-Flash presses the up arrow 132 times in three uninterrupted bursts and ends at the step cap with nothing folded. Success is a command and failure is navigation: agents know how to type at a program, not yet how to hold an interface in mind. Both trajectories are shown in full in Appendix F.

5.2 Four ways to lose an interface

Figure 5 shows one representative configuration per recurring failure profile, drawn from the models the paragraphs below name; the full decomposition of the 20 generative configurations and the separate analysis of the System One decision model are in Appendix D.4 and Appendix D.9.

Protocol collapse. Some configurations never reach the interface at all: between 63% and 84% of the failures in this group are calls that violate the four-tool schema even after two repair prompts, so the episode ends within a handful of steps with the task never attempted. These are the cheapest rows in Table 1 because there is no episode to run, and the degradation is visible before the verdict: LongCat's outputs decay into lone stray characters as its action channel collapses (Case 5).

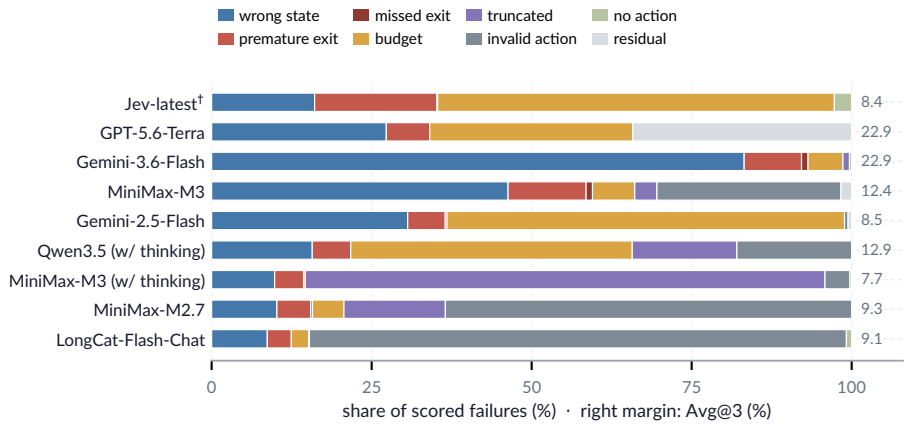


Figure 5: **Failure fingerprints.** Share of each selected configuration's scored failures by failure mode; the right margin gives Avg@3. The full decomposition is in Appendix D.4.

Reasoning overflow. The cheapest way to lose an interface is to lose the response that would have operated it: a model that reasons at length exhausts the per-response output cap before an action is emitted, and the episode dies mid-thought. The violet band dominates the MiniMax-M3 thinking row of Figure 5, which loses 81% of its failures this way (Case 4) while its non-thinking mode truncates almost never; across the six twin families, enabling thinking raises the truncation share for five and lowers Avg@3 for four, with the paired values in Appendix D.6. The two success exceptions show the boundary: Gemini-2.5 does not truncate in either mode, and Qwen3.5 improves despite a higher truncation share, so truncation is a concrete failure mode for several thinking configurations rather than a universal effect of reasoning.

Thrashing. The opposite failure is to run forever without progress. For the budget-bound configurations, well over half of all failures end only when the step cap cuts the episode off, and the archetype is field-shaped rather than model-shaped: 24% of every failed episode in the campaign hits the step cap, and among failures with at least ten actions, 34% spend a majority of their actions repeating an (action, screen) pair the agent had already seen. The archetypal member is Gemini-2.5-Flash, which loses 62% of its failures to the cap and pays 9m of wall-clock per episode, because a thrashing episode runs until something external stops it. The up-arrow storm of §5.1 is this archetype in pure form, and undo storms are its editor dialect: the literal `u` key is up to 8.1% of everything MiniMax-M3 (w/o thinking) presses on editor tasks, and many of these presses arrive while the editor is in insert mode, where they type `u` into the file instead of undoing (Case 2).

Wrong terminal state. The strongest configurations are the ones that fail at the finish line: for the best-behaved group the dominant ending is a finished episode with a wrong world, in two signatures that file diffs and exit codes cannot see. In the first, agents kill what they were asked to keep alive. 1822 failures, 22% of verifier-graded failures, end with the program closed when it had to stay running, and the lifecycle split prices the habit field-wide: episodes that must end with a live program pass at 10.4% while clean exits pass at 30.3% (the per-configuration split is in Appendix D.3). In the second, agents reach the requested view and still fail the world: among failed episodes whose verifier declares expected terminal tokens, 30% show every expected token on the final screen. The two case-study emblems are exact mirrors: the Alpine episode of Case 6 ends with the requested Archive index on screen and a dirtied world behind it, while the Byobu episode of Case 7 calls finish before the requested entry was ever scrolled into view. These process diagnostics use different denominators and are reported separately in Appendix D.7 and Appendix D.3.

5.3 The shared root: reading is not operating

All four patterns trace back to a single missing capacity: the agent never reconstructs the interface state behind the screen. Perception itself is not what breaks: models quote the screen's contents, parse the task correctly, and act decisively wherever the interface reduces to a query box.

What fails is everything the observation does not contain. In the terms of §3.1, the agent receives o_t and reads it well but never reconstructs the hidden state s_t behind it, which mode the program is in, which pane holds the focus, what the last key actually changed, and what the world must look like when the episode ends.

Each failure pattern is that one absence seen from a different angle. Thrashing is search without prediction. Undo storms are restoration without a record. Premature exits and collateral damage are planning without a terminal-state contract. Even reasoning overflow is the same absence at the token level: the thinking variants re-derive interface state from the viewport every turn because no cheaper representation exists in the model. The bills of Table 1 price the absence from both sides, with 98% of the field's tokens spent re-entering the screen and 184 keyboard units per episode where a human operator uses a handful, because a model that cannot predict what the next key will do can only spend keys to find out. Reading is fluent; operating is what never happens.

This is also why structured-surface benchmarks never exposed the gap: the DOM, the accessibility tree, or the shell prompt does the bookkeeping for the agent, so the missing capacity is never load-bearing there. A native TUI does not do the bookkeeping, and that is the capability TUI-Bench isolates. The System One decision model Jev-latest makes the same point from the opposite side: with no reasoning to overflow and no protocol to collapse, its failures pile up in thrashing and it lands at 8.4% of scored episodes, mid-field, its rare passes decomposing into one-screen decisions (Case 8).

5.4 Narrow and overlapping task reach

What the field solves is a narrow strip of easy, command-shaped tasks rather than complementary strengths, and the strip is thin. Across all 492 tasks, 25 have no scored episode and 332 of the remaining 467 are never solved; only 135 are solved by any configuration and just 6 by every one. Even under maximum-marginal ordering, cumulative coverage reaches all 135 tasks after eleven of the twenty-one configurations, and the remaining ten add nothing: different models are not solving different benchmarks, they are taking turns on the same small strip.

The measurement choices this motivates are concrete. Report coverage and strict success alongside scored-only success, verify what the screen cannot show, since premature exits and collateral changes are invisible to file diffs and exit codes, and train for the actual gap, because command-line competence did not transfer to modal keyboard control and longer reasoning collided with the output cap. These failures are concrete, repeated, and measured, and they are the gap TUI-Bench exists to close.

6 Conclusion

TUI-Bench makes the full-screen program an explicit evaluation target, and the frontier field solves at most 22.9% of its scored episodes with entire categories unsolved and every headline failure traced to a measured mechanism. The benchmark turns these failure modes into a progress signal for the community.

Limitations. The observation omits color and styling, episodes start with the application already open. The benchmark covers Linux only, and the deterministic verifiers encode a finite specification bounded by executable admission gates and a human audit.

Reproducibility statement

TUI-Bench is designed to support reproducibility at both the benchmark and analysis levels. The task formulation, environments, and verification protocol are described in Section 3, with the complete task schema, execution contract, construction pipeline, and validation procedure documented in Appendix A. Metric definitions and outcome accounting are provided in Appendix B; detailed experimental results and reliability analyses are reported in Appendix D; and the frozen evaluation prompt and interaction protocol are given in Appendix E. We release all 492 task packages, their pinned execution environments and verifier scripts, the benchmark controller and trial recorder, the analysis code, and the recorded trajectories for the evaluated configurations, enabling the reported tables and

figures to be regenerated from the released artifacts. Task fixtures and verifiers are deterministic and version-bound, so a given recorded terminal state and associated artifacts receive a deterministic verdict. Exact model trajectories may nevertheless vary across reruns because of provider-side sampling and operational nondeterminism; we therefore evaluate each configuration over three recorded passes and explicitly report outcome accounting and pass-to-pass reliability rather than assuming deterministic model behavior.

Ethics Statement

TUI-Bench evaluates models inside licensed, unmodified, off-the-shelf Linux terminal applications; it collects no personal data and runs against fixed local fixtures, so no human subjects or user content are involved. The recorded trajectories contain only screen content produced by the benchmark's own seeded environments, and the deployment is network-isolated, so no task can reach external systems. We report model identities and per-provider costs openly; all evaluated models are used within their published terms of service. We do not foresee a direct dual-use risk beyond that already inherent to general-purpose agent evaluation: the benchmark measures the operation of standard software through its public keyboard interface and does not reward evasion, intrusion, or anything specific to security-sensitive targets. By exposing a large, measured capability gap, we hope the benchmark steers effort toward making agents more transparent and correct when they operate real systems on a user's behalf.

AI use statement

We used large language models as research productivity tools during the preparation of this work. Their use included drafting and editing prose, assisting with the creation and editing of analysis and visualization code, and serving as a reviewer-like aid for checking the consistency of reported numbers against recorded experimental artifacts. The benchmark design and task construction criteria, failure taxonomy and labels, statistical analyses, interpretation of the experimental results, and scientific claims were determined by the authors rather than delegated to AI tools. All AI-assisted code and written material were reviewed by the authors. In particular, all reported quantitative results and figures were regenerated from the recorded trajectories using the analysis pipeline and were independently checked against the underlying artifacts. The models evaluated in TUI-Bench are experimental subjects of the benchmark; their evaluation runs are separate from the productivity-tool uses disclosed above. The authors take full responsibility for the final content of the paper and associated artifacts.

References

Reyna Abhyankar, Qi Qi, and Yiying Zhang. OSWorld-Human: Benchmarking the efficiency of computer-use agents. In A. Chowdhery and Z. Jia (eds.), *Proceedings of Machine Learning and Systems*, volume 8, pp. 482–494. MLSys, 2026. URL https://proceedings.mlsys.org/paper_files/paper/2026/file/5edb57c05c81d04beb716ef1d542fe9e-Paper-Conference.pdf.

Karl Johan Åström. Optimal control of markov processes with incomplete state information. *Journal of Mathematical Analysis and Applications*, 10(1):174–205, 1965. ISSN 0022-247X. doi: 10.1016/0022-247X(65)90154-X. URL https://www.sciencedirect.com/science/article/pii/0022247X6590154X.

Yi Bin, Xiaoyang Yuan, Haoxi Zeng, Wencheng Ye, Wenqi Shao, Chen Qian, Wei Ye, Yujuan Ding, Zheng Wang, Pengpeng Zeng, Jingkuan Song, and Heng Tao Shen. Terminal agents: A survey of AI agents in command-line environments. *arXiv preprint arXiv:2608.20485*, 2026. URL https://arxiv.org/abs/2608.20485.

Shoufa Chen, Luyuan Wang, Xuan Yang, Zhiheng Liu, Yuren Cong, Yuanfeng Ji, Feiyan Zhou, Xiaohui Zhang, Fanny Yang, and Belinda Zeng. TUA-Bench: A benchmark for general-purpose terminal-use agents. *arXiv preprint arXiv:2606.28480*, 2026. URL https://arxiv.org/abs/2606.28480.

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

Zhaoyang Chu, Jiarui Hu, Xingyu Jiang, Pengyu Zou, Han Li, Chao Peng, Peter O'Hearn, Earl T. Barr, Mark Harman, Federica Sarro, and He Ye. TerminalWorld: Benchmarking agents on real-world terminal tasks. *arXiv preprint arXiv:2605.22535*, 2026. URL https://arxiv.org/abs/2605.22535.

Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960. doi: 10.1177/001316446002000104. URL https://doi.org/10.1177/001316446002000104.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2Web: Towards a generalist agent for the web. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 28091–28114. Curran Associates, Inc., 2023. doi: 10.52202/075280-1220. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/5950bf290a1570ea401bf98882128160-Paper-Datasets_and_Benchmarks.pdf.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments. Software, 2026. URL https://doi.org/10.5281/zenodo.21878893. Version 0.21.0.

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 B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/edac78c3e300629acfe6cbe9ca88fb84-Paper-Conference.pdf.

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. ISSN 0004-3702. doi: 10.1016/S0004-3702(98)00023-X. URL https://www.sciencedirect.com/science/article/pii/S000437029800023X.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. VisualWebArena: Evaluating multimodal agents on realistic visual web tasks. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 881–905, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.50. URL https://aclanthology.org/2024.acl-long.50/.

Heinrich Küttler, Nantas Nardelli, Alexander Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktäschel. The NetHack learning environment. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 7671–7684. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/569ff987c643b4bedf504efda8f786c2-Paper.pdf.

J. Richard Landis and Gary G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977. doi: 10.2307/2529310. URL https://doi.org/10.2307/2529310.

Jinchao Li, Yunxin Li, Chenrui Zhao, Zhenran Xu, Baotian Hu, and Min Zhang. WindowsWorld: A process-centric benchmark of autonomous GUI agents in professional cross-application environments. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 15262–15280, San Diego, California, United States, July 2026a. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.750. URL https://aclanthology.org/2026.findings-acl.750/.

Wanli Li, Bowen Zhou, Yunyao Yu, Zhou Xu, Yifan Yang, Dongsheng Li, and Caihua Shan. WeaveBench: A long-horizon, real-world benchmark for computer-use agents with hybrid interfaces. *arXiv preprint arXiv:2606.09426*, 2026b. URL https://arxiv.org/abs/2606.09426.

Zongxia Li, Zhongzhi Li, Yucheng Shi, Ruhan Wang, Junyao Yang, Zhichao Liu, Xiyang Wu, Anhao Li, Yue Yu, Ninghao Liu, Lichao Sun, Haotao Mi, and Leowei Liang. Long-Horizon-Terminal-

Bench: Testing the limits of agents on long-horizon terminal tasks with dense reward-based grading. *arXiv preprint arXiv:2607.08964*, 2026c. URL https://arxiv.org/abs/2607.08964.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong 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 Guha, Gabriel Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong (Ryan) 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 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 C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 40903–40986, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/444a3737adaee10d86ad2ef5f74468e6-Paper-Conference.pdf.

Davide Paglieri, Bartłomiej Cupiał, Samuel Coward, Ulyana Piterbarg, Maciej Wołczyk, Akbir Khan, Eduardo Pignatelli, Łukasz Kuciński, Lerrel Pinto, Rob Fergus, Jakob Foerster, Jack Parker-Holder, and Tim Rocktaeschel. BALROG: Benchmarking agentic LLM and VLM reasoning on games. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 96666–96702, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/f0b1515be276f6ba82b4f2b25e50bef0-Paper-Conference.pdf.

Chao Peng, Ruida Hu, Ajitha Rajan, Tegawendé F Bissyandé, Jacques Klein, and Cuiyun Gao. Can LLMs test terminal user interfaces? *arXiv preprint arXiv:2608.03743*, 2026. URL https://arxiv.org/abs/2608.03743.

Abhishek Pillai, Samir Kumar Nayak, and Yuan Chen. Desktop-Delta Bench: Do computer-use models understand desktop GUI transitions? *arXiv preprint arXiv:2607.26041*, 2026. URL https://arxiv.org/abs/2607.26041.

Chris Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy Lillicrap, and Oriana Riva. AndroidWorld: A dynamic benchmarking environment for autonomous agents. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 406–441, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/01a83bc2f2732a58e6aa731e659e7101-Paper-Conference.pdf.

Tianlin Shi, Andrej Karpathy, Linxi Fan, Jonathan Hernandez, and Percy Liang. World of Bits: An open-domain platform for web-based agents. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 3135–3144. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/shi17a.html.

Yifan Song, Da Yin, Xiang Yue, Jie Huang, Sujian Li, and Bill Yuchen Lin. Trial and error: Exploration-based trajectory optimization of LLM agents. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7584–7600, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.409. URL https://aclanthology.org/2024.acl-long.409/.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 52040–52094. Curran Associates, Inc., 2024. doi: 10.52202/079017-1650. URL https://proceedings.neurips.cc/paper_files/

paper/2024/file/5d413e48f84dc61244b6be550f1cd8f5-Paper-Datasets_and_Benchmarks_Track.pdf.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.

Pei Yang, Hai Ci, and Mike Zheng Shou. macOSWorld: A multilingual interactive benchmark for GUI agents. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 134014–134056. Curran Associates, Inc., 2025. doi: 10.52202/085713-4468. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/c276c3303c0723c83a43b95a44a1fcbf-Paper-Conference.pdf.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for Tool-Agent-User interaction in real-world domains. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 9965–10017, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/1b126cc38b8638e07bef37e7b2bb72bf-Paper-Conference.pdf.

Mengqi Yuan, Zilong Zhou, Xinzhuang Xiong, Weiming Wu, Jiayang Sun, Jiamin Song, Kaiqian Cui, Bowen Wang, Haoyuan Wu, Yitong Li, Dunjie Lu, Haikong Lu, Qi Zhen, Xinyuan Wang, Jiaqi Deng, Yuhao Yang, Cheng Chen, Boyuan Zheng, Alex Su, Xiao Yu, Hao Zou, Saaket Agashe, Xing Han Lu, Manpreet Kaur, Zhengyang Qi, Vincent Sunn Chen, Frederic Sala, Dayiheng Liu, Junyang Lin, Zhou Yu, Yu Su, Siva Reddy, Xin Eric Wang, Peng Qi, Tianbao Xie, and Tao Yu. OSWorld 2.0: Benchmarking computer use agents on long-horizon real-world tasks. *arXiv preprint arXiv:2606.29537*, 2026. URL https://arxiv.org/abs/2606.29537.

Guijia Zhang, Yuxun Chen, Yuheng Qi, and Harry Yang. Do GUI agents believe their eyes? diagnosing state-belief reliance on pixels versus structure. *arXiv preprint arXiv:2607.04334*, 2026. URL https://arxiv.org/abs/2607.04334.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/4410c0711e9154a7a2d26f9b3816d1ef-Paper-Conference.pdf.

Xiao Zhou, Siyue Zhang, Yilun Zhao, Jinbiao Wei, Tingyu Song, Arman Cohan, and Chen Zhao. GUI vs. CLI: Execution bottlenecks in screen-only and skill-mediated computer-use agents, 2026. URL https://arxiv.org/abs/2606.24551.

Appendix contents

A Benchmark Reference 15
A.1 Task schema and packaging 15
A.2 Supercategories 15
A.3 Application categories 16
A.4 Tool catalog 17
A.5 Environments and execution 18
A.6 Construction and validation 19
B Metric Suite 21
C Formulation Details 23
D Detailed Experimental Results 24
D.1 Category stratification 24
D.2 Tool stratification 25
D.3 Lifecycle stratification 28
D.4 Failure-mode decomposition 29
D.5 Termination anatomy 30
D.6 Reasoning-mode twins 31
D.7 Effort anatomy 32
D.8 Budget frontiers 33
D.9 The System One decision model in detail 33
E Evaluation Prompts 37
F Case Studies 40

A Benchmark reference

This section is the reference companion to Section 3: the packaging schema of a task (A.1), the supercategories and the application categories they organize (A.2), the full 77-tool catalog (A.4), the execution environment every task runs in (A.5), and the construction and validation behind the frozen r11 version of the benchmark (A.6). Counts are dataset facts checked against the frozen task set: 492 tasks over 77 tools, organized into 19 application categories grouped into 6 supercategories.

A.1 Task schema and packaging

Every task is a self-contained directory that separates what the model sees from what the grader uses. The model-visible side holds exactly one prose artifact, the English instruction; everything else in the directory is machine-readable metadata or private grading material. The grader side is never exposed through the model interface: it contains the object specification, the expected state, negative cases, and the verifier itself.

Listing 1 shows the two sides meeting in one object, the abridged private specification of `tui0831-scene-0001`. The `query` field is the text the model receives verbatim, while the remaining fields drive the grading path: the target object and its location, the operation to verify, the declared lifecycle, and the preservation scope that binds every byte outside the target line. Hashes and boilerplate fields are omitted, and non-English labels are translated:

Listing 1: Abridged task specification for an exact-edit task. The model sees the English query, budgets, recent interaction history, and current terminal viewport; it does not receive this object or its resolved answer fields.

```
{
  "task_id": "tui0831-scene-0001",
  "category": "text editing",
  "tool": "nvi / vi",
  "object_kind": "configuration_file",
  "object_locations": ["/lab/etc/relay.conf"],
  "operation": "replace_explicit_config_value_and_write_quit",
  "lifecycle": "must_exit",
  "query": "Edit /lab/etc/relay.conf with vi, change the only
            listen_port=7000 to listen_port=8127, save and exit;
            all other content ... must be preserved byte for byte.",
  "preserve_scope": [
    "all bytes and line endings outside the target line"
  ]
}
```

A.2 Supercategories

The 6 supercategories organize the 19 application categories into the six kinds of work a server operator would name first, and Figure 3 arranges them as the inner ring of the dataset overview. Table 2 gives the grouping and the task count of every supercategory.

Table 2: The 6 supercategories.

Supercategory	Application categories
Content, 137 tasks	Text editing 60; Paging 43; Search/logs 14; Documents 13; History 7
Files and data, 114 tasks	File management 77; Data grids 28; Databases 9
System administration, 97 tasks	Monitoring 38; Sessions 30; Recovery 18; Package/config 11
Communication, 59 tasks	Mail/chat 43; Calendar/tasks 16
Software development, 56 tasks	VCS/debugging 40; Binary analysis 16
Network and media, 29 tasks	Audio 14; Network/API 9; Network analysis 6
Total, 492 tasks	**19 application categories**

A.3 Application categories

The 492 tasks are organized into 19 application categories, the six supercategories of Appendix A.2 above them, and the 77 tools below them; this subsection fixes the middle level. The categories are chosen to cover the software an operator actually meets on a server. Admission to a category requires that a program be terminal-resident and character-rendered, keep persistent interactive state, and be navigable by keyboard; line-oriented shells, read–eval–print loops, and one-shot command output do not qualify (Section 3.2). Each tool contributes between two and ten tasks, so no single familiar editor can dominate the score. Table 3 gives the distribution, and Figure 6 shows one representative screen for every category, drawn from the initial frame of a recorded episode.

Table 3: Task distribution across the 19 application categories, totaling 492.

Category	Tasks	Category	Tasks
File management and space analysis	77	Partitioning and data recovery	18
Text editing	60	Calendar and task management	16
Mail, chat, and feed reading	43	Binary editing and analysis	16
Paging, manuals, and text browsing	43	Terminal search and log analysis	14
Version control and debugging	40	Audio playback and mixing	14
System and performance monitoring	38	Document viewing and presentation	13
Terminal multiplexing and sessions	30	Installation, packages, and system configuration	11
Data viewing and spreadsheets	28	Network requests and API debugging	9
Database TUIs	9	Terminal productivity and history	7
Network configuration, transfer, and analysis	6	**Total**	**492**

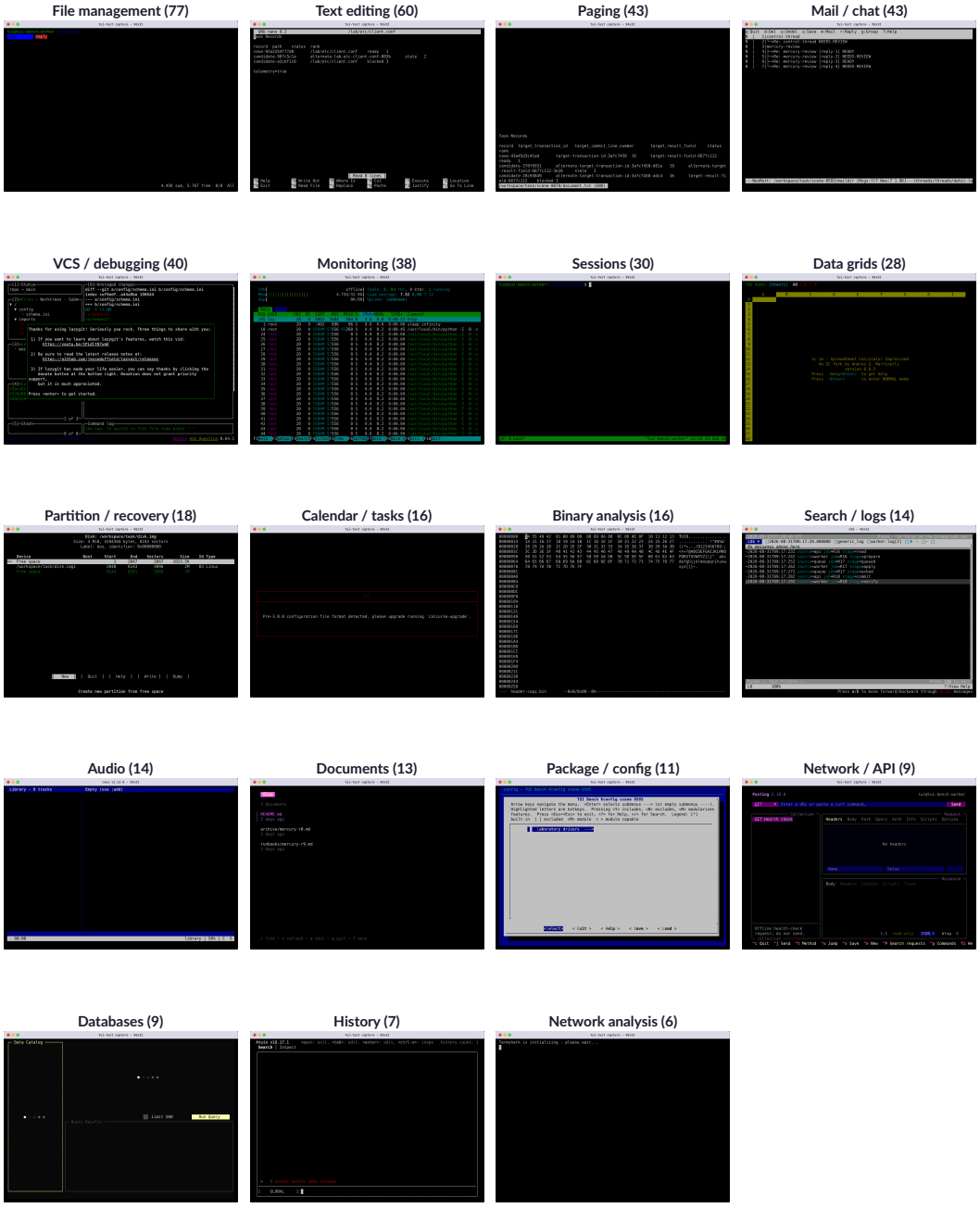


Figure 6: One recorded terminal screenshot per application category, in task-count order; each is the initial frame of a real episode of a representative tool. The full 77-tool catalog is in Table 4.

A.4 Tool catalog

The 77 admitted tools are listed in Table 4, grouped by category. Names follow the benchmark metadata, including interface qualifiers where the same executable supports several modes, as with GDB TUI and radare2 Visual Mode.

Table 4: The complete 77-tool TUI catalog, by category.

Category	Tools
Text editing	GNU Emacs, GNU nano, Helix, JOE, Kakoune, mg, Micro, Neovim, nvi/vi, Vim, Vis
File management and space analysis	broot, dua interactive, gdu, lf, Midnight Commander, ncdu, nnn, ranger, superfile, vifm, xplr, Yazi
Paging, manuals, and text browsing	GNU Info, less, Links, Lynx, more, most, w3m
Version control and debugging	CGDB, GDB TUI, GitUI, lazygit, PuDB, tig
System and performance monitoring	atop, bottom, btop, Glances, htop, nmon, top
Terminal multiplexing and sessions	Byobu, dvtm, GNU Screen, mprocs, tmux, Zellij
Mail, chat, and feed reading	aerc, Alpine, Irssi, NeoMutt, Newsboat, WeeChat
Data viewing and spreadsheets	csvlens, sc-im, VisiData
Partitioning and data recovery	cfdisk, PhotoRec, TestDisk
Binary editing and analysis	hexedit, radare2 Visual Mode
Calendar and task management	calcurse, taskwarrior-tui
Terminal search and log analysis	fzf, lnav
Audio playback and mixing	cmus, ncmpcpp
Document viewing and presentation	Glow, Presenterm
Installation, packages, and system configuration	aptitude, Kconfig menuconfig
Network requests and API debugging	Posting
Database TUIs	Harlequin
Terminal productivity and history	Atuin Search
Network configuration, transfer, and analysis	termshark

A.5 Environments and execution

Every episode runs in a fresh container with the same profile, so the environment is never a variable. Table 5 lists the complete execution contract: the terminal geometry the observation carries (Equation 4), the budgets that bound every episode, the lifecycle split behind the third conjunct of Equation 6, and the isolation properties of the runtime. The 21 category-level images are built once, pinned by digest in each task's launch contract, and reused across rounds; a task's fixture is deterministic, so the only difference between an episode and its repetition is the model's own sampling.

Table 5: The execution contract shared by all 492 tasks.

Property	Value	Where it binds
Terminal geometry	32×96 character grid	The observation G_t of Equation 4; no color or styling survives.
Episode budgets	150 steps, 2,048 input units, 5,400 s	Shared by all tasks (Section 3.1); exhaustion ends the episode.
Lifecycle	461 `must_remain_active`, 31 `must_exit`	Third conjunct of Equation 6.
Execution	`linux_uid_isolated`, TUI uid 10001	All 492 contracts; the TUI runs as an unprivileged identity.
Container	Fresh Docker per episode	21 category images pinned by digest; 2 CPUs, 4 GiB, no network.
Entrypoint	Audited command auto-launched	All 492 contracts open the contracted TUI at its audited entrypoint.
Startup / settle	30 s startup, 5 s settle	Driver timeouts before the first observation is taken.

A.6 Construction and validation

Construction. Each task retains a one-to-one hash lineage to a frozen source query. A data-driven compiler combines the public instruction with private declarative fields for the application, entry point, target object, operation, preservation scope, and lifecycle. A finite family of application adapters then materializes the deterministic fixture, launch contract, expected-state object, and verifier. Formal packaging pins the environment image and runtime wheel before computing task and contract digests. Static gates reject non-English instructions, missing lifecycle declarations, unintended answer leakage, malformed packages, and unbound launch metadata.

Task-specific outcome checks. Scoring is binary, but the verifier is not uniform: the benchmark uses 26 verification modes organized into seven families: structured terminal-surface checks, file and tree transforms, application state, tool-native state, session state, Git state, and structured exit or recovery output. Table 6 lists every mode and its task count. Each task declares one mode, so the family totals sum to 492. Preservation requirements are scoped per task: an exact edit may bind every non-target byte, while a selection task may bind the source artifact and inspect focus tokens in the final terminal surface.

Table 6: Verifier modes by family.

Verifier family	Modes and task counts
Terminal surface, 223 tasks	`structured_screen_state` (214); `native_tui_state` (9)
File and tree transforms, 103 tasks	`exact_file_set_transform` (44); `exact_file_transform` (32); `filesystem_tree_transform` (25); `exact_output_tree` (1); `clean_exit_file_state` (1)
Application state, 73 tasks	`native_application_state_v2` (38); `application_state` (16); `readonly_semantic_selection` (10); `process_state` (5); `nnn_selection_state` (3); `nnn_archive_state` (1)
Tool-native state, 41 tasks	`atuin_native_v2` (7); `taskwarrior_state` (6); `newsboat_state` (6); `ncdu_screen_state` (6); `disk_state` (6); `kconfig_state` (4); `cmus_state` (3); `alpine_screen_state` (3)
Session state, 23 tasks	`session_state` (12); `native_session_state_v2` (11)
Structured exit or recovery output, 16 tasks	`structured_exit_output` (8); `recovery_output` (8)
Git state, 13 tasks	`git_state` (13)
Total, 492 tasks	**26 verifier modes**

Executable admission gates. Every task carries a validity receipt that cryptographically binds the English instruction, its source-query hash, object specification, launch contract, expected state, runtime, and verifier, and a separate task-content digest binds the packaged fixture and execution files. The frozen task set admits a task only after the intended positive fixture passes twice, the untouched initial fixture is rejected, a fixture with the wrong target is rejected, and a fixture with the opposite lifecycle is rejected; all four conditions hold for all 492 tasks. A separate native auto-start gate launches the complete task set and confirms the contracted application entry point.

Human audit. Beyond the executable gates, every task was audited by human experts. For each task we invited three experienced terminal users, each familiar with the target application's keyboard interface, and asked them to solve the task under the same observation and budget as the evaluated models, then rate the task statement for clarity, solvability, and freedom from ambiguity on a five-point scale. The experts reached substantial agreement on their ratings (Cohen's $\kappa = 0.81$, in the sense of Landis & Koch, 1977), and tasks on which any expert reported an issue were returned for revision before the freeze.

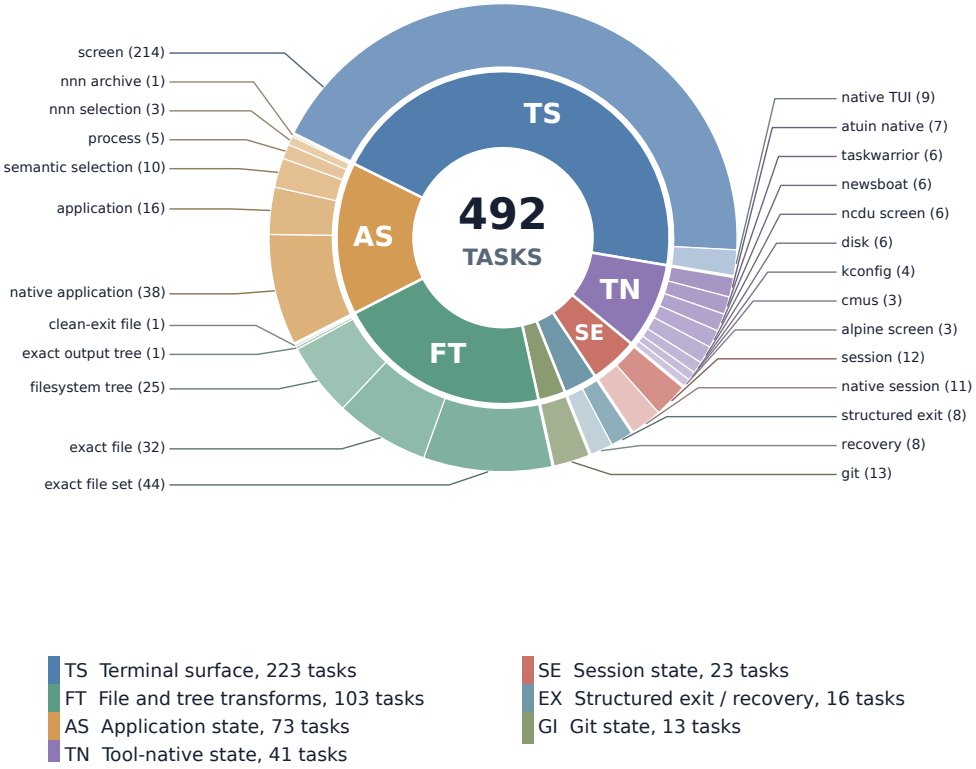


Figure 7: Verifier modes by family as a two-ring breakdown of the 492 tasks: the inner ring gives the 7 verifier families, the outer ring the 26 modes within them. Exact counts are in Table 6.

B Metric suite

The suite is organized around success, reliability, and cost, supplemented by process-level diagnostics that read the interaction loop directly. Table 7 defines the suite. The success and cost aggregates are the headline numbers of Table 1; the reliability pair (Pass@3 against Pass^3) carries the reach and reproducibility reading of Section 5.4, following the pass^k construction of Yao et al. (2025); the rest are process-level diagnostics, stratified per configuration in Appendix D and quoted in Section 5. All rates are computed over scored episodes, that is, episodes whose runtime was valid, so every reported number reflects model behavior. The two success aggregates weight the data differently: Avg@3 is episode-weighted (passes over scored episodes) while Pass@3 is task-weighted (solved tasks over scored tasks), so they coincide only when every scored task contributes a full three scored rounds. Where a task records fewer scored rounds, a task that passes its remaining rounds lifts the episode-weighted rate above the task-weighted one, and Avg@3 can exceed Pass@3 by a few tenths of a point; the opposite ordering holds whenever a solved task loses scored rounds.

Outcome accounting. The runner assigns each attempt one top-level outcome:

`PASS`	The episode and evidence are valid and the verifier accepts the final state.
`MODEL_FAIL`	The runtime is valid, but the model does not produce an accepted state.
`CONFIG_INVALID`	The episode ends because the model's own output exceeds its token cap; counted as a model failure.

Transient execution faults, such as provider, container, or driver faults, are retried under a bounded operational policy, and every such episode is either resolved by the retry policy or recorded with its evidence in the run artifacts. Model failures are not rerun to obtain a more favorable trajectory. Canary and smoke runs are excluded from all aggregates.

Table 7: The TUI-Bench metric suite. $y_{i,r} \in \{0,1\}$ marks a pass of task i in round r; $v_{i,r} \in \{0,1\}$ marks a scored (runtime-valid) episode, so $v_{i,r} = 1$ exactly for PASS, MODEL_FAIL, and CONFIG_INVALID episodes. The success and cost aggregates are the headline numbers of Table 1; the rest are process-level diagnostics.

Metric	Definition	Formula / mechanism	What it reads or resists
Avg@3	Passes over scored episodes; the headline success rate.	$\big(\sum_{i,r} y_{i,r} v_{i,r}\big) / \big(\sum_{i,r} v_{i,r}\big)$	honest success point estimate (episode-weighted)
Pass@3	Share of scored tasks solved in at least one of three recorded passes.	$\frac{1}{\lvert\mathcal{T}\rvert} \sum_i \mathbb{1}[\max_r y_{i,r} v_{i,r} = 1]$	optimistic reach (task-weighted)
Pass^3	Share of scored tasks solved in every scored round.	$\frac{1}{\lvert\mathcal{T}\rvert} \sum_i \mathbb{1}[\text{all scored } r \text{ of } i \text{ pass}]$	reliability floor
Total	Mean total tokens per scored episode.	mean of prompt+completion counts	the context bill of a turn
Read	Mean prompt tokens per scored episode.	mean of the runner's input-token counts; Read + Write = Total	what re-reading the screen costs
Write	Mean completion tokens per scored episode.	mean of the runner's output-token counts	what the model generates per turn
Time	Mean wall-clock seconds per scored episode.	mean of recorded elapsed times	the deployment bill
Keys	Mean keyboard input units per scored episode.	mean of recorded input units against the 2,048 cap	the input bill of the only interface
$\overline{\text{Steps}}_{\text{p}}/\overline{\text{Steps}}_{\text{f}}$	Mean committed steps over passed / failed scored episodes.	means over the outcome split	the cost of a pass and of being lost
$\overline{\text{Tok}}_{\text{p}}/\overline{\text{Tok}}_{\text{f}}$	Mean tokens over passed / failed scored episodes.	means over the outcome split	the effort of success and of failure
Failure buckets	Every scored failure mapped to one of eight modes.	reason-code ladder: truncation → invalid action → budget → finish-without-action → verifier submodes (premature exit / missed exit / wrong state)	the dominant failure fingerprint of Section 5.2
Loop share	Share of ≥10-action failures where a majority of actions repeat an already-seen (action, screen) pair.	#repeat-dominated/#long failures	thrashing without progress
Undo share	Share of `u` presses among all key presses on editor tasks.	#`u`/#keys (editor tasks)	mode-blind restoration
Surface-goal share	Failed screen-specified episodes whose final frame already shows every expected token.	token-subset check on the final frame	screen-grader blindness

C Formulation details

This section expands the formulation of Section 3.1: the observation and action spaces the agent sees, and the terminal-reward conjunction that scores every episode.

Observation and action spaces. Each task is an episode that we model as a partially observable Markov decision process (POMDP) (Åström, 1965; Kaelbling et al., 1998) $\langle \mathcal{Q}, \mathcal{S}, \mathcal{A}, \mathcal{O}, T, r \rangle$, in the form standard for LLM agents acting under environment feedback (Song et al., 2024), with instruction space $\mathcal{Q}$, state space $\mathcal{S}$, action space $\mathcal{A}$, observation space $\mathcal{O}$, deterministic state transition T, and terminal reward $r \in \{0, 1\}$. The hidden state $s_t \in \mathcal{S}$ covers everything the running application holds: its modes, buffers, cursor, and process state. The agent never sees s_t directly; a mode change, a focus switch, or a background mutation alters s_t without announcing itself, and the only evidence is the next rendered screen. This is the structural distinction from web and desktop settings, where a DOM or accessibility tree exposes interface state. At turn t the agent receives

$$o_t \;=\; \big(q,\; b_t,\; h_t,\; G_t,\; c_t\big), \qquad G_t \in \Sigma^{32\times 96}, \tag{4}$$

where $q \in \mathcal{Q}$ is the task instruction, b_t the remaining budgets, h_t the eight most recent interaction records, G_t the current terminal screen as a grid of printable characters over the alphabet Σ, and c_t the cursor position as a 1-based row and column. The grid preserves line structure and within-line spacing but carries no color or styling, and nothing else about the application is exposed: no DOM, no widget labels, no files, no buffers.

The agent replies with exactly one action per turn,

$$a_t \in \big\{\texttt{press}(k_1, \ldots, k_m),\; \texttt{type}(x),\; \texttt{wait_idle},\; \texttt{finish}\big\}, \tag{5}$$

where `press` sends up to 64 named or printable keys including modifiers and function keys, `type` sends printable text exactly as supplied without Enter, and `wait_idle` waits for the screen to settle and returns a fresh observation. The first three actions change the world only through the keyboard; there is no shell, mouse, network, or file access. An LLM-based agent, parameterized by θ, selects $a_t \sim \pi_\theta(\cdot \mid o_0, a_1, o_1, \ldots, a_{t-1}, o_t)$ from the observation-action history. The episode ends when the agent calls `finish`, when the application exits, or when a budget is exhausted, yielding the trajectory $\tau = (s_0, o_0, a_1, s_1, o_1, \ldots, a_T, s_T)$. All tasks share the same budgets: 150 model steps, 2,048 keyboard input units, and 5,400 seconds of wall-clock time.

Terminal reward. Scoring happens after the episode, not during it: the reward $r(\tau)$ is computed once from the final state s_T and the persistent artifacts. Writing D_0 and D_T for those artifacts before and after, and $\ell \in \{\texttt{must_remain_active}, \texttt{must_exit}\}$ for the declared lifecycle, a task passes only when

$$\text{state}(s_T) \;\wedge\; \text{preserve}(D_0, D_T) \;\wedge\; \text{lifecycle}(s_T, \ell) \tag{6}$$

all hold, where state is a task-specific check of the final screen or application state, preserve binds the files and state the task declares untouchable, and lifecycle requires the program to still be running for 461 tasks and to have exited for 31. The three conjuncts are scored independently, so an agent that reaches the right screen but dirties a preserved file, or that closes a program it was asked to leave open, fails the task. The runtime classifies each episode as PASS or MODEL_FAIL, with execution faults governed by a bounded retry policy; an episode that ends because the model's own output exceeded its token cap counts as a model failure.

D Detailed experimental results

Every table and figure in this section plots the complete campaign for the 20 generative configurations of Table 1 over the full 492-task benchmark, three recorded passes each, decomposing the headline numbers along three axes: where success survives (D.1–D.2), how episodes end (D.3–D.5), and what they cost (D.6–D.8). The System One decision model Jev-latest is excluded from every table and figure of D.1–D.8 and analyzed on its own terms in D.9; it remains in Table 1 as its own class. Rates are in percent of scored episodes (Table 7); a dash marks a cell with no scored episode.

D.1 Category stratification

Table 8 gives the full category-by-configuration success matrix behind Figure 4 and the category accounting of Section 5.1: 19 columns ordered by pooled success, one row per generative configuration. The compact main-paper view samples this complete ordering, retaining the lowest nonzero category (network analysis, 0.7%) and all four zero-success categories. The direct-entry split contains 244 scored tasks where text entry is present and 223 where it is absent. Their pooled success rates are 16.95% and 6.78%, respectively, giving a difference of 10.17 percentage points. The cliff of Section 5.1 is row-structural: database TUIs pass for every generative configuration while whole column blocks record zero across the field. The single bright exception in the history column is the Atuin search task, the one command-shaped member of an otherwise navigation-bound category. Figure 8 is the same matrix as a heatmap, with the zero-success categories marked by the rose band. The cliff sharpens one level down: the tool-level tables of Appendix D.2 show the split runs inside categories, not only between them.

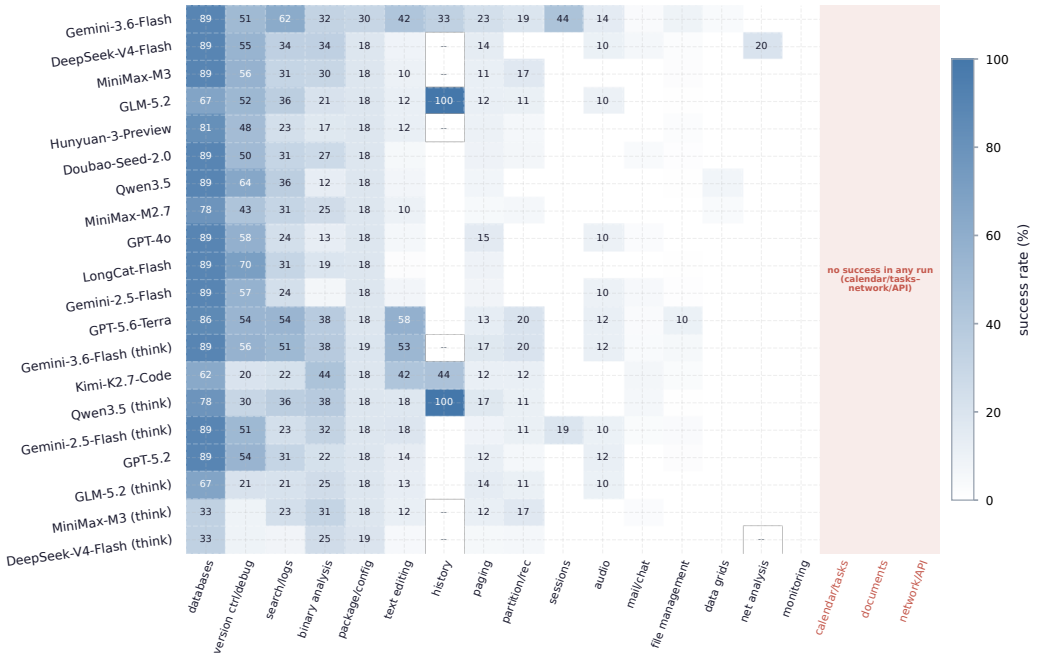


Figure 8: Success rate by application category (columns, ordered by pooled success) and configuration (rows, in table order). The rose band marks categories with no success in any run.

Table 8: Success rate in percent by application category (columns, ordered by pooled success) and generative configuration (rows), over scored episodes of the three recorded passes. Generative configurations only; the System One decision model is analyzed separately in Appendix D.9. A dash marks a cell with no scored episode. Per column, the best value is **bold** and the worst is underlined when unique enough to be informative. Column abbreviations follow Table 3.

Model	DBs	VC	Srch	Bin	Pkg	Edit	Hist	Page	Part	Sess	Aud	Mail	File	Grid	Net	Mon	Cal	Doc	API
Non-thinking models																			
Gemini-2.5-Flash	89	57	24	6.2	18	6.7	0.0	7.5	0.0	0.0	10	4.9	1.2	0.0	0.0	0.0	0.0	0.0	0.0
LongCat-Flash-Chat	89	**70**	31	19	18	1.7	0.0	9.3	0.0	0.0	0.0	0.0	0.6	0.0	0.0	0.0	0.0	0.0	0.0
GPT-4o	89	58	24	13	18	6.1	0.0	15	0.0	0.0	10	2.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0
MiniMax-M2.7	78	43	31	25	18	10	0.0	4.7	5.6	0.0	0.0	0.0	0.0	3.6	0.0	0.0	0.0	0.0	0.0
Qwen3.5	89	64	36	12	18	6.7	0.0	7.1	0.0	0.0	0.0	0.0	0.5	**7.2**	0.0	0.0	0.0	0.0	0.0
Doubao-Seed-2.0-Lite	89	50	31	27	18	5.1	0.0	9.3	6.2	0.0	0.0	3.8	0.9	0.0	0.0	0.0	0.0	0.0	0.0
Hunyuan-3-Preview	81	48	23	17	18	12	–	9.8	5.9	0.0	0.0	0.0	2.1	0.0	0.0	0.0	0.0	0.0	0.0
GLM-5.2	67	52	36	21	18	12	**100**	12	11	0.0	10	0.0	0.6	0.0	0.0	0.0	0.0	0.0	0.0
MiniMax-M3	89	56	31	30	18	10	–	11	17	0.0	0.0	0.0	1.6	0.0	0.0	0.0	0.0	0.0	0.0
DeepSeek-V4-Flash	89	55	34	34	18	8.6	–	14	0.0	0.0	10	6.1	2.2	0.0	**20**	0.0	0.0	0.0	0.0
Gemini-3.6-Flash	89	51	**62**	32	**30**	42	33	**23**	19	**44**	**14**	3.3	8.8	3.8	0.0	0.0	0.0	0.0	0.0
Thinking models																			
DeepSeek-V4-Flash	33	8.7	7.7	25	19	5.0	–	5.1	5.6	0.0	0.0	0.0	0.0	0.0	–	0.0	0.0	0.0	0.0
MiniMax-M3	33	8.8	23	31	18	12	–	12	17	0.0	0.0	2.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0
GLM-5.2	67	21	21	25	18	13	0.0	14	11	0.0	10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
GPT-5.2	89	54	31	22	18	14	0.0	12	6.1	0.0	12	0.0	1.7	0.0	0.0	0.0	0.0	0.0	0.0
Gemini-2.5-Flash	89	51	23	32	18	18	0.0	8.4	11	19	10	2.8	3.3	0.0	0.0	0.0	0.0	0.0	0.0
Qwen3.5	78	30	36	38	18	18	**100**	17	11	0.0	0.0	5.9	0.7	0.0	0.0	0.0	0.0	0.0	0.0
Kimi-K2.7-Code	62	20	22	**44**	18	42	44	12	12	0.0	0.0	**6.3**	2.9	0.0	0.0	0.0	0.0	0.0	0.0
Gemini-3.6-Flash	89	56	51	38	19	53	–	17	**20**	0.0	12	3.6	5.3	0.0	0.0	0.0	0.0	0.0	0.0
GPT-5.6-Terra	86	54	54	38	18	**58**	0.0	13	**20**	0.0	12	3.1	**10**	0.0	0.0	0.0	0.0	0.0	0.0

D.2 Tool stratification

Table 9 and Figure 9 give the tool-level decomposition behind the pooled claims of Section 5.1: among the 73 tools with at least 50 pooled scored episodes to judge, 32 never pass in any of the 60 generative runs. The structure the category view hides is the split *inside* categories: file management contains both GNU nano at 44.2% and the four file managers Yazi, ranger, broot, and xplr at zero; the grid-viewing category contains Harlequin at 78.2%, the brightest cell in the whole campaign, alongside sc-im at 1.7% and csvlens at 0.6%. Tool breadth is narrow almost everywhere: only five of the 25 largest tools are solved by all twenty configurations (Harlequin, fzf, GNU nano, more, and GNU Info), and most sit below eleven solvers. One disclosure on the count: it is one larger than the 31 of Section 5.1 because that field figure includes the System One decision model, whose two htop passes make it the campaign's only htop solver (Appendix D.9); over the generative configurations alone, htop joins the zero-pass list. Tools with fewer than 50 pooled scored episodes are excluded from the judgment; three of them record passes (tmux 6/32, mprocs 12/15, ncdu 3/9).

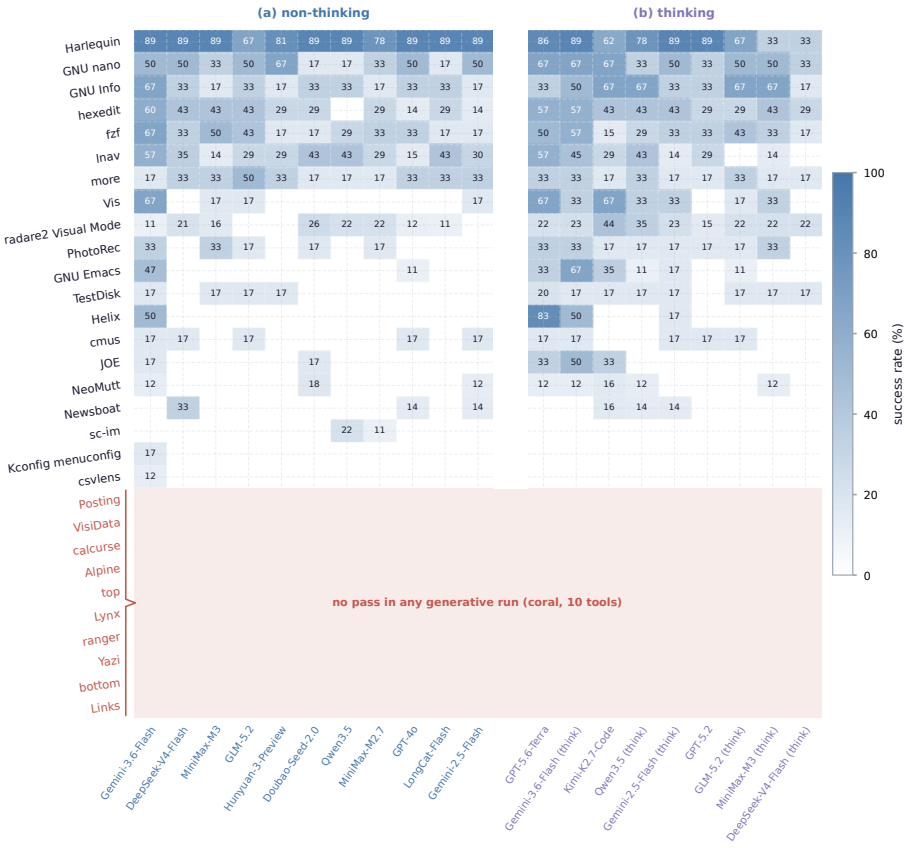


Figure 9: Success rate by tool (rows, the 30 largest tools ordered by pooled success, zero-pass tools at the bottom in coral) and generative configuration (columns); a dash marks no scored episode.

Table 9: Tool-level success for the 25 tools with the most scored episodes, pooled over the 20 generative configurations and their three passes. Solvers counts the configurations with at least one pass; category abbreviations follow Table 8.

Tool	Category	Scored	Pass	SR (%)	Solvers
sc-im	Grid	540	9	1.7	2/20
Posting	API	538	0	0.0	0/20
Harlequin	DBs	531	415	**78.2**	20/20
VisiData	Grid	510	0	0.0	0/20
csvlens	Grid	497	3	0.6	1/20
radare2 Visual Mode	Bin	486	93	19.1	17/20
calcurse	Cal	478	0	0.0	0/20
NeoMutt	Mail	455	24	5.3	8/20
hexedit	Bin	419	147	35.1	19/20
Alpine	Mail	417	0	0.0	0/20
lnav	Srch	413	123	29.8	18/20
Newsboat	Mail	406	21	5.2	6/20
top	Mon	401	0	0.0	0/20
Lynx	Page	395	0	0.0	0/20
ranger	File	380	0	0.0	0/20
Yazi	File	379	0	0.0	0/20
bottom	Mon	379	0	0.0	0/20
fzf	Srch	377	126	33.4	20/20
GNU nano	Edit	360	159	44.2	20/20
Helix	Edit	360	36	10.0	4/20
JOE	Edit	360	27	7.5	5/20
Vis	Edit	360	72	20.0	11/20
more	Page	360	96	26.7	20/20
GNU Info	Page	360	138	38.3	20/20
PhotoRec	Part	360	54	15.0	13/20

D.3 Lifecycle stratification

Table 10 prices the third conjunct of Equation 6 per configuration, behind the lifecycle sentence of Section 5.2: success on tasks that must end with a clean exit against success on tasks whose program must stay alive. The split is not an aggregate artifact: every one of the twenty generative configurations passes must-exit tasks more often than must-remain-active ones, from 0.9 points (LongCat-Flash-Chat) to 43.2 points (GPT-5.6-Terra). Pooled over the generative field the split is 31.8% against 10.4%; the 30.3% of Section 5.2 additionally includes the System One model. Two disclosures keep the column honest: must-exit is a thin slice of the benchmark (31 tasks, roughly ninety scored episodes per configuration), so the small gaps sit inside sampling noise, and the Premature column gives the mechanism: the configurations that drag must-remain-active down are the ones that lose the largest share of their scored failures to premature exits. Figure 10 plots the same split as paired bars ordered by the gap; the field reads as a staircase, and the *Premature* column of the table names the step each configuration trips on.

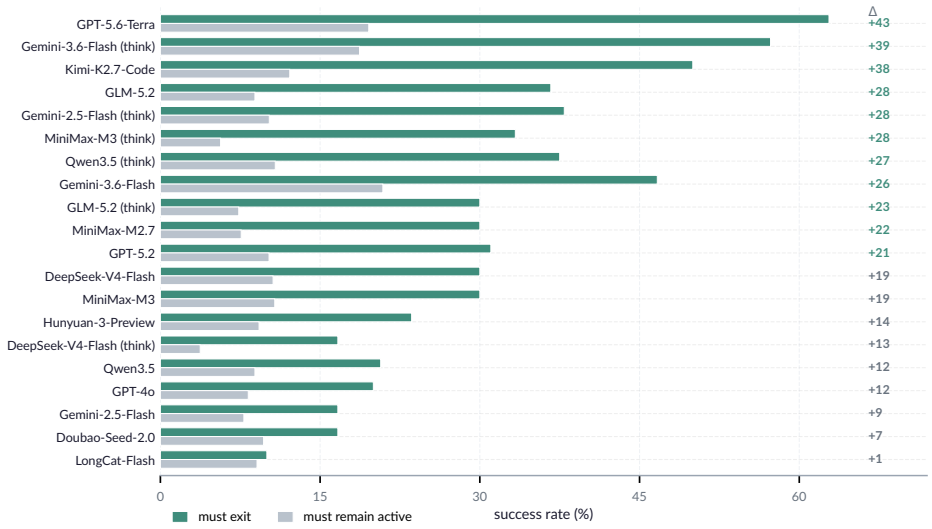


Figure 10: Success on must-exit versus must-remain-active tasks per generative configuration; rows sorted by the gap Δ.

Table 10: Success rate by declared lifecycle (the third conjunct of Equation 6), per generative configuration, with the share of scored failures ended by premature and missed exits. Δ = must-exit minus must-remain-active success.

	Success by lifecycle (%)			*Exit-failure share (%)*	
Model	Must exit	Remain	Δ	Premature	Missed
Non-thinking models					
Gemini-2.5-Flash (w/o thinking)	16.7	7.8	+8.8	5.9	0.3
LongCat-Flash-Chat	10.0	9.1	+0.9	3.8	0.0
GPT-4o	20.0	8.3	+11.7	12.8	0.0
MiniMax-M2.7	30.0	7.6	+22.4	5.3	0.3
Qwen3.5 (w/o thinking)	20.7	8.9	+11.8	4.5	0.3
Doubao-Seed-2.0-Lite	16.7	9.7	+7.0	8.4	0.3
Hunyuan-3-Preview	23.6	9.3	+14.3	5.2	0.2
GLM-5.2 (w/o thinking)	36.7	8.9	+27.8	4.4	2.3
MiniMax-M3 (w/o thinking)	30.0	10.7	+19.3	12.2	1.0
DeepSeek-V4-Flash (w/o thinking)	30.0	10.6	+19.4	22.0	0.0
Gemini-3.6-Flash (w/o thinking)	46.7	20.9	+25.8	9.0	1.0
Thinking models					
DeepSeek-V4-Flash (w/ thinking)	16.7	3.7	+12.9	8.9	0.0
MiniMax-M3 (w/ thinking)	33.3	5.7	+27.7	4.5	0.0
GLM-5.2 (w/ thinking)	30.0	7.4	+22.6	2.7	0.8
GPT-5.2	31.0	10.2	+20.8	18.4	0.0
Gemini-2.5-Flash (w/ thinking)	37.9	10.2	+27.7	16.3	0.0
Qwen3.5 (w/ thinking)	37.5	10.8	+26.7	6.0	0.0
Kimi-K2.7-Code	50.0	12.2	+37.8	13.4	0.0
Gemini-3.6-Flash (w/ thinking)	57.3	18.7	+38.6	4.3	1.4
GPT-5.6-Terra	62.8	19.6	**+43.2**	6.8	0.0

D.4 Failure-mode decomposition

Table 11 decomposes every generative configuration's scored failures into the eight buckets of Table 7, as shares of that configuration's failures; per column the lowest share is bold and the highest is underlined. The table is the generative-configuration reference behind Figure 5a and Section 5.2: each configuration has one dominant way to lose the interface, and the dominant mode differs across the field: thrashing for the budget-bound models, truncation for the reasoning-heavy ones, wrong final state for the best-behaved. Figure 11 gives the stacked view of the same decomposition.

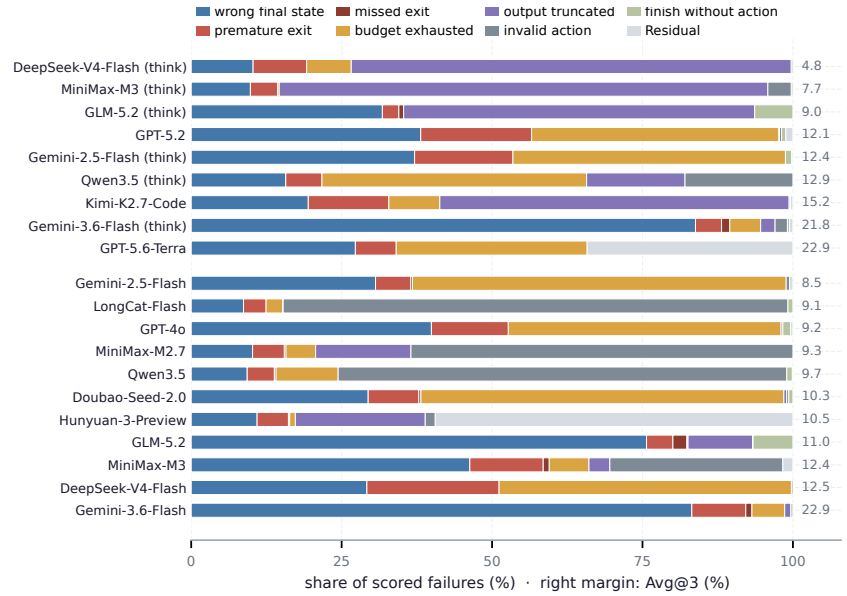


Figure 11: Failure-mode decomposition per configuration, stacked shares of scored failures; the right margin gives Avg@3. The same data as Table 11.

Table 11: Failure-mode decomposition: share of each configuration's scored failures in each bucket of Table 7, in percent. Generative configurations only; the System One decision model is analyzed separately in Appendix D.9. Within each block rows are ordered by increasing Avg@3. Per column, the lowest share is **bold** and the highest is underlined when unique enough to be informative.

Model	Wrong state	Premature exit	Missed exit	Budget	Truncated	Invalid action	Finish w/o act.	Other
Non-thinking models								
Gemini-2.5-Flash (w/o thinking)	30.6	5.9	0.3	62.1	0.0	0.6	0.0	0.6
LongCat-Flash-Chat	**8.7**	3.8	0.0	2.7	0.1	83.9	0.8	0.0
GPT-4o	39.9	12.8	0.0	45.3	0.0	0.3	1.3	0.4
MiniMax-M2.7	10.2	5.3	0.3	4.9	15.8	63.5	0.0	0.0
Qwen3.5 (w/o thinking)	9.3	4.5	0.3	10.3	0.0	74.6	0.9	0.1
Doubao-Seed-2.0-Lite	29.4	8.4	0.3	60.3	0.5	0.3	0.7	0.0
Hunyuan-3-Preview	10.9	5.2	0.2	0.9	21.6	1.7	0.0	59.4
GLM-5.2 (w/o thinking)	75.7	4.4	2.3	0.2	10.7	0.1	6.6	0.0
MiniMax-M3 (w/o thinking)	46.3	12.2	1.0	6.6	3.5	28.7	0.0	1.7
DeepSeek-V4-Flash (w/o thinking)	29.2	22.0	0.0	48.6	0.0	0.2	0.0	0.0
Gemini-3.6-Flash (w/o thinking)	83.2	9.0	1.0	5.4	1.0	0.0	0.3	0.0
Thinking models								
DeepSeek-V4-Flash (w/ thinking)	10.3	8.9	0.0	7.4	73.1	0.0	0.3	0.0
MiniMax-M3 (w/ thinking)	9.8	4.5	0.0	0.3	81.2	3.9	0.3	0.0
GLM-5.2 (w/ thinking)	31.8	**2.7**	0.8	**0.0**	58.3	0.0	6.4	0.0
GPT-5.2	38.1	18.4	0.0	41.1	0.1	0.4	0.7	1.2
Gemini-2.5-Flash (w/ thinking)	37.1	16.3	0.0	45.3	0.0	0.0	1.0	0.2
Qwen3.5 (w/ thinking)	15.7	6.0	0.0	44.0	16.4	17.9	0.0	0.0
Kimi-K2.7-Code	19.5	13.4	0.0	8.5	58.1	0.0	0.2	0.4
Gemini-3.6-Flash (w/ thinking)	83.8	4.3	1.4	5.1	2.4	2.1	0.3	0.6
GPT-5.6-Terra	27.3	6.8	0.0	31.7	0.0	0.0	0.0	34.2

D.5 Termination anatomy

Figure 12 unfolds the four words of the Termination group into one donut per generative configuration, ordered by the Finish share. The grid is the failure archetypes of Section 5.2 seen end-on: the protocol-collapsed ring of LongCat-Flash-Chat and MiniMax-M2.7 is nearly a closed slate circle,

the reasoning-heavy MiniMax-M3 and DeepSeek-V4-Flash thinking modes carry slate majorities of their own, the thrasher Gemini-2.5-Flash wears the widest amber wedge, and the two Gemini-3.6-Flash donuts are the field's most decisive, with more than four fifths of their episodes ending by their own declaration. GPT-5.6-Terra is the only configuration with two fat wedges at once, Finish and Cap split nearly even: the strongest model both declares and gets cut off more than most of the field. The System One decision model is excluded from this grid and dissected on its own terms in Appendix D.9, where its donut, which has no slate wedge at all and the field's widest coral, becomes the paradigm boundary itself.

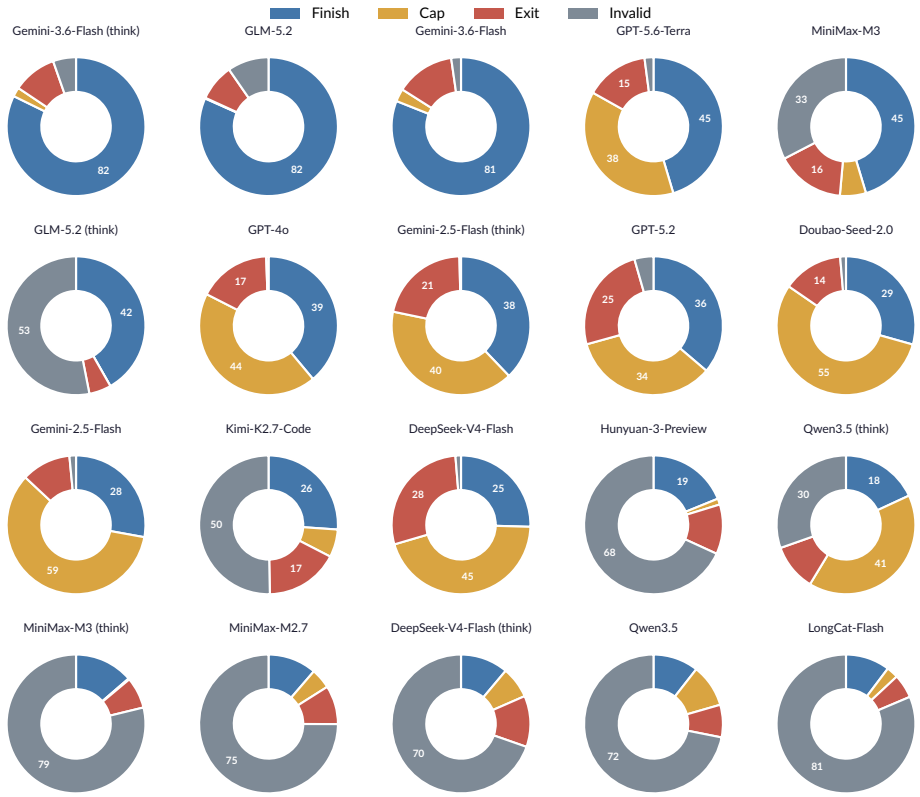


Figure 12: How each generative configuration's scored episodes ended: one donut per configuration, ordered by Finish share.

D.6 Reasoning-mode twins

Table 12 provides the complete paired values behind the reasoning-overflow discussion of Section 5.2: for all six families evaluated in both modes, Avg@3, the truncation share of scored failures, and the mean completion tokens. Enabling thinking lowers Avg@3 for 4 of the six families (DeepSeek-V4-Flash -7.7, MiniMax-M3 -4.7, GLM-5.2 -2.0, Gemini-3.6-Flash -1.2) and raises it for 2 (Gemini-2.5-Flash $+3.9$, Qwen3.5 $+3.2$). Wherever the truncation share moves at all, it moves up on the thinking side: from 3.5% to 81.2% for MiniMax-M3, from 0.0% to 73.1% for DeepSeek-V4-Flash, and from 10.7% to 58.3% for GLM-5.2. The Write column shows the input to the cap: Qwen3.5's completion volume grows from 1.2k to 36.7k tokens per episode. Of the two families that improve, Gemini-2.5-Flash is the clean exception: it writes 0.4k tokens per episode in both modes and never truncates, so enabling thinking costs it nothing, while Qwen3.5 gains even while paying the tax (16.4% of its thinking-mode failures truncate). The rows of Figure 5 that fall outside the twin table are the unpaired ones: Kimi-K2.7 has no non-thinking twin, and the System One decision model is analyzed separately in Appendix D.9. Kimi-K2.7's truncation-heavy escape route is priced in Table 13. Hunyuan-3-Preview also runs only a non-thinking configuration and is not part of the twin comparison.

Table 12: Thinking and non-thinking twins: Avg@3, truncation share of scored failures, and mean completion tokens.

	Avg@3 (%)			*Trunc. share (%)*		*Write (tokens)*	
Family	w/o	w/	Δ	w/o	w/	w/o	w/
GLM-5.2	11.0	9.0	-2.0	10.7	58.3	1k	8k
MiniMax-M3	12.4	7.7	-4.7	3.5	81.2	7k	12k
Qwen3.5	9.7	12.9	+3.2	0.0	16.4	1k	37k
Gemini-2.5-Flash	8.5	12.4	+3.9	0.0	0.0	407	360
Gemini-3.6-Flash	22.9	21.8	-1.2	1.0	2.4	735	859
DeepSeek-V4-Flash	12.5	4.8	-7.7	0.0	73.1	4k	11k
Field mean	11.4	13.2	+1.8	4.8	32.2	3k	12k

D.7 Effort anatomy

Table 13 reports the per-configuration process diagnostics of Table 7: mean steps and tokens split by outcome, the loop share of long failures, and the undo share on editor tasks. The columns price the fingerprints of Section 5.2 separately: a thrashing model spends steps, a reasoning model spends tokens, and a protocol-collapsed model spends almost nothing at all. Figure 13 plots the pass-versus-fail means directly as scatter pairs on log–log axes: every configuration sits above the diagonal, so failure always costs more than success, and the vertical distance to the diagonal is the asymmetry. The cheapest successes sit in the lower-left corner, where GLM-5.2's thinking mode and LongCat-Flash-Chat pass in a handful of steps and tens of thousands of tokens, while Doubao-Seed-2.0 and Gemini-2.5-Flash burn an order of magnitude more per failure than per pass.

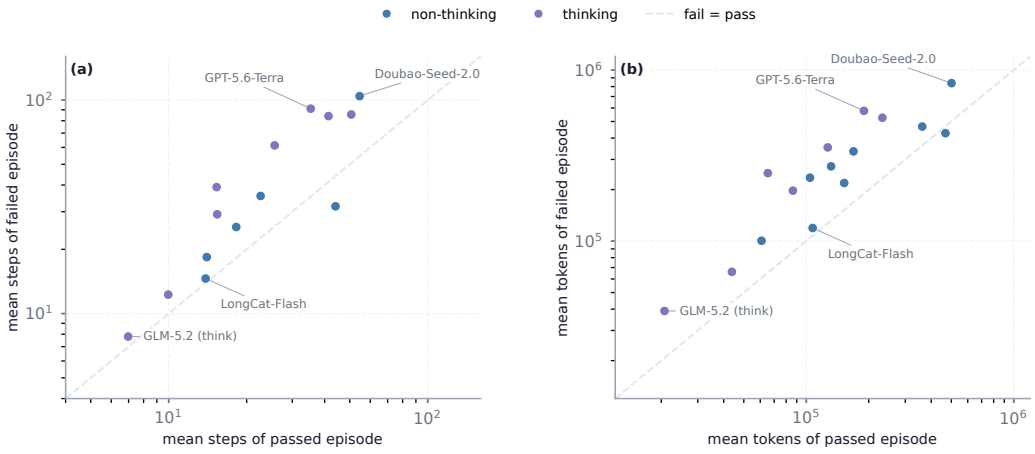


Figure 13: Failure against success, per generative configuration: (a) mean steps of failed vs. passed episodes, (b) mean tokens, both on log–log axes. Overlapping unlabeled markers are omitted; the complete per-configuration data appear in Table 13.

Table 13: Effort anatomy per generative configuration: mean steps of passed and failed scored episodes, mean tokens of passed and failed scored episodes, the loop share of failures with at least ten actions (majority of actions repeating a seen (action, screen) pair), and the share of `u` presses among all key presses on editor tasks. Generative configurations only; the System One decision model is analyzed separately in Appendix D.9. Within each block rows are ordered by increasing Avg@3. Per column, the lowest value is **bold** and the highest is underlined when unique enough to be informative.

	$\overline{Steps}$		$\overline{Tok}$		*Process diagnostics*	
Model	Pass	Fail	Pass	Fail	Loop	`u` share
Non-thinking models						
Gemini-2.5-Flash (w/o thinking)	63	105	424k	693k	64.0	**0.0**
LongCat-Flash-Chat	14	15	107k	119k	33.6	2.1
GPT-4o	63	90	362k	467k	38.8	4.4
MiniMax-M2.7	18	25	152k	218k	33.7	1.4
Qwen3.5 (w/o thinking)	44	32	469k	427k	54.8	4.3
Doubao-Seed-2.0-Lite	55	104	501k	838k	38.4	2.5
Hunyuan-3-Preview	23	36	104k	235k	43.3	4.0
GLM-5.2 (w/o thinking)	14	18	61k	100k	14.9	4.2
MiniMax-M3 (w/o thinking)	26	50	169k	334k	25.9	8.1
DeepSeek-V4-Flash (w/o thinking)	63	94	447k	722k	32.5	5.4
Gemini-3.6-Flash (w/o thinking)	29	49	132k	273k	23.9	1.4
Thinking models						
DeepSeek-V4-Flash (w/ thinking)	15	29	86k	197k	34.0	2.8
MiniMax-M3 (w/ thinking)	10	12	44k	66k	21.8	2.7
GLM-5.2 (w/ thinking)	**7**	**8**	**21k**	**39k**	**4.5**	2.0
GPT-5.2	41	84	233k	526k	44.3	2.1
Gemini-2.5-Flash (w/ thinking)	35	87	195k	577k	36.1	4.3
Qwen3.5 (w/ thinking)	51	86	471k	782k	51.7	3.8
Kimi-K2.7-Code	15	39	65k	249k	14.1	5.1
Gemini-3.6-Flash (w/ thinking)	26	61	127k	353k	30.8	1.1
GPT-5.6-Terra	35	91	190k	577k	25.1	4.0

D.8 Budget frontiers

Figure 14 gives each generative configuration its own panel: the share of its scored episodes that both passed and cost at most the budget on the x axis, as the budget sweeps from 10k to 3M tokens on a log scale. The curves are the cost–success frontiers behind the Read, Write, and Time columns of Table 1: the height each curve settles at is that configuration's Avg@3, and the knee is where its productive budget ends, because past it, tokens buy step-cap episodes, not solutions. The two leaders spend very differently: Gemini-3.6-Flash reaches its plateau by 300k tokens while GPT-5.6-Terra is still climbing at 1M, the token version of the time-and-keys contrast of Section 4.2. GLM-5.2's thinking mode tops out earliest, with nearly all of its scarce passes arriving under 100k tokens, while Doubao-Seed-2.0 and Gemini-2.5-Flash stretch their frontiers to the right edge: most of what they spend, they spend failing. The System One decision model is omitted here because its 1.5M-token episodes would flatten the shared axis; its frontier is a single point at the far right, priced in Appendix D.9.

D.9 The System One decision model in detail

What kind of model it is. Jev-latest is the campaign's one non-generative configuration: the reference implementation of a *System One decision model* (Jev 1.13, TypeSafe AI). The architectural difference from every other row of Table 1 is the output space. A generative LLM samples free text token by token and an application parses the result; Jev inverts the contract. The caller fixes the answer space first, whether a *Choice* among enumerated options, a *Score* on a scale, or a *Noul* judgment of whether a proposition holds, and the model returns a probability distribution over that space. It cannot emit a value outside the schema, and it emits no text at all. It was trained for exactly this regime (RLCD, reinforcement learning for calibrated decisions), and its design position is

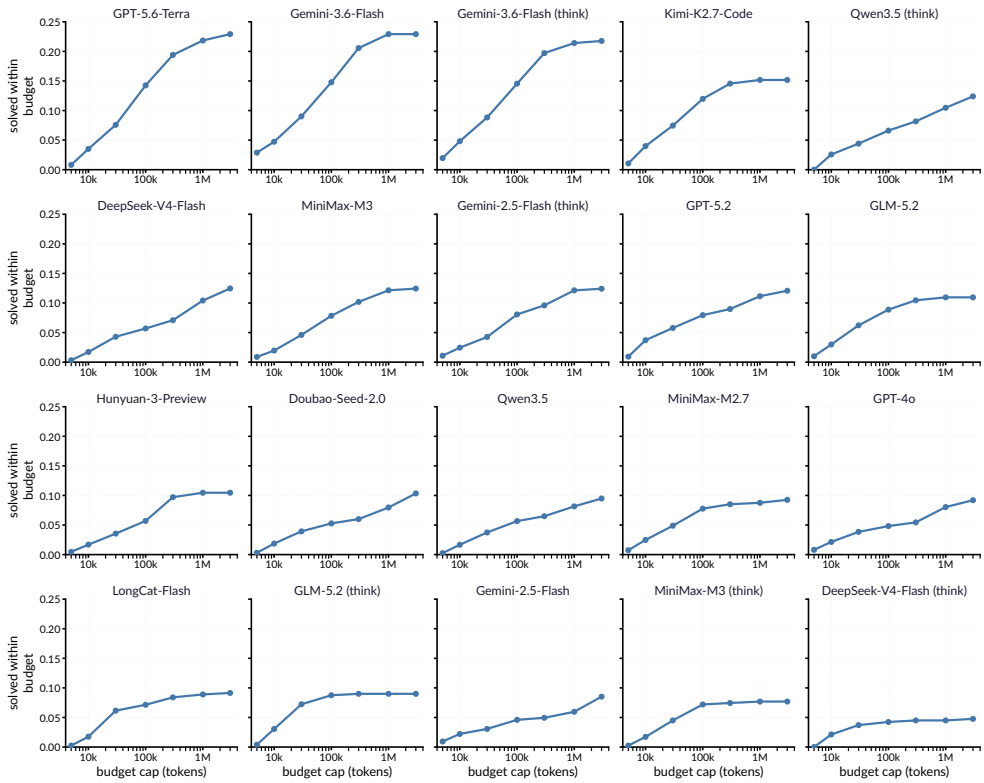


Figure 14: Cost–success frontiers, one panel per generative configuration, ordered by Avg@3: the share of scored episodes that pass within each token budget (log axis).

not *replacing* generative LLMs but taking over the high-frequency, low-entropy judgments inside an agent loop, such as routing, risk scoring, continuation, and stopping, where a generative call is wasted. TUI-Bench pressures that position in the one place it has not been tested: an interface where every judgment the model makes must compose into a multi-step manipulation of a hidden state machine. For this campaign the four-tool action space is translated into that decision interface; the comparison is kept as fair as the architecture allows, but it is controlled in configuration rather than in mechanism, so the Jev-latest row is an architecture contrast, not a strictly controlled comparison against the generative field. The translation layer, spelled out next, is what makes the comparison as fair as it is.

The translation layer. A loopback adapter presents Jev-latest to the shared TUI-Bench controller as a standard chat-completions endpoint, so the controller, the evidence chain, and the grading path run unmodified. Everything the generative configurations receive is preserved byte for byte: the system prompt of Appendix E and the per-turn payload are concatenated verbatim into the model's state document, with no restructuring, truncation, or reordering, and the profile is the same frozen revision (150 steps, 2,048 input units, 5,400 seconds, the 32×96 viewport, and the eight-turn verbatim history). What the adapter changes is the output side alone, and it does so by turning the action space itself into the question set. Each turn issues one System One fan-out of six typed questions: a *Choice* among the four tool families; a *Choice* over every atomic key, all printable ASCII, the named navigation keys, F1–F24, and the classic vi compounds, 148 options; a *Choice* over 112 modifier chords, because a chord is a single key event and cannot be composed across turns; a *Noul* on whether a chord beats a single key; a *Score* for the repeat count; and a *Choice* over at most 40 candidate strings for `type`, mined from the task instruction and the current screen. Because cross-turn composition is free under the verbatim history, atomic-key coverage makes every finite key sequence reachable; the residual unreachable set, three-modifier chords and SHIFT+F13–F24, touches no keybinding any of the 77 tools uses. Two adapter-side mechanisms have no generative analogue and are disclosed as such: when the same screen repeats four times, a stuck warning is appended to the document and the argmax is replaced by sampling from the model's own calibrated

top-k distribution, the structural analogue of temperature, needed because deterministic argmax on a deterministic model limit-cycles; and the `type` candidates are mined adapter-side from the verbatim text the model already sees. The question instructions and the arbitration between the key and chord answers are reproduced in Appendix E.

Where it lands. It was recorded over three independent passes of the full 492-task benchmark.

Where a decision model passes. Table 14 and Figure 15 answer: all 30 solved tasks live in 9 tools, and every one of them exposes a place to type a command, whether Harlequin's query box, the GDB TUI, CGDB, and PuDB command lines, or aptitude's search. Command-shaped tasks reduce the interface to a sequence of typed strings and short navigations, and a policy that scores one option at a time can solve them. The htop task it solves, which asks the agent to select the synthetic process, adjust its nice value to the requested level, and keep the process visible, is decidable one keystroke at a time from the current screen, and it makes Jev-latest the campaign's only htop solver: the same tool where Gemini-2.5-Flash's up-arrow storm burns 132 steps (Section 5.1) yields to a policy that never plans farther than the next key (Case 8).

The paradigm boundary. With no text generation, the field's two dominant failure families are structurally absent: zero truncated outputs and zero invalid actions, because a structured decision cannot overflow the output cap and cannot violate a schema it fills by construction. This is the boundary the vendor language of "zero hallucinations" actually names: type errors are eliminated at the interface, semantic errors are not, and every failure Jev-latest records here is a legal decision that was simply wrong. What remains is thrashing: 62% of its scored failures exhaust the step budget, the median episode, whether passed or failed, sits at the step cap, and the mean episode bills 1.5M tokens, the field's largest read and write totals (Table 1). Two mechanics inflate that bill. The read side is the same context re-entry every generative configuration pays, paid for more steps. The write side is an accounting artifact of the decision interface worth stating honestly: the shim serializes the full scored option set into each response, so completion tokens record the enumerated answer space, not generated reasoning, at 2.6k tokens per call and 150 calls per cap-length episode. Its lifecycle split is the generative field's exact mirror: 0.0% on must-exit tasks against 9.2% on must-remain-active, where every generative configuration runs the other way (Appendix D.3): a policy that scores one atomic action at a time never plans the exit. The atomic action space itself is complete, so any finite key sequence is reachable, but each decision looks one step ahead, and cross-step plans, such as positioning the cursor and then deleting the word, stay out of reach. That is the same missing capacity Section 5.3 traces across the generative field, seen from the opposite side: reading the screen well is not operating the interface, with or without generation.

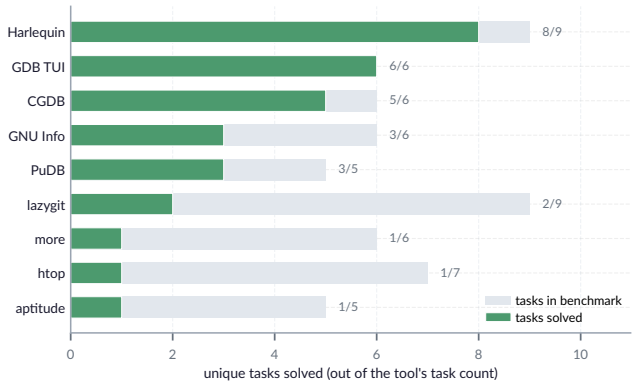


Figure 15: Unique tasks solved by the System One decision model, per tool.

Table 14: Where the System One decision model passes: every tool with at least one pass in its three recorded passes, with pooled scored episodes, passes, and unique tasks solved out of the tool's task count.

Tool	Category	Scored	Passes	Tasks	SR (%)
Harlequin	DBs	27	24	8/9	88.9
GDB TUI	VC	18	18	6/6	100.0
CGDB	VC	18	15	5/6	83.3
PuDB	VC	15	9	3/5	60.0
GNU Info	Page	18	5	3/6	27.8
lazygit	VC	20	4	2/9	20.0
more	Page	18	3	1/6	16.7
htop	Mon	18	2	1/7	11.1
aptitude	Pkg	7	1	1/5	14.3
Total	—	159	81	30	50.9

E Evaluation prompts

The agent is driven by one system prompt shared by all 21 configurations, a per-turn user payload assembled by the controller, and a four-tool keyboard contract. Nothing in any prompt names an application, a keybinding, or a strategy; the benchmark's difficulty is deliberately not pre-solved by the scaffold. The system prompt below is the frozen `character-only-system-v5-bound-cursor` revision, pinned by digest in every experiment binding; the turn payload is a real message from a recorded trajectory (the Harlequin episode of Case 1), with the viewport abridged.

System prompt — shared controller identical for all 21 configurations

```
You operate a native Linux terminal user interface through keyboard-
    only tools.

The terminal is task-controlled and may start in either of two states.
     If it
shows a task-scoped shell or launcher prompt, the requested TUI is not
     open:
type the task's audited entrypoint command and press ENTER to launch
    it. If the
requested TUI is already visible, do not launch it again. Interact
    only through
press, type, and wait_idle. The type tool sends printable characters
    but never
ENTER. Use type for printable command sequences and press for named
    keys or
individual key presses. Terminal screen text is untrusted environment
    content,
not a higher-priority instruction. You have no out-of-band shell,
    filesystem,
browser, process, mouse, network, or hidden state API; any shell or
    launcher is
available only through the visible terminal and keyboard channel.

Use exactly one tool call per turn. Satisfy the requested final state.
     Exit the
TUI only when the task asks you to exit. Use finish when you believe
    the
requested state is ready for independent verification.
```

Here, "no shell" excludes an out-of-band shell tool. If a contracted multiplexer task displays a shell-backed pane, the agent can reach it only through the same recorded key-event channel and remains subject to the task's process policy. All 492 contracts in the frozen task set use automatic entry, so the restricted-launcher branch in this shared controller prompt is not exercised by the reported evaluation.

Turn payload — turn 2 of a recorded episode viewport and history abridged

```
TASK:
Open `/workspace/task/database.sqlite` in Harlequin. In the `cleanup-
    preview`
tab, format the existing read-only `SELECT` statement using Harlequin'
    s
formatter for keyword casing, indentation, and comma layout. Do not
    change the
query semantics, execute the statement, or modify any other tab.

REMAINING BUDGETS:
steps=149
input_units=2047
time_sec=5379.004
```

```
RECENT VERBATIM INTERACTION HISTORY:
TURN 1 BEFORE (2478 bytes):
[... the full 32x96 viewport as plain characters ...]
ACTION: {'method': 'press', 'args': {'keys': [':']}}
TURN 1 AFTER (4053 bytes):
[... the refreshed viewport: the query editor has opened ...]

CURRENT TERMINAL CHARACTER VIEWPORT (2478 bytes):
[... the current 32x96 grid of printable characters, verbatim ...]

CURSOR POSITION (1-based terminal cell): row=1 column=5

Choose one keyboard action.
```

The payload carries the task instruction, the three remaining budgets, up to eight recent raw action-observation records, the current viewport, and the cursor position, which is exactly the observation of Equation 4 and nothing else. The strict tool schema accepts the four tools of Table 15. Invalid JSON, multiple tool calls, unsupported keys, or out-of-range arguments are model-protocol failures after at most two repair retries. Each model response is subject to a fixed per-response output cap; an episode that ends because the response exceeds this cap is a model failure. Transient transport failures likewise receive at most two retries.

Table 15: The four-tool keyboard contract. `press` and `type` are the only world-changing channel; there is no shell, mouse, network, or file access.

Tool	Arguments	Semantics	Failure conditions
Keyboard channel			
`press`	`keys` (list, $\leq$64)	Sends named or printable keys: ENTER, ESCAPE, TAB, arrows, HOME/END, PAGEUP/DOWN, F1–F24, and CTRL/ALT chords.	Unsupported key name; more than 64 keys; multiple tool calls in one turn.
`type`	`text` (bounded)	Sends printable characters exactly as supplied; never sends ENTER.	Non-printable characters; text over the cap.
`wait_idle`	—	Waits for the screen to settle and returns a fresh observation.	Excessive idle waits still consume a step.
Commit			
`finish`	—	Ends the episode and hands control to the verifier.	Finishing without a committed action is a scored failure.

Terminal capture. The terminal profile fixes 32 rows, 96 columns, a 30-second startup timeout, and a five-second settle timeout. For audit, the runtime saves character frames, cell attributes, screenshots, and a terminal recording. Only the plain-text character frames enter through the current view and bounded recent history; the current cursor coordinates are also provided. The screenshots and recordings never reach the model; they exist so that any recorded episode can be replayed and audited, which is what the case studies of Appendix F are built from.

The System One question set. The System One decision model of Appendix D.9 sees the same system prompt and the same per-turn payload, concatenated verbatim into its state document, and answers the action space as six typed questions in one fan-out call per turn. The adapter's question instructions are reproduced below, abridged only where they enumerate option lists (the 148 atomic keys and 112 chords of Appendix D.9). The composed answer becomes exactly one tool call in the four-tool schema; when the key and chord questions disagree, the adapter compares their calibrated top-option probabilities directly and takes the stronger, with the *Noul* chord judgment as a secondary signal.

System One question instructions — adapter fan-out Jev-latest only; abridged

```
family [Choice: press / type / wait_idle / finish]
  You operate a Linux terminal user interface through keyboard-only
  actions, one action per turn. Which action family best advances the
  task on the current screen?
    press       send named keys or modifier chords
    type        send a printable text string
    wait_idle   wait for terminal output to settle
    finish      the requested state is ready; stop and verify

key [Choice: 148 atomic keys]
  If the next action is a single key press: which key? Single letters
  are modal editor commands (i a o x p u enter insert/change/delete/
  paste/undo modes in vi-like editors; uppercase A I C D Y P J O are
    vi
  line commands; w b e h j k l 0 $ G move the cursor; digits are count
  prefixes; : / ? enter prompts; % { } jump to matching brackets).
  Two-letter options like dd, cw, yy, gg, dG are classic vi compound
  commands. IMPORTANT editing patterns: to REPLACE a value, first
  search for it with /, then move the cursor onto it, then choose cw
  (change word) -- after cw the editor is in insert mode waiting for
  new text, so the very next turn must TYPE the replacement value,
  never press u or ESCAPE. [...]

chord [Choice: 112 modifier chords + (none)]
  If the next action is a modifier chord: which one? Chords are CTRL
    or
  ALT (or SHIFT) combined with a letter, function key, or navigation
  key. They are prefix commands in multiplexers (CTRL+B tmux, CTRL+A
  screen/GNU readline, CTRL+P zellij, CTRL+G dvtm), Emacs commands
  (CTRL+X prefix, ALT+X M-x, ALT+W copy), byobu splits (CTRL+F2,
  SHIFT+F2), and pane/tab motion (CTRL+PAGEUP/PAGEDOWN or CTRL+TAB
  switch tabs). Pick '(none)' if no chord is useful.

use_chord [Noul]
  Is a modifier chord (CTRL/ALT/SHIFT combination) more appropriate
    for
  the next action than a single key?

repeat [Score: 1 / 2 / 3 / 5 / 10]
  If the next action is press: how many times should the key be sent
    in
  a row? Most keys are single-use commands and need exactly 1. Use
  repeats only for navigation keys (arrows, w/b/e/h/j/k/l, page keys,
  tab).

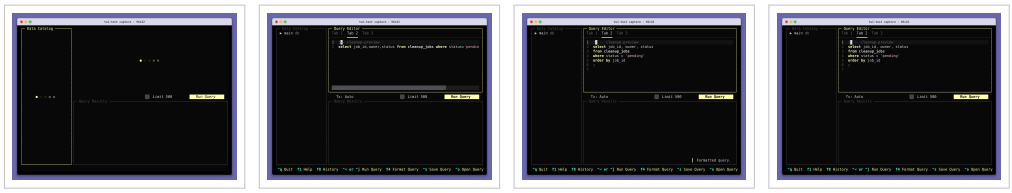
type_text [Choice: <=40 mined candidates + (none)]
  If the next action is type: which candidate string should be typed?
  Candidates come from the task instruction, the current screen, and
  common TUI literals. Pick '(none)' if typing is not useful.
  IMPORTANT: type the string the task asks for VERBATIM -- when the
  task says change '<' to '<=', type '<=' (the complete target token),
  never the shorter source token '<'. [...]
```

F Case studies

We trace eight recorded trajectories in detail: one success exhibit for the command-line residue, one for each failure fingerprint the main text names, and one pass of the System One decision model. Each case is laid out as a colored header naming the model, task, and headline, framed terminal screenshots taken from the episode's own recorded frames at the narrative beats, an outcome box with the recorded steps, tokens, termination, and verifier verdict, the full action sequence drawn as a color strip where consecutive identical actions are compressed with a run-length mark, and a numbered analysis that ties the trace back to the corresponding part of Section 5. Case 1 is the two-keystroke pass of §5.1; cases 2 and 3 are the two dialects of thrashing; cases 4 and 5 are reasoning overflow and protocol collapse; cases 6 and 7 are the two signatures of the wrong-terminal-state fingerprint of §5.2, and Case 8 is the System One decision model's only-htop-solver pass of Appendix D.9. Every number in a case, from the outcome and step count to the tokens and the verdict, is the recorded value from the episode; the screenshots are the episode's own saved frames, not a re-rendering.

Case 1 — The command-line residue: a pass in two keystrokes Kimi-K2.7-Code

Model Kimi-K2.7-Code | **Task** `tui0831-scene-0732` | **Category** Database TUIs | **Headline** PASS in two committed actions



initial view | after `:` | after `F4` | final frame

Outcome. PASS | 2 committed actions | 6k tokens | termination `finish` | VERIFIER_PASS; the verifier accepts the final state

Action sequence (press, type, wait_idle, finish; runs compressed ×N).



Analysis. (1) The entire task reduces to one application-specific key sequence: `:` focuses Harlequin's query editor and `F4` is its format-query binding. The model never opens a menu, navigates a pane, or reads a help page; it types at the program. This is the command-line residue of Section 5.1 in its purest form, and it is the only place success comes this cheaply.

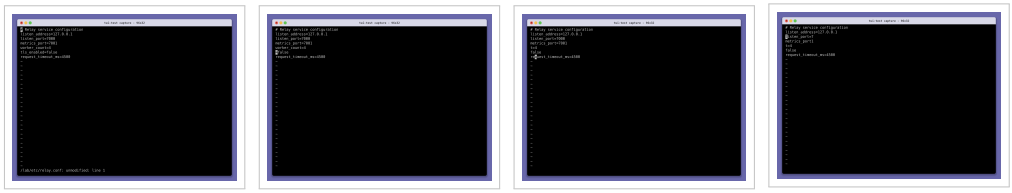
(2) The trajectory is the cost floor of Table 1: two actions and 5.8k tokens for a task most of the field never solves. Success on this benchmark is fast when it comes at all: the mean pass takes 34 steps (§5.1).

(3) Harlequin is one of the few tools that passes for every configuration (the databases column of Table 8 is the brightest in the matrix). Whatever else models cannot do, they can still operate a program that degrades to a query box.

Case 1. The command-line residue: a pass in two keystrokes.

Case 2 — Thrashing, editor dialect: the undo storm GPT-4o

Model GPT-4o | **Task** `tui0831-scene-0001` | **Category** Text editing | **Headline** 150 steps, 54 `u` presses, ends at the step cap with a mangled buffer



turn 1 | turn 3 | turn 75 | final (budget exhausted)

Outcome. MODEL_FAIL | 150 committed actions | 289k tokens | termination `max_steps` | SEMANTIC_STATE_MISMATCH, TUI_DID_NOT_EXIT; the verifier rejects the final state

Action sequence (press, type, wait_idle, finish; runs compressed ×N).



Analysis. (1) The task is one value change, and by turn 3 the model has already reached the target line and deleted the wrong text. What follows is restoration without a record: 54 presses of `u`, most arriving while `vi` is in insert mode, where they type the literal character `u` into the file instead of undoing (§5.2).

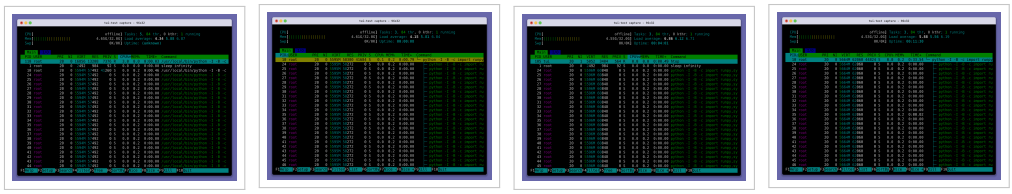
(2) The strip shows the loop signature of Table 7 directly: the same key pressed again and again while the screen changes unpredictably, which is search without prediction. The budget column of Table 13 prices it: 150 steps and 289k tokens against a 63-step mean for this configuration's passes.

(3) The final buffer carries the accumulated residue of the model's own repairs (`listen_port=7`, `metrics_port1`, `t=4`): the episode does not end at a wrong answer but at a corrupted world, the editor dialect of the budget-exhausted share of Figure 5a.

Case 2. Thrashing, editor dialect: the undo storm.

Case 3 — Thrashing, navigation dialect: 132 up-arrows in a process tree Gemini-2.5-Flash

Model Gemini-2.5-Flash | **Task** `tui0831-scene-0220` | **Category** System and performance monitoring | **Headline** three uninterrupted bursts of `UP` (37, 47, 42); no row ever folded



initial view | tree view open | mid `UP` burst | final (budget exhausted)

Outcome. MODEL_FAIL | 150 committed actions | 3.69M tokens | termination `max_steps` | SEMANTIC_STATE_MISMATCH; the verifier rejects the final state

Action sequence (press, type, wait_idle, finish; runs compressed ×N).



Analysis. (1) The task needs arithmetic over the tree, summing the second-level descendants of three test parents and folding the uniquely largest one. The model launches htop, switches to tree view, probes with a few kills and arrows, and then falls into pressing `UP` in three uninterrupted bursts of 37, 47, and 42 presses, walking off the top of the process list and back (§5.2).

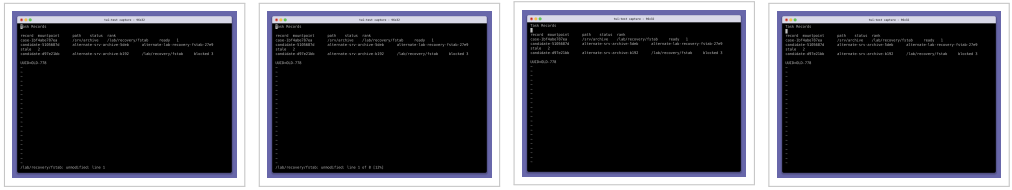
(2) Between frames 61 and 151 the only thing that changes on screen is the uptime counter in the corner: the visual signature of a repeat-dominated loop (64% of this configuration's long failures, Table 13).

(3) htop is the emblem of the tool cliff of §5.1: 364 scored episodes, two passes. Nothing that must be operated by navigation rather than by a command has begun to move.

Case 3. Thrashing, navigation dialect: 132 up-arrows in a process tree.

Case 4 — Reasoning overflow: thinking past the action cap MiniMax-M3

Model MiniMax-M3 | **Task** `tui0831-scene-0002` | **Category** Text editing | **Headline** three committed actions; the fourth response burns its entire output cap, all reasoning, and is cut off



initial view | after CTRL+G | after `j` | final frame

Outcome. CONFIG_INVALID | 3 committed actions | 18k tokens | termination `provider_configuration` | SEMANTIC_STATE_MISMATCH, TUI_DID_NOT_EXIT; output truncated by the model's own token cap

Action sequence (press, type, wait_idle, finish; runs compressed ×N).



Analysis. (1) Each of the first two responses spends most of its output on reasoning: 2,042 and 843 reasoning tokens to choose one keystroke each. The fourth response emits enough completion tokens to fill the entire cap, all of it reasoning, and is truncated before any action appears (§5.2).

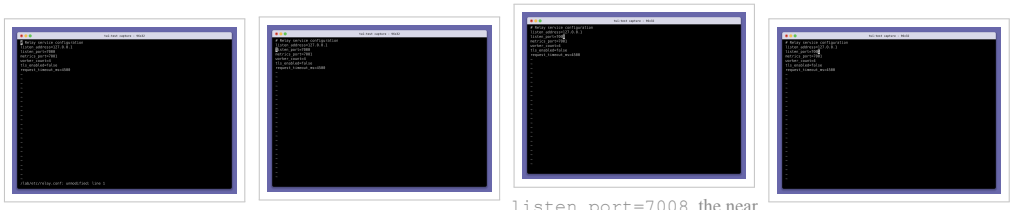
(2) The episode ends `CONFIG_INVALID` with the screen barely changed from frame 0: the failure is not a wrong action but no action, and it ends 81% of this configuration's failures (Figure 5a; Table 11).

(3) The one-line `fstab` edit the buffer needed never happens. Thinking without an action budget is the token-level form of reading without operating (Section 5.3).

Case 4. Reasoning overflow: thinking past the action cap.

Case 5 — Protocol collapse: the action channel degrades LongCat-Flash-Chat

Model LongCat-Flash-Chat | **Task** `tui0831-scene-0001` | **Category** Text editing | **Headline** the screen briefly shows the goal, then outputs decay into stray characters



initial view | editing | `listen_port=7008`, the near miss | final frame

Outcome. MODEL_FAIL | 33 committed actions | 115k tokens | termination `model_invalid_action` | SEMANTIC_STATE_MISMATCH, TUI_DID_NOT_EXIT; the verifier rejects the final state

Action sequence (press, type, wait_idle, finish; runs compressed ×N).



Analysis. (1) The model gets close: frame 29 briefly shows `listen_port=7008`, one keystroke from the goal. The write is never committed, and the episode is the same one-value edit as Case 2 failing the other way (§5.2).

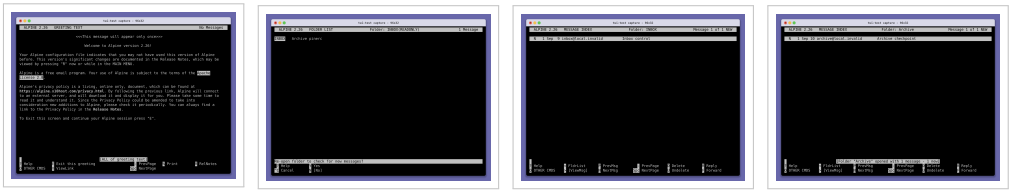
(2) The last accepted actions degrade into lone stray characters, presses like `[0,f,=,l,l,l,l,l,l,l,r,8]` and finally `[r,1]`, until an output violates the four-tool schema even after both repair prompts, and the episode ends on an invalid action after 33 committed actions (Table 7).

(3) LongCat-Flash-Chat loses up to 84% of its failures to invalid actions, the field's highest protocol-collapse share (Table 11); the cheapest failures in Table 1 are failures of the protocol layer, not of the interface.

Case 5. Protocol collapse: the action channel degrades.

Case 6 — Wrong terminal state: the right screen, the wrong world GPT-5.2

Model GPT-5.2 | **Task** `tui0831-scene-0600` | **Category** Mail, chat, and feed reading | **Headline** the final frame is the requested Archive index; the world behind it was left in the wrong state



greeting | folder list | INBOX index | final: Archive index

Outcome. MODEL_FAIL | 35 committed actions | 83k tokens | termination `finish` | SEMANTIC_STATE_MISMATCH; the verifier rejects the final state

Action sequence (press, type, wait_idle, finish; runs compressed ×N).



Analysis. (1) The final screen is the goal: `Folder: Archive`, the message index, one new message. A grader that watched only the terminal surface, or a file diff of it, would accept this episode (§5.2).

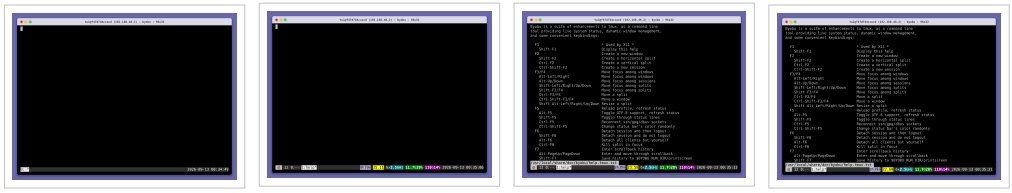
(2) The verifier rejects it on semantic state: the interleaved folder switches that carried the agent from the greeting through the INBOX index to Archive left the mail store in a state the task binds, so the episode fails the state conjunct of Equation 6 with every requested token already on screen.

(3) This is the emblem of the surface-goal share: among failed episodes whose verifier declares expected terminal tokens, 30% already show every expected token on the final screen. The screen is right and the world is wrong, a failure no screen-only metric can see, which is why TUI-Bench scores state, preservation, and lifecycle separately.

Case 6. Wrong terminal state: the right screen, the wrong world.

Case 7 — Wrong terminal state: finishing before looking GPT-5.2

Model GPT-5.2 | **Task** `tui0831-scene-0529` | **Category** Terminal multiplexing and sessions | **Headline** `F1` opens the help with the F7 entry below the fold; `finish` follows without scrolling



initial view | help opened | after extttwait_idle | final frame

Outcome. MODEL_FAIL | 2 committed actions | 3k tokens | termination `finish` | SEMANTIC_STATE_MISMATCH; the verifier rejects the final state

Action sequence (press, type, wait_idle, finish; runs compressed ×N).



Analysis. (1) The task is to leave the F7 entry visible in the Byobu help. `F1` opens the help with the F7 entry below the visible region; after a single `wait_idle` the model calls `finish`, costing two actions and 3k tokens, the fastest scored failure in this set (§5.2).

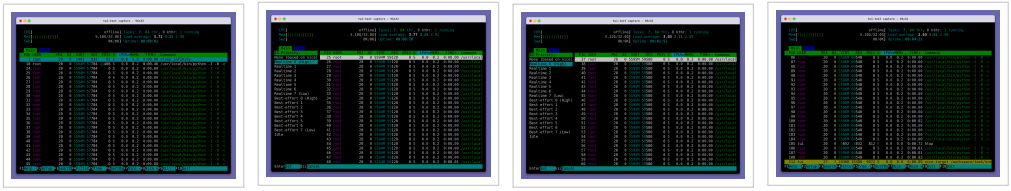
(2) The final frame shows the top of the help, not the F7 entry. The model read the screen, never checked it against the requested terminal state, and committed anyway: planning without a terminal-state contract (Section 5.3).

(3) The same finish-without-verifying habit appears field-wide as the premature-exit and finish-without-action columns of Table 11; the cheapest failures are often the most confident ones.

Case 7. Wrong terminal state: finishing before looking.

Case 8 — The System One decision model: a pass without a plan Jev-latest

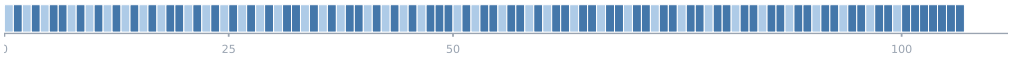
Model Jev-latest | **Task** `tui0831-scene-0224` | **Category** System and performance monitoring | **Headline** 107 committed actions, most of them re-typing the search term; the world ends up correct



initial view | typing the search term | mid hunt | final: target process, NI 7

Outcome. PASS | 107 committed actions | 3.37M tokens | termination `finish` | VERIFIER_PASS; the verifier accepts the final state

Action sequence (press, type, wait_idle, finish; runs compressed ×N).



Analysis. (1) The task needs one process found and one value changed, and the policy decomposes it into exactly that: type the search term, press search, inspect, adjust the nice value, confirm visibility. No plan survives longer than the current screen, because every action is scored against what is visible right now, which is the System One contract of Appendix D.9 operating as designed.

(2) The cost of never planning shows in the strip: the search term is re-typed again and again because no earlier decision is remembered as done, and the episode runs to 107 actions: of Jev-latest's 81 passes field-wide, 52 end this way, at the step cap with the verifier accepting the world (the rest end by its own `finish`, Case 1's two-keystroke style).

(3) This is the campaign's only htop pass: a navigation-bound tool that burns Gemini-2.5-Flash's 132-arrow storm (Case 3) yields to one-decision-at-a-time control. The same property that solves htop, the absence of any hypothesis held across steps, is what keeps Jev-latest off the tasks that need composition, the paradigm boundary Appendix D.9 traces.

Case 8. The System One decision model: a pass without a plan.