A Multi-Agent Framework for High-Interaction Terminal Simulation

Kai Wei[1], Yuwen Cui[1], Kehan Shen[1], Hua Wei[2], Guangjing Wang[1]
[1]University of South Florida [2]Arizona State University

Abstract

Terminal simulation, framed as a terminal command-level Turing test, is a long-standing symbolic language generation problem in dialogue and interactive systems. Prior scripted simulators lack flexibility for complex, multi-turn interactions, while LLM-based approaches often misinterpret commands, break output formats, drift from system state, and remain vulnerable to prompt injection. In this work, we propose MANTIS, a terminal simulation framework that improves realism, consistency, and robustness for command language generation. MANTIS integrates a multi-agent architecture with a filter-based routing model that safely dispatches commands to external tools or an LLM-based agent to support interactive commands and defend against prompt injection attacks. In addition, we design an agentic file system with history memory pruning for long-term state consistency. We release three datasets: 28,045 real terminal input–output pairs, a 1,000 multi-turn interaction session dataset, and a 25,849 labeled classification dataset. MANTIS outperforms state-of-the-art baselines by more than 9%, achieving over 95% accuracy on multi-turn terminal simulation. The dataset and source code are available at https://github.com/kaiwei666a/MANTIS_Terminal_Simulation.

1 Introduction

Terminal simulation is fundamentally a symbolic command-language generation problem for dialogue and interactive systems. As a primary command-line interface, the Unix-based terminal plays an important role in human-computer interaction, such as remote access and system administration. In particular, terminal simulation has been widely used as a key mechanism for cyber deception to lure malicious attackers and collect threat intelligence (Fan et al., 2024; Christli et al., 2024; Otal and Canbaz, 2024).

Terminal simulation focuses on modeling interactive command-line environments. A high-interaction terminal simulator must preserve not only command syntax, but also execution semantics, environmental continuity, and state consistency across multi-turn command interactions. The terminal simulation is challenging because both the output content and format must adapt accurately to each input in a continuously interactive state. Meanwhile, we need to ensure cross-session consistency (i.e., stateful coherence across sessions) in long-term, multi-turn command-line interactions. The challenges become especially demanding when users are computer-proficient operators who actively probe the system, test corner cases, and notice subtle inconsistencies in outputs, file states, permissions, or command behavior.

Traditional terminal simulations often rely on static scripted logic, which is brittle and inflexible under complex multi-turn interactions (Kumar et al., 2024). More recently, large language models (LLMs) have been explored for terminal simulation (Sladić et al., 2024; Sezgin and Boyacı, 2025; Zhou et al., 2025). Yet, general-purpose LLMs struggle in high-interaction terminal simulation. LLMs may misinterpret shell commands, produce outputs in incorrect terminal formats, hallucinate content for chained or state-dependent commands, and generate responses that are inconsistent with interaction history (Shi et al., 2023; Bridges et al., 2025). LLMs also remain vulnerable to instruction-following failures and prompt injection attacks (He and Vechev, 2023; Wei and Wang, 2025), which is particularly problematic in adversarial, high-interaction environments.

In this work, we address the above limitations by designing **MANTIS** (**M**ulti-**A**ge**N**t **T**erminal **I**nteraction **S**imulation). We introduce a multi-agent framework to enable parallel task execution, as a high-interaction terminal simulation requires distinct, tightly constrained subproblems that inter-

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 32828–32845

act over long horizons. An *Arbiter Agent* coordinates and dispatches tasks to both a *Strategic Agent* and a *Response Agent* to improve efficiency and context consistency. In addition, we build a large-scale terminal interaction dataset to fine-tune small language models, ensuring robust interpretation of chained and complex commands with limited computational resources. Furthermore, we design an agentic file system that preserves consistency with the interaction history while reducing token consumption. Finally, we propose a dynamic routing tool that defends against prompt-injection attacks and handles dynamic and interactive commands such as *scp* and *nano*.

We evaluate MANTIS through statistical tests along multiple dimensions, including syntax accuracy, state consistency, and robustness to prompt injection attack. MANTIS consistently outperforms state-of-the-art baselines across all metrics. For example, on a terminal interaction dataset of 1,000 multi-turn terminal sessions, MANTIS achieves more than 95% accuracy, exceeding HoneyGPT (Wang et al., 2025) by 9%. The history pruning algorithm also saved 18% of token consumption while maintaining 96% accuracy. Finally, in a user study of 20 participants, MANTIS received an average mean opinion score of at least 4.6 out of 5 for terminal simulation. In summary, we make the following contributions:

- **Dataset.** We construct a shell command dataset comprising 28,045 input-output pairs from a real Linux system. We also curate a multi-turn terminal interaction dataset with 1,000 sessions to evaluate the accuracy and state consistency of LLM-based terminal interactions. Furthermore, we labeled a classification dataset containing 25,849 different instructions for command routing modeling.

- **Agentic Design.** We propose a multi-agent framework that supports parallel task execution and command-line simulation, enabling accurate shell-command interpretation and realistic terminal outputs in interactive honeypot environments. Specifically, we introduce an agentic file-system abstraction to make MANTIS maintain context-consistent responses. In addition, we design a history pruning algorithm that reduces token consumption and response latency, improving accuracy.

- **Terminal Simulation.** We implement a real-world testbed on top of the proposed terminal simulation framework. In particular, we introduce a routing tool for a response agent that classifies incoming commands and dispatches them either to dedicated execution tools or to the LLM-based simulator. This design improves security against prompt-injection attacks while efficiently handling both simple commands and highly dynamic, interactive terminal commands with high fidelity.

2 Related Work

High-interaction terminal simulation can be viewed as a command-level Turing test. Compared to judging free-form natural language fluency and safety (Chen et al., 2023, 2024b), we evaluate whether the simulated terminal behaves like a real computer system under sustained adversarial interaction. Terminal simulation is particularly important in cyber deception, where a simulated terminal, as a honeypot (Guan et al., 2024; Wang et al., 2025), can accept and process an attacker's commands in a virtual environment rather than on real systems, thereby reducing security risks.

The low- and medium-interaction terminal simulators generate command responses using manually scripted input–output pairs, which are limited to predefined command sets and easily exposed by exploratory probing (Shi et al., 2019; Oosterhof et al., 2025). To improve realism, recent work adopts LLMs to generate more realistic and context-aware terminal responses (Guan et al., 2024; Christli et al., 2024; Sladić et al., 2024, 2025). However, LLM-based terminal simulation struggles to maintain a dynamic and interactive system state across multi-turn interactions.

For example, HoneyLLM (Guan et al., 2024) is an LLM-based terminal simulator, but it often mishandles malformed inputs and violates constraints related to filesystem state and permission checks. This is because the general-purpose LLMs are trained on public web corpora with limited exposure to shell syntax and operating system behaviors (Deng et al., 2025; Fan et al., 2024; Otal and Canbaz, 2024), making it difficult to consistently simulate state-dependent command outputs. In addition, LLMs are prone to hallucinations and to inconsistent use of prior context (Liu et al., 2023), which makes reliable state tracking difficult and thus hampers execution of chained or dynamically evolving shell commands.

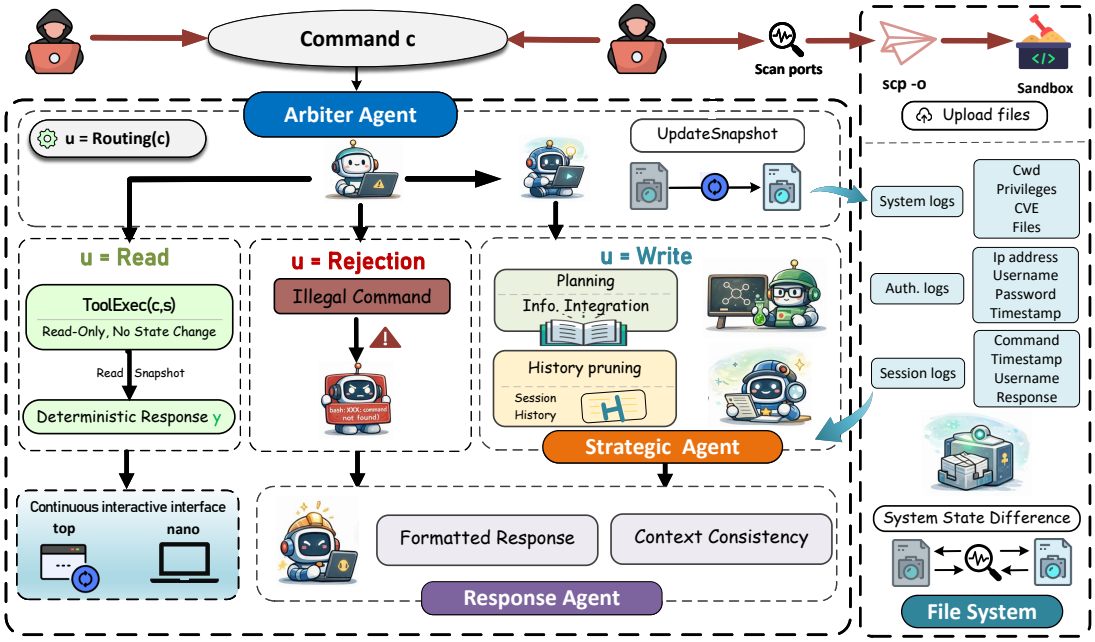

Figure 1: The MANTIS system comprises multiple cooperating agents: an Arbiter Agent, a Strategic Agent, and a Response Agent, together with an agentic file system that includes a history-pruning mechanism. MANTIS supports routing incoming commands, maintaining and reasoning over system state, and generating realistic terminal outputs.

3 System Design

In this section, we introduce MANTIS, including multi-agent design with a dynamic routing tool, an agentic file system with history pruning for a realistic and interactive terminal simulation.

3.1 Multi-Agent Design

LLM-based multi-agent systems extend the capabilities of pure LLM via external tool integration, structured memory, and customized workflow planning and multi-agent coordination. Prior work (Jia et al., 2025; Chen et al., 2024a; Wang et al., 2023) shows that interactive environments can surface multi-agent coordination failures and consistency issues that are often obscured in single-agent or single-turn evaluations. A new interactive environment is needed to assess output consistency, dynamic task planning, parallel execution, and communication-efficient coordination for multi-agent evaluation. Terminal simulation provides such a practical and interactive environment for assessing LLM-based multi-agent behavior.

As shown in Figure 1, the terminal simulation is organized as a coordinated multi-agent system composed of an *Arbiter Agent*, a *Strategic Agent*, and a *Response Agent*. The *Arbiter Agent* acts as the control plane of the terminal simulation. For every incoming command, the *Arbiter Agent* records the pre-execution system snapshot and incorporates the observable execution effects into the maintained snapshot logs. The *Arbiter Agent* maintains system, session, and authentication logs, and routes how each incoming command should proceed.

We develop a command routing model as a tool to help the *Arbiter Agent* decide how each command proceeds and call other tools. Given an input command, the routing model classifies the command into one of three routing categories. (i) Read category corresponds to read-only operations, such as directory listing or status queries. These commands do not induce persistent state changes and are handled by fast, deterministic handlers that directly read from the maintained system state. (ii) Write category includes state-changing or context-sensitive operations that affect the filesystem, process management, and I/O operations. These commands are forwarded to the *Strategic Agent* for history context pruning and providing context for *Response Agent*. (iii) Rejection category includes invalid, malformed, or unsafe inputs, such as prompt-injection attempts. These commands are immediately rejected and return standard shell-style error messages consistent with real terminal behavior.

The *Strategic Agent* manages contextual information required for terminal simulation, including pruned history and system state. The history pruning retains only interactions that are important to explain the current environment configuration, pri-

oritizing state-altering commands. The resulting context provides a minimal yet causally consistent view of the system state for response generation. To lure external adversaries, the *Strategic Agent* can optionally generate a plan log that specifies how the system should respond to the incoming command to simulate a potential vulnerability, as described in detail in Appendix E. This plan log is provided to the LLM as guidance for response generation.

The *Response Agent* generates the final terminal output based on the current system state. In our multi-agent design, a limited task scope by task delegation allows the *Response Agent* to focus on output-format and state-consistency requirements, enabling more effective task execution with fewer task instructions. The multi-agent terminal interaction is shown in Algorithm 1. Given a command c_t, the current system snapshot s_{t-1}, and the retained interaction history W_{t-1}, the algorithm first computes a routing decision $u_t = \text{Routing}(c_t)$, which determines the execution path. Invalid or adversarial inputs are rejected with shell-consistent error responses (lines 3-4 in Algorithm 1) while non-state-changing commands are handled via a tool execution path $y_t = \text{ToolExec}(c_t, s_{t-1})$ (lines 5-7).

For write-level commands ($u_t = \text{Write}$), if MANTIS aims to simulate a vulnerability as a honeypot, MANTIS performs planning based on the current snapshot and the pruned history, producing a response plan $P_t^* = \text{Plan}(s_{t-1}, c_t, W_{t-1})$ (lines 8-10). In parallel, the system updates the snapshot according to command semantics using $\text{UpdateSnapshot}(s_{t-1}, c_t)$, and prunes the interaction history by $\text{PruneHistory}(s_{t-1}, c_t, W_{t-1})$. Finally, conditioned on the optional plan P_t^*, the updated snapshot s_t, and the pruned history W_{t-1}, the *Response Agent* generates the terminal output $y_t = \text{Response}(s_t, c_t, W_{t-1}, P_t^*)$ and records an execution trace $\pi_t = (u_t, H_t, P_t^*, y_t, W_t)$ for next round analysis.

Algorithm 1: Agentic Terminal Simulation

Input: command c, snapshot s, timestamp t, history H, honeypot flag F, pruned history W
Output: response y and trace π

1 $id \leftarrow \text{uuid}()$;
2 $u \leftarrow \text{Routing}(c)$;
3 **if** $u = \text{Rejection}$ **then**
4 **return** ("ShellError", Trace(id, u_t));
5 **if** $u = \text{Read}$ **then**
6 $y \leftarrow \text{ToolExec}(c, s_{t-1})$;
7 **return** $(y_t, \text{Trace}(id, u_t, y_t))$;
8 Run in parallel:
9 **if** F == *True* **then**
10 $P^\star \leftarrow \text{Planning}(s_{t-1}, c, W_{t-1})$;
11 $s_t \leftarrow \text{UpdateSnapshot}(s_{t-1}, c_t)$;
12 $W_t,\ H_t \leftarrow \text{PruneHistory}(s_{t-1}, c, W_{t-1})$;
13 $y \leftarrow \text{Response}(s_t, c_t, W_{t-1}, P^\star)$;
14 **return** $(y, \text{Trace}(id, u_t, H_t, P_t^*, y_t, W_t))$;

3.2 Agentic File System Design

As LLMs are increasingly embedded into agentic workflows, performance and reliability are constrained more by context engineering. The context engineering includes the end-to-end process of acquiring, organizing, compressing, and updating the information an agent uses to reason and act. Although recent LLMs support large context windows (e.g., 128K), LLMs exhibit strong positional bias in long-context settings, with degraded memory and reasoning performance for context in earlier positions (Mohsin et al., 2025; Liu et al., 2025).

To provide more useful information within an effective window, existing history pruning methods primarily rely on truncation, summarization, or relevance-based retrieval (Anagnostidis et al., 2023; Fu et al., 2024). While effective at reducing token consumption, existing methods undermine the precision with which LLMs reason about the current fine-grained system state. In interactive terminal environments, even minor deviations in executable details (e.g., file paths, permissions, flags, or working directories) can alter command semantics and lead to inconsistent or invalid system states (Holister et al., 2025).

3.2.1 Role-based File System

Inspired by the Unix philosophy that "everything is a file", we design an agentic file system that unifies context artifacts under a common namespace and maintains multiple task-specific logging channels to record terminal commands and metadata (e.g., timestamp, user). Specifically, (i) System log maintains a structured file of directories, files, metadata, and permission changes that result from external input commands, such as file creation, deletion, modification, and movement. (ii) Authentication log records SSH authentication activities, including login attempts, credential usage, and session establishment events. (iii) Session log stores

the chronological interaction history between the attacker and the terminal simulator, including issued commands and corresponding responses. In addition, we introduce a sandbox for isolated file-handling. All attacker-uploaded or modified files are redirected into a dedicated Docker container that provides filesystem isolation and disables code execution and external networking. The sandbox is used solely to record file contents and structural changes, preventing malicious payloads from being executed on the host system.

3.2.2 History Pruning for In-Context Memory

The complete interaction history H_t up to time t is stored in different logs for inspection, such as for calling the shell command $history$. Besides, *Arbiter Agent* retains the current system state and state-changing commands in the pruned history $\mathcal{W}_t$. $\mathcal{W}_t$ is a subsequence of H_t that acts as in-context memory for *Response Agent* to act correctly at each step. The history-pruning algorithm for efficient in-context memory is shown in Algorithm 2.

Specifically, the system observes a new terminal interaction $e_t = (c_t, y_t)$ at time step t, where c_t denotes the command and y_t denotes the response. Upon observation, the e_t is appended to H_{t-1}, denoted as $H_t = H_{t-1} \parallel e_t$. The complete execution trace is retained independently of any context pruning decisions. We use JSON format to track system entities (e.g., files, processes, services, network endpoints) and their observable properties (e.g., existence, permissions, ownership, configuration values). The *Arbiter Agent* updates the system entities from the pre-execution state s_{t-1} to the post-execution state s_t, reflecting the observable system snapshot after executing c_t.

We define a pruning priority $U(t)$ based on $[e_t, \mathcal{W}_{t-1}, s_t, s_{t-1}]$, and $U(t)$ returns a three-component priority vector as $U(t) = \big(U_{\text{dup}}(t)^{***}, U_{\text{std}}(t)^{**}, U_{\text{stale}}(t)^{*}\big)$, where $U_{\text{dup}}(t)$ has highest priority, $U_{\text{std}}(t)$ is the second, and $U_{\text{stale}}(t)$ is last to consider when removing a history record. Specifically, the initial value of $U_{\text{dup}}(t)$ for each command is 0. When there exists an interaction $e_h \in \mathcal{W}_{t-1}$ whose stored command is identical to c_t, the redundancy term $U_{\text{dup}}(e_t)+1$ (lines 5-6 in Algorithm 2). The repeated commands accumulate larger $U_{\text{dup}}(\cdot)$ and are more likely to be evicted.

We then compute the corresponding state difference grade as $U_{\text{std}}(t) \triangleq \text{StDiff}(s_{t-1}, s_t)$, which assigns a discrete severity level to the snapshot transition from s_{t-1} to s_t (line 7 in Algorithm 2). In-

Algorithm 2: State-Aware History Pruning

Input: new interaction $e_t = (c_t, y_t)$; snapshots s_{t-1}, s_t; previous pruned history $\mathcal{W}_{t-1}$; persistent history H_{t-1}; capacity K

Output: pruned history $\mathcal{W}_t$, updated history H_t

1 $H_t \leftarrow H_{t-1} \parallel e_t$;
2 $U(t) \leftarrow \text{Evict}(e_t, \mathcal{W}_{t-1}, s_t, s_{t-1})$;
3 $U(t) = \big(U_{\text{dup}}(t),\, U_{\text{std}}(t),\, U_{\text{stale}}(t)\big)$;
4 **foreach** $h_n \in \mathcal{W}_{t-1}$ **do**
5 **if** $c_n = c_t$ **then**
6 $U_{\text{dup}}(n) \leftarrow U_{\text{dup}}(n) + 1$;
7 $U_{\text{std}}(t) \triangleq \text{StDiff}(s_{t-1}, s_t) \in \{0, 1, 2, 3\}$;
8 $U_{\text{stale}}(e_t) = t$;
9 $h_t \leftarrow \big(e_t,\, \text{timestamp}_t,\, U(t)\big)$;
10 $\mathcal{W}_t \leftarrow \mathcal{W}_{t-1} \parallel h_t$;
11 **while** $|\mathcal{W}_t| > K$ **do**
12 $h^\star \leftarrow \arg\max_{h \in \mathcal{W}_t} U(h)$;
13 $\mathcal{W}_t \leftarrow \mathcal{W}_t \setminus \{h^\star\}$;
14 **return** $\mathcal{W}_t, H_t$;

spired by the Linux Audit (Red Hat, 2025), MITRE ATT&CK (MITRE, 2025), and advanced persistent threat (APT) detection work (Microsoft, 2025; King and Chen, 2003), we define grade rules to quantify the state difference $U_{\text{std}}(t)$. (i) If any privilege or identity change, persistence modification, or network exposure change is present, $U_{\text{std}}(t) = 0$. (ii) If any critical configuration file is modified, then $U_{\text{std}}(t) = 1$. (iii) If any local filesystem write/creation/deletion/content modification occurs, $U_{\text{std}}(t) = 2$. (iv) $U_{\text{std}}(t) = 3$ if no persistent side effect is observed (read-only). When multiple change types co-occur, $U_{\text{std}}(t)$ is conservatively determined by the minimum applicable grade level. Thus, given a command, for each related field (e.g., a specific file or folder), $U_{\text{std}}(t) \in \{0, 1, 2, 3\}$, obtained by a field-wise comparison between fields of s_{t-1} and s_t.

The staleness term $U_{\text{stale}}(e_t)$ measures how old an interaction is. We define it as the elapsed rounds since the interaction occurred. The initial value of $U_{\text{stale}}(e_t) = t$ is 0, and is increased by 1 if a new terminal interaction occurs. Thus, earlier interactions have larger staleness values. During pruning, system commands are compared first by redundancy $U_{\text{dup}}(t)$, then by grade-based priority $U_{\text{std}}(t)$, and finally by temporal staleness $U_{\text{stale}}(t)$.

Table 1: Commercial LLM API Performance Comparison for Terminal Interaction Simulation on Multi-turn Dataset.

Metric	Claude-4	DeepSeek-R1-0528	Gemini-2.5-Pro	GPT-OSS-120B	Qwen3-Coder
Accuracy (%)	85.3	82.7	87.6	83.9	89.3
Avg-Latency (s)	3.65	3.41	2.57	1.42	1.08
P50 Latency (s)	2.50	3.28	1.81	1.16	0.71
P95 Latency (s)	12.34	5.01	5.81	2.76	2.98
Total Tokens	4,979,724	6,137,005	5,010,700	4,437,379	4,255,288

Metrics: Accuracy (%) is the percentage of sessions with correct responses based on system settings and history states; Avg-Latency (s) is the average end-to-end latency per command. P50 Latency (s) is the median end-to-end per-command latency. P95 Latency (s) is the 95th-percentile latency reflecting tail delays. Total Tokens is the cumulative token usage across all requests.

We then construct a new history entry $h_t = \big(e_t, \text{timestamp}_t, U(t)\big)$ and append it to pruned history $\mathcal{W}_t$. We define K as an upper bound on the number of interactions retained in the working context and directly reflects the effective context budget available to the language model. K is calculated as: $K = \left\lfloor \frac{B_{\max} - B_{\text{fixed}}}{b} \right\rfloor$, where $B_{\max} = 8,000$, B_{fixed} denotes non-history tokens, and b is the estimated per-interaction token cost.

Based on NoLiMa (Modarressi et al., 2025), GPT-4o's performance degrades noticeably once the input context grows beyond 8K, known as memory decay. Thus, we account for the memory decay and keep the total prompt length fed to the *Response Agent* below $B_{max} = 8,000$ tokens when choosing the context capacity K. In our implementation, B_{fixed} is measured from the actual prompt construction, which consists of the system prompt (235 tokens) and a snapshot field in the *Strategic Agent* prompt. As the snapshot is initially empty but can grow after commands are executed, we use a conservative upper bound by empirically approximating the steady-state snapshot footprint as two snapshots (1335 tokens per snapshot).

We empirically set $b = 100$ as an estimate of the average token footprint per interaction, including command, response, and metadata. With the above settings, we calculate K as 50 interactions in the working context. When $|\mathcal{W}_{t-1} \parallel e_t| > K$, we iteratively evict the interaction with the three-component priority vector $U(\cdot)$ until $|\mathcal{W}_t| \leq K$. We acknowledge that advances in LLM capabilities may increase the feasible value of $B_{\max}$. Our framework can readily accommodate such changes by adjusting these parameters accordingly.

4 Implementation and Evaluation

4.1 Dataset and Instruction Tuning

Datasets. We constructed three datasets in our experiments. (i) *Response Dataset*: A response-generation dataset of 28,045 Linux shell commands collected from a Kali Linux environment, where each instance pairs a raw command c with its ground-truth terminal output y obtained by executing the command in a controlled Linux environment. This dataset is used for instruction tuning on LLMs in *Response Agent*.

(ii) *Multi-turn Dataset*: We crafted 1,000 multi-turn interaction sessions (3–20 turns) that simulate diverse behaviors. The dataset is used for LLMs in multi-turn response benchmarking, with a fixed workspace directory structure. More details are in Appendix B. Additionally, we constructed 20 multi-turn interaction sessions, each comprising 50–100 turns, resulting in a total of 1,403 commands. We use these sessions to evaluate the performance of history pruning with respect to token consumption and contextual state consistency.

(iii) *Routing Dataset*: We merge commands from the *Response Dataset* and *Multi-turn Dataset*, followed by deduplication and labeling as *Read or Write*. Besides, we crafted 2,361 syntax-error commands (e.g., sl, $ls\text{-}al$), and prompt-injection inputs, labeling them as *Rejection*. In total, we have 25,849 instances in the routing dataset. 11.71% are *Write* operations and 9.13% are *Rejection* cases, while the remaining 79.14% are *Read* operations, consistent with the read-dominant nature of real-world terminal interactions. More detail in Appendix F.

Models and Training. We fine-tuned multiple open-source small language models, including Llama, Qwen, DeepSeek, CodeLlama, and CodeGemma variants, and adopted Llama-3.2-3B-Instruct (FT) as the "brain" of our *Response Agent*. Specifically, fine-tuning is performed on a single NVIDIA RTX 4090 GPU using Low-Rank Adaptation (LoRA) supervised fine-tuning, implemented in the LLaMA-Factory framework. Models are trained for three epochs using the AdamW optimizer with a learning rate of 5×10^{-5}, gradient accumulation steps of 8, and a maximum gradi-

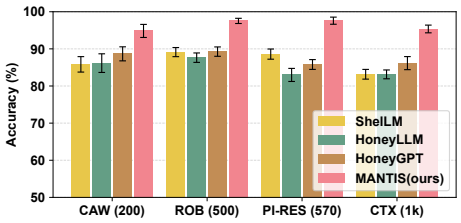

Figure 2: System comparison across four metrics. CAW = command execution accuracy on normal commands; ROB = robustness to syntax error commands; PI-RES = resistance to prompt-injection inputs; CTX = multi-turn context consistency (Multi-turn Dataset).

ent norm of 1.0. A cosine learning-rate scheduler is applied with 7 warm-up steps. The per-device batch size is set to 2 for both training and evaluation. LoRA adapters are applied to all target modules with a rank of 8 and a scaling factor of 16. In addition, we fine-tune a dynamic routing model for command-level classification as described in Section 3.1. More details are in Appendix F.

4.2 Terminal Response LLM Benchmarking

We first benchmark the performance of commercial APIs using the Vertex AI service (Google Cloud, 2025b) on our multi-turn dataset for terminal simulation. The evaluated models include **Claude-4** (Anthropic, 2025), **DeepSeek-R1-0528** (DeepSeek AI, 2025b), **Gemini-2.5-Pro** (Google Cloud, 2025a), **GPT-OSS-120B** (OpenAI et al., 2025), and **Qwen3-Coder** (Qwen Team, 2025) using the same prompts. As shown in Table 1, the current response accuracy is below 90% and should be improved to support high-interaction terminal simulation. Besides, using commercial LLM APIs introduces large delays under different command testing. For example, the 95th-percentile latency is between 2.76 and 12.34s.

Then, we compare MANTIS against state-of-the-art LLM-based terminal interaction baselines, including ShellLM (Sladić et al., 2024), HoneyLLM (Guan et al., 2024), and HoneyGPT (Wang et al., 2025). For fair comparison, we re-implement all baselines using the GPT-4o model with different prompts in the corresponding papers. We build four test sets: (i) CAW: 200 well-formed commands sampled from the *Response Dataset* to evaluate the correctness of command execution and output formatting. (ii) ROB: 500 syntax-error commands to evaluate robustness to confusing or malformed commands. (iii) PI-RES: 570 prompt-injection inputs (Toyer et al., 2023) to evaluate resistance to prompt-level misuse that induces non-standard or misleading outputs. (iv) CTX: 1,000 multi-turn interaction sessions with diverse interaction patterns to evaluate context consistency.

First, we measure multi-turn context consistency (CTX) score, as well as the accuracy in Table 1 by comparing the model output against the ground-truth terminal output using exact string matching. A session is counted as correct if strings are identical at each evaluated turn. For example, we mark a turn correct if (i) all state-dependent entities (paths, filenames, permissions, counts) are consistent with the snapshot, and (ii) terminal-faithful formatting is preserved (line breaks and ordering). All other cases are counted as incorrect.

We remove inherently non-deterministic fields (e.g., timestamps, process IDs, and run-specific numeric IDs such as job IDs) and strip rendering artifacts that do not alter semantic content (e.g., shell prompt prefixes and escape-related symbols). This pre-processing does not affect evaluation accuracy as we write non-deterministic fields (e.g., timestamps, process IDs, and run-specific numeric IDs such as job IDs) into a log file, instead of being memorized by LLMs. We observe that even on the same machine, rerunning the same job spawns a new process instance that is typically assigned a different PID by the OS, leading to mismatches that are unrelated to semantic correctness. As a result, these critical changes are preserved in the retained session logs and explicitly appended to the next-round prompt when generating the next response. We want to emphasize that logging the process ID in our agentic file system guarantees consistency.

Second, for command execution accuracy on normal commands (CAW), we manually verify every instance while applying the same normalization and exact-match criteria as in our automatic evaluation (i.e., manual checks only confirm whether the normalized strings should be considered identical under the predefined rules). Third, for robustness to syntax error commands (ROB) and resistance to prompt injection inputs (PI-RES), we score outputs fully automatically by exactly matching the normalized model output to the required target output for each prompt.

As shown in Figure 2, our system consistently outperforms all baselines across all evaluation dimensions. For example, MANTIS achieves 95.36% accuracy on the multi-turn dataset (CTX), outperforming state-of-the-art systems (e.g., HoneyGPT 86.16% accuracy) and commercial APIs

Table 2: Mean opinion score on multi-turn terminal interaction quality evaluation (1: Bad, 5: Excellent).

Metric	5	4	3	2	1	Avg	Std
Interactivity	15	2	3	0	0	4.60	0.754
Latency	14	4	2	0	0	4.60	0.681
Consistency	15	4	1	0	0	4.7	0.571
Format	16	4	0	0	0	4.80	0.410
Robustness	14	5	1	0	0	4.65	0.587

in Table 1. For syntax-error handling, MANTIS achieves 97.53% accuracy, while HoneyGPT only achieves 89.26% accuracy. Overall, MANTIS reduces incorrect responses and formatting failures while maintaining stable multi-turn behavior, underscoring the benefit of explicit file-system state and the integrated agent framework.

4.3 User Evaluation on Terminal Simulation

We conduct a user study to assess the overall realism of MANTIS for terminal simulation. We run the terminal simulation on a dedicated Ubuntu server equipped with an NVIDIA RTX 5090 GPU. The server exposes SSH on port 22 to the public Internet, allowing external clients to connect and interact with the simulated terminal environment. To reduce operational risk, the deployment is isolated from any internal production infrastructure and is restricted to the services required for terminal interaction. This setup enables us to observe real-world adversarial behaviors (e.g., automated authentication probing and interactive sessions) while evaluating the practicality of our LLM-driven terminal simulation under realistic network conditions.

We recruited 20 graduate students with extensive experience in Linux-related environments. Each participant interacts with MANTIS for full terminal simulation, including command execution, multi-turn interactions, and error handling. Participants are asked to complete a set of exploratory tasks and freely probe MANTIS behavior. Each participant gives opinion scores on interactivity with commands, response latency, consistency with history states, output format correctness, and robustness to syntax errors and prompt injection. As shown in Table 2, 85% of participants rated good or excellent for interactivity, reporting that MANTIS produced realistic terminal responses and behaved comparably to a real Linux environment. In addition, 80% of participants gave MANTIS as a score of 5 for the stability of output formatting and the coherence of system behavior.

Table 3: Different routing model performance.

Backbone Model	Accuracy	Macro F1
DistilBERT-base-uncased	98.09	0.930
RoBERTa-base	86.15	0.823
DeBERTa-v3-base	85.99	0.794
ModernBERT-base	**98.64**	**0.976**
Electra-base	93.99	0.917
Albert-base-v2	97.71	0.960

DistilBERT-base-uncased (Sanh et al., 2019); RoBERTa-base (Liu et al., 2019); DeBERTa-v3-base (He et al., 2021); ModernBERT-base (Warner et al., 2024); Electra-base (Google, 2020); Albert-base-v2 (Lan et al., 2019).

4.4 Ablation Study

This ablation study examines the contribution of each core component in MANTIS. The routing module determines whether an incoming command should be executed, simulated, or rejected, and therefore mainly contributes to safe control flow and robustness against malformed or adversarial inputs. The history-pruning module regulates how prior interactions are preserved in the working context, directly affecting long-horizon state consistency, response latency, and token efficiency. The response model is responsible for terminal-output generation and primarily contributes to local response quality, including formatting fidelity and command-to-output correctness.

4.4.1 Routing Tool Model Evaluation

We evaluate the *Arbiter Agent*'s routing tool model performance. The router tool classifies each incoming command into three categories: *read* (benign, non-state-changing queries), *write* (state-changing operations), and *rejection* (syntax-error or prompt-injection inputs). We split the *Routing Dataset* into 80% training, 10% validation, and 10% test sets using a label-stratified sampling to preserve the class distribution.

Table 3 reports the classification performance of six encoder-based models after domain-specific fine-tuning. Among all candidates, the fine-tuned ModernBERT-base model achieves the best overall performance, reaching 98.64% accuracy with a Macro-F1 of 0.976, indicating consistently strong performance across *read*, *write*, and *rejection*. This is particularly important because *write* and *rejection* cases are less frequent but more safety-critical, and misclassification can lead to incorrect subsequent behavior. Accurate routing is therefore essential because it determines whether a command is executed against the agentic file-system state, triggers a state update, or is rejected before reaching downstream agents.

Table 4: History pruning evaluation in the main setting and under a stronger backbone.

Metric	Full history	Pruned history
Consistency (%)	92.01	96.07 (↑)
Response Latency (s)	1.574	1.298 (↓)
Token Consumption	3,155,012	2,679,863 (↓)
GPT 5.2 Consis. (%)	93.16	96.43 (↑)
GPT 5.2 Latency (s)	1.848	1.785 (↓)

Note: The main setting uses Llama 3.2 3B (FT) as the response backbone. GPT 5.2 results are included to verify that the benefit of pruning remains consistent under a stronger backbone.

4.4.2 History Pruning Evaluation

To assess whether history pruning preserves agent behavior while reducing cost, we evaluate 20 independent sessions (50–100 steps each). For each step, we fix the same incoming command and the same pre-execution snapshot, and then generate outputs under the same settings using either (i) the full interaction history and (ii) the pruned working history produced by our method. We evaluate effectiveness using three metrics: response consistency, response latency, and total token consumption.

As shown in Table 4, pruning maintains strong response consistency and slightly improves it (96.07 vs. 92.01), indicating that removing low-utility turns does not harm state-faithful generation and can improve stability by reducing distracting context. Pruning also reduces average response latency (1.298 s vs. 1.574 s) and lowers total tokens (2,679,863 vs. 3,155,012). We observe that token savings grow over time as the unpruned history accumulates. For example, pruning saves about 10% of tokens around interaction turn-55, and the savings increase to about 18% by around interaction turn-75. In the GPT 5.2 rows of Table 4, pruning likewise improves both consistency and response latency, indicating that the benefit of pruning is preserved under a stronger backbone.

Overall, the proposed history-pruning method achieves response correctness comparable to, and in some cases better than, retaining the full interaction history, while substantially reducing inference latency and token cost in longer sessions.

4.4.3 Response Model Evaluation

We evaluate the model in *Response Agent*, focusing solely on terminal response generation without planning, routing, and state-management. We report results on a curated set of 200 shell commands, comparing multiple open-source models in both base and fine-tuned forms. Fine-tuning is performed on *Response Dataset*. As shown in Table 5, L3.2-3B (FT) matches DS-R1-DQ7B (FT) in accuracy (80.5%) while responding faster (0.596 s vs. 1.184 s). Although Q3-8B (FT) attains the highest accuracy (85.0%), it comes with substantially higher latency. Considering a balance between accuracy and latency, we use L3.2-3B (FT) in our *Response Agent* for output generation.

Table 5: Performance of Base and Fine-tuned Models.

Model	Acc. (%)	Resp. Time (s)	Total Tokens
L3-8B	42.0	1.219	30,736
L3-8B (FT)	76.5	1.225	28,938
DS-R1-DQ7B	49.5	1.063	29,635
DS-R1-DQ7B (FT)	80.5	1.184	28,386
OpenC-8B	42.5	1.234	30,744
OpenC-8B (FT)	69.5	1.366	29,194
L3.2-3B	34.5	0.644	30,343
L3.2-3B (FT)	**80.5**	**0.596**	**29,162**
Q2.5-Coder-7B	53.5	0.976	28,345
Q2.5-Coder-7B (FT)	75.0	1.032	28,084
DS-ProvV2-7B	40.5	0.939	29,367
DS-ProvV2-7B (FT)	69.5	0.942	28,703
Q3-8B	67.5	1.357	30,467
Q3-8B (FT)	85.0	1.377	33,762
CL-7B	48.5	1.125	31,126
CL-7B (FT)	73.0	1.167	34,371
CG-7B	54.5	1.396	30,821
CG-7B (FT)	67.5	1.264	31,632

Abbreviations: L3-8B = Llama-3-8B (Meta, 2024a); DS-R1-DQ7B = DeepSeek-R1-Distill-Qwen-7B (DeepSeek-AI, 2025); OpenC-8B = Opencoder-8B-instruct (Huang et al., 2024); L3.2-3B = Llama-3.2-3B-instruct (Meta, 2024b); Q2.5-Coder-7B = Qwen2.5-Coder-7B (Hui et al., 2024); DS-ProvV2-7B = DeepSeek-Prover-V2-7B (DeepSeek AI, 2025a); Q3-8B = Qwen3-8B-Base (Team, 2025); CL-7B = Codellama-7B (Meta, 2023); CG-7B = CodeGemma-7B (Google, 2024).

5 Conclusion

In this work, we proposed a high-interaction terminal simulation framework that replaces brittle scripts and pure LLM-based methods with a multi-agent architecture grounded in an explicit file-system-backed terminal state. We curated the first large-scale datasets for terminal interaction training and benchmarking, and introduced a context-efficient history pruning algorithm to maintain long-horizon consistency. Experiments and a user study demonstrate improvements in output-format fidelity, state consistency, and efficiency. The primary target application is terminal honeypots, a long-studied setting for attracting external attackers and analyzing their strategies. More broadly, we believe this problem offers a promising new testbed for studying long-horizon, stateful, and controllable language interaction.

Limitations

Generalization beyond the curated environment. The datasets are derived from a controlled Kali Linux setup with a fixed workspace layout, which may limit external validity. In addition, shell behavior is highly nuanced, including redirection and pipelines, background jobs, environment variables, and inherently nondeterministic outputs (timestamps, process IDs, network-dependent responses). Consequently, more efforts are required for transferring the proposed approach to other operating systems and distributions, alternative filesystem organizations, locale and encoding settings, or different shells and configuration profiles.

Coverage and explainability trade-off in history pruning. The current history-pruning mechanism relies on deterministic, code-driven criteria derived from snapshot and state comparisons. While transparent and explainable, it may miss latent dependencies and thus prune context needed for edge cases (e.g., implicit dependencies, indirect side effects, or higher-level semantic links between past actions and current state). LLM-assisted pruning could expand coverage by inferring such dependencies, but would reduce interpretability in safety-critical and adversarial settings.

Cost, latency, and accuracy trade-offs in deployment. MANTIS introduces practical deployment trade-offs among token consumption, inference latency, and response accuracy. Larger backbone models typically improve output fidelity and state consistency, but they also increase per-interaction token cost and end-to-end latency. The latency can hinder scalability for long sessions or high-concurrency deployments. Conversely, smaller models reduce cost and improve responsiveness but may degrade robustness (e.g., formatting stability, rare-command handling, and adversarial resilience). Optimizing these coupled trade-offs, potentially via adaptive model selection, caching, or mixed-precision, remains an important direction. Based on the optimization, we will further deploy the multi-agent-driven terminal simulation as a honeypot to attract external attackers in the real world for threat intelligence collection.

Acknowledgements

We appreciate the constructive feedback from reviewers and meta-reviewers. This work was partially supported by the National Science Foundation (NSF) Grant No. 2321270 and No. 2404741.

References

Sotiris Anagnostidis, Dario Pavllo, Luca Biggio, Lorenzo Noci, Aurelien Lucchi, and Thomas Hofmann. 2023. Dynamic context pruning for efficient and interpretable autoregressive transformers. *Advances in Neural Information Processing Systems*, 36:65202–65223.

Anthropic. 2025. Claude 4. Official Anthropic announcement for Claude 4 (including Claude Opus 4 and Claude Sonnet 4), accessed April 17, 2026.

Robert A Bridges, Thomas R Mitchell, Mauricio Muñoz, and Ted Henriksson. 2025. Sok: Honeypots & llms, more than the sum of their parts? *arXiv preprint arXiv:2510.25939*.

Bocheng Chen, Hanqing Guo, Guangjing Wang, Yuanda Wang, and Qiben Yan. 2024a. The dark side of human feedback: Poisoning large language models via user inputs. *arXiv preprint arXiv:2409.00787*.

Bocheng Chen, Nikolay Ivanov, Guangjing Wang, and Qiben Yan. 2024b. Multi-turn hidden backdoor in large language model-powered chatbot models. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, pages 1316–1330.

Bocheng Chen, Guangjing Wang, Hanqing Guo, Yuanda Wang, and Qiben Yan. 2023. Understanding multi-turn toxic behaviors in open-domain chatbots. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 282–296.

Jason Aljenova Christli, Charles Lim, and Yevonnael Andrew. 2024. Ai-enhanced honeypots: Leveraging llm for adaptive cybersecurity responses. In *2024 16th International Conference on Information Technology and Electrical Engineering (ICITEE)*, pages 451–456.

DeepSeek AI. 2025a. DeepSeek-Prover-V2-7B. Hugging Face model card, accessed April 17, 2026.

DeepSeek AI. 2025b. DeepSeek-R1-0528. Hugging Face, accessed April 17, 2026.

DeepSeek-AI. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Jiangyi Deng, Xinfeng Li, Yanjiao Chen, Yijie Bai, Haiqin Weng, Yan Liu, Tao Wei, and Wenyuan Xu. 2025. RACONTEUR: A Knowledgeable, Insightful, and Portable LLM-Powered Shell Command Explainer. In *NDSS Symposium 2025*.

Wenjun Fan, Zichen Yang, Yuanzhen Liu, Lang Qin, and Jia Liu. 2024. Honeyllm: A large language model-powered medium-interaction honeypot. In *International Conference on Information and Communications Security*, pages 253–272. Springer.

Qichen Fu, Minsik Cho, Thomas Merth, Sachin Mehta, Mohammad Rastegari, and Mahyar Najibi. 2024. Lazyllm: Dynamic token pruning for efficient long context llm inference. *Preprint*, arXiv:2407.14057.

Google. 2020. ELECTRA Base Discriminator. Hugging Face model card, accessed April 17, 2026.

Google. 2024. CodeGemma 7B. Hugging Face model card, accessed April 17, 2026.

Google Cloud. 2025a. Gemini 2.5 Pro. Vertex AI documentation, accessed April 17, 2026.

Google Cloud. 2025b. Vertex AI. Google Cloud official website, accessed December 25, 2025.

Chongqi Guan, Guohong Cao, and Sencun Zhu. 2024. Honeyllm: Enabling shell honeypots with large language models. In *2024 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9.

Jingxuan He and Martin Vechev. 2023. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1865–1879, New York, NY, USA. Association for Computing Machinery.

Pengcheng He, Jianfeng Gao, and Weizhu Chen. 2021. Debertav3: Improving deberta using electra-style pre-training with gradient-disentangled embedding sharing. *Preprint*, arXiv:2111.09543.

Samuel Holister, Blaine Kingsley, Xavier Martin, Felix Ashworth, Andrew Scolto, and Peter McAllister. 2025. Contextual self-referential memory trajectories for large language model consistency.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J. Yang, J. H. Liu, Chenchen Zhang, Linzheng Chai, Ruifeng Yuan, Zhaoxiang Zhang, Jie Fu, Qian Liu, Ge Zhang, Zili Wang, Yuan Qi, Yinghui Xu, and Wei Chu. 2024. Opencoder: The open cookbook for top-tier code large language models.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Qi Jia, Xiang Yue, Tuney Zheng, Jie Huang, and Bill Yuchen Lin. 2025. SimulBench: Evaluating language models with creative simulation tasks. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 8118–8131, Albuquerque, New Mexico. Association for Computational Linguistics.

Samuel T King and Peter M Chen. 2003. Backtracking intrusions. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 223–236.

Vishal Kumar, Sawan Bhardwaj, Pradeep Chouksey, Praveen Sadotra, and Mayank Chopra. 2024. Emerging trends in honeypot research: A review of applications and techniques. *International Journal of Human Computations & Intelligence*, 3(6):370–377.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. ALBERT: A lite BERT for self-supervised learning of language representations. *CoRR*, abs/1909.11942.

Jiaheng Liu, Dawei Zhu, Zhiqi Bai, Yancheng He, Huanxuan Liao, Haoran Que, Zekun Wang, Chenchen Zhang, Ge Zhang, Jiebin Zhang, Yuanxing Zhang, Zhuo Chen, Hangyu Guo, Shilong Li, Ziqiang Liu, Yong Shan, Yifan Song, Jiayi Tian, Wenhao Wu, and 18 others. 2025. A comprehensive survey on long context language modeling. *Preprint*, arXiv:2503.17407.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts. *Preprint*, arXiv:2307.03172.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692.

Meta. 2023. Code Llama 7B. Hugging Face model card, accessed April 17, 2026.

Meta. 2024a. Llama 3.1 8B. Hugging Face model card, accessed April 17, 2026.

Meta. 2024b. Llama 3.2 3B Instruct. Hugging Face model card, accessed April 17, 2026.

Microsoft. 2025. Event Tracing for Windows. Microsoft Learn, accessed December 28, 2025.

MITRE. 2025. MITRE ATT&CK Framework. MITRE ATT&CK official website, accessed December 28, 2025.

Ali Modarressi, Hanieh Deilamsalehy, Franck Dernoncourt, Trung Bui, Ryan A. Rossi, Seunghyun Yoon, and Hinrich Schütze. 2025. Nolima: Long-context evaluation beyond literal matching. *Preprint*, arXiv:2502.05167.

Muhammad Ahmed Mohsin, Muhammad Umer, Ahsan Bilal, Zeeshan Memon, Muhammad Ibtsaam Qadir, Sagnik Bhattacharya, Hassan Rizwan, Abhiram R. Gorle, Maahe Zehra Kazmi, Ayesha Mohsin, Muhammad Usman Rafique, Zihao He, Pulkit Mehta, Muhammad Ali Jamshed, and John M. Cioffi. 2025. On the fundamental limits of llms at scale. *Preprint*, arXiv:2511.12869.

Michel Oosterhof, dependabot[bot], fe7ch, github-actions[bot], Upi Tamminen, g0tmi1k, Dave Germiquet, SecPascal, Olivier Bilodeau, Guilherme Borges,

lelonek1, Peter Šufliarsky, IridiumXOR, Jc2k, Katarina Durechova, NunoNovais, Sam Edwards, Claud Xiao, Aabed, and 11 others. 2025. cowrie/cowrie. https://github.com/cowrie/cowrie.

OpenAI, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, Che Chang, and 107 others. 2025. gpt-oss-120b & gpt-oss-20b model card. *Preprint*, arXiv:2508.10925.

Hakan T. Otal and M. Abdullah Canbaz. 2024. Llm honeypot: Leveraging large language models as advanced interactive honeypot systems. In *2024 IEEE Conference on Communications and Network Security (CNS)*, page 1–6. IEEE.

Qwen Team. 2025. Qwen 3 Coder. Google Cloud Vertex AI documentation, accessed April 18, 2026.

Red Hat. 2025. Red Hat Documentation. Accessed December 28, 2025.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *ArXiv*, abs/1910.01108.

Anıl Sezgin and Aytuğ Boyacı. 2025. Decoypot: A large language model-driven web api honeypot for realistic attacker engagement. *Computers & Security*, page 104458.

Jie Shi, Sihang Jiang, Bo Xu, Jiaqing Liang, Yanghua Xiao, and Wei Wang. 2023. Shellgpt: Generative pre-trained transformer model for shell language understanding. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 671–682.

Leyi Shi, Yuwen Cui, Xu Han, Honglong Chen, and Deli Liu. 2019. Mimicry honeypot: an evolutionary decoy system. *Int. J. High Perform. Comput. Netw.*, 14(2):157–164.

Muris Sladić, Veronica Valeros, Carlos Catania, and Sebastian Garcia. 2024. Llm in the shell: Generative honeypots. In *2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 430–435.

Muris Sladić, Veronica Valeros, Carlos Catania, and Sebastian Garcia. 2025. Vellmes: A high-interaction ai-based deception framework. In *2025 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, page 671–679. IEEE.

Qwen Team. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Sam Toyer, Olivia Watkins, Ethan Adrian Mendes, Justin Svegliato, Luke Bailey, Tiffany Wang, Isaac Ong, Karim Elmaaroufi, Pieter Abbeel, Trevor Darrell, and 1 others. 2023. Tensor trust: Interpretable prompt injection attacks from an online game. *arXiv preprint arXiv:2311.01011*.

Vulhub. 2025. GitHub - vulhub/vulhub: Pre-Built Vulnerable Environments Based on Docker-Compose — github.com. https://github.com/vulhub/vulhub. [Accessed 21-06-2025].

Guangjing Wang, Ce Zhou, Yuanda Wang, Bocheng Chen, Hanqing Guo, and Qiben Yan. 2023. Beyond boundaries: A comprehensive survey of transferable attacks on ai systems. *arXiv preprint arXiv:2311.11796*.

Ziyang Wang, Jianzhou You, Haining Wang, Tianwei Yuan, Shichao Lv, Yang Wang, and Limin Sun. 2025. Honeygpt: Breaking the trilemma in terminal honeypots with large language model. *Preprint*, arXiv:2406.01882.

Benjamin Warner, Antoine Chaffin, Benjamin Clavié, Orion Weller, Oskar Hallström, Said Taghadouini, Alexis Gallagher, Raja Biswas, Faisal Ladhak, Tom Aarsen, Nathan Cooper, Griffin Adams, Jeremy Howard, and Iacopo Poli. 2024. Smarter, better, faster, longer: A modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference. *Preprint*, arXiv:2412.13663.

Kai Wei and Guangjing Wang. 2025. Poster: Agentic shell honeypot using structured logging. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pages 4803–4805.

Ce Zhou, Qian Li, Chen Li, Jun Yu, Yixin Liu, Guangjing Wang, Kai Zhang, Cheng Ji, Qiben Yan, Lifang He, and 1 others. 2025. A comprehensive survey on pretrained foundation models: A history from bert to chatgpt. *International Journal of Machine Learning and Cybernetics*, 16(12):9851–9915.

A Log File System

All logs are stored in JSON format to maximize interoperability and machine readability. The JSON-based logging structure enhances automated analysis, event correlation, and integration. Authentication logs and session logs follow an append-only pattern, where new entries are recorded sequentially using the format above. In contrast, system logs primarily store the environment snapshot, a dynamically maintained structured representation of the terminal simulator's current state. Rather than appending discrete messages, it is updated in place to reflect key system properties, including the directory tree, file metadata, and contents, and privilege-related information.

A.1 System Snapshot Representation

The system snapshot is represented as a structured JSON object that captures the observable terminal state at a given interaction. It includes metadata, security-relevant runtime attributes, and a hierarchical filesystem representation. Key fields are summarized as follows:

- **timestamp** and **step**: The recording time and the interaction index.
- **cwd**: The current working directory.
- **identity**: The active user and identity attributes (e.g., user name, UID/GID, effective UID/GID, and group list).
- **privilege**: Privilege-related settings, including whether privilege escalation via sudo is permitted (privilege.sudo_available), as well as umask and capability sets.
- **persistence**: Persistent execution hooks and service configurations (e.g., enabled systemd units, cron state, and SSH authorized keys fingerprints).
- **network**: Externally visible network state, including listening ports and condensed firewall/routing/interface fingerprints.
- **critical_configs**: Fingerprints and metadata of a predefined whitelist of security-critical configuration files (e.g., /etc/passwd, /etc/ssh/sshd_config).
- **filesystem**: A recursive representation of directories and files under allowed roots, including per-file contents and metadata (e.g., mode, owner, timestamps, size, and content hashes).
- **vulnerabilities**: A list of simulated vulnerabilities exposed by the system.
- **last_output**: The terminal output produced by the previous interaction.

Listing 1 shows a truncated example snapshot used by our terminal simulator. For the complete snapshot, please refer to our source code and data. Listing 2 is an example of JSONL-formatted SSH events. Listing 3 shows example command–response records used in our evaluation.

B Datasets

B.1 Response Agent Fine-tune Dataset

The Response Agent is trained on a domain-specific dataset consisting of 28,045 real-world terminal input–output pairs, manually collected from a Kali Linux environment. The dataset covers a broad range of commonly used Linux commands and their corresponding system-level responses, with careful curation to preserve realistic output formatting and execution semantics.

Each training instance follows a structured command–response representation designed to closely mirror real Linux terminal behavior. An instance comprises a system-level instruction that constrains the model to act strictly as a Linux terminal, a user-issued shell command, and the corresponding ground-truth terminal output. Outputs are recorded exactly as produced by the operating system, including formatted standard output, error messages, and empty responses for commands that succeed silently (e.g., mkdir). All outputs are enclosed in code blocks to preserve formatting fidelity. This design enables the model to learn not only command semantics but also realistic output structure and execution behavior, including cases where no output is emitted.

B.2 System Evaluation Dataset

This dataset contains 200 carefully selected instructions from the Response Agent fine-tuning dataset. These commands are frequently used and require strict, format-faithful terminal outputs.

B.3 Multi-turn Dialogue Dataset

Our dataset consists of manually crafted multi-turn interaction sessions that simulate diverse attacker behaviors, with conversation lengths ranging from 3 to 20 turns. Approximately 800 sessions contain fewer than five turns, while the remaining sessions are distributed across longer ranges in a largely random manner. Since all sessions were created by humans rather than generated automatically, the dataset captures a wide variety of simulated adversarial actions, including file creation and deletion, directory navigation, system state inspection, configuration modification, and exploratory probing. This intentional diversity produces a realistic spectrum of behavior patterns and ensures that the dataset covers both short, task-focused interactions and longer, more complex attack sequences.

Listing 1: Example system snapshot in JSON format (truncated)

```
{
  "timestamp": "2025-12-15T21:58:27Z",
  "cwd": "/home/user/logs",
  "identity": { "user": "user", "uid": 1000, "gid": 1000, "euid": 1000, "egid": 1000 },
  "privilege": { "sudo_available": true, "umask": "0022" },

  "persistence": { "systemd": { "enabled_units": [ ] }, "cron": { "system_crontab_hash": "" } },

  "network": {
    "listening_ports": [
      { "proto": "tcp", "ip": "0.0.0.0", "port": 22, "process": "sshd" },
      { "proto": "tcp", "ip": "0.0.0.0", "port": 8080, "process": "java" }
    ]
  },

  "filesystem": {
    "/home/user": {
      "files": ["README.txt"],
      "folders": { "logs": { "files": ["access.log", "error.log"] } },
      "file_meta": { "README.txt": { "mode_octal": "0644", "size": 22, "hash": "3025cf2e..." } }
    },
    "/tmp": { "files": [ ] },
    "_omitted": true
  }
}
```

Listing 2: Example auth snapshot in JSON format

```
{"timestamp":"2025-12-01␣15:52:48","event":"ssh_auth_attempt","proto":"ssh","local_port":22,"
    session_id":"<uuid>","username":"<user>","hostname":"<host>","remote_ip":"<ip>","
    remote_port":"<port>","success":false,"note":"password␣auth␣failed","password":"<redacted>"}
{"timestamp":"2025-12-01␣15:52:49","event":"disconnect","proto":"ssh","local_port":22,"session_id":"
    <uuid>","username":"root","hostname":"<host>","remote_ip":"<ip>","remote_port":"<port>",
    "success":true,"note":"session␣closed"}
{"timestamp":"2025-12-01␣15:52:55","event":"connect","proto":"ssh","local_port":22,"session_id":"<
    uuid>","username":"root","hostname":"<host>","remote_ip":"<ip>","remote_port":"<port>","
    success":true,"note":"session␣opened"}
```

Listing 3: Example session log in JSON format

```
{
  "timestamp": "2025-11-25T21:21:25Z",
  "command": "ls␣-al",
  "response": "total␣20
drwxr-xr-x␣5␣user␣user␣4096␣Nov␣25␣16:20␣.
drwxr-xr-x␣3␣user␣user␣4096␣Nov␣25␣16:20␣..
drwxr-xr-x␣2␣user␣user␣4096␣Nov␣25␣16:20␣test123
drwxr-xr-x␣2␣user␣user␣4096␣Nov␣25␣16:21␣test1234
drwxr-xr-x␣4␣user␣user␣4096␣Nov␣25␣16:20␣logs
-rw-r--r--␣1␣user␣user␣␣␣10␣Nov␣25␣16:20␣README.txt
-rw-r--r--␣1␣user␣user␣␣␣12␣Nov␣25␣16:20␣main.py
drwxr-xr-x␣4␣user␣user␣4096␣Nov␣25␣16:20␣ops_ec
drwxr-xr-x␣2␣user␣user␣4096␣Nov␣25␣16:20␣test
-rw-r--r--␣1␣user␣user␣␣␣␣6␣Nov␣25␣16:20␣test.txt",
  "classification": "safe"
}
{
  "timestamp": "2025-11-27T00:32:02Z",
  "command": "ls",
  "response": "README.txt␣␣main.py␣␣test.txt␣␣logs␣␣ops_ec␣␣test␣␣test123␣␣test1234",
  "classification": "safe"
}
```

You are the Strategic Agent for a Linux honeypot terminal. Task: Produce the exact terminal output and response advice. You MUST be consistent with the provided snapshot and the interaction history.
Single source of truth (snapshot): - If a path/name is not present in the snapshot, it does not exist. - Never assume anything not explicitly present in the snapshot.
Read snapshot: - snapshot is a JSON object that contains at least: * cwd: current working directory (string) * filesystem: directory tree (absolute paths / nested nodes) * (optional) processes/services/network/vulnerabilities/plan_log - Resolve relative paths using snapshot.cwd. - Use permissions/owner/timestamps/content fingerprints if provided.
Output requirements: - Output the parsed version of the current command, along with suggested responses. - Do NOT echo or repeat the input command. - Preserve realistic terminal formatting (exact newlines, spacing, canonical error messages). - Do not ask questions.
Strict constraints: - Do NOT invent nonexistent files/dirs/permissions/users/processes/-ports. - Do NOT invent new state changes. - When uncertain, prefer a safe canonical failure message (e.g., 'No such file or directory').

Figure 3: Instruction for the Strategic Agent: read snapshot and generate advice

C Strategic Agent Prompt

Figure 3 shows an example of prompts used by the Strategic Agent.

D History Pruning

D.1 Problem Formulation

We consider an interactive terminal environment evolving over discrete time steps. At time step t, the system observes a new interaction

$$e_t = (c_t, y_t),$$

where c_t denotes the command and y_t is the response or execution result. The environment maintains a structured system snapshot s_t capturing observable system state, including filesystem contents, permissions and identities, service and persistence configuration, and network status.

We maintain: (i) a persistent history store H_t that records all past interactions, and (ii) a bounded working context $W_t \subset H_t$ that is provided to the language model. The working context is subject to a hard capacity constraint:

$$|C_t| \leq K.$$

Crucially, the pruning policy is *stateful*. The previous working context W_{t-1} is treated as an explicit internal state of the controller. The system evolves according to:

$$H_t \leftarrow H_{t-1} \,\|\, e_t,$$
$$U(t) \leftarrow \mathrm{Evict}(e_t, \mathcal{W}_{t-1}, s_t, s_{t-1}),$$

where Evict denotes the history pruning policy.

The pruning strategy must satisfy the following:

- **Hard constraint:** $|W_t| \leq K$.
- **Retention priority:** interactions that induce higher-grade system state transitions are retained preferentially.
- **Eviction priority:** read-only duplicates, low-grade events, stale entries, and redundant interactions are removed first.

D.2 Unimportance as Deletability

Unimportance characterizes the *cost of removal* of a history entry under a hard capacity constraint, rather than its intrinsic importance. Intuitively, it measures how little explanatory power would be lost if an interaction were evicted from the working context. For each interaction h, we define a lexicographically ordered deletion key as

$$U(h) = \big(U_{\text{dup}}(h),\ U_{\text{std}}(h),\ U_{\text{stale}}(h)\big),$$

which induces a deterministic priority over candidate evictions.

Each component is computed using simple, rule-based criteria derived solely from observable state differences and interaction order:

- **Redundancy $U_{\text{dup}}(h)$.** This term captures repeated commands within the working context. If there exists an interaction in the current context whose stored command is identical to that of h, we increase the redundancy count. Concretely, whenever a new entry arrives, we

increment $U_{\text{red}}(\cdot)$ by 1 for *all* existing interactions whose stored command matches the new command, so commands that appear more frequently accumulate larger $U_{\text{red}}(\cdot)$ and are more likely to be evicted.

- **Grade-based priority $U_{\textbf{std}}(h)$.** This term directly reflects the discrete state-transition grade. We use the grade itself as the comparison key, where higher-grade interactions are considered more deletable.

- **Temporal staleness $U_{\textbf{stale}}(h)$.** This term captures how outdated an interaction is with respect to the current time step. It is derived from interaction order rather than wall-clock time, such that older interactions receive larger staleness values. Within the same grade, this induces a chronological eviction policy.

In the resulting ordering, interactions are compared first by redundancy, then by grade-based priority, with temporal staleness used only to break ties. Concretely, entries with larger $U_{\text{red}}(\cdot)$ are evicted first; for equal redundancy, lower-grade interactions are removed before higher-grade ones; among the remaining candidates, older interactions are removed before more recent ones. By relying on lexicographic comparison instead of scalar weighting or learned parameters, this formulation yields a transparent, deterministic, and reproducible pruning behavior.

D.3 State Difference

We then compute the structured snapshot transition $\Delta s_t \triangleq (s_{t-1}, s_t)$, which captures concrete side effects induced by executing c_t. We decompose Δs_t into the following dimensions:

- Δs_t^{perm} (**identity/privilege delta**): any change in identity or privilege that alters execution authority, including updates to user/identity attributes (e.g., identity.user, UID/GID, effective UID/GID, and the group list) and privilege-related settings (e.g., privilege.sudo_available, privilege.umask, and capability sets). In addition, permission-only updates in filesystem metadata (e.g., owner/-mode, SUID/SGID, ACLs) are also counted as Δs_t^{perm} even if file contents remain unchanged.

- $\Delta s_t^{\text{persist}}$ (**persistence/services delta**): any change in the persistence group, including persistence hooks and service configuration state (e.g., systemd, cron, and ssh authorized-keys fingerprints) that can affect future executions.

- Δs_t^{net} (**network delta**): any change in the network group that alters externally visible network behavior, including the set of listening ports and condensed fingerprints of firewall, routing, and interface state.

- Δs_t^{cfg} (**critical configuration delta**): any modification to the predefined whitelist of security-critical configuration files recorded under critical_configs.files, detected via changes in the per-file fingerprint and metadata (e.g., hash, mode_octal, uid, gid) for paths such as /etc/passwd and /etc/ssh/sshd_config. This category is triggered regardless of edit magnitude.

- Δs_t^{fs} (**filesystem content/structure delta**): any filesystem side effect captured in the filesystem group within the allowed roots, including directory-structure changes (created/deleted files or folders reflected in files/folders), file content updates (changes in file_contents or content hashes), and write-induced metadata updates (e.g., mtime/size in file_meta).

- Δs_t^{read} (**read-only**): interactions with no persistent side effects, where none of the above groups, identity, privilege, persistence, network, critical_configs, or filesystem changes between s_{t-1} and s_t; differences are limited to bookkeeping/output fields such as timestamp, step, and last_output (and optionally cwd if treated as an ephemeral session variable).

- **Grade 0 (Critical State Transition):** identity or permission changes (Δs_t^{perm}); establishment of persistence or services ($\Delta s_t^{\text{persist}}$); network exposure or policy changes (Δs_t^{net}).

- **Grade 1 (Sensitive System Modification):** modifications to critical system configuration files (Δs_t^{cfg}).

- **Grade 2 (Local State Modification):** filesystem writes, creations, deletions, or content modifications (Δs_t^{fs}).

- **Grade 3 (No Persistent State Change):** read-only interactions or failed executions with no observable persistent side effects (Δs_t^{read}).

When multiple change types co-occur, we conservatively assign $U_{\text{std}}(t)$ to the most severe applicable grade, i.e., the minimum grade among all triggered categories.

D.4 Capacity-Constrained Pruning with Grade-Aware Ordering

The pruning policy aims to curb the accumulation of redundant or low-utility context by maintaining a bounded working memory, i.e., enforcing $|W_t| \leq K$ over time. This hard capacity constraint induces a replacement strategy:

- **Soft mode:** when $|W_t| < K$, interactions are ranked by grade and then by deletability.
- **Hard mode:** when $|W_t| > K$, interactions with the lowest grades and highest deletability are evicted until feasibility is restored.

E Vulnerability simulation

Whenever the honeypot needs to expose an attack surface, it selects a set of CVE-based vulnerabilities and records them in the snapshot as an explicit vulnerability list (e.g., "vulnerabilities": ["CVE-2021-4034: ...", "Port 22 open with weak SSH credentials."]). Each entry includes a CVE identifier (or observable weakness) and a one-line visible description, which serves as the single source of truth for subsequent deception and cross-command consistency. The system then uses each CVE identifier as a key to retrieve a corresponding vulnerability profile from Vulhub (Vulhub, 2025).

Each profile contains only the minimal information required for simulation: the affected component and claimed version; the exposure surface (port/service/banner/path); the artifacts to stage (configuration files, logs, directory structure, and plausible traces); and response templates for common probing commands. These retrieved profiles are compiled into a structured plan log (e.g., ports to expose, process names to surface, files/log snippets to materialize, and fields that must remain mutually consistent) and written back into the snapshot, ensuring stable and reproducible behavior across turns. At runtime, before generating each terminal output, the LLM reads both the vulnerability list and the plan log from the snapshot and injects them into the prompt as structured constraints, steering the LLM to respond under a consistent "vulnerable world state".

F Routing Model Fine-tuning

Dataset. The routing model is trained on a dataset constructed from two sources: (i) 3,978 individual commands extracted from approximately 1,000 multi-turn interaction sessions, where each command corresponds to a single turn, and (ii) 28,045 commands previously collected for training the response generation model. After merging the two sources, a series of data preprocessing steps, including filtering and deduplication, are applied, resulting in 23,501 command instances.

To further improve robustness against out-of-distribution and malformed inputs, we augment the training set with 2,361 externally collected adversarial commands. These commands are syntactically similar to valid Linux commands but fail under normal shell semantics (e.g., sl, ls-al), and the collection also includes prompt-injection-style inputs that attempt to elicit non-shell behaviors. The final training dataset, therefore, contains 25,849 command instances. All training instances consist solely of raw command strings paired with routing labels, without any auxiliary context, system prompts, or dialogue history.

Routing labels are assigned through a two-stage annotation process. Each command is first independently labeled by two large language models, namely GPT and Gemini, following the same annotation guidelines. For commands where the two model annotations agree, the label is accepted directly. In cases of disagreement, the command is manually reviewed and labeled by a human annotator. The validation and test sets are split directly from this dataset using standard proportion-based partitioning, with no additional data sources introduced. All splits preserve the original label distribution to ensure fair and consistent evaluation.

Model and Training Setup. We fine-tuned different BERT-style encoders, including lightweight distilled models (DistilBERT), widely adopted general-purpose models (RoBERTa), attention-enhanced architectures (DeBERTa), and modern efficiency-oriented designs (ModernBERT). All experiments are conducted on a single NVIDIA RTX 5090 GPU. The routing model is trained for four epochs using the AdamW optimizer with a learning rate of 3×10^{-5}. A cosine learning-rate scheduler with a warm-up ratio of 0.06 is applied,

and gradients are clipped to a maximum norm of 1.0. All runs use a fixed random seed of 42.

G Snapshot Update

In our implementation, snapshot updates follow a hybrid design: **rule-first, LLM-backup**, balancing determinism and coverage. Given a command c_t, the *Arbiter Agent* first performs lightweight parsing and routing. For common commands covered by our rule set (e.g., cd, mkdir, touch, rm, mv, cp, chmod, chown, sudo, and simple service/port-exposure commands), we apply a hand-written semantic updater and deterministically execute UpdateSnapshot(s_{t-1}, c_t). The updater edits the file tree only within controlled roots and synchronizes key metadata such as permissions, ownership, modification time, file size, and content hashes, making the snapshot self-consistent and reproducible. For these commands, the update is fully code-driven and does not rely on the LLM, avoiding hallucinated state changes.

For commands outside the rule coverage (e.g., pipelines, combined redirections, short script fragments, or rare utilities), we enable an LLM backup updater. The *Arbiter Agent* provides the pre-execution snapshot s_{t-1}, command c_t, and basic constraints, and prompts the LLM to propose the necessary state updates in a structured form (e.g., created/deleted files, modified permission fields, or newly opened ports). We then validate and sanitize the proposed updates (e.g., restrict writable paths and reject unsupported entities) before merging them into the snapshot to obtain s_t.

H Interactive Program Simulation

Beyond single-shot commands (e.g., ls, cat), real attackers frequently rely on highly interactive TUI programs (e.g., top, nano) that maintain a full-screen interface, continuously refresh outputs, and accept non-line-buffered keystrokes. These programs pose a challenge for LLM-based terminal emulation because they require (i) stateful screen rendering, (ii) low-latency, incremental updates, and (iii) precise control sequences (cursor movement, screen clear, key handling). We therefore implement top and nano as *built-in interactive primitives* that bypass the standard routing pipeline, while remaining *snapshot-grounded*: observable system state is read from a structured snapshot object, and any file modifications are committed back to the same snapshot (filesystem subtree), ensuring multi-turn consistency.

***top*: snapshot-grounded periodic frame loop.** Each top refresh constructs a top_state dictionary derived from snapshot. The process table is synthesized from observable snapshot artifacts (e.g., a baseline init process and one row per entry in snapshot["network"]["listening_ports"] using its (pid, process) fields, plus a session shell process for realism). The state also includes uptime/load averages and CPU/memory/swap summaries, with deterministic defaults when missing.

To simulate stable runtime evolution, we maintain a per-PID accumulator and update TIME across frames based on elapsed time and %CPU. The interactive loop redraws a full-screen frame every fixed interval (4 s), polls keystrokes in non-blocking mode, and exits on q or Ctrl-C, after which it restores terminal state (cursor visibility and channel timeouts). For presentation, each refresh exposes top_state via snapshot and calls the response generator to format a single frame; we validate the generated frame using lightweight structural checks (e.g., requiring a PID header). If validation fails or the LLM is unavailable, we fall back to a deterministic renderer that formats the same top_state. We additionally support batch mode (top -b with optional -n) by returning n frames as a standard line-based response rather than entering full-screen mode.

***nano*: editor with snapshot-grounded commit.** Upon nano <path>, we resolve paths against snapshot["cwd"], load initial contents from snapshot["filesystem"][dir]["file_contents"][file], and create an empty buffer if the file is absent. We implement a minimal full-screen editor with a header (filename and modified flag), a scrollable text viewport, a help line for supported shortcuts, and a message line for prompts/status.

The keystroke handler supports printable character insertion, newline, backspace (including line merges), arrow-key navigation, Ctrl+O (write-out), and Ctrl+X (exit with save confirmation when modified). On write-out, we commit the buffer back into the snapshot by updating file_contents and file_meta (e.g., mtime and size), ensuring edits become visible to subsequent commands. We include a privilege gate: writes under /etc/ require euid==0 as recorded in snapshot["identity"]; otherwise nano reports a canonical permission error.