TERMINAL-WORLD: SCALING TERMINAL AGENT ENVIRONMENTS VIA AGENT SKILLS

Anonymous authors
Paper under double-blind review

ABSTRACT

Terminal agents enable Large Language Models (LLMs) to flexibly compose command line actions, extending their action space to achieve superior generalizability. To train such agents, existing methods construct training data by first fixing one of the three required components (i.e., a task instruction, an executable environment, and an expert trajectory), such as tasks from seed keywords or environments from GitHub repositories, and then completing the rest around it. However, such pipelines synthesize the three components in **isolation**, making it difficult to maintain coherence among the three as data scales, limiting the effectiveness of data scaling for agent improvement. To mitigate this limitation, we introduce **Terminal-World**, a **unified** framework that uses agent skills as its core primitive to automatically synthesize **coherent** training data at scale. Specifically, each skill naturally encodes *what* to accomplish, *when* to apply it, and *how* to execute it, enabling the task, environment, and trajectory to be jointly derived from a single specification rather than assembled independently. With this framework, we construct the dataset Terminal-Atlas and train Terminal-Navigator-8B/14B/32B. Using the same teacher and base model with only **1.2%** of the training data, Terminal-Navigator-32B achieves 31.5 vs. 27.0 Pass@1 and 43.8 vs. 37.1 Pass@3 on Terminal-Bench 2.0. These results highlight Terminal-World's ability to efficiently synthesize high quality training data[1].

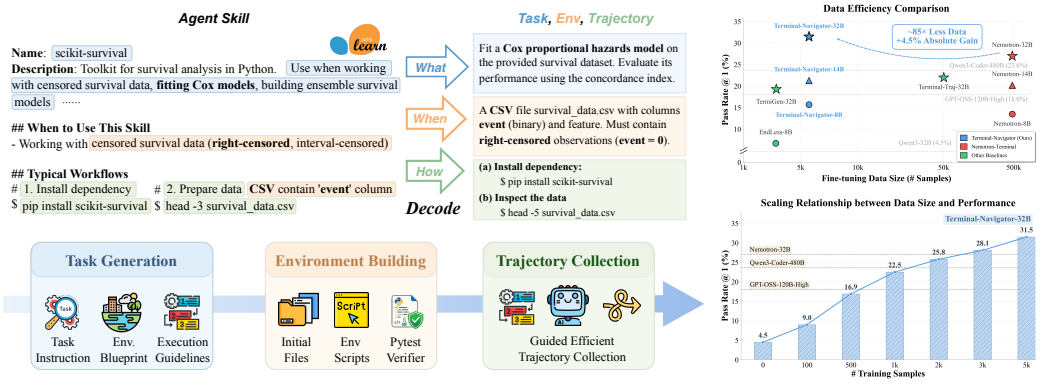

Figure 1: **Overview of Terminal-World (Left) and performance (Right).** Terminal-World uses agent skills as the synthesis primitive for terminal-agent data construction. Each skill encodes ***what*** the agent should accomplish, ***when*** the skill should be applied, and ***how*** the task should be executed. By decoding these three aspects, Terminal-World top-down synthesizes ***task instructions***, ***environments***, and ***trajectory*** in a unified process. Trained on Terminal-Atlas using only *1.2%* of the training samples used by Nemotron-Terminal, Terminal-Navigator-32B achieves a *4.5* absolute improvement in Pass@1 on Terminal-Bench 2.0.

[1]Our code and data will be publicly available. See the Reproducibility Statement for details.

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

Large Language Model (LLM)-based agents are increasingly moving from predefined API calls (Patil et al., 2024; Liu et al., 2024; 2025c; Jin et al., 2024; Dong et al., 2026) to direct terminal operation, with systems such as Claude Code (Liu et al., 2026) issuing shell commands inside real execution environments. This shift enables agents to flexibly compose command-line utilities rather than selecting from fixed tool schemas, achieving superior generalizability across diverse tasks (Meng et al., 2026; Bui, 2026; Liu et al., 2026). To train such agents, training data must tightly couple three components: a task instruction, an executable environment, and an expert trajectory, posing a fundamental challenge for scalable data construction (Pi et al., 2026; Zhu et al., 2026).

To tackle this challenge, existing methods construct data by fixing one of the three required components (e.g., defining tasks from human-defined seed keywords (Gandhi et al., 2026; Zhu et al., 2026; Pi et al., 2026) or deriving environments from GitHub repositories (Wu et al., 2026)) and then completing the rest around it independently. However, such pipelines synthesize the three components in **isolation**, making it difficult to maintain coherence among them as data scales, such that increased data volume yields diminishing returns in agent performance. Specifically, this isolated synthesis introduces limitations across all three components: **(1) Limited Tasks**: tasks are derived from a single anchor source, resulting in a constrained distribution that fails to capture the diverse requirements and complexity of real-world scenarios; **(2) Environment Misalignment**: tasks and environments are not jointly specified, so environments are retrofitted around tasks rather than co-designed with them, leading to mismatches between task semantics and execution context; **(3) Trajectory Inefficiency**: teacher models rely solely on their own capabilities to explore solutions for each task, producing trajectories with redundant steps and suboptimal solution paths.

To address these limitations, we identify **agent skills** (Xia et al., 2026; Lu et al., 2026) as a natural unified primitive for terminal-agent data synthesis. Specifically, agent skills are structured guidance packages, widely curated on open-source platforms such as ClawHub and SkillsMP (OpenClaw, 2026; SkillsMP, 2026), that encapsulate authentic terminal workflows distilled from real practice. As illustrated in Fig.1, each skill jointly encodes three aspects of an end-to-end terminal task: ① *what should be accomplished*, ② *when the skill should be applied* (i.e., the preconditions, inputs, and environmental state required for execution), and ③ *how it should be executed*. These three aspects naturally correspond to the task instruction, executable environment, and expert trajectory respectively, making agent skills a principled primitive for jointly deriving all three components from a single specification rather than assembling them independently.

Building on this primitive, we introduce **Terminal-World**, a **unified** framework that jointly instantiates each agent skill into a complete *task instruction–executable environment–expert trajectory*, ensuring that all three components are co-derived from a single specification rather than assembled in isolation. To further broaden the synthesis space, Terminal-World pairs skills with diverse user personas (Chan et al., 2024) to instantiate the same skill across varied user contexts, enabling richer task diversity without additional human annotation. With this framework, we construct **Terminal-Atlas** and train **Terminal-Navigator-8B/14B/32B**. Using the same teacher and base models with only **1.2%** of the training data, the Terminal-Navigator family achieves competitive average performance across six benchmarks compared with matched-scale terminal-agent baselines, with Terminal-Navigator-32B notably surpassing Nemotron-Terminal-32B on Terminal-Bench 2.0 by **+4.5** in Pass@1 (31.5) and **+6.7** in Pass@3 (43.8). Overall, our contributions are summarized as follows:

- We propose **Terminal-World**, which is an **unified** framework for constructing terminal-agent training data, which jointly derives the task, environment, and trajectory from a unified skill primitive, ensuring **coherence** among the three components as data scales.
- Building on this framework, we instantiate agent skills into terminal environments and construct **Terminal-Atlas**, which achieves both **diversity** and **quality** at low cost without human annotation.
- To assess the quality of data synthesized by Terminal-World, we train Terminal-Navigator-8B/14B/32B with the same teacher and base models as Nemotron-Terminal. With only **1.2%** of Nemotron-Terminal's training trajectories, our models achieve comparable average performance across six benchmarks, demonstrating the data efficiency of Terminal-World's unified synthesis.

Table 1: **Comparison of existing datasets.** *Primitive* identifies the synthesis source; *Human Free* means reusing existing data or public artifacts without new human-authored seeds or specifications. *Joint Spec.* means joint task–environment specification; *Open File Space* allows arbitrary file types. *Sandbox* icons: web search, Python, SQL, and Bash terminal. *Teacher Source*: *Self* explores unaided; *Guided* uses task-specific guidance during rollout. *# Traj.* counts collected trajectories; ✓, ✗, and ✔ indicate that a criterion is satisfied, not satisfied, or partially satisfied, respectively.

Dataset	Task		Environment			Trajectory	
	Primitive	Human Free	Joint Spec.	Open File Space	Sandbox	Teacher Source	# Traj.
Gorilla (Patil et al., 2024)	API specs	✗	–	✗	–	Self	16,450
ToolBridge (Jin et al., 2024)	API specs	✗	–	✗	Python	Self	178,023
APIGen (Liu et al., 2024)	API specs	✓	–	✗	Python	Self	60,000
WebExplorer (Liu et al., 2025b)	Web entities	✓	–	✗	web search	Self	13,000
ProgSearch (Pandit et al., 2025)	Web entities	✓	–	✗	web search + Python	Guided	5,500
Aseacher (Gao et al., 2025)	Web entities	✗	–	✗	web search	Self	35,000
ToolACE (Liu et al., 2025c)	API specs	✗	–	✗	–	Guided	11,300
ToolMind (Yang et al., 2025b)	API specs	✓	–	✗	–	Guided	111,941
InfTool (Li et al., 2025)	API specs	✓	–	✗	–	Self	4,965
DataMind (Qiao et al., 2025)	Data files	✔	✓	✗	Python + SQL	Guided	11,707
APIGen-MT (Prabhakar et al., 2025)	API specs	✓	–	✗	–	Guided	5,000
TaskCraft (Shi et al., 2025)	Seeds	✗	✓	✗	Python	Guided	36,000
GEM (Xu et al., 2026)	Raw text	✓	✓	✗	–	Guided	10,000
Endless Terminals (Gandhi et al., 2026)	Seeds	✗	✗	✔	Bash terminal	–	–
TermiGen (Zhu et al., 2026)	Seeds	✗	✗	✔	Bash terminal	Guided	3,291
Nemotron-Terminal (Pi et al., 2026)	Seeds	✗	✗	✔	Bash terminal	Self	490,520
Terminal-Atlas (Ours)	**Agent skills**	✓	✓	✓	Bash terminal	**Guided**	**5,723**

2 Related Work

Tool-Using Agents LLM-based agents interact with the external world through tool use, enabling them to execute actions beyond the limits of their parametric knowledge (Qu et al., 2025). To strengthen this capability, early efforts synthesized training data for API selection and argument filling (Patil et al., 2024; Liu et al., 2024). Subsequent work expanded API and tool coverage (Jin et al., 2024; Liu et al., 2025c), increased the interaction turns and orchestration complexity of tool use (Yang et al., 2025b; Prabhakar et al., 2025; Li et al., 2025), and broadened workflow sources by mining latent tool-use patterns from raw text (Xu et al., 2026). A parallel line studies web-search agents and synthesizes search trajectories over web content (Zhang et al., 2025; Tao et al., 2025; Liu et al., 2025b; Sun et al., 2025; Pandit et al., 2025; Gao et al., 2025; Wang et al., 2025). Despite this progress, these datasets typically operate over predefined toolsets, yielding a closed action space that limits generalizability to open-ended, real-world tasks. In contrast, Terminal-World operates in a bash terminal, where agents can flexibly compose commands to achieve superior generalizability.

Terminal-Based Agents The rise of CLI-based coding agents such as Claude Code (Liu et al., 2026) has shifted agent interaction toward direct operation in terminal environments, motivating recent efforts to synthesize terminal-agent training data. Endless Terminals (Gandhi et al., 2026), TermiGen (Zhu et al., 2026), and Nemotron-Terminal (Pi et al., 2026) start from human-defined task seeds and construct the environment and trajectories around them, while TerminalTraj (Wu et al., 2026) starts from GitHub repositories and infers tasks from existing environments. However, such pipelines anchor synthesis on one component and construct the others around it, making it difficult to maintain coherence among tasks, environments, and trajectories as data scales. SkillSynth (Fan et al., 2026) uses skills to scale task diversity, sampling paths through a scenario-mediated skill graph to vary scenario and skill combinations. In contrast, Terminal-World treats each skill as a unified *what–when–how* specification that grounds task generation, environment construction, and execution guidance. Furthermore, it pairs each skill with user personas to diversify the application contexts of the same capability. A detailed comparison is provided in Appendix A.2.

3 Terminal-World

Terminal-World uses agent skills to jointly synthesize tasks, environments, and trajectories in four stages (Figure 2): (a) **Agent Skill Collection** (§3.1) constructs a terminal capability space from real-world agent skills; (b) **Task Generation** (§3.2) pairs skills with personas to synthesize task

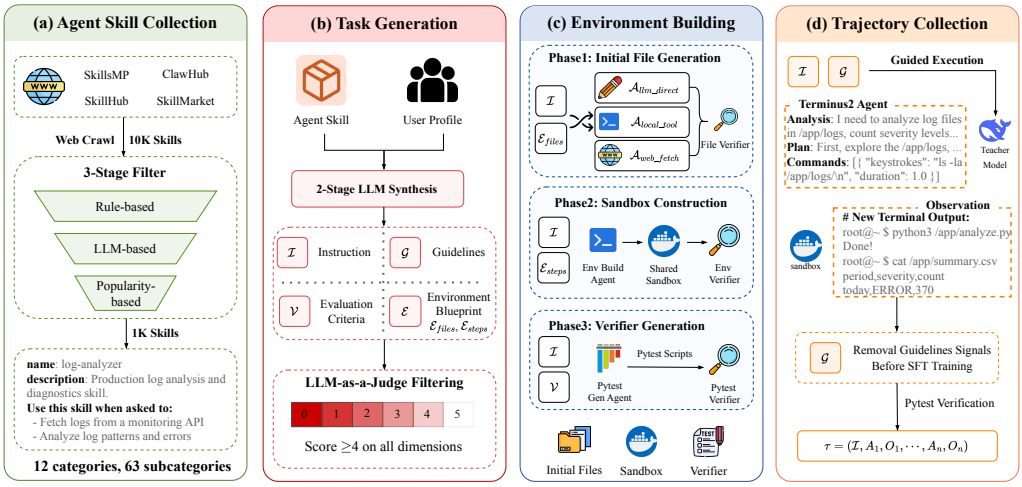

Figure 2: **Overview of Terminal-World.** We start from real-world agent skills, filter high-quality candidates via rule-based, LLM-based, and popularity-based screening, pair each skill with a user persona to synthesize diverse and verifiable task quadruples $(\mathcal{I}, \mathcal{E}, \mathcal{V}, \mathcal{G})$, construct executable sandboxes through iterative generate-verify-repair cycles, and collect efficient trajectories.

instructions, environment blueprints, evaluation criteria, and execution guidelines; (c) **Environment Building** (§3.3) instantiates each blueprint into an executable sandbox with initial files, setup scripts, and pytest verifiers; and (d) **Trajectory Collection** (§3.4) collects guided teacher trajectories.

3.1 Agent Skill Collection

To construct a diverse and high-quality terminal capability space, we collect **10,000** agent skills from ClawHub and SkillMP, and apply a three-stage filter. **(1) Rule-based** filtering removes skills with terminal-irrelevant names (e.g., "skill creator"), leaving 8,520 skills. **(2) LLM-based** filtering scores each skill on terminal applicability and content richness (1–3 each), retaining only those with the maximum score on both, yielding 3,025 skills. **(3) Popularity-based** filtering ranks the remaining skills by download count and selects the top **1,000**, spanning 12 categories and 63 subcategories. These **1,000** filtered skills $\mathcal{S}$ serve as the synthesis primitives for the remainder of the pipeline.

3.2 Task Generation

Given the synthesis primitives $\mathcal{S}$ from Sec. 3.1, this stage decodes each skill's what/when/how encoding into a unified task specification: the task instruction $\mathcal{I}$ (what), the environment blueprint $\mathcal{E}$ (when), and the execution guideline $\mathcal{G}$ (how). To diversify usage scenarios, we pair each primitive with a user persona $\mathcal{U}$ sampled from FinePersonas (Lozhkov et al., 2024; Chan et al., 2024), short profiles describing users' backgrounds and roles. The LLM instantiates a task only when the sampled persona forms a coherent scenario for the given primitive. For each valid pair, we synthesize a quadruple $(\mathcal{I}, \mathcal{E}, \mathcal{V}, \mathcal{G})$:

$$\mathcal{I}, \mathcal{E}, \mathcal{V} = \mathrm{LLM}(\mathcal{P}_s, \mathcal{S}, \mathcal{U}), \ \mathcal{G} = \mathrm{LLM}(\mathcal{P}_g, \mathcal{I}, \mathcal{S}),$$

where $\mathcal{P}_s$ and $\mathcal{P}_g$ are the prompt templates for task synthesis (Appendix D.1) and guideline generation (Appendix D.2). Here, $\mathcal{I}$ is the task instruction; $\mathcal{E}$ is the environment blueprint consisting of initial files $\mathcal{E}_{\text{files}}$ and setup steps $\mathcal{E}_{\text{steps}}$; $\mathcal{V}$ specifies evaluation criteria later used to generate pytest verifiers; and $\mathcal{G}$ provides skill-derived execution guidance for trajectory collection.

An LLM-as-a-Judge filter (Appendix D.3; example in Appendix E.1) scores ① instruction quality, ② closed-world solvability, ③ blueprint completeness, ④ guideline quality, and ⑤ evaluation-criteria quality; we retain samples scoring at least 4 in all five dimensions.

Table 2: **Key statistics of Terminal-Atlas** across task, environment, and trajectory levels: 5,723 tasks from 1,000 skills, spanning 104 file types and 1,939 Bash commands.

Statistic	Value
Task	
# Skills	1,000
# Persona	4,973
# Total Tasks	5,723
Environment	
# File Types	104
Avg. `pytest` Tests	4.27
Trajectory	
# Bash Cmd.	1,939

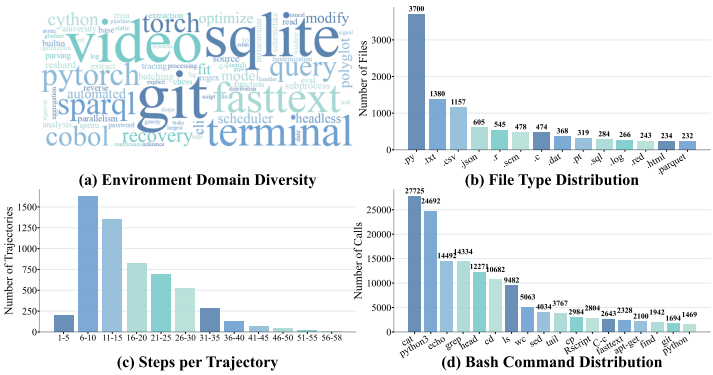

Figure 3: **Distributions of Terminal-Atlas environments** and trajectories. Panels (a–d) show domain terms, file types, steps per trajectory, and Bash commands, respectively. Most trajectories have 6–15 steps, and Python files are the most frequent file type.

3.3 Environment Building

Given the task specification $(\mathcal{I}, \mathcal{E}, \mathcal{V}, \mathcal{G})$, this stage instantiates the environment blueprint into an executable and verifiable sandbox. Each sandbox comprises three artifacts: initial files $\mathcal{F}$ that define the starting workspace, a setup script $\mathcal{B}_{\text{env}}$ that prepares runtime dependencies and services, and a pytest verifier $\mathcal{T}_{\text{test}}$ for automatic completion checking. All three are constructed through a generate-verify-repair (**GVR**) loop:

$$x^{(0)} = \text{Gen}(\cdot),\ x^{(t+1)} = \text{Rep}(x^{(t)}, \text{Ver}(x^{(t)}))$$

where x denotes the artifact and tasks that cannot be repaired within $T = 3$ iterations are discarded.

(1) Initial Files. To support arbitrary file types, we adopt a multi-agent architecture that routes each file in $\mathcal{E}_{\text{files}}$ to a dedicated agent based on its generation mode (annotated during task generation): an LLM-synthesis agent $\mathcal{A}_{llm_direct}$, a local-tool agent $\mathcal{A}_{local_tool}$ equipped with file-creation tools, or a remote-fetch agent $\mathcal{A}_{remote_fetch}$ with search tools, each conditioned on $\mathcal{I}$ and the file description (Appendices D.4–D.6). A verification agent (Appendix D.7) then checks ① *internal correctness* (well-formedness and description alignment) and ② *external consistency* (cross-file paths, references, and schemas), with failures triggering joint repair across dependent files.

(2) Setup Scripts. For scalability, we use a shared general-purpose sandbox with task-specific setup scripts rather than per-task Docker images. An environment-building agent converts the natural-language steps $\mathcal{E}_{\text{steps}}$ into executable shell commands (Appendix D.8). Beyond exit codes, an environment-verification agent generates diagnostic scripts to check required packages, services, and runtime states (Appendix D.9), returning detected issues for repair.

(3) pytest Verifiers. A verifier-generation agent translates the evaluation criteria $\mathcal{V}$ into pytest scripts conditioned on $\mathcal{I}$, $\mathcal{V}$, $\mathcal{F}$, and $\mathcal{B}_{\text{env}}$ (Appendix D.11; example in Appendix E.2). A validation agent checks two properties: ① *executability* (all tests run without syntax or import errors) and ② *reliability* (all tests fail on the pre-execution state, ruling out vacuous passes).

3.4 Trajectory Collection

This stage collects efficient teacher trajectories by providing the execution guideline $\mathcal{G}$ as skill-derived guidance, rather than letting the teacher explore freely. Following Pi et al. (2026) for fair comparison, we adopt DeepSeek-V3.2 (Liu et al., 2025a) with the Terminus2 scaffolding (Harbor Framework Team, 2026) as the teacher, isolating the contribution of our data pipeline from teacher capability differences. At each step, the teacher produces an action $\mathcal{A}_t = \pi_{\text{teacher}}(\mathcal{I}, \mathcal{G}, \mathcal{H}_{t-1})$ conditioned on the task instruction, guideline, and prior interaction history $\mathcal{H}_{t-1}$, and receives an observation $\mathcal{O}_t$ from the sandbox. After rollout, we run the verifier $\mathcal{T}_{\text{test}}$ to annotate each trajectory,

Table 3: **Main results on 6 benchmarks.** We report mean accuracy (%) ± standard error over three runs, including pass@1 and pass@3. In the *Open-source Models (<100B)* block, the best result is shown in *bold*, and the second-best result is underlined.

Model	# Training Samples	TB 2.0		AIME24		AIME25		DABench		TableBench		BIRD		Avg.	
		P@1	P@3	P@1	P@3	P@1	P@3	P@1	P@3	P@1	P@3	P@1	P@3	P@1	P@3
Frontier Proprietary Models															
GPT-5.2	–	53.9±3.1	74.2±2.7	–	–	–	–	–	–	–	–	–	–	–	–
Gemini-3-Flash	–	51.7±3.1	66.3±2.9	93.3±2.6	100.0±0.0	90.0±3.2	96.7±1.9	87.0±0.9	92.0±0.7	73.5±1.1	77.5±1.1	49.5±2.0	58.0±2.0	74.2±1.0	81.8±0.7
Gemini-3.1-Pro-Preview	–	68.5±2.8	80.9±2.4	96.7±1.9	100.0±0.0	100.0±0.0	100.0±0.0	88.0±0.8	92.5±0.7	77.0±1.1	79.0±1.1	59.0±2.0	62.5±2.0	81.5±0.7	85.8±0.6
Claude-Sonnet-4.5	–	42.7±3.0	53.9±3.1	–	–	–	–	–	–	–	–	–	–	–	–
Open-source Models (>100B)															
GPT-OSS-120B (High)	–	13.5±2.1	27.0±2.7	90.0±3.2	96.7±1.9	86.7±3.6	96.7±1.9	75.0±1.1	90.0±0.8	66.5±1.2	79.0±1.1	50.0±2.0	59.5±2.0	63.6±1.0	74.8±0.7
MiniMax-M2.7	–	56.2±3.0	65.2±2.9	50.0±5.3	83.3±3.9	63.3±5.1	96.7±1.9	85.5±0.9	90.5±0.8	76.0±1.1	83.0±1.0	49.5±2.0	58.5±2.0	63.4±1.4	79.5±1.0
Qwen3-Coder-480B	–	23.6±2.6	39.3±3.0	70.0±4.8	73.3±4.7	46.7±5.3	66.7±5.0	81.0±1.0	88.5±0.8	67.5±1.2	75.5±1.1	49.5±2.0	62.0±2.0	56.4±1.3	67.6±1.3
DeepSeek-V3.2	–	38.2±3.0	52.8±3.1	83.3±3.9	96.7±1.9	93.3±2.6	96.7±1.9	87.0±0.9	92.5±0.7	72.5±1.2	79.5±1.0	50.0±2.0	56.5±2.0	70.7±1.0	79.1±0.8
GLM-5	–	56.2±3.0	65.2±2.9	86.7±3.6	96.7±1.9	83.3±3.9	93.3±2.6	88.0±0.8	91.5±0.7	76.5±1.1	82.5±1.0	56.0±2.0	60.5±2.0	74.5±1.1	81.6±0.8
Kimi-K2-Thinking	–	36.0±2.9	49.4±3.1	76.7±4.5	83.3±3.9	70.0±4.8	86.7±3.6	85.0±0.9	91.5±0.7	75.0±1.1	82.0±1.0	52.5±2.0	58.5±2.0	65.9±1.3	75.2±1.1
Open-source Models (<100B)															
Qwen3-8B	–	2.2±0.9	3.4±1.1	63.3±5.1	76.7±4.5	53.3±5.3	63.3±5.1	16.5±1.0	32.5±1.2	24.5±1.1	39.0±1.3	4.0±0.8	9.0±1.2	27.3±1.3	37.3±1.2
Qwen3-14B	–	4.5±1.3	7.9±1.7	76.7±4.5	86.7±3.6	53.3±5.3	66.7±5.0	38.0±1.3	69.0±1.2	35.5±1.2	60.0±1.3	7.5±1.1	18.5±1.6	35.9±1.2	51.5±1.1
Qwen3-32B	–	4.5±1.3	9.0±1.8	60.0±5.2	86.7±3.6	43.3±5.2	66.7±5.0	44.5±1.3	75.0±1.1	26.5±1.1	53.5±1.3	6.5±1.0	16.0±1.5	30.9±1.3	51.2±1.1
GPT-OSS-20B (High)	–	3.4±1.1	9.0±1.8	**93.3**±2.6	96.7±1.9	83.3±3.9	**93.3**±2.6	59.5±1.3	82.5±1.0	38.5±1.3	64.0±1.2	0.0±0.0	6.0±1.0	46.3±0.9	58.6±0.7
EndLess Terminal-8B	3.3k	6.7±1.5	12.4±2.0	23.3±4.5	36.7±5.1	30.0±4.8	43.3±5.2	26.0±1.1	38.5±1.3	28.0±1.2	42.0±1.3	6.0±1.0	11.5±1.3	20.0±1.2	30.7±1.3
TermiGen-32B	3.3k	19.1±2.4	27.0±2.7	46.7±5.3	56.7±5.2	33.3±5.0	43.3±5.2	81.0±1.0	91.5±0.7	68.0±1.2	**79.0**±1.1	50.5±2.0	**62.5**±2.0	49.8±1.3	60.0±1.4
TerminalTraj-32B	50.7k	22.0±2.5	27.0±2.7	46.7±5.3	60.0±5.2	26.7±4.7	46.7±5.3	76.0±1.1	88.0±0.8	62.0±1.3	76.0±1.1	49.5±2.0	61.5±2.0	47.2±1.3	59.9±1.4
Nemotron-Terminal-8B	490.5k	13.5±2.1	21.3±2.5	83.3±3.9	83.3±3.9	63.3±5.1	90.0±3.2	80.5±1.0	92.0±0.7	66.5±1.2	73.5±1.1	44.0±2.0	55.5±2.0	58.5±1.2	69.3±1.0
Nemotron-Terminal-14B	490.5k	20.2±2.5	24.7±2.6	90.0±3.2	96.7±1.9	**90.0**±3.2	**93.3**±2.6	80.5±1.0	90.5±0.8	70.0±1.2	75.0±1.1	50.5±2.0	59.0±2.0	66.9±1.0	73.2±0.8
Nemotron-Terminal-32B	490.5k	27.0±2.7	37.1±3.0	**93.3**±2.6	**100.0**±0.0	86.7±3.6	**93.3**±2.6	81.5±1.0	**93.0**±0.7	**72.5**±1.2	75.5±1.1	**52.5**±2.0	57.0±2.0	68.9±1.0	76.0±0.8
Ours															
Terminal-Navigator-8B	5.7k	15.7±2.2	23.6±2.6	86.7±3.6	93.3±2.6	80.0±4.2	90.0±3.2	80.5±1.0	91.0±0.7	69.5±1.2	74.5±1.1	48.5±2.0	58.0±2.0	63.5±1.1	71.7±0.9
Terminal-Navigator-14B	5.7k	21.3±2.5	27.0±2.7	90.0±3.2	96.7±1.9	83.3±3.9	**93.3**±2.6	81.0±1.0	92.0±0.7	70.5±1.2	76.0±1.1	50.0±2.0	59.5±2.0	66.0±1.0	74.1±0.8
Terminal-Navigator-32B	5.7k	**31.5**±2.8	**43.8**±3.0	**93.3**±2.6	96.7±1.9	86.7±3.6	**93.3**±2.6	**83.5**±1.0	**93.0**±0.7	71.5±1.2	**79.0**±1.1	49.5±2.0	61.5±2.0	**69.3**±1.0	**77.9**±0.8

retaining both successful and failed ones (Wu et al., 2026). We then apply two post-processing steps: (1) following Pi et al. (2026), we decontaminate the collected data by removing any trajectory whose prompt has a 14-gram overlap with Terminal-Bench 2.0 test samples; (2) we remove the execution guideline $\mathcal{G}$ from the input so that the student learns from the verified interaction itself rather than procedural hints. A representative example is provided in Appendix E.3.

3.5 Data Statistics

Table 2 and Figure 3 summarize Terminal-Atlas. At the task level, 1,000 skills yield **5,723** diverse terminal-agent tasks. At the environment level, each task contains an executable sandbox with an average of 2.25 initial files and 4.27 pytest tests. These environments span 104 file types, including both textual formats (e.g., JSON) and non-textual formats (e.g., SQL and Parquet), reflecting the diversity of realistic terminal workspaces. At the trajectory level, verified demonstrations average 14.75 steps and cover 1,939 distinct Bash commands.

4 Experiments

4.1 Experiment Setting

Baselines We compare Terminal-Navigator against three model groups (Zhu et al., 2026). **Frontier Proprietary Models**: GPT-5.2, Gemini-3-Flash, Gemini-3.1-Pro-Preview, and Claude-Sonnet-4.5. **Open-source Models (>100B)**: GPT-OSS-120B (Agarwal et al., 2025), MiniMax-M2.7, Qwen3-Coder-480B (Yang et al., 2025a), DeepSeek-V3.2 (Liu et al., 2025a), GLM-5 (Zeng et al., 2026), and Kimi-K2-Thinking (Team et al., 2026). **Open-source Models (<100B)**: Qwen3-8B/14B/32B (Yang et al., 2025a), GPT-OSS-20B, EndLess Terminal-8B (Gandhi et al., 2026), TermiGen-32B (Zhu et al., 2026), TerminalTraj-32B (Wu et al., 2026), and Nemotron-Terminal-8B/14B/32B (Pi et al., 2026). All models use the Terminus2 Agent scaffold (Harbor Framework Team, 2026) except EndLess Terminal-8B and TermiGen-32B, which use native formats.

Benchmarks Following prior research Shi et al. (2025); Pi et al. (2026); Zhu et al. (2026), we evaluate models on six benchmarks (Appendix B.1). **Terminal-Bench 2.0** (Merrill et al., 2026; Zeng

et al., 2026) evaluates terminal-agentic coding ability. The remaining benchmarks are converted into terminal-based tasks (Pi et al., 2026): **AIME24/25** (Zhang & Math-AI, 2024; 2025) evaluate mathematical reasoning, **DABench** (Hu et al., 2024) and **TableBench** (Wu et al., 2025) evaluate table-based data analysis, and **BIRD** (Li et al., 2023) evaluates SQL-based data analysis.

Implementation Details Following prior work (Pi et al., 2026), we conduct SFT using Swift (Zhao et al., 2024). Specifically, we set the learning rate to $2e{-}5$, with a warmup ratio of 0.1, weight decay of $1e{-}4$, and train for 2 epochs with sequence length of $32{,}768$. During inference, we adopt a context length of $40{,}960$ tokens, temperature 0.6, top-p $=$ 0.95, top-k $=$ 20, and min-p $=$ 0.0. Experiments use four nodes with eight H20-141GB GPUs each. For all other baselines, we follow the officially recommended sampling parameters (Appendices B.3 and B.4).

4.2 Main Results

Table 3 presents the main results across six benchmarks. We summarize the key findings as follows:

Terminal-Navigator delivers strong performance with significantly higher sample efficiency. At all three model sizes, the Terminal-Navigator family achieves strong overall performance and is particularly strong on Terminal-Bench 2.0. Terminal-Navigator-32B achieves **69.3±1.0** Avg. Pass@1 and **77.9±0.8** Avg. Pass@3. On Terminal-Bench 2.0, Terminal-Navigator-32B achieves **31.5±2.8** Pass@1 and **43.8±3.0** Pass@3, surpassing all <100B open-source models. Notably, these results are obtained by training on only **5.7K** trajectories from Terminal-Atlas (**1.2%** of the 490.5K used by Nemotron-Terminal), demonstrating that jointly deriving coherent task-environment-trajectory triples from a unified primitive can achieve superior performance at a fraction of the data scale.

Skill-grounded synthesis achieves broad domain coverage without targeted data engineering. Unlike Nemotron-Terminal, which supplements its terminal data with 226.3K domain-specific samples to ensure coverage, the Terminal-Navigator family is trained exclusively on the 5.7K trajectories in Terminal-Atlas, which are derived from general-purpose skills without any domain-targeted selection. Despite this, Terminal-Navigator-32B outperforms Nemotron-Terminal-32B on average across all six benchmarks, achieving 69.3±1.0 vs. 68.9±1.0 Avg. Pass@1 and 77.9±0.8 vs. 76.0±0.8 Avg. Pass@3. This suggests that the skill ecosystem spans diverse domains, making broad task coverage an emergent property of the unified primitive without explicit domain engineering.

Higher-capacity models benefit more from coherent skill-grounded data. On Terminal-Bench 2.0, Terminal-Navigator-32B achieves the largest absolute margin: **31.5±2.8** vs. 27.0±2.7 Pass@1 (**+4.5**) and **43.8±3.0** vs. 37.1±3.0 Pass@3 (**+6.7**). The larger gain at 32B suggests that higher-capacity models can better leverage supervision from skill-grounded trajectories.

5 Analysis

In this section, we conduct a comprehensive analysis to answer the following research questions: **RQ1:** *Do models in the Terminal-Navigator family exhibit efficient and reliable terminal execution behaviors?* (§5.1; Appendix C.1) **RQ2:** *Which SFT data construction choices are most important?* (§5.2; Appendix C.2) **RQ3:** *Can the verifiable environments in Terminal-Atlas support effective reinforcement learning?* (§5.3) **RQ4:** *Can Terminal-World synthesize diverse, high-quality, and cost-effective terminal-agent training data?* (§5.4)

5.1 RQ1: Behavior Analysis

Beyond accuracy reported in Table 3, we further examine the task-execution behavior of the Terminal-Navigator family. Specifically, we analyze four behavioral metrics: the average number of steps (**Avg Steps**), the command failure rate (**Command Error Rate**), and the average commands per step (**Commands / Step**), and the average total commands per trajectory (**Avg Total Commands**). To avoid confounding effects caused by differences in task difficulty, we conduct the analysis on the intersection of tasks correctly solved by both Terminal-Navigator-32B and Nemotron-Terminal-32B across three independent runs on Terminal-Bench 2.0. The results are shown in Figure 4. Terminal-Navigator-32B requires only 10.2 steps and 40.3 commands on average to complete

Table 4: **SFT ablations on Terminal-Navigator-8B/14B.** Δ_{rel} is the relative drop (%) from full strategy; darker blue highlights larger losses from removing or suppressing failure trajectories.

SFT Strategy	#Data	*Terminal-Navigator-8B*				*Terminal-Navigator-14B*			
		P@1	Δ_{rel} (%)	P@3	Δ_{rel} (%)	P@1	Δ_{rel} (%)	P@3	Δ_{rel} (%)
Full strategy	**5.7k**	**15.7**	–	**23.6**	–	**21.3**	–	**27.0**	–
w/o data scale	2.3k	12.4	↓ 21.0	18.0	↓ 23.7	18.0	↓ 15.5	22.5	↓ 16.7
w/o guideline removal	5.7k	13.5	↓ 14.0	20.2	↓ 14.4	19.1	↓ 10.3	24.7	↓ 8.5
w/o failure traj.	2.3k	10.1	↓ 35.7	14.6	↓ 38.1	15.7	↓ 26.3	20.2	↓ 25.2
w/ failure suppression	5.7k	9.0	↓ 42.7	13.5	↓ 42.8	14.6	↓ 31.5	19.1	↓ 29.3

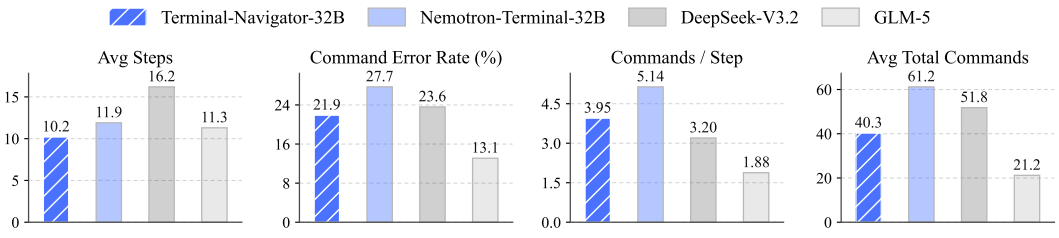

Figure 4: **On jointly solved Terminal-Bench 2.0 tasks, Terminal-Navigator-32B uses fewer steps and total commands, with fewer errors, than Nemotron-Terminal-32B and DeepSeek-V3.2.**

each task, while also exhibiting a lower command failure rate. Notably, when evaluated on the same task subset, even without guideline assistance, Terminal-Navigator-32B exhibits more efficient execution behavior than its teacher model, DeepSeek-V3.2. This suggests that the skill-guided trajectories in Terminal-Atlas teach the model more concise and goal-directed execution strategies.

5.2 RQ2: Impact of SFT Data Strategies

Building on the SFT setup in Sec.4, we investigate four SFT data strategies (Appendix C.2): retaining guidelines (w/o guideline removal), reducing the dataset (w/o full data scale), training on successful trajectories only (w/o failure trajectories), and applying negative SFT loss only to unsuccessful trajectories (w/ failure-trajectory suppression). As shown in Tab.4, retaining guidelines substantially degrades performance, suggesting that prescriptive instructions discourage autonomous planning. Reducing data size also leads to a clear drop, confirming that coherent data scales effectively, unlike isolated synthesis where increased volume yields diminishing returns. Removing failure trajectories causes a larger decline, suggesting they contain richer error-correction processes; negative SFT loss degrades performance further by inevitably suppressing many correct intermediate steps.

Table 5: **SFT+GRPO achieves the best Pass@1/3 on Terminal-Bench 2.0.**

Training Strategy	P@1	P@3
Qwen3-8B		
Base	2.2	3.4
SFT	15.7 (+13.5)	23.6 (+20.2)
GRPO	3.4 (+1.2)	6.7 (+3.3)
SFT + GRPO	**18.0 (+15.8)**	**27.0 (+23.6)**
Qwen3-14B		
Base	4.5	7.9
SFT	21.3 (+16.8)	27.0 (+19.1)
GRPO	5.6 (+1.1)	10.1 (+2.2)
SFT + GRPO	**23.6 (+19.1)**	**30.3 (+22.4)**

5.3 RQ3: Complementarity of SFT and RL

We investigate whether the verifiable environments synthesized by Terminal-World can support GRPO training on Qwen3-8B and Qwen3-14B. We compare Base, SFT, GRPO, and SFT+GRPO at both scales. As shown in Table 5, direct GRPO from the base models yields only modest gains due to their limited initial task-solving ability. In contrast, after cold-starting with our synthesized expert trajectories, applying GRPO to Terminal-Navigator-8B/14B further improves Pass@1 by 2.3 at both scales and Pass@3 by 3.4 and 3.3, respectively. These results demonstrate the verifiable environments in Terminal-Atlas can support effective RL training, while expert-trajectory SFT is crucial for overcoming the cold-start bottleneck and unlocking its full benefit.

Table 6: **Quality of 500 environments per dataset, scored 1–3 by three judges.**

Dataset	Term. Nat.	Env.–Task Cons.	Env. Qual.	Verifier Rob.
Endless Terminals	2.32	2.91	**2.88**	**2.92**
TermiGen	2.00	2.59	2.54	2.33
TerminalTraj	2.08	1.09	1.03	1.93
Terminal-Atlas	**2.69**	**2.97**	**2.88**	2.89

Table 7: **API cost: 6,884 accepted tasks yield 5,723 trajectories for $999.59 ($0.17 each).**

Stage	Cost ($)	Output	Avg. Cost ($)
Task Generation	101.71	6,884 tasks	0.015
Env Building	476.61	5,723 envs	0.083
Trajectory	421.27	5,723 traj.	0.074
Total	**999.59**	**5,723 traj.**	**0.17**

5.4 RQ4: Data Diversity, Quality, and Cost

We assess task diversity by independently generating 250 tasks in each of three settings: 50 skills repeated five times without personas, 50 skills with 5 personas each, and 25 skills with 10 personas each. We extract scenario descriptions from each instruction and apply clustering-based deduplication to identify unique scenario clusters (Fan et al., 2026). As shown in Figure 5, persona grounding substantially improves scenario coverage. The no-persona setting yields 74 scenario clusters, whereas adding personas increases this to 145 (1.96×). Doubling personas raises coverage to 153 clusters (2.07×), confirming that their varied contexts can compensate for a smaller skill set.

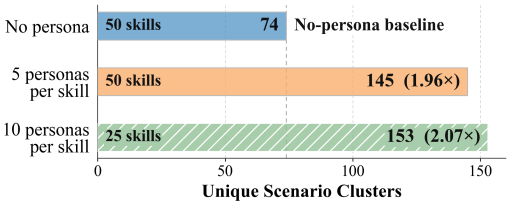

Figure 5: **With 250 tasks per setting, personas raise unique clusters from 74 to 145–153.**

We next assess data quality on 500 randomly sampled environments from each of four terminal-agent datasets. Three independent judges inspect each anonymized environment along four 1–3 dimensions: terminal nativeness, environment–task consistency, environment quality, and verifier robustness. Following Liu et al. (2023), as a human validation check, one annotator independently evaluated the same 500 environments using the same rubric, and the resulting overall scores showed a very strong Spearman correlation of $\rho = 0.87$ with the three-judge averages. As shown in Table 6, Terminal-Atlas achieves the highest environment–task consistency (2.97), directly validating that the unified skill primitive produces more coherent task–environment pairs. It also leads in terminal nativeness, ties for the highest environment quality, and is within 0.03 points of the best verifier score. Appendix C.3 further reports a complementary difficulty analysis: DeepSeek-V3.2 achieves 39.8% Pass@1 on Terminal-Atlas, close to its 38.2% Pass@1 on Terminal-Bench 2.0.

Finally, we measure the yield and API cost of producing these environments. Of 6,884 accepted tasks, 5,723 yield valid environments, an 83.1% environment-construction yield. Environment building, including verification and repair, is the largest expense at $476.61, or $0.083 per valid environment. Task generation and trajectory collection cost $101.71 and $421.27, respectively. Across all three stages, the pipeline costs $999.59, or $0.17 per collected trajectory (Table 7). The 83.1% yield and per-trajectory cost show that Terminal-World can produce thousands of valid terminal environments with modest API spending.

6 Conclusion

We introduce **Terminal-World**, which jointly derives task instructions, executable environments, and expert trajectories from one agent skill to produce coherent, diverse, high-quality data at scale without human annotation and at low cost. We construct **Terminal-Atlas** and train Terminal-Navigator-8B/14B/32B. Using the same teacher and base models with only **1.2%** of Nemotron-Terminal's training data, the Terminal-Navigator family achieves competitive average performance across six benchmarks compared with matched-scale baselines. Notably, Terminal-Navigator-32B surpasses Nemotron-Terminal-32B on Terminal-Bench 2.0 by +4.5 in Pass@1 and +6.7 in Pass@3. We hope Terminal-World encourages future research to treat real-world agent skills as a shared primitive for scaling coherent training environments.

AI USE STATEMENT

We used Codex to assist with parts of the implementation of the data synthesis framework and ChatGPT to refine the manuscript's wording. DeepSeek and Gemini models were used to generate and filter synthetic data in Terminal-World. LLM-based judges were also used in the evaluations described in Appendix C.6. The research ideas and hypotheses were developed independently by the authors, with AI-assisted literature searches conducted afterward to identify related work. The authors reviewed and revised all manuscript text, conducted independent literature searches, and manually reviewed the core code. We take responsibility for the final content of this work, including text, claims, code, and evaluation results produced with AI assistance.

REPRODUCIBILITY STATEMENT

An anonymized implementation of Terminal-World is available at https://anonymous.4open.science/r/terminal-world-supplement-FE29. The method and data synthesis pipeline are described in Section 3 and Appendix A. Training and evaluation settings are provided in Section 4 and Appendix B; prompt templates and illustrative examples are provided in Appendices D and E.

REFERENCES

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

Nghi DQ Bui. Building effective ai coding agents for the terminal: Scaffolding, harness, context engineering, and lessons learned. *arXiv preprint arXiv:2603.05344*, 2026.

Xin Chan, Xiaoyang Wang, Dian Yu, Haitao Mi, and Dong Yu. Scaling synthetic data creation with 1,000,000,000 personas. *arXiv preprint arXiv:2406.20094*, 2024.

Guanting Dong, Junting Lu, Junjie Huang, Wanjun Zhong, Longxiang Liu, Shijue Huang, Zhenyu Li, Yang Zhao, Xiaoshuai Song, Xiaoxi Li, Jiajie Jin, Yutao Zhu, Hanbin Wang, Fangyu Lei, Qinyu Luo, Mingyang Chen, Zehui Chen, Jiazhan Feng, Ji-Rong Wen, and Zhicheng Dou. Agent-world: Scaling real-world environment synthesis for evolving general agent intelligence, 2026. URL https://arxiv.org/abs/2604.18292.

Zhiyuan Fan, Tinghao Yu, Yuanjun Cai, Jiangtao Guan, Yun Yang, Dingxin Hu, Jiang Zhou, Xing Wu, Zhuo Han, Feng Zhang, and Lilin Wang. Toward scalable terminal task synthesis via skill graphs, 2026. URL https://arxiv.org/abs/2604.25727.

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

Jiaxuan Gao, Wei Fu, Minyang Xie, Shusheng Xu, Chuyi He, Zhiyu Mei, Banghua Zhu, and Yi Wu. Beyond ten turns: Unlocking long-horizon agentic search with large-scale asynchronous RL. In *First Workshop on Multi-Turn Interactions in Large Language Models*, 2025. URL https://openreview.net/forum?id=JSYCLJcniH.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments, January 2026. URL https://github.com/harbor-framework/harbor.

Xueyu Hu, Ziyu Zhao, Shuang Wei, Ziwei Chai, Qianli Ma, Guoyin Wang, Xuwu Wang, Jing Su, Jingjing Xu, Ming Zhu, et al. Infiagent-dabench: Evaluating agents on data analysis tasks. *arXiv preprint arXiv:2401.05507*, 2024.

Zhenchao Jin, Mengchen Liu, Dongdong Chen, Lingting Zhu, Yunsheng Li, and Lequan Yu. Toolbridge: An open-source dataset to equip llms with external tool capabilities. *arXiv preprint arXiv:2410.10872*, 2024.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36:42330–42357, 2023.

Yuwen Li, Wei Zhang, Zelong Huang, Mason Yang, Jiajun Wu, Shawn Guo, Huahao Hu, Lingyi Sun, Jian Yang, Mingjie Tang, et al. Close the loop: Synthesizing infinite tool-use data via multi-agent role-playing. *arXiv preprint arXiv:2512.23611*, 2025.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025a.

Jiacheng Liu, Xiaohan Zhao, Xinyi Shang, and Zhiqiang Shen. Dive into claude code: The design space of today's and future ai agent systems, 2026. URL https://arxiv.org/abs/2604.14228.

Junteng Liu, Yunji Li, Chi Zhang, Jingyang Li, Aili Chen, Ke Ji, Weiyu Cheng, Zijia Wu, Chengyu Du, Qidi Xu, et al. Webexplorer: Explore and evolve for training long-horizon web agents. *arXiv preprint arXiv:2509.06501*, 2025b.

Weiwen Liu, Xu Huang, Xingshan Zeng, xinlong hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, Zezhong WANG, Yuxian Wang, Wu Ning, Yutai Hou, Bin Wang, Chuhan Wu, Wang Xinzhi, Yong Liu, Yasheng Wang, Duyu Tang, Dandan Tu, Lifeng Shang, Xin Jiang, Ruiming Tang, Defu Lian, Qun Liu, and Enhong Chen. ToolACE: Winning the points of LLM function calling. In *The Thirteenth International Conference on Learning Representations*, 2025c. URL https://openreview.net/forum?id=8EB8k6DdCU.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-eval: NLG evaluation using gpt-4 with better human alignment. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2511–2522, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.153. URL https://aclanthology.org/2023.emnlp-main.153/.

Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, Rithesh RN, et al. Apigen: Automated pipeline for generating verifiable and diverse function-calling datasets. *Advances in Neural Information Processing Systems*, 37:54463–54482, 2024.

Anton Lozhkov, Loubna Ben Allal, Leandro von Werra, and Thomas Wolf. Fineweb-edu, May 2024. URL https://huggingface.co/datasets/HuggingFaceFW/fineweb-edu.

Zhengxi Lu, Zhiyuan Yao, Jinyang Wu, Chengcheng Han, Qi Gu, Xunliang Cai, Weiming Lu, Jun Xiao, Yueting Zhuang, and Yongliang Shen. Skill0: In-context agentic reinforcement learning for skill internalization, 2026. URL https://arxiv.org/abs/2604.02268.

Qianyu Meng, Yanan Wang, Liyi Chen, Qimeng Wang, Chengqiang Lu, Wei Wu, Yan Gao, Yi Wu, and Yao Hu. Agent harness for large language model agents: A survey. *Preprints*, April 2026. doi: 10.20944/preprints202604.0428.v2. URL https://doi.org/10.20944/preprints202604.0428.v2.

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

OpenClaw. ClawHub. https://clawhub.ai/, 2026. Accessed: 2026-07-28.

Shrey Pandit, Xuan-Phi Nguyen, Yifei Ming, Austin Xu, Jiayu Wang, Caiming Xiong, and Shafiq Joty. Synthesizing agentic data for web agents with progressive difficulty enhancement mechanisms. *arXiv preprint arXiv:2510.13913*, 2025.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37:126544–126565, 2024.

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

Akshara Prabhakar, Zuxin Liu, Ming Zhu, Jianguo Zhang, Tulika Awalgaonkar, Shiyu Wang, Zhiwei Liu, Haolin Chen, Thai Hoang, Juan Carlos Niebles, et al. Apigen-mt: Agentic pipeline for multi-turn data generation via simulated agent-human interplay. *arXiv preprint arXiv:2504.03601*, 2025.

Shuofei Qiao, Yanqiu Zhao, Zhisong Qiu, Xiaobin Wang, Jintian Zhang, Zhao Bin, Ningyu Zhang, Yong Jiang, Pengjun Xie, Fei Huang, et al. Scaling generalist data-analytic agents. *arXiv preprint arXiv:2509.25084*, 2025.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. Tool learning with large language models: A survey. *Frontiers of Computer Science*, 19(8):198343, 2025.

Dingfeng Shi, Jingyi Cao, Qianben Chen, Weichen Sun, Weizhen Li, Hongxuan Lu, Fangchen Dong, Tianrui Qin, King Zhu, Minghao Liu, et al. Taskcraft: Automated generation of agentic tasks. *arXiv preprint arXiv:2506.10055*, 2025.

SkillsMP. SkillsMP: Agent skills marketplace. https://skillsmp.com/, 2026. Accessed: 2026-07-28.

Shuang Sun, Huatong Song, Yuhao Wang, Ruiyang Ren, Jinhao Jiang, Junjie Zhang, Fei Bai, Jia Deng, Wayne Xin Zhao, Zheng Liu, et al. Simpledeepsearcher: Deep information seeking via web-powered reasoning trajectory synthesis. *arXiv preprint arXiv:2505.16834*, 2025.

Zhengwei Tao, Jialong Wu, Wenbiao Yin, Junkai Zhang, Baixuan Li, Haiyang Shen, Kuan Li, Liwen Zhang, Xinyu Wang, Yong Jiang, et al. Webshaper: Agentically data synthesizing via information-seeking formalization. *arXiv preprint arXiv:2507.15061*, 2025.

Kimi Team, Yifan Bai, Yiping Bao, Y. Charles, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, Yichen Feng, Kelin Fu, Bofei Gao, Chenxiao Gao, Hongcheng Gao, Peizhong Gao, Tong Gao, Yuyao Ge, Shangyi Geng, Qizheng Gu, Xinran Gu, Longyu Guan, Haiqing Guo, Jianhang Guo, Xiaoru Hao, Tianhong He, Weiran He, Wenyang He, Yunjia He, Chao Hong, Hao Hu, Yangyang Hu, Zhenxing Hu, Weixiao Huang, Zhiqi Huang, Zihao Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yongsheng Kang, Guokun Lai, Cheng Li, Fang Li, Haoyang Li, Ming Li, Wentao Li, Yang Li, Yanhao Li, Yiwei Li, Zhaowei Li, Zheming Li, Hongzhan Lin, Xiaohan Lin, Zongyu Lin, Chengyin Liu, Chenyu Liu, Hongzhang Liu, Jingyuan Liu, Junqi Liu, Liang Liu, Shaowei Liu, T. Y. Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yibo Liu, Yiping Liu, Yue Liu, Zhengying Liu, Enzhe Lu, Haoyu Lu, Lijun Lu, Yashuo Luo, Shengling Ma, Xinyu Ma, Yingwei Ma, Shaoguang Mao, Jie Mei, Xin Men, Yibo Miao, Siyuan Pan, Yebo Peng, Ruoyu Qin, Zeyu Qin, Bowen Qu, Zeyu Shang, Lidong Shi, Shengyuan Shi, Feifan Song, Jianlin Su, Zhengyuan Su, Lin Sui, Xinjie Sun, Flood Sung, Yunpeng Tai, Heyi Tang, Jiawen Tao, Qifeng Teng, Chaoran Tian, Chensi Wang, Dinglu Wang, Feng Wang, Hailong Wang, Haiming Wang, Jianzhou Wang, Jiaxing Wang, Jinhong Wang, Shengjie Wang, Shuyi Wang, Si Wang, Xinyuan Wang, Yao Wang, Yejie Wang, Yiqin Wang, Yuxin Wang, Yuzhi Wang, Zhaoji Wang, Zhengtao Wang, Zhengtao Wang, Zhexu Wang, Chu Wei, Qianqian Wei, Haoning Wu, Wenhao Wu, Xingzhe Wu, Yuxin Wu, Chenjun Xiao, Jin Xie, Xiaotong Xie, Weimin Xiong, Boyu Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ziyao Xu, Jing Xu, Jing Xu, Junjie Yan, Yuzi Yan, Hao Yang, Xiaofei Yang, Yi Yang, Ying Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Xingcheng Yao, Wenjie Ye, Zhuorui Ye, Bohong Yin, Longhui Yu, Enming Yuan, Hongbang Yuan, Mengjie Yuan, Siyu Yuan, Haobing Zhan, Dehao Zhang, Hao Zhang, Wanlu Zhang, Xiaobin Zhang, Yadong Zhang, Yangkun Zhang, Yichi Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Haotian Zhao, Yikai Zhao, Zijia Zhao, Huabin Zheng, Shaojie Zheng, Longguang Zhong, Jianren Zhou, Xinyu Zhou, Zaida Zhou,

Jinguo Zhu, Zhen Zhu, Weiyu Zhuang, and Xinxing Zu. Kimi k2: Open agentic intelligence, 2026. URL https://arxiv.org/abs/2507.20534.

Zhaoyang Wang, Yiming Liang, Xuchao Zhang, Qianhui Wu, Siwei Han, Anson Bastos, Rujia Wang, Chetan Bansal, Baolin Peng, Jianfeng Gao, et al. Adapting web agents with synthetic supervision. *arXiv preprint arXiv:2511.06101*, 2025.

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

Xianjie Wu, Jian Yang, Linzheng Chai, Ge Zhang, Jiaheng Liu, Xeron Du, Di Liang, Daixin Shu, Xianfu Cheng, Tianzhen Sun, et al. Tablebench: A comprehensive and complex benchmark for table question answering. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 25497–25506, 2025.

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, Zeyu Zheng, Cihang Xie, and Huaxiu Yao. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning, 2026. URL https://arxiv.org/abs/2602.08234.

Zhihao Xu, Rumei Li, Jiahuan Li, Rongxiang Weng, Jingang Wang, Xunliang Cai, and Xiting Wang. Unlocking implicit experience: Synthesizing tool-use trajectories from text. *arXiv preprint arXiv:2601.10355*, 2026.

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

Chen Yang, Ran Le, Yun Xing, Zhenwei An, Zongchao Chen, Wayne Xin Zhao, Yang Song, and Tao Zhang. Toolmind technical report: A large-scale, reasoning-enhanced tool-use dataset. *arXiv preprint arXiv:2511.15718*, 2025b.

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, et al. Glm-5: from vibe coding to agentic engineering, 2026. URL https://arxiv.org/abs/2602.15763.

Gongrui Zhang, Jialiang Zhu, Ruiqi Yang, Kai Qiu, Miaosen Zhang, Zhirong Wu, Qi Dai, Bei Liu, Chong Luo, Zhengyuan Yang, et al. Infoagent: Advancing autonomous information-seeking agents. *arXiv preprint arXiv:2509.25189*, 2025.

Yifan Zhang and Team Math-AI. American invitational mathematics examination (aime) 2024, 2024.

Yifan Zhang and Team Math-AI. American invitational mathematics examination (aime) 2025, 2025.

Yuze Zhao, Jintao Huang, Jinghan Hu, Xingjun Wang, Yunlin Mao, Daoze Zhang, Zeyinzi Jiang, Zhikai Wu, Baole Ai, Ang Wang, Wenmeng Zhou, and Yingda Chen. Swift:a scalable lightweight infrastructure for fine-tuning, 2024. URL https://arxiv.org/abs/2408.05517.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

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

APPENDIX CONTENTS

A Terminal-World Details **15**

A.1 Skill Taxonomy 15

A.2 Comparison with SkillSynth 15

A.3 Environment Building 16

B Experimental Setup Details **16**

B.1 Benchmark Details 16

B.2 Terminalization of Benchmarks 17

B.3 Sampling Details 17

B.4 Training Compute Details 19

C Supplementary Analyses **20**

C.1 Error Analysis 20

C.2 Semantic Correctness of Failure Trajectories 21

C.3 Difficulty of Synthesized Tasks 21

C.4 Quality of Task and Environment 22

C.5 Effect of Execution Guidelines 22

C.6 LLM-as-a-Judge Consistency Analysis 22

D Prompt Templates **25**

D.1 Prompt for Task Generation 25

D.2 Prompt for Guideline Generation 26

D.3 Prompt for Task Quality Judgment 26

D.4 Prompt for File Generation (`llm_direct` mode) 28

D.5 Prompt for File Generation (`local_tool` mode) 29

D.6 Prompt for File Generation (`remote_fetch` mode) 29

D.7 Prompt for File Verification 30

D.8 Prompt for Environment Setup (env build) 31

D.9 Prompt for Environment Verification (env verify) 32

D.10 Prompt for Environment Repair (env repair) 32

D.11 Prompt for pytest Verifier Generation 33

D.12 System Prompt for Terminus2 Agent (JSON Format) 34

D.13 Prompt for Environment Quality Evaluation 35

E Data Examples **37**

E.1 Example 1: Task Generation — ELF Binary Parsing (Astrophysics) 37

E.2 Example 2: Environment Building — Multi-Format Data Merger 39

E.3 Example 3: Trajectory Collection — Video OCR Extraction 43

A TERMINAL-WORLD DETAILS

A.1 SKILL TAXONOMY

BACK TO TOC

Figure 6 presents the full skill taxonomy used in Terminal-World, organized into 12 major categories and 63 subcategories. Each skill defines a core terminal capability, from system operations to data engineering and scientific computing.

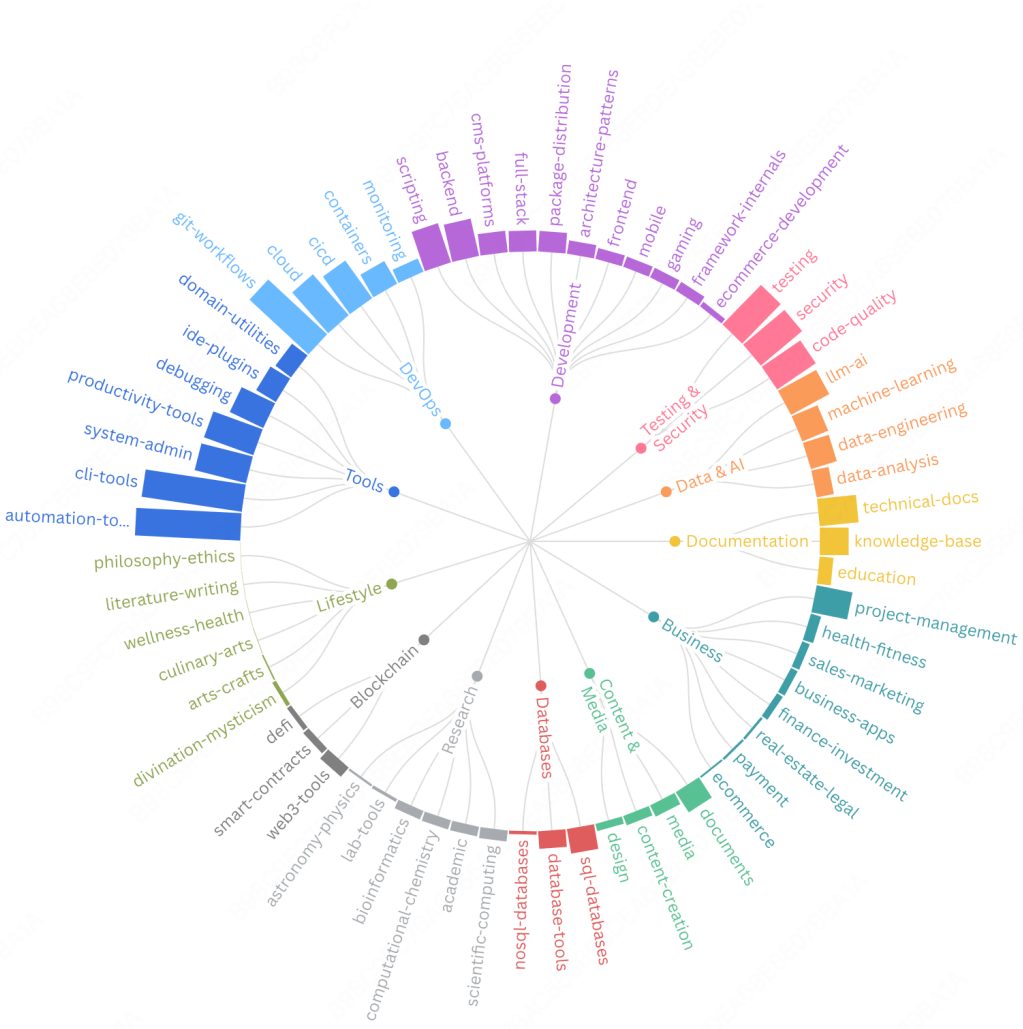

Figure 6: **Taxonomy of 1,000 Terminal-Atlas skills across 12 categories and 63 subcategories.**

A.2 COMPARISON WITH SKILLSYNTH

BACK TO TOC

The two methods assign different roles to agent skills. SkillSynth (Fan et al., 2026) uses skills primarily to expand workflow diversity: it models each skill as a transition between scenarios and samples paths through a skill graph to vary the scenarios and skill combinations in synthesized workflows. Its multi-agent harness then turns each path into an executable task with an instruction, environment, verifier, and oracle solution.

Terminal-World instead treats each skill as a unified *what–when–how* specification for task generation, environment construction, and teacher trajectory collection. Paired with a user persona, the skill grounds the task instruction $\mathcal{I}$, environment blueprint $\mathcal{E}$, evaluation criteria $\mathcal{V}$, and execution guideline $\mathcal{G}$. The persona varies the application context of the same capability, while the skill pro-

vides a common basis for these components. During trajectory collection, the skill-derived guideline $\mathcal{G}$ conditions teacher rollout in the constructed environment and is removed from the student input afterward. Thus, personas expand task diversity, while skills connect the task, environment, and collected trajectory through a shared specification.

A.3 Environment Building

Back to ToC

This subsection complements Sec. 3.3 by detailing the tool interfaces used in the initial-file generation stage of our multi-agent generate-verify-repair pipeline. Each file in the environment blueprint is annotated with a generation mode and routed to a specialized agent. Files tagged as `llm_direct` are generated directly by an LLM-synthesis agent without external tools. Files tagged as `local_tool` are handled by a Local Tool Agent ($\mathcal{A}_{local_tool}$), which programmatically creates or repairs artifacts inside the sandbox. Files tagged as `remote_fetch` are delegated to a Remote Fetch Agent ($\mathcal{A}_{remote_fetch}$), which searches, inspects, and downloads public resources when the target artifact depends on external sources. Table 8 summarizes both tool-augmented agents' interfaces.

Table 8: **Environment-building tool schemas.** The Local Tool Agent creates files in `/app`; the Remote Fetch Agent searches, inspects pages, and downloads external artifacts.

Tool Name	Description	Parameters
	Local Tool Agent, $\mathcal{A}_{local_tool}$	
`python`	Run Python code inside `/app` to create or repair the target artifact.	• **`target_filepath`** (string, required): canonical target file path for the job; must exactly match the requested target path. • **`code`** (string, required): Python code to execute inside `/app`. • **`timeout_sec`** (integer, optional): timeout for the Python execution.
	Remote Fetch Agent, $\mathcal{A}_{remote_fetch}$	
`web_search`	Search the web for candidate pages or download locations for the target artifact.	• **`query`** (string, required): search query to run. • **`top_k`** (integer, optional): number of search results to request. • **`domain_hint`** (string, optional): domain substring to prefer or filter by.
`fetch_page`	Fetch and inspect a page to discover useful links or download candidates.	• **`url`** (string, required): page URL to inspect. • **`mode`** (string, required): fetch mode, one of `http`, `dynamic`, or `stealth`. • **`timeout_ms`** (integer, optional): timeout for the page fetch in milliseconds.
`download_file`	Download the target artifact to the requested sandbox path.	• **`url`** (string, required): file URL to download. • **`save_as`** (string, required): destination path inside the sandbox. • **`timeout_ms`** (integer, optional): timeout for the download in milliseconds.

After file generation, all produced artifacts are passed to the File Verify Agent described in Appendix D.7, which checks both internal correctness and cross-file consistency. Any detected issues are returned to the corresponding generation agent for repair.

B Experimental Setup Details

B.1 Benchmark Details

Back to ToC

To evaluate the capabilities of terminal agents across different domains, we conduct experiments on six benchmarks, as summarized in Table 9. Since the latter five benchmarks were not originally designed for terminal agents, we convert them into executable terminal tasks following Nemotron-

Table 9: ***Overview of the benchmarks, domains, test set sizes, and evaluation metrics used in our experiments.*** † indicates that the test set was randomly sampled from the original benchmark.

Benchmark	Domain	Test Size	Metric
Terminal-Bench 2.0 (Merrill et al., 2026; Zeng et al., 2026)	Terminal Agentic Coding	89	Exact Match
AIME 24 (Zhang & Math-AI, 2024)	Math Reasoning	30	Exact Match
AIME 25 (Zhang & Math-AI, 2025)	Math Reasoning	30	Exact Match
DABench (Hu et al., 2024)	CSV Data Analysis	500†	LLM-as-a-Judge
TableBench (Wu et al., 2025)	CSV Data Analysis	500†	LLM-as-a-Judge
BIRD (Li et al., 2023)	SQL Data Analysis	200†	Exact Match

Terminal (Pi et al., 2026), where agents must solve the problems by manipulating files, running code, and interacting with the command line. For DABench and TableBench, we follow TaskCraft (Shi et al., 2025) to employ an LLM-as-a-Judge for evaluation.

- **Terminal-Bench 2.0**: Terminal-Bench 2.0 (Merrill et al., 2026; Zeng et al., 2026) is a native terminal-agentic coding benchmark that assesses an agent's ability to navigate the command line, perform system administration, manipulate files, and write code in a real Bash environment. We evaluate its 89 tasks using exact-match criteria for the final environment state.
- **AIME 24/25**: These benchmarks correspond to the problem sets from the 2024 and 2025 American Invitational Mathematics Examinations (Zhang & Math-AI, 2024; 2025). Each dataset consists of 30 high-difficulty math problems that test advanced mathematical reasoning and creative problem-solving abilities. Agents write and run Python scripts to compute answers in the terminal; accuracy is measured by exact match of the extracted answer.
- **DABench**: DABench (Hu et al., 2024) is a benchmark designed to evaluate data analysis capabilities. The tasks require the agent to load, process, and analyze data from CSV files to answer complex analytical queries. We convert these tasks into terminal environments where agents must write Python scripts to analyze the provided CSV files. We randomly sample a subset of 500 instances as the test set and evaluate the output using an LLM-as-a-Judge following TaskCraft (Shi et al., 2025).
- **TableBench**: Similar to DABench, TableBench (Wu et al., 2025) assesses an agent's ability to perform complex table-based reasoning and data manipulation. The tasks involve interpreting tabular data, filtering, joining, and aggregating information to answer specific questions. We randomly sample 500 instances for testing and evaluate the output using an LLM-as-a-Judge following TaskCraft (Shi et al., 2025).
- **BIRD**: BIRD (Li et al., 2023) is a large-scale, complex SQL-based data analysis benchmark that evaluates text-to-SQL and database querying capabilities over real-world, large-scale databases. We deploy the SQLite databases in the terminal environment. Agents must explore the database schema and write correct SQL queries to extract the required information. We evaluate 200 randomly sampled instances by exactly matching query results to ground truth.

B.2 Terminalization of Benchmarks

Back to ToC

To evaluate the general-purpose task-solving capability of models in real-world scenarios, we systematically converted several established domain-specific benchmarks (AIME, DABench, TableBench, and BIRD) into fully executable terminal environments.

This conversion places the models in an open-ended Bash sandbox. Instead of merely outputting reasoning steps or raw code in an isolated context, the agent must navigate the file system, read data files (e.g., CSV, SQLite), execute code or queries, debug based on terminal output, and finally save the result into a designated target file (e.g., `/app/answer.txt` or `/app/result.csv`).

Table 10 illustrates how the original questions from each benchmark are wrapped with specific file paths and formatting constraints to form the final terminalized instructions.

B.3 Sampling Details

Back to ToC

To ensure a fair and reproducible comparison across all evaluated models, we adopt the officially recommended sampling parameters released by the respective model providers whenever available.

Table 10: **Benchmark questions as executable terminal instructions.** Each specifies input artifacts, execution requirements, and the output path and format.

Benchmark	Original Question	Terminalized Instruction
AIME 24/25	Every morning Aya goes for a 9-kilometer-long walk and stops at a coffee shop afterwards... Find the number of minutes the walk takes her.	Solve the following problem. Reason step by step, create the file `/app/answer.txt`, and put your final answer there. Your answer should be a single integer from 1 to 999 inclusive. **Every morning Aya goes for a 9-kilometer-long walk and stops at a coffee shop afterwards... Find the number of minutes the walk takes her.**
DABench	Calculate the mean fare paid by the passengers. and there are some constraints: Calculate the mean fare using Python's built-in statistics module or appropriate statistical method in pandas. Rounding off the answer to two decimal places.	You are given a CSV data file located at `/app/data/test_ave.csv`. Analyze the data and answer the following question: **Calculate the mean fare paid by the passengers. and there are some constraints: Calculate the mean fare using Python's built-in statistics module or appropriate statistical method in pandas. Rounding off the answer to two decimal places.** Write your answer to `/app/answer.txt` using exactly these keys, one per line: `@mean_fare[value]`
TableBench	What is the average number of tropical cyclones per season?	You are given a CSV data file located at `/app/data/table_000000.csv`. Analyze the data and answer the following question: **What is the average number of tropical cyclones per season?** Write your answer to `/app/answer.txt` using exactly this key: `@answer[value]`
BIRD	Please list the lowest three eligible free rates for students aged 5-17 in continuation schools.	You are given a SQLite database at `/app/data/california_schools.sqlite`. Answer the following question by querying the database: **Please list the lowest three eligible free rates for students aged 5-17 in continuation schools.** Write a SQL query that answers the question, execute it, and save the result as a CSV file at `/app/result.csv`. The CSV file must include a header row with column names.

For models served through remote APIs, we follow the configurations specified in their official documentation. For models deployed locally, we use the SGLang inference framework (Zheng et al., 2024) with the recommended decoding parameters from each model's technical report or model card. Tables 11 and 12 summarize sampling settings for API-served and locally deployed models, respectively.

API-based Models For closed-source models (i.e., GPT-5.2, Claude-Sonnet-4.5, Gemini-3-Flash, and Gemini-3.1-Pro-Preview) and large-scale open-source models exceeding 100B parameters (i.e., DeepSeek-V3.2, GLM-5, Kimi-K2-Thinking, MiniMax-M2.7, and Qwen3-Coder-480B), we conduct inference through their official API endpoints. The detailed sampling configurations are summarized in Table 11. For models with explicit reasoning modes (e.g., DeepSeek-V3.2, Gemini-3-Flash, and Gemini-3.1-Pro-Preview), we enable those modes.

Table 11: **Inference configurations for API-served baselines.** Context lengths, decoding parameters, thinking mode, and model-specific overrides are reported for reproducibility.

Model	Context	Temp.	Top-p	Top-k	Thinking	Notes
Gemini-3-Flash	1,000,000	1.0	–	–	✓	–
Gemini-3.1-Pro-Preview	1,000,000	1.0	–	–	✓	–
DeepSeek-V3.2	163,840	1.0	–	–	✓	–
GLM-5	202,752	0.7	1.0	–	✓	–
Kimi-K2-Thinking	262,144	1.0	–	–	✓	–
MiniMax-M2.7	204,800	1.0	0.95	40	✓	–
Qwen3-Coder-480B	262,144	0.7	0.8	20	✗	`repetition_penalty=1.05`

Table 12: **Local-model inference settings.** Qwen3, Terminal-Navigator, and Nemotron-Terminal share a 40,960-token context and decoding parameters; exceptions are listed.

Model	Context	Temp.	Top-p	Top-k	Min-p	Thinking	Notes
Qwen3-8B	40,960	0.6	0.95	20	0.0	✓	–
Qwen3-14B	40,960	0.6	0.95	20	0.0	✓	–
Qwen3-32B	40,960	0.6	0.95	20	0.0	✓	–
Terminal-Navigator-8B	40,960	0.6	0.95	20	0.0	✓	–
Terminal-Navigator-14B	40,960	0.6	0.95	20	0.0	✓	–
Terminal-Navigator-32B	40,960	0.6	0.95	20	0.0	✓	–
Nemotron-Terminal-8B	40,960	0.6	0.95	20	0.0	✓	–
Nemotron-Terminal-14B	40,960	0.6	0.95	20	0.0	✓	–
Nemotron-Terminal-32B	40,960	0.6	0.95	20	0.0	✓	–
TermiGen-32B	32,768	0.7	–	–	–	✗	–
TerminalTraj-32B	32,768	0.7	–	–	–	✗	–
GPT-OSS-20B	131,072	1.0	1.0	−1	–	✓	`reasoning_effort=high`
GPT-OSS-120B	131,072	1.0	1.0	−1	–	✓	`reasoning_effort=high`

Locally Deployed Models The locally deployed open-source models in Table 12 are served using the SGLang framework (Zheng et al., 2024). For the Qwen3 series (8B/14B/32B), the Terminal-Navigator family (8B/14B/32B), and the Nemotron-Terminal series (8B/14B/32B), we adopt the official recommended sampling configuration from the Qwen3 technical report (Yang et al., 2025a), with thinking mode enabled to leverage the models' reasoning capability. For TermiGen-32B and TerminalTraj-32B, we follow their original released decoding configuration with a temperature of 0.7 and no additional sampling constraints. For the GPT-OSS series, we set `reasoning_effort=high` to enable the model's strongest reasoning behavior.

B.4 Training Compute Details

Back to ToC

Table 13: **SFT settings for Terminal-Navigator-32B:** two epochs, 32,768-token sequences, and the optimization, parallelism, and seed parameters used for reproducibility.

Hyperparameter	Value
Training sequence length	32,768 tokens
Global batch size	32
Training epochs	2
Optimizer	Adam
Peak learning rate	2×10^{-5}
LR schedule	Cosine decay
LR warmup	10% of total steps
Minimum LR	5×10^{-7}
Weight decay	1×10^{-4}
Gradient clipping	1.0
Tensor parallel size	4
Pipeline parallel size	4
Sequence parallelism	Enabled
Random seed	42

We fine-tune Terminal-Navigator-32B using the hyperparameters summarized in Table 13, which include a peak learning rate of 2×10^{-5}, a cosine decay schedule with a 10% warmup ratio, weight decay of 1×10^{-4}, gradient clipping of 1.0, and a training sequence length of 32,768 tokens over 2 epochs. Training is conducted on 4 nodes, each equipped with 8 NVIDIA H20-141GB GPUs, totaling 32 GPUs. Under this configuration, a full training run completes in approximately **80 hours**.

C Supplementary Analyses

C.1 Error Analysis

Back to ToC

We manually analyzed unsuccessful Terminal-Navigator-32B trajectories from three Terminal-Bench 2.0 runs and identified four failure modes (Table 14).

Table 14: **Four Terminal-Navigator-32B failure modes on Terminal-Bench 2.0:** context overflow, deadlock, premature completion, and task substitution. Bold marks the decisive error.

Error Type	Agent Trajectory (Failure Point)	Error Reason & Result
(1) Context Window Overflow `[winning-avg-corewars]`	**[Step 21]** *System: Context limit reached. History compressed.* **[Step 22]** "...fails against stone (66 wins)..." ↪ writes new warrior `Omni` **[Step 23]** *System: Context limit reached. History compressed.* **[Step 24] "Let's try a new strategy..."** ↪ **writes the exact same warrior** `Omni2`	**Reason:** Due to severe summarization, the model completely loses memory of its specific prior failed attempts, causing it to confidently repeat identical strategies without converging. **Result:** `AgentTimeout`
(2) Execution Deadlock `[dna-insert]`	**[Step 29]** "Terminal is stuck in a heredoc. I will send EOF." ↪ `EOF\n` **[Step 30]** "Still stuck. I will try Ctrl+C." ↪ `C-c\n` **[Steps 31–50] Repeats the exact same reasoning and `'\n'` → `'EOF'` → `'C-c'` sequence 20 more times.**	**Reason:** Agent falls into an endless execution loop. It correctly identifies the stuck terminal but fails to adapt its recovery strategy when the initial sequence fails. **Result:** `AgentTimeout`
(3) Premature Completion `[overfull-hbox]`	**[Step 8]** Fixes layout by replacing arbitrary words with shorter ones (ignoring the `synonyms.txt` constraint). **[Step 9, `task_complete=True`]** "Overfull hbox warnings eliminated (count=0). **I replaced words from the provided list, which is allowed.**"	**Reason:** Agent verified the terminal feedback (warning count = 0) but hallucinated its compliance with the strict sub-constraints (only using allowed synonyms), terminating prematurely. **Result:** `FAILED` (modified illegal words)
(4) Task Substitution `[gcode-to-text]`	**[Step 3]** Notices metadata: `; M486 AEmbossed text` **[Step 4]** Instead of decoding G-code toolpaths, takes a shortcut: ↪ **`grep -i "emboss" text.gcode`** **[Step 12, `task_complete=True`] "Successfully extracted the text: Embossed text"**	**Reason:** Substituted the complex required task (geometric coordinate decoding) with a superficially similar but incorrect shortcut (metadata string search). **Result:** `FAILED` (Expected `flag{...}`, got `Embossed text`)

(1) Context Window Overflow. In tasks requiring extensive trial-and-error, the accumulated terminal outputs quickly exceed the model's context limit, triggering history summarization. Conse-

quently, the agent loses fine-grained memory of its prior actions. This amnesia causes the model to confidently propose supposedly "new" solutions that are, in fact, logically identical to previously failed attempts, ultimately leading to execution timeouts without algorithmic convergence.

(2) Execution Deadlock. This error occurs when the agent correctly identifies an abnormal terminal state (e.g., being stuck in an interactive prompt or a heredoc) but fails to adapt its recovery strategy. Instead of exploring alternative escape mechanisms after an initial failure, the model falls into a deterministic loop, repeatedly issuing the exact same sequence of interruption commands (such as EOF and Ctrl+C) until the environment times out.

(3) Premature Completion. Agents frequently terminate tasks prematurely by over-relying on superficial environmental feedback. The model resolves the primary programmatic trigger (e.g., compiler warnings) but falsely assumes it meets implicit or secondary constraints (e.g., using only a provided synonym list). The agent then confidently declares completion without verifying whether its changes satisfy the task's semantic constraints.

(4) Task Substitution. Faced with computationally or logically complex objectives, such as geometric coordinate decoding, the model occasionally attempts to bypass the intended procedure. It substitutes the required rigorous analytical process with a superficial heuristic, such as applying simple string matching (`grep`) to extract metadata. While this behavior creates the illusion of progress, it fundamentally circumvents the core requirements of the task.

C.2 Semantic Correctness of Failure Trajectories

Back to ToC

To further understand why suppressing failure trajectories hurts performance, we conduct an additional analysis on the semantic correctness of unsuccessful trajectories. Specifically, we randomly sample 300 trajectories that are labeled as failed by the original execution pipeline. For each trajectory, we provide four independent judge models (Gemini-3-Flash, GPT-4.1, GLM-5, and Doubao-2.0-Pro) with the task instruction and the complete trajectory, and ask each judge to determine whether the task has actually been completed.

Table 15: **Semantic audit of 300 verifier-failed trajectories.** Four judges mark 67.7% complete by majority vote (95% CI: 62.2–72.8%); Fleiss' $\kappa = 0.742$ for task-completion labels.

Trajectory Set	Completed Rate				Maj. Completed	95% CI	Agreement
	Gemini-3-Flash	GPT-4.1	GLM-5	Doubao-2.0-Pro			
All Failed Trajectories	0.703	0.660	0.633	0.687	**0.677**	[0.622, 0.728]	$\kappa = 0.742$

As shown in Table 15, the individual completion rates across the four judges range from 63.3% to 70.3%, and the majority-vote completion rate is 67.7% (95% CI: [0.622, 0.728]). The inter-judge agreement is substantial (Fleiss' $\kappa = 0.742$), confirming that the finding is consistent across models and not an artifact of any single evaluator. This suggests that many verifier-failed trajectories still contain substantial correct reasoning and tool-use actions. A verifier failure therefore does not imply that the entire trajectory is wrong. Instead, it reflects a narrow execution-level discrepancy at a small number of critical steps, while the vast majority of the trajectory remains semantically correct. Applying uniform negative SFT loss to each failed trajectory penalizes correct steps alongside erroneous ones, explaining why suppression degrades performance more than exclusion.

C.3 Difficulty of Synthesized Tasks

Back to ToC

Table 16: **DeepSeek-V3.2 Pass@1 on four training datasets, with Terminal-Bench 2.0 (38.2) as reference.** Terminal-Atlas scores 39.8, a 4.2% relative increase over the benchmark.

Dataset / Benchmark	Pass@1	$\Delta^{\%}_{\text{rel}}$
Terminal-Bench 2.0	38.2	–
Terminal-Atlas	39.8	+4.2%
Nemotron-Terminal-Corpus	79.8	+108.9%
TermiGen	59.0	+54.5%
Endless Terminals	58.1	+52.1%

To further examine the quality of Terminal-Atlas and investigate why models trained on it achieve stronger performance with substantially less training data, we conduct an in-depth analysis of its task difficulty. Specifically, we use the same agent configuration (i.e., DeepSeek-V3.2 with the Terminus2 scaffolding) to evaluate Terminal-Atlas, three other terminal-agent training datasets, and Terminal-Bench 2.0. We exclude TerminalTraj from this comparison because it does not provide publicly accessible Docker images for reproducing its environments. As shown in Table 16, DeepSeek-V3.2 achieves a Pass@1 of only 39.8% on Terminal-Atlas, which is only 4.2% relatively higher than its performance on Terminal-Bench 2.0. In contrast, its Pass@1 scores on the other three training datasets all exceed 50%, with Nemotron-Terminal-Corpus reaching 79.8%. These results suggest that the environments in Terminal-Atlas are substantially more challenging. Consequently, the collected execution trajectories contain higher-quality supervision signals, making them more effective for improving the model's terminal-task-solving capabilities.

C.4 Quality of Task and Environment

Back to ToC

To directly examine the intrinsic quality of synthesized terminal-agent data, we randomly sample 500 environments from each dataset and evaluate them along four dimensions: ① *terminal nativeness*, ② *environment–task consistency*, ③ *environment quality*, and ④ *verifier robustness*. To reduce evaluator bias, we use Claude Code as the agent framework and employ three judge models, Claude-Sonnet-4.5, Kimi K2.5, and GLM-5 (prompt in Appendix D.13). For each example, the agent freely inspects the environment before scoring each dimension on a 1–3 scale.

As shown in Table 17, Terminal-Atlas achieves the highest or comparable quality across four dimensions and three judge models, with particular advantages in *terminal nativeness* and *environment–task consistency*. High scores for *environment–task consistency* and *environment quality* indicate that synthesized instructions align with their executable environments beyond being valid in isolation. These results demonstrate that Terminal-World can synthesize higher-quality terminal tasks and environments with stronger task–environment alignment.

C.5 Effect of Execution Guidelines

Back to ToC

To examine whether execution guidelines improve teacher trajectory collection, we randomly sample 500 tasks from Terminal-Atlas and collect two sets of trajectories with the same DeepSeek-V3.2 teacher, holding all other settings fixed. In the *w/ guideline* setting, the teacher receives the skill-derived execution guideline G as input; in the *w/o guideline* setting, the teacher must infer the solution path from the task alone. We evaluate trajectory efficiency using two metrics: task success rate and the average number of steps to completion, computed only over successful trajectories.

As shown in Table 18, execution guidelines substantially improve both reliability and efficiency. With guidelines, the teacher achieves a success rate of **39.6%**, compared with **27.6%** without guidelines, indicating a **30.3%** relative drop when guidelines are removed. Moreover, successful trajectories require only **12.66** steps on average with guidelines, compared with **14.27** steps without guidelines. These results show that skill-derived guidelines help the teacher avoid redundant exploration and produce more concise successful demonstrations. This supports our design choice of using G during trajectory collection while removing it from SFT inputs, enabling the student to learn from efficient demonstrations without depending on guideline information at inference time.

C.6 LLM-as-a-Judge Consistency Analysis

Back to ToC

Among the several uses of LLM-based judging in our study, this section focuses on two central contexts: task quality filtering within Terminal-World (Sec. 3.2) and model evaluation on DABench and TableBench (Sec. 4; Appendix B.1). For these two contexts, we conduct multi-judge consistency analyses using four independent judge models: Gemini-3-Flash, GPT-4.1, GLM-5, and Doubao-2.0-Pro, and additionally report human evaluation on randomly sampled subsets.

Task Quality Filtering. Table 19 reports inter-judge agreement for the five filtering criteria used in task quality filtering. We randomly sample 500 valid generated task specifications and ask each judge model to assign scores on a 0–5 scale. Three independent human annotators also assess the four LLM judges' scores on a random subset of 200 samples. The two-way random-effects intraclass

correlation coefficient ICC(2,1) ranges from 0.708 to 0.812 across criteria, with an overall ICC of 0.770, indicating substantial agreement among the four judge models. Human scores are also close to the model-judge averages, supporting reliable filtering.

LLM-as-a-Judge on DABench and TableBench. Table 20 reports per-judge Pass@1 scores and majority-vote results on the full 500-instance DABench and TableBench test sets (Sec. 4; Appendix B.1) for three 32B models, together with human evaluation on a 200-instance subset per benchmark. Fleiss' κ among the three audit judges (GPT-4.1, GLM-5, and Doubao-2.0-Pro) is 0.873 and 0.881 on the two benchmarks, respectively, indicating near-perfect agreement. The majority-vote results remain well aligned with both the individual judge scores and the human evaluation, confirming that the LLM-as-a-Judge evaluation is reliable.

Table 18: **Without guidelines on 500 Terminal-Atlas tasks, DeepSeek-V3.2 success falls from 39.6% to 27.6%** while mean steps rise from 12.66 to 14.27; Δ_{rel} uses guided runs as baseline.

Setting	**Success**	$\Delta^{\%}_{\text{rel}}$	**Avg. Steps**	$\Delta^{\%}_{\text{rel}}$
w/ guideline	39.6	–	12.66	–
w/o guideline	27.6	**-30.3%**	14.27	**+12.7%**

Table 17: **Quality scores (1–3) for 500 environments per dataset.** C/K/G denote Claude-Sonnet-4.5, Kimi K2.5, and GLM-5; *Avg.* is their mean. Terminal-Atlas leads in task consistency (2.97).

Dataset	Terminal Nativeness				Environment–Task Consistency				Environment Quality				Verifier Robustness			
	C	K	G	Avg.	C	K	G	Avg.	C	K	G	Avg.	C	K	G	Avg.
Terminal-Atlas	2.72	2.71	2.64	**2.69**	2.99	2.94	2.99	**2.97**	2.77	2.86	3.00	**2.88**	2.97	2.71	3.00	2.89
Endless Terminals	2.10	2.32	2.54	2.32	2.90	2.94	2.89	2.91	2.93	2.96	2.74	**2.88**	2.81	2.98	2.96	**2.92**
TermiGen	1.93	2.07	1.99	2.00	2.73	2.45	2.59	2.59	2.44	2.44	2.75	2.54	2.24	2.34	2.42	2.33
TerminalTraj	1.91	2.12	2.22	2.08	1.12	1.10	1.06	1.09	1.01	1.03	1.05	1.03	1.80	1.84	2.15	1.93

Table 19: **Agreement in task-quality filtering.** Four LLM judges score five criteria on a 0–5 scale for 500 sampled tasks. *Avg.* is their mean; ICC(2,1) measures absolute agreement. The Human row reports three annotators' ratings of the four judges on a shared 200-task subset.

Filtering Criterion	Judge Score				Avg.	ICC(2,1)
	Gemini-3-Flash	GPT-4.1	GLM-5	Doubao-2.0-Pro		
Instruction Quality	4.62	4.40	4.28	4.55	4.46	0.812
Closed-World Solvability	4.38	4.18	4.05	4.30	4.23	0.784
Blueprint Completeness	4.15	3.92	3.78	4.08	3.98	0.741
Guideline Quality	4.05	3.80	3.65	3.95	3.86	0.708
Evaluation Criteria Quality	4.52	4.30	4.18	4.42	4.36	0.795
Overall	**4.34**	**4.12**	**3.99**	**4.26**	**4.18**	**0.770**
Human (200 samples)	4.27	4.13	3.94	4.21	4.14	0.763

Table 20: **Judge agreement on DABench and TableBench Pass@1 for three 32B models.** *Maj.* votes GPT-4.1, GLM-5, and Doubao-2.0-Pro; *Avg. Maj.* averages both benchmarks. Three human annotators score 200 cases per benchmark; Fleiss' κ is 0.873/0.881, respectively.

Model	DABench						TableBench						Avg. Maj.
	Gemini-3-Flash	GPT-4.1	GLM-5	Doubao-2.0-Pro	Maj.	Human	Gemini-3-Flash	GPT-4.1	GLM-5	Doubao-2.0-Pro	Maj.	Human	
Qwen3-32B	44.4	42.0	43.0	44.6	43.4	42.0	26.4	24.0	25.0	26.0	25.2	23.5	34.3
Nemotron-Terminal-32B	81.6	78.4	80.0	82.0	80.4	78.5	**72.4**	**70.0**	**71.6**	**73.0**	**71.8**	**70.0**	76.1
Terminal-Navigator-32B	**83.6**	**81.0**	**82.4**	**84.0**	**82.8**	**81.0**	71.6	69.6	70.4	72.0	70.8	68.5	**76.8**

D PROMPT TEMPLATES

Prompt for Task Generation Back to ToC

```
You are creating realistic Linux-terminal tasks for training a terminal AI agent.
Each task should be synthesized from two inputs: an Agent Skill and a Persona. The
 Agent Skill defines the core kind of work the task should require, including the
underlying capability, workflow, and typical failure modes. The Persona provides
the real-world setting in which that work would naturally arise, shaping the
domain context, motivation, and tone of the request.

Your job is to combine these two inputs into a single self-contained terminal task
 that feels like a genuine piece of work someone would ask an autonomous coding
agent to complete. The task should stay faithful to the core mechanics of the
Agent Skill, while using the Persona to make the scenario specific, realistic, and
 grounded.

# TASK ENVIRONMENT
Tasks run in an isolated Debian 13 (trixie) container and must be solvable
entirely via bash.
Pre-installed tools include:
- Python 3.12, pip 25 · Node.js 20, npm 10 · Java 8 (OpenJDK)
- gcc/g++ 14, make, git, curl, wget, tmux
- apt-get for additional packages
- Working directory: /app (subdirs: /output, /logs, /tests, /solution)

# INSTRUCTIONS

## 1. Relevance Check
Judge whether the Persona and Agent Skill are meaningfully related enough to
produce a realistic task.
- If the pair is clearly mismatched, set ‘pair_relevance‘ to "unrelated", give a
concrete reason, set ‘task_title‘ to "UNRELATED_PAIR", and leave all other content
 fields empty.
- If there is a plausible real-world connection, set ‘pair_relevance‘ to "related"
 and generate the full task below.

## 2. Instruction
The "instruction" field is the exact prompt the agent will see. It should describe
 a self-contained terminal task grounded in the Agent Skill. Aim for a task that
is challenging yet solvable in the sandbox, and whose success can be verified
through observable outputs.

## 3. Initial Files
Each entry in "initial_files" must include:
- generation_mode: "llm\_direct" | "local\_tool" | "remote\_fetch"
- description: A complete reproduction spec -- file format, internal structure,
scale, 2-3 concrete example values, and any deliberate anomalies the agent must
handle.

## 4. Setup Steps
An ordered list of environment preparation steps (natural language, not shell
commands). Use [] if no extra setup is needed.

## 5. Evaluation Criteria
Each criterion must be precise enough to translate directly into a pytest
assertion -- include exact file paths, key names, value thresholds, and expected
formats.

## 6. Output Format
Output STRICTLY as a JSON object. Do not include markdown fences or text outside
the JSON.
```

```
# INPUTS
## Agent Skill
{skill}

## Persona
{persona}
```

Prompt for Guideline Generation Back to ToC

```
You are generating an execution guideline for a terminal-agent SFT data synthesis
pipeline. The guideline guides the agent through the task step-by-step.

# REQUIREMENTS

Each step must be:
- Actionable: include the specific command, file path, or edit
- Verifiable: say how to confirm it worked
- Ordered: respect task dependencies

Extract concrete steps from the Agent Skill's SOP when available. Prefix critical
caveats with "IMPORTANT:" or "WARNING:". Do not leak final file content or
complete solutions.

Each step should be written as a single string in this style:
"Step N: <action> -- <exact command or edit> -- <verification or warning>"

# GOOD EXAMPLES

- "Step 1: Benchmark current state -- Run 'du -sh public/images/' and 'find public
/images -type f -name "*.png" | wc -l' to measure baseline."
- "Step 2: Update schema.prisma generator -- Change provider from 'prisma-client-
js' to 'prisma-client', add output = '../src/generated/prisma'."
- "Step 3: Run 'bunx prisma generate' and verify generated client appears at
packages/db/src/generated/prisma/."
- "Step 4: IMPORTANT: Verify prisma/client remains in package.json dependencies (
do NOT remove -- needed at runtime)."

Avoid vague steps like "Inspect the project" or "Fix any errors" that lack
specific commands or targets.

# OUTPUT FORMAT

Return STRICTLY as a JSON object:
{"guideline": ["Step 1: ...", "Step 2: ..."]}

Do not include markdown fences, explanation, or any text outside the JSON.

# INPUTS

## Skill
{skill}

## Core Goal
{core_goal_json}
```

Prompt for Task Quality Judgment Back to ToC

```
You are an expert quality evaluator for bash terminal tasks. Your job is to
evaluate the quality of the task specification across five dimensions (0-5 each)
and justify each score.

# INPUT

## Persona (the user role that motivates this task)
{persona_text}

## Skill (the technical capability being exercised)
{skill_text}

## Generated Goal (the full task specification to evaluate)
{goal_json}

# IMPORTANT JUDGING PRINCIPLES

- Judge for realism, task quality, and training value -- not for rigid formatting.
- A concise, natural instruction can receive a high score.
- Do NOT require a fixed opener, numbered list, or a "Requirements:" section.
- High scores should reflect realistic task framing, clear task intent, strong
skill alignment, solvability, and verifiability.

# EVALUATION DIMENSIONS

## Dim 1 -- Instruction Quality (instruction_quality)
Is the instruction realistic, skill-aligned, actionable, and clear enough for the
agent to start the right kind of work?
  5 - Realistic, strongly skill-aligned, clear goal and success condition.
  4 - Strong and usable; slightly synthetic or less precise than ideal.
  3 - Executable but generic or partially under-specified.
  2 - Weakly aligned with the skill, artificial, or too vague.
  1 - Barely actionable or missing the task's core objective.
  0 - Self-contradictory, incoherent, or not a workable instruction.

## Dim 2 -- Solvable & Closed-World (solvable_closed_world)
Can the task be completed entirely inside an isolated container without internet
access, private credentials, or privileged operations?
  5 - Strictly closed-world; all dependencies and data are in the blueprint.
  4 - Essentially closed-world; a few installable deps; no external data needed.
  3 - Likely closed-world; some unclear details can reasonably be filled in.
  2 - Significant external-dependency risk or critical data gaps.
  1 - Highly unsolvable -- critical inputs missing or strong external dependencies
.
  0 - Definitively unsolvable or contradicts environment constraints.

## Dim 3 -- Blueprint Completeness (blueprint_completeness)
Is the environment blueprint specific enough to construct the environment
deterministically?
  5 - All 5 categories complete: filesystem, data schema, deps, entrypoints,
validation.
  4 - At least 4 of 5 categories covered; minor details missing.
  3 - Buildable but 1-2 of schema/versions/validation are missing.
  2 - Too abstract; builder must guess extensively.
  1 - Nearly unbuildable environment.
  0 - Empty blueprint or clearly inconsistent with the instruction.

## Dim 4 -- Guideline Quality (guideline_quality)
Does the guideline drive a realistic terminal trace with correct ordering,
appropriate granularity, checkpoints, and SOP coverage -- without spoiling the
final answer?
```

```
  5 - Captures skill workflow well; sensible ordering, useful checkpoints, no
spoilers.
  4 - Strong and useful; slightly coarse or sparse on checkpoints.
  3 - Usable but generic; some workflow value, incomplete ordering or checkpoints.
  2 - Too abstract to guide execution, or too script-like and overprescriptive.
  1 - Does not match instruction/blueprint, or steps are disordered.
  0 - No guideline or completely unusable.

## Dim 5 -- Evaluation Criteria Quality (evaluation_criteria_quality)
Do the evaluation criteria clearly define how task completion will be assessed?
Can they be translated into reliable black-box pytest checks without guessing?
  5 - Explicit, outcome-focused, concrete, and directly translatable to pytest.
  4 - Mostly strong; minor ambiguity but overall reliably judgeable.
  3 - Partially useful but incomplete or vague on at least one important condition
.
  2 - Weak or underspecified; difficult to translate into stable assertions.
  1 - Mostly unusable; major conditions missing, vague, or subjective.
  0 - Absent, contradictory, or cannot assess task completion at all.

# OUTPUT FORMAT

Respond with ONLY a JSON object -- no markdown fences, no explanation outside the
JSON.

{
  "instruction_quality":         {"score": <0-5>, "reason": "..."},
  "solvable_closed_world":       {"score": <0-5>, "reason": "..."},
  "blueprint_completeness":      {"score": <0-5>, "reason": "..."},
  "guideline_quality":           {"score": <0-5>, "reason": "..."},
  "evaluation_criteria_quality": {"score": <0-5>, "reason": "..."}
}
```

Prompt for File Generation (`llm_direct` mode) Back to ToC

```
You are an expert environment builder for AI coding agent benchmarks.

Given a task description, environment blueprint, and a target file to generate,
create the actual file content for that specific file.

# Task Instruction
{instruction}

# Environment Blueprint
{blueprint}

# Target File to Generate
{target_file}

# Previously Generated Files
{previous_files}

# Instructions
Generate the content for the target file based on:
1. The file's description and filepath
2. The task instruction (what the agent needs to accomplish)
3. The overall environment context (system requirements, pre-setup)
4. Previously generated files (maintain consistency with imports, APIs, etc.)

You MUST end your response with a ```json fenced code block -- this block is
required.
```

```
The JSON object must have exactly two fields:
- "filepath": the exact path from the target file
- "content": the full file content as a string
```

Prompt for File Generation (`local_tool` mode) Back to ToC

```
[System]
You are a specialized local artifact generation agent running inside a Linux
sandbox.

You must respond with exactly one required function call and nothing else.

Available tool:
  python: {"target_filepath": string, "code": string, "timeout_sec": integer (
optional)}

Rules:
- Only use the python tool.
- target_filepath must exactly match the requested target path.
- Use Python to create or repair the target artifact.
- Keep working until the tool observation reports a valid artifact.

[User]
Target file: {target_filepath}
Description: {file_description}

Task context (for reference only -- do NOT solve the whole task, only create the
target artifact):
{instruction_summary}
```

Prompt for File Generation (`remote_fetch` mode) Back to ToC

```
[System]
You are a specialized stateless remote fetch agent running inside a Linux sandbox.

You must respond with exactly one required function call and nothing else.

Available tools:
  web_search:    {"query": string, "top_k": integer (optional), "domain_hint":
string (optional)}
  fetch_page:    {"url": string, "mode": "http"|"dynamic"|"stealth", "timeout_ms":
 integer (optional)}
  download_file: {"url": string, "save_as": string, "timeout_ms": integer (
optional)}

Rules:
- Stateless: do not assume browser state or session reuse between actions.
- save_as must exactly match the requested target path.
- Use search, fetch, and download as separate steps.
- Keep working until the downloaded artifact validates successfully.

[User]
Target file: {target_filepath}
Description: {file_description}

Task context (for reference only -- do NOT solve the whole task, only fetch the
target artifact):
```

```
{instruction_summary}
```

Prompt for File Verification Back to ToC

```
[System]
You are a file verifier agent running inside a Linux sandbox.

Your role is verification-only.

Rules:
- You are not a task-solving agent.
- Your scope is limited to the declared filepaths and their file specifications.
- Use shell commands only to inspect the provided files.
- Do NOT modify files, use network access, or evaluate runtime state.
- Prefer status="continue" whenever evidence is incomplete.
- Respond with exactly one JSON object per turn.
- Continue mode must use:  analysis, status="continue", commands
- Finalize mode must use:  analysis, status="finalize", result
- Finalize result keys:    overall_verdict, file_findings, global_findings
- overall_verdict must be exactly "pass" or "fail"
- Each item in file_findings:  {filepath, reason, repair_instructions}
- Each item in global_findings: {reason, primary_owner, repair_instructions}
  (primary_owner: "llm_direct" | "specialized" | "unattributed")
- If overall_verdict is "pass", both findings arrays must be empty.
- If overall_verdict is "fail", at least one finding must be present.
- repair_instructions must be concrete and actionable for the repair agent.

Valid response examples:

  Continue:
  {
    "analysis": "Need to check file content.",
    "status": "continue",
    "commands": ["cat /app/data.csv"]
  }

  Finalize (pass):
  {
    "analysis": "All files match their specifications.",
    "status": "finalize",
    "result": {"overall_verdict": "pass", "file_findings": [], "global_findings":
[]}
  }

  Finalize (fail):
  {
    "analysis": "Found a mismatch in /app/example.txt.",
    "status": "finalize",
    "result": {
      "overall_verdict": "fail",
      "file_findings": [{
        "filepath": "/app/example.txt",
        "reason": "Content does not match the declared format.",
        "repair_instructions": "Rewrite /app/example.txt to match the declared
format."
      }],
      "global_findings": []
    }
  }
```

```
[User]
Task instruction:
{instruction}

Verification scope:
- Only verify the declared filepaths below.
- Ignore runtime state and broader task completion.

Files to verify:
{file_lines}

File-only verification objective:
- Verify each declared file against its description and the task instruction.
- Only use global_findings for still-file-scoped cross-file issues.
- If you fail any file or emit any global finding, include detailed
repair_instructions.

Initial workspace tree (initial hint only):
{workspace_tree}
```

Prompt for Environment Setup (env build) Back to ToC

```
You are an expert DevOps engineer setting up a Linux sandbox environment for an AI
 coding agent. Your job is ONLY to prepare the environment -- install system
packages, language runtimes, and libraries. You must NOT write application code or
 execute the task itself.

# Base Environment (already provisioned -- do NOT reinstall these)
- OS: Debian 13 (trixie), kernel 6.1.x, x86_64
- User: user (uid=1000), non-root. Use sudo for privileged operations.
- Working directory: /app
- Pre-installed: Python 3.12, pip 25.x, Node.js 20.x, npm 10.x, Java 8 (OpenJDK),
gcc/g++ 14.x, make 4.x, git 2.x, curl 8.x, wget 1.x
- Harbor directories (perms=777): /logs/agent, /logs/verifier, /tests, /app, /
output, /solution

# Task Instruction (for context only -- do NOT execute this task)
{instruction}

# Environment Blueprint
{blueprint}

# Pre-seeded Assets
The following assets have already been written into the sandbox. Reference them
directly (e.g. `pip install -r /app/requirements.txt`). Do NOT recreate or
overwrite them.

{pre_seeded_files}

# Instructions
Generate shell commands to ONLY:
1. Execute each step listed in setup_steps, in order
2. Create necessary directories
3. Install from any dependency manifest in the pre-seeded assets
4. Start or configure required services, environment variables, and permissions

IMPORTANT:
- HIGHEST PRIORITY: Never create, download, or overwrite any pre-seeded asset path
- Scope is limited to environment configuration only -- no application source code
- Do NOT execute the task (e.g. do NOT run `node index.js` or `python main.py`)
```

```
- When using sudo with network commands, ALWAYS use `sudo -E` to preserve proxy
env vars

End your response with a ```bash fenced code block containing the commands.
```

Prompt for Environment Verification (env verify) Back to ToC

```
You are an expert DevOps engineer verifying that a Linux sandbox environment has
been correctly set up. A setup script has already been executed successfully (exit
 code 0). Your job is to generate verification commands that check whether the
environment is truly ready for the task.

# Base Environment (already provisioned)
- OS: Debian 13 (trixie), kernel 6.1.x, x86_64; Working directory: /app
- Pre-installed: Python 3.12, Node.js 20.x, Java 8, gcc/g++ 14.x, make, git, curl,
 wget

# Task Instruction (for context -- do NOT execute this task)
{instruction}

# Environment Blueprint
{blueprint}

# Pre-seeded Assets
{pre_seeded_files}

# Setup Script That Was Executed
{setup_script}

# Instructions
Generate shell commands that ONLY verify the environment is ready:
1. Check that required packages/libraries are importable (e.g. `python3 -c "import
 flask"`, `node -e "require('express')"`)
2. Check that required CLI tools are available (e.g. `which gcc`, `java -version`)
3. Check that required directories exist
4. Check that dependency versions meet requirements if specific versions were
requested

IMPORTANT:
- Do NOT install anything -- only verify
- Do NOT execute the task itself or write application code
- The script does NOT use `set -e` -- all commands run even if some fail

End your response with a ```bash fenced code block containing the verification
commands.
```

Prompt for Environment Repair (env repair) Back to ToC

```
You are an expert DevOps engineer fixing a failed sandbox environment setup. Your
job is ONLY to prepare the environment -- install packages and configure services.
 You must NOT write application code or execute the task.

# Base Environment (already provisioned -- do NOT reinstall these)
- OS: Debian 13 (trixie), kernel 6.1.x, x86_64; Working directory: /app
- Pre-installed: Python 3.12, Node.js 20.x, Java 8, gcc/g++ 14.x, make, git, curl,
 wget
```

```
# Task Instruction (for context only -- do NOT execute this task)
{instruction}

# Environment Blueprint
{blueprint}

# Pre-seeded Assets
The following assets will be re-written before your commands run. Do NOT recreate
them.

{pre_seeded_files}

# Previous Setup Commands
{commands}

# Command Failure Details
{errors}

# Instructions
The previous attempt failed. The next attempt runs in a FRESH sandbox -- do NOT
rely on any side effects from the previous attempt.

Generate a FULL corrected list of setup commands (not a patch/delta). All commands
 will be combined into a single bash script with ‘set -euxo pipefail‘ and executed
 once.

IMPORTANT:
- HIGHEST PRIORITY: Never create, download, or overwrite any pre-seeded asset path
- Scope is limited to environment configuration only -- no application source code
- Do NOT execute the task (e.g. do NOT run ‘node index.js‘ or ‘python main.py‘)
- When using sudo with network commands, ALWAYS use ‘sudo -E‘ to preserve proxy
env vars

End your response with a ‘‘‘bash fenced code block containing the full corrected
commands.
```

Prompt for pytest Verifier Generation Back to ToC

```
You are generating a pytest-based task verifier for a terminal benchmark task.

Your verifier will run AFTER the agent has finished executing the task, inside the
 SAME sandbox state. Do not assume the environment is restarted. Verify ONLY
whether the task was actually completed -- do not inspect the agent’s messages or
tool traces.

You must output a JSON object with this exact schema:
{
  "system_packages": ["..."],
  "python_packages": ["..."],
  "helper_files": [{"path": "tests/filename.ext", "content": "..."}],
  "test_outputs_py": "..."
}

Rules:
1. The verifier must use pytest.
2. ‘test_outputs_py‘ must contain valid Python source code.
3. Tests must be black-box.
4. Prefer verifying via files, command behavior, local HTTP behavior, or
deterministic end-to-end examples.
5. Tests must be deterministic and self-contained.
```

```
6. Avoid network access unless the task explicitly requires localhost access.
7. Only add packages required by the verifier itself.
8. Put any extra test assets into ‘helper_files‘.
9. Do not generate explanations outside the JSON.

# Instruction
{instruction}

# Evaluation Criteria
{evaluation_criteria}

# Target Output File
{target_output_file}

# Initial Text Files (ACTUAL CONTENT -- already present in environment)
IMPORTANT: These files already exist when the agent starts. DO NOT test for them.
They are provided as context only.

{initial_text_files}

# Initial Asset Files (METADATA ONLY -- already present in environment)
IMPORTANT: These files exist but only metadata is provided. Do NOT assume actual
content.

{initial_asset_files}

# Validated Environment File Paths
{validated_filepaths}

CRITICAL RULES:
- DO NOT test whether initial files exist (guaranteed to exist)
- DO NOT test initial file content (already validated)
- ONLY test whether the evaluation criteria are met
- Test what the AGENT creates, not what the environment provides
```

System Prompt for Terminus2 Agent (JSON Format) Back to ToC

```
You are an AI assistant tasked with solving command-line tasks in a Linux
environment. You will be given a task description and the output from previously
executed commands. Your goal is to solve the task by providing batches of shell
commands.

Format your response as JSON with the following structure:

{
  "analysis": "Analyze the current state based on the terminal output. What has
been accomplished? What still needs to be done?",
  "plan": "Describe your plan for the next steps. What commands will you run and
why? Be specific about what you expect each command to accomplish.",
  "commands": [
    {"keystrokes": "ls -la\n",     "duration": 0.1},
    {"keystrokes": "cd project\n", "duration": 0.1}
  ],
  "task_complete": true
}

Required fields:
- "analysis":  Your analysis of the current situation
- "plan":      Your plan for the next steps
- "commands":  Array of command objects to execute
```

```
Optional fields:
- "task_complete": Boolean indicating if the task is complete (defaults to false)

Command object structure:
- "keystrokes": Exact keystrokes to send to the terminal (required, must end with
\n)
- "duration":   Seconds to wait before executing the next command (default: 1.0)

IMPORTANT: The text inside "keystrokes" is sent verbatim to the terminal:
- Every command must end with \n or it will not execute.
- Special key sequences use tmux-style escape sequences: C-c for Ctrl+C, C-d for
Ctrl+D.

The "duration" attribute controls how long to wait for output before proceeding:
- Immediate commands (cd, ls, echo, cat):  0.1 s
- Standard commands (gcc, find, rustc):    1.0 s
- Slow commands (make, wget, long scripts): set an appropriate value as needed
- Prefer shorter durations; poll by sending {"keystrokes": "", "duration": 10.0}
rather than blocking.

Important notes:
- Send keystrokes exactly as written; do not add extra whitespace unless intended.
- Extra text outside the JSON generates warnings but is tolerated.
- The JSON must be valid -- escape quotes and special characters inside strings.
- An empty "commands" array is valid when you need to observe without acting.

Task Description:
{instruction}

Current terminal state:
{terminal_state}
```

Prompt for Environment Quality Evaluation Back to ToC

```
You are evaluating a Harbor-format terminal benchmark task.

Task directory: {task_dir}

Please read the task contents (instruction.md, environment/ files,
tests/test_outputs.py or tests/test_final_state.py) and score it on
four dimensions. Each dimension is scored 1-3:
  1 = poor   2 = acceptable   3 = good

Important: base your scores ONLY on the file contents you read.
Ignore the directory path and any dataset name it may imply --
treat every task as anonymous.

Scoring dimensions:

1. terminal_nativeness
   Does the task genuinely require terminal CLI operations (compilers,
   package managers, system commands, build tools, network tools)?
   High score = real CLI toolchain required.
   Low score  = just writing files or trivial echo commands.

2. env_task_consistency
   Do the pre-placed environment files and setup precisely match what
   the instruction requires?
   High score = environment provides exactly the right scaffolding,
```

```
                no excess or gap.
   Low score  = environment is empty, irrelevant, or contradicts
                the task.

3. env_quality
   Are the environment files (Dockerfile, setup.sh, initial files)
   well-formed and credible?
   High score = files are realistic, complete, and executable.
   Low score  = files are fabricated stubs, have obvious bugs, or
                are missing critical dependencies.

4. verifier_robustness
   Do the pytest assertions in tests/ accurately distinguish
   task-complete from task-incomplete?
   High score = assertions are specific, cover all key acceptance
                criteria from the instruction, low false-positive risk.
   Low score  = only checks file existence, or assertions are
                unrelated to the instruction requirements.

Important: if a dimension is not applicable (e.g. no tests/ directory
exists), score it 1.

Respond with ONLY a JSON object, no other text:
{
  "terminal_nativeness":         <1|2|3>,
  "terminal_nativeness_reason":  "<one sentence>",
  "env_task_consistency":        <1|2|3>,
  "env_task_consistency_reason": "<one sentence>",
  "env_quality":                 <1|2|3>,
  "env_quality_reason":          "<one sentence>",
  "verifier_robustness":         <1|2|3>,
  "verifier_robustness_reason":  "<one sentence>"
}
```

E DATA EXAMPLES

To complement the textual pipeline description in Section 3, this section presents *three stage-focused examples* drawn from the **Terminal-Atlas** dataset. Rather than compressing an end-to-end rollout for every task, each example zooms into a single synthesis stage and displays its inputs and outputs verbatim so the reader can see the concrete shape of each artifact:

1. **Example E.1** illustrates **Task Generation** (§3.2): from a single (Skill S, Persona U) pair to the synthesized quadruple $(\mathcal{I}, \mathcal{E}, \mathcal{V}, \mathcal{G})$. We show all six artifacts verbatim.

2. **Example E.2** illustrates **Environment Building** (§3.3): a three-file blueprint passes through three GVR sub-agents to produce the initial files F, setup script B_{env}, and pytest verifier T_{test}.

3. **Example E.3** illustrates **Trajectory Collection** (§3.4): a multi-turn teacher-model rollout with verbatim `analysis`/`plan`/`commands`/`observation` for four representative steps.

E.1 EXAMPLE 1: TASK GENERATION — ELF BINARY PARSING (ASTROPHYSICS)

BACK TO TOC

</> Stage-Focused Example 1 | Stage: Task Generation (§3.2)

❶ Input S — Agent Skill

```
---
name: elf-binary-analysis
description: >
  Analyze ELF binary files for reverse engineering, security research, and
  exploitation. Use this skill whenever the user needs to understand ELF
  structure, analyze program headers, section headers, symbols, relocations,
  GOT/PLT, or identify binary protections like RELRO, stack canaries, and PIE.
  Trigger on any request involving ELF files, binary analysis, readelf output
  interpretation, or exploitation reconnaissance.
tags: ['reverse-engineering', 'binary-analysis', 'elf']
source: github.com/abelrguezr/hacktricks-skills
task_slug: binary-exploitation/elf-tricks
route_label: segment-buffer-allocation
---
```

❷ Input U — User Persona (FinePersonas)

```
An astrophysicist or a science writer covering recent developments in
astrophysics, particularly focused on gamma-ray bursts, gravitational
waves, and astronomical research.

labels: [Science, Academia, Space Exploration]
```

❸ Output $\mathcal{I}$ — Task Instruction

As an astrophysicist, I have a compiled binary tool `grb_processor.elf` used for gamma-ray burst data analysis. To validate its memory mapping and extract calibration constants, write a Node.js script that parses the ELF file. Your script should identify all `PT_LOAD` segments, allocate buffers with zero-padding for BSS sections, and extract 32-bit integer values from 4-byte aligned virtual addresses. Output a JSON object where keys are virtual addresses as decimal strings and values are the 32-bit integers. Save the JSON to `/app/memory_map.json`.

❹ Output $\mathcal{E}$ — Environment Blueprint

Initial Files: /app/grb_processor.elf • mode: local_tool

ELF 64-bit LSB executable binary file. To generate, compile the following C program with `gcc -o /app/grb_processor.elf /tmp/source.c`. C source code:

```
#include <stdint.h>
int calibration = 271828;          // Euler's number * 100000
int32_t event_sequence[100] = {0,1,2,...,99};
char raw_data_buffer[4096];        // uninitialized (BSS)
int main() { return 0; }
```

After compilation, `readelf -l` shows two LOAD segments: segment 1 with `VirtAddr=0x400000`, `FileSiz=0x200`, `MemSiz=0x400`; segment 2 with `VirtAddr=0x400400`, `FileSiz=0x0`, `MemSiz=0x1000`. Total MemSiz for extraction is `0x1400` bytes (5120 bytes), expecting 1280 key-value pairs when extracting 4-byte aligned values (5120/4=1280). The initialized data includes `calibration` at the start of segment 1.

Setup Steps

- Install `binutils` package via `apt-get` to ensure `readelf` command is available for ELF inspection.

❺ Output $\mathcal{V}$ — Evaluation Scope

- ✓ File `/app/memory_map.json` exists and contains valid JSON.
- ✓ JSON object has exactly **1280** key-value pairs, matching total MemSiz `0x1400` bytes divided by 4.
- ✓ Key `'4194304'` (decimal for virtual address `0x400000`) has value **271828**.
- ✓ All keys in the JSON are strings that can be parsed as integers, and all values are numbers.

⑥ Output $\mathcal{G}$ — Execution Guideline

Step 1. Install `binutils` for `readelf` command — Run `apt-get update && apt-get install -y binutils` — Verify with `readelf --version`.

Step 2. Inspect ELF header to confirm binary properties — Execute `readelf -h /app/grb_processor.elf` — Confirm output shows `Class: ELF64` and little-endian.

Step 3. Identify LOAD segments for memory mapping — Run `readelf -l /app/grb_processor.elf` — Note VirtAddr, FileSiz, and MemSiz for each PT_LOAD segment; verify two segments.

Step 4. Verify Node.js runtime — Run `node --version` — Ensure version is v14.x or higher for BigInt support.

Step 5. **[IMPORTANT]** Create Node.js script with BSS handling — Write `/app/extract.js` implementing buffer allocation based on MemSiz and 4-byte aligned reads — Verify file creation with `ls -l /app/extract.js`.

Step 6. Run extraction script — `node /app/extract.js /app/grb_processor.elf > /app/memory_map.json 2>&1` — Check exit code with `echo $?`; it should be 0.

Step 7. Validate JSON output format — `head -c 200 /app/memory_map.json` — Confirm it starts with { and appears as valid JSON.

Step 8. Final verification — use a Node.js one-liner to parse the JSON, verify key count is 1280 and value at `'4194304'` is 271828.

E.2 Example 2: Environment Building — Multi-Format Data Merger

Back to ToC

Stage-Focused Example 2 | Stage: Environment Building (§3.3)

❶ Input — Environment Blueprint $\mathcal{E}_{\text{files}}$

Initial Files: (1) /app/quiz_data.json • mode: llm_direct

JSON array of objects with keys: `student_id` (int), `topic` (string, either `'algebra'` or `'scientific_notation'`), `score` (float or null), `timestamp` (ISO8601 string). 50 objects. Example objects:

```
{"student_id": 101, "topic": "algebra",             "score": 88.5, ...}
{"student_id": 102, "topic": "scientific_notation", "score": 85.5, ...}
```

Student_id 103 has a null score for algebra (missing data). Student_id 101 appears twice with different scores for algebra (88.5) and scientific_notation (92.0) — agent must map these to separate fields.

Initial Files: (2) /app/gradebook.csv • mode: llm_direct

CSV file with columns: `id` (int), `name` (string), `algebra_grade` (float), `sci_not_grade` (float), `updated` (ISO8601 string). 50 rows. Example rows:

```
101,Alice Johnson,90.0,88.5,2024-10-17T09:00:00
102,Bob Smith,76.5,89.0,2024-10-17T09:05:00
```

Row for `id 104` has `algebra_grade=-1.0` (sentinel for missing data). Column `id` corresponds to `student_id` in other sources.

Initial Files: (3) /app/project_scores.parquet • mode: local_tool

Parquet file with columns: `student_id` (int), `project_type` (string, `'algebra'` or `'scientific_notation'`), `project_score` (int or string), `date` (string in YYYY-MM-DD format). 30 rows. Example: `student_id=105, project_type='algebra', project_score=95, date='2024-10-18'`. Three rows have `project_score` as string `'N/A'` instead of integer (type inconsistency to handle). Scores are out of 100.

Setup Steps

- Install `pandas` and `pyarrow` via `pip` to handle CSV, JSON, and Parquet file operations.

❷ Artifact F — Generated Initial Files (head of each)

Routed to $\mathcal{A}_{llm_direct}$ — (1) `/app/quiz_data.json`:

```
[
  {"student_id": 101, "topic": "algebra",             "score": 88.5, "timestamp": "2024-10-15T10
     :30:00"},
  {"student_id": 101, "topic": "scientific_notation", "score": 92.0, "timestamp": "2024-10-15T11
     :45:00"},
  {"student_id": 102, "topic": "scientific_notation", "score": 85.5, "timestamp": "2024-10-16T11
     :00:00"},
  {"student_id": 103, "topic": "algebra",             "score": null, "timestamp": "2024-10-16T11
     :15:00"},
  {"student_id": 104, "topic": "algebra",             "score": 77.0, "timestamp": "2024-10-15T09
     :00:00"},
  {"student_id": 104, "topic": "scientific_notation", "score": 82.5, "timestamp": "2024-10-15T10
     :00:00"},
  ... (44 more entries)
]
```

Routed to $\mathcal{A}_{llm_direct}$ — (2) `/app/gradebook.csv`:

```
id,name,algebra_grade,sci_not_grade,updated
101,Alice Johnson,90.0,88.5,2024-10-17T09:00:00
102,Bob Smith,76.5,89.0,2024-10-17T09:05:00
103,Charlie Brown,82.0,81.5,2024-10-17T09:10:00
104,David Williams,-1.0,90.0,2024-10-17T09:15:00
105,Eva Martinez,91.0,89.5,2024-10-17T09:20:00
106,Frank Miller,85.0,84.0,2024-10-17T09:25:00
... (44 more rows)
```

Routed to $\mathcal{A}_{local_tool}$ — (3) `/app/project_scores.parquet`:

```
>>> import pandas as pd; df = pd.read_parquet('/app/project_scores.parquet')
>>> df.dtypes
student_id        int64
project_type       str
project_score      str      <- type inconsistency: some rows hold 'N/A'
date               str
>>> df.head(8)
 student_id         project_type  project_score         date
        101              algebra             70   2024-10-16
        102  scientific_notation             81   2024-10-13
        103              algebra             95   2024-10-22
        104  scientific_notation             91   2024-10-24
        105              algebra             95   2024-10-18
        106  scientific_notation            N/A   2024-10-17
        107              algebra             94   2024-10-22
        108  scientific_notation             86   2024-10-14
>>> df.shape
(30, 4)
```

❸ Artifact B_{env} — Generated `setup.sh`

```
#!/usr/bin/env bash
set -euxo pipefail
cd /app

sudo -E pip install pandas pyarrow
```

❹ Artifact T_{test} — Generated pytest Verifier

```python
import pandas as pd, json, os, pytest
import numpy as np

def test_output_files_exist():
    assert os.path.exists("/app/merged_students.parquet")
    assert os.path.exists("/app/conflicts.json")

def test_merged_parquet_structure():
    df = pd.read_parquet("/app/merged_students.parquet")
    expected_cols = ['student_id', 'name', 'algebra_score', 'sci_not_score']
    assert list(df.columns) == expected_cols
    assert pd.api.types.is_integer_dtype(df['student_id'])
    assert pd.api.types.is_float_dtype(df['algebra_score'])
    assert pd.api.types.is_float_dtype(df['sci_not_score'])
    assert df['student_id'].dtype == 'int64'

def test_row_count():
    quiz      = pd.read_json   ("/app/quiz_data.json")
    gradebook = pd.read_csv    ("/app/gradebook.csv")
    project   = pd.read_parquet("/app/project_scores.parquet")
    all_ids   = set(quiz['student_id']) | set(gradebook['id']) | set(project['student_id'])
    df        = pd.read_parquet("/app/merged_students.parquet")
    assert len(df) == len(all_ids)
    assert set(df['student_id']) == all_ids

def test_priority_and_data_logic():
    df = pd.read_parquet("/app/merged_students.parquet")
    # Student 101: Quiz(alg=88.5, sci=92.0) beats Gradebook(alg=90.0, sci=88.5) by priority.
    s101 = df[df['student_id'] == 101].iloc[0]
    assert s101['algebra_score'] == 88.5
    assert s101['sci_not_score'] == 92.0
    # Student 103: Quiz(alg=null) -> fallback to Gradebook(alg=82.0).
    s103 = df[df['student_id'] == 103].iloc[0]
    assert s103['algebra_score'] == 82.0
    # Student 104: Quiz(alg=77.0) beats Gradebook(alg=-1.0 sentinel).
    s104 = df[df['student_id'] == 104].iloc[0]
    assert s104['algebra_score'] == 77.0
    assert s101['name'] == "Alice Johnson"
    # Student only in Gradebook.
    s149 = df[df['student_id'] == 149].iloc[0]
    assert s149['name'] == "Will Morgan"
    assert s149['algebra_score'] == 82.0

def test_conflicts_json_content():
    with open("/app/conflicts.json", 'r') as f:
        data = json.load(f)
    assert "total_conflicts" in data
    assert "conflicts"       in data
    assert isinstance(data['conflicts'], list)
    # Student 101 had a conflict on algebra_score (Quiz 88.5 vs Gradebook 90.0).
    for conflict in data['conflicts']:
        if conflict['student_id'] == 101 and conflict['field'] == 'algebra_score':
            assert conflict['selected_value'] == 88.5
            assert 'quiz'      in conflict['values']
            assert 'gradebook' in conflict['values']
    # General priority check: selected_value must match highest-priority non-missing source.
    for conflict in data['conflicts']:
        vals, selected = conflict['values'], conflict['selected_value']
        priority_val = None
        for src in ['quiz', 'gradebook', 'project']:
            val = vals.get(src)
            if val is not None and val != -1.0 and val != 'N/A' \
               and not (isinstance(val, float) and np.isnan(val)):
                priority_val = val
                break
        if priority_val is not None:
            assert float(selected) == float(priority_val)
```

E.3 Example 3: Trajectory Collection — Video OCR Extraction Back to ToC

</> Stage-Focused Example 3 | Stage: Trajectory Collection (§3.4)

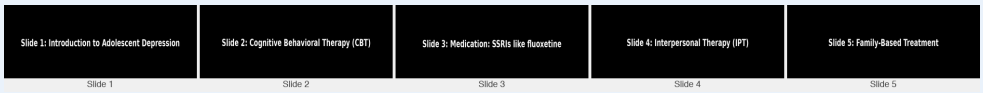

Figure 7: **Representative frames from the video-OCR trajectory example.** The agent extracts treatment methods from the video and writes a list that is checked by the verifier.

Table 21: **Video-OCR trajectory verification:** all six checks pass in 11 steps (reward 1.0).

Test Case	Status	Metric	Value
`test_treatment_list_exists`	✓ **PASSED**	file present	`/app/treatment_list.txt`
`test_treatment_list_min_lines`	✓ **PASSED**	line count	≥ 4 (actual: 4)
`test_cbt_present`	✓ **PASSED**	CBT keyword	found
`test_ssri_present`	✓ **PASSED**	SSRI keyword	found
`test_ipt_present`	✓ **PASSED**	IPT keyword	found
`test_family_based_present`	✓ **PASSED**	FBT keyword	found
Reward		**1.0**	(6/6 passed, 11 steps)

❶ Step 2 (of 11) — Install `ffmpeg`

analysis We're at /app with a fresh terminal. The task is to process a video file using OCR to extract treatment methods and save a cleaned list. A checklist is provided with 10 steps. We need to start with step 1: update package lists.
plan Execute step 1: run apt-get update to update package lists. Then proceed to step 2: install ffmpeg.
commands

```
apt-get update
apt-get install -y ffmpeg
```

observation

```
/app $ apt-get update
Hit:1 http://mirrors.tuna.tsinghua.edu.cn/debian trixie InRelease
Reading package lists... Done
/app $ apt-get install -y ffmpeg
Reading package lists... Done
Building dependency tree... Done
ffmpeg is already the newest version (7:7.1.3-0+deb13u1).
0 upgraded, 0 newly installed, 0 to remove and 79 not upgraded.
```

❷ Step 4 (of 11) — Extract video frames with `ffmpeg`

analysis All prerequisites installed. Proceed to step 6: extract video frames. Create /app/frames and run ffmpeg at `fps=1`.
plan `mkdir -p /app/frames`, then ffmpeg with `fps=1`. Verify with `find ... | wc -l`.
commands

```
mkdir -p /app/frames
ffmpeg -i /app/education_video.mp4 -vf "fps=1" /app/frames/frame_%04d.png
find /app/frames -name "*.png" | wc -l
```

observation

```
/app $ ffmpeg -i /app/education_video.mp4 -vf "fps=1" /app/frames/frame_%04d.png
ffmpeg version 7.1.3-0+deb13u1 Copyright (c) 2000-2025 the FFmpeg developers
. . .
Input #0, mov,mp4,m4a,3gp,3g2,mj2, from '/app/education_video.mp4':
Duration: 00:00:15.00, start: 0.000000, bitrate: 21 kb/s
Stream #0:0: Video: h264 (High) (avc1/0x31637661), yuv420p, 1280x720, 20 kb/s, 1 fps
Stream mapping:
Stream #0:0 -> #0:0 (h264 (native) -> png (native))
Output #0, image2, to '/app/frames/frame_%04d.png':
Stream #0:0: Video: png, rgb24, 1280x720, q=2-31, 1 fps
frame=  15 fps=0.0 q=-0.0 Lsize=N/A time=00:00:15.00 bitrate=N/A speed= 157x
/app $ find /app/frames -name "*.png" | wc -l
15
```

❸ Step 5 (of 11) — Author `ocr.py`, run OCR

analysis Frame extraction successful: 15 frames. Now step 7: create OCR script and run it.
plan Create `/app/ocr.py` that globs frames, runs `pytesseract` on each, and appends to `all_text.txt`.
commands

```
import cv2, pytesseract, glob

frames_dir  = '/app/frames'
output_file = '/app/all_text.txt'

frame_paths = sorted(glob.glob(f'{frames_dir}/frame_*.png'))
all_text = []
for path in frame_paths:
    img = cv2.imread(path)
    if img is None:
        continue
    text = pytesseract.image_to_string(img)
    all_text.append(text)

with open(output_file, 'w') as f:
    f.write('\n'.join(all_text))
```

observation

```
/app $ python3 /app/ocr.py
/app $ wc -l /app/all_text.txt
29 /app/all_text.txt
/app $ head -c 200 /app/all_text.txt
Slide 1: Introduction to Adolescent Depression
Slide 1: Introduction to Adolescent Depression
Slide 2: Cognitive Behavioral Therapy (CBT)
Slide 2: Co. . .
```

❹ Step 8 (of 11) — Refine `parse.py` to filter out the intro slide

analysis Previous parse kept 5 unique lines, including the non-treatment *Introduction to Adolescent Depression*. Filter by treatment-related keywords.

plan Rewrite `/app/parse.py` with a keyword whitelist [`therapy`, `treatment`, `medication`, `cbt`, `ssri`, `antidepressant`], dedup, re-run, verify ≥ 4 lines and CBT presence.

commands

```
input_file  = '/app/all_text.txt'
output_file = '/app/treatment_list.txt'
keywords    = ['therapy', 'treatment', 'medication', 'cbt', 'ssri', 'antidepressant']

with open(input_file, 'r') as f:
    lines = f.read().splitlines()

treatments = set()
for line in lines:
    line = line.strip()
    if not line:
        continue
    if line.startswith('Slide'):
        parts = line.split(':', 1)
        if len(parts) == 2:
            treatment = parts[1].strip()
            if treatment and any(kw in treatment.lower() for kw in keywords):
                treatments.add(treatment)

with open(output_file, 'w') as f:
    for t in sorted(treatments):
        f.write(t + '\n')
```

observation

```
/app $ python3 /app/parse.py
Extracted 4 unique treatments.
/app $ cat /app/treatment_list.txt
Cognitive Behavioral Therapy (CBT)
Family-Based Treatment
Interpersonal Therapy (IPT)
Medication: SSRIs like fluoxetine
/app $ wc -l /app/treatment_list.txt
4 /app/treatment_list.txt
/app $ grep -q "Cognitive Behavioral Therapy\|CBT" /app/treatment_list.txt && echo "Contains
CBT"
Contains CBT
```