STRATA: STRATIFIED TRAJECTORY-TO-KNOWLEDGE AUGMENTATION FOR TERMINAL AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Terminal agents complete software and system tasks by issuing shell commands, and every command changes the environment state. They still fail on long-horizon tasks, where a single wrong command can invalidate every later step. Post-training improves them at the cost of trajectory corpora, executable environments, and optimization, while existing experience-reuse methods keep what worked, or reduce failures to global rules, rather than recording when a plausible procedure fails on a task and what succeeded instead. We propose **STRATA**, a training-free framework that turns past terminal trajectories into stratified external knowledge for a frozen agent. Successful trajectories become Skills, reusable multi-step procedures; contrasts between successful and failed attempts on the same task become Memories, rules about when a procedure fails and what succeeded instead; both are organized by capability level, from command syntax to long-horizon planning, and an agentic retriever selects the entries relevant to each task. On Terminal-Bench 2.0, STRATA raises DeepSeek-V4-Pro from 53.2% to 63.3% Pass Rate, above three experience-augmentation baselines built from the same trajectories and level with frontier models on the same scaffold; ablations attribute the gain to task-selected, stratified trajectory knowledge rather than to prompt budget or the library alone.

1 INTRODUCTION

Terminal agents act in real computing environments through the command line, completing software development, system administration, and data processing tasks by issuing shell commands, reading the output, and adapting the next action to the evolving state. Because files, processes, dependencies, and permissions change with every command, a terminal task is long-horizon and state dependent, and a single wrong action can invalidate everything that follows. Current agents fail in characteristic ways: they lose track of state, recover poorly from errors, and repeat strategies that have already failed. The knowledge needed to avoid these failures exists in past runs, but it is scattered across trajectories and left unused.

Post-training, including supervised fine-tuning, reinforcement learning, and trajectory distillation, improves terminal capability but is slow and costly, since it requires data collection and optimization in executable environments (Gandhi et al., 2026; Lin et al., 2026; Wu et al., 2026). A complementary route keeps reusable experience outside the model parameters and retrieves it at inference time; workflows, executable skills, and procedural memory already establish its value (Wang et al., 2024; 2023; Fang et al., 2026). These methods record what worked. Successful trajectories show how to make progress: which flags to use, how to compose tools through pipes, and what to verify before a destructive change. They do not show why agents fail, when a plausible command becomes unsafe because the environment has changed, or how to recover after a wrong action. That knowledge lives in failed attempts, and it becomes visible only in contrast with a success on the same task. Methods that do learn from failures reduce them to a global rule list (Zhao et al., 2024) or to notes on single trajectories (Ouyang et al., 2026), detached from the successful alternative on the same task. A terminal agent therefore needs both kinds of knowledge, organized by the level at which a decision is made and selected for the task at hand.

We propose STRATA, a training-free framework that augments terminal agents with structured knowledge distilled from prior trajectories. STRATA builds two complementary resources. A Skill

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.

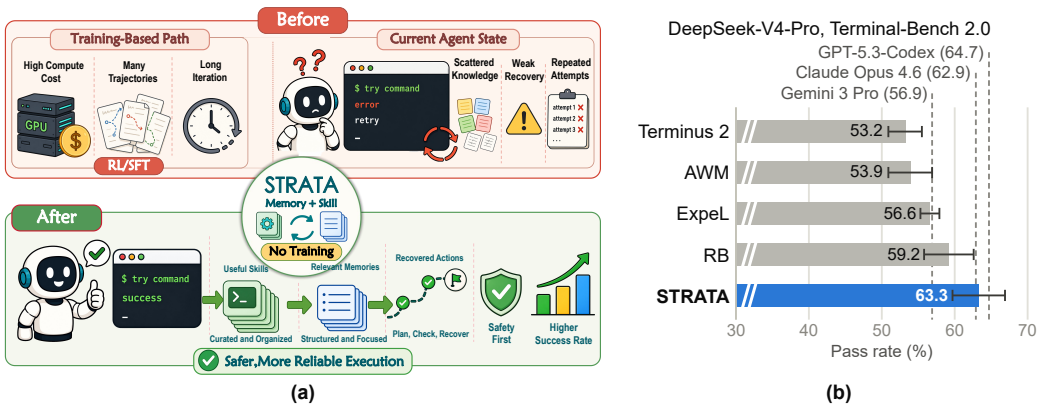


Figure 1: **(a)** Background and motivation of STRATA. Training-based improvement for terminal agents is costly and slow, while current agents still suffer from scattered knowledge, weak recovery, and repeated attempts. STRATA organizes past trajectories into Skills and Memories without model training, providing procedures and failure warnings for execution. **(b)** Pass Rate of DeepSeek-V4-Pro on Terminal-Bench 2.0 with the Terminus 2 scaffold (Table 2). Bars show means over three runs and whiskers one sample standard deviation; dashed lines mark leaderboard entries of frontier models on the same scaffold. RB denotes ReasoningBank (Ouyang et al., 2026).

Library abstracts reusable procedures from successful trajectories: command composition, multi-step execution, and safety checks. A Memory Pool distills contrasts between successful and failed trajectories of the same task into rules that record the failed action, the successful alternative, and the context in which the contrast applies. Both resources are stratified by capability level, from command syntax to long-horizon planning. At inference time an agentic retriever selects the Skills and Memories relevant to the current task through category selection, catalog browsing, and entry selection, and the selected entries are placed in the agent context before execution. The backbone model and the terminal scaffold remain unchanged.

On Terminal-Bench 2.0, STRATA raises DeepSeek-V4-Pro from 53.2% to $63.3 \pm 3.6\%$ over three runs, above offline Agent Workflow Memory (AWM) (Wang et al., 2024), ExpeL (Zhao et al., 2024), and ReasoningBank (Ouyang et al., 2026) built from the same trajectory pool (53.9%, 56.6%, and 59.2%; Figure 1b), and on the same Terminus 2 scaffold level with Claude Opus 4.6 and above Gemini 3 Pro. Replacing the retrieved knowledge with static tips, agentic selection with embedding retrieval, or the stratified library with a flat one loses 11.6, 12.7, and 6.0 points, so the gain comes from trajectory-derived content that is selected per task and organized by level. The same library raises a weaker Qwen3-32B backbone from 4.1% to 10.9%.

Our contributions are as follows:

- We introduce STRATA, a training-free framework that augments a frozen terminal agent with Skills and Memories distilled from past trajectories and organized by capability level, without changing the backbone model or the scaffold.
- We build a stratified Skill Library from successful trajectories, capturing reusable command composition and multi-step execution routines, and a contrastive Memory Pool from success–failure pairs of the same task, capturing error patterns, state-related pitfalls, and recovery knowledge that successes alone do not reveal.
- We design an agentic retrieval mechanism that selects and combines task-relevant Skills and Memories at inference time.
- We evaluate STRATA on Terminal-Bench 2.0 with two backbones against three experience-augmentation methods under one protocol, show with matched ablations that content, selection, and stratification each contribute, and report source–benchmark overlap, cost, and failure modes.

2 Related Work

Terminal agents. Terminal agents complete software development, system operation, and data processing tasks by issuing shell commands and reading text feedback (Yang et al., 2024; Wang et al., 2025). Terminal-Bench supplies verifiable evaluation tasks (Merrill et al., 2026), and Endless Terminals and CLI-Gym generate executable environments at scale (Gandhi et al., 2026; Lin et al., 2026). Most work on raising terminal capability trains the model, through supervised fine-tuning and reinforcement learning in these environments (Gandhi et al., 2026; Lin et al., 2026), data engineering (Pi et al., 2026), and large-scale trajectory generation (Wu et al., 2026). Training-free work improves the scaffold and context engineering (Bui, 2026) or compresses observations (Ren et al., 2026). STRATA is training-free and uses trajectories as external knowledge for a frozen agent rather than as optimization targets.

Skill libraries and procedural memory. External memory lets agents retain experience beyond one context window (Majumder et al., 2023; Packer et al., 2023; Xu et al., 2026; Yang et al., 2025b). We focus on what that memory should contain. Voyager accumulates executable skills from environment feedback (Wang et al., 2023), AWM induces workflows shared across successful trajectories and inserts them into the prompt (Wang et al., 2024), MemP builds procedural memory as verbatim steps or abstracted scripts (Fang et al., 2026), and later work compiles hierarchical skills, learns procedural memory with non-parametric policy updates, and distills trajectory-local lessons into transferable skills (Carta et al., 2025; Mi et al., 2026; Ni et al., 2026; Xu & Yan, 2026; Jiang et al., 2026). These libraries are built from successful behavior, so they record how to make progress but not when a plausible procedure fails or how to recover. The STRATA Skill Library belongs to this line and adds a capability level to each Skill.

Learning from failures. Reflexion reflects on a failed attempt and retries the same task (Shinn et al., 2023). Across tasks, ExpeL distills success–failure comparisons into one global rule list (Zhao et al., 2024), AutoGuide derives state-conditioned guidelines from offline web experience (Fu et al., 2024), ReasoningBank extracts items from each successful or failed trajectory and retrieves them by task similarity (Ouyang et al., 2026), and Learn-by-Interact synthesizes trajectories for training or retrieval (Su et al., 2025). Among the methods we compare against, AWM stores successes only, ExpeL shares one rule list across all tasks, and ReasoningBank distills each failure item from a single failed attempt without a paired success. STRATA instead keeps procedures and contrastive lessons as separate resources, records for each lesson the failed action, the successful alternative on the same task, and its context, stratifies both resources by capability level, and selects entries with an agentic retriever that reads the catalog rather than matching embeddings. Section 5 compares the four methods under one protocol and isolates the organization and retrieval choices with matched-content ablations.

3 Preliminaries

3.1 Terminal Agent Task Formulation

We formulate a terminal task as an interactive decision process between an agent and a command-line environment. A task is specified by an instruction I, an initial environment state E_0, an action space A of executable shell commands, and an observation space O of terminal outputs. At step t, the agent conditions on the instruction and on the history of previous actions and observations and produces a command $a_t \in A$. The environment executes the command, updates its hidden state from E_t to E_{t+1}, and returns an observation $o_t \in O$. The interaction ends when the agent emits a termination action or exhausts the step budget. A task succeeds when a verifier V accepts the final state, $V(E_T) = 1$.

3.2 Agent Trajectory

A trajectory for task I records the interaction,

$$\tau = \{(a_1, o_1), (a_2, o_2), \ldots, (a_T, o_T)\},$$

where a_t is the command issued at step t and o_t the observation returned by the environment. The verifier outcome splits trajectories into successful ones, τ^+ with $V(E_T) = 1$, and failed ones, τ^-

Table 1: Knowledge layers of terminal experience.

Layer	Knowledge scope	Description
L_1	Command syntax	Validity of a single command: shell syntax, argument formatting, quoting, paths, and redirection.
L_2	Single-tool usage	Correct use of one command-line tool, with its options, for a local subgoal.
L_3	Tool composition	Coordination of multiple commands and their intermediate outputs into a reusable multi-step procedure.
L_4	Long-horizon planning	Task decomposition, subgoal ordering, progress monitoring, recovery from failed attempts, and termination.

with $V(E_T) = 0$; we write $V(\tau)$ for this outcome. Different models or sampling runs on the same task produce trajectories that differ in command choices and in outcome. STRATA builds Skills from the successful trajectories and Memories from contrasts between a successful and a failed trajectory of the same task.

3.3 Knowledge Layers of Terminal Experience

Beyond its commands and observations, a trajectory exposes the knowledge that made those commands effective, and that knowledge is exercised at different levels of decision: a quoting rule, the options of one tool, a multi-command procedure, or the ordering of subgoals. STRATA organizes terminal knowledge into four capability levels, or layers,

$$\mathcal{L} = \{L_1, L_2, L_3, L_4\},$$

from local command validity to task-level planning (Table 1). The layers are an annotation space shared by both knowledge types: every Skill and every Memory is tagged with the layer at which it applies, and the catalog and the prompt assembly use the tag. Skills are abstracted from L_3 and L_4, where successful trajectories reveal reusable procedures and planning routines, whereas Memories span all four layers, since failures range from local pitfalls to strategic errors.

3.4 Skills and Memories

Skills and Memories are two complementary forms of external knowledge for terminal agents. A Skill s is a reusable procedure distilled from successful trajectories, written as a sequence of action templates with their purposes,

$$s = \{(u_1, p_1), (u_2, p_2), \ldots, (u_n, p_n)\},$$

where u_i is a command pattern or operation step and p_i its role in the procedure. Skills guide execution: tool composition, multi-step planning, and safety checks before destructive operations.

A Memory m is a lesson distilled from a successful and a failed trajectory of the same task. It records the action that failed, the alternative that succeeded, and the context in which the contrast applies. Memories constrain execution and support recovery, for example by avoiding unsafe command flags, re-checking a changed environment state, or leaving a loop of repeated failed attempts. The Skills form the Skill Library $\mathcal{S}$ and the Memories the Memory Pool $\mathcal{M}$. At inference time, STRATA retrieves a task-specific subset of both,

$$(S_I, M_I) = \mathcal{R}(I, \mathcal{S}, \mathcal{M}),$$

and places it in the agent context.

4 Methodology

4.1 Overview

STRATA turns trajectories of previous terminal tasks into stratified external knowledge and reuses it at inference (Figure 2). The framework has three stages. Trajectory collection (Section 4.2) runs several

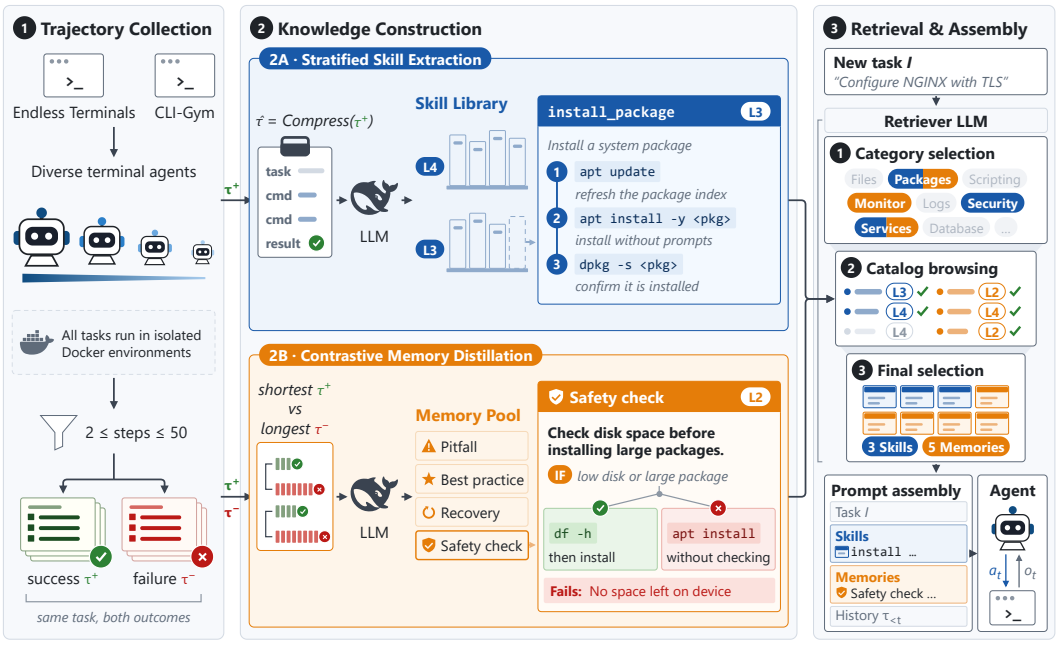


Figure 2: Overview of STRATA. Knowledge is constructed offline from filtered build-task trajectories. Skills encode successful procedures; Memories contrast successful and failed attempts on the same task. At inference, category selection, catalog browsing, and entry selection supply up to 3 Skills and 5 Memories to a frozen terminal agent. The library remains fixed during evaluation.

backbone models on the same source tasks, so that each task exposes both successful and failed attempts. Knowledge construction (Sections 4.3 and 4.4) builds the Skill Library from successful trajectories and the Memory Pool from success–failure pairs of the same task, and indexes both by topic category and capability layer. Agentic retrieval (Section 4.5) selects a task-specific subset of both resources and places it in the agent context before the first command. The backbone model and the terminal scaffold are unchanged, and the library is fixed during evaluation.

4.2 Multi-Model Trajectory Collection

Source tasks come from the build splits of Endless Terminals (Gandhi et al., 2026) and CLI-Gym (Lin et al., 2026). Each task is attempted by DeepSeek-V4-Pro (DeepSeek-AI, 2026), GLM-5 (Zeng et al., 2026), Kimi-K2.5 (Team et al., 2026), and Qwen3-VL-30B-A3B-Instruct (Yang et al., 2025a): the stronger models supply successful trajectories, and the weaker one exposes representative failures. Every model interacts with a fresh Docker container initialized from the same task environment, so attempts share the initial state but not each other's modifications. We retain trajectories with 2–50 steps and normalize the instruction, commands, observations, outcome, and source information. Tasks with both a successful and a failed attempt supply the pairs for Memory distillation.

4.3 Stratified Skill Extraction

Skill extraction abstracts each verifier-passing trajectory into the procedure it followed. A trajectory is first compressed to its instruction, key command sequence, terminal evidence, and outcome, $\hat{\tau} = \text{Compress}(\tau^+)$, and compressed trajectories are processed in batches by an extraction model, $\mathcal{S}_{\text{raw}} = \text{LLM}_{\text{extract}}(\hat{\tau}_1, \ldots, \hat{\tau}_B)$. The extractor targets L_3 tool composition and L_4 planning. Its prompt excludes task-specific facts, file names, and values unless they express a general pattern, and asks for safety checks around destructive operations, so that a Skill transfers across tasks instead of replaying one solution. The raw set is normalized by exact deduplication (same name and layer), LLM-based semantic merging of Skills that express the same procedure in different words, and frequency filtering,

$$\mathcal{S} = \{s \in \text{Merge}(\mathcal{S}_{\text{raw}}) \mid \text{count}(s) \geq 2\},$$

where count(s) is the number of source tasks supporting s, so a Skill seen in one task only is discarded as task-specific. Retained Skills are indexed by a set $\mathcal{C}$ of 20 categories of general CLI concepts, such as file operations and networking, fixed before evaluation, to form a Skill Catalog $\mathcal{C}_S$. Categories index topics, and the L_1–L_4 layers index capability. Extraction and merging use DeepSeek-V4-Pro. The final library contains 725 Skills (298 at L_3, 427 at L_4); Appendix A.5 gives its composition.

4.4 Contrastive Memory Distillation

Memory distillation recovers why an attempt failed and what the successful attempt did instead. For each task I_i with both outcomes, STRATA pairs the shortest successful trajectory with the longest failed one,

$$\tau_i^+ = \arg\min_{V(\tau)=1} |\tau|, \qquad \tau_i^- = \arg\max_{V(\tau)=0} |\tau|,$$

so that the positive reference is compact and the negative one exposes unsuccessful decisions and recovery attempts. Because outcomes differ across rollout models, a pair may combine trajectories from different models. Compressed pairs are processed in batches by the same model, $\mathcal{M}_{\text{raw}} = \text{LLM}_{\text{contrast}}((\tau_1^+, \tau_1^-), \ldots, (\tau_B^+, \tau_B^-))$, which compares command choices, observations, and outcomes and writes rules at any of the four layers. Each Memory is a tuple

$$m = (r, a^+, a^-, c, l, t),$$

with rule r, successful and failed actions a^+ and a^-, applicable context c, layer l, and type t (pitfall, best practice, recovery pattern, or safety check). The same deduplication, semantic merging, and frequency filtering yield the Memory Pool $\mathcal{M}$, indexed by category, layer, and type in a Memory Catalog $\mathcal{C}_M$. The final pool contains 168 Memories from 657 raw entries (Appendix A.5).

4.5 Agentic Retrieval and Prompt Assembly

At inference time, STRATA selects a task-specific subset of the library in three steps, each performed by a lightweight selection model, DeepSeek-V4.1-Flash in non-thinking mode, and run separately for Skills and for Memories. Given the instruction I, the retriever first selects the most relevant categories, $\mathcal{C}_{\text{sel}} = \text{LLM}_{\text{select}}(I, \mathcal{C})$. It then browses the catalog summaries under these categories (names, layers, types, and short descriptions) to build a candidate set. Finally it reads the full candidate entries and selects up to 3 Skills $S_I \subset \mathcal{S}$ and 5 Memories $M_I \subset \mathcal{M}$. The catalog step lets the retriever judge whether an entry applies to the task instead of ranking entries by embedding similarity to the instruction. Section 5.3 tests this choice. The library is fixed: the retriever reads it, and test-task outcomes never enter it.

During prompt assembly, selected Skills are ordered by layer, L_3 composition patterns before L_4 planning routines, and Memories are formatted as concise warnings with the rule, its context, the failed and successful actions, and the suggested correction. The agent thus receives candidate procedures and error constraints before its first command, and each command is generated as

$$a_t = \pi_\theta\big(I, \tau_{<t}, \text{Assemble}(S_I, M_I)\big),$$

where retrieval and assembly are external to the backbone π_θ and can be attached to any terminal agent.

5 Experiments

5.1 Setup

Benchmark and metric. We evaluate on the 89 tasks of Terminal-Bench 2.0 (Merrill et al., 2026) (4 Easy, 55 Medium, 30 Hard), each with a task-specific verifier. Pass Rate is the fraction of tasks whose verifier passes within the step budget, and overall values are task-weighted. The knowledge library is built from the source tasks of Section 4.2 and fixed before evaluation.

Baselines and protocol. All agents run the Terminus 2 scaffold of Terminal-Bench. An experience-augmented agent differs from the zero-shot scaffold only in a knowledge block appended to the task instruction under a fixed 7,200-character budget. We compare STRATA with the offline mode of

Table 2: Main results on Terminal-Bench 2.0 with the Terminus 2 scaffold (Merrill et al., 2026). Pass Rate (%); our results are means over three runs with sample standard deviations as subscripts, and * marks leaderboard values with their reported 95% confidence intervals. Bold and underlined values are the best and second best within each backbone.

Agent	Model	Easy (4)	Med. (55)	Hard (30)	Avg.
Reference backbones					
Terminus 2	GPT-5.3-Codex	–	–	–	$64.7_{\pm 2.7}$*
Terminus 2	Claude Opus 4.6	–	–	–	$62.9_{\pm 2.7}$*
Terminus 2	Gemini 3 Pro	–	–	–	$56.9_{\pm 2.5}$*
Weaker backbone					
Terminus 2	Qwen3-32B	$8.3_{\pm 14.4}$	$6.1_{\pm 3.8}$	$0.0_{\pm 0.0}$	$4.1_{\pm 1.7}$
AWM	Qwen3-32B	$0.0_{\pm 0.0}$	$3.6_{\pm 0.0}$	$0.0_{\pm 0.0}$	$2.2_{\pm 0.0}$
ExpeL	Qwen3-32B	$\underline{16.7}_{\pm 28.9}$	$\underline{8.5}_{\pm 2.1}$	$\mathbf{2.2}_{\pm 3.8}$	$\underline{6.7}_{\pm 2.2}$
ReasoningBank	Qwen3-32B	$0.0_{\pm 0.0}$	$7.3_{\pm 3.6}$	$0.0_{\pm 0.0}$	$4.5_{\pm 2.2}$
STRATA	Qwen3-32B	$\mathbf{50.0}_{\pm 25.0}$	$\mathbf{12.7}_{\pm 3.1}$	$\mathbf{2.2}_{\pm 1.9}$	$\mathbf{10.9}_{\pm 3.4}$
Stronger backbone					
Terminus 2	DeepSeek-V4-Pro	$\underline{91.7}_{\pm 14.4}$	$63.0_{\pm 2.1}$	$30.0_{\pm 5.8}$	$53.2_{\pm 2.3}$
AWM	DeepSeek-V4-Pro	$83.3_{\pm 14.4}$	$66.1_{\pm 1.0}$	$27.8_{\pm 5.1}$	$53.9_{\pm 3.0}$
ExpeL	DeepSeek-V4-Pro	$83.3_{\pm 14.4}$	$66.7_{\pm 6.9}$	$\underline{34.4}_{\pm 10.2}$	$56.6_{\pm 1.3}$
ReasoningBank	DeepSeek-V4-Pro	$\mathbf{100}_{\pm 0.0}$	$\underline{69.7}_{\pm 4.6}$	$\underline{34.4}_{\pm 1.9}$	$\underline{59.2}_{\pm 3.4}$
STRATA	DeepSeek-V4-Pro	$\underline{91.7}_{\pm 14.4}$	$\mathbf{75.2}_{\pm 3.8}$	$\mathbf{37.8}_{\pm 5.1}$	$\mathbf{63.3}_{\pm 3.6}$

AWM (Wang et al., 2024), ExpeL (Zhao et al., 2024), and ReasoningBank (Ouyang et al., 2026). Each baseline is built once from the same trajectory pool with the same construction model, frozen, and run in the same single-attempt protocol. Appendix A.3 lists what is kept from each official implementation and what is adapted to the terminal setting. The same library serves both backbones, DeepSeek-V4-Pro (DeepSeek-AI, 2026) and Qwen3-32B (Yang et al., 2025a). Every method is evaluated in three independent runs, reported as mean and sample standard deviation. Appendix A.1 gives the step budget, decoding settings, and model roles.

Ablations. Each ablation in Table 3 removes or replaces one design factor of STRATA and keeps the scaffold, injection surface, and budget. Static tips (w/o trajectory content) replace the retrieved knowledge with generic CLI advice of matched format, topics, and length (about 4.1K characters). Raw trajectories (w/o distillation) inject the most similar successful source trajectories without distillation, in the style of Synapse (Zheng et al., 2024). Embedding top-k (w/o agentic retrieval) replaces agentic selection with embedding retrieval over the same library. Flat library (w/o layer tags) removes the layer tags, so neither retrieval nor prompt assembly can use them. Skill only (w/o Memory Pool) and Memory only (w/o Skill Library) remove one knowledge type. All ablations use the DeepSeek-V4-Pro backbone.

5.2 Main Results

STRATA outperforms experience-augmentation baselines built from the same trajectories. With DeepSeek-V4-Pro, STRATA reaches $63.3 \pm 3.6\%$, above AWM, ExpeL, and ReasoningBank (Table 2). Relative to the zero-shot scaffold, the four methods add 0.7, 3.4, 6.0, and 10.1 points. The lowest STRATA run is above every run of AWM, ExpeL, and the zero-shot scaffold, and only the best ReasoningBank run reaches the STRATA range (Appendix A.2); run-to-run standard deviations are similar across methods. All four libraries come from the same pool and the same construction model, so the ordering tracks what each method keeps from the trajectories and how it is selected: AWM keeps successful workflows only, ExpeL and ReasoningBank also extract from failures, and STRATA keeps procedures and contrastive lessons as separate resources and selects both per task. Section 5.3 tests this reading with ablations.

Table 3: Ablation of STRATA on Terminal-Bench 2.0 with the DeepSeek-V4-Pro backbone (Pass Rate, %). Each ablation removes or replaces one design factor and keeps the scaffold, injection surface, and prompt budget. Full STRATA and Zero-shot are three-run means; ablation rows are single runs (Appendix A.2). Δ is the difference from Full STRATA in percentage points.

Ablation	Change	Easy (4)	Med. (55)	Hard (30)	Avg.	Δ
Zero-shot	w/o knowledge	91.7	63.0	30.0	53.2	-10.1
Content	w/o trajectory content	100	63.6	23.3	51.7	-11.6
	w/o distillation	75.0	70.9	33.3	58.4	-4.9
Selection	w/o agentic retrieval	75.0	60.0	30.0	50.6	-12.7
Organization	w/o layer tags	100	67.3	33.3	57.3	-6.0
Components	w/o Memory Pool	75.0	70.9	33.3	58.4	-4.9
	w/o Skill Library	100	72.7	36.7	61.8	-1.5
Full STRATA	–	91.7	75.2	37.8	**63.3**	–

The gain lands on Medium and Hard tasks. The Easy subset has four tasks, and every DeepSeek-V4-Pro agent solves nearly all of them with large run-to-run variation, so it does not separate the methods. STRATA raises the scaffold by 12.2 points on Medium and 7.8 on Hard tasks, the highest means in the block, 5.5 and 3.4 points above ReasoningBank and 9.1 and 10.0 above AWM, although the Hard margin over ReasoningBank is within one standard deviation of the STRATA runs. These are the tasks in which a wrong command invalidates later steps, the setting that motivates the contrastive Memory Pool.

A frozen backbone reaches the frontier band. The reference rows list Terminus 2 entries of frontier models from the Terminal-Bench leaderboard, whose intervals are 95% confidence intervals rather than run standard deviations. Without STRATA, DeepSeek-V4-Pro is 11.5 points below GPT-5.3-Codex and 3.7 below Gemini 3 Pro. With STRATA it is above Gemini 3 Pro, level with Claude Opus 4.6, and 1.4 points below GPT-5.3-Codex. External knowledge thus recovers most of the gap between this backbone and the strongest leaderboard entry without changing the model or the scaffold.

The library transfers to a much weaker backbone. With Qwen3-32B, the zero-shot scaffold solves 4.1% of the tasks and STRATA 10.9%, 2.7 times as many; AWM falls below the scaffold, and ReasoningBank and ExpeL stay near it. The gain is on Medium and Easy tasks, while Hard tasks stay at 2.2% or below for every method, so the library raises what this backbone can already attempt and does not substitute for the long-horizon execution it lacks. A library built by other models on other tasks thus transfers to a backbone that solves few tasks on its own, whereas injected experience is not uniformly helpful: workflows from AWM lower Qwen3-32B below its zero-shot rate.

5.3 Ablation Study

Table 3 removes or replaces one design factor of STRATA at a time, with the scaffold, injection surface, and prompt budget unchanged, and asks which factors the gain over the zero-shot scaffold depends on: the content of the injected knowledge, how entries are selected, the layered organization, and the two knowledge types.

Content: trajectory content is necessary and distillation adds to it. Static tips of matched format, topics, and length lose 11.6 points to Full STRATA and fall 1.5 points below the zero-shot scaffold, with the largest loss on Hard tasks. The same prompt budget filled with plausible shell guidance does not help. Raw trajectories, the most similar successful source trajectories injected without distillation, recover 5.2 points over the zero-shot scaffold but remain 4.9 below Full STRATA, with the loss on Medium and Hard tasks. Trajectory content therefore helps even undistilled, and distilling it into Skills and Memories adds the remaining 4.9 points. The content must be derived from trajectories, distilled, and selected for the task.

Selection: the library alone is not enough. Embedding top-k retrieval over the same library loses 12.7 points to Full STRATA and falls 2.6 points below zero-shot, the largest drop in the table, on both Medium and Hard tasks. The library thus has no value by itself: how entries are chosen for a task decides whether it helps, and loosely matched entries distract the agent rather than merely fail to help. The two retrievers differ in the judgment of applicability that the catalog step makes (Section 4.5), which similarity to the instruction does not encode.

Organization: stratification contributes beyond content. Flattening the same knowledge, so that neither retrieval nor prompt assembly can use the capability layers, loses 6.0 points, on Medium and Hard tasks. The entries, the retriever, and the budget are unchanged, and only the layer tags differ. The organization of the library therefore contributes on its own, consistent with the role of the tags in retrieval and assembly: composition patterns and planning routines are browsed separately and injected in order, so the agent receives a coherent procedure rather than a mixed list.

Components: the two knowledge types are complementary. Skill only and Memory only are both below the full method on Medium and Hard tasks; removing the Memory Pool costs 4.9 points and removing the Skill Library 1.5. Skills supply multi-step procedures, whereas Memories flag a precondition, a state mismatch, or a previously unsuccessful action; the larger cost of removing Memories is consistent with a backbone that already knows plausible commands but benefits from explicit warnings about which strategies are unreliable, the knowledge that successful trajectories alone do not expose. Appendix A.5 varies the number of retrieved Skills on Qwen3-32B.

No single factor accounts for the gain: every ablation lowers the Pass Rate, and the two largest drops, agentic selection and trajectory-derived content, exceed the gain itself, so those ablated agents fall below the zero-shot scaffold.

5.4 Analysis

Three checks bound what the gain can be attributed to. First, it is not reuse of benchmark content. Against all 2,880 source tasks, no Terminal-Bench task reaches the near-duplicate criteria at the task-text level, official solutions share at most 0.04 4-gram Jaccard with any trajectory in the pool, the knowledge retrieved in 103 solved trials from earlier runs of the same protocol shares at most 0.7% 4-gram overlap with the official solutions, and the sources share no repositories or base templates with the benchmark (Appendix A.4). Second, the cost is small relative to execution. Knowledge construction is a one-time 1.53M tokens reused across all tasks, and retrieval adds a mean of 5.8 LLM calls and 21.9K tokens per task, about 1.2% of the 1.8M input tokens of a mean episode (Appendix A.5). Third, the remaining failures lie in execution rather than retrieval. Injection completes in 97.8% of trials, 67.3% of the Memory Pool is used at least once, and episodes that fail despite injected knowledge exhibit long-horizon planning, budget, and verification failures while relevant guidance is available; we found no case in which an incorrect Memory misled an otherwise-correct trajectory. Appendix A.7 traces one incident-response task in which the injected guidance organizes the validation and output checks that the zero-shot agent lacks.

6 Conclusion

STRATA organizes successful terminal procedures and contrastive failure lessons into stratified external knowledge and retrieves them per task for a frozen agent. Under matched conditions on Terminal-Bench 2.0, it raises DeepSeek-V4-Pro from 53.2% to $63.3 \pm 3.6\%$, above three experience-augmentation baselines and within the band of frontier models on the same scaffold, and it raises a much weaker backbone as well. Ablations attribute the gain to trajectory-derived content, task-specific selection, and layered organization rather than to prompt budget or the library alone. Environment-specific lessons about tool versions, states, and conventions belong in runtime knowledge; the more general decision rules that success–failure contrasts expose are also candidates for post-training, and combining the two is a natural next step.

AI USE STATEMENT

In this work, we used generative AI tools for generating synthetic datasets, for developing the conceptual framework and refining hypotheses, for designing and giving feedback on the experimental methodology, for implementing methods, for cleaning and reformatting data, for interpreting results, and for assisting with translation. The synthetic data are the terminal trajectories rolled out by the LLMs listed in Section 4.2 and the Skills and Memories distilled from them; these LLM calls are components of the proposed method and are reported in Sections 4.3 to 4.5 and Appendix A.1. We have not used generative AI tools for qualitative and thematic data analysis, and formulating mathematical claims, providing ingredients for proofs, and writing proofs are not applicable to this work. Additionally, we used generative AI tools to identify and summarize related literature, to brainstorm and discover research directions, to draft parts of the paper, to edit the paper for readability, to create or edit code for data processing, baselines, evaluation, and analysis, and to generate some visual elements and icons in the figures; the figures themselves were designed and drawn by the authors. We have reviewed all AI-assisted work: AI-generated code was reviewed and tested by the authors, all reported numbers come from experiments run by the authors and were checked against the run logs, AI-suggested references were verified by reading the original papers, and all AI-drafted or AI-edited text was revised and checked against the experimental records by the authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

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

Thomas Carta, Clément Romac, Loris Gaven, Pierre-Yves Oudeyer, Olivier Sigaud, and Sylvain Lamprier. Herakles: Hierarchical skill compilation for open-ended llm agents. *arXiv preprint arXiv:2508.14751*, 2025.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. Memp: Exploring agent procedural memory. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 17490–17502, 2026.

Yao Fu, Dong-Ki Kim, Jaekyeom Kim, Sungryull Sohn, Lajanugen Logeswaran, Kyunghoon Bae, and Honglak Lee. Autoguide: Automated generation and selection of context-aware guidelines for large language model agents. *Advances in Neural Information Processing Systems*, 37:119919–119948, 2024.

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

Yanna Jiang, Delong Li, Haiyu Deng, Baihe Ma, Xu Wang, Qin Wang, and Guangsheng Yu. Sok: Agentic skills–beyond tool use in llm agents. *arXiv preprint arXiv:2602.20867*, 2026.

Yusong Lin, Haiyang Wang, Shuzhe Wu, Lue Fan, Feiyang Pan, Sanyuan Zhao, and Dandan Tu. Cli-gym: Scalable cli task generation via agentic environment inversion. *arXiv preprint arXiv:2602.10999*, 2026.

Bodhisattwa Prasad Majumder, Bhavana Dalvi Mishra, Peter Jansen, Oyvind Tafjord, Niket Tandon, Li Zhang, Chris Callison-Burch, and Peter Clark. Clin: A continually learning language agent for rapid task adaptation and generalization. *arXiv preprint arXiv:2310.10134*, 2023.

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.

Qirui Mi, Zhijian Ma, Mengyue Yang, Haoxuan Li, Yisen Wang, Haifeng Zhang, and Jun Wang. Procmem: Learning reusable procedural memory from experience via non-parametric ppo for llm agents. *arXiv preprint arXiv:2602.01869*, 2026.

Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Erchao Zhao, Xiaoxi Jiang, and Guanjun Jiang. Trace2skill: Distill trajectory-local lessons into transferable agent skills. *arXiv preprint arXiv:2603.25158*, 2026.

Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, et al. Reasoningbank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, volume 2026, pp. 94327–94354, 2026.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G Patil, Ion Stoica, and Joseph E Gonzalez. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

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.

Jincheng Ren, Siwei Wu, Yizhi Li, Kang Zhu, Shu Xu, Boyu Feng, Ruibin Yuan, Wei Zhang, Riza Batista-Navarro, Jian Yang, et al. A self-evolving framework for efficient terminal agents via observational context compression. *arXiv preprint arXiv:2604.19572*, 2026.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Hongjin Su, Ruoxi Sun, Jinsung Yoon, Pengcheng Yin, Tao Yu, and Sercan Arik. Learn-by-interact: A data-centric framework for self-adaptive agents in realistic environments. In *International Conference on Learning Representations*, volume 2025, pp. 89044–89097, 2025.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Ziwei Chai, Y Charles, HS Che, Cheng Chen, et al. Kimi k2. 5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models, 2023. *URL https://arxiv. org/abs/2305.16291*, 2(11), 2023.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024.

Siwei Wu, Yizhi Li, Yuyang Song, Wei Zhang, Yang Wang, Riza Batista-Navarro, Xian Yang, Mingjie Tang, Bryan Dai, Jian Yang, et al. Large-scale terminal agentic trajectory generation from dockerized environments. *arXiv preprint arXiv:2602.01244*, 2026.

Renjun Xu and Yang Yan. Agent skills for large language models: Architecture, acquisition, security, and the path forward. *arXiv preprint arXiv:2602.12430*, 2026.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *Advances in Neural Information Processing Systems*, 38:17577–17604, 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.

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

Wei Yang, Jinwei Xiao, Hongming Zhang, Qingyang Zhang, Yanna Wang, and Bo Xu. Coarse-to-fine grounded memory for llm agent planning. *arXiv preprint arXiv:2508.15305*, 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. *arXiv preprint arXiv:2602.15763*, 2026.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. Synapse: Trajectory-as-exemplar prompting with memory for computer control. In *International Conference on Learning Representations*, volume 2024, pp. 19036–19066, 2024.

A Appendix

A.1 Implementation Details

Trajectory collection. For trajectory collection, we run multiple rollout models through a hosted API endpoint. The rollout pool covers different capability levels, including DeepSeek-V4-Pro, GLM-5, Kimi-K2.5, and Qwen3-VL-30B-A3B-Instruct. Stronger models are used to obtain more successful trajectories, while the weaker backbone is used to expose representative failure behaviors. Each model uses temperature 0.7 and a maximum generation length of 4096 tokens during rollout. For each task, we build the Docker image once and start a fresh container for each model rollout, so different models interact with the same initial environment without sharing modified states.

Docker execution. All terminal interactions are executed inside Docker containers. Each task uses the Dockerfile provided by the corresponding benchmark environment, rather than a single global base image. We allocate 2 CPU cores and 3 GB memory to each container, and set the Docker API timeout to 1200 seconds to cover both image construction and command execution. During interaction, the maximum number of agent steps is set to 50, and each command observation is truncated to 5000 characters to keep the prompt length stable. Rollout collection uses task level parallelism with concurrency 4.

Skill and Memory construction. For knowledge construction, trajectories are first filtered by step count. We keep trajectories with 2 to 50 steps and truncate task instructions to 300 characters during compression. Skill extraction uses successful trajectories only. Each trajectory is compressed into a compact summary containing the task instruction, command sequence, and outcome, then processed in batches of 20 trajectories. Memory distillation uses tasks with both successful and failed trajectories. For each task, we select the shortest successful trajectory and the longest failed trajectory, then process contrast pairs in batches of 12. Both Skill extraction and Memory distillation use DeepSeek-V4-Pro with temperature 0.2, top $p = 0.95$, JSON output mode, and a maximum output length of 8192 tokens.

Deduplication and filtering. After raw extraction, STRATA applies two stage normalization to both Skills and Memories. The first stage performs exact deduplication. Skills are merged when their names and layers match, while Memories are merged when their rule, layer, and type match. The second stage performs LLM assisted semantic merging within the same layer for Skills and within the same layer and type for Memories. The merge batch size is 35. After merging, entries with source frequency lower than 2 are removed, which filters task specific noise and keeps reusable knowledge.

Retrieval and prompt assembly. At inference time, agentic retrieval uses DeepSeek-V4.1-Flash in non-thinking mode as a lightweight selection model with temperature 0.1, JSON output mode, and maximum output lengths of 512, 1024, and 512 tokens for the three steps. Skills and Memories are retrieved by separate runs of the three steps, so a task uses up to six selection calls. The retriever first selects up to 3 relevant categories for Skills and up to 3 relevant categories for Memories. It then browses the corresponding catalogs and selects the final knowledge entries. In the default configuration, the agent receives up to 3 Skills and 5 Memories. Skills are formatted as procedural guidance, while Memories are formatted as concise warnings with rule, context, and evidence.

Evaluation protocol. Our self-run method comparisons use a common Docker execution protocol, Terminus 2 scaffold, verifier, and maximum step budget. An attempt is one episode, which may contain multiple terminal actions and within-episode repairs; a failed episode is not retried. Each three-run entry in Table 2 consists of independent runs; evaluation requests pass neither a temperature nor a reasoning setting, so the endpoint defaults apply, which differs from the explicitly configured temperatures for rollout collection and knowledge construction above. Pass Rate is the percentage of tasks for which the verifier returns success within the step limit. The static-tip ablation uses generic CLI advice of matched prompt budget, format, topics, and approximate length (4.1K characters).

Separation of model roles. The rollout pool supplies successful and failed source attempts. DeepSeek-V4-Pro performs knowledge extraction and merging; DeepSeek-V4.1-Flash performs agentic selection. The execution backbone then interacts with the terminal using the selected context.

A.2 Run-Level Results

Table 4: Overall Pass Rate (%) of each run behind the three-run Terminal-Bench 2.0 entries of Table 2; parentheses give the number of solved tasks out of 89.

Agent	Model	Run 1	Run 2	Run 3	Mean $\pm$ std
Terminus 2	Qwen3-32B	2.2 (2)	5.6 (5)	4.5 (4)	4.1 ± 1.7
AWM	Qwen3-32B	2.2 (2)	2.2 (2)	2.2 (2)	2.2 ± 0.0
ExpeL	Qwen3-32B	9.0 (8)	4.5 (4)	6.7 (6)	6.7 ± 2.2
ReasoningBank	Qwen3-32B	2.2 (2)	4.5 (4)	6.7 (6)	4.5 ± 2.2
STRATA	Qwen3-32B	14.6 (13)	10.1 (9)	7.9 (7)	10.9 ± 3.4
Terminus 2	DeepSeek-V4-Pro	53.9 (48)	50.6 (45)	55.1 (49)	53.2 ± 2.3
AWM	DeepSeek-V4-Pro	57.3 (51)	52.8 (47)	51.7 (46)	53.9 ± 3.0
ExpeL	DeepSeek-V4-Pro	55.1 (49)	57.3 (51)	57.3 (51)	56.6 ± 1.3
ReasoningBank	DeepSeek-V4-Pro	56.2 (50)	58.4 (52)	62.9 (56)	59.2 ± 3.4
STRATA	DeepSeek-V4-Pro	61.8 (55)	60.7 (54)	67.4 (60)	63.3 ± 3.6

Table 4 gives the rounded run-level values corresponding to the reported means and standard deviations. Each ablation in Table 3 was run once; the zero-shot and Full STRATA rows of that table repeat the three-run entries of Table 2. The Single Skill configuration in Table 10 was run once; Multi Skill is run 1 of the three Qwen3-32B STRATA runs.

A.3 Baseline Adaptations

Every experience-augmented agent in Table 2 is a thin wrapper around the unmodified Terminus 2 agent: the wrapper builds a knowledge block from a frozen library and appends it to the task instruction under a 7,200-character budget (about 1,800 tokens, the budget of the STRATA knowledge context). The scaffold, response format, step budget, and verifier are shared. Knowledge for all methods is built once from the same trajectory pool and the same construction model, then frozen. Table 5 lists, for each method, the component taken from the official implementation and every adaptation made for the terminal setting; prompts are reproduced verbatim from the official repositories except for the domain sentences and the AWM one-shot example noted in the table.

A.4 Overlap Between the Source Datasets and Terminal-Bench 2.0

Table 6 lists the overlap checks over the pooled build corpus that Section 5.4 summarizes. Table 7 separates the task-level comparison by source dataset for the 89 Terminal-Bench 2.0 tasks. Each Terminal-Bench 2.0 instruction is compared with every build-task instruction from Endless Terminals (1,727 tasks) and CLI-Gym (1,153 tasks) under three similarity measures: word-level TF-IDF cosine with inverse document frequencies computed over the 2,880 build instructions, Jaccard similarity of word 3-grams, and cosine similarity of Qwen3-Embedding-8B instruction embeddings. Instructions are truncated to 4,000 characters, and to 2,000 characters for embedding. For every Terminal-Bench 2.0 task we take its maximum similarity within each source and summarize these per-task maxima. A pair counts as a near-duplicate when its TF-IDF cosine exceeds 0.6 or its 3-gram Jaccard similarity exceeds 0.35. The 241-task Terminal-Bench 1.0 superset is used only for the semantic-neighbor check in Table 6. The 103 observations of the retrieved-knowledge check are solved trials from earlier runs of the same protocol, so a task may be counted more than once across runs.

Endless Terminals is the closer source under every measure. It holds the nearest build task for 83 of the 89 Terminal-Bench 2.0 tasks under the embedding measure and for 72 under TF-IDF, and it supplies all three neighbors with embedding cosine above 0.6. The closest pair, `openssl-selfsigned-cert` and an Endless Terminals certificate task (embedding cosine 0.740, TF-IDF 0.293), shares the procedure of generating a 2048-bit self-signed RSA certificate with OpenSSL. The Terminal-Bench 2.0 task additionally requires a specified subject, restricted key permissions, a combined PEM file, a verification file with the SHA-256 fingerprint, and a Python checker; the two tasks also use different paths and certificate subjects, so a solution to one does not satisfy the other. The second-closest pair (`log-summary-date-ranges`, cosine 0.699) counts log severities into a CSV file in both tasks, over

Table 5: Correspondence between the official implementations and the offline terminal adaptations used in Tables 2 and 3. "Pool" is the shared trajectory pool; "verifier label" means the pass/fail outcome of the rollout verifier.

Method	Kept from the official implementation	Adapted for this study
AWM	Workflow induction prompt and one-shot example (rewritten for shell commands), cumulative induction over successful trajectories, whole-workflow injection	Trajectories grouped by the 20 STRATA categories instead of website metadata; at test time a selector model chooses two categories and their workflows are injected up to the budget, because the full workflow set exceeds it
ExpeL	Insight-extraction prompts, the AGREE/REMOVE/EDIT/ADD operation set and rule-count arithmetic, rule cap of 20, chunks of 8 successful trajectories, sequential updates at temperature 0	Experience pool taken from the pool instead of Reflexion retries (one successful–failed pair per task, 928 tasks); rules only, the retrieved-trajectory few-shot component is disabled because terminal trajectories exceed the budget; environment sentence in the system prompt names a Linux terminal
ReasoningBank	Success and failure extraction prompts, at most three items per trajectory, item format and paragraph splitting, task-similarity retrieval with the instruct prefix, top-1 prior task, injection preamble	Offline and frozen instead of streaming; outcome from the verifier label instead of an LLM judge; domain sentence changed from code-repository issues to terminal tasks; embeddings from Qwen3-Embedding-8B; block appended to the instruction instead of the system prompt
Raw-trajectory retrieval	Exemplar memory keyed by task text with top-3 nearest exemplars (Zheng et al., 2024)	Exemplars are the shortest successful trajectory per pool task (2–50 steps), rendered as instruction plus command sequence; as many exemplars as fit the budget, at most three

Table 6: Overlap between the trajectory sources and Terminal-Bench 2.0. The retrieved-knowledge statistic is a 4-gram overlap over 103 solved trials from earlier runs of the same protocol.

Level	Observation
Task text	Against 2,880 build tasks: maximum TF-IDF cosine 0.29, maximum 3-gram Jaccard 0.02, no semantic-cosine pair above 0.9.
Semantic neighbors	Two pairs above 0.8 in a 241-task Terminal-Bench 1.0 superset: same domain, different objectives, non-interchangeable verifiers.
Solution templates	Maximum 4-gram Jaccard 0.026 between benchmark solutions and the passing Endless Terminals trajectories, and 0.040 against every trajectory in the pool.
Retrieved knowledge	At most 0.7% 4-gram overlap with official solutions across 103 solved trials from earlier runs of the same protocol.
Environment sources	No shared repositories or base templates.

Table 7: Task-level similarity between the 89 Terminal-Bench 2.0 tasks and each source dataset. For every Terminal-Bench 2.0 task, the maximum similarity to any build task of the source is computed; the table gives the mean and maximum of these per-task maxima and the number of Terminal-Bench 2.0 tasks whose maximum embedding cosine exceeds 0.6 or 0.8. No Terminal-Bench 2.0 task reaches the near-duplicate thresholds for either source.

		TF-IDF		Jaccard		Embedding		#Tasks	
Source	Build tasks	Mean	Max	Mean	Max	Mean	Max	> 0.6	> 0.8
Endless Terminals	1,727	0.110	0.293	0.009	0.020	0.436	0.740	3	0
CLI-Gym	1,153	0.085	0.198	0.005	0.018	0.318	0.476	0	0
Combined	2,880	0.114	0.293	0.010	0.020	0.438	0.740	3	0

five date ranges in Terminal-Bench 2.0 and per service and day in Endless Terminals. CLI-Gym stays below 0.48 embedding cosine and 0.20 TF-IDF cosine for every Terminal-Bench 2.0 task.

At the solution level, the official Terminal-Bench 2.0 solution scripts share at most 0.026 word 4-gram Jaccard similarity (mean 0.004) with the command sequence of the shortest passing Endless Terminals trajectory of any build task, and at most 0.040 (mean 0.005) with any Endless Terminals or CLI-Gym trajectory in the pool.

A.5 Library Composition, Cost, and Skill Quantity

Table 8: Composition of the final knowledge resources. The capability layers are defined in Table 1; categories form a separate, predefined topic index.

Resource	Breakdown	Total
Skills	L3: 298; L4: 427	725
Memories by layer	L1: 19; L2: 72; L3: 34; L4: 43	168
Memories by type	Best practice: 75; pitfall: 67; safety check: 21; recovery pattern: 5	168
Topic taxonomy	Predefined general CLI categories	20

The final library contains 725 Skills and 168 Memories (Table 8). Skills focus on composition and planning, while Memories span all four capability levels. The Memory construction funnel contains 657 raw entries, 420 after semantic merging, and 168 after filtering for support from at least two source tasks. The type distribution includes both failure warnings and positively phrased best practices: Memories are defined by their derivation from contrasting outcomes and their role as contextual decision guidance, and 75 of them are phrased as best practices. Examples appear in Appendix A.6.

Table 9: Reported resource use by stage. M and K denote million and thousand tokens. Token figures are given with their accounting scope.

Stage or statistic	Reported quantity and scope
Knowledge construction	1.53M tokens once: 1.16M for distillation and 0.37M for merging.
Retrieval	Mean 5.8 LLM calls and 21.9K tokens per task.
Agent episode	Approximately 1.8M input tokens per episode.
Successful episodes	Mean 27.8 turns over solved tasks.

Table 9 lists the available accounting by stage. The 21.9K retrieval-token figure is approximately 1.2% of the 1.8M episode-input-token figure. The mean of 27.8 turns is over successful episodes.

Table 10 reports 5.6% with a single Skill and 14.6% with multiple Skills. The single-Skill result corresponds to 1 of 4 Easy, 4 of 55 Medium, and 0 of 30 Hard tasks. Multiple procedures may cover different parts of a terminal workflow, such as inspecting files, applying a change, and validating the result; the agent must still adapt and compose the retrieved procedures.

Table 10: Sensitivity to Skill quantity on Terminal-Bench 2.0 with Qwen3-32B (point estimates). Memory count is fixed. The Single Skill overall value is 5.6% (5/89). Multi Skill is the default configuration; its value is run 1 of the three Qwen3-32B STRATA runs in Table 4.

Configuration	Easy (4)	Med. (55)	Hard (30)	Avg.
Single Skill	25.0	7.3	0.0	5.6
Multi Skill	75.0	16.4	3.3	14.6

A.6 Skill and Memory Examples

This section provides representative examples from the Skill Library and Memory Pool. The Skills illustrate reusable procedural patterns extracted from successful trajectories, while the Memories illustrate experience rules distilled from successful and failed trajectory contrasts.

Table 11: **Examples of L3 Skills.** L3 Skills describe reusable tool composition patterns.

Name	**Steps**	**Common Errors**	**Applicable Scenarios**
Find and batch process files	1. Use `find` to locate files by path, name, time, or size. 2. Use `-print0` and `xargs -0` for safe batch processing. 3. Verify the affected file list before applying changes.	Splitting file names with spaces. Applying deletion before checking matched files.	File cleanup, log management, archive processing
Parse and aggregate command output	1. Generate structured output with tools such as `grep`, `awk`, or `jq`. 2. Pipe the output into aggregation commands such as `sort`, `uniq`, or `wc`. 3. Check whether headers or empty lines should be removed.	Counting headers as data. Matching partial strings that should be excluded.	Data processing, text analysis, log inspection
Inspect service or process state	1. Locate relevant processes with `ps`, `pgrep`, or service commands. 2. Inspect logs or status output for failure evidence. 3. Restart or modify configuration only after identifying the cause.	Restarting before reading logs. Confusing process existence with service health.	System diagnosis, service repair, environment debugging

These examples cover common categories such as file operations, log management, data processing, and system diagnosis. They show the functional separation between Skills and Memories. Skills provide reusable procedures for completing tasks, while Memories provide constraints and recovery knowledge that help the agent avoid repeated mistakes.

Table 12: **Examples of L4 Skills.** L4 Skills capture longer planning routines that combine checking, execution, and verification.

Name	Steps	Common Errors	Applicable Scenarios
Safe destructive operation	1. Identify target files or resources. 2. Run a dry run or list operation. 3. Confirm that the matched targets satisfy the task goal. 4. Apply the destructive command. 5. Verify the final state.	Skipping dry run. Using broad wildcards. Failing to verify after modification.	Cleanup, migration, permission repair
Diagnose and repair environment failure	1. Reproduce the failure with the provided command. 2. Read the error message and locate the missing dependency or configuration issue. 3. Apply the minimal repair. 4. Re-run the original command. 5. Inspect residual errors before terminating.	Installing unrelated packages. Treating symptoms without reproducing the failure.	Build repair, dependency setup, repository diagnosis

Table 13: **Examples of Memories.** Memories provide warnings and recovery guidance derived from contrastive trajectory analysis.

Type	Layer	Rule	Evidence and Context
pitfall	L2	When using `find -mtime`, verify the meaning of `+N` and `-N` before filtering files by age.	Failed trajectories often used the wrong sign and selected recent files instead of old files. Successful trajectories first listed matched files and checked timestamps.
best_practice	L3	When passing file names between commands, prefer null separated output with `-print0` and `xargs -0`.	Failed trajectories broke on file names containing spaces. Successful trajectories used null separated pipelines before batch compression or deletion.
recovery_pattern	L4	After a command fails, inspect the exact error message before retrying with a modified command.	Failed trajectories repeatedly tried similar commands without reading stderr. Successful trajectories used the error text to identify missing paths, wrong permissions, or absent packages.

A.7 CASE STUDY

This section presents a qualitative case study of terminal execution with and without STRATA. Both trajectories take 14 steps and differ in repair and verification behavior. We use an incident response task in which the agent must validate several JSON event files against a schema, copy valid files into a target directory, extract high severity events, and generate two summary files. The task is challenging because it combines schema validation, exact error reporting, JSON field filtering, CSV generation, and final artifact verification.

Zero-shot trajectory. Without STRATA, the agent first inspects the input directory, checks whether validation tools are available, and reads both the schema and source JSON files. It then writes a Python validation script, but the first version generates an incorrect failure reason for the invalid event file. Instead of reporting `missing key source_ip`, it reports an incomplete schema error message. The agent notices the mismatch only after printing the report, then rewrites the script with more careful error extraction. Later, it generates the high severity JSON file but keeps an extra `details` field, which violates the requirement that each output object contain exactly five keys. This mistake requires another repair step. The final trajectory passes the verifier in 14 steps, including repairs to formatting and field selection.

STRATA enhanced trajectory. With STRATA, the agent retrieves a Skill for structured file validation and report generation, together with Memories about JSON Schema validation, exact output fields, and error driven recovery. The Skill suggests using a single Python script to load the schema, validate files, copy valid records, and generate all downstream artifacts in one controlled workflow. The Memories warn the agent to avoid unreliable command line validators when the environment may not contain the expected tool, to extract missing required keys from the exact validation error, and to verify that filtered JSON objects contain only the required fields. Guided by this knowledge, the agent first confirms that `python3` and the `jsonschema` package are available, writes a single script that handles validation and artifact generation, then verifies the report, copied files, filtered JSON, CSV summary, and count file before stopping.

Table 14: **Case study trajectory comparison.** The zero-shot agent eventually solves the task but requires several repair steps. The trajectories illustrate different repair and verification behavior with procedural Skills and failure-aware Memories; both use 14 steps.

Stage	Zero-shot	STRATA
Initial inspection	Inspects files, checks tools, reads schema and events.	Inspects the same task state, then uses retrieved knowledge to choose Python based validation and artifact generation.
Main execution	Writes an initial validation script, but produces an incorrect missing key reason.	Writes one script that validates each file, extracts missing keys, copies valid files, and prepares all required outputs.
Error handling	Prints intermediate outputs, discovers the wrong report text, and rewrites the validation logic.	Uses Memory guidance to parse validation errors before writing the report, avoiding the repeated repair step.
Output formatting	Creates high severity JSON with an extra field, then performs a later correction.	Uses Memory guidance to keep exactly the required output keys before saving the filtered JSON file.
Final behavior	Completes the task in 14 steps after multiple corrections.	Completes the task in 14 steps with fewer repairs and more direct verification.