NexForge: Scaling Agent Capabilities through Requirement-Driven Task Synthesis for LLMs

Jiarong Zhao[1], Zhikai Lei[2,*], Zhiheng Xi[2], Rui Zheng[2], Hang Yan[2], Jie Zhou[1,*], Qin Chen[1], Liang He[1]

[1]East China Normal University [2]Shanghai Qiji Zhifeng Co., Ltd

[*]Corresponding authors

Abstract

Synthesizing training data to scale agent capabilities in LLM post-training is bottlenecked by substrate-bound task synthesis: tasks are generated from fixed tools, repositories, or skill graphs, so expanding coverage requires manual substrate engineering, transferring to a new domain demands bespoke infrastructure, and the resulting distributions inherit substrate biases rather than reflecting real-world demand. We introduce NexForge, a requirement-driven framework that synthesizes diverse, executable agent tasks and expert trajectories for SFT from high-level capability requirements. NexForge first profiles real-world demand into representative scenarios and task profiles, then samples task forms per scenario. It then performs distribution-aware compilation, automatically retrieving or constructing files, repositories, dependencies, and runtime configurations to instantiate each task, followed by synthesizing expert rollouts and distilling trajectories. The same pipeline generates 3,600 terminal tasks and 2,000 office tasks without any domain-specific infrastructure, improving Qwen3.5-35B-A3B Base from 22.5% to 52.0% on Terminal-Bench 2.0 and from 813 to 1338 Elo on GDPval. Scaling to 43.2K terminal tasks further improves performance to 58.4%, surpassing Claude Opus 4.6. At scale, NexForge-synthesized trajectories supervise SFT of Nex-N2, a family of open agent models that advance Qwen3.5-35B-A3B to 75.3% on Terminal-Bench 2.1 and 1585 Elo on GDPval—achieving state-of-the-art open-source performance and surpassing several frontier proprietary systems. Nex-N2 models are available at `https://nex.sii.edu.cn/`.

1 Introduction

Building capable LLM agents for real-world tasks is bottlenecked by the scarcity of high-quality, diverse training data. Post-training these agents requires not only task descriptions, but also grounded materials, executable environments, and long-horizon interaction trajectories. Producing such data at

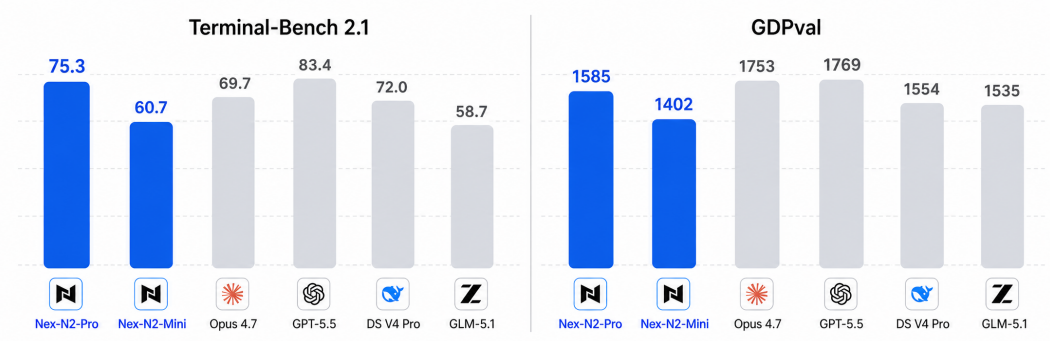

Figure 1: **Nex-N2 performance on Terminal-Bench 2.1 and GDPval via NexForge data scaling.**

scale remains labor-intensive: practitioners must determine what the agent should practice, design representative tasks, collect appropriate materials, and package them into runnable workspaces. This process is expensive, domain-specific, and difficult to scale.

Recent task-synthesis methods have made substantial progress by generating executable tasks tied to predefined repositories, tools, skills, or execution traces (Dong et al., 2026; Wang et al., 2026; Chen et al., 2026; Xie et al., 2025; Fan et al., 2026; Cheng et al., 2026; Yang et al., 2025; Jain et al., 2025). However, these substrate-bound approaches face three fundamental scaling limitations. *First*, scaling is bounded by the input substrate: the volume and diversity of generated tasks are constrained by the number of available repositories, tools, or skill specifications, and expanding coverage requires costly manual curation. *Second*, each domain typically demands a bespoke pipeline with domain-specific infrastructure – skill libraries for terminal tasks (Fan et al., 2026; Cheng et al., 2026), repository harnesses for software engineering (Yang et al., 2025; Jain et al., 2025), or simulated tool ecosystems (Dong et al., 2026) – making transfer to new capabilities costly. *Third*, the available substrate implicitly determines the task distribution: repository-based pipelines naturally favor implementation and debugging tasks even when real-world users also require configuration analysis, data processing, or system operation, producing distributions that reflect substrate biases rather than real-world demand.

We argue that scaling agent capabilities through LLM post-training requires decoupling task generation from predefined substrates. Environment synthesis should instead begin with capability requirements: before any workspace is built, the system must determine both which tasks are representative of the desired capability and the diverse real-world contexts in which they arise. Only then should the materials and runtime environments be constructed to match. This requirement-driven formulation separates *what an agent should practice* from *how that practice is made executable*, enabling the same pipeline to scale across domains without building new domain-specific infrastructure for each target.

To this end, we introduce NexForge, a requirement-driven framework that synthesizes diverse, executable agent tasks and expert trajectories for SFT, all from high-level capability requirements. NexForge first profiles real-world demand into representative scenarios and task profiles, then samples task forms for each scenario. It then applies distribution-aware compilation, combining sampled task forms with concrete scenarios into a task directive for subsequent instantiation. Finally, NexForge instantiates each task into a complete, executable workspace, and synthesizes expert rollouts and distills trajectories. By doing so, NexForge controls training data composition independently of predefined tools or repositories and applies the same synthesis pipeline across substantially different capability domains. We evaluate NexForge on two disparate capabilities: 3,600 tasks for terminal operations improve Qwen3.5-35B-A3B from 22.5% to 52.0% on Terminal-Bench 2.0; 2,000 tasks for office work improve the same model from 813 to 1338 Elo on GDPval, both synthesized without domain-specific engineering. Moreover, scaling up the training data further enables Nex-N2 to reach state-of-the-art open-source performance on both capabilities, as shown in Figure 1.

Our contributions are threefold:

- We formulate environment scaling for agent post-training as a *requirement-driven scaling problem*, identifying three bottlenecks of substrate-bound synthesis: input-bounded scale, high transfer cost, and substrate-biased distributions.
- We propose NexForge, an end-to-end framework that profiles real-world demand, instantiates executable training environments, and collects expert trajectories, without domain-specific infrastructure.
- We demonstrate that the same framework generates effective training data for both terminal and office capabilities, substantially improving the performance of the downstream agent in distinct domains.

2 Related Works

Instruction and task-distribution synthesis. Scalable data synthesis often decomposes a broad capability space before generating individual examples. Self-Instruct (Wang et al., 2023) bootstraps instruction-following data from a small seed set, while GLAN (Li et al., 2024) organizes generation through a hierarchical taxonomy of disciplines and tasks. TRouter (Liu et al., 2026) controls synthetic

examples using explicit task profiles, and Benchmark Agent (Xiong et al., 2026) decomposes evaluation goals into structured benchmark specifications. These studies show that explicit distribution modeling improves coverage over unconstrained generation, but they primarily synthesize text instructions or evaluation benchmarks rather than complete executable environments. Extending these approaches to agent training would additionally require grounding each task with materials, dependencies, and runtimes—a step these methods do not address.

Agent task and environment synthesis. Recent work has explored executable agent-data construction through simulated environments and tool ecosystems (Dong et al., 2026; Wang et al., 2026), execution traces and subtask composition (Chen et al., 2026; Xie et al., 2025; Shi et al., 2025), and domain-specific repositories and skills (Fan et al., 2026; Cheng et al., 2026; Yang et al., 2025; Jain et al., 2025). Agent-World (Dong et al., 2026) and Agent World Model (Wang et al., 2026) synthesize stateful tool environments, while DIVE (Chen et al., 2026) derives grounded and verifiable tasks from execution evidence. AgentSynth (Xie et al., 2025) and TaskCraft (Shi et al., 2025) construct complex tasks through subtask composition and difficulty control. For terminal agents, SkillSynth (Fan et al., 2026) generates tasks from structured skill specifications, Terminal-World (Cheng et al., 2026) builds environments around reusable agent skills, CLI-Universe (Hua et al., 2026) constructs terminal tasks from a predefined capability taxonomy, and TerminalTraj (Wu et al., 2026), Nemotron-Terminal (Pi et al., 2026), LiteCoder-Terminal (Peng et al., 2026), and OpenThoughts-Agent (Raoof et al., 2026) collect or curate terminal interaction trajectories and training corpora. In software engineering, SWE-smith (Yang et al., 2025) and R2E-Gym (Jain et al., 2025) generate executable tasks from repositories, tests, and issue-like workflows.

These methods establish the importance of executable environments but share three scaling limitations with substrate-bound approaches. *(1) Input-bounded scale*: the volume and diversity of generated tasks are capped by the predefined substrate—repositories, skills, tools, or traces—and expanding coverage requires manual substrate curation. *(2) High transfer cost*: each method builds domain-specific infrastructure (skill libraries for terminal tasks, repository harnesses for software engineering, simulated tool ecosystems for general agents) that cannot be directly reused for a different capability target. *(3) Substrate-biased distributions*: because tasks are derived from what the substrate supports, the resulting corpus tends to overrepresent convenient task types (e.g., implementation and debugging) at the expense of less substrate-friendly but equally important work (e.g., configuration, data processing, system operation). NexForge addresses all three by starting from high-level capability requirements rather than predefined substrates: the same pipeline scales across distinct agent capabilities (terminal and office) without new domain-specific infrastructure.

3 METHOD

NexForge scales from a high-level user requirement to diverse executable workspaces and interaction trajectories for agent post-training. As illustrated in Figure 2, the framework contains three main stages. First, it profiles real-world demand to discover representative task demands and working scenarios associated with the requested capability. Second, it composes the discovered demands into concrete task directives under an explicit task distribution. Finally, it retrieves or constructs the required materials, packages each directive into an executable workspace, and collects teacher trajectories for training. This requirement-driven pipeline determines *what capabilities the agent should acquire* before constructing *the materials and runtime environments that make the practice executable*, enabling the same architecture to scale across agent capability targets.

3.1 PROBLEM FORMULATION

NexForge takes as input a high-level capability requirement I, a requested task count N, and optional batch-level constraints B that let the user override the induced distribution along specific dimensions such as language or difficulty. It outputs a collection

$$\mathcal{T} = \{\tau_1, \ldots, \tau_N\}, \tag{1}$$

where each τ_i is an executable task package containing a task instruction, grounded materials, workspace files, software dependencies, and a runtime configuration, designed for trajectory collection and agent training.

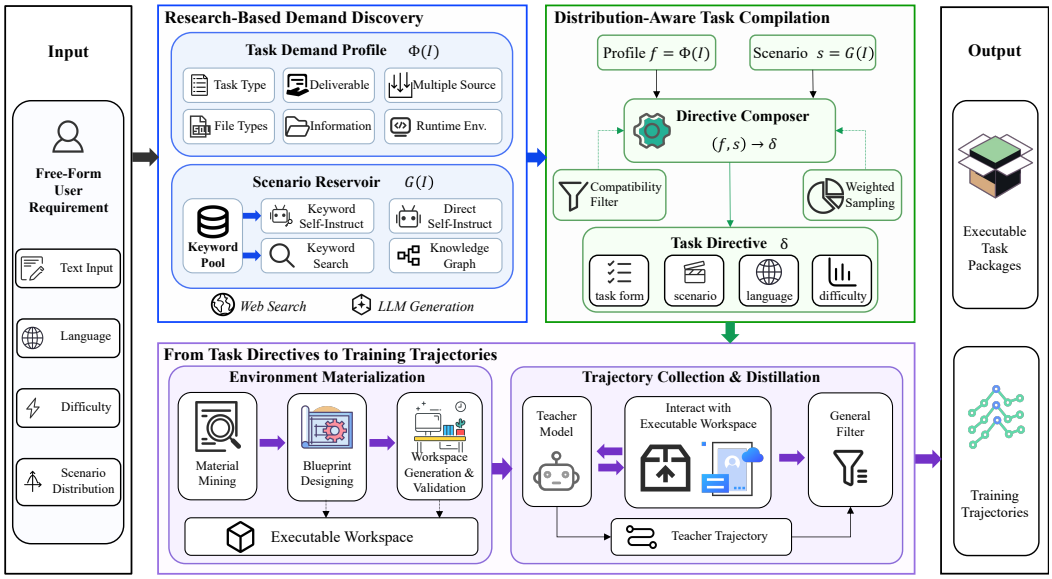

Figure 2: **Overview of NexForge.** Given a high-level user requirement, NexForge first profiles real-world demand to construct a task demand profile and scenario reservoir, then composes distribution-aware task directives across scenarios. Each directive is subsequently instantiated into an executable workspace, where teacher interactions are collected and distilled into training trajectories.

We represent each task using a task form f_i and a scenario s_i:

$$\tau_i \leftarrow (f_i, s_i). \tag{2}$$

The task form $f_i = (t_i, d_i, \sigma_i, e_i, \ell_i, h_i)$ specifies the primary task type t_i, expected deliverable d_i, source strategy σ_i, runtime environment e_i, language ℓ_i, and difficulty h_i. The scenario describes the concrete context in which the task occurs, such as an organization, professional role, workflow, software system, or operational event. This decomposition enables explicit control over the corpus-level task distribution while grounding each task in a realistic context.

3.2 Research-Based Demand Discovery

Given the requirement I, NexForge first investigates what the requested capability entails in practice. A web-enabled research agent collects evidence from professional workflows, technical documentation, role descriptions, public examples, and other sources relevant to the domain. The collected evidence, together with model knowledge, is organized into a weighted *task demand profile* and a diverse *scenario reservoir*.

Task demand profile. The task demand profile $\Phi(I)$ defines the intended distribution over the dimensions of the task form:

$$\Phi(I) = \{\Phi_t, \Phi_d, \Phi_\sigma, \Phi_e, \Phi_\ell, \Phi_h\}, \tag{3}$$

where each Φ_k is a weighted categorical distribution. Candidate options are derived from research evidence and model knowledge, and are subsequently reviewed by an LLM to merge near-duplicates, remove ambiguous categories, and improve mutual exclusivity. Evidence-derived weights govern the relative sampling proportion of each option for the requested capability.

The profile specifies the overall composition of the generated corpus without committing to particular repositories, documents, or source files. It therefore separates the intended work distribution from the materials later used to instantiate individual tasks.

Scenario reservoir. The scenario reservoir $\mathcal{G}(I)$ contains concrete scenario guides (*scenarios* for short) that describe working contexts in which task forms can be instantiated. NexForge populates it

through four mechanisms: (1) *keyword-conditioned self-instruct*, which samples keywords from a domain keyword pool and prompts the model to generate scenarios; (2) *keyword-conditioned research*, which uses sampled keywords as web-search queries to discover real-world scenarios; (3) *self-instruct*, which directly prompts the model to generate scenarios, sampling from already accepted scenarios as few-shot examples to encourage novelty; and (4) *knowledge-graph expansion*, which iteratively expands sub-scenarios from the input requirement. The domain keyword pool is generated via web research or model knowledge. The relative sampling proportion of each mechanism can be configured through the batch-level constraints B. All candidate scenarios undergo embedding-based semantic deduplication to ensure diversity across the reservoir.

Each scenario intentionally describes only the working context and omits task type, deliverable, source strategy, and runtime. This prevents scenarios from independently determining the task distribution.

3.3 Distribution-Aware Task Compilation

NexForge composes the task-form profile and scenario reservoir into directives. For each scenario $g_j \in \mathcal{G}(I)$, the composer instantiates it into a concrete scenario, then sequentially filters profile options by compatibility with the scenario and previously selected fields.

Let $K = \{t, d, \sigma, e\}$ denote the ordered task-form dimensions that require compatibility decisions. For each dimension $k \in K$, the composer obtains a compatible candidate set

$$C_{j,k} = F_\theta(I, g_j, k, \Phi_k, S_{j,<k}), \tag{4}$$

where $S_{j,<k}$ contains options selected in earlier steps. An option is then sampled according to the profile weights restricted to the compatible set:

$$f_{j,k} \sim \text{Normalize}\left(\Phi_k|_{C_{j,k}}\right). \tag{5}$$

Language and difficulty are sampled according to the corresponding profile distributions and batch constraints B.

The resulting directive is

$$\delta_j = \langle g_j, f_j, A_j, \ell_j, h_j, R_j \rangle, \tag{6}$$

where A_j records the compatible candidate sets and R_j stores the composition rationale. The directive determines the intended work before any repository or document is selected, constraining downstream stages to find or construct materials that realize the sampled task. The compatibility filter ensures that the resulting directive is internally consistent; its effect on directive quality is evaluated in Section 5.4.

3.4 Agent Post-Training Trajectory Generation

A task directive specifies the intended work but is not yet executable. NexForge instantiates each directive into an executable workspace and subsequently collects and distills teacher interactions into post-training trajectories.

Environment instantiation. Given a task directive δ_j, NexForge instantiates the intended task through three successive steps: material mining, blueprint planning, and workspace generation. A research agent first identifies realistic workflows and collects the resources required by the directive, including public repositories, technical documents, datasets, spreadsheets, configuration files, and other domain-relevant artifacts. When suitable external resources are unavailable, the research agent specifies the local materials that should be generated. A planning agent then inspects the collected resources and produces a structured blueprint defining the task objective, input materials, expected deliverable, workspace organization, required transformations, software dependencies, and runtime constraints. Based on this blueprint, a coding agent constructs the executable workspace by adapting repositories, retrieving public files, generating local artifacts, installing dependencies, and configuring an unprivileged CPU-only Docker environment. Automated checks verify material completeness, source-strategy consistency, dependency availability, runtime restrictions, and the absence of leaked solutions or expected deliverables. Separating research, planning, and implementation into distinct stages isolates noise introduced during resource discovery and data preparation, preventing it from propagating into downstream workspace construction.

Trajectory collection and distillation. Each workspace is assigned to a teacher model under a fixed interaction budget. The teacher model interacts with the executable environment using the available tools, producing a trajectory that records model responses, tool calls, environment observations, intermediate failures, and recovery behaviors. NexForge then converts the collected interactions into a standardized training format through task-independent trajectory cleaning, removing malformed conversations, invalid message structures, duplicate-role messages, error-only outputs, and trivially short interactions. The framework does not require manually authored reference answers or task-specific success verifiers; therefore, incomplete but meaningful trajectories are retained when they contain useful long-horizon reasoning, tool-use patterns, or error-recovery signals. This process produces both controllable executable task packages and requirement-aligned training trajectories that can be directly used for agent post-training.

4 DATASETS

We apply NexForge to two independently written capability requirements, producing **Terminal-3.6K** for terminal agent post-training and **Office-2K** for office agent post-training. The two corpora differ substantially in their task forms, grounding materials, and runtime requirements, demonstrating that the same synthesis pipeline can scale across distinct user-defined capabilities without domain-specific engineering. To study data scaling, we additionally produce Terminal-2K, Terminal-43.2K, and Office-22K using the same pipeline.

4.1 TASK-SET SPECIFICATIONS

The terminal specification covers agent command-line capabilities such as repository inspection, software building, configuration editing, data processing, system operation, and local validation, while the office specification includes agent office-work capabilities such as spreadsheet analysis, document review, information aggregation, summarization, drafting, planning, and evidence-based recommendations. Both use a 50/50 Chinese–English split and balanced difficulty distributions. Terminal-3.6K contains 3,600 tasks, matching SkillSynth in task count and interaction budget, whereas Office-2K contains 2,000 independently synthesized office tasks.[1]

4.2 CORPUS STATISTICS AND ROLLOUTS

Table 1 summarizes corpus scale, trajectory complexity, task diversity, and grounding characteristics. We collect 3 teacher rollouts per task for all reported corpora. All synthesis agents are driven by GPT-5.5. After trajectory conversion and task-independent trajectory cleaning, this yields 8,521 trajectories for Terminal-3.6K (2.37 per task) and 5,940 for Office-2K (2.97 per task), reflecting natural failure rates.

The two specifications induce clearly different data distributions. Terminal-3.6K realizes 24 task types and is primarily grounded through public repository adaptation, whereas Office-2K realizes 15 task types and more frequently relies on public documents, spreadsheets, and reports. The ten most frequent task-form signatures account for only 20.6% and 15.1% of the two corpora, respectively, indicating that scale is not obtained by repeatedly instantiating a small number of templates.

Unless otherwise specified, SFT uses Qwen3.5-35B-A3B Base for five epochs with a maximum context length of 262K, packed bfloat16 sequences, a global batch size of 64, and a learning rate of 10^{-5} with cosine decay and 5% warmup. The Qwen3-32B comparison uses Terminal-3.6K with the same configuration adapted to the dense base model.

4.3 COMPARISON WITH EXISTING AGENT DATASETS

Table 2 compares NexForge with representative agent-task synthesis pipelines. Existing methods typically start from predefined tools, repositories, traces, atomic tasks, or manually designed skill structures, each requiring domain-specific infrastructure that limits scale and transfer. NexForge instead takes a high-level user requirement as input, profiles real-world demand to identify representative task demands and their relative prevalence, and then constructs the materials and

[1] All synthesis stages use GPT-5.5.

Table 1: **Corpus statistics.** Task counts, trajectory scale, and ablation variants. *w/o profile* removes task-form sampling; *w/o scenario* replaces grounded scenarios with neutral seeds.

Statistic	Term.-3.6K	Office-2K	w/o profile	w/o scenario
Executable tasks	3,600	2,000	2,000	2,000
Converted trajectories	8,521	5,940	4,574	4,541
Trajectories per task	2.37	2.97	2.29	2.27
Avg. tokens per traj.	129K	107K	124K	154K
Avg. tool calls per traj.	123	54	116	133
Chinese / English	50/50	50/50	50/50	50/50
Difficulty allocation	Balanced	Balanced	Balanced	Balanced
Realized task types	24	15	$\sim$3	$\sim$16
Top-10 signature cov.	20.6%	15.1%	78.5%	24.9%
Mean package size	10.9 MB	9.4 MB	5.8 MB	9.6 MB

Table 2: **Dataset comparison.** Columns: requirement-driven design (Req.), scenario reservoir (Res.), distribution control (Dist.), environment materialization (Env.), and cross-capability transfer (Cap.). $\triangle$: partial support.

Dataset	Synthesis Prior	Scale	Req.	Res.	Dist.	Env.	Cap.
AgentSynth (Xie et al., 2025)	Composable subtasks	>6K	–	–	$\triangle$	–	$\triangle$
TaskCraft (Shi et al., 2025)	Atomic tasks & tools	$\sim$36K	–	–	$\triangle$	–	$\triangle$
DIVE (Chen et al., 2026)	Executed tool traces	48K+3.2K	–	–	$\triangle$	–	$\triangle$
R2E-Gym (Jain et al., 2025)	Repos & commits	>8.7K	–	–	–	✓	–
SWE-smith (Yang et al., 2025)	Repos & tests	50K	–	–	–	✓	–
SkillSynth (Fan et al., 2026)	Skill graph	3.6K	–	–	$\triangle$	✓	–
Term.-World (Cheng et al., 2026)	Agent skills	5.7K	–	–	$\triangle$	✓	–
CLI-Univ. (Hua et al., 2026)	Capability taxonomy	6K	–	✓	$\triangle$	✓	–
NexForge	**Free-form requirement**	**5.6K / 14.5K**	✓	✓	✓	✓	✓

executable environments needed to instantiate them. This requirement-driven design separates corpus-level task distribution from grounding resources and enables scalable agent post-training across diverse capability domains through the same synthesis pipeline.

5 Experiments

5.1 Experimental Setup

Training. We perform full-parameter SFT on Qwen3-32B and Qwen3.5-35B-A3B Base for five epochs with a maximum context length of 262K, packed bfloat16 sequences, a global batch size of 64, a learning rate of 10^{-5} with cosine decay and 5% warmup, on 128 H100 GPUs. The Qwen3-32B comparison uses Terminal-3.6K with the same configuration adapted to the dense base model.

Evaluation. We evaluate on Terminal-Bench 2.0 (Terminal-Bench Team, 2026) and GDPval (Patwardhan et al., 2025) to assess agent capabilities on terminal and office tasks. All controlled runs of our models use the NexAU evaluation harness with consistent tools and inference settings. Baseline results are from published papers or public leaderboards and may use different scaffolds. We compare our models with frontier proprietary models and terminal-specialized 32B baselines (Peng et al., 2026; Wu et al., 2026; Raoof et al., 2026; Pi et al., 2026; Fan et al., 2026; Hua et al., 2026; Cheng et al., 2026).

5.2 Main Results

Table 3 presents the results on Terminal-Bench 2.0. Three observations stand out. First, Terminal-3.6K consistently improves models with substantially different initial capabilities: Qwen3-32B increases from $5.6\%\pm0.9$ to $32.3\%\pm2.4$, while Qwen3.5-35B-A3B increases from $22.5\%\pm0.9$ to $52.0\%\pm3.6$, corresponding to absolute gains of 26.7 and 29.5 points, respectively. This suggests that the

Table 3: **Terminal-Bench 2.0 pass@1 accuracy.** Left: frontier proprietary models; right: terminal-specialized 32B models. NexAU rows are our controlled runs.

Baselines				Specialized		
Model	**Params**	**Scaffold**	**TB 2.0**	**Model**	**Scaffold**	**TB 2.0**
Gemini 3.1 Pro	–	TongAgents	80.2 ± 2.6	LiteCoder-Terminal-32B-SFT	Terminus 2	18.5 ± 3.4
DeepSeek-V4-Pro	1.6T	official	67.9	TerminalTraj-32B	Terminus 2	22.0 ± 5.8
Claude Opus 4.6	–	Claude Code	58.0 ± 2.9	OpenThinkerAgent-32B	Terminus 2	26.2 ± 1.6
MiniMax M2.7	230B	official	57.0	Nemotron-Terminal-32B	Terminus 2	27.4 ± 2.4
GPT-5.2	–	Terminus 2	54.0 ± 2.9	Qwen3-32B + SkillSynth	Terminus 2	29.6 ± 1.6
GLM 5	744B	Terminus 2	52.4 ± 2.6	CLI-Universe-32B (DS)	Terminus 2	31.2
Claude Opus 4.5	–	OpenHands	51.9 ± 2.9	Terminal-World-32B	Terminus 2	31.5
Gemini 3 Flash	–	Terminus 2	51.7 ± 3.1	**Ours**		
GPT-5.1	–	Terminus 2	47.6 ± 2.8			
Kimi K2.5	1T	Terminus 2	43.2 ± 2.9	Qwen3-32B	NexAU	5.6 ± 0.9
MiniMax M2.5	230B	Terminus 2	42.2 ± 2.6	+ Terminal-3.6K	NexAU	32.3 ± 2.4
DeepSeek-V3.2	671B	Terminus 2	39.6 ± 2.8	Qwen3.5-35B-A3B Base	NexAU	22.5 ± 0.9
Qwen 3 Coder	480B	Terminus 2	23.9 ± 2.8	+ Terminal-3.6K	NexAU	$\mathbf{52.0} \pm \mathbf{3.6}$

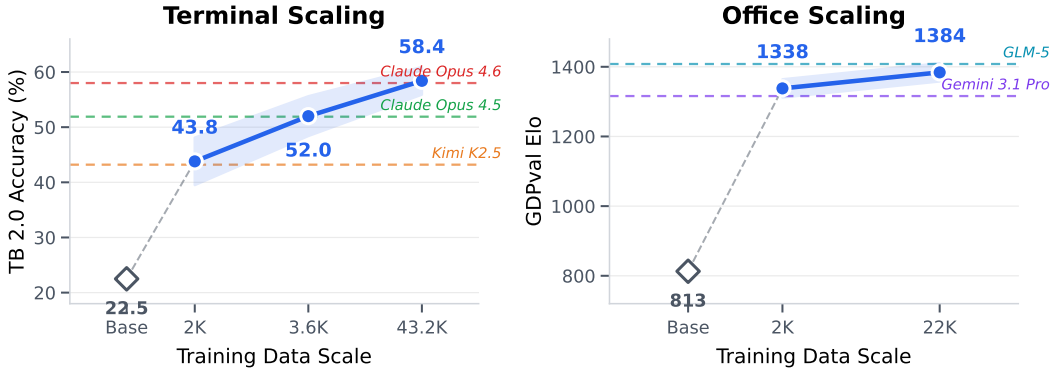

Figure 3: **Data scaling in two domains.** Terminal-Bench 2.0 accuracy (left) and GDPval Elo (right) as a function of NexForge data scale on Qwen3.5-35B-A3B. Dashed lines mark frontier reference models. Performance improves monotonically with data volume, and the same pipeline transfers across domains with only a specification change.

synthesized data provides useful terminal interaction patterns that improve agent capabilities rather than merely compensating for weaknesses of a particular base model. Second, NexForge-trained Qwen3-32B achieves the highest mean accuracy among the compared terminal-specialized 32B models, despite NexForge being a general requirement-driven framework rather than a pipeline tailored to terminal tasks. Third, combining Terminal-3.6K with the stronger Qwen3.5 base brings the resulting model into the performance range of several frontier proprietary systems, substantially narrowing the gap without changing the model architecture. Overall, the results show that requirement-driven environment synthesis produces high-quality post-training data that scales agent capabilities across model scales and remains competitive with domain-specific data construction pipelines.

5.3 Scaling Analysis

Beyond the primary Terminal-3.6K run, NexForge scales along two independent axes, both requiring only configuration changes rather than manual curation or domain-specific engineering.

Domain transfer. Because NexForge is requirement-driven, transferring to a new domain requires only a new capability specification—no domain-specific task generator, tool collection, or pipeline redesign. To demonstrate this, we replace the terminal requirement with an office-work specification while keeping the pipeline unchanged. NexForge constructs 2,000 tasks covering document

Table 4: **GDPval Elo results.** Office capability evaluation. Elo is calibrated to the GDPval-AA v1 (Artificial Analysis, 2025) with GPT-5.1 anchored at 1000.

Group	Model	GDPval Elo
Baselines	GPT-5.4	1667
	Claude Sonnet 4.6	1633
	GLM-5	1408
	Gemini 3.1 Pro	1316
	MiniMax M2.5	1202
	Gemini 3 Flash	1191
	GPT-5.1	1000
Ours	Qwen3.5-35B-A3B Base	813
	+ Office-2K	**1338**

processing, spreadsheet analysis, planning, communication, and evidence-based recommendations. As shown in Table 4, Office-2K improves Qwen3.5-35B-A3B Base from 813 to 1338 GDPval Elo, surpassing several frontier reference models. Scaling to 22K office tasks yields 1384 Elo, further closing the gap with GLM-5 (1408) to 24 Elo. The same pipeline that produces terminal data thus enables agent post-training for an entirely different capability with only a specification change.

Data volume scaling. Within any target domain, NexForge's data production is not bottlenecked by manually curated materials or substrate availability—scaling requires only increasing the number of tasks. On Terminal-Bench 2.0, scaling from Terminal-2K (43.8%) to Terminal-3.6K (52.0%) to Terminal-43.2K (58.4%) yields continuous improvement in agent performance, as shown in Table 3 and Figure 3: Terminal-2K matches Kimi K2.5, Terminal-3.6K matches Claude Opus 4.5, and Terminal-43.2K surpasses Claude Opus 4.6, confirming that NexForge-generated data effectively scales agent capabilities. The same pattern holds on GDPval: Office-2K (1338) to Office-22K (1384) shows consistent gains. Since GDPval Elo depends on the comparison pool, comparisons with public models are treated as contextual references; the controlled base-versus-scaled comparison provides the primary evidence.

5.4 Ablation Studies

We examine whether task-form control and scenario grounding provide complementary signals during task compilation. Under the same 2,000 tasks with 3 rollouts per task and SFT configuration, we construct three variants on Qwen3.5-35B-A3B Base. **Terminal-2K w/o profile** retains the scenarios but removes the sampled task-form directive, allowing the teacher model to infer tasks directly from each scenario. **Terminal-2K w/o scenario** retains the task-form profile but replaces concrete scenarios with neutral seeds. **Terminal-2K** corresponds to the complete NexForge pipeline, where task forms are sampled conditioned on concrete scenarios.

DATC directive quality. On the terminal run, the compatibility filter retains on average only 4.4 of 24 task types, 5.2 of 8 deliverables, 2.7 of 4 source strategies, and 4.5 of 20 runtimes per scenario, so just 2.0% of the full $24\times8\times4\times20$ task-form combination space is scenario-compatible. To evaluate whether DATC produces coherent directives under this constraint, we ask an LLM judge (Qwen3.7-Max) to assess whether each task-form directive is compatible with its guide scenario, and compare against a uniform-random baseline that ignores both profile weights and compatibility filters (*w/o DATC*). Over the full 2,000 tasks of Terminal-2K, DATC directives achieve a match rate of 81.0%, while random directives achieve only 7.0%, as reported in Table 5. The large gap confirms that scenario-conditioned compilation is essential for producing coherent task directives.

Table 6 shows that the full pipeline achieves the highest mean accuracy. Figure 4(c,d) reveals the underlying mechanism: w/o profile collapses to a single dominant task type (78.5%, $\sim$3 effective types), while w/o scenario broadens coverage ($\sim$16 effective types) but yields a flat distribution that under-represents frequent workflows. Terminal-2K strikes an intermediate balance ($\sim$6 effective types) with the longest trajectories (114 assistant messages, 149 tool calls). Task-form control preserves coverage; scenario grounding structures the training distribution.

Table 5: **Directive compatibility.** LLM-judge match rate between directives and scenarios. *w/o DATC* replaces scenario-conditioned compilation with uniform-random task-form assignment.

Corpus	Method	Match count	Match rate
Terminal-2K	DATC	1,620 / 2,000	81.0%
	w/o DATC	140 / 2,000	7.0%

Table 6: **Ablation studies.** Asst./Tool: mean assistant messages and tool calls per trajectory. TB 2.0: pass@1 accuracy on Terminal-Bench 2.0.

Variant	Scenario	Task Form	Asst.	Tool	TB 2.0
Terminal-2K	Reservoir	Profile	114	149	**43.8** $\pm$ **4.3**
– w/o profile	Reservoir	None	98	133	40.2 $\pm$ 5.0
– w/o scenario	Neutral seed	Profile	95	125	42.7 $\pm$ 4.2

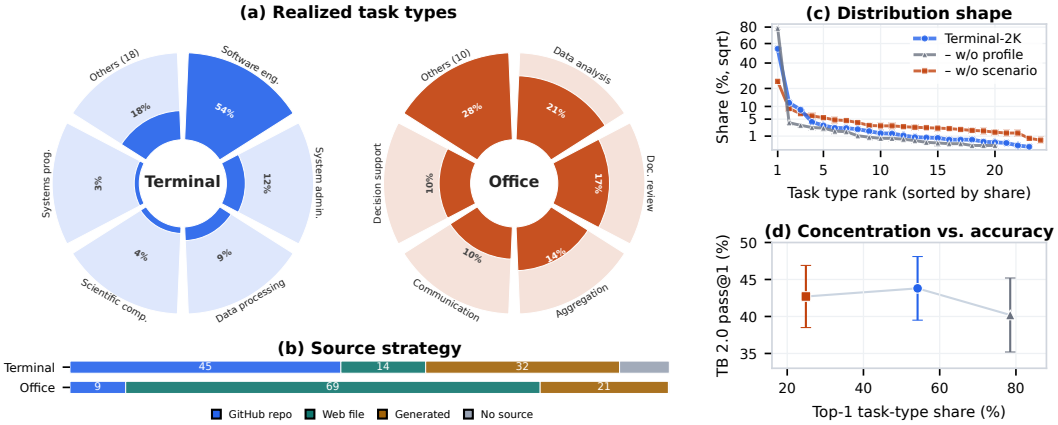

Figure 4: **Distribution analysis.** (a) Task-type frequencies, (b) grounding-source composition, (c) assistant message counts, and (d) tool-call counts per trajectory across ablation variants.

5.5 Further Analysis

Cross-domain distribution shift. Figure 4(a,b) shows NexForge adapts to different requirements. The terminal corpus is dominated by software engineering (54%) but covers 23 additional types; the office corpus is more evenly distributed. Grounding differs accordingly: 45% of terminal tasks use public repositories vs. 69% of office tasks using web files. These shifts arise from input specifications alone, confirming cross-domain scalability.

Controlled diversity. The scenario reservoir contains 5,600 accepted scenarios with no exact or embedding near-duplicates under a cosine-similarity threshold of 0.85. The generated corpora also avoid repeated task configurations: the ten most frequent task-form signatures account for only 20.6% of terminal tasks and 15.1% of office tasks. Meanwhile, terminal and office tasks can be separated with 0.98 nearest-centroid accuracy, indicating that NexForge preserves within-domain diversity while producing clearly differentiated distributions across capability targets.

Environment quality. We manually inspect stratified task packages on instruction clarity, material completeness, workspace executability, difficulty calibration, and absence of solution leakage. Overall quality is high: the majority of packages score at least 4 on all five dimensions, and none receives a score below 3. Terminal-2K w/o profile and Terminal-2K obtain comparable quality, while Terminal-2K w/o scenario scores lower because its tasks generated without scenario grounding contain fewer grounded materials. However, Terminal-2K w/o scenario still outperforms Terminal-2K w/o profile on Terminal-Bench 2.0, suggesting that the performance gap between ablations is not

explained solely by package quality. Together with Figure 4(c,d), these results indicate that task-form control and scenario grounding mainly improve training by shaping a more effective task distribution.

6 Conclusion and Future Work

We present NexForge, a requirement-driven framework that scales agent post-training data synthesis from high-level capability requirements. By decoupling task-form control and scenario grounding from predefined substrates, NexForge enables the same pipeline to scale agent capabilities across distinct domains without domain-specific infrastructure. Experiments on terminal and office tasks demonstrate strong cross-domain scaling of agent capabilities. NexForge scales to tens of thousands of tasks, training the publicly released Nex-N2 model family: Nex-N2-Pro reaches 75.3% on Terminal-Bench 2.1 and 1585 Elo on GDPval, competitive with frontier proprietary systems, validating that requirement-driven synthesis effectively scales agent capabilities at production scale. Ablations confirm that combining task-form control with scenario grounding is essential for producing coherent directives. Future work will extend NexForge with automatically generated reference answers and machine-checkable verifiers, enabling its use in benchmark construction, reinforcement learning, and other settings that require reliable outcome-based feedback.

References

Artificial Analysis. GDPval-AA v1: Standardized elo evaluation for economically valuable tasks. `https://artificialanalysis.ai/evaluations/gdpval-aa`, 2025.

Aili Chen, Chi Zhang, Junteng Liu, Jiangjie Chen, Chengyu Du, Yunji Li, Ming Zhong, Qin Wang, Zhengmao Zhu, Jiayuan Song, Ke Ji, Junxian He, Pengyu Zhao, and Yanghua Xiao. DIVE: Scaling diversity in agentic task synthesis for generalizable tool use. *arXiv preprint arXiv:2603.11076*, 2026.

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

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. *arXiv preprint arXiv:2604.18292*, 2026.

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. *arXiv preprint arXiv:2604.25727*, 2026.

Zhanbo Hua, Yifan Yao, Weihao Xie, Yongchi Zhao, Minghao Liu, Ruizhi Qiu, Zhewei Huang, Zun Wang, Yiyan Ji, Yunhai Ye, Letian Zhu, Xinping Lei, Han Li, Zhiyuan Ma, Zili Wang, Zhaoxiang Zhang, and Jiaheng Liu. CLI-Universe: Towards verifiable task synthesis engine for terminal agents. *arXiv preprint arXiv:2606.22883*, 2026.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents. *arXiv preprint arXiv:2504.07164*, 2025.

Haoran Li, Qingxiu Dong, Zhengyang Tang, Chaojun Wang, Xingxing Zhang, Haoyang Huang, Shaohan Huang, Xiaolong Huang, Zeqiang Huang, Dongdong Zhang, Yuxian Gu, Xin Cheng, Xun Wang, Si-Qing Chen, Li Dong, Wei Lu, Zhifang Sui, Benyou Wang, Wai Lam, and Furu Wei. Synthetic data (almost) from scratch: Generalized instruction tuning for language models. *arXiv preprint arXiv:2402.13064*, 2024.

Hui Liu, Bin Zou, Kecheng Chen, Jie Liu, Wenya Wang, and Haoliang Li. Task-aware llm routing with multi-level task-profile-guided data synthesis for cold-start scenarios. *arXiv preprint arXiv:2604.09377*, 2026.

Tejal Patwardhan, Rachel Dias, Elizabeth Proehl, Grace Kim, Michele Wang, Olivia Watkins, Simón Posada Fishman, Marwan Aljubeh, Phoebe Thacker, Laurance Fauconnet, Natalie S. Kim, Patrick Chao, Samuel Miserendino, Gildas Chabot, David Li, Michael Sharman, Alexandra Barr, Amelia Glaese, and Jerry Tworek. GDPval: Evaluating ai model performance on real-world economically valuable tasks. *arXiv preprint arXiv:2510.04374*, 2025.

Xiaoxuan Peng, Kaiqi Zhang, Xinyu Lu, Boxi Cao, Yaojie Lu, Hongyu Lin, Xianpei Han, and Le Sun. LiteCoder-Terminal: Scaling long-horizon terminal environments for learning language agents. *arXiv preprint arXiv:2605.29559*, 2026.

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

Negin Raoof, Richard Zhuang, Marianna Nezhurina, Etash Guha, Atula Tejaswi, Ryan Marten, Charlie F. Ruan, Tyler Griggs, Alexander Glenn Shaw, Hritik Bansal, E. Kelly Buchanan, Artem Gazizov, Reinhard Heckel, Chinmay Hegde, Sankalp Jajee, Daanish Khazi, Emmanouil Koukoumidis, Xiangyi Li, Hange Liu, Shlok Natarajan, Harsh Raj, Nicholas Roberts, Ethan Shen, Nishad Singhi, Michael Siu, Ashima Suvarna, Hanwen Xing, Patrick Yubeaton, Robert Zhang, Leon Liangyu Chen, Xiaokun Chen, Steven Dillmann, Saadia Gabriel, Xunyi Jiang, Anurag Kashyap, Boxuan Li, Yein Park, Minh Pham, Sujay Sanghavi, Lin Shi, Ke Sun, Yixin Wang, Zhiwei Xu, Erica Zhang, Siyan Zhao, Wanjia Zhao, Jenia Jitsev, Alex Dimakis, Benjamin Feuer, and Ludwig Schmidt. Openthoughts-agent: Data recipes for agentic models. *arXiv preprint arXiv:2606.24855*, 2026.

Dingfeng Shi, Jingyi Cao, Qianben Chen, Weichen Sun, Weizhen Li, Hongxuan Lu, Fangchen Dong, Tianrui Qin, King Zhu, Minghao Liu, Jian Yang, Ge Zhang, Jiaheng Liu, Changwang Zhang, Jun Wang, Yuchen Eleanor Jiang, and Wangchunshu Zhou. Taskcraft: Automated generation of agentic tasks. *arXiv preprint arXiv:2506.10055*, 2025.

Terminal-Bench Team. Terminal-Bench 2.0 leaderboard. `https://www.tbench.ai/leaderboard/terminal-bench/2.0`, 2026. Accessed 2026-06-22.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*, pp. 13484–13508, 2023.

Zhaoyang Wang, Canwen Xu, Boyi Liu, Yite Wang, Siwei Han, Zhewei Yao, Huaxiu Yao, and Yuxiong He. Agent world model: Infinity synthetic environments for agentic reinforcement learning. *arXiv preprint arXiv:2602.10090*, 2026.

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

Jingxu Xie, Dylan Xu, Xuandong Zhao, and Dawn Song. Agentsynth: Scalable task generation for generalist computer-use agents. *arXiv preprint arXiv:2506.14205*, 2025.

Shiyun Xiong, Dongming Wu, Peiwen Sun, Yuang Ai, Bokang Yang, Wencheng Han, Xiao-Hui Li, and Xiangyu Yue. Benchmark everything everywhere all at once. *arXiv preprint arXiv:2606.06462*, 2026.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. *arXiv preprint arXiv:2504.21798*, 2025.

APPENDIX

This supplementary material documents the implementation details, complete statistics, and additional analyses that support the main paper's claims on scalable agent post-training data synthesis. It is organized as follows. We first give the formal scenario-conditioned task-compilation procedure

(Algorithm 1) and then walk through a complete end-to-end case study of one synthesized terminal task, from the sampled directive to the runnable package. We next report the synthesis configuration, the two capability requirement specifications that define the pipeline inputs, and corpus-level statistics, audits of scenario-reservoir and task-description diversity, and the full training configuration. We then provide detailed evaluation results and the exact evaluation protocols behind the main-paper scores, including per-task and pass@k analyses and the terminal ablation. Finally, we include faithful English renderings of the four stage prompts and the teacher-rollout runtime prompt that drive the pipeline.

A SCENARIO-CONDITIONED COMPILATION PROCEDURE

Algorithm 1 details how NexForge composes a task directive δ_j from the task demand profile $\Phi(I)$ and a single scenario g_j drawn from the reservoir $\mathcal{G}(I)$. The composer processes the ordered task-form dimensions $\{t, d, \sigma, e\}$ (task type, deliverable, source strategy, runtime) one at a time. For each dimension k, a compatibility filter F_θ inspects the requirement, the scenario, and the fields already selected, and returns the subset C of profile options that remain coherent in this context; the realized field $f_{j,k}$ is then sampled from the profile weights restricted to C. Because earlier decisions constrain later ones, the directive stays internally consistent (for example, a chosen runtime does not admit an incompatible deliverable), while the profile weights still guide the corpus-level distribution. Language ℓ_j and difficulty h_j are drawn last under the batch-level constraints rather than from scenario compatibility, and the retained candidate sets A together with the per-step rationale R are recorded so that downstream stages instantiate the sampled work rather than re-inferring it from whatever materials happen to be available.

Algorithm 1 Scenario-conditioned task compilation

Input: requirement I, profile $\Phi(I)$, scenario g_j
Output: task directive δ_j

$S, A, R \leftarrow \emptyset, \emptyset, \emptyset$
for $k \in \{t, d, \sigma, e\}$ **do**
  $C \leftarrow F_\theta(I, g_j, k, \Phi_k, S)$
  $A[k] \leftarrow C$
  $f_{j,k} \sim \mathrm{Normalize}(\Phi_k|_C)$
  $S \leftarrow S \cup \{f_{j,k}\}$
  Append the compatibility rationale to R
end for
Sample ℓ_j and h_j under the batch constraints
return $\delta_j = \langle g_j, S, A, \ell_j, h_j, R \rangle$

B END-TO-END CASE STUDY

To make the pipeline concrete, we trace a single terminal task from the sampled directive to the runnable package (Figure 5). The example illustrates the central claim of the paper: the work contract is fixed by the compiled directive *before* any material is gathered, so the substrate is selected to serve the intended task rather than the task being inferred from whatever substrate is convenient.

Scenario and directive. The scenario describes a systems-programming context around *pMARS*, the portable Memory Array Redcode Simulator used to run Core War tournaments. Following Algorithm 1, the composer conditions the profile on this scenario and sequentially selects compatible task-form fields: a *build, packaging, and toolchain integration* task type, a *command-line executable plus reproducible build script* deliverable, a *GitHub repository adaptation* source strategy, and a *shell/CLI* runtime. Crucially, the same scenario could have supported a build repair, a benchmarking harness, a data-extraction task, or a cross-compilation deliverable; the sampled directive commits to one primary goal—producing an offline, statically linked cross-compilation of pMARS—so later stages do not silently drift toward the most convenient interpretation of the repository.

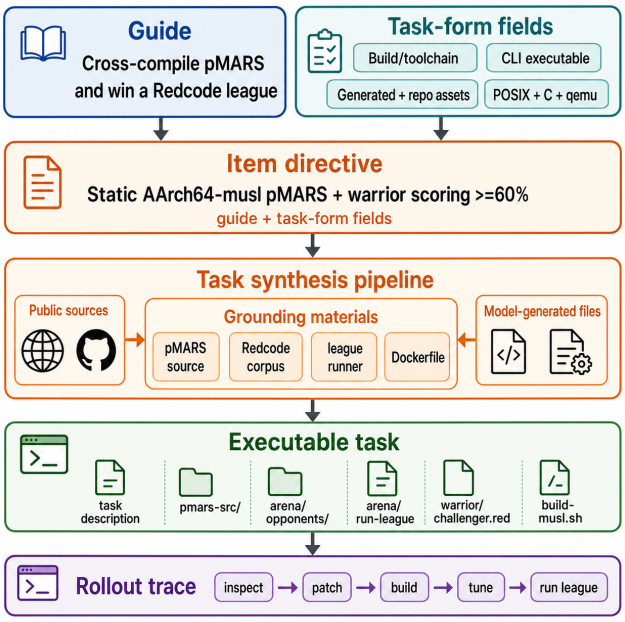

Figure 5: End-to-end Terminal-3.6K example: pMARS cross-compilation scenario composed into an executable workspace.

Material mining. Conditioned on the directive, the mining stage performs targeted web research to recover the real workflow and candidate materials: the upstream pMARS source tree, a set of opponent Redcode warriors, and a league runner that scores warriors head-to-head. Mining records each asset with a concrete source, format, and intended role, and distinguishes solver-facing inputs from materials reserved for the runtime environment, so that the downstream package contains genuine artifacts rather than synthetic stand-ins.

Blueprint ideation. The ideation stage consolidates the mined evidence into a single-schema blueprint. It fixes the objective (reconstruct a musl-based static pMARS binary offline and validate it by running a warrior through the league), the input materials, the expected deliverables and their success boundary, the workspace layout, and the software dependencies. Because the solver environment is a CPU-only unprivileged container, ideation resolves any environment-sensitive choices at design time—selecting an offline source archive, pinning the toolchain, and avoiding privileged or GPU-dependent steps—so the resulting task is guaranteed to be executable under the standard runtime.

Workspace generation. The generation stage instantiates the blueprint into a solver-facing package: it lands the prepared pMARS sources and opponent programs, adds the league runner and any generated fixtures, writes a concise instruction that states the goal, starting materials, and delivery contract without leaking internal checks, and emits a CPU-only, unprivileged Dockerfile that installs the declared dependencies. Automated checks then verify material completeness, source-strategy consistency, runtime restrictions, and the absence of leaked solutions before the package is admitted for teacher rollouts.

Why the directive matters. This example shows the difference between requirement-driven and substrate-bound synthesis in miniature. A substrate-bound pipeline that began from the pMARS repository would most naturally emit an implementation or test-repair task, because that is what the repository most directly supports. By fixing a cross-compilation build contract before mining, NexForge instead produces a long-horizon toolchain task whose difficulty and structure are determined by the intended work, and the resulting trajectory is correspondingly deep (many build, inspection, and validation steps rather than a single edit-and-test loop).

Table 7: Trajectory signal scale per task group.

Task group	Packages	Distinct	Attempts	Records	Tok. B	Avg. tok/att.	Calls/att.
Terminal-3.6K	3,600	3,501	8,521	17,635	3.350	129.3K	123.3
Office-2K	2,000	2,000	5,940	6,388	0.693	106.7K	53.6
Terminal-2K	2,000	1,947	4,656	9,214	1.760	128.7K	131.7
w/o profile	2,000	1,916	4,574	6,555	1.019	154.0K	132.8
w/o scenario	2,000	1,894	4,541	11,039	1.826	123.7K	116.4

Table 8: Token and operation mix in converted trajectories.

Task group	Steps	Tool	Reason B	Tool-call B	Read	Write	Build/test
Terminal-3.6K	82.2	16.0	1.171	1.574	26.4	51.5	4.7
Office-2K	55.6	39.7	0.072	0.313	42.8	34.1	1.0
Terminal-2K	81.6	16.2	0.526	0.907	24.7	52.2	4.7
w/o profile	76.3	22.1	0.552	0.837	28.5	45.1	3.2
w/o scenario	68.3	30.0	0.257	0.435	34.8	48.3	3.8

C Agent Post-Training Data: Synthesis Configuration and Corpus Statistics

This section documents the configuration that drives synthesis and the corpus-level statistics of the resulting data. We first report the reviewed task-form profile and keyword pools that define the candidate space, then the scale and token/operation composition of the collected trajectories.

C.1 Reviewed Profile and Keyword Pools

The reviewed task-form profile defines the candidate space that later sampling draws from. For Terminal-2K it contains 24 primary task types, 20 runtime environments, 8 deliverable types, and 4 source strategies, supported by a domain keyword pool of 3,678 candidates; the office run contains 15 task types, 12 runtimes, 23 deliverables, and 4 source strategies with a 773-keyword pool. The terminal profile spans more task types and runtimes, reflecting the broader operational surface of command-line work, whereas the office profile enumerates more deliverable types, matching its emphasis on documents, analyses, and recommendations. All candidate lists are pruned by the LLM review step for mutual exclusivity and clear semantics before sampling; the realized coverage and concentration of each dimension are reported in Table 17, and the per-item scene-conditioned narrowing of these pools in Table 16.

C.2 Trajectory Scale and Token Mix

Table 7 separates original teacher rollouts from the framework-internal cleaned records for each task group and reports per-rollout token and tool-call averages, while Table 8 shows how tokens and operations distribute across message roles and tool classes. The terminal corpora are markedly more tool- and build-intensive, whereas the office corpus yields shorter, more read- and document-oriented trajectories.

C.3 Capability Requirement Specifications

The two corpora are generated from independently written capability requirements. Each specification is a high-level natural-language description of the target capability that serves as the pipeline input I (Section 4.1). Below are faithful English renderings of the two specifications.

Terminal capability requirement.

```
Design terminal-agent tasks that cover the distribution of multi-domain technical tasks in
real command-line environments. The task scope spans software engineering, system
administration, security engineering, scientific computing, data science, data processing,
data querying, machine learning, model training, mathematical problem solving, optimization,
code comprehension, file operations, game or strategy solving, multimedia processing,
personal-assistant automation, and other primary directions, with room for further expansion
based on real terminal task forms. Each item should require the agent to carry out real
technical work through shell commands in a controlled terminal environment and produce
deliverables with a well-defined final state, external behavior, or file-level result. Task
outputs should be reproducible, executable, and machine-verifiable---for example,
source-code or script changes, system configurations or environment states, SQL or
structured-data results, model or numerical outputs, optimization plans, processed files or
media, game/strategy solutions, automation execution results, build artifacts, or test
outcomes. Generated items should vary substantially across task category, runtime
environment, input content, required tool chain, operation path, boundary conditions, and
success criteria.
```

Office capability requirement.

```
Design high-value professional work tasks situated in real organizations. The task scope
covers digital knowledge work in high-output industries and large organizations, including
real estate and leasing, government and public services, manufacturing, professional
scientific and technical services, healthcare, finance and insurance, retail and wholesale,
information and media, logistics, and operations support. Task roles should cover operations
management, asset management, public services, compliance oversight, engineering and
manufacturing support, procurement and supply chain, information systems, legal, accounting,
project management, healthcare administration, customer service, financial analysis, sales,
news editing, audio/video production, and other functions. Each item should require the
agent to understand a concrete work scenario, leverage the given information, structured
data, communication records, business rules, system exports, public web information, or
lightweight tool environments to complete analysis, judgment, computation, planning,
coordination, communication, comparison, content production, media processing, or decision
support. Task outputs should be concrete deliverables directly usable in a work
setting---for example, structured tables, business conclusions, replies to clients or
colleagues, process schedules, decision memos, priority judgments, resource-allocation
recommendations, calibration notes, updated documents, presentation materials, charts,
content drafts, edit lists, subtitle files, media outputs, or audio/video processing
results. Generated items should vary substantially across industry, role, objective,
information carrier, constraint conditions, collaboration target, time requirements,
judgment criteria, and delivery format.
```

D AGENT POST-TRAINING: FINE-TUNING CONFIGURATION

This section records the teacher rollout and supervised fine-tuning configuration behind every trained model, so that each reported run can be reproduced from its logged provenance.

Across the experiments, teacher rollouts use DeepSeek V4 Pro. The records retain all teacher rollouts that survive trace cleaning, which drops only malformed and degenerate trajectories. The same inclusion rule is used for the terminal ablation and Qwen3-32B comparison. All SFT runs use 16 nodes with 8 NVIDIA H100 GPUs per node, for 128 H100 GPUs in total.

All experimental SFT runs use full-parameter SFT for five epochs with maximum sequence length 262,144, packed sequences, bfloat16, global batch size 64, micro batch size 1, learning rate 10^{-5}, minimum learning rate 10^{-7}, cosine decay, 5% warmup, and the `ignore_empty_think` loss scale. The Qwen3.5-35B-A3B runs use the `qwen3_5` template and MoE expert parallelism; the Qwen3-32B comparison uses the `qwen3` template and pipeline parallelism. For the terminal ablations, final training-log summaries give approximately 1.61B packed train tokens for Terminal-2K, 0.97B for Terminal-2K w/o profile, and 1.72B for Terminal-2K w/o scenario. These values are computed from the logged mean packed sequence length multiplied by the logged packed train-set size. They differ slightly from the converted-record tokenizer audit in Table 7, which sums visible content, reasoning content, and serialized tool calls before training-framework packing and without chat-template overhead.

Table 9: SFT run provenance from training logs.

Run	Student	Task set	Train	Val	TP/EP/CP/PP	Time	Ckpt
Terminal-2K	Qwen3.5-35B-A3B	Terminal-2K both-use	9,198	16	2/8/8/1	12.2h	521
w/o profile	Qwen3.5-35B-A3B	Terminal-2K scenario-only	6,539	16	2/8/8/1	7.4h	313
w/o scenario	Qwen3.5-35B-A3B	Terminal-2K profile-only	11,023	16	2/8/8/1	12.7h	541
Qwen3-32B comp.	Qwen3-32B	Terminal-3.6K	17,619	16	2/1/8/4	6.5d	997

E Detailed Evaluation Results

Table 10 reports the controlled Terminal-Bench 2.0 pass@1 statistics behind the scalar values in the main paper. All rows use the same in-house NexAU scaffold, task-suite revision, tools, inference settings, and denominator of 89 tasks per run. Unfinished trials, still-running trials, agent timeouts, verifier timeouts, nonzero exits, and verifier errors are counted as failures.

Table 11 reports available GDPval Elo confidence intervals behind the scalar values in the main paper. The scoring follows the GDPval-AA v1 methodology of Artificial Analysis: the outputs of GPT-5.1 are fixed as the 1000-point anchor; every pair of models is then compared on all 220 GDPval tasks with an LLM judge that returns a win, loss, or tie for each task; and Elo scores are fit from the full set of pairwise match outcomes. All rows are fit in one in-house NexAU-based joint comparison pool and should not be interpreted as external GDPval leaderboard scores.

Table 12 records the external Terminal-Bench reference values used for context in the main paper. These reference values come from the public leaderboard and published terminal/agent-data studies (Terminal-Bench Team, 2026; Fan et al., 2026; Pi et al., 2026; Peng et al., 2026; Wu et al., 2026; Cheng et al., 2026; Raoof et al., 2026).

E.1 Per-Task Outcomes and Pass@k

Beyond mean accuracy, we examine how the four independent runs distribute over individual tasks. Table 13 reports, for the base model and the Terminal-3.6K model, how many of the 89 tasks are solved in $0, 1, \ldots, 4$ of the four runs. Training does not merely convert a handful of borderline tasks: the number of tasks that never pass drops from 54 to 29, while the number solved in all four runs rises from 7 to 29. Terminal-3.6K therefore adds robustly solvable tasks rather than occasional lucky passes.

Figure 6 extends this to task-level pass@k, estimating whether at least one of k sampled attempts solves each task. NexForge improves the entire pass@k curve for both base models: Qwen3.5-35B-A3B rises from 52.0% at pass@1 to approximately 68% at pass@4 and stays roughly 30 points above its base throughout, while Qwen3-32B rises from 32.3% to approximately 55% (its base reaches only about 14% at four attempts). The NexForge-trained Qwen3-32B also stays above the published terminal-specialized models across the evaluated sampling budgets. These trends indicate that Terminal-3.6K expands the set of tasks for which a valid solution can be discovered through repeated sampling, not only single-attempt accuracy; the specialized-model curves are contextual references evaluated with a different scaffold.

Table 10: Controlled Terminal-Bench 2.0 pass@1 accuracy statistics.

Model	Runs	Pass	Mean	95% CI	Range
Qwen3.5-35B-A3B Base	4	80/356	22.5	0.9	[21.3, 23.6]
Qwen3.5-35B-A3B + Terminal-3.6K	4	185/356	52.0	3.6	[48.3, 56.2]
Terminal-2K	4	156/356	43.8	4.3	[39.3, 49.4]
w/o profile	4	143/356	40.2	5.0	[33.7, 46.1]
w/o scenario	4	152/356	42.7	4.2	[37.1, 46.1]
Qwen3-32B Base	4	20/356	5.6	0.9	[4.4, 7.0]
Qwen3-32B + Terminal-3.6K	4	115/356	32.3	2.4	[30.3, 36.0]

Table 11: GDPval Elo estimates from our internal joint comparison pool.

Model	Elo	95% CI
GPT-5.4	1667	[1639, 1697]
Claude Sonnet 4.6	1633	[1605, 1663]
GLM-5	1408	[1385, 1433]
Qwen3.5-35B-A3B + Office-22K	1384	[1358, 1410]
Qwen3.5-35B-A3B + Office-2K	1338	[1312, 1364]
Gemini 3.1 Pro Preview	1316	[1291, 1343]
MiniMax M2.5	1202	[1175, 1231]
Gemini 3 Flash Preview	1191	[1164, 1220]
GPT-5.1	1000	[1000, 1000]
Qwen3.5-35B-A3B Base	813	[787, 840]

Table 12: Reference Terminal-Bench 2.0 scores used in the main paper.

Group	Model	Params	TB 2.0
External	Gemini 3.1 Pro	–	80.2 ± 2.6
	DeepSeek-V4-Pro	1.6T	67.9
	Claude Opus 4.6	–	58.0 ± 2.9
	MiniMax M2.7	230B	57.0
	GPT-5.2	–	54.0 ± 2.9
	Claude Sonnet 4.6	–	53.4 ± 2.8
	GLM 5	744B	52.4 ± 2.6
	Claude Opus 4.5	–	51.9 ± 2.9
	Gemini 3 Flash	–	51.7 ± 3.1
	GPT-5.1	–	47.6 ± 2.8
	Kimi K2.5	1T	43.2 ± 2.9
	MiniMax M2.5	230B	42.2 ± 2.6
	DeepSeek-V3.2	671B	39.6 ± 2.8
	Qwen 3 Coder	480B	23.9 ± 2.8
Qwen3 Based	LiteCoder-Terminal-32B-SFT	32B	18.5 ± 3.4
	TerminalTraj-32B	32B	22.0 ± 5.8
	OpenThinkerAgent-32B	32B	26.2 ± 1.6
	Nemotron-Terminal-32B	32B	27.4 ± 2.4
	Qwen3-32B + SkillSynth	32B	29.6 ± 1.6
	CLI-Universe-32B (DeepSeek)	32B	31.2
	Terminal-World-32B	32B	31.5

Table 13: Per-task pass-count distribution over four runs.

Model	0/4	1/4	2/4	3/4	4/4
Base	54	12	8	8	7
Qwen3.5 + Terminal-3.6K	29	8	8	15	29

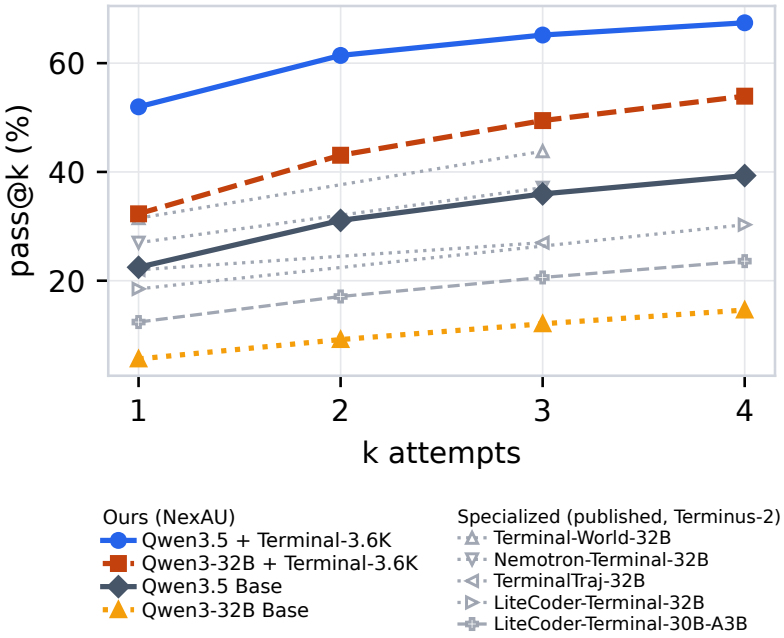

Figure 6: Task-level pass@k on Terminal-Bench 2.0.

F Terminal Ablation Details

The terminal ablation keeps the student, rollout budget, SFT recipe, and Terminal-Bench 2.0 target fixed while changing the synthesis controls. The Terminal-2K w/o profile variant keeps the diverse scenario reservoir but removes selected task form fields from the directive interface. In Terminal-2K w/o profile, downstream stages receive the scenario but not selected task form dimensions. The Terminal-2K w/o scenario variant keeps sampled task form fields and replaces the concrete scenario with a neutral scenario seed. In the implementation, Terminal-2K w/o scenario samples directly from the reviewed profile without scenario-conditioned filtering and uses the same neutral scenario text for all items. The Terminal-2K control is the original scenario-conditioned directive construction.

Table 14: Primary task type distribution for the terminal ablation (%).

Variant	Labeled	Software	Sys. admin	Data proc.
Terminal-2K	2000	54.3	11.8	8.6
w/o profile	1997	78.5	0.5	2.7
w/o scenario	2000	24.9	9.1	5.5

Table 14 shows the strongest distributional effect. w/o profile collapses toward software engineering tasks and almost eliminates some terminal task families, consistent with the model's default mapping from terminal scenarios to tasks. w/o scenario is much flatter because task form fields are sampled from the reviewed profile without a concrete scenario filter. At the reconstructed construction-dimension level, w/o scenario has normalized required-task type entropy 0.875 and effective support 16.15, compared with 0.565/6.02 for w/o profile and 0.571/6.13 for Terminal-2K. These effective-support values count distinct dimension candidates available before generation; they are not the realized task-type distribution in Table 14, which is far more concentrated for w/o profile (post-hoc labeling collapses to an effective count near one) because the generator defaults to software-engineering tasks when the directive removes selected task types. w/o profile and Terminal-2K therefore share almost the same construction-level support, yet differ sharply in realized diversity and downstream accuracy. This explains why w/o scenario remains competitive at 2,000 tasks, while the scenario reservoir is still important for large-scale synthesis where repeated neutral scenarios would make concrete task contexts generic.

G TASK-PACKAGE AUDITS FOR AGENT TRAINING DATA QUALITY

We audit the synthesized packages along three axes: static integrity, scenario-conditioned filtering, and the realized distribution over task-form dimensions. Table 15 checks that each package contains the expected artifacts and is grounded in real sources; Table 16 reports how aggressively the scenario-conditioned filter narrows the candidate pool before sampling; and Table 17 with Figure 7 show the realized coverage and concentration of each dimension. Terminal-1.6K is an intermediate corpus produced during the filtering-table audit; it uses a narrower profile (18 rather than 24 primary task types) but otherwise follows the same synthesis pipeline.

Table 15: Static synthesis audit.

Task group	Tasks	Scenarios	Core files	Real src.
Terminal-3.6K	3,600	3,600	3,600	2,292
Office-2K	2,000	2,000	2,000	1,569

G.1 TASK QUALITY AUDIT

To confirm that the synthesized packages are coherent and feasible, we manually inspect stratified task packages sampled from each corpus and ablation variant on a rubric covering five dimensions: description clarity (D1), material completeness (D2), workspace executability (D3), difficulty calibration (D4), and absence of internal-artifact leakage (D5). Overall quality is high: the majority of inspected packages score at least 4 on all five dimensions, and none receives any score below 3. Difficulty calibration is the most frequent deduction—a few tasks pack more work than the stated hour budget—while no internal-artifact leakage is observed. The Profile-only variant scores lowest because its from-scratch tasks contain fewer grounded materials, yet it still outperforms Scenario-only on Terminal-Bench 2.0, confirming that the ablation gap is driven by task-distribution shape rather than per-task package quality.

Table 16: Scenario-conditioned compatibility filtering.

Task group	Type pool→kept	File pool→kept
Terminal-2K	24→4.42	167→27.37
Terminal-1.6K	18→4.04	175→37.78
Office-2K	15→6.63	105→21.61

Table 17: Realized distribution across task-form and material dimensions.

Dimension	Control role	Terminal-2K	Office-2K
Task type	Primary work form	24/24; top 27.9%	15/15; top 17.3%
Deliverable	Output contract	8/8; top 19.1%	23/23; top 11.9%
Source strategy	Grounding choice	4; GitHub 45.2%	4; web 69.2%
Runtime	Execution medium	20/20; top 22.4%	12/12; top 15.7%
Info. carrier	Evidence form	8/8; top 17.1%	19/19; top 14.3%
Material file type	File variety	167/167; top 3.6%	97/105; top 4.6%

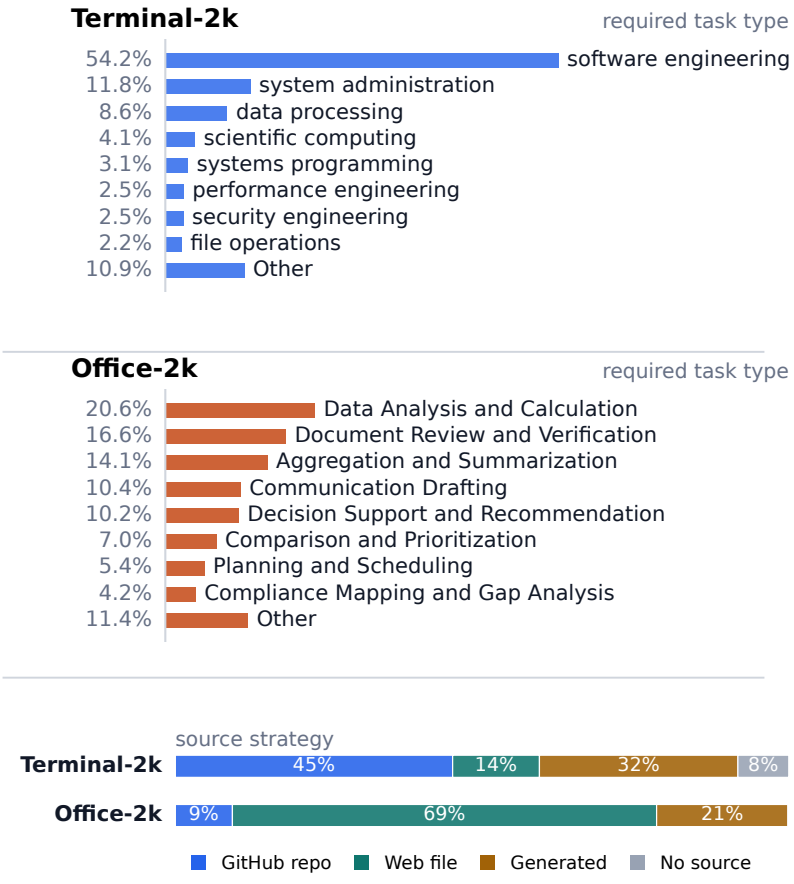

Figure 7: Task type and source-strategy distributions for Terminal-2K and Office-2K.

H SCENARIO RESERVOIR AUDIT

We audit scenarios with the same embedding model used by the online novelty filter. All statistics in Table 18 are computed over the accepted scenario texts before scenario mining and task form composition. Pairwise cosine is the mean cosine over all unordered scenario pairs in a branch. Nearest-neighbor (NN) median measures local redundancy within the branch. The target silhouette uses cosine distance and the two target groups, terminal and office; higher values indicate stronger separation by target. Cross-target NN is the share of scenarios whose nearest neighbor inside the same branch belongs to the other target group.

Table 18: Branch-level embedding audit for accepted scenarios.

Branch	Scenarios	Pair	NN	Silh.	Cross NN
Self-instruct	1,868	0.278	0.678	0.141	0.16%
Knowledge graph	1,868	0.314	0.655	0.071	3.64%
Keyword self-inst.	932	0.295	0.625	0.098	0.86%
Keyword research	932	0.310	0.622	0.113	0.32%

The table supports three observations used in the main text. First, self-instruct is the most target-conditioned branch: it has the highest target silhouette and the lowest cross-target nearest-neighbor rate. Second, keyword research and keyword self-instruct are close in local compactness, but keyword research separates the two target intents slightly more strongly. Third, knowledge graph expansion has the weakest target separation, which is consistent with the branch exploring reusable entities, workflows, and contexts that can appear in both terminal and office settings.

I TASK-DESCRIPTION EMBEDDING

We embed the final solver-facing task descriptions of the matched Terminal-2K and Office-2K sets (4,000 descriptions, 2,000 per target, 50/50 English–Chinese) with `text-embedding-3-small` and analyze their geometry. The two targets separate cleanly: a nearest-centroid classifier reaches 0.980 accuracy (0.980 for the four target–language groups), and 97.8% of nearest neighbors share the target (97.2% share target and language). Separation is not template collapse, however: within-target random pairs have mean cosine 0.571 versus 0.503 across targets, a modest gap, and nearest-neighbor cosine has median 0.797 with maximum 0.910, so descriptions are semantically concentrated by target and language without duplicating one another. Figure 8 shows the PCA projection, with four visible target–language clusters.

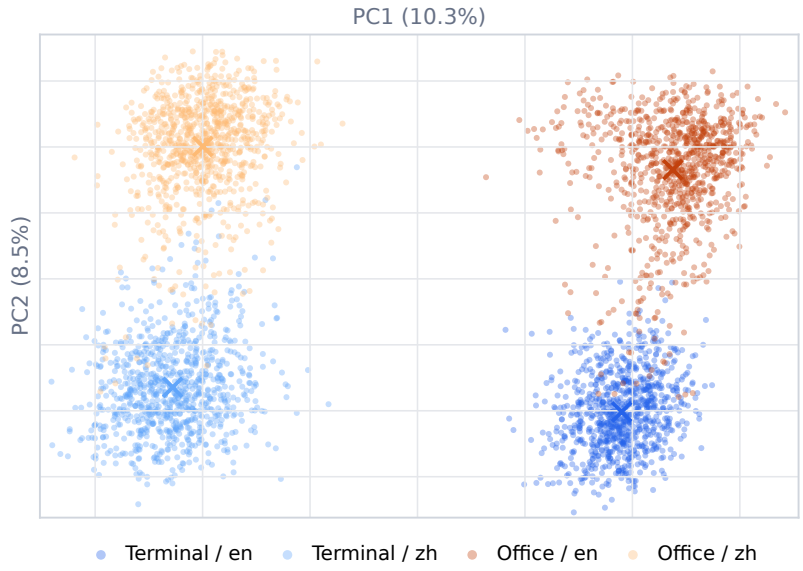

Figure 8: PCA projection of task description embeddings for Terminal-2K and Office-2K.

J SYNTHESIS PROMPT TEMPLATES

The synthesis pipeline is driven by four stage prompts: *diverse* (task profile), *mine* (scenario research), *ideate* (task blueprint), and *gen* (task materialization); teacher rollouts use the NexAU runtime prompt. The stage prompts are written in Chinese in the released code; below are faithful English renderings. Jinja variables are shown as {{name}}, and the ablation-mode conditionals (`guide_only`, `intent_only`) are collapsed to the default both-use path; where a branch changes wording we note it inline. The verbatim original templates are included in the released prompt archive under `analysis/prompts/`.

J.1 PLANNING: TASK PROFILE (DIVERSE)

This prompt turns a free-form task-set specification into the reviewed task-form profile $\Phi(I)$.

```
Carry out task-planning-oriented research based on the user input and organize it into a
structured task profile.

User input:
{{ root_query }}

First research and read relevant material around the user input, covering multiple reliable
sources. Prefer broad taxonomies, task collections, label systems, tool/ecosystem catalogs,
standards, or domain surveys to build the overall frame, then fill long-tail directions with
finer material. Scope the research to the domains, workflows, tools, objects, data forms,
common deliverables, and runtime environments the user input naturally points to.

The goal is a comprehensive, cleanly structured profile, not a summary and not concrete
items. The lists are the reusable candidate space for later generation: cover every
candidate with real discriminative value while avoiding semantic overlap within a list. In
particular, `task_types` is later sampled by weight for the primary task type, so it must
stay same-level, mutually exclusive, and clearly bounded; cross-cutting capabilities,
material forms, runtime environments, and delivery forms go in their own fields, and
`task_types` keeps only the primary work type.

Output requirements:
- task_types: mutually exclusive primary work categories, i.e. the single kind of work the
solver mainly does in one item. Not a scene, exception, material type, runtime, deliverable,
verification, or difficulty source. Each item has name, description, when_applicable, and a
positive weight (relative sampling weight in the real distribution, need not be normalized;
spread head/mid/tail weights apart). Merge items that usually co-occur.
```

```
- runtime_environments: runnable, verifiable, interactive environments the task package or
Dockerfile must prepare, not a business location. Each item has name, description,
tools_or_services, when_applicable (e.g. shell/CLI, Python data analysis, Node/Java/Go/Rust
build-test, local web service, database service, browser automation, PDF/OCR, media
processing, scientific computing, container tooling, offline document review).
- deliverables: abstract deliverable types the solver should ultimately produce. Each has
name, description, typical_success_criteria (an internal profile field summarizing the final
quality boundary, NOT a solver-facing acceptance/verification procedure).
- information_carriers: carrier forms of the input information (not deliverables, not
runtimes). Each has name, description, when_needed (e.g. code, logs, config, DB exports,
issues, email, screenshots, API docs, contracts, policies, media assets).
- material_file_types: a flat list of lowercase suffixes / common format names only (e.g.
pdf, csv, xlsx, json, yaml, log, ipynb, pcap). No grouping, no descriptions.
- When the input points to command line / terminal / local repo / executable terminal work,
task_types and deliverables should favor runnable, editable, testable, reproducible,
machine-checkable work and products.
- Remove near-synonymous items in each list; keep task_types more restrained and mutually
exclusive than the others. Keep abstract categories abstract and file suffixes / tool
environments concrete.

The final action must call GenerateIntentProfile with output_path={{ output_path }},
providing an intent_profile object containing all of: task_types, runtime_environments,
deliverables, information_carriers, material_file_types.
```

J.2 MINING: SCENARIO RESEARCH (MINE)

Conditioned on the item directive (a scenario composed with compatible task-form fields), this prompt gathers realistic workflows, terminology, resources, and candidate materials.

```
Do focused research around the given scene, distilling the task form, runtime environment,
required knowledge, usable information carriers, and delivery boundary that fit this item,
and produce a research record for blueprint design.

Scene input (may contain `item_directive`):
{{ seed_scene_json }}

Research focus:
- The work the solver must actually do in the scene, its key objects and constraints.
- How `item_directive.selected` (task type, runtime, deliverable, source type) shapes the
task; `required_task_type` is a high-level direction, so research should clarify what it can
concretely become within the original user intent, not shrink it to a fixed parameter or
fixed procedure.
- How `item_directive.applicable_options` (information carriers, material file types) match
the scene.
- The domain, tool, and workflow knowledge and the judgment needed to finish the task.
- Real resources, data, code, logs, config, docs, pages, tests, or interfaces that help. If
difficulty_hours is present, treat it as a lower bound on expert time and look for real
complexity, edge cases, and delivery boundaries that support it.

Proceed by: (1) understanding the task scene (core need, key objects, constraints, target
deliverable); (2) progressive retrieval, splitting more specific queries from prior
findings, covering domains, tools, environments, interfaces, data forms, deliverables, and
quality boundaries; (3) curating usable resources/assets, each with a concrete name, link,
format/type, purpose, and relation to the task, judging whether it is substantial enough to
support reasoning, operation, construction, comparison, analysis, decision, delivery, or
verification.

Produce a research record with four parts: (I) task interpretation; (II) knowledge and
information required; (III) execution key points and pitfalls; (IV) usable resources and
asset suggestions, marking which are suited to be solver-facing inputs and which to hidden
evaluation, environment dependencies, or background. Output natural-language text with
accessible links.
```

J.3 IDEATION: TASK BLUEPRINT (IDEATE)

This prompt turns the directive and mined materials into a single-schema blueprint (`ideation.json`), preparing real sources via `DownloadSourceFile` / `PrepareGitHubRepo`. A variant (*history-to-blueprint*) emits the same schema from the recorded research history as an XML-like document.

```
You are a professional task-blueprint designer. Based on the input query, scene research,
task profile, and item directive, design a high-quality, high-difficulty, executable task
blueprint.
```

```
Inputs: {{ scene_query }} ; scene report {{ scene_report }} ; task profile {{
intent_profile_json }} ; item directive {{ item_directive_json }}. Target language: {{
target_language_label }}. Difficulty: an expert should need at least {{ difficulty_hours }}
hours (a lower bound, not a target to shrink to).

Requirements:
- Understand what the query wants the solver to do and the real-world scene behind it. Do
not merely rewrite the guide into a task description; extract scene, role, objects, boundary
conditions, final target state, and material forms, then expand into deep real work within
the user-intent boundary.
- Choose the task type, runtime, deliverable, source type, and asset form fitting the scene.
Runtime means the internal execution environment the package/Dockerfile prepares, not a
business location and not a setting to write into the prompt. If
`item_directive.selected.required_*` fields exist they set the high-level
type/deliverable/runtime/source and must be honored; optional_* are candidates, not an
output checklist; default to a single primary task type.
- The final task must have one clear primary goal and one primary task type; other types or
deliverables only appear as supporting dimensions. Do not stack several medium tasks or
independent products to fake difficulty.
- The solver environment is a CPU-only, ordinary unprivileged container. Do not plan tasks
needing GPU/CUDA/NVIDIA/TPU or host-level daemons, Docker Engine/socket, docker compose,
KVM, PID 1 systemd, kernel modules, loop mount, or privileged containers; container/OCI
directions become file-level checks/fixes on Dockerfile/Containerfile, manifest, tar, or OCI
layout. These execution facts are an internal design constraint and must NOT be written into
task_description unless the task object itself is a container/OCI artifact.
- Use WebSearch / WebFetch to add background and to look for publicly accessible,
appropriately sized real files or GitHub repositories matching the direction. Record the
final source choice in `source_selection` (prepared vs to-be-generated assets), referencing
canonical tool outputs (remote_rel_path). Source priority: github_repo_adaptation >
internet_file_download > generated_files > no_prerequisite_files; use generated files mainly
to fill context, fixtures, edge samples, and consistency material, not to replace real data
when available.
- The initial solver-facing package must contain only real files and directories (no
symlinks); do not materialize /proc, /sys, /dev, snapshots, infinite link trees, full .git
history, or large vendor/build output.
- task_description is a concise, direct, solver-facing statement (goal, background, starting
materials or clean-environment contract, constraints, deliverable, paths, final state,
external behavior contract). Its first sentence starts from the most distinctive
information. Do not prescribe the solver's solution order, planning, debugging route, or
verification steps; do not expose hidden tests, reference answers, expected value ranges,
scoring details, injected bugs, or a known-root-cause list. Keep local paths relative.
- Do not mention internal artifacts (blueprint, ideation, item directive, intent profile,
source_selection, prepared/generated assets, manifest, research process, tool names, hidden
checks, the generation stage, or "intentionally injected/constructed/simulated"
meta-information) in task_description.
- Derive difficulty on the spot (hard_topics, difficulty_budget.justification) from the
chosen fields, materials, deliverable, sources, and difficulty_hours; do not use a preset
difficulty list.

After research, source preparation, field design, path/environment checks, and key
value/contract self-checks, call GenerateIdeationBlueprint with output_path={{ output_path
}} (a heavy final tool that regenerates the whole blueprint from the conversation; make
small fixes by editing ideation.json directly). Blueprint schema requirements: {{
schema_requirements }}.
```

J.4 GENERATION: TASK MATERIALIZATION (GEN)

This prompt writes the solver-facing package (task description, materials, Dockerfile, and internal audit files) from the blueprint, using prepared sources.

```
Synthesize a real task from the task blueprint. You only synthesize; an independent solver
will complete it. The blueprint's task_description is a draft to align paths, add detail,
and ground materials on. Prepared external files from ideate are used directly.

Blueprint: {{ ideation_json }}. Prepared sources dir {{ prepared_sources_dir }} and manifest
{{ prepared_sources_manifest_path }} (read it before handling
source_selection.prepared_assets). Target language: {{ target_language_label }}. Generate
the Dockerfile at {{ dockerfile_output_path }} from environment_plan.

source_selection handling: prepared_assets are already prepared by ideate; locate them via
the manifest (path = remote_rel_path) and copy/trim/extend/extract them into the final
package. Source priority github_repo_adaptation > internet_file_download > generated_files >
no_prerequisite_files; real sources should be the core material, generated assets only fill
context/fixtures/edge samples. Record used assets in generation_summary.json (name,
source_type, source_path, copied_to/adapted_to, usage).
```

```
Solver-facing filesystem constraints: only real ordinary files and directories; no symlink
of any kind; do not copy upstream symlinks, /proc, /sys, /dev, virtual filesystems,
snapshots, infinite link trees, full .git history, dependency/build caches, or large
vendor/build output; trim to the minimal real subset the task needs.

Internal execution constraints: CPU-only (no GPU/CUDA/NVIDIA/TPU in tasks, Dockerfile,
scripts, tests, or data pipelines; shrink ML/image/video/scientific work to small models,
small data, CPU inference/training, or pure-algorithm substitutes). Ordinary unprivileged
container (no host daemon, Docker Engine/socket, docker compose, KVM, PID 1 systemd, kernel
modules, loop mount, or privileged container; container/OCI work is file-level only). Do not
write these execution facts into the task description unless the task object is itself a
container/OCI artifact.

Landing principles: task_type is a high-level direction, not a fixed script; keep the
blueprint intent, source choice, and delivery boundary; deepen one primary goal rather than
stacking medium subtasks. The final task_description reads like a concise issue or benchmark
instruction (starting materials or clean-environment contract, goal, constraints,
deliverable, key interfaces/output schema, final state, external behavior contract); no
numbered subtask checklist, no solver solution route, no hidden tests / full verification
commands / reference answers / expected value ranges / scoring internals, no known-defect or
preset-root-cause list. For fix/recovery/troubleshooting tasks the starting materials may
contain failing tests, logs, half-done implementations, or inconsistent data, but only
visible symptoms and contracts go into the description.

Deliverables: (1) the task description file at {{ task_description_path }} (relative paths
only); (2) reference material files (reuse prepared real sources first, then create real
supplementary files that are readable, parseable, and actually used in solving); (3) the
Dockerfile at {{ dockerfile_output_path }} (base_image matches the blueprint, installs
declared dependencies, CPU-only, unprivileged-compatible, does not COPY the task files in).
generation_summary.json and generation_validation.json are internal audit files at the
generation root, never inside the solver package and never referenced by the task
description.

Workflow: understand the blueprint; read the prepared-sources manifest; land prepared
assets, then generated assets and other referenced files; write the Dockerfile; align
task_description paths; self-check goal/material/interface consistency and output directory;
write generation_validation.json. Finish with a checklist confirming files exist,
hard_topics are reflected, materials are real, prepared assets are used and logged, paths
are relative and consistent, no leaked internal/execution facts, no symlinks, CPU-only,
unprivileged-compatible, and difficulty >= {{ difficulty_hours }} hours.
```

J.5 TEACHER ROLLOUT RUNTIME (DISTILLATION)

Teacher rollouts that produce the distillation trajectories run under the NexAU runtime prompt (already in English). The base system prompt and the context-compaction prompt are shown below; tool descriptions are provided per run.

```
You are an AI agent named '{{ agent_name }}' built on the NexAU framework.

You have access to the following tools:
{% for tool in tools %}- {{ tool.name }}: {{ tool.description }}
{% endfor %}
You can delegate tasks to the following sub-agents:
{% for sub_agent in sub_agents %}- {{ sub_agent.name }}: {{ sub_agent.description }}
{% endfor %}

Your goal is to help users accomplish their tasks efficiently by:
1. Understanding the user's request
2. Determining if you can handle it with your available tools
3. Delegating to appropriate sub-agents when their specialized capabilities are needed
4. Executing the necessary actions and providing clear, helpful responses
```

```
You have been working on the task described above but have not yet completed it. Write a
continuation summary that will allow you (or another instance of yourself) to resume work
efficiently in a future context window where the conversation history will be replaced with
this summary. Include: (1) Task Overview -- the core request, success criteria, and
constraints; (2) Current State -- what is completed, files created/modified/analyzed with
paths, key outputs; (3) Important Discoveries -- technical constraints, decisions and
rationale, errors and resolutions, approaches that failed and why; (4) Next Steps --
specific remaining actions, blockers, open questions, priority order; (5) Context to
Preserve -- user preferences, non-obvious domain details, promises made. Be concise but
complete; err on the side of preventing duplicate work or repeated mistakes.
```