# KAT-Coder-V2.5 Technical Report

KwaiKAT Team



## Abstract

We present KAT-Coder-V2.5, a coding-focused agentic model trained to act autonomously inside real, executable repositories rather than as a single-turn code generator. Its capability is bottlenecked less by model scale than by the scarcity of reproducible environments, verifiable rewards, and high-value trajectories, which we address with an end-to-end agentic post-training framework. AutoBuilder reconstructs multilingual repositories into sandboxed environments with fail-to-pass and pass-to-pass verification at scale, from which we regenerate self-contained task specifications, recover near-miss trajectories, and distill supervision through process-aware filtering, while KwaiClawEnv synthesizes large-scale tool-use trajectories from executable services and real task seeds. We further scale reinforcement learning with harness randomization, a reliability-hardened sandbox, an asymmetric actor–critic PPO with hindsight-augmented value estimation, and a harness-oriented reward framework, and unify SWE, Agent-Claw, and WebCoding experts via Multi-Teacher On-Policy Distillation. Across six software-engineering and agentic benchmarks, KAT-Coder-V2.5 delivers the best agentic tool-use result on PinchBench and ranks second only to the frontier Opus 4.8 on repository-level software engineering. Our service is available at `https://streamlake.com/product/kat-coder`.


Figure 1 | Results of KAT-Coder-V2.5 on various SWE and agent benchmarks.

## 1. Introduction

Coding models [1] have moved beyond passive code completion toward autonomous agents that understand requirements, navigate repositories, edit files, run tests, and iteratively repair their own failures [2, 3]. This shift redefines what capability means: a strong coding agent must combine repository understanding, tool use, long-horizon planning, engineering discipline, and verification-driven problem solving, not merely generate syntactically correct snippets. Requirements are stated in natural language, relevant logic is scattered across many files, and the agent must search, localize, design, edit, verify, and recover under limited context.

We argue that the primary obstacle to building such agents is not model scale but the training infrastructure around it, which manifests as three concrete challenges.

**Scalable executable environments.** Real repositories differ substantially in dependencies, build systems, testing frameworks, and runtime assumptions. Producing environments that are reproducible, executable, and truly verifiable is difficult at scale, and naive construction silently admits environments that never run the intended tests [4, 5].

**Trajectory quality beyond final rewards.** Raw issues and pull-request descriptions are often incomplete, ambiguous, or inconsistent with the merged change. Worse, filtering trajectories by final test success alone is misleading: some passing trajectories rely on hard-coding, mechanism bypassing, or test-oriented shortcuts, while some failed trajectories still contain valuable search, localization, and repair behavior.

**Stable long-horizon RL.** Long-horizon agentic reinforcement learning suffers from sparse rewards, unstable environment feedback, coarse credit assignment, overfitting to a fixed harness, and difficulty in fusing capabilities across specialized experts.

To address these challenges, we build KAT-Coder-V2.5 around a systematic post-training framework that treats agentic capability as a systems problem, not merely a matter of scaling data or parameters. For software-engineering tasks, we propose AutoBuilder, an agent-driven pipeline that reconstructs multilingual, reproducible, executable repository environments and formulates each SWE sample as a verifiable task defined by a task description, an execution environment, and a verifier with fail-to-pass and pass-to-pass tests. From golden and test patches, we regenerate structured, self-contained task descriptions so the model trains on clear executable requirements rather than noisy raw issue text, and we introduce process-aware trajectory construction that evaluates exploration, localization, design, editing, verification, and recovery behavior, removing passing-but-undesirable trajectories and recovering informative near-miss failures into high-quality samples. For general agentic capabilities, we develop KwaiClawEnv, a scalable environment synthesis framework organized around Service, Task, and Eval layers that produces large-scale tool-use trajectories over heterogeneous tools, multi-service workflows, and long-horizon reasoning.

During reinforcement learning, we treat the harness itself as part of the training distribution, introducing both white-box and black-box harnesses to reduce overfitting to a single tool protocol, context format, or control-flow pattern. We couple this with a reliability-hardened sandbox that reduces reward corruption from environment errors, and combine an asymmetric actor–critic PPO with a harness-oriented reward framework to deliver stable, fine-grained signals for long-horizon tasks. Finally, we train a set of specialized experts spanning agentic software engineering, agentic tool use (Claw), terminal use, web coding, and general knowledge, and fuse them through Multi-Teacher On-Policy Distillation on trajectories sampled from the student's own policy, alleviating the see-saw effect common in multi-expert merging. In this report we

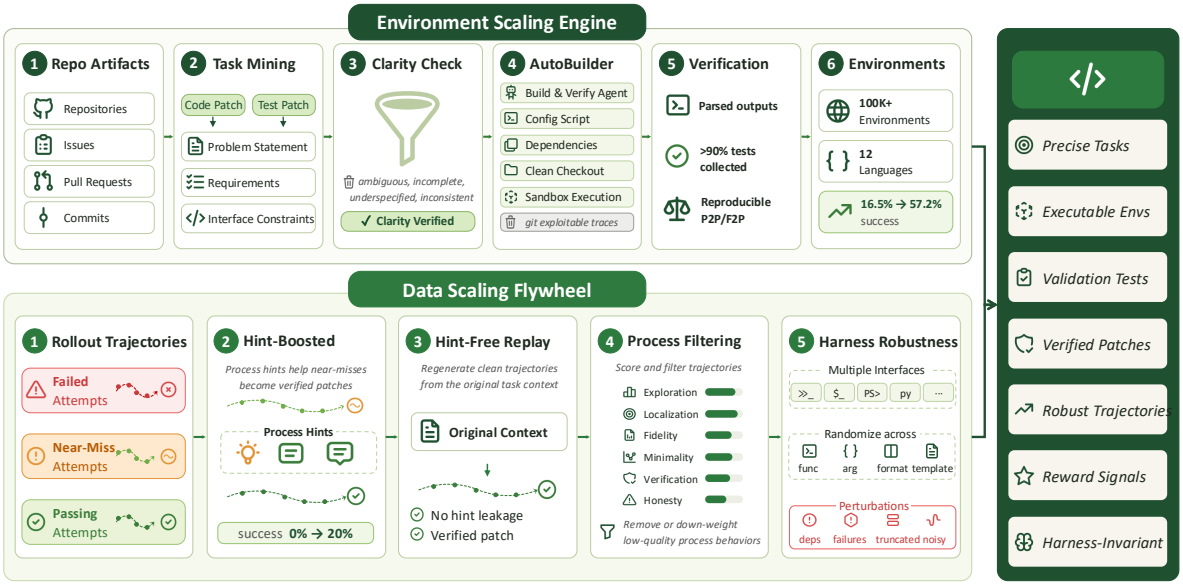


Figure 2 | The agentic software-engineering data pipelines.

focus on the software-engineering and Claw experts, which best illustrate our environment-construction and trajectory-filtering methodology, while the remaining experts follow the same recipe. Our main contributions are summarized as follows:

- We build a **verifiable SWE training infrastructure** with AutoBuilder, which constructs reproducible repository environments and grounds tasks with executable fail-to-pass and pass-to-pass verification signals.
- We introduce a **process-aware trajectory pipeline** that filters agent behavior beyond final pass rates and recovers near-miss failures through hint-free reasoning regeneration.
- We establish a **unified agentic post-training framework** that combines KwaiClawEnv, harness randomization, a reliability-hardened sandbox, asymmetric PPO, reward modeling, and multi-teacher on-policy distillation.
- We propose two internal benchmarks, **KAT Code Bench** and **KAT Claw Bench**, to evaluate coding agents under realistic engineering tasks and tool-use workflows.

Evaluated under a unified Claude Code harness across six benchmarks, KAT-Coder-V2.5 shows strong performance in agentic tool use and repository-level software engineering, achieving the top result on PinchBench and ranking second on both SWE-Bench Pro and KAT Code Bench among the evaluated models.

## 2. Agentic Software-Engineering Capabilities

Agentic software engineering requires tasks that are precisely specified, executable, and objectively verifiable in reproducible environments. We scale agentic software-engineering capabilities along two complementary axes. *Environment Scaling* constructs large-scale executable and verifiable task environments from real repositories, while *Data Scaling* turns rollout trajectories into higher-value training signals through failure recovery, process-aware filtering, and distributional generalization.

### 2.1. Environment Scaling Engine

Scaling verifiable software-engineering tasks from real-world repositories poses two central challenges. First, raw issue and PR descriptions are often ambiguous, incomplete, project-specific, or misaligned with the final merged changes, making them unreliable as task specifications. Second, repository environments are brittle to reconstruct at scale, and naive validation based on command exit codes or surface-level logs can falsely accept environments that never execute the intended tests. We address these challenges through verifiable task mining and automated environment construction.

**Verifiable Task Mining.** A verifiable task is defined as a triplet consisting of a precise task description, an executable repository environment, and a set of validation tests used to determine correctness. Given such a task, an agent starts from the initial repository state and produces a patch; the patch is considered correct if and only if it passes all validation tests.

We mine tasks primarily from real pull requests and commits, where the merged code change provides a golden patch and the accompanying test change provides a test patch [2]. However, raw issue and PR text is often unreliable: it may be vague, incomplete, inconsistent with the final merged change, or heavily dependent on project-specific context. Therefore, we regenerate structured task descriptions from the underlying verifiable artifacts. Each description contains three components: a problem statement that characterizes the bug or missing functionality, grounded mainly in the golden patch; requirements that specify the expected behavior, derived primarily from the test patch; and interface constraints that define the APIs, variables, data structures, and compatibility assumptions the solution must respect, inferred from both patches. We then apply a clarity check to remove samples whose descriptions are ambiguous, incomplete, underspecified, or internally inconsistent, ensuring that each retained task is precise, self-contained, and aligned with the validation tests.

**Verifiable Environment Construction.** We introduce **AutoBuilder**, an agent-driven pipeline for multilingual execution-environment construction. AutoBuilder operates as a sandboxed build–verification loop: a build agent analyzes the repository and generates a configuration script that installs dependencies and runs tests from a clean checkout, while a verification agent executes the script in an isolated sandbox and validates the result. Crucially, verification does not rely on command exit codes or superficial log patterns. Instead, it parses structured test-framework outputs and accepts an environment only when more than 90% of the expected tests are collected and the pass/fail outcomes are reproducible across runs. When verification fails, structured failure information is fed back to the build agent for iterative repair.

For scalability, AutoBuilder combines a preconfigured base environment, language- and build-system-specific templates, and a retrievable library of successful configurations distilled into reusable build recipes. These components reduce unnecessary trial-and-error in dependency installation and test invocation, increasing the environment-construction success rate from 16.5% to 57.2% and yielding over 100,000 verifiable environments across 12 languages.

To keep agent rollouts focused on code reasoning rather than repository setup or environment manipulation, we pre-apply dependency updates and environment-configuration edits from the reference change when they are not part of the intended programming challenge. We also remove git history, commit metadata, and other exploitable traces that could reveal the reference solution, so that agents must solve the task from the provided description, repository state, and executable tests alone.

### 2.2. Data Scaling Flywheel

Data Scaling turns rollout trajectories into higher-value training signals by recovering informative failures, filtering undesirable behavior, and broadening the deployment distribution observed during training.

**Hint-Boosted Rollout Pass Rate** Many failed trajectories are near misses: the model localizes the relevant code region but misses a decisive step, such as reading the decisive test assertion, matching an exact schema, reusing an existing mechanism, or continuing diagnosis after an initial failure. Instead of discarding such trajectories, we recover them with a two-stage hint-in-the-loop procedure. First, we inject targeted process-level hints that indicate what to inspect or verify without revealing the solution, which alone raises the pass rate of previously zero-pass tasks to roughly 20%. Since hinted trajectories may contain information unavailable at inference time, we then fix the verified patch and regenerate a hint-free trajectory from the original task context, retaining only samples that pass verification, contain no hint leakage, and remain consistent with the verified patch and test outcomes. Hints thus act only as temporary scaffolding, while the final training data stays faithful to the original task distribution.

**Process-Score-Driven Trajectory Filtering** Passing tests is necessary but not sufficient for high-quality supervision: some successful trajectories rely on brittle shortcuts, hard-coded behavior, test tampering, or weak verification. Large-scale replay reveals recurring low-quality patterns, including insufficient exploration, poor localization, poor specification matching, bypassing existing repository mechanisms, and incomplete validation. We therefore combine rule-based gates with heuristic process scoring: the rule-based stage removes invalid, unstable, or exploitative trajectories, while the scoring stage evaluates process dimensions such as exploration, localization, pre-edit reasoning, specification fidelity, adherence to repository conventions, patch minimality, verification quality, recovery behavior, and honesty. Shortcut-based successes are down-weighted or removed, and recoverable near misses are routed back into the recovery pipeline above. These process annotations also provide positive and negative trajectory signals for preference learning, rejection sampling, and process reward modeling.

**Harness Rewriting for Robustness** Training only in a fixed harness and clean sandbox can induce interface overfitting and weak recovery under realistic failures. To improve cross-harness generalization, we rewrite tool names, argument conventions, output formats, and prompt templates through randomization while preserving their underlying functionality. Because verification is anchored to structured test outcomes rather than harness-specific traces, the same task can be re-served under multiple equivalent harness configurations, forcing the model to learn harness-invariant problem-solving strategies. On top of this, we inject realistic perturbations—missing or mismatched dependencies, transient command failures, truncated outputs, and noisy logs—so the model learns to continue diagnosis and verification under anomalous execution conditions rather than stopping at the first failure signal.

## 3. General Agentic Capabilities

### 3.1. Overview of KwaiClawEnv

To evolve LLMs from passive language processing tools into agents capable of autonomous decision-making and action in real-world settings, models must develop strong multi-environment

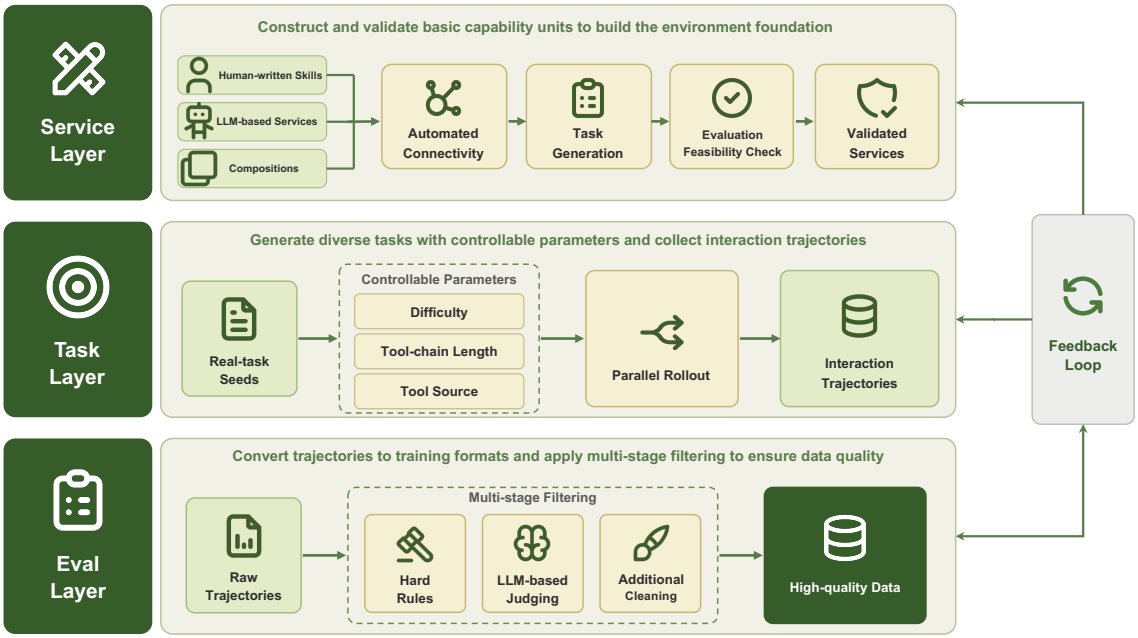


Figure 3 | Pipeline of KwaiClawEnv.

adaptability and robust tool-use capabilities. Achieving this requires large-scale training in high-quality, diverse, and interactive environments.

Existing approaches face several key limitations: real systems are difficult to access at scale due to permission, security, and stability constraints. LLM-simulated environments often suffer from hallucinations and inconsistent states. And manually built sandboxes, while reliable, are costly and hard to scale. In addition, open-source datasets are often biased toward a limited set of scenarios, leaving long-tail business needs underrepresented and restricting model generalization. Existing environment synthesis methods also tend to be environment-specific, lacking unified modeling and transferability across heterogeneous settings [6].

To address these challenges, we propose KwaiClawEnv, an environment synthesis framework designed around real business needs and Claw-style Agent tasks [7]. KwaiClawEnv adopts a structured generation paradigm based on Skill templates and real Tasks [8], ensuring semantic consistency, logical completeness, and reliable state transitions while reducing hallucination-related quality issues. The framework supports unified modeling across heterogeneous environments, enabling capability transfer across tools, tasks, and interaction workflows. It also provides controllable task complexity by flexibly adjusting tool-chain length, task steps, and interaction difficulty, supporting scalable data construction from basic tool use to complex multi-step reasoning. Overall, KwaiClawEnv offers a high-quality, cost-effective, and scalable solution for large-scale agent training [9, 10].

### 3.2. Pipeline of KwaiClawEnv

KwaiClawEnv is organized around three core questions: how to generate data, how to ensure data quality, and how to continuously improve the generation process. To address these challenges, it adopts a three-layer architecture consisting of the **Service**, **Task**, and **Eval** layers, which are connected through a closed feedback loop. The Service layer constructs executable capabilities, the Task layer generates tasks and samples interaction trajectories [11], and the Eval

layer filters and formats the resulting data while feeding quality signals back to earlier stages for iterative improvement.

#### *3.2.1. Service Layer: Capability Construction and Filtering*

The Service layer defines the atomic capability units of KwaiClawEnv and provides the executable environment for agent interaction. It supports two categories of sources: human-authored **Skills** and LLM-generated **Services**. Based on these atomic Services, the system further constructs composite capabilities by chaining or nesting multiple Services to satisfy real-world task requirements.

Before being admitted into the task-generation pipeline, each Service undergoes automatic validation to ensure executability, interface consistency, and logical correctness. Only Services that pass this validation are retained for downstream task generation and trajectory sampling.

#### *3.2.2. Task Layer: Task Generation and Trajectory Sampling*

The Task layer instantiates concrete tasks from the available Service pool and executes them to obtain complete interaction trajectories. Starting from real tasks as seeds, the system derives multiple task variants that differ in execution paths, tool combinations, and task constraints, thereby expanding task diversity while preserving real-world relevance. Task generation is governed by configurable parameters, including task difficulty, tool-chain length, and tool source. During execution, the system performs parallel rollouts and records the full interaction process, including model decisions, tool invocations, tool outputs, and state transitions. These records form end-to-end traceable trajectories from user input to final task completion.

#### *3.2.3. Eval Layer: Data Processing and Quality Control*

The Eval layer processes and filters the trajectories produced by the Task layer. Raw trajectories are first converted into a unified training format, such as SFT-ready samples, with missing or auxiliary fields populated to ensure consistency across samples. A multi-stage filtering pipeline is then applied to retain only reliable and high-quality trajectories.

The resulting quality signals are propagated back to the Service and Task layers, enabling continuous refinement of subsequent data generation. Through this closed-loop mechanism, KwaiClawEnv can iteratively improve both capability construction and task generation, ultimately producing scalable, diverse, and high-quality agent training data. Evaluation follows trustworthy agent evaluation principles including trajectory-transparent grading and multi-dimensional quality assessment [12, 13].

### 3.3. Environment Scaling

To construct large-scale, diverse, and high-quality training data, KwaiClawEnv introduces a two-level scaling mechanism that combines Service-level expansion with Task-level derivation. By jointly scaling executable Services and derived Tasks, the system amplifies a small set of high-quality seeds into tens of thousands of diverse interaction trajectories, enhancing the model's adaptability to long-tail scenarios, heterogeneous tools, and complex workflows.

#### *3.3.1. Service-level Scaling: From Skills to Executable Environments*

At the Service layer, KwaiClawEnv employs a dual-source generation strategy. The first source leverages standardized Skill definitions from open-source communities such as OpenClaw. The system parses these definitions to extract API specifications, parameter schemas, and usage constraints, then converts them into deployable services equipped with OpenAPI specifications, container configurations, and fixture data. This approach inherits quality signals from curated community Skills and achieves a generation success rate exceeding 90%.

The second source uses category-guided LLM generation to synthesize new Skill variants for underrepresented domains. Starting from a set of atomic Services, the system further constructs composite environments through service chaining and orchestrated combination. After applying semantic constraints and multi-stage validation, a large pool of high-quality environment compositions is retained, expanding the environment scale by an order of magnitude compared with manual construction.

#### *3.3.2. Task-level Scaling: Seed Derivation and Complexity Control*

At the Task layer, KwaiClawEnv employs a structured derivation engine to expand real business tasks into large-scale training instances. The process begins with high-quality seed tasks, each defined by a clear objective, tool-use exemplars, and machine-verifiable success criteria. The engine then generates variants through three mechanisms: parameter expansion, constraint augmentation, and tool-chain orchestration.

Task complexity is governed by configurable controls including difficulty ratio, tool-chain length bounds, and tool source composition. In practice, KwaiClawEnv generates millions of candidate tasks and retains over one hundred thousand high-quality instances after multi-stage verification. The resulting trajectories contain an average of 15 tool calls, with the longest exceeding 100 steps, covering scenarios that range from basic single-tool usage to complex long-horizon reasoning.

### 3.4. Data Validation and Reliability Assurance

KwaiClawEnv significantly improves the efficiency of environment construction. However, large-scale automated generation also introduces new quality-control challenges. In early experiments, we observed consistency issues across service generation, task synthesis, and evaluation execution. For instance, generated task files may fail schema validation, tool calls may reference unimplemented endpoints, and synthesized trajectories may conflict with fixture data. While such issues are manageable at a small scale, they can accumulate and propagate as the system scales to hundreds of services and thousands of tasks.

To address these challenges, we design an end-to-end consistency validation framework that covers the full lifecycle from service generation to task execution. The framework follows a progressive three-stage pipeline. First, **service availability validation** checks endpoint reachability, OpenAPI specification completeness, and inter-service dependency connectivity. Second, **task generation validation** verifies schema legality, tool-reference correctness, parameter consistency, and the machine-verifiability of success criteria. Third, **execution environment validation** runs tasks within a containerized sandbox to verify service startup, fixture loading, agent interaction, trajectory integrity, and scoring correctness. Each sample is assigned a unified disposition, namely *repairable failure*, *rejected defect*, or *production-ready output*, enabling targeted remediation.

At the trajectory level, KwaiClawEnv further applies a two-layer filtering strategy. The

first layer, **hard-rule filtering**, removes unsafe, invalid, or hallucinated trajectories through deterministic checks, including tool-blacklist detection, file-existence validation, mandatory tool-coverage verification, and state-consistency enforcement. The second layer, **LLM-as-Judge evaluation**, scores the remaining trajectories along three dimensions: semantic correctness, execution efficiency, and interaction naturalness. Only trajectories that pass both layers are retained for training.

This framework enables KwaiClawEnv to expand a limited set of atomic services into diverse, high-quality environments and to produce reliable trajectories covering representative scenarios such as coding, writing, and data analysis.

## 4. Reinforcement Learning

### 4.1. Harness Scaling

In agentic RL, if training relies solely on a single fixed harness, the model often learns not "how to solve the task" but "how to solve the task under that particular harness's interface conventions." This implicit coupling manifests as overfitting in three forms:

- Format overfitting: The model becomes anchored to a specific action format, and the parsing failure rate rises significantly once the tool-invocation protocol is switched.
- Context-structure overfitting: The model depends on the training harness's specific history-concatenation order, and its behavior degrades when the context layout is rearranged.
- Control-flow overfitting: The model relies on the reflection timing and stopping conditions provided by the training harness, and fails to plan autonomously in harnesses that lack such explicit scaffolding.

Therefore, we introduce diverse harnesses during the RL rollout phase, so that the model's competence consolidates at the level of "task solving" rather than "interface adaptation," thereby achieving cross-harness generalization. In essence, this is a form of domain randomization applied at the environment level. The key to Harness Scaling lies not in the number of harnesses but in whether the diversity falls along dimensions that are useful for generalization. We systematically construct diversity along the following "axes of variation":

- Tool-invocation protocol: structured function-calling, text code-block protocols, tag-based protocols, and others, mitigating format overfitting.
- Context-management strategy: full history, sliding window, summary compression, and different observation-truncation strategies, improving the model's robustness in scenarios with incomplete information.
- Control-flow complexity: ranging from minimal ReAct to complex control flows with explicit planning and self-reflection, enabling the model to operate under both "guided" and "unguided" conditions.

Based on the considerations above, we categorize harnesses into two classes according to how they organize trajectories. Both participate in RL training but serve distinct roles: (1) **White-box harness**: mini-swe-agent. Its control flow is simple, it performs no trajectory compression, and its tool-invocation scale is relatively small. The trajectory structure is clear and closely mirrors the raw "task-solving" process. This type of harness provides clean, low-noise training signals, helping to directly strengthen the model's intrinsic, foundational agentic capabilities. (2) **Black-box harnesses**: ClaudeCode, Codex, OpenClaw, OpenHands and etc. These cover the diverse

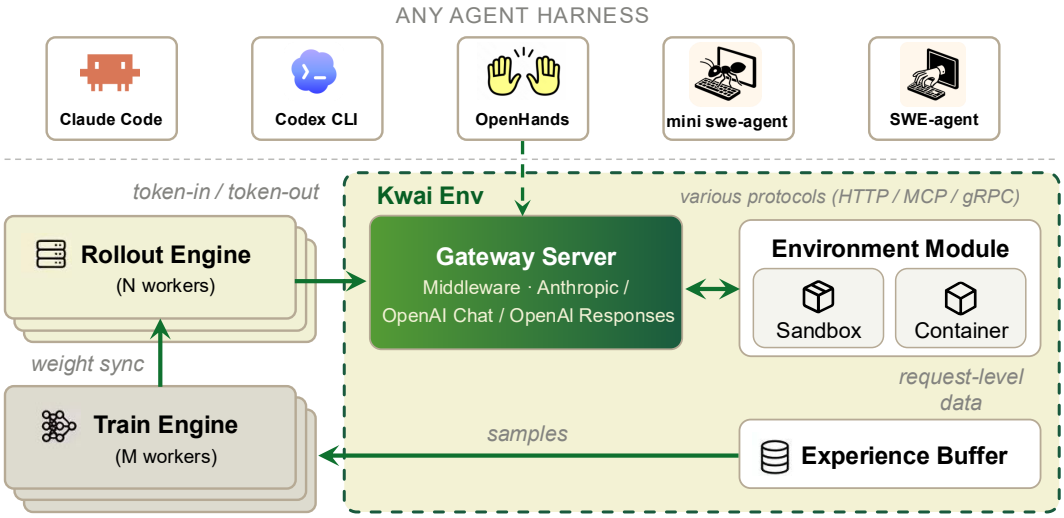


Figure 4 | Overall architecture of our Agentic RL training infrastructure.

tool-exposure forms and control-flow complexities found in real deployment scenarios, and they commonly introduce trajectory-compression and context-reorganization mechanisms, making them the closest match to the actual deployment distribution. Training on this type of harness allows the model to adapt to incomplete-information conditions in which the context has been truncated, compressed, or reorganized, thereby improving its robustness and generalization in real-world complex environments.

### 4.2. RL and Sandbox Infrastructure

Similar to KAT-Coder-V2 [14], our training infrastructure consists of three core modules: (1) **Rollout Engine**, which is responsible for policy-model inference and generates model responses during the agent's interaction with the environment; (2) **Train Engine**, which performs policy updates based on the collected trajectories; and (3) **KwaiEnv**, which manages environments and integrates harnesses by hosting corresponding execution environments.

#### *4.2.1. Gateway Server: Bridging the Model and the Harness*

To better support harness scaling, we introduce an additional **Gateway Server** module on top of KwaiEnv. The Gateway Server serves as an intermediary layer that connects the inference–interaction loop with the training loop. The Rollout Engine and the environment module do not communicate directly; instead, all traffic is mediated by the Gateway Server. Once a trajectory is completed, the Gateway Server writes it into the Experience Buffer, from which the Train Engine samples batches for policy updates. This design decouples the inference–interaction loop from the training loop and fully isolates the execution details of the harness from the trainer. As a result, an arbitrary harness with an arbitrary execution environment can be mounted onto the training system as an opaque black box, without exposing its internal state machine to the trainer.

The second responsibility of the Gateway Server is to enforce token consistency. Mainstream inference engines typically expose a `chat` endpoint, which internally re-applies `apply_chat_template` and re-tokenizes the request. For long-horizon agentic tasks, the resulting token drift is non-negligible. In our experiments on agentic tasks at the ~200-turn scale, we observe

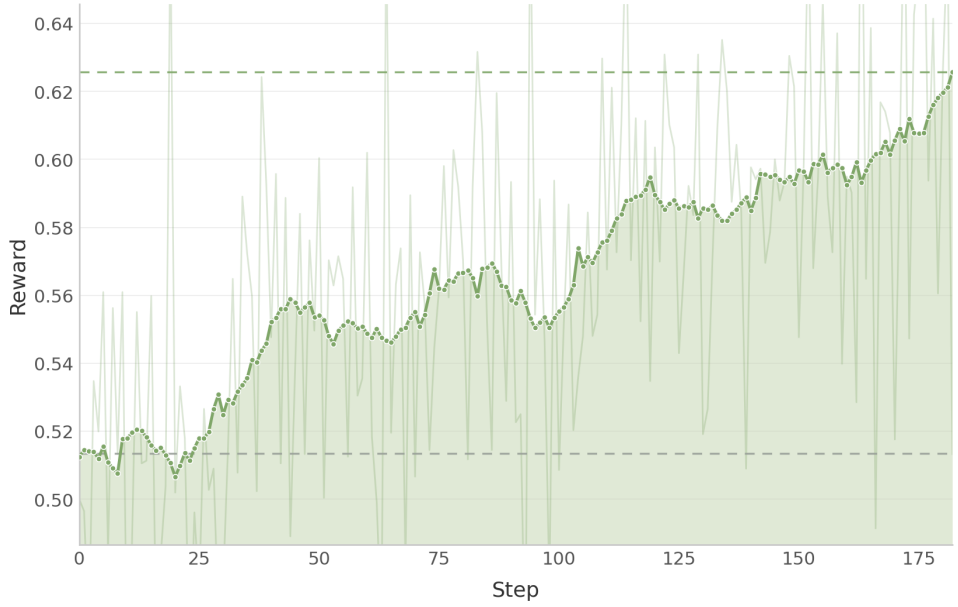


Figure 5 | RL training curve in the SWE scenario. The reward exhibits a consistently stable upward trend throughout training.

token drift in approximately **40%** of the samples. This phenomenon, commonly referred to as **retokenization drift**, has also been documented in NVIDIA Polar, as well as in the vLLM and Agent Lightning ecosystems. Our implementation bypasses the `chat` interface entirely and sends every request directly to the inference backend through its `/generate` endpoint. This eliminates retokenization drift at its source and guarantees that every trainable token is identical to the token emitted by the behavior policy during rollout.

#### *4.2.2. Sandbox Optimization: Improving Reward Signal Reliability*

During the early stages of agentic RL training for KAT-Coder-V2, we frequently encountered training crashes. More commonly, the reward curve improved only slowly, and convergence was substantially delayed. We initially attributed these failures to the RL algorithm itself, investing considerable effort in hyperparameter tuning and exploring new algorithmic variants. However, as we scaled up the number of training samples and introduced deeper supervision mechanisms, we found that a significant portion of training failures actually originated from the training environment rather than the learning algorithm. In a sampling-based audit of early rollouts, roughly 16% of trajectories contained at least one failure attributable to the sandbox itself rather than to the model policy, which explains why tuning RL hyperparameters alone was insufficient to improve the reward curve. These environment-related failures were often infrequent and difficult to reproduce, but their impact was severe. Once triggered, they could cause training samples to receive incorrect sandbox feedback, thereby corrupting reward signals and degrading the stability of RL training. In the most severe cases, a single sandbox boundary misalignment could cause the observations of the subsequent ~40 rollout steps to become empty, corrupting the reward signal of the entire trajectory.

During the development of KAT-Coder-V2.5, we devoted substantial effort to diagnosing and resolving a wide range of environment-related issues. These issues can be broadly categorized into two types: **sandbox stability** and **sandbox execution correctness**.

For sandbox stability, a representative example is container image management. RL training requires pulling a large number of container images concurrently, and the images used by our latest training environments are substantially large. This nearly exhausted the physical disk capacity of the machines hosting the sandbox service, which in turn triggered frequent garbage collection during execution. As a result, sandbox initialization and execution occasionally became slow enough to cause timeouts. Prior to optimization, peak-time disk usage reached roughly 95%, garbage collection ran almost continuously, and timeout-induced invalid rollouts accounted for approximately 6%–7% of all rollouts.

To address this issue, we redesigned the image management module of the sandbox service and introduced an early-release policy that proactively removes images unlikely to be reused in subsequent steps. After this optimization, steady-state disk usage dropped to roughly 60%, and the fraction of timeout-induced invalid rollouts fell to below 1%. This optimization also reduced sandbox cold-start latency and improved rollout efficiency during RL training. Most sandbox execution correctness issues originated from bugs in our framework. For example, certain system environment variables set during remote sandbox initialization could override system configurations, causing some verifiers to read incorrect environment variables and produce erroneous verification results. Prior to the fix, this class of issues corrupted verifier outputs on roughly 6%–7% of samples, effectively flipping the corresponding reward signals. After the fix, the error rate dropped to below 1%.

After resolving the majority of these issues, training collapses became rare. Overall, the sandbox feedback error rate dropped from roughly 16% to below 2%, and the frequency of training collapses decreased by approximately an order of magnitude. These improvements enabled the RL training of KAT-Coder-V2.5 to proceed smoothly and reliably.

### 4.3. Asymmetric PPO for Long-Horizon Tasks

Agentic RL is naturally a long-horizon, partially observable, and stochastic-transition problem. In this setting, trajectory-level critic-free methods such as GRPO [15] and its anchor-state variants [16] suffer from coarse credit assignment and high gradient variance. We therefore adopt Proximal Policy Optimization (PPO) [17] as the algorithmic backbone.

PPO is preferable for three reasons. First, production-grade harnesses often produce multiple structurally split samples from the same session, e.g., after compacting, sub-agent splitting, or query rewriting. These samples may share the same final outcome but have different prefixes, making trajectory-level group baselines difficult to define consistently, whereas PPO supports per-token gradient contributions naturally. Second, retaining a Critic allows us to inject training-time privileged information, such as final rewards, test outcomes, and patch-level signals, into value estimation, reducing variance and improving sample efficiency. Third, PPO combined with Generalized Advantage Estimation (GAE) and reward shaping enables turn-level credit assignment, allowing us to penalize localized bad behaviors such as erroneous tool calls or regressive intermediate patches without uniformly affecting the whole trajectory.

#### *4.3.1. PPO Objective and Advantage Estimation*

We use the standard clipped PPO objective:

$$\mathcal{J}_{\text{PPO}}(\theta) = \mathbb{E}_{q\sim P(Q),\, o\sim\pi'(\cdot|q)}\left[\frac{1}{|o|}\sum_{t=1}^{|o|}\min\left(r_t\,\hat{A}_t,\ \text{clip}(r_t, 1-\epsilon, 1+\epsilon)\,\hat{A}_t\right)\right], \tag{1}$$

where $r_t = \pi_\theta(a_t \mid s_t)/\pi'(a_t \mid s_t)$ is the per-token importance ratio and $\pi'$ is the behavior policy. The advantage is estimated using GAE:

$$\hat{A}_t = \sum_{l=0}^{T-t-1} (\gamma\lambda)^l \delta_{t+l}, \qquad \delta_t = r_t + \gamma V'(s_{t+1}) - V'(s_t), \tag{2}$$

where $V'$ denotes the value network from the previous iteration. The Critic is trained by regressing to the return target $R_t = \hat{A}_t + V'(s_t)$:

$$\mathcal{L}_{\text{critic}}(\psi) = \mathbb{E}_{(s_t,R_t)\sim\mathcal{D}} \left[ \left( V(s_t;\psi) - R_t \right)^2 \right]. \tag{3}$$

#### *4.3.2. Hindsight-Augmented Critic*

Following asymmetric actor-critic and centralized-critic paradigms [18, 19], we let the Critic access privileged hindsight information during training, while the Actor only observes the normal harness state available at rollout time. This design improves value estimation without changing the deployed policy.

At rollout, the Actor observes only the interaction history up to the current turn, including tool outputs, file snippets, and compacted dialogue summaries. At training time, the Critic additionally receives a hindsight context $c_t$, which may include the final pass/fail reward, unit-test outcome distribution, coverage signals, patch-level differences, task metadata, trajectory statistics, and subsequent turns. The asymmetric value function is therefore written as $V_\psi(s_t, c_t)$.

The asymmetric Critic is trained with the same MSE objective:

$$\mathcal{L}^{\text{asym}}_{\text{critic}}(\psi) = \mathbb{E}_{(s_t,c_t,R_t)\sim\mathcal{D}} \left[ \left( V(s_t, c_t;\psi) - R_t \right)^2 \right], \qquad R_t = \hat{A}_t + V'(s_t, c_t). \tag{4}$$

GAE is computed as in Eq. 2, with each value term replaced by the asymmetric value function $V(\cdot, c_t;\psi)$. The Actor still optimizes the standard PPO objective in Eq. 1. At inference time, the Critic and hindsight context are discarded, and only the Actor is deployed.

### 4.4. Harness-Oriented Reward Framework

To address common challenges in harness-based reinforcement learning, including sparse reward signals, reward-hacking behaviors, non-standard operational patterns, and the lack of effective learning signals from failed trajectories, we design a harness-guided reward shaping framework that combines rule-based verification with model-based evaluation.

#### *4.4.1. Rule-based Reward*

We design a three-layer, structured rule-based reward system built on top of harness execution feedback, with the full set of rules summarized in Table 1. The reward is organized into three tiers: **Core Task Score**, **Standard Behavior Constraints**, and **Failed Trajectory Incentives**. These tiers respectively capture final task success, process-level engineering discipline, and partial progress in incomplete trajectories. By jointly optimizing for task completion and standardized development behavior, the system provides fine-grained supervision for agent behavior.

**Core Task Score.** The Core Task Score is the primary reward component and carries the highest weight in the overall reward system. It serves as the main criterion for evaluating task

completion and contains no sub-items. A full score is awarded only when all `fail_to_pass` test cases, i.e., defect-revealing tests that should be fixed, and all `pass_to_pass` test cases, i.e., baseline regression tests that should remain stable, are passed. This design prevents the agent from exploiting reward-hacking strategies such as generating invalid code, bypassing intended logic, or weakening test cases. By requiring both bug-fixing success and regression stability, the Core Task Score anchors optimization to the true objective of the task: fully resolving the target defect while preserving existing functionality.

**Standard Behavior Constraints.** This layer applies auxiliary penalties throughout the entire interaction trajectory. Through rule-based verification, it penalizes inefficient, abnormal, or non-standard behaviors regardless of the final task outcome. The goal is to encourage standardized, efficient, and clean engineering behavior in long-horizon agent interactions. This layer includes the following components:

- **Content Duplication Rate**: Penalizes redundant or repeated text in the thinking content and final response, reducing textual redundancy and improving information density.
- **Garbled Content Rate**: Penalizes garbled characters, illegal symbols, and other abnormal output patterns, reducing invalid content and avoiding downstream parsing errors.
- **Tool Call Accuracy**: Detects tool-call failures caused by missing or incorrect parameters, marked by `<tool_use_error>`, encouraging standardized tool invocation.
- **Abnormal Tool Call Position Rate**: Penalizes tool-call instructions incorrectly placed inside the `reasoning` field, ensuring that tool calls follow the expected format and can be parsed and executed reliably.
- **Repeated Single-Round Tool Calls**: Penalizes repeated calls to the same tool within a single interaction round, reducing redundant interactions.
- **Tool Call Parallelism**: Penalizes excessive parallel tool calls beyond a predefined threshold, while allowing reasonable batch calls to improve efficiency without compromising training stability.

**Failed Trajectory Incentives.** A key limitation of conventional reinforcement learning for agentic tasks is that incomplete or failed trajectories often receive zero reward. Consequently, the model obtains little feedback for partially correct intermediate behaviors, which weakens learning efficiency. To mitigate this issue, we introduce a failed trajectory incentive layer that assigns positive rewards to meaningful progress within failed trajectories, thereby densifying sparse reward signals. This layer contains two components:

- **File Search Accuracy**: Evaluates repository file retrieval with an $F_2$ score that jointly considers precision and recall, encouraging the agent to locate relevant files while avoiding meaningless broad searches.
- **Unit Test Pass Rate**: Assigns rewards based on the pass status of the two types of unit tests. The model receives positive feedback when it passes any subset of test cases, providing useful learning signals for incomplete trajectories.

The reward tiers form a top-down, full-scenario guidance framework: (1) The top-level core reward anchors the ultimate task objective and prevents the model from deviating from the primary problem-solving goal. (2) The middle-layer behavior constraints uniformly regularize all trajectories, continuously improving the agent's output quality, tool usage, and engineering behavior. (3) The bottom-layer failed trajectory incentives compensate for reward sparsity and

improve the training value of incomplete or failed samples. Through the synergy of these reward components, the overall reward system improves the pass rate of code-repair tasks, reduces token consumption during interactions, and standardizes the behavior.

#### *4.4.2. Model-Based Reward*

Rule-based rewards are effective for easily verifiable signals, such as test outcomes and tool misuse, but they are insufficient for evaluating context-dependent behaviors. Assessing test sufficiency, strategy adaptation, and failure coverage requires holistic trajectory-level reasoning. Exhaustive hard-coded rules would introduce high complexity, poor generalization, and potentially unfair penalties for valid exploration. We therefore introduce a model-based judge, guided by a trajectory-level rubric, to complement rule-based rewards.

The rubric converts key code-repair behaviors into multi-dimensional process rewards, providing dense and fine-grained feedback beyond sparse task-level outcomes. It is designed to encourage robust, efficient, and auditable problem-solving behavior.

We construct the rubric through manual failure analysis of real sampled trajectories, covering tool sequences, environment feedback, code edits, and test operations. Table 3 shows representative bad cases identified in this analysis. From these cases, we extract recurring, evidence-backed failure patterns that significantly impair task performance, and abstract them into standardized criteria with explicit triggers, applicable scopes, and reward signals.

Our analysis shows that performance bottlenecks often arise from unstable execution strategies rather than incorrect code edits alone. Typical failure patterns include running tests without validating entry points or environments, blindly scanning large files, overusing Bash instead of specialized code tools, and skipping targeted regression tests after fixes. We organize the model-based process rewards into three dimensions: **fault diagnosis and reproduction**, **post-fix verification**, and **execution strategy**. Representative examples are given in Table 2.

**GRM Training.** The raw base model fails to strictly follow rubric rules during trajectory evaluation. We apply targeted RL to train a specialized judge model GRM for consistent rubric violation detection. Training samples are historical agent trajectories paired with human annotations of triggered rubric criteria and rationales. We manually filter labels to keep only fully evidence-backed ones, eliminating ambiguous data to help the GRM learn reliable classification boundaries. We define the trajectory reward for GRM optimization:

$$r = \begin{cases} \dfrac{|GT \cap Pred|}{|GT|} - \left|Pred \setminus GT\right| \cdot \lambda, & GT \neq \emptyset \\ 1 - \left|Pred \setminus GT\right| \cdot \lambda, & GT = \emptyset \end{cases} \tag{5}$$

The recall-based term maximizes ground-truth coverage, while the penalty term restrains excessive false predictions. After RL, the GRM better incorporates full trajectory context when judging post-fix validation behaviors, reducing errors from superficial keyword matching.

## 5. Multi-Teacher On-Policy Distillation

After training domain-specific experts for agentic software engineering, general agentic reasoning, terminal use, web coding, and general knowledge, the key post-training challenge is to fuse their capabilities into a single unified student model. Conventional merging methods,

such as parameter averaging, Task Arithmetic [20], or sequential multi-domain SFT, often suffer from a *see-saw effect*: improving one domain may degrade another. This is mainly because parameter-space merging can disrupt expert-specific structures, while offline data-space supervision introduces train–inference distribution mismatch.

To address this, we introduce **Multi-Teacher On-Policy Distillation** (MOPD), which performs capability fusion in the function space. Let the student policy be $\pi_\theta$, and the domain experts be $\{\pi_{T_k}\}_{k=1}^{K}$, where $K = 5$ corresponds to five experts. For each sample $(x, d)$, where $d$ denotes its domain, the student first generates an on-policy trajectory $y \sim \pi_\theta(\cdot \mid x)$. The corresponding domain teacher $\pi_{T_d}$ then provides token-level logit supervision on the same trajectory. The student is optimized with reverse KL:

$$\mathcal{L}_{\mathrm{MOPD}}(\theta) = \mathbb{E}_{(x,d)\sim\mathcal{D}}\mathbb{E}_{y\sim\pi_\theta(\cdot|x)}\left[\sum_{t=1}^{|y|} w_t \,\mathrm{KL}\big(\pi_\theta(\cdot \mid x, y_{<t}) \,\big\|\, \pi_{T_d}(\cdot \mid x, y_{<t})\big)\right], \tag{6}$$

where $w_t \in [0, 1]$ is a drift-aware token weight. The reverse KL objective is mode-seeking, encouraging the student to concentrate probability mass on the teacher's high-confidence regions and reducing cross-domain interference in multi-teacher distillation [21].

### 5.1. Stabilizing Long-Context On-Policy Distillation

Pure on-policy distillation becomes unstable on long-context tasks such as multi-turn agent trajectories and repository-level software engineering tasks. As the student-generated prefix grows, it may diverge from the teacher's training distribution. In this case, the teacher distribution conditioned on the student prefix becomes unreliable, producing biased token-level KL signals. This can lead to loss oscillation, entropy collapse, and gradient norm spikes, especially because reverse KL may further amplify over-confident collapse toward an incorrect local mode. We address this issue with two stabilizing mechanisms: off-policy cold start and drift-aware dynamic truncation.

**Off-policy cold start.** Before on-policy MOPD, we initialize the student with expert-generated trajectories:

$$\mathcal{L}_{\mathrm{cold}}(\theta) = \mathbb{E}_{(x,d)\sim\mathcal{D}}\mathbb{E}_{y\sim\pi_{T_d}(\cdot|x)}\left[-\sum_{t=1}^{|y|}\log\pi_\theta(y_t \mid x, y_{<t})\right]. \tag{7}$$

This phase pre-aligns the student with the teacher distributions, reduces early prefix divergence, and accelerates convergence in the subsequent on-policy stage.

**Drift-aware dynamic truncation.** Even after cold start, local student and teacher drift may still appear in the latter part of long trajectories. Following the top-$k$ overlap idea in Prune-OPD [22], we define the compatibility between the teacher and student at token position $t$ as

$$\rho_t = \frac{|\mathcal{T}_t^k \cap \mathcal{S}_t^k|}{k}, \qquad \rho_t \in [0, 1], \tag{8}$$

where $\mathcal{T}_t^k$ and $\mathcal{S}_t^k$ are the teacher's and student's top-$k$ prediction sets. A high $\rho_t$ indicates that the teacher still provides reliable supervision under the current student prefix.

We use $\rho_t$ to control token weights and truncation. Specifically, $w_t$ in Eq. 6 is set according to a monotonic function of $\rho_t$, and tokens with low compatibility receive reduced or zero weight.

If $\rho_t$ stays below a hard threshold for $m$ consecutive tokens, we truncate the trajectory and stop backpropagating through subsequent tokens. This removes unreliable teacher supervision while reallocating compute to new samples. To avoid length bias caused by truncating long trajectories, we retain all valid prefix tokens before the truncation point, apply truncation only as gradient masking rather than an explicit optimization objective, and use length-stratified batching to preserve the proportion of long-context samples. Together, MOPD combines multi-teacher on-policy reverse-KL supervision, off-policy cold start, and drift-aware truncation. This enables stable long-context distillation and effectively fuses all five domain capabilities—spanning agentic software engineering, general agentic reasoning, terminal use, web coding, and general knowledge—into a single unified student model without the severe cross-domain degradation typical of weight-space merging.

## 6. Evaluation

We conduct a comprehensive evaluation of KAT-Coder-V2.5 across the full spectrum of coding-agent competencies, spanning repository-level software engineering, long-horizon agentic tool use, and terminal and scientific coding. To ensure fair and reproducible comparison, all systems are evaluated under an identical protocol: unless otherwise noted, we adopt **Claude Code** as the unified evaluation harness and hold the tool set, context budget, execution environment, and decoding configuration fixed across models, so that the measured gaps are attributable to model capability rather than to scaffold or environment discrepancies.

### 6.1. Benchmarks

**KAT Code Bench** To measure the practical utility of coding agents in authentic software-engineering settings, we curate KAT Code Bench, a repository-level SWE suite distilled from Kuaishou's internal development workload. Unlike code completion, single-file repair, or competitive-programming tasks, KAT Code Bench evaluates the full engineering loop: parsing a natural-language requirement, navigating a real codebase to localize cross-file edit sites, following the repository's established design conventions, producing a minimal yet correct patch, and closing an executable verification loop that certifies the change introduces no regressions. The suite is designed around four principles: **authenticity**, **verifiability**, **reproducibility**, and **discriminative power**. It spans 12 programming languages and covers heterogeneous frameworks, project structures, and change types, including defect fixes, feature completion, interface compatibility, behavioral-consistency correction, cross-module edits, and regression repair. Each task pins its base commit, runtime environment, and verification entry point, ensuring that all models operate under identical context, tooling, and resource budgets. This design isolates intrinsic SWE capability from variance introduced by scaffolds or environments. During curation, we explicitly suppress recurrent sources of measurement noise, including irreproducible environments and flaky tests, verifiers that are over-coupled to the reference implementation and reject behaviorally correct but structurally different solutions, description–verifier mismatches, and over-templated tasks that leak the intended implementation path and weaken discriminative power. After iterative sampling, execution, human review, and trajectory auditing, we retain only tasks that stably reflect a model's true SWE capability. Every task is solved by an agent inside a controlled sandbox. Beyond the final pass rate, we log the complete behavioral trajectory, making KAT Code Bench not only a cross-model benchmark but also a substrate for downstream data selection, failure attribution, and data augmentation.

**KAT Claw Bench** To evaluate how well LLMs perform in realistic business-oriented tool-use scenarios, we build KAT Claw Bench, a benchmark suite derived from Kuaishou's internal agent application needs. Existing Claw-type benchmarks exhibit significant limitations in practice: task granularity is too fine, mostly remaining at the script-level or single-point scope without assessing complete task chains; scenario coverage is insufficient for critical production needs such as project-level delivery, complex agent workflows, and business analysis; and evaluation tasks diverge from actual business contexts, failing to reflect real-world model performance. These gaps cause benchmark results to deviate from production reality, providing unreliable guidance for model selection and optimization. To address these limitations, KAT Claw Bench complements capability dimensions that general benchmarks cannot reach, more accurately reflects model usability and delivery value in real business contexts, and provides targeted evaluation evidence to support model selection and deployment decisions.

KAT Claw Bench is constructed through a structured pipeline to ensure authenticity, reproducibility, and evaluation reliability. We first collect and expand representative queries from real business scenarios, then filter them based on authenticity, feasibility, and evaluability. Retained queries are converted into complete benchmark tasks with standardized descriptions, required materials, category labels, difficulty annotations, and scoring rubrics. Scoring adopts automatic checks for objective outputs-such as file existence, numerical correctness, format compliance, and code executability-and hierarchical evaluation that combines automatic validation with human judgment for open-ended deliverables, with the specific combination determined by task type. To improve quality, each task undergoes multi-model review to verify self-containedness and scoring validity, as well as cross-model validation to detect ambiguous requirements or unstable criteria. Tasks with unclear instructions or inconsistent scoring behavior are revised or removed.

KAT Claw Bench is organized by an operation-oriented taxonomy covering seven major categories: personal productivity and office work, content creation and operations, software development and engineering, data analysis and insights, information retrieval and processing, automated monitoring and alerting, and investment analysis and decision-making. These tasks span different scenarios, including short video, live streaming, e-commerce, advertising, and workplace automation, with emphasis on realistic business workflows such as project-level software delivery, platform-aware content generation, decision-oriented data analysis, and continuous competitive intelligence tracking.

### 6.2. Main Results

Table 4 reports the head-to-head comparison of KAT-Coder-V2.5 against a panel of leading frontier models—GLM-5.1, GLM-5.2, Kimi-K2.6, and Opus 4.8—across all six benchmarks.

**Repository-level software engineering.** On the Coding category, KAT-Coder-V2.5 ranks second on both SWE-Bench Pro and KAT Code Bench, trailing only the frontier general-purpose Opus 4.8 while outperforming the GLM-5 series and Kimi-K2.6 by clear margins. Among all evaluated peers it is thus the strongest system short of the leading frontier model on realistic repository-level engineering, reflecting robust repository comprehension, cross-file localization, and verification-driven repair.

**Long-horizon agentic tool use.** On the Claw category, KAT-Coder-V2.5 attains the best overall result on PinchBench, edging out Opus 4.8 and surpassing the GLM-5 series by a wide margin.

On the more demanding, business-grounded KAT Claw Bench it remains tightly competitive with the strongest proprietary and open peers. These results confirm that our agentic post-training, harness scaling, and large-scale multi-tool trajectory learning translate into robust workflow execution, tool selection, and state tracking under interactive environments.

Table 4 | Comparison between KAT-Coder-V2.5 and open-source/proprietary models. The highest score for each benchmark is **bolded**, and the second highest is underlined. PinchBench scores are averages (Avg) taken from `https://pinchbench.com/` (retrieved on July 2, 2026). Data points marked with * are for Kimi-K2.7-Code.

| | KAT Coder V2.5 | GLM 5.1 | GLM 5.2 | Kimi K2.6 | Opus 4.8 |
|---|---|---|---|---|---|
| *Coding* | | | | | |
| SWE-Bench Pro | 65.2 | 58.4 | 62.1 | 58.6 | **69.2** |
| KAT Code Bench | 53.1 | 49.6 | 50.3 | 48.9 | **57.3** |
| *Claw* | | | | | |
| PinchBench(Avg) | **94.9** | - | 87.0 | 80.7* | 93.5 |
| KAT Claw Bench | 85.5 | 84.4 | 86.8 | 85.2 | **90.7** |
| *AA Coding Index* | | | | | |
| Terminal-Bench 2.1 | 60.7 | 61.8 | 77.9 | 73.0 | **84.6** |
| SciCode | 50.3 | 43.8 | 50.5 | **53.5** | **53.5** |

**Terminal and scientific coding.** On the AA Coding Index category, larger general-purpose models retain an edge on these broader, less repository-centric tasks; nevertheless KAT-Coder-V2.5 stays competitive on scientific programming, remaining on par with GLM-5.2.

Taken together, the results delineate a clear capability profile: best-in-class agentic tool use and near-frontier repository-level software engineering, achieved with an optimization budget deliberately concentrated on the tasks that dominate real-world development.

## 7. Conclusion

In this report, we presented KAT-Coder-V2.5, a coding-focused agentic model built on the premise that the primary bottleneck to stronger coding agents lies in training infrastructure rather than model scale. We addressed this through an end-to-end agentic post-training framework: AutoBuilder reconstructs multilingual repositories into reproducible, executable environments with fail-to-pass and pass-to-pass verification, from which we regenerate self-contained tasks, recover near-miss trajectories, and distill supervision via process-aware filtering, while KwaiClawEnv synthesizes large-scale tool-use trajectories through a scalable closed-loop pipeline. On top of these environments, we scaled reinforcement learning with harness randomization, a reliability-hardened sandbox that cut the feedback error rate from roughly 16% to below 2%, an asymmetric actor–critic PPO with hindsight-augmented value estimation, and a harness-oriented reward framework, and finally fused five domain experts into a single student via Multi-Teacher On-Policy Distillation while mitigating the see-saw effect of weight-space merging.

Across six software-engineering and agentic benchmarks, KAT-Coder-V2.5 delivers the best agentic tool-use result on PinchBench and ranks second only to the frontier Opus 4.8 on

repository-level software engineering, while remaining competitive on terminal and scientific coding. These results indicate that treating environment construction, trajectory quality, and RL stability as first-class systems problems yields a clear capability profile achieved by concentrating the optimization budget on the tasks that dominate real-world development. Extending verifiable environment construction to broader settings, strengthening long-horizon credit assignment, and improving generalization on terminal and scientific tasks remain promising directions, and we hope the methodology described here offers a reusable foundation for the next generation of autonomous coding agents.

## 8. Contribution

Contributors' names are listed in alphabetical order by first name.

**Core Contributors**

| | | | | |
|---|---|---|---|---|
| Bo Huang | Fengxiang Li | Hao Xu | Haoyang Huang | Hongyi Fu |
| Jinhua Hao | Kun Yuan | Minglei Zhang | Pengcheng Xu | Shiyang Liu |
| Wenhao Zhuang | Yuze Shi | Zongxian Feng | | |

**Contributors**

| | | | | |
|---|---|---|---|---|
| Chao Wang | Cheng He | Chongling Rao | Deyu Cao | Fan Yang |
| Gang Xiong | Haochen Liu | Jiabao Li | Jian Liang | Jinghui Jia |
| Jingwen Chang | Jun Du | Junyu Shi | Min Li | Mingqi Wu |
| Qiang Gao | Shangpeng Yan | Shaotong Qi | Shu Xu | Shuo Zhou |
| Tiankuo Xu | Tong Zheng | Weilun Zhao | Xiancheng Meng | Xianda Sun |
| Xiaoyu Jiang | Xunhao Jia | Yao Xia | Yimeng Xu | Yinghan Cui |
| Yingpeng Chen | Yiwen Ning | Yong Wang | Yuxuan Sun | Zhongsheng Liu |

**Tech Leads**

| | | | | |
|---|---|---|---|---|
| Ming Sun | Cheng Luo | Chen Yang | Han Li | Kun Gai |

## References


[1] Fengxiang Li, Han Zhang, Haoyang Huang, Jinghui Wang, Jinhua Hao, Kun Yuan, Mengtong Li, Minglei Zhang, Pengcheng Xu, Wenhao Zhuang, Yizhen Shao, Zongxian Feng, Can Tang, Chao Wang, Chengxiao Tong, Fan Yang, Gang Xiong, Haixuan Gao, Han Gao, Hao Wang, Haochen Liu, Hongliang Sun, Jiabao Li, Jingwen Chang, Jun Du, Junyi Peng, Leizhen Cui, Meimei Jing, Mingqi Wu, Shangpeng Yan, Shaotong Qi, Suzhe Xu, Wenxuan Zhao, Xianda Sun, Xuan Xie, Yanbo Wang, Yao Xia, Yinghan Cui, Yingpeng Chen, Yong Wang, Yuze Shi, Zhiwei Shen, Ziyu Wang, Ming Sun, Lin Ye, and Bin Chen. Kat-coder-v2 technical report. arXiv preprint arXiv:2603.27703, 2026.

[2] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In The Twelfth International Conference on Learning Representations, 2024.

[3] Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? arXiv preprint arXiv:2509.16941, 2025.

[4] 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.

[5] Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. Swe-rebench v2: Language-agnostic swe task collection at scale. arXiv preprint arXiv:2602.23866, 2026.

[6] Jiamian Wang, Ruiyi Zhang, Tong Yu, et al. Docarena: Turning raw documents into controllable training environments for document search agents. arXiv preprint arXiv:2606.26122, 2026.

[7] Yinjie Wang, Xuyang Chen, Xiaolong Jin, et al. Openclaw-rl: Train any agent simply by talking. arXiv preprint arXiv:2603.10165, 2026.

[8] Xinze Li, Yuhang Zang, Yixin Cao, et al. Skill-as-pseudocode: Refactoring skill libraries to pseudocode for llm agents. arXiv preprint arXiv:2605.27955, 2026.

[9] Xirui Li, Ming Li, Ion Stoica, et al. Clawenvkit: Automatic environment generation for claw-like agents. arXiv preprint arXiv:2604.18543, 2026.

[10] Fei Bai, Huatong Song, Shuang Sun, Daixuan Cheng, Yike Yang, Chuan Hao, Renyuan Li, Feng Chang, Yuan Wei, Ran Tao, et al. Clawgym: A scalable framework for building effective claw agents. arXiv preprint arXiv:2604.26904, 2026.

[11] Siwei Wu, Yizhi Li, Yuyang Song, et al. Large-scale terminal agentic trajectory generation from dockerized environments. arXiv preprint arXiv:2602.01244, 2026.

[12] Bowen Ye, Rang Li, Qibin Yang, et al. Claw-eval: Towards trustworthy evaluation of autonomous agents. arXiv preprint arXiv:2604.06132, 2026.

[13] PinchBench. Pinchbench: Openclaw benchmark. `https://pinchbench.com/`, 2026.

[14] Zizheng Zhan, Ken Deng, Jinghui Wang, Xiaojiang Zhang, Huaixi Tang, Minglei Zhang, Zhiyi Lai, Haoyang Huang, Wen Xiang, Kun Wu, et al. Kat-coder technical report. arXiv preprint arXiv:2510.18779, 2025.

[15] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. arXiv preprint arXiv:2402.03300, 2024.

[16] Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. Group-in-group policy optimization for llm agent training. Advances in Neural Information Processing Systems, 38:46375–46408, 2026.

[17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347, 2017.

[18] Jiashun Liu, Johan Obando-Ceron, Han Lu, Yancheng He, Weixun Wang, Wenbo Su, Bo Zheng, Pablo Samuel Castro, Aaron Courville, and Ling Pan. Asymmetric proximal policy optimization: mini-critics boost llm reasoning. arXiv preprint arXiv:2510.01656, 2025.

[19] Lerrel Pinto, Marcin Andrychowicz, Peter Welinder, Wojciech Zaremba, and Pieter Abbeel. Asymmetric actor critic for image-based robot learning. arXiv preprint arXiv:1710.06542, 2017.

[20] Gabriel Ilharco, Marco Tulio Ribeiro, Mitchell Wortsman, Ludwig Schmidt, Hannaneh Hajishirzi, and Ali Farhadi. Editing models with task arithmetic. In International Conference on Learning Representations (ICLR), 2023.

[21] Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos Garea, Matthieu Geist, and Olivier Bachem. On-policy distillation of language models: Learning from self-generated mistakes. In International Conference on Learning Representations (ICLR), 2024.

[22] Zhicheng Yang, Zhijiang Guo, Yifan Song, Minrui Xu, Yongxin Wang, Yiwei Wang, Xiaodan Liang, and Jing Tang. Prune-opd: Efficient and reliable on-policy distillation for long-horizon reasoning. arXiv preprint arXiv:2605.07804, 2026.

Table 1 | Rule-based reward components. The rewards are organized into a primary task objective, standard behavioral constraints, and failure-path incentives.

| No. | Reward | Description |
|---|---|---|
| *Core Task Objective* | | |
| 1 | Core Task Score | Uses the outcome of task-provided unit tests as the primary optimization criterion. This reward prevents reward hacking through ineffective code changes or simplified self-authored tests, keeping resolution of the underlying code issue as the agent's highest-priority objective. |
| *Standard Behavioral Constraints* | | |
| 2 | Content Repetition Rate | Penalizes excessive redundant overlap between the `think` and `content` fields, reducing unnecessary verbosity and improving output quality. |
| 3 | Garbled Content Rate | Penalizes malformed, garbled, or illegal characters in generated text to reduce low-quality output. |
| 4 | Tool Invocation Accuracy | Encourages valid and well-formed tool calls, reducing execution failures caused by missing or incorrect parameters. |
| 5 | Invalid Tool-call Placement | Encourages the model to place tool calls in the designated output region rather than in the `think` field, preventing parsing failures. |
| 6 | Redundant Intra-turn Tool Calls | Discourages repeated invocations of the same tool within a single interaction turn, improving interaction efficiency. |
| 7 | Tool-call Parallelism | Encourages tool calls to be batched within an appropriate range, improving interaction efficiency while avoiding excessive concurrency that may destabilize training. |
| 8 | Debug Artifact Cleanup | Encourages removal of temporary reproduction scripts, validation files, logs, and cache files after task completion, preventing repository pollution and interference with subsequent execution or evaluation. |
| *Failure-Path Incentives* | | |
| 9 | File Search Accuracy | Measures whether the agent retrieves relevant repository files precisely and completely. Retrieval quality is evaluated using an $F_2$ score that jointly considers precision and recall, encouraging accurate file localization while discouraging overly broad searches. |
| 10 | Unit Test Pass Rate | Provides partial learning signals for trajectories that do not completely solve the task but make measurable progress without introducing regressions. This reward distinguishes partially useful failures from entirely unproductive trajectories. |

Table 2 | Representative trajectory-level rubric criteria. (Partial Display)

| Category | Representative Criterion | Evaluation Description |
|---|---|---|
| **Bug Reproduction** | No Bug Reproduction Attempt | The task involves defect fixing, but the trajectory performs no reproduction, test execution, or debugging step to validate the reported issue before modification. |
| | Static Bug Localization | The original failure is not dynamically reproduced, but the trajectory accurately identifies a root cause consistent with the problem statement through source inspection, existing tests, commit history, or framework-level reasoning. |
| **Post-Fix Validation** | Missing Custom Validation | After applying the fix, the agent does not construct or execute a targeted validation script for the failure scenarios explicitly described in the problem statement. |
| | Relevant Existing Tests Not Executed | Relevant repository tests exist for the reported failure scenario, but the agent does not execute them after the fix. This criterion does not apply when no relevant tests are available. |
| **Regression Testing** | Missing Broader Regression Testing | The agent validates only the directly affected failure scenario and does not run broader tests to assess whether the modification introduces regressions. |
| | Invalid Broader Test Execution | The agent attempts broader testing, but the command or script fails because of syntax, configuration, or invocation errors, leaving no valid regression evidence. |
| **Behavioral Strategy** | Complex `python -c` Execution | The agent places lengthy or multi-step Python logic in an inline `python -c` command, leading to escaping, encoding, readability, debugging, or maintenance issues. |
| | Repeatedly Incorrect Test Entry Points | The trajectory repeatedly invokes incorrect test commands, working directories, test labels, frameworks, or module paths, resulting in avoidable trial-and-error and unreliable validation. |

Table 3 | Typical bad cases. (Partial Display)

| Category | Example |
|---|---|
| **Complex code execution** | {"function": { "name": "Bash", "arguments": "{\"command\": "python -c \"\\nfrom django.core.validators import URLValidator\\nfrom django.core.exceptions import ValidationError\\n\\nvalidator =..... |
| **Tool call failed** | <tool_response>\n<tool_use_error>String to replace not found in file.\nString: return Perm(perm)\n \n @classmethod..... |
| **Special tools not used** | {"function": "name": "Bash", "arguments": {"command": "grep -r "DJANGO_SETTINGS_MODULE" /testbed/tests/*.py 2>/dev/null ..... |
| **Using non-existent modules** | {"role": "tool", "tool_call_id": "call_53d347c9917c4b36bab2d659", "content": "/opt/miniconda3/envs/testbed/bin/python: No module named pytest"} |