

KAT-Coder Technical Report

Zizheng Zhan^{*†}, Ken Deng^{*†}, Xiaojiang Zhang*, Jinghui Wang*, Huaixi Tang*, Zhiyi Lai*, Haoyang Huang*, Wen Xiang*, Kun Wu*, Wenhao Zhuang, Minglei Zhang, Shaojie Wang, Shangpeng Yan, Kepeng Lei, Zongxian Feng, Huiming Wang, Zheng Lin, Mengtong Li, Mengfei Xie, Yinghan Cui, Xuxing Chen, Chao Wang, Weihao Li, Wenqiang Zhu, Jiarong Zhang, Jingxuan Xu, Songwei Yu, Yifan Yao, Xinping Lei, Han Li, Junqi Xiong, Zuchen Gao, Dailin Li, Haimo Li, Jiaheng Liu, Yuqun Zhang, Junyi Peng, Haotian Zhang[†], Bin Chen

Kwaipilot Team

{zhanzizheng, dengken, zhanghaotian}@kuaishou.com

Abstract

Recent advances in large language models (LLMs) have enabled progress in agentic coding, where models autonomously reason, plan, and act within interactive software development workflows[1–8]. However, bridging the gap between static text-based training and dynamic real-world agentic execution remains a core challenge. In this technical report, we present KAT-Coder, a large-scale agentic code model trained through a multi-stage curriculum encompassing Mid-Term Training, Supervised Fine-Tuning (SFT), Reinforcement Fine-Tuning (RFT), and Reinforcement-to-Deployment Adaptation. The Mid-Term stage enhances reasoning, planning, and reflection capabilities through a corpus of real software engineering data and synthetic agentic interactions. The SFT stage constructs a million-sample dataset balancing twenty programming languages, ten development contexts, and ten task archetypes. The RFT stage introduces a novel multi-ground-truth reward formulation for stable and sample-efficient policy optimization. Finally, the Reinforcement-to-Deployment phase adapts the model to production-grade IDE environments using Error-Masked SFT and Tree-Structured Trajectory Training[9]. In summary, these stages enable KAT-Coder to achieve robust tool-use reliability, instruction alignment, and long-context reasoning, forming a deployable foundation for real-world intelligent coding agents. Our **KAT** series 32B model, **KAT-Dev**, has been open-sourced on 😊 <https://huggingface.co/Kwaipilot/KAT-Dev>.

^{*}Equal contribution. [†]Corresponding author.

1. Introduction

Recent advances in Large Language Models (LLMs) have catalyzed a shift from static text generation toward agentic intelligence, where models autonomously reason, plan, and act within dynamic environments. In software engineering, this transformation manifests as agentic coding—a paradigm in which models function as collaborative problem solvers rather than passive code generators. Despite rapid progress, a fundamental challenge persists: bridging the gap between static, text-based training and interactive, real-world execution. Conventional code models, typically trained on massive but inert text corpora, lack the adaptive reasoning and contextual control required to operate reliably in live integrated development environments (IDEs).

Early frameworks such as Codex, CodeLlama, and DeepSeekCoder established the foundation for code generation, yet they remain limited to single-turn, instruction-following behavior. More recent efforts, including SWE-Agent[1], OpenHands[7], and Claude Code[6], introduce planning and tool-use capabilities, signaling a broader trend toward agentic execution. However, these models are often constrained by narrow domain coverage, short reasoning horizons, and homogeneous datasets that inadequately represent real software engineering workflows. As a result, their performance deteriorates when transferred from benchmark environments to production-grade systems characterized by heterogeneous tools, long-term dependencies, and frequent context shifts.

To address these limitations, we introduce **KAT-Coder** designed to unify reasoning, planning, and deployment robustness within a single training framework. Specifically, KAT-Coder is developed through a four-stage hierarchical curriculum that progressively enhances the model’s cognitive and operational competence:

- **Mid-Term Training** — Broadens reasoning, planning, and reflection abilities through a combination of real software engineering corpora and synthetic agentic trajectories, forming a bridge between general pretraining and code-oriented supervision.
- **Supervised Fine-Tuning (SFT)** — Constructs a million-sample dataset spanning over 20 programming languages, 10 development contexts, and 10 task archetypes, ensuring balanced coverage and cross-domain generalization.
- **Reinforcement Fine-Tuning (RFT)** — Introduces a multi-ground-truth reward formulation and relative evaluation scheme for stable and sample-efficient policy optimization.
- **Reinforcement Learning (RL)** — Employs Error-Masked SFT and Tree-Structured Trajectory Training (TST) to adapt the model to production environments with heterogeneous toolchains and non-linear context boundaries.

This curriculum reflects a closed-loop design philosophy: cognitive enrichment precedes structured supervision, which in turn grounds reinforcement learning and real-world adaptation. Through this progressive alignment, KAT-Coder evolves from a general LLM into a deployable agentic developer capable of reasoning about tasks, managing tools, and collaborating in complex software workflows.

2. Mid-Term Training

The Agentic capability of a model represents a composite form of intelligence that integrates multiple dimensions—tool use, instruction following, long-context reasoning, code generation, and multi-turn dialogue. These dimensions collectively determine the model’s capacity for

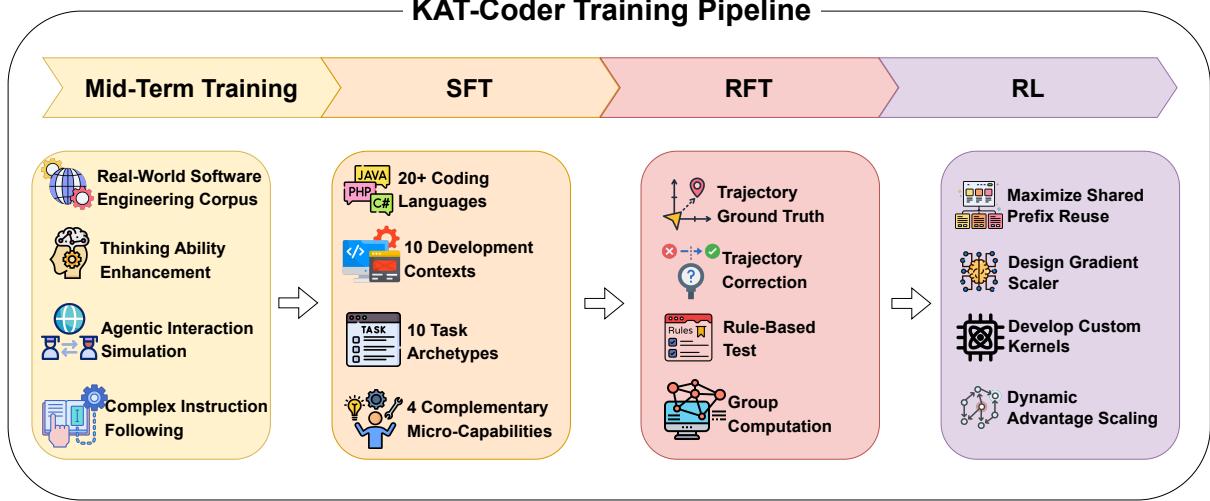


Figure 1 | KAT-Coder Training Pipeline

autonomous decision-making and adaptive interaction within real-world coding environments. To fully unlock these abilities before introducing real Agentic supervision data, we perform an extensive Mid-Term Training phase designed to broaden the model’s reasoning, planning, and interactive scope, thereby establishing a solid foundation for subsequent Code-Oriented SFT.

Training Recipe Overview Our Mid-Term Training recipe spans a diverse set of domains and task formulations, targeting both the structural and cognitive aspects of Agentic behavior. The design consists of four main components:

(1) **Real-World Software Engineering Corpus:** We collected approximately 20B tokens of real user programming data from GitHub, including pull requests, issues, commits, and corresponding code diff patches. This corpus captures authentic human–code interactions and evolution patterns across collaborative development workflows[10–17]. (2) **Reasoning and Reflection Enhancement:** To activate the model’s reasoning, planning, and reflective thinking capabilities, we utilized advanced open-source reasoning models to generate chain-of-thought trajectories for solving complex software engineering problems, competition-level STEM challenges, and logical puzzles. This promotes robust multi-domain reasoning and systematic thinking[18–22]. (3) **Agentic Interaction Simulation:** To familiarize the model with real Agentic interaction patterns, we constructed simulated environments that synthesize trajectories reflecting Plan–Action–Observation loops under diverse contexts. These trajectories train the model to dynamically adapt its plans and decisions based on environmental feedback[1–7]. (4) **Complex Instruction Following and Constraint Alignment:** We further curated instruction datasets with verifiable logical and structural constraints, enabling the model to improve its consistency, controllability, and robustness in handling complex, multi-condition instructions[23–27].

This Mid-Term Training recipe substantially strengthens the model’s foundational reasoning, reflection, and interaction capabilities, providing a crucial bridge between general pretraining and Agentic SFT. It serves as a decisive stage for expanding the upper limit of the model’s cognitive and operational competence in real-world code-oriented tasks.

Existing open-source corpora for agentic coding tasks exhibit a strong distributional bias, with the majority of samples focusing on Python-based bug-fixing activities. However, real-world programming practices extend far beyond this narrow scope, encompassing a wide range of languages, development contexts, and task archetypes. To enable robust generalization

and realistic adaptation in practical engineering scenarios, we systematically redesign the dataset along three orthogonal axes—programming languages, development contexts, and task types—ensuring balanced coverage and diversity across them.

Data Sources and Statistical Analysis Our dataset construction is grounded in large-scale mining and analysis of open-source repositories and community discussions from GitHub and Stack Overflow. By examining commit histories, code diffs, review comments, and Q&A threads, we extracted and summarized patterns of user activity and developer intent. This statistical foundation enables principled sampling and categorization across language, context, and task dimensions.

Programming Language Dimension To reflect the diversity of modern software ecosystems, we cover over twenty mainstream programming languages. Beyond high-frequency languages such as Python, Java, TypeScript, JavaScript, C, C++, C#, Kotlin, Go, Rust, PHP, and Ruby, we extend coverage to include Swift, Objective-C, Scala, R, Shell/Bash, SQL, MATLAB, Dart, Lua, Elixir, Haskell, and Perl. This broad spectrum—from scripting to systems programming—ensures generalization across heterogeneous language paradigms.

Development Context Dimension We identify ten representative development contexts through statistical analysis of real-world coding activities, encompassing the full spectrum of software engineering practices: application development, system and infrastructure development, UI/UX engineering, data science and engineering, database systems, machine learning and artificial intelligence, algorithm design and analysis, testing and debugging, system architecture and maintenance, and specialized programming domains. Balanced sampling across these contexts prevents domain overrepresentation and enhances the dataset’s robustness for cross-context generalization.

Task Type Dimension At the task level, we distill ten fundamental archetypes that capture the essential forms of software development behavior: implementation, modification and feature enhancement, debugging and bug fixing, refactoring, performance optimization, code explanation and documentation, code analysis, code generation, test case generation, and configuration and deployment.

This taxonomy spans the full development lifecycle—from problem formulation to solution deployment—capturing both cognitive and operational aspects of real-world programming.

Dataset Scale and Distribution The resulting SFT corpus comprises over one million samples, covering a rich combination of languages, contexts, and task types. Such diversity ensures balanced representation of programming practices and provides a solid supervised foundation for subsequent reinforcement fine-tuning (RFT) and reinforcement learning (RL) stages.

2.1. Reinforcement-to-Deployment Adaptation: Bridging Research Agents and Real-World Workflows

Motivation Existing datasets for code agents in the supervised fine-tuning (SFT) stage are primarily derived from research-oriented frameworks such as SWE-Agent[1], which rely on linear, single-session dialogues and homogeneous operation pipelines. While these datasets are effective for controlled academic evaluation, they fail to capture the complexity of real-world agentic environments.

In practical software engineering, agents must operate within heterogeneous toolchains, dynamically manage long-horizon dependencies, and adapt to non-linear conversational trajectories involving frequent context switches and multi-turn reasoning. This gap between research

benchmarks and production-grade workflows leads to a significant distribution mismatch, limiting the generalization ability of agentic code models when deployed in real development systems.

Data Construction across Production Environments To bridge this research–deployment gap, we construct a new generation of Agentic Workflow training data by integrating our early-stage KAT-Coder models with production-grade IDE-based systems such as Claude Code[6], Cline[5], Roo Code[4], and CodeFlicker[8]. These environments offer realistic execution traces, tool invocations, and iterative human–agent interactions, enabling the construction of data that reflects true software development dynamics.

This integration yields trajectories that are far more diverse and realistic but also introduces new training challenges.

Training Challenges in Production-Grade Trajectories Production workflows differ substantially from benchmark settings in two major aspects: (1) Expanded Tool Spectrum — Real-world agents interact with dozens of heterogeneous tools (e.g., debuggers, linters, package managers), leading to frequent erroneous or redundant tool calls. (2) Non-Linear Context Boundaries — Compression checkpoints, context truncation, and mode switches (e.g., between coding, planning, and execution) introduce branching points that disrupt the continuity of dependency chains. These challenges make direct imitation learning unstable, as gradient propagation from noisy tool calls or broken trajectories can degrade convergence and overfit to spurious behaviors.

Methodology: Error-Masked SFT and Tree-Structured Trajectory Training To address these issues, we adopt two complementary strategies designed for agentic fine-tuning under complex tool and context dynamics: (1) Error-Masked SFT (EM-SFT) We leverage execution feedback logs to identify tool-use failures and selectively mask gradients from erroneous tool calls. This prevents error amplification during backpropagation while retaining the model’s exposure to self-corrective reasoning signals. (2) Tree-Structured Trajectory Training (TST) We decompose multi-branch trajectories into locally coherent subtrees defined by context compression boundaries and mode transitions. Within each subtree, standard supervised fine-tuning is performed independently, ensuring stable optimization and improved temporal consistency. Together, these strategies enable the policy model to learn from realistic, production-grade trajectories without sacrificing training stability or semantic coherence. This forms the foundation for aligning the model’s behavior with human engineering workflows—an essential step toward fully deployable agentic coding systems.

3. RFT

During the reinforcement learning (RL) phase, we introduce multiple trajectory ground truths as reference signals to enhance rollout efficiency and training stability. Traditional RL approaches typically rely on an absolute reward computed directly from the model’s output, which is highly sensitive to the reward scale and often leads to unstable optimization or inefficient sample utilization.

To address this, we propose a relative evaluation framework, where the model is optimized based on the discrepancy between generated samples and ground-truth trajectories rather than absolute reward magnitudes. This transformation stabilizes the training process and significantly improves sampling efficiency.

As illustrated in Figure 2, given an input query q , the policy model π_θ produces a sequence of trajectory outputs o_1, o_2, \dots, o_n . A set of trajectory ground truths is then used for **Trajectory**

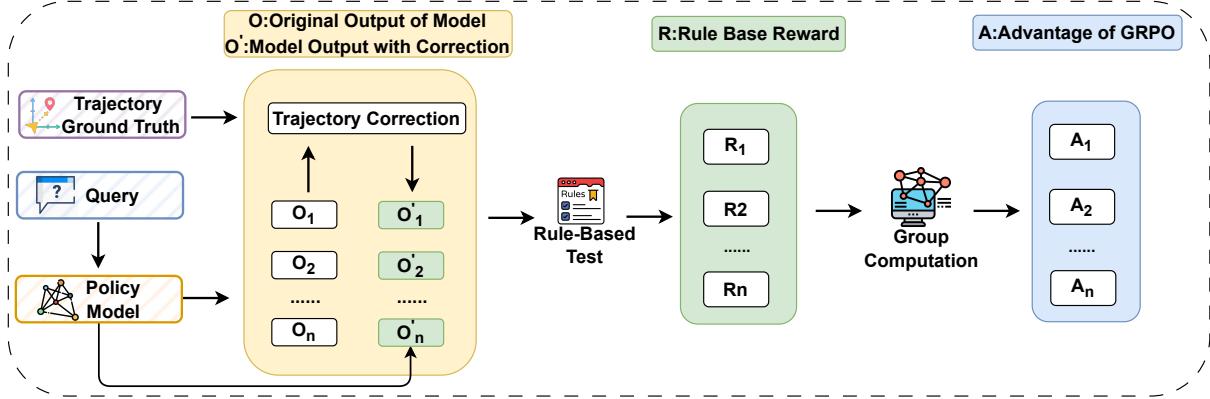


Figure 2 | Method overview of RFT. (1) Trajectory Ground Truth: provides evaluation and correction signals for generated trajectories. (2) Trajectory Correction: performs online correction for outputs deviating from reference trajectories. (3) Rule-Based Test: generates stable and interpretable rule-based reward signals. (4) Group Computation: normalizes and aggregates group-level advantages within the GRPO framework.

Correction, generating corrected outputs o'_i . These are evaluated through a **Rule Base Test**, which yields rule-based rewards r_i . Subsequently, the **Group Computation** stage aggregates and normalizes the sample groups under the GRPO [28] framework to produce final advantages A_i for policy gradient updates. This hierarchical correction and grouping mechanism enables the policy to converge towards semantically correct and structurally consistent generation behaviors.

Training Stability and Sample Efficiency By replacing absolute reward computation with relative discrepancy estimation, the method effectively mitigates instability caused by model fluctuations or reward scale drift. Additionally, early termination and resampling mechanisms for trajectories that deviate significantly from ground truths improve overall sample utilization and rollout efficiency.

Empirical Insights This design yields substantial improvements during RL training, including: (1) More stable reward signals, reducing scale drift and enhancing convergence. (2) Higher sample efficiency, by filtering invalid rollouts through correction and resampling. (3) Better semantic alignment, as generated trajectories align more closely with human-validated ground truths.

4. Agentic RL

4.1. Trie Packed Training

In agentic LLM training, an agent's behavior is often highly diverse, so its trajectory history is usually not a linear trajectory like prompt1 → response1 → prompt2 → response2. Instead, a single task can produce multiple trajectories, many of which share common prefixes. As shown in figure 3, these trajectories can be naturally organized into a prefix tree (Trie).

In this scenario, we realized that the computation of prefixes shared across multiple trajectories could be performed just once, which would significantly improve training throughput. This idea is similar to the prefix caching used for LLM inference. However, due to the involvement of back propagation during training, we cannot directly reuse cached results; doing so would

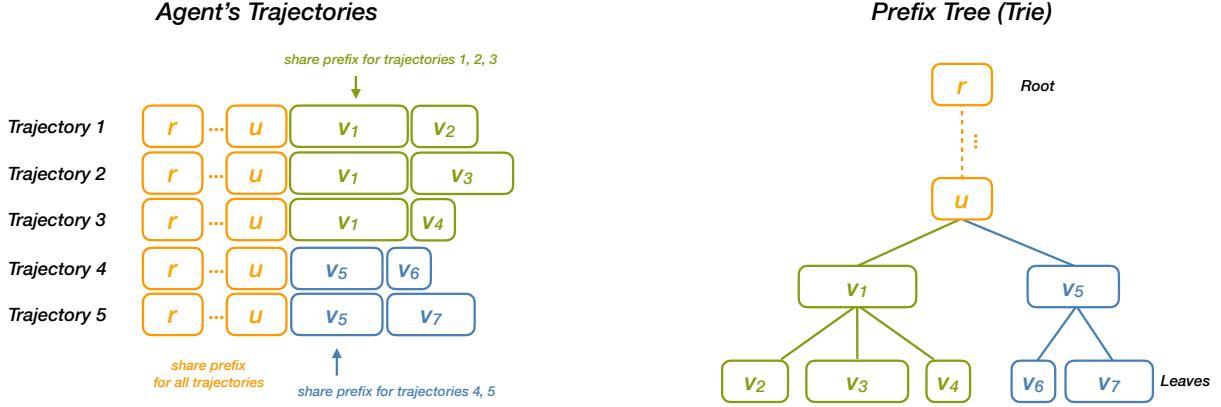


Figure 3 | Overview of Trie-Packed Training in agentic RL.

ignore the gradient contributions from suffix tokens to the prefix tokens, leading to incorrect computations. Our goal is computing each shared prefix only once during both forward and backward and thus improve agentic LLM training throughput. To achieve this, we took the following steps:

- (1) Maximize shared prefix reuse through Trie Packing: Since there are many training trajectories, even after merging them into a tree structure, we usually can't fit the entire tree into one batch due to GPU memory constraints. Using a combination of dynamic programming and greedy algorithms, we proposed a practical approach to pack the tree under memory constraints which maximizes the reuse of shared prefixes.
- (2) Design Gradient Scaler to ensure correct gradient computation: During back propagation, the gradient contribution of a shared prefix differs across different trajectories. We implemented a tree-structured gradient scaler that mathematically ensures each shared prefix contributes correctly to the gradients.
- (3) Develop custom kernels: We implemented an efficient shared-prefix mask attention and modified position embedding for flattened trie pack data, ensuring both training correctness and high throughput.

4.2. Enhancing Exploration via Difficulty- and Entropy-Aware Advantage Rescaling

In policy gradient reinforcement learning, the advantage function $A(s, a)$ determines each sample's contribution to the update, which can be computed as:

$$\theta \leftarrow \theta + \eta \nabla_{\theta} \log \pi_{\theta}(a|s) A(s, a)$$

Larger advantages increase a sample's influence, but standard GRPO tends to assign the largest advantages to medium-difficulty tasks, limiting exploration on very easy or hard tasks and potentially causing entropy collapse. To address this, we propose a difficulty- and entropy-aware advantage scaling method that adjusts group-level advantages according to task difficulty. For the i -th task group, difficulty is defined based on the average success rate:

$$D_i = 1 - r_i$$

where r_i denotes the group's average success rate. Higher D_i indicates tasks that are not yet mastered and require stronger optimization, while lower D_i corresponds to easier tasks with

reduced emphasis. The group-level scaling factor is defined as:

$$\alpha_i^{\text{group}} = 1 + \lambda(D_i - \bar{D})$$

where \bar{D} is the average difficulty of the current batch and $\lambda > 0$ controls the magnitude of scaling.

Within each group, we further adjust the advantage based on the policy entropy of each sample. High-entropy samples indicate greater uncertainty in the model's action selection and thus higher exploratory value, whereas low-entropy samples correspond to more certain decisions. The sample-level scaling factor is defined as:

$$\beta_{ij}^{\text{sample}} = 1 + \mu(H_{ij} - \bar{H}_i)$$

where H_{ij} is the policy entropy of sample j in group i , \bar{H}_i is the group's average entropy, and $\mu > 0$ controls the scaling strength.

Finally, the difficulty- and entropy-aware scaled advantage for each sample is given by:

$$A'_{ij} = \alpha_i^{\text{group}} \beta_{ij}^{\text{sample}} A_{ij}$$

This method dynamically allocates training resources to amplify high-difficulty tasks and emphasize high-entropy samples, maintaining policy diversity, preventing entropy collapse, and enhancing exploration, robustness, and generalization.

5. Model Evaluation and Comparative Analysis

We comprehensively evaluate KAT-Coder across diverse benchmarks covering instruction following (IFEval[29]), tool invocation (TAU2-Bench Retail[30]), mathematical reasoning (AIME 2025), code generation (LiveCodeBench V6[31], HumanEval[32]), general knowledge (GPQA-Diamond[33]), and agentic coding (SWE-Bench-Verified[34]). The results, summarized in Table 1, are compared with leading contemporary large models including Qwen3-Coder-480B[35, 36], Kimi-k2-0905[37], and Claude 4 Sonnet[38].

Benchmark	KAT-Coder	Qwen3-Coder-480B	Kimi-k2-0905	Claude 4 Sonnet
IFEval	86.0	84.8	89.3	88.2
TAU2-Bench Retail	62.3	57.9	56.5	64.2
AIME 2025	72.5	44.3	49.5	70.5
LiveCodeBench V6	48.2	48.2	48.9	46.5
HumanEval	96.3	95.1	88.4	98.2
GPQA-Diamond	68.2	60.6	70.2	68.7
SWE-Bench-Verified	73.4	69.6	65.8	72.7

Table 1 | Benchmark results comparing KAT-Coder with contemporary large models.

Across a broad range of evaluation tracks, KAT-Coder demonstrates consistently strong and balanced performance. It achieves competitive results in instruction following, tool invocation, mathematical reasoning, code generation, and general knowledge, matching or surpassing leading proprietary and open-source baselines such as Qwen3-Coder-480B and Claude 4 Sonnet. In particular, KAT-Coder attains 73.4 on SWE-Bench Verified when evaluated under Claude Code.

These results validate the effectiveness of our four-stage training pipeline (Mid-Term Training → SFT → RFT → RL). Each phase contributes distinct improvements—Mid-Term broadens

reasoning depth, SFT enhances cross-language generalization, RFT stabilizes policy learning through relative rewards, and RL with Trie-Packed Training yields high-efficiency multi-trajectory optimization. The overall performance indicates that KAT-Coder not only matches proprietary systems on general reasoning and coding tasks but also demonstrates superior adaptability in authentic agentic coding scenarios.

6. Conclusion

In this report, we have detailed the design, training, and adaptation pipeline of KAT-Coder, a large-scale agentic code model developed to bridge the divide between research-grade models and deployable coding agents.

Through a structured four-stage process—Mid-Term Training, SFT, RFT, and Reinforcement-to-Deployment Adaptation—KAT-Coder achieves consistent improvements in reasoning stability, tool reliability, and contextual understanding. The integration of real software artifacts, synthetic reasoning traces, and production-grade agentic trajectories allows the model to operate effectively in complex, real-world development workflows. Our findings demonstrate that agentic capability does not emerge from a single phase of optimization, but rather through the cumulative interaction between data diversity, reasoning supervision, and reinforcement alignment. The success of KAT-Coder highlights the importance of mid-term reasoning enrichment, structured multi-dimensional datasets, and robust RL adaptation in achieving deployable intelligence.

Looking ahead, future work will explore multi-modal agentic collaboration (e.g., code execution, GUI manipulation, and document editing), long-horizon memory persistence, and hierarchical planning architectures that allow models like KAT-Coder to serve as fully autonomous software collaborators in complex engineering environments.

References

- [1] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024.
- [2] Cursor. Cursor - the ai code editor. <https://cursor.com/>, 2025.
- [3] OpenAI. codex. <https://github.com/openai/codex>, 2025.
- [4] RooCode Inc. Roo-code. <https://github.com/RooCodeInc/Roo-Code>, 2025.
- [5] Cline. cline. <https://github.com/cline/cline>, 2025.
- [6] Anthropic. claude-code. <https://github.com/anthropics/claude-code>, 2025.
- [7] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for AI software developers as generalist agents. In The Thirteenth International Conference on Learning Representations, 2025.
- [8] Kuaishou Team. Codeflicker. <https://www.codeflicker.ai/>, 2025.
- [9] Shaojie Wang, Jinghui Wang, Yinghan Cui, Xuxing Chen, Chao Wang, and Xiaojiang Zhang. Packingtree, 2025.
- [10] Zhenyu He, Qingping Yang, Wei Sheng, Xiaojian Zhong, Kechi Zhang, Chenxin An, Wenlei Shi, Tianle Cai, Di He, Jiaze Chen, and Jingjing Xu. Swe-swiss: A multi-task fine-tuning and rl recipe for high-performance issue resolution. <https://www.notion.so/SWE-Swiss-A-Multi-Task-Fine-Tuning-and-RL-Recipe-for-High-Performance-Issue-Resolution-21e174dedd4880ea829ed4c861c44f88>, 2025. Notion Blog.
- [11] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models. arXiv preprint arXiv:2308.07124, 2023.
- [12] Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents, 2025.
- [13] Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym, 2025.
- [14] 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, 2025.
- [15] Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. Swe-bench goes live!, 2025.

- [16] Leon Guertler, Bobby Cheng, Simon Yu, Bo Liu, Leshem Choshen, and Cheston Tan. Textarena, 2025.
- [17] Jiajun Shi, Jian Yang, Jiaheng Liu, Xingyuan Bu, Jiangjie Chen, Junting Zhou, Kaijing Ma, Zhoufutu Wen, Bingli Wang, Yancheng He, Liang Song, Hualei Zhu, Shilong Li, Xingjian Wang, Wei Zhang, Ruibin Yuan, Yifan Yao, Wenjun Yang, Yunli Wang, Siyuan Fang, Siyu Yuan, Qianyu He, Xiangru Tang, Yingshui Tan, Wangchunshu Zhou, Zhaoxiang Zhang, Zhoujun Li, Wenhao Huang, and Ge Zhang. Korgym: A dynamic game platform for llm reasoning evaluation, 2025.
- [18] Xiaoyu Tian, Yunjie Ji, Haotian Wang, Shuaiting Chen, Sitong Zhao, Yiping Peng, Han Zhao, and Xiangang Li. Not all correct answers are equal: Why your distillation source matters, 2025.
- [19] Intelligent Internet. Ii-thought : A large-scale, high-quality reasoning dataset, 2025.
- [20] Yang Chen, Zhuolin Yang, Zihan Liu, Chankyu Lee, Peng Xu, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. Acereason-nemotron: Advancing math and code reasoning through reinforcement learning, 2025.
- [21] Jujie He, Jiacai Liu, Chris Yuhao Liu, Rui Yan, Chaojie Wang, Peng Cheng, Xiaoyu Zhang, Fuxiang Zhang, Jiacheng Xu, Wei Shen, Siyuan Li, Liang Zeng, Tianwen Wei, Cheng Cheng, Bo An, Yang Liu, and Yahui Zhou. Skywork open reasoner 1 technical report, 2025.
- [22] Ken Deng, Zizheng Zhan, Wen Xiang, Wenqiang Zhu, Tianhao Peng, Xinping Lei, Weihao Li, Jingxuan Xu, Kun Wu, Yifan Yao, Haoyang Huang, Huaixi Tang, Kepeng Lei, Zhiyi Lai, Songwei Yu, Zongxian Feng, Zuchen Gao, Weihao Xie, Chenchen Zhang, Yanan Wu, Yuanxing Zhang, Lecheng Huang, Yuqun Zhang, Jie Liu, Zhaoxiang Zhang, Haotian Zhang, Bin Chen, and Jiaheng Liu. Hipo: Hybrid policy optimization for dynamic reasoning in llms, 2025.
- [23] Jian Yang, Wei Zhang, Shukai Liu, Linzheng Chai, Yingshui Tan, Jiaheng Liu, Ge Zhang, Wangchunshu Zhou, Guanglin Niu, Zhoujun Li, Binyuan Hui, and Junyang Lin. Ifevalcode: Controlled code generation, 2025.
- [24] Qinyan Zhang, Xinping Lei, Ruijie Miao, Yu Fu, Haojie Fan, Le Chang, Jiafan Hou, Dingling Zhang, Zhongfei Hou, Ziqiang Yang, Changxin Pu, Fei Hu, Jingkai Liu, Mengyun Liu, Yang Liu, Xiang Gao, Jiaheng Liu, Tong Yang, Zaiyuan Wang, Ge Zhang, and Wenhao Huang. Inverse ifeval: Can llms unlearn stubborn training conventions to follow real instructions?, 2025.
- [25] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022.
- [26] Zhengyan Shi, Adam X. Yang, Bin Wu, Laurence Aitchison, Emine Yilmaz, and Aldo Lipani. Instruction tuning with loss over instructions, 2024.
- [27] Peiding Wang, Li Zhang, Fang Liu, Lin Shi, Minxiao Li, Bo Shen, and An Fu. Codeif-bench: Evaluating instruction-following capabilities of large language models in interactive code generation, 2025.

- [28] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024.
- [29] Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. Instruction-following evaluation for large language models, 2023.
- [30] Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2 -bench: Evaluating conversational agents in a dual-control environment, 2025.
- [31] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024.
- [32] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021.
- [33] David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. Gpqa: A graduate-level google-proof q&a benchmark, 2023.
- [34] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024.
- [35] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengan Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yingqi Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025.
- [36] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2. 5-coder technical report. [arXiv preprint arXiv:2409.12186](#), 2024.
- [37] Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, Yichen Feng, Kelin Fu,

Bofei Gao, Hongcheng Gao, Peizhong Gao, Tong Gao, Xinran Gu, Longyu Guan, Haiqing Guo, Jianhang Guo, Hao Hu, Xiaoru Hao, Tianhong He, Weiran He, Wenyang He, Chao Hong, Yangyang Hu, Zhenxing Hu, Weixiao Huang, Zhiqi Huang, Zihao Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yongsheng Kang, Guokun Lai, Cheng Li, Fang Li, Haoyang Li, Ming Li, Wentao Li, Yanhao Li, Yiwei Li, Zhaowei Li, Zheming Li, Hongzhan Lin, Xiaohan Lin, Zongyu Lin, Chengyin Liu, Chenyu Liu, Hongzhang Liu, Jingyuan Liu, Junqi Liu, Liang Liu, Shaowei Liu, T. Y. Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yibo Liu, Yiping Liu, Yue Liu, Zhengying Liu, Enzhe Lu, Lijun Lu, Shengling Ma, Xinyu Ma, Yingwei Ma, Shaoguang Mao, Jie Mei, Xin Men, Yibo Miao, Siyuan Pan, Yebo Peng, Ruoyu Qin, Bowen Qu, Zeyu Shang, Lidong Shi, Shengyuan Shi, Feifan Song, Jianlin Su, Zhengyuan Su, Xinjie Sun, Flood Sung, Heyi Tang, Jiawen Tao, Qifeng Teng, Chensi Wang, Dinglu Wang, Feng Wang, Haiming Wang, Jianzhou Wang, Jiaxing Wang, Jinhong Wang, Shengjie Wang, Shuyi Wang, Yao Wang, Yejie Wang, Yiqin Wang, Yuxin Wang, Yuzhi Wang, Zhaoji Wang, Zhengtao Wang, Zhexu Wang, Chu Wei, Qianqian Wei, Wenhai Wu, Xingzhe Wu, Yuxin Wu, Chenjun Xiao, Xiaotong Xie, Weimin Xiong, Boyu Xu, Jing Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ziyao Xu, Junjie Yan, Yuzi Yan, Xiaofei Yang, Ying Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Xingcheng Yao, Wenjie Ye, Zhuorui Ye, Bohong Yin, Longhui Yu, Enming Yuan, Hongbang Yuan, Mengjie Yuan, Haobing Zhan, Dehao Zhang, Hao Zhang, Wanlu Zhang, Xiaobin Zhang, Yangkun Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Haotian Zhao, Yikai Zhao, Huabin Zheng, Shaojie Zheng, Jianren Zhou, Xinyu Zhou, Zaida Zhou, Zhen Zhu, Weiyu Zhuang, and Xinxing Zu. Kimi k2: Open agentic intelligence, 2025.

[38] Anthropic. Claude 4. <https://www.anthropic.com/news/claude-4>, 2025.