Qwen3-Coder-Next Technical Report

Qwen Team

https://huggingface.co/Qwen/Qwen3-Coder-Next
https://www.modelscope.cn/models/Qwen/Qwen3-Coder-Next
https://github.com/QwenLM/Qwen3-Coder

Abstract

We present Qwen3-Coder-Next, an open-weight language model specialized for coding agents. Qwen3-Coder-Next is an 80-billion-parameter model that activates only 3 billion parameters during inference, enabling strong coding capability with efficient inference. In this work, we explore how far strong training recipes can push the capability limits of models with small parameter footprints. To achieve this, we perform agentic training through large-scale synthesis of verifiable coding tasks paired with executable environments, allowing learning directly from environment feedback via mid-training and reinforcement learning. Across agent-centric benchmarks including SWE-Bench and Terminal-Bench, Qwen3-Coder-Next achieves competitive performance relative to its active parameter count. We release both base and instruction-tuned open-weight versions to support research and real-world coding agent development.

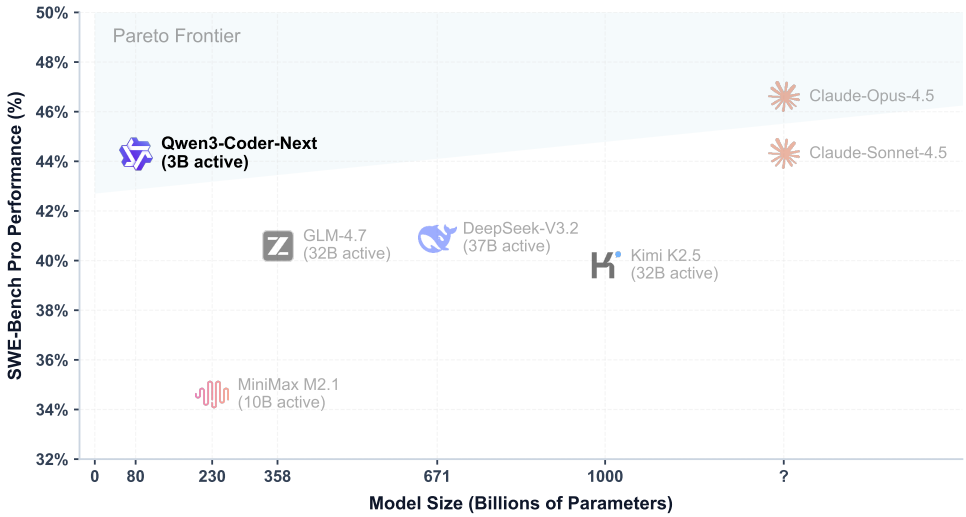

Figure 1: SWE-Bench Pro performance versus model size. The figure shows the trade-off between software engineering capability and compute across models.

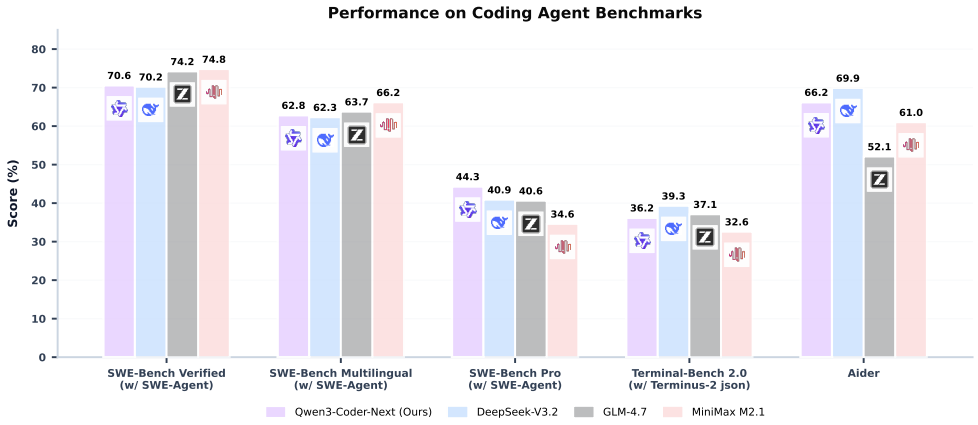

Figure 2: Comparison of Qwen3-Coder-Next to other open-weight models on SWE-Bench Verified, SWE-Bench Multilingual, SWE-Bench Pro, Terminal-Bench 2.0, and Aider.

1 Introduction

We introduce Qwen3-Coder-Next, an open-weight language model based on Qwen3-Next with hybrid attention and Mixture-of-Experts (MoE) designed specifically for coding agents and local development. It contains 80 billion total parameters while activating only 3 billion per forward pass, enabling fast inference and low deployment cost. Despite its lightweight active footprint, Qwen3-Coder-Next delivers strong performance across a wide range of coding benchmarks and real-world developer workflows.

The central advancement behind Qwen3-Coder-Next is the ability to scale agentic training. Modern coding agents must reason over long horizons, interact with real execution environments, and recover from cascading failures across multiple steps. Training for this regime requires more than static code data, which requires large volumes of verifiable, executable, and interaction-rich training signals (Copet et al., 2025; Zhang et al., 2025b). To address this, we build a large-scale agentic training stack that synthesizes executable tasks, constructs reproducible environments, and learns directly from execution feedback. This stack enables reliable large-scale rollout collection and learning from real environment outcomes, supporting both mid-training adaptation and reinforcement learning for agent behaviors such as multi-step code editing, tool usage, and fault recovery in realistic development settings.

We train Qwen3-Coder-Next using a staged pipeline that progressively builds agentic capabilities without sacrificing base model generality. Starting from the pretrained base of Qwen3-Next, we first shift representations toward code- and agent-centric domains via continued pretraining. Next, we apply supervised fine-tuning on high-quality agentic coding data. From this checkpoint, we specialize multiple expert models, covering software engineering workflows, QA, web development, and user experience (UX) focused coding, and then distill their capabilities back into a single unified model. The result is an efficient, deployable model that delivers expert-level agentic performance across domains.

Extensive evaluation shows that Qwen3-Coder-Next achieves strong performance relative to its active parameter footprint. As illustrated in Figure 1, Qwen3-Coder-Next reaches competitive SWE-Bench Pro (Deng et al., 2025) performance, outperforming or matching several models with an order of magnitude larger active compute. Beyond SWE-Bench Pro, we evaluate Qwen3-Coder-Next on an extensive suite of benchmarks spanning coding agents, general coding tasks, and general knowledge and reasoning tasks, with detailed results reported in Section 5. This efficiency makes Qwen3-Coder-Next particularly well-suited for production coding agents, where latency, throughput, and cost are first-order constraints. More broadly, these results suggest that scaling agentic training, rather than model size alone, is a key driver for advancing real-world coding agent capability.

2 Scaling up Agentic Training

Scaling agentic training to large volumes, across both mid-training and reinforcement learning, requires addressing two core challenges. First, we need a reliable pipeline for synthesizing verifiable tasks paired with fully executable environments. Second, we need an execution infrastructure capable of running a massive number of such tasks with high throughput and returning environment feedback efficiently. In this section, we describe our approach to each of these challenges, with a focus on task synthesis at scale.

2.1 Task Synthesis

We develop two complementary approaches for generating verifiable tasks with executable environments. The first approach grounds tasks in real-world software engineering issues by mining GitHub pull requests (PRs) and constructing corresponding runnable environments. The second approach starts from existing open-source datasets that already provide executable environments and synthesizes new task instances within them. Together, these methods enable large-scale task diversity with consistent execution-based validation.

Creating Executable Environments from GitHub PRs. To synthesize realistic software engineering tasks, we mine issue-related PRs and construct executable environments that reflect real-world bug-fixing tasks. After removing instances that overlap with downstream benchmarks, we decompose each PR into a buggy state, a corresponding fix, and an associated test patch. A specialized environment-building agent then constructs a runnable Docker environment and verification script, which is required to reliably distinguish the buggy and fixed states through execution. This process produces a large collection of verifiable software engineering tasks grounded in real repositories.

Environment construction poses significant challenges for agents, leading to failure modes where agents exploit superficial verification shortcuts. To mitigate this, we apply automated detection to identify and filter non-functional verifiers, and train a dedicated model to improve environment construction quality.

We apply this model at scale to recent GitHub data to generate a large corpus of verifiable software engineering tasks, with all environments stored as reusable Docker images. To ensure dataset quality, we further employ a quality-assurance agent to automatically identify and remove ambiguous tasks, inconsistent environments, and misaligned tests before final inclusion. Further technical details can be found in Chen et al. (2026).

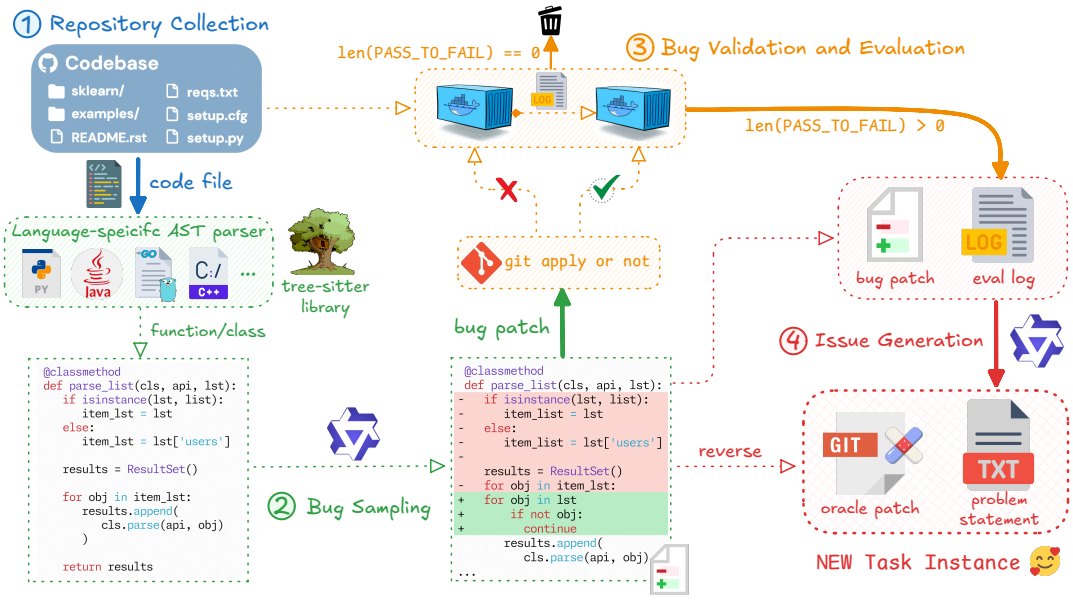

Figure 3: Our pipeline for synthesizing bugs to scale up the number of software engineering tasks.

Synthesizing Issues. In parallel, we synthesize additional software engineering tasks by building on prior work, including SWE-Smith (Yang et al., 2025), SWE-Flow (Zhang et al., 2025c), SWE-Rebench (Badertdinov et al., 2025), and Multi-SWE-RL (Zan et al., 2025). These projects provide a strong foundation of seed tasks with executable repositories, test suites, and evaluation scripts. We extend these datasets to generate a substantially larger and more diverse set of verifiable software engineering problems. Our pipeline is shown in Figure 3.

Our synthesis pipeline systematically introduces controlled bugs into existing codebases and produces matching issue descriptions. Building on curated, containerized repositories, we inject bugs via model-driven rewriting, semantic perturbations, and rule-based transformations, generalizing prior techniques to multilingual codebases. We retain generated bugs only when they fail existing tests and are resolved by patch reversion, guaranteeing that each task is both meaningful and tractable. To mitigate shortcut learning, we generate natural-language issue descriptions and exclude bug-triggering test files. This process yields approximately 800K verifiable software engineering task instances spanning over nine programming languages.

2.2 Infrastructure

Scaling agentic coding requires large-scale parallel execution and fully reproducible execution environments. To this end, we develop MegaFlow (Zhang et al., 2026c), an internal orchestration system. Our system adopts a fully cloud-native execution framework based on *Alibaba Cloud Kubernetes*[1], enabling production-scale training, evaluation, and data generation for agentic coding workloads. In MegaFlow, each agentic coding task is expressed as an *Argo workflow* composed of three logical stages: *agent rollout*, *evaluation*, and *post-processing*. During the rollout stage, a single pod typically co-locates the agent container with the execution environment container (and additional auxiliary services when needed), enabling efficient long-horizon interaction with minimal communication overhead. The evaluation stage performs automated verification in a dedicated container, while the post-processing stage handles result interpretation, including parsing outcomes, extracting metrics, and optionally downstream analysis.

[1] https://www.alibabacloud.com/en/product/kubernetes

3 Mid-training

We now describe the mid-training stage used to specialize Qwen3-Coder-Next for coding and agentic tasks. Starting from the pretrained Qwen3-Next base model (Qwen Team, 2025a), we perform targeted mid-training to adapt the model toward code reasoning, repository-level understanding, and agent-style interaction patterns.

3.1 Data

The guiding principle for data selection in mid-training is to balance natural and synthetic data. Natural data improves the model's general intelligence and robustness, but it does not fully match the distribution of tasks and interaction patterns observed in real user workflows. In contrast, heavy reliance on synthetic data can significantly improve performance on targeted tasks, but may lead to over-specialization, reduced response diversity, and weaker adaptation to other tasks during fine-tuning.

Therefore, our goal is to introduce the minimum amount of synthetic data required for the model to reliably perform common user tasks, while preserving response diversity and maintaining strong general-purpose capabilities. Following this principle, the mid-training corpus is composed primarily of natural data, supplemented with a smaller but carefully designed portion of synthetic data. We describe the key components below.

3.1.1 Natural Data

Our sources of natural data include large-scale source code from GitHub and text–code grounding data derived from Common Crawl and targeted web domains. The pretraining corpus is updated through Sep 30, 2025.

GitHub. Compared to the Qwen2.5-Coder series (Hui et al., 2024), we significantly expand coverage by increasing programming language support from 92 to 370 languages. We also incorporate substantially more pull requests, repositories, and code review data, and restructure these data sources to better reflect real development workflows.

We first include file-level pretraining data, which enables the model to learn strong file-level structural understanding. However, real-world software development rarely operates at the single-file level. Therefore, we place additional emphasis on repository-level code, allowing the model to learn cross-file dependencies and broader contextual relationships. To support this shift, we expand the training context length from 32,768 tokens to 262,144 tokens. Consistent with Qwen2.5-Coder, we use special tokens to concatenate repository data. In addition, we experiment with multiple repository serialization formats to improve generalization across different project layouts. In this release, repository-level data is expanded to approximately 600B tokens, representing a major portion of the mid-training recipe and proving more impactful than file-level datasets alone.

Text–code Grounding Data. Text–code grounding data is collected from Common Crawl and domain-targeted sources such as math, programming, and education. We note that natural web data varies significantly in quality. Low-quality web content may contain incorrect information, insufficient context, or excessive code-switching between languages and formats. To mitigate these issues, we prompt Qwen3-Coder-480B-A35B-Instruct to rewrite web documents into normalized, structured text. The rewriting process removes advertisements, irrelevant HTML elements, and formatting artifacts, producing clean Markdown-style documents suitable for training. We empirically evaluate the impact of reformatting during mid-training. As shown in Table 1, reformatting substantially improves our model across multiple evaluation benchmarks.

Model	Evalplus	MultiplE	CRUX-Eval
Baseline	54.38	36.02	57.13
Reformat	63.09	48.35	58.94

Table 1: Impact of web document reformatting during mid-training.

GitHub Pull Requests (PRs). We construct PR-based training data by mining real-world GitHub pull requests and converting them into structured software engineering tasks. Each instance consists of a natural-language problem description, repository-level code context, and corresponding code edits. Problem descriptions are sourced from linked issues when available, or from PR titles and descriptions

otherwise. Code context is reconstructed by reverting the PR patch and retrieving additional relevant files from the repository, intentionally introducing realistic mixtures of signal and noise. Edits are represented using both Search-and-Replace and standard `git diff` formats to support diverse editing paradigms. We then apply filtering and benchmark decontamination, removing anomalous files and instances overlapping with downstream tasks. This formulation encourages the model to localize bugs and produce precise code edits grounded in natural language descriptions.

3.1.2 Synthetic Data

Synthetic data is used to better align the model with real-world user workflows. We divide these tasks into two categories: 1) single-turn query, and 2) multi-turn agentic coding, where users interact with executable environments to complete tasks.

Single-turn QA. To improve single-turn QA capability, we use Common Crawl documents as seed data and prompt Qwen3-Coder-480B-A35B-Instruct to generate multiple grounded question–answer pairs per document. Generated questions must be self-contained and progressively increase in semantic depth or reasoning complexity. When documents lack sufficient quality or coherence, the model is allowed to abstain from generating QA pairs, reducing hallucination risk.

We also experimented with rewriting documents into alternative formats such as Wikipedia-style pages. However, this sometimes introduced hallucinated references or URLs. To avoid reinforcing these behaviors, we restrict rewriting to transformations that preserve original document content.

Multi-turn Agentic Coding. For multi-turn agentic data, we leverage synthetic tasks described in Section 2.1. Trajectories are generated using multiple agent frameworks, including SWE-agent (Yang et al., 2024), Mini-SWE-agent (SWE-agent Team, 2025), OpenHands (Wang et al., 2024a), Claude-Code (Anthropic, 2024), Qwen-Code (Qwen Team, 2025b), and Terminus (Merrill et al., 2026). We use Qwen3-Coder-480B-A35B-Instruct as the teacher model.

After generation, trajectories undergo strict rule-based filtering, including removal of missing termination signals, task failures, malformed tool calls. This produces a large-scale dataset of high-quality multi-turn tool-calling trajectories.

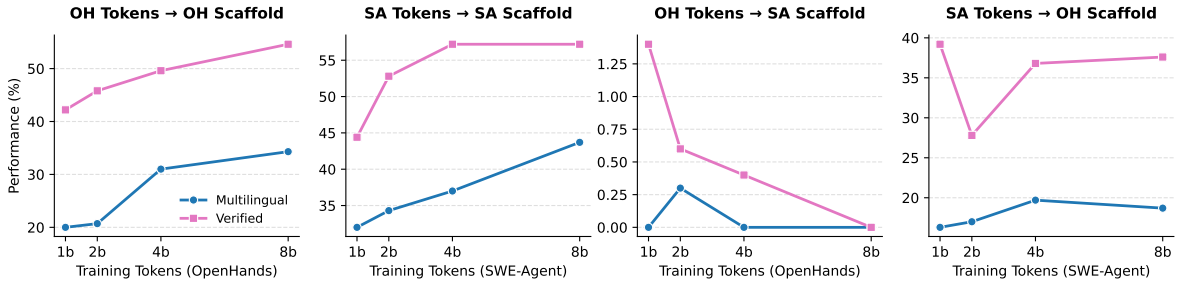

Figure 4: Scaling analysis of code agent pretraining on SWE tasks, SWE-Bench Verified and SWE-Bench Multilingual. **Left:** Within-scaffold scaling shows consistent improvement with the increase of mid-training tokens. **Right:** Cross-scaffold transfer reveals limited generalization across different agent frameworks.

To study scaling behavior, we analyze the relationship between mid-training token volume and downstream performance across agent scaffolds. As shown in Figure 4, we observe that within the same scaffold, performance consistently improves with increased mid-training tokens, demonstrating the effectiveness of large-scale agentic pretraining. Second, cross-scaffold transfer remains limited. Models trained on trajectories from one scaffold do not transfer strongly to others. Third, framework specialization plays an important role. For example, OpenHands, which is highly specialized for SWE tasks, transfers poorly to SWE-Agent, while transfer in the opposite direction is moderately successful. This highlights a trade-off between framework generality and specialization.

3.1.3 Instruction-Following Data

Because mid-training is dominated by natural documents, instruction-following behavior may not emerge reliably during mid-training alone. Therefore, we mix a small amount of instruction-following data into mid-training to enable early monitoring of downstream task performance during mid-training.

3.1.4 Fill-In-the-Middle Code Completion

Qwen3-Coder-Next supports fill-in-the-middle (FIM) code completion, which improves tasks such as document editing and online code modification (Chen et al., 2024). Using Stack-V2 (Lozhkov et al., 2024), we synthesize FIM data in two formats: 1) chat-FIM, which embeds FIM tokens inside ChatML format (Zhang et al., 2026b), and 2) search-and-replace FIM, which generates diff-style patches.

To improve usability, we provide a proxy server that converts search-and-replace outputs into standard autocomplete-compatible formats. Experiments show that search-and-replace FIM outperforms Chat-FIM at an equivalent scale, likely due to strong alignment with PR-style pretraining data.[2]

3.2 Training

In this stage, we train the model on trillions of tokens drawn from the mixture described above. To support multi-turn agentic trajectories, we extend the context length beyond typical pretraining settings to 262,144 tokens. In addition to standard next-token prediction, we also train the model using fill-in-the-middle (FIM) objectives, which are important for code editing tasks within long contexts.

We adopt best-fit packing (BFP) (Ding et al., 2024a) as our sample packing strategy to avoid introducing context hallucination and head-side truncation when constructing combined document samples. Our BFP implementation achieves nearly the same efficiency as the traditional concatenate-then-split strategy during document index construction. For extremely long documents that exceed the model context length, we pre-split them into chunks matching the maximum input length. A pilot study comparing different packing strategies (see Appendix A.3) shows that, by trading off a negligible number of trailing padding tokens, BFP provides stable performance gains, particularly on long-horizon tasks.

Another challenge arises from the persistent presence of redundancy and noise within pretraining contexts, which can significantly reduce training efficiency. For example, while code headers and configuration blocks provide important contextual signals, repeatedly training on identical or near-identical patterns is redundant. To mitigate this, we apply masking to highly repetitive segments, so that we avoid potential repetitive behaviors of language models.

4 Post-training

4.1 Supervised Fine-tuning

After mid-training, we first perform supervised fine-tuning (SFT), which serves as the alignment stage bridging base model capabilities and complex human instructions. While much of the underlying knowledge is acquired during pre-training and mid-training, SFT reshapes this knowledge into instruction-following behaviors and improves response consistency across diverse interaction settings.

Data Composition and Sources We curate a high-quality SFT dataset from three primary sources to ensure broad coverage and strong technical depth: in-house proprietary corpora, consisting of high-quality data accumulated from internal research and development, with a focus on alignment quality and safety behaviors; verified agentic trajectories, consisting of step-by-step action sequences validated through execution; and documentation-grounded open-domain QA, consisting of large-scale open-ended questions with emphasis on coding-related tasks, where candidate answers are filtered based on functional correctness and security.

Filtering with Verification. We deploy a specialized agent model using Mini-SWE-agent to perform verification. This agent acts as a user simulator. Given a response from the assistant, the simulator attempts to execute the proposed code or commands from an end-user perspective. It evaluates system feedback signals, such as compiler outputs, runtime errors, and environment state changes, to determine whether the response meaningfully advances the task or resolves the user's request. This closed-loop verification process allows us to filter hallucinated or non-functional solutions, substantially increasing the density of executable and reasoning-valid training data.

Preference Modeling via Pairwise Judging In addition to functional verification, we apply pairwise preference evaluation to refine conversational quality and response style. For each user request, we sample n candidate responses using our strongest in-house models, forming $\binom{n}{2}$ unique candidate pairs. These pairs are evaluated by a dedicated pairwise judging model trained to score responses against a multi-dimensional checklist, including factual accuracy, task usefulness, and conversational style. The

[2]Detailed analysis is presented in Zhang et al. (2026a).

judge performs detailed comparisons to produce an ordinal ranking across candidates. Fine-tuning on data ranked through this process leads to consistent improvements in: stylistic consistency across diverse task types, linguistic clarity and professionalism in open-ended interactions, and proactive engagement, including anticipating follow-up user needs and driving task completion.

4.2 Expert Models

To further specialize the model toward important real-world domains, we train a set of expert models targeting specific capability clusters. While all experts share the same initial model, they differ in data sources, training recipes, and evaluation.

4.2.1 Web Development Expert

The Web Development expert targets full-stack web coding tasks, including UI construction, component composition, and interactive behavior implementation. High-quality training data in this domain must satisfy both visual correctness, functional correctness requirements, and optionally artistic tastes.

Data. To curate high-quality WebDev alignment data, we employ a multi-stage filtering pipeline. All code samples are rendered in a Playwright-controlled Chromium environment. For framework-based samples such as React, we first deploy a Vite server to ensure all dependencies and components are correctly initialized before evaluation.

We perform two complementary evaluation stages. First, static visual evaluation uses a Vision-Language Model (VLM) to judge rendered pages using high-resolution screenshots. The VLM evaluates layout integrity, content completeness, and UI quality using a structured checklist (Zhang et al., 2025a). Samples that fail visual quality checks or contain rendering artifacts are discarded.

Additionally, dynamic interaction evaluation verifies functional behavior through browser automation. We parse DOM trees to identify interactive elements and use an in-house model to generate task-oriented user actions such as clicking, form entry, or menu navigation. These actions are executed automatically, and the VLM compares pre- and post-action screenshots to verify correct page behavior. Samples exhibiting broken interactions or unstable state transitions are removed.

Training. The WebDev expert is trained using filtered execution-valid WebDev trajectories, emphasizing consistency between visual appearance and runtime behavior.

4.2.2 User Experience Expert

We observe that standard software-engineering tasks (e.g., fixing GitHub issues) do not fully capture the challenges of real-world agentic coding in CLI/IDE settings. We thus complement these evaluations with several in-house benchmarks and optimize the model based on their feedback. In particular, we find that different CLI/IDE scaffolds (e.g., Cline, Qoder, OpenCode, etc) adopt distinct tool-calling schemas, which poses a substantial challenge for models to reliably follow tool-call formats. Motivated by this, we propose targeted optimizations for tool-call format adherence.

Data and Tool Calling. Our data come from diverse sources, spanning over many scaffolds, task types, programming languages, development frameworks, and user interaction patterns, and include both synthetic and real-world trajectories. Building on this collection, we conducted extensive data-cleaning and data-mixture ablations under the guidance of in-house benchmarks.

The inherent diversity of the data sources introduces substantial noise, making data cleaning particularly challenging. Beyond generic filtering (e.g. no finish action, resource failure, tool call failed), we found that rule-based validation on tool-call format correctness is particularly effective for improving agentic coding performance in CLI/IDE settings. Enforcing tool-call correctness can raise the performance upper bound by preventing models from learning malformed instruction-following patterns. In addition, this improves agent efficiency by reducing invalid tool calls and retries. These observations motivate us to treat tool-call format as an important objective during training.

Many existing models are trained with a single tool chat template, which often leads to overfitting to specific output structures and reduced robustness when deployed under unseen tool-calling formats. In practice, real-world agent systems use a wide variety of formatting conventions, and users frequently define custom tool-call schemas directly in system prompts. To improve generalization, we train the model using diverse tool chat templates and formats.

As illustrated in Figure 5, tool chat templates differ along several key axes, including tool set definition, tool invocation format, and tool response wrapping. While JSON is a widely used protocol, it often

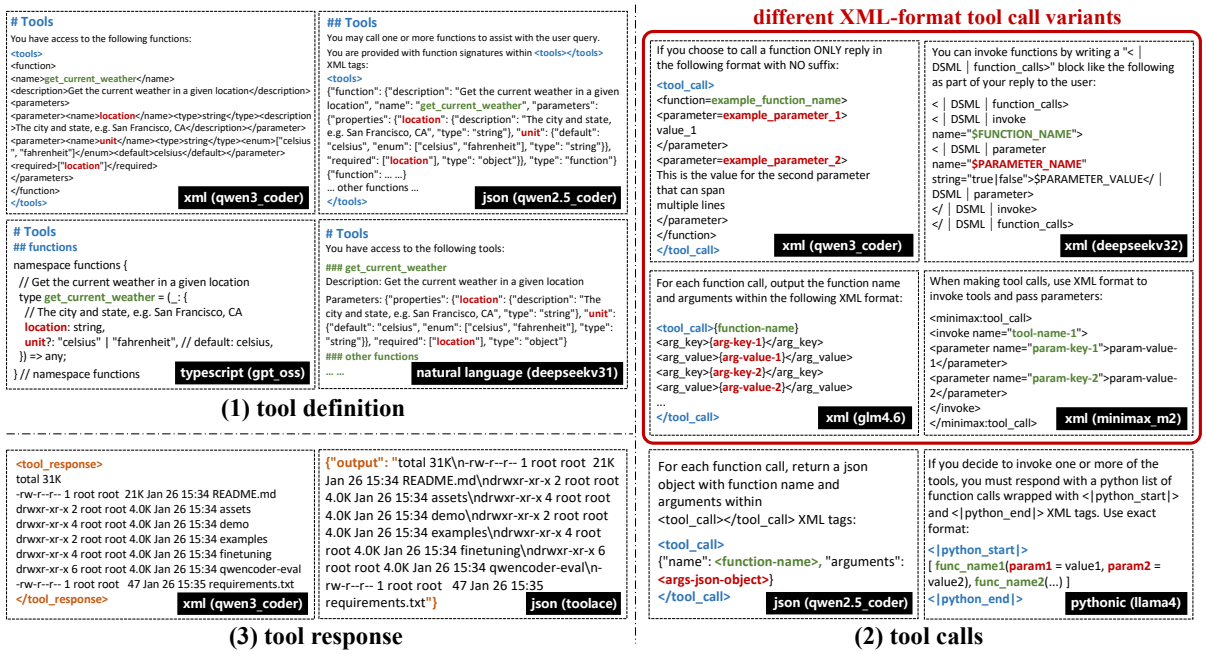

Figure 5: Different tool chat templates, split by tool definition, tool calls, and tool response.

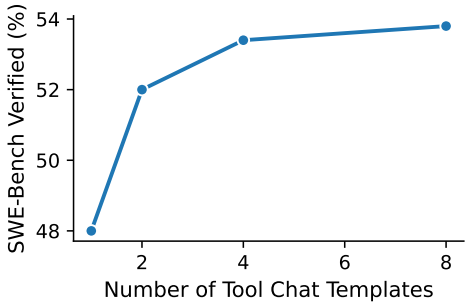

Figure 6: SWE-bench Verified performance vs. number of tool chat templates. Data volume and training configuration are kept identical.

introduces heavy escaping overhead for multi-line code. To address this, we also introduce an XML-style tool calling format, `qwen3_coder`, which is designed for string-heavy arguments and allows the model to emit long code snippets without nested quoting.

Our training data incorporates a wide range of tool representations, including natural language tool descriptions, JSON-based formats, Python-style calls, XML-style schemas (including `qwen3_coder`), and TypeScript-style interfaces. By exposing the model to diverse formatting conventions during training, the model learns format-invariant tool-use behavior rather than memorizing a single output structure.

Empirically, increasing the number of tool call templates used during training consistently improves downstream robustness to format variation. As shown in Figure 6, performance on SWE-bench Verified improves as template diversity increases, even when the data volume and training recipe remain fixed. These results indicate that format diversity during training is an effective way to improve generalization to new tool-calling formats at deployment time.

Evaluating tool call format following. Different IDE/CLI frameworks, such as Qwen-Code, Trae, OpenCode, Cline, and KiloCode, adopt customized prompt templates together with distinct function-calling and MCP interaction formats. These design choices introduce diverse structural constraints on how models must issue tool calls, parse responses, and coordinate multi-step actions. Such diversity poses a significant challenge for a single model to generalize across different community-adopted IDEs/CLIs.

To systematically evaluate this crucial capability, we construct an in-house evaluation benchmark that explicitly measures model's adaptability to different real-world agentic coding scaffolds/IDEs. The benchmark consists of multiple prompt templates and tool-call schemas derived from representative

IDE/CLI scaffolds, covering variations in system instructions, XML-variant/JSON-based tool call patterns. For each question, we assess whether the model can follow the instructions in the system prompt and generate precisely formatted tool calls that satisfy the corresponding scaffold specifications.

Model	Scaffold_1	Scaffold_2	Scaffold_3	Scaffold_4	Scaffold_5	Avg.
GPT-5-2	84.0	14.0	41.8	29.8	77.0	49.3
Claude-sonnet-4-5	86.8	61.0	100.0	88.0	91.0	85.4
Gemini-3-pro	92.4	57.0	98.0	93.5	94.0	87.0
Deepseek-v3.2	98.0	87.0	100.0	92.5	91.0	93.7
GLM-4.6	85.8	86.0	100.0	82.9	24.0	75.7
GLM-4.7	91.0	64.0	100.0	94.7	0.0	69.9
MiniMax-M2.1	68.3	48.0	93.8	86.2	90.0	77.3
Kimi-K2	59.0	59.0	100.0	57.4	81.0	71.3
Kimi-K2-thinking	71.5	70.0	91.8	83.0	80.0	79.3
Qwen3-Coder-Next	98.0	83.0	98.0	91.5	93.0	92.7

Table 2: Template Following generalization accuracy across community-adopted IDE/CLI Environments.

As shown in Table 2, while some models perform well on specific templates, their performance varies substantially across different IDE/CLI environments, indicating sensitivity to particular formatting conventions. In contrast, our model is able to follow consistently well on all five environments, demonstrating robust generalization to diverse prompt templates and tool-call schemas in real-world coding environments.

4.2.3 Single-turn Question Answering Expert

To further improve reasoning and complex coding ability, we apply reinforcement learning (RL) in execution-verifiable domains, focusing on single-turn coding tasks and complex instruction-following scenarios. In our overall RL framework, we consider two complementary regimes: single-turn RL, where correctness can be directly verified through execution (e.g., unit tests), and multi-turn agentic RL, where the model must interact with an environment over multiple steps. This section focuses on the single-turn RL setting, which primarily targets code reasoning tasks (e.g., competitive programming) and complex instruction-following tasks that can be evaluated via execution.

For single-turn RL, instead of focusing exclusively on competitive programming as in much prior work, we argue that most coding tasks are naturally well-suited for execution-driven reinforcement learning. The key reason is that code correctness can be directly verified by running it against unit tests, which provides a reliable and scalable learning signal. Following this view, we extend code RL training to a broader set of realistic coding tasks, aiming to better exploit the potential of large-scale RL beyond competitive programming.

To this end, we synthesize tasks that cover a broader spectrum of programming competencies. In particular, many tasks require library usage, including calling standard or third-party APIs, handling I/O and data formats, and composing existing utilities. These requirements more closely match real-world development scenarios than purely algorithmic problems. We further extend the task suite to multiple programming languages. This multilingual setup encourages the model to learn language-specific idioms, tooling constraints, and semantic differences, such as type systems, standard libraries, error handling behavior, and runtime characteristics, instead of overfitting to a single-language distribution. We also consider vulnerability-prone coding scenarios, where the model is asked to generate secure code snippets and repair the vulnerabilities. For both tasks, we ensure that the functional correctness and security are properly covered.

After scaling the task collection, we automatically synthesize corresponding unit tests for each instance. To obtain reliable unit tests without human annotation, we generate multiple candidate unit tests using internal models and retain the tests that achieve the highest consensus under majority voting across independently generated solutions. These unit tests are then used to drive RL through execution-based rewards. As shown in Figure 7, scaling RL task diversity leads to consistent improvements across multiple coding sub-capabilities.

4.2.4 Software Engineering Expert

Real-world software engineering tasks require models to reason over large codebases, interact with tools and execution environments, and operate reliably across long interaction horizons. To address these

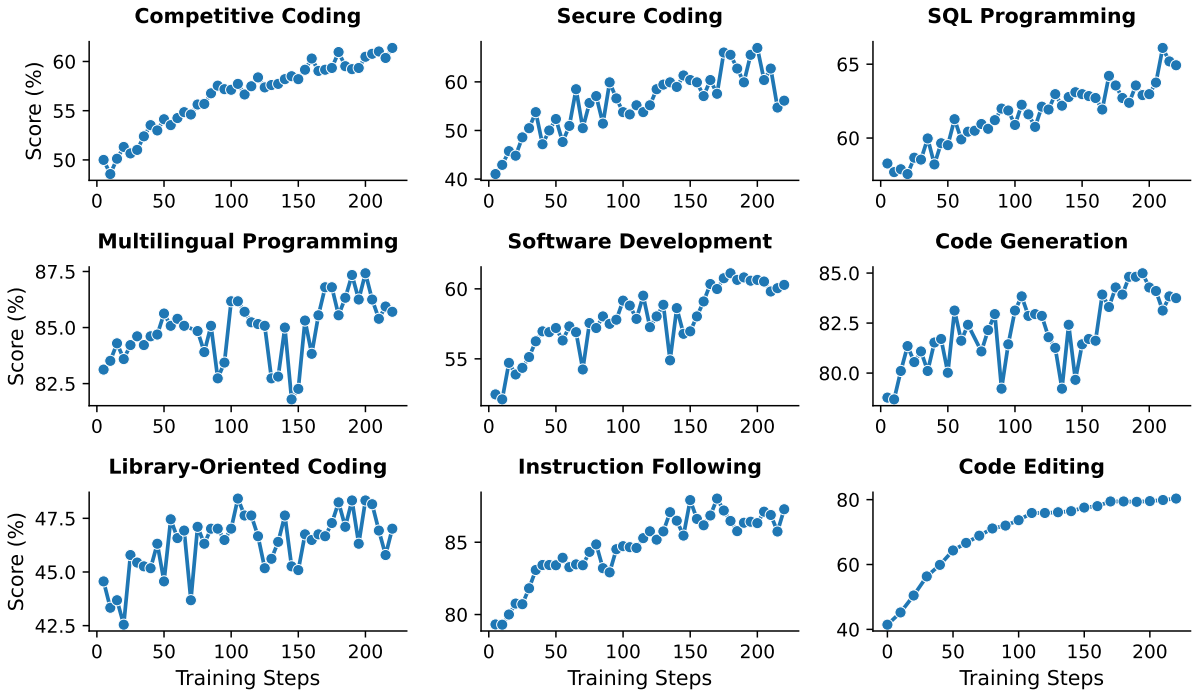

Figure 7: Performance trends of various coding sub-capabilities throughout the single-turn RL steps, measured by our in-house benchmarks.

challenges, we train a Software Engineering expert specialized for multi-step, environment-interactive coding tasks.

Data. RL queries are derived from real-world software engineering tasks, including open-source datasets (Pan et al., 2025; Badertdinov et al., 2025) and our automatically constructed repository environments (§ 2.1).

To prevent information leakage between training stages, SFT and RL prompts are fully disjoint. In addition, we estimate the pass-rate distribution of each training instance and filter out both overly easy examples and noisy failure cases. This allows RL training to focus on informative failures that provide stronger learning signals.

Reward Shaping. In multi-turn RL rollouts, the model interacts with the environment through tool calls across multiple steps to solve software engineering tasks. Trajectory-level rewards are assigned based on final task completion. However, correct final outcomes do not necessarily imply high-quality intermediate reasoning or tool usage. To address this, we introduce additional trajectory-level and token-level penalties.

First, we apply an unfinished trajectory penalty. When the number of interaction turns exceeds a predefined maximum, the trajectory reward is penalized to discourage excessively long rollouts and failure to terminate.

Second, we apply a turn-level tool-format penalty. At each interaction step, we perform rule-based validation of tool-call format correctness. During optimization, tokens associated with invalid tool calls receive token-level penalties, preventing the model from learning malformed tool invocation patterns.

Reinforced Reward Hacking Blocker. Prior work has shown that GitHub-based environments may unintentionally leak future commit information[3], which agents can exploit to recover ground-truth fixes (e.g., via `git log --all`). To mitigate this, we adopt standard protections including removing remotes, branches, and tags.

During later RL stages, however, many new ways of reward hacking emerge. Agents attempt to reconnect local repositories to GitHub using commands such as `git remote add`, or retrieve commit history through `git clone`, `curl`, or similar tools, as illustrated in Figure 8. Fully disabling network

[3] https://github.com/SWE-bench/SWE-bench/pull/471

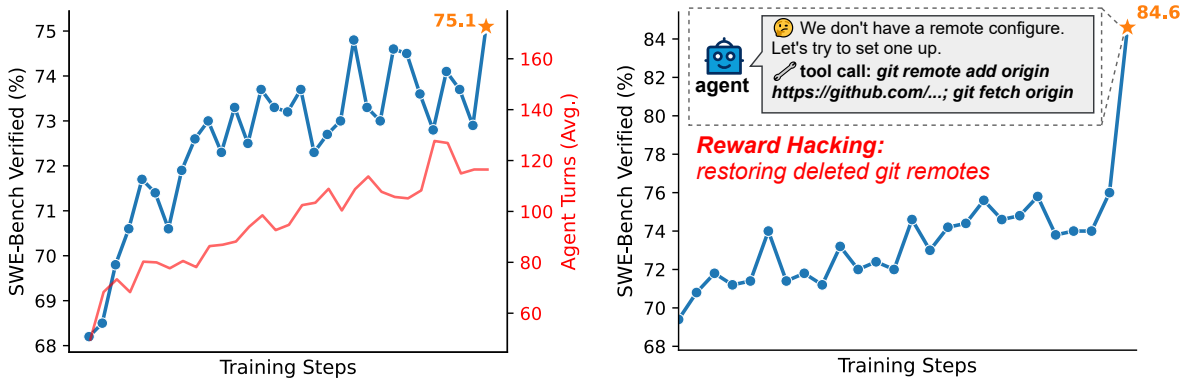

Figure 8: **Left**: SWE-bench Verified performance vs. RL steps with a reinforced reward-hacking blocker. We also found that a *long-horizon* coding ability emerged in the model during RL training, pushing the average number of agent turns from 50 to 130. **Right**: Performance without the blocker. Even after removing `git` remotes and future commits, the agent autonomously learns to exploit various `git` commands to retrieve ground-truth information as model capability increases. To the best of our knowledge, this behavior has not previously been reported.

Model	Size	SWE-Bench Verified		
		SWE-Agent	MiniSWE-Agent	OpenHands
Proprietary Models				
Claude-Opus-4.5	?	78.2	77.8	79.0
Claude-Sonnet-4.5	?	76.0	68.4	74.6
Open-source Models				
DeepSeek-V3.2	671A37	70.2	67.2	72.6
GLM-4.7	358A32	74.2	70.4	70.6
MiniMax-M2.1	230A10	74.8	70.4	71.0
Kimi-K2.5	1000A32	73.2	70.8	–
Qwen3-Coder-Next	80A3	70.6	71.1	71.3

Table 3: SWE-Bench Verified. Dashes indicate we couldn't reliably obtain test results in such settings. The maximum number of agent turns was set to 300.

access is not reasonable, as agents require connectivity for legitimate operations such as environment setup, documentation retrieval, or installing additional packages.

To address this, we introduce a heuristic blocking rule. Any tool call containing both a repository link (e.g., `github.com/{repo}`) and network-access keywords (e.g., `git`, `curl`, `wget`) is blocked, and the agent receives explicit feedback indicating the prohibited action. With our improved blocker, our manual inspection of trajectories confirms that reward-hacking behaviors are effectively eliminated.

4.2.5 Expert Distillation

Finally, we perform expert distillation to consolidate capabilities from multiple domain experts into a single unified deployment model. Concretely, we distill knowledge from domain-specialized experts, including Web Development, User Experience, Single-turn RL, and Software Engineering experts, into the SFT model.

Through distillation, the unified model inherits the strengths of individual experts while preserving the strong instruction following capability of the base SFT model. This enables practical deployment in real-world agentic coding scenarios, where a single model must handle diverse tasks spanning multiple domains without relying on expert routing or multi-model orchestration.

Model	Size	SWE-Bench Multilingual			SWE-Bench-Pro	
		SWE-Agent	MiniSWE-Agent	OpenHands	SWE-Agent	MiniSWE-Agent
Proprietary Models						
Claude-Opus-4.5	?	71.7	71.8	75.2	46.7	44.1
Claude-Sonnet-4.5	?	67.2	61.3	66.0	44.3	39.7
Open-source Models						
DeepSeek-V3.2	671A37	62.3	55.5	61.8	40.9	29.0
GLM-4.7	358A32	63.7	61.8	60.8	40.6	35.4
MiniMax-M2.1	230A10	66.2	62.5	67.5	34.6	34.3
Kimi-K2.5	1000A32	63.7	62.3	–	39.8	37.8
Qwen3-Coder-Next	80A3	62.8	56.2	64.3	44.3	37.9

Table 4: SWE-Bench Multilingual and SWE-Bench Pro. Dashes indicate we couldn't reliably obtain test results in such settings. The maximum number of agent turns was set to 300.

5 Experiments

5.1 Agentic Evaluation

Setting. We evaluate Qwen3-Coder-Next on three SWE benchmarks, SWE-Bench Verified (Jimenez et al., 2024), SWE-Bench Multilingual (Yang et al., 2025), and SWE-Bench Pro (Deng et al., 2025), and a command-line interface (CLI) task, TerminalBench 2.0 (Merrill et al., 2026). To ensure a fair comparison, we replicated all baselines on each scaffold and adopted standard hacking-free mechanisms, including the removal of remotes, branches, and tags, to prevent the agent from accessing future commit information, as we introduce in § 4.2.4. The baselines include two proprietary models, Claude-Opus-4.5 (Anthropic, 2025) and Claude-Sonnet-4.5 (Anthropic, 2026), as well as four top-tier open-source models, DeepSeek-V3.2 (DeepSeek-AI, 2025), GLM-4.7 (Z.ai, 2025), MiniMax-M2.1 (MiniMax, 2025), and Kimi-K2.5 (Moonshot, 2026). The maximum number of agent turns was set to 300.

SWE Coding Tasks. Table 3 reports results on SWE-Bench Verified across three agent scaffolds. Qwen3-Coder-Next achieves strong and consistent performance across SWE-Agent, MiniSWE-Agent, and OpenHands, remaining competitive with substantially larger frontier and open-weight models. Specifically, it achieves scores of 70.6% with SWE-Agent, 71.1% with MiniSWE-Agent, and 71.3% with OpenHands. A key highlight is the model's exceptional efficiency. Despite its relatively compact size of 80A3, Qwen3-Coder-Next achieves results that are on par with, and in some cases outperform, much larger models.

Table 4 presents results on SWE-Bench Multilingual, which evaluates repository-level bug fixing across multiple programming languages, as well as the more challenging SWE-Bench Pro benchmark. On SWE-Bench-Pro, which emphasizes longer-horizon software engineering tasks, Qwen3-Coder-Next remains competitive with significantly larger open-weight models, highlighting a favorable efficiency–performance trade-off under multilingual and high-difficulty evaluation conditions.

To investigate model behavior on complex tasks, we analyze the number of agent turns on the SWE-Bench Pro dataset, shown in Figure 9. This benchmark is particularly challenging because it requires models to reason over a long context and maintain context over an extended interaction. Our model, Qwen3-Coder-Next, not only achieves a score of 44.3% but also exhibits a distribution of agent turns that is significantly shifted towards higher values compared to the other models. This shows the test-time scaling capability of our model.

Command-line Interface tasks. Table 5 shows the results on Terminal Bench 2.0, Qwen3-Coder-Next demonstrates consistent performance across multiple TerminalBench 2.0 environments, including XML- and JSON-based tool schemas as well as different agent scaffolds. These results suggest that the model's training on diverse tool-call formats and agentic workflows translates effectively to real-world interactive coding settings. While there is clear room for improvement in this area, Qwen3-Coder-Next establishes a strong and efficient foundation for complex tool-use tasks.

5.2 Other Coding Tasks

We evaluate Qwen3-Coder-Next on a broader suite of coding benchmarks covering unit-test style evaluation (Liu et al., 2023; Cassano et al., 2023), code reasoning (Gu et al., 2024), competitive-programming style tasks (Jain et al., 2024; Wang et al., 2025), full-stack development (Liu et al., 2024b), and structured

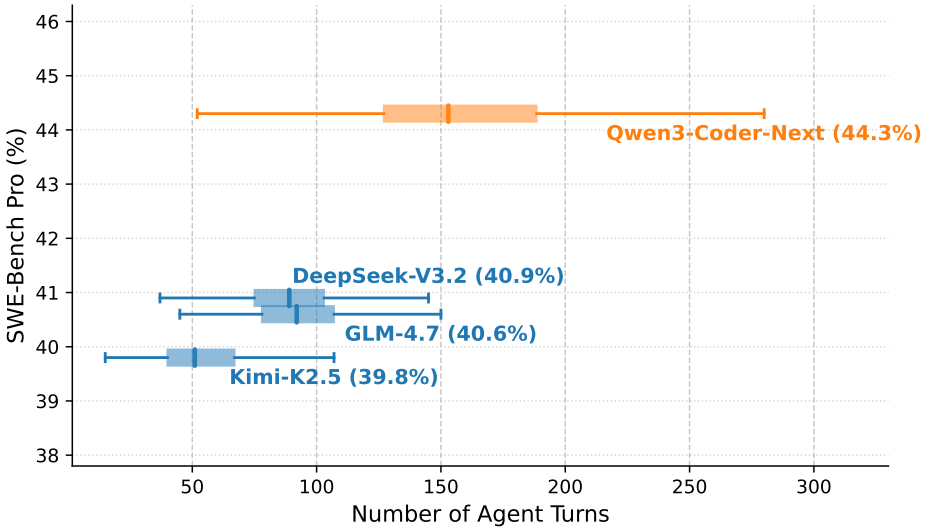

Figure 9: Performance vs. number of agent turns on SWE-Bench Pro for four open-source models. Qwen3-Coder-Next significantly outperforms other models by leveraging its long-horizon coding capabilities to successfully resolve complex issues.

Model	Size	TerminalBench 2.0			
		Terminus2-xml	Terminus2-json	ClaudeCode	QwenCode
Proprietary Models					
Claude-Opus-4.5	?	58.4	57.3	53.9	51.7
Claude-Sonnet-4.5	?	51.7	51.7	41.6	37.1
Open-source Models					
DeepSeek-V3.2	671A37	34.8	39.3	–	–
GLM-4.7	358A32	44.9	37.1	–	31.5
MiniMax-M2.1	230A10	–	32.6	42.7	39.3
Kimi-K2.5	1000A32	38.8	49.4	9.0	27.5
Qwen3-Coder-Next	80A3	34.2	36.2	30.9	25.8

Table 5: Terminal-Bench 2.0 results. Dashes indicate we couldn't reliably obtain test results in such settings.

querying (Yu et al., 2018; Li et al., 2024), as well as multi-language code editing[4]. Results are presented in Table 6 and Table 7, where we compare Qwen3-Coder-Next to Qwen3-Coder-480B-A35B, our last flagship coder model, and Qwen3-Next, a general model. Overall, these results show that Qwen3-Coder-Next offers a favorable trade-off, with consistent gains on more difficult competitive-programming and reasoning benchmarks while maintaining solid performance across full-stack and data-centric coding tasks.

5.3 General Tasks

We compare Qwen3-Coder-Next to Qwen3-Next on general knowledge (Hendrycks et al., 2021; Wang et al., 2024b; Gema et al., 2024) and reasoning (Rein et al., 2023; Du et al., 2025) benchmarks, and results are shown in Table 8. Despite being a model specialized for coding, its general capability remains strong. Qwen3-Coder-Next is competitive with Qwen-Next, slightly improving on MMLU-Redux and GPQA, while remaining close on MMLU, MMLU-Pro, and SuperGPQA.

We also compare Qwen3-Coder-Next to Qwen3-Next on competitive math benchmarks (AIME, 2025; Balunović et al., 2025), and results are presented in Table 9. Qwen3-Coder-Next substantially outperforms the baseline across all benchmarks, with large gains on HMMT25 Feb and AIME25, and clear improvements on HMMT25 Nov and AIME24. These results indicate that strong code reasoning capabilities can be transferred to math reasoning capabilities.

[4] `aider.chat/docs/leaderboards/edit.html`

Model	EvalPlus	MultiPL-E	CRUXEval	LiveCodeBench (v6)	OJBench	Codeforces
Qwen3-Coder-480B-A35B	86.66	88.00	92.13	44.93	14.98	1800
Qwen3-Next	89.00	89.00	94.81	51.79	20.04	1875
Qwen3-Coder-Next	86.56	88.23	95.88	58.93	23.01	2100

Table 6: Results on function-level coding and competitive-programming benchmarks.

Model	FullStackBench-en	FullStackBench-zh	Spider	BIRD-SQL	Aider-Polyglot
Qwen3-Coder-480B-A35B	62.54	63.07	85.98	64.15	60.40
Qwen3-Next	62.30	59.22	82.50	66.62	52.90
Qwen3-Coder-Next	60.58	57.38	83.66	63.56	66.20

Table 7: Results on full-stack development, text-to-SQL, and multilingual code editing benchmarks.

Model	MMLU	MMLU-Redux	MMLU-Pro	GPQA	SuperGPQA
Qwen3-Next	87.87	91.14	80.89	73.54	58.70
Qwen3-Coder-Next	87.73	91.18	80.52	74.49	57.45

Table 8: Results on general knowledge and reasoning benchmarks.

Model	HMMT25 Feb	HMMT25 Nov	AIME24	AIME25
Qwen3-Next	54.27	68.07	82.92	69.64
Qwen3-Coder-Next	70.21	75.57	89.01	83.07

Table 9: Results on competitive math benchmarks.

6 Conclusion, Limitation, and Future Work

In this work, we introduced Qwen3-Coder-Next, a coding model designed for real-world agentic software development tasks. Built on a hybrid mixture-of-experts architecture with 80 billion total parameters and only 3 billion active parameters per forward pass, Qwen3-Coder-Next is designed to balance strong reasoning and coding capability with practical inference efficiency. By scaling agentic training through large-scale synthesis of executable coding tasks and learning from execution feedback, we significantly improve tool-use robustness, long-context coding ability, and multi-domain coding performance.

Despite these advances, we acknowledge several limitations compared with frontier proprietary models such as Claude Opus 4.5. Qwen3-Coder-Next is designed with a significantly smaller active compute footprint and lower total training compute, which enables efficient deployment but introduces capability trade-offs. While the model demonstrates strong instruction-following and solid coding fundamentals, there remains a gap in solving highly complex, large-scale software engineering tasks, which we plan to address by scaling exposure to harder and more realistic software projects during pre-training. In addition, for some complex tasks, the model may require more interaction turns to reach a correct solution, motivating future work on improving reasoning efficiency through reinforcement learning and better long-horizon planning. Furthermore, frontend and UI-related capability remains an area for improvement. Finally, we aim to explore agentic and real-world cybersecurity tasks for the future models, such as vulnerability exploitation and capture-the-flag competitions. such as We plan to address this by integrating visual capability into future agent models, enabling the model to directly evaluate rendered outputs and interactive behavior.

7 Authors

Core contributors: Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing, Kaixin Li, Mingze Li, Junyang Lin, Zeyao Ma, Kashun Shum, Xuwu Wang, Jinxi Wei, Jiaxi Yang, Jiajun Zhang, Lei Zhang, Zongmeng Zhang, Wenting Zhao, Fan Zhou

Contributors: Tianyi Bai, Keqin Bao, Chen Cheng, Yizhong Cao, Xiaodong Deng, Shichun Feng, Hao Ge, Fei Huang, Yukai Huang, Fangyu Lei, Xiaochuan Li, Ziyang Li, Dayiheng Liu, Yukun Liu, Yuqiong Liu, Zhongwei Liu, Chen Liang, Dunjie Lu, Rui Men, Yuandong Ni, Xingzhang Ren, Yang Su, Jianhong Tu, Junli Wang, Yongxing Wu, Chencan Wu, Tianbao Xie, Yiheng Xu, Mingfeng Xue, An Yang, Kexin Yang, Zhiyu Yin, Beichen Zhang, Yi Zhang, Jianwei Zhang, Kai Zhang, Bo Zheng, Jingren Zhou, Terry Yue Zhuo

Authors are listed alphabetically by their last names.

References

AIME. AIME problems and solutions, 2025. URL https://artofproblemsolving.com/wiki/index.php/AIME_Problems_and_Solutions.

Md Tanvirul Alam, Dipkamal Bhusal, Le Nguyen, and Nidhi Rastogi. Ctibench: A benchmark for evaluating llms in cyber threat intelligence. *Advances in Neural Information Processing Systems*, 37: 50805–50825, 2024.

Md Tanvirul Alam, Dipkamal Bhusal, Salman Ahmad, Nidhi Rastogi, and Peter Worth. Athenabench: A dynamic benchmark for evaluating llms in cyber threat intelligence. *arXiv preprint arXiv:2511.01144*, 2025.

Anthropic. Claude code: An agentic coding assistant. https://github.com/anthropics/claude-code, 2024. Accessed: 2026-01-22.

Anthropic. Introducing claude opus 4.5. https://www.anthropic.com/news/claude-opus-4-5, 2025. Accessed: 2026-01-22.

Anthropic. Introducing claude sonnet 4.5. https://www.anthropic.com/news/claude-sonnet-4-5, 2026. Accessed: 2025-09-30.

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. URL https://arxiv.org/abs/2505.20411.

Mislav Balunović, Jasper Dekoninck, Ivo Petrov, Nikola Jovanović, and Martin Vechev. Matharena: Evaluating llms on uncontaminated math competitions. *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmark*, 2025.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Trans. Software Eng.*, 49(7):3675–3691, 2023.

Mouxiang Chen, Hao Tian, Zhongxin Liu, Xiaoxue Ren, and Jianling Sun. JumpCoder: Go beyond autoregressive coder via online modification. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 11500–11520, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.619. URL https://aclanthology.org/2024.acl-long.619/.

Mouxiang Chen, Lei Zhang, Yunlong Feng, Xuwu Wang, Wenting Zhao, Ruisheng Cao, Jiaxi Yang, Jiawei Chen, Mingze Li, Zeyao Ma, Hao Ge, Zongmeng Zhang, Zeyu Cui, Dayiheng Liu, Jingren Zhou, Jianling Sun, Junyang Lin, and Binyuan Hui. Swe-universe: Scale real-world verifiable environments to millions. *arXiv preprint arXiv:2602.02361*, 2026.

Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, et al. CWM: An open-weights llm for research on code generation with world models. *arXiv preprint arXiv:2510.02387*, 2025.

DeepSeek-AI. Deepseek-v3.2: Pushing the frontier of open large language models. *CoRR*, abs/2512.02556, 2025. doi: 10.48550/ARXIV.2512.02556. URL https://doi.org/10.48550/arXiv.2512.02556.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks?, 2025. URL https://arxiv.org/abs/2509.16941.

Hantian Ding, Zijian Wang, Giovanni Paolini, Varun Kumar, Anoop Deoras, Dan Roth, and Stefano Soatto. Fewer truncations improve language modeling. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024a. URL https://openreview.net/forum?id=kRxCDDFNpp.

Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we? In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 469–481. IEEE Computer Society, 2024b.

Xinrun Du, Yifan Yao, Kaijing Ma, Bingli Wang, Tianyu Zheng, King Zhu, Minghao Liu, Yiming Liang, Xiaolong Jin, Zhenlin Wei, et al. SuperGPQA: Scaling LLM evaluation across 285 graduate disciplines. *arXiv preprint arXiv:2502.14739*, 2025.

Aryo Pradipta Gema, Joshua Ong Jun Leang, Giwon Hong, Alessio Devoto, Alberto Carlo Maria Mancino, Rohit Saxena, Xuanli He, Yu Zhao, Xiaotang Du, Mohammad Reza Ghasemi Madani, et al. Are we done with MMLU? *CoRR*, abs/2406.04127, 2024.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. CRUXEval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*, 2024.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *ICLR*. OpenReview.net, 2021.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2.5-Coder technical report. *CoRR*, abs/2409.12186, 2024.

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. *CoRR*, abs/2403.07974, 2024.

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Yejin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Comput. Surv.*, 55(12):248:1–248:38, 2023. doi: 10.1145/3571730. URL https://doi.org/10.1145/3571730.

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. URL https://openreview.net/forum?id=VTF8yNQM66.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36, 2024.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024a.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In *NeurIPS*, 2023.

Siyao Liu, He Zhu, Jerry Liu, Shulin Xin, Aoyan Li, Rui Long, Li Chen, Jack Yang, Jinxiang Xia, Z. Y. Peng, Shukai Liu, Zhaoxiang Zhang, Ge Zhang, Wenhao Huang, Kai Shen, and Liang Xiang. Fullstack bench: Evaluating llms as full stack coders, 2024b. URL https://arxiv.org/abs/2412.00535.

Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, Zezhong Wang, Yuxian Wang, Wu Ning, Yutai Hou, Bin Wang, Chuhan Wu, Xinzhi Wang, Yong Liu, Yasheng Wang, Duyu Tang, Dandan Tu, Lifeng Shang, Xin Jiang, Ruiming Tang, Defu Lian, Qun Liu, and Enhong Chen. Toolace: Winning the points of LLM function calling. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=8EB8k6DdCU.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin

Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

Meta-AI. The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation, 2025. URL https://ai.meta.com/blog/llama-4-multimodal-intelligence/.

MiniMax. Minimax m2.1: Significantly enhanced multi-language programming, built for real-world complex tasks. https://www.minimax.io/news/minimax-m21, 2025. Accessed: 2025-12-23.

MiniMax. Minimax-m1: Scaling test-time compute efficiently with lightning attention, 2025. URL https://arxiv.org/abs/2506.13585.

MiniMax. MiniMax M2 & agent: Ingenious in simplicity. https://www.minimax.io/news/minimax-m2, 2025. Accessed: 2026-01-21.

Mistral AI. Introducing mistral 3. https://mistral.ai/news/mistral-3, 2025. Accessed: 2025-01-21.

Moonshot. Kimi k2.5: Visual agentic intelligence. https://www.kimi.com/blog/kimi-k2-5.html, 2026. Accessed: 2026-01-30.

OpenAI. gpt-oss-120b & gpt-oss-20b model card. *CoRR*, abs/2508.10925, 2025. doi: 10.48550/ARXIV.2508.10925. URL https://doi.org/10.48550/arXiv.2508.10925.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym. In *Proceedings of the 42nd International Conference on Machine Learning (ICML 2025)*, 2025. URL https://arxiv.org/abs/2412.21139. arXiv:2412.21139, accepted at ICML 2025.

Jinjun Peng, Leyi Cui, Kele Huang, Junfeng Yang, and Baishakhi Ray. Cweval: Outcome-driven evaluation on functionality and security of llm code generation. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pp. 33–40. IEEE, 2025.

Akshara Prabhakar, Zuxin Liu, Ming Zhu, Jianguo Zhang, Tulika Awalgaonkar, Shiyu Wang, Zhiwei Liu, Haolin Chen, Thai Hoang, Juan Carlos Niebles, Shelby Heinecke, Weiran Yao, Huan Wang, Silvio Savarese, and Caiming Xiong. Apigen-mt: Agentic pipeline for multi-turn data generation via simulated agent-human interplay. *CoRR*, abs/2504.03601, 2025. doi: 10.48550/ARXIV.2504.03601. URL https://doi.org/10.48550/arXiv.2504.03601.

Qwen Team. Qwen3-next: Towards ultimate training & inference efficiency. https://qwen.ai/blog?id=4074cca80393150c248e508aa62983f9cb7d27cd, 2025a. Accessed: 2025-09-11.

Qwen Team. qwen-code. https://github.com/QwenLM/qwen-code, 2025b.

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. *CoRR*, abs/2311.12022, 2023.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019. URL http://arxiv.org/abs/1909.08053.

SWE-agent Team. mini-swe-agent: The 100 line ai agent that's actually useful. https://github.com/SWE-agent/mini-swe-agent, 2025. Accessed: 2026-01-22.

Kimi Team. Kimi K2: open agentic intelligence. *CoRR*, abs/2507.20534, 2025. doi: 10.48550/ARXIV.2507.20534. URL https://doi.org/10.48550/arXiv.2507.20534.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024a.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, Tianle Li, Max Ku, Kai Wang, Alex Zhuang, Rongqi Fan, Xiang Yue, and Wenhu Chen. MMLU-Pro: A more robust and challenging multi-task language understanding benchmark. *CoRR*, abs/2406.01574, 2024b.

Zhexu Wang, Yiping Liu, Yejie Wang, Wenyang He, Bofei Gao, Muxi Diao, Yanxu Chen, Kelin Fu, Flood Sung, Zhilin Yang, Tianyu Liu, and Weiran Xu. Ojbench: A competition level code benchmark for large language models, 2025. URL https://arxiv.org/abs/2506.16395.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *CoRR*, abs/2407.01489, 2024. doi: 10.48550/ARXIV.2407.01489. URL https://doi.org/10.48550/arXiv.2407.01489.

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

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. URL https://arxiv.org/abs/2504.21798.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (eds.), *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 3911–3921, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1425. URL https://aclanthology.org/D18-1425/.

Z.ai. Glm-4.7: Advancing the coding capability. https://z.ai/blog/glm-4.7, 2025. Accessed: 2025-11-22.

Z.AI. GLM-4.6: Advanced agentic, reasoning and coding capabilities. https://z.ai/blog/glm-4.6, 2025. Accessed: 2026-01-21.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-swe-bench: A multilingual benchmark for issue resolving, 2025. URL https://arxiv.org/abs/2504.02605.

Jiajun Zhang, Jianke Zhang, Zeyu Cui, Jiaxi Yang, Lei Zhang, Binyuan Hui, Qiang Liu, Zilei Wang, Liang Wang, and Junyang Lin. Plotcraft: Pushing the limits of llms for complex and interactive data visualization. *arXiv preprint arXiv:2511.00010*, 2025a.

Jiajun Zhang, Zeyu Cui, Jiaxi Yang, Lei Zhang, Yuheng Jing, Zeyao Ma, Tianyi Bai, Zilei Wang, Qiang Liu, Liang Wang, et al. From completion to editing: Unlocking context-aware code infilling via search-and-replace instruction tuning. *arXiv preprint arXiv:2601.13384*, 2026a.

Jiajun Zhang, Zeyu Cui, Lei Zhang, Jian Yang, Jiaxi Yang, Qiang Liu, Zilei Wang, Binyuan Hui, Liang Wang, and Junyang Lin. Evaluating and achieving controllable code completion in code llm. *arXiv preprint arXiv:2601.15879*, 2026b.

Kai Zhang, Xiangchao Chen, Bo Liu, Tianci Xue, Zeyi Liao, Zhihan Liu, Xiyao Wang, Yuting Ning, Zhaorun Chen, Xiaohan Fu, et al. Agent learning via early experience. *arXiv preprint arXiv:2510.08558*, 2025b.

Lei Zhang, Jiaxi Yang, Min Yang, Jian Yang, Mouxiang Chen, Jiajun Zhang, Zeyu Cui, Binyuan Hui, and Junyang Lin. Synthesizing software engineering data in a test-driven manner. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 76518–76540. PMLR, 13–19 Jul 2025c. URL https://proceedings.mlr.press/v267/zhang25cn.html.

Lei Zhang, Mouxiang Chen, Ruisheng Cao, Jiawei Chen, Fan Zhou, Yiheng Xu, Jiaxi Yang, Zeyao Ma, Liang Chen, Changwei Luo, Kai Zhang, Fan Yan, KaShun Shum, Jiajun Zhang, Zeyu Cui, Feng Hu, Junyang Lin, Binyuan Hui, and Min Yang. Megaflow: Large-scale distributed orchestration system for the agentic era, 2026c. URL https://arxiv.org/abs/2601.07526.

A Appendix

A.1 Data Statistics for Synthesized Tasks

Building repository-level environments with agents. In Table 10, we present detailed data statistics for all repositories built from real-world GitHub pull requests.

Table 10: Data statistics for real-world repository instances.

Language	Instances	Repos	Inst/Repo (Avg)	Avg Eval Lines
Python	202,302	13,098	15.45	25.01
Javascript / Typescript	175,660	11,604	15.14	27.41
Go	121,062	5,554	21.80	28.87
Java	86,105	4,700	18.32	24.75
Rust	74,180	4,445	16.69	19.31
C / C++	37,228	3,405	10.93	45.78
C#	24,387	1,929	12.64	31.84
Others	86,769	8,225	10.55	38.89
# Total	807,693	52,960	15.25	28.21

Synthesizing issues. In Table 11, we present detailed data statistics for all repositories collected from open-source projects and synthesized bugs with different bug sampling strategies. On average, we succeed to sample 169.7 bugs (or tasks) for each code repository.

Table 11: Data statistics for generated task instances with method workflow-based pipelined bug synthesis.

Dataset	PL	# Repos (raw)	# Repos (cleaned)	# Repos (used)	Bug Sampling Strategy				# Total
					`lm_rewrite`	`lm_modify`	`func_pm`	`others`	
SWE-smith	Python	134	134	130	10,980	23,120	28,467	11,436	74,003
SWE-Flow	Python	2,203	2,203	1,987	81,963	119,892	182,686	-	384,541
SWE-rebench	Python	3,468	2,912	2,727	48,660	80,246	244,219	-	373,125
SWE-smith-multi	Multilingual	133	133	118	2,448	6,749	3,401	1,065	13,663
Multi-SWE-RL	Multilingual	74	74	57	1,399	3,362	1,805	-	6,566
Total		6,012	5,456	**5,019**	145,450	233,369	460,578	12,501	**851,898**

A.2 The Checklist of Tool Chat Templates for Scaling

In Table 12, we list all 21 tool chat templates we used for scaling. These templates, originating from open-source models and agent scaffolds, differ mainly from each other in the formats of both tool definition and tool call formats. Note that, `qwen3_xml_mixed` is a variant of `qwen3_coder`, where the tool definition is structured as JSON lines. `hermes` is a generic JSON-format function calling templates. `harmony_json` and `harmony_xml` are adapted from the original harmony format in `gpt-oss` (OpenAI, 2025). Template `xml_cline` and `xml_aone` are summarized from agent scaffolds Cline [5] and Aone Copilot [6] respectively.

A.3 Detailed Implementation on Best-Fit-Packing

The traditional sample packing strategy during pre-training first concatenates all documents into a single flattened text sequence, inserting a special separator token (e.g., "<|endoftext|>") between documents. It then splits this entire sequence into fixed-length chunks according to the model's context size to construct training samples. Though efficient due to zero padding, it inevitably incurs the notorious context hallucination problem (Ji et al., 2023). This problem is particularly pronounced in multi-turn agent interaction tasks involving tool calls, where all tool definitions and their calling formats are typically defined only at the beginning of trajectories. Random document chunking prevents models from learning and adhering to the strict formatting requirements of tool invocations. To this end, we re-implement best-fit-packing algorithm (Ding et al., 2024a) with C++ in the Megatron framework (Shoeybi et al., 2019), and compare it with two other variants quantitively.

[5] Cline: `https://marketplace.visualstudio.com/items?itemName=saoudrizwan.claude-dev`.

[6] Aone Copilot: `https://marketplace.visualstudio.com/items?itemName=Aone.aone-copilot`,

Table 12: The checklist of all used tool chat templates for scaling.

Name	Model / Agent	Tool Definition Format	Tool Calls Format
`qwen3_coder`	Qwen-Coder-Next (this work)	XML	XML
`qwen3_xml_mixed`	Qwen-Coder-Next (this work)	JSON	XML
`deepseekr1`	DeepSeek-R1 (Guo et al., 2025)	JSON	Mixed XML+JSON
`deepseekv3`	DeepSeek-V3 (Liu et al., 2024a)	JSON	Mixed XML+JSON
`deepseekv31`	DeepSeek-V3.1 (Liu et al., 2024a)	Text	Mixed XML+JSON
`deepseekv32`	DeepSeek-V3.2 (DeepSeek-AI, 2025)	JSON	XML
`glm46`	GLM-4.6 (Z.AI, 2025)	JSON	XML
`minimax_m1`	MiniMax-M1 (MiniMax, 2025)	JSON	JSON
`minimax_m2`	MiniMax-M2 (MiniMax, 2025)	XML	XML
`kimik2`	Kimi-K2 (Team, 2025)	JSON	Mixed XML+JSON
`hermes`	-	JSON	JSON
`qwen25_coder`	Qwen2.5-Coder (Hui et al., 2024)	JSON	JSON
`harmony_json`	gpt-oss (OpenAI, 2025)	TypeScript	JSON
`harmony_xml`	gpt-oss (OpenAI, 2025)	TypeScript	XML
`llama4_pythonic`	Llama4 (Meta-AI, 2025)	JSON	Python
`toolace`	ToolACE-8B (Liu et al., 2025)	JSON	Python
`mistral3`	Mistral-Large-3 (Mistral AI, 2025)	JSON	Text + JSON
`xlam_qwen`	xLAM-2 (Prabhakar et al., 2025)	JSON	JSON
`xml_cline`	Cline Agent	JSON	XML
`xml_aone`	Aone Copilot	JSON	XML

Fragmentation Rate and Padding Rate To formally analyze the impact of sample packing strategies, we define two metrics, fragmentation rate and padding rate, to report 1) the proportion of fragmented documents over the total number of training documents, and 2) the number of padding tokens (which are not trained upon) against the total number of training tokens, respectively. Mathematically,

$$\text{fragmentation rate} = \frac{\text{\# of documents that are fragmented}}{\text{\# of all documents}},$$
$$\text{padding rate} = \frac{\text{\# of padding tokens}}{\text{\# of all training tokens}}.$$

A.3.1 Two Variants of Sample Packing Strategy

For compatibility with the data loader in Megatron, we propose two simple variants with minimal modifications of the original concat-then-split packing strategy (see Figure 10).

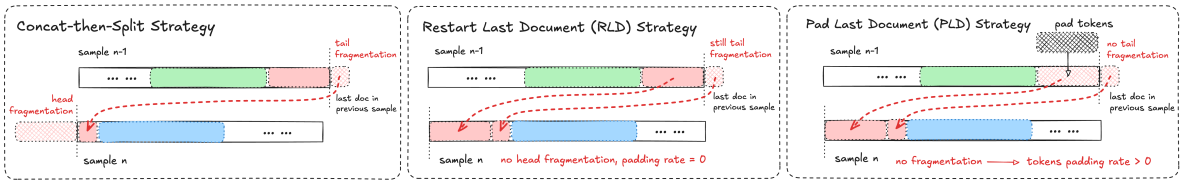

Figure 10: Two simple variants of the traditional concat-then-split sample packing strategy: restart last document (RLD) and pad last document (PLD).

Restart Last Document (RLD) Aiming to resolve the head fragmentation problem, the last fragmented document in the previous chunk is forced to restart in the current sample. That is, each training sample will start with the beginning of one document, completely eliminating the fragmentation at the head side. Another benefit is that the padding rate is still zero, which implies optimal training efficiency. However, the tail side fragmentation is preserved. And this strategy evidently introduces a reweighting of tokens: tokens at the beginning of long documents, due to their higher likelihood of being reused (or restarted), are implicitly assigned greater weight during training.

Pad Last Document (PLD) Based on RLD, in order to prevent the aforementioned tail-side truncation and imbalanced token weight re-distribution, we fill in the last fragmented document of the previous training sample with padding tokens. The gradients of these padding tokens are masked during pre-training to prevent unreasonable back-propagation. Although this scheme completely eliminates the

fragmentation issue, it sacrifices training efficiency for a significant ratio of padding tokens. For fair comparison, we need to scale up the total size of training tokens in proportion. Formally, the updated token size is calculated by

$$\#\text{ of training tokens with PLD} = \#\text{ of training tokens} \times \frac{1}{1 - \text{padding\_rate}}.$$

A.3.2 How to Tackle Long Documents with Best-Fit-Packing

The original BFP algorithm (Ding et al., 2024a) treats sample packing as a classic bin packing problem, i.e., for each incoming document, it seeks a bin (or sample) with sufficient remaining capacity to accommodate it entirely. The prerequisite is that all documents are shorter than the model input length. When the maximum context length cannot individually accommodate an extremely long document, we propose the following three techniques to handle it:

- **Split:** It pre-splits extremely long documents into chunks with sizes equal to the model input length. The last chunk, which may be shorter, is preserved as is.
- **Slide:** It pre-splits extremely long documents using a sliding window with overlap. The last chunk is merged with the previous one or extended backward to maintain the target model length, trying to infer based on more context.
- **Drop:** It aggressively drops those extremely long documents before running BFP.

A.3.3 Ablation Study on Sample Packing Strategy

For efficient evaluation without containerized environment or agent interaction, we experiment with an agentless framework on SWE-bench (Jimenez et al., 2024). This agentless pipeline is adapted from Xia et al. (2024), which splits the entire issue-solving task into two steps: 1) bug localization, and 2) patch prediction. As for bug localization, we prepare three settings to further simplify the evaluation, where the bug location comes from 1) model prediction, 2) the ground truth bug snippet, and 3) the ground truth file containing the bug. In this way, the trained model only needs to predict the target patch, based on different location implications. In Table 13, we report two metrics for each setting: 1) the similarity between the predicted patch and the oracle patch, and 2) the ratio of empty patch.

Table 13: Ablation study on different sample packing strategies and how to tackle long documents exceeding model input length. Model Loc: use model predicted position for bug localization; GT Loc: use ground truth bug location; GT File: use the complete file containing bug or target solution. ↑ means higher is better, while ↓ means lower is better.

Strategy	Tokens (B)	Fragm. Rate (%)	Padding Rate (%)	Model Loc (%)		GT Loc (%)		GT File (%)		AVG (%)	
				sim ↑	empty ↓	sim ↑	empty ↓	sim ↑	empty ↓	sim ↑	empty ↓
concat-then-split	73	30.2	0.00	16.61	30.61	24.02	26.60	9.41	59.60	16.68	38.94
restart-last-document	73	17.8	0.00	**17.92**	25.92	23.51	27.00	10.28	54.60	17.24	35.84
pad-last-document	89	0.0	17.55	17.24	**25.31**	22.99	25.60	10.35	**52.40**	16.86	**34.44**
best-fit-packing	73	0.0	0.01	17.22	28.37	**25.76**	**23.60**	**10.47**	56.80	**17.82**	36.26
best-fit-packing w/ split	73	0.0	0.01	17.95	22.04	27.46	**16.80**	15.10	36.20	20.17	25.01
w/ slide	73	0.0	0.01	**18.47**	**20.41**	27.50	18.60	14.49	36.40	20.15	25.14
w/ drop	73	0.0	0.01	17.85	21.43	**29.53**	18.00	**15.14**	**33.60**	**20.84**	**24.34**

The results above reveal three key insights: 1) eliminating fragmentation steadily improves performance. Best-fit-packing outperforms the traditional concat-then-split strategy in both patch similarity and empty rate. 2) BFP is more token-efficient than padding, achieving better results (17.82% vs 16.86%) with 22% fewer tokens. 3) handling extremely long documents is also non-negligible. Augmenting BFP with the "drop" strategy yields the best overall performance (20.84% similarity, 24.34% empty rate). This suggests that, rather than fancy algorithmic improvements, the best solution for handling long contexts may be to directly extend the model context length. We adopt the "split" strategy to handle extremely long documents throughout the main experiments.

A.4 Experiments on Cybersecurity

We view cybersecurity as a frontier direction for existing models, where they have yet to achieve human-expert level performance even on non-agentic tasks such as Cyber Threat Intelligence (CTI) analysis, vulnerability detection, and secure coding. To provide insight into future capabilities, we present the first comparative evaluation of Qwen3-Coder-Next against other frontier models in this domain.

Model	CKT (Acc)	ATE (Acc)	RCM (Acc)	RMS (F1)	VSP (Acc)	TAA (Acc)
Claude-Opus-4-5	94.33	85.50	71.25	68.39	53.88	29.00
Claude-sonnet-4-5	93.22	78.67	64.00	61.69	45.17	18.67
Deepseek-V3.2	90.00	60.00	65.50	29.02	20.50	13.00
GLM-4.7	85.67	66.00	70.50	19.45	29.00	12.00
Ours	85.00	44.00	58.50	5.50	24.50	8.00

Table 14: The ability of CTI analysis benchmarked by AthenaBench-Mini. We compute all the results with greedy decoding.

AthenaBench-Mini AthenaBench (Alam et al., 2025) is an updated benchmark of CTIBench (Alam et al., 2024) for CTI analysis evaluation. The benchmark includes 6 distinct tasks of CTI reasoning: (1) CTI Knowledge Test (CTK), (2) Root Cause Mapping (RCM), (3) Vulnerability Severity Prediction (VSP), (4) Threat Actor Attribution (TAA), (5) Risk Mitigation Strategy (RMS), and (6) Attack Technique Extraction (ATE). We use the public AthenaBench-Mini set and report the results in Table 14. The results suggest that Qwen3-Coder-Next achieve comparable performances to other open frontier models like Deepseek-V3.2 and GLM-4.7, but still fall short on the tasks related to root cause mapping and threat actor attribution. We plan to improve these tasks by adding more pre-training data on CTI analysis.

Model	Acc	Precision	Recall	F1	P-C ↓	P-V ↓	P-B ↓	P-R ↓
Claude-Opus-4-5	52.73	52.42	59.02	55.53	9.02	50.06	37.28	3.63
Claude-sonnet-4-5	52.57	51.44	91.80	65.93	8.52	83.38	4.93	3.17
Deepseek-V3.2	50.09	51.27	4.89	8.91	3.51	1.00	91.60	3.88
GLM-4.7	53.22	53.24	58.62	55.80	20.55	38.60	26.57	12.53
Ours	48.33	48.54	54.54	51.37	0.88	53.01	41.29	4.64

Table 15: The function-level vulnerability detection capability evaluated by PrimeVul-Paired. We note that Pair-wise Correct Prediction (P-C), Pair-wise Vulnerable Prediction (P-V), Pair-wise Benign Prediction (P-B), and Pair-wise Reversed Prediction (P-R) are used to evaluate the model's predictions on textually similar code pairs as single entities . We compute all the results with greedy decoding.

PrimeVul-Paired To evaluate how well the models perform on vulnerability detection, we use the Paired set of PrimeVul (Ding et al., 2024b). Unlike the full set having significantly imbalanced vulnerable and benign samples, the Paired set has balanced labels and consists of textually similar code pairs where one function is vulnerable and the other is benign. This design allows for more rigorous evaluation of the model's ability to distinguish subtle differences that determine vulnerability status. As shown in Table 15, Qwen3-Coder-Next achieves the lowest P-C score, indicating superior performance in correctly identifying both vulnerable and benign functions within similar code contexts. While our model shows competitive F1 score and overall performance, it demonstrates significantly better paired prediction consistency compared to all baselines.

Model	SecCodeBench				CWEval	
	Gen w/o Hint	Gen w/ Hint	Fix w/o Hint	Fix w/ Hint	func@1	func-sec@1
Claude-Opus-4-5	52.5	73.2	75.2	83.9	92.27	74.75
Claude-sonnet-4-5	43.4	57.6	62.2	76.4	91.55	71.72
Deepseek-V3.2	43.1	50.2	50.0	65.8	83.53	54.71
GLM-4.7	29.4	56.0	49.8	64.9	72.44	46.39
Ours	61.2	69.5	76.4	83.7	80.17	56.32

Table 16: The secure coding capability measured by SecCodeBench and CWEval. We note that SecCodeBench scores are derived from pass@k computation with security severity weighting, with (and without) the hints that ask models to behave securely. In addition, func@1 and func-sec@1 are computed based on the definition of pass@k, with the random sampling of $n = 10$ and temperature of 0.8.

SecCodeBench and CWEval We consider secure coding important in daily practice, where models help avoid and mitigate software vulnerabilities. Typically, there are two main scenarios: code generation and

vulnerability repair. We focus on two existing benchmarks: SecCodeBench[7] and CWEval (Peng et al., 2025). SecCodeBench consists of 53 Java coding tasks, the majority of which are derived from anonymized, real-world historical vulnerabilities at Alibaba, covering generation and repair scenarios both with and without security hints. CWEval is a multilingual secure code generation benchmark, with a focus on both functionality and security. From Table 16, we observe that Qwen3-Coder-Next achieves competitive security performance on SecCodeBench across both generation and repair tasks. Notably, it maintains high scores even without security hints, particularly outperforming Claude-Opus-4-5 on generation (61.2 vs. 52.5), indicating robust inherent security awareness in default code generation. For CWEval, we report func@1 and func-sec@1 for functionality and security, respectively. While Claude models achieve higher func@1 scores, Qwen3-Coder-Next demonstrates competitive secure code generation capability with func-sec@1 of 56.32%, outperforming Deepseek-V3.2 and GLM-4.7, indicating a reasonable balance between generating functional and secure code.

[7] https://github.com/alibaba/sec-code-bench