IQuest -Coder-V1 Technical Report

IQuest Coder Team

Abstract

In this report, we introduce the IQuest-Coder-V1 series-(7B/14B/40B/40B-Loop), a new family of code large language models (LLMs). Moving beyond static code representations, we propose the `code-flow` multi-stage training paradigm, which captures the dynamic evolution of software logic through different phases of the pipeline. Our models are developed through the evolutionary pipeline, starting with the intial pre-training consisting of code facts, repository, and completion data. Following that, we implement a specialized mid-training stage that integrates reasoning and agentic trajectories in 32k-context and repository-scale in 128k-context to forge deep logical foundations. The models are then finalized with post-training of specialized coding capabilities, which is bifurcated into two specialized paths: the thinking path (utilizing reasoning-driven RL) and the instruct path (optimized for general assistance). IQuest-Coder-V1 achieves state-of-the-art performance among competitive models across critical dimensions of code intelligence: agentic software engineering, competitive programming, and complex tool use. To address deployment constraints, the IQuest-Coder-V1-Loop variant introduces a recurrent mechanism designed to optimize the trade-off between model capacity and deployment footprint, offering an architecturally enhanced path for efficient efficacy-efficiency trade-off. We believe the release of the IQuest-Coder-V1 series, including the complete white-box chain of checkpoints from pre-training bases to the final thinking and instruct models, will advance research in autonomous code intelligence and real-world agentic systems.

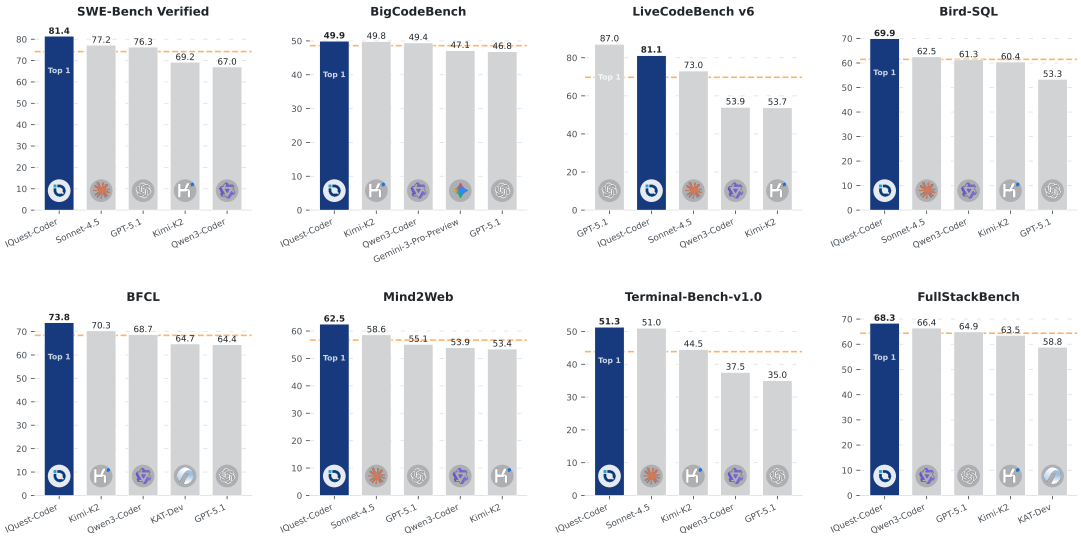

Figure 1. IQuest-Coder-V1 performance across different benchmarks. The score of LiveCodeBench v6 is from IQuest-Coder-V1-40B-Loop-Thinking model, and the rest are IQuest-Coder-V1-40B-Loop-Instruct model. The orange dash line represents the average score of the selected models.

1. Introduction

The current generation of large language models (LLMs) has demonstrated that general-purpose intelligence can be significantly amplified through domain-specific specialization [26]. However, in the field of code intelligence, a wide gap remains between open-weights models and proprietary leaders like Claude 4.5 Sonnet[1]. This gap is most evident in long-horizon reasoning and the ability to navigate complex, multi-file codebases [20]. We introduce **IQuest-Coder-V1** series, a family of dense models ranging from 7B to 40B parameters, built to close this gap by maximizing the intelligence density through a structured, multi-phase evolution of logic.

Our technical contributions are centered around a four-pillar **`Code-Flow`** pipeline (Figure 2):

- Pre-training & High-Quality Annealing: We begin with a two-stage pre-training process that transitions from stage-1 general data to stage-2 broad code data. This is followed by a targeted annealing phase using high-quality curated code, ensuring the model's base representations are primed for the complex logical tasks that follow.
- Dual-Phase Mid-training: To bridge the gap between static knowledge and agentic action, we introduce a dedicated mid-training stage with reasoning, agentic, and long-context coding data.
- Bifurcated Post-training: Recognizing that different use cases require different optimization profiles, we offer two distinct post-training paths focusing on instruction tuning and thinking paths.
- Efficient Architectures: Our loop model incorporates a recurrent structure to enable iterative computation over complex code segments, providing a scalable architectural path within the constraints of real-world deployment.

IQuest-Coder models are developed through a rigorous training methodology that combines large-scale pretraining on extensive code repositories with specialized instruction tuning. Our pretraining corpus encompasses billions of tokens from diverse sources, including public code repositories, technical documentation, and programming-related web content. We employ sophisticated data cleaning and filtering techniques to ensure high-quality training data, implementing both repository-level and file-level processing strategies to capture code structure and context effectively. The model series demonstrates three key characteristics: (1) Superior Performance: Our flagship IQuest-Coder-40B model achieves state-of-the-art results on major coding benchmarks, demonstrating competitive performance with leading proprietary models. (2) Comprehensive Coverage: With five distinct model sizes ranging from 3B to 40B parameters, IQuest-Coder addresses the diverse needs of the developer community, from resource-constrained edge deployment to high-performance cloud applications. (3) Balanced Capabilities: Beyond code generation, IQuest-Coder maintains strong performance in general language understanding and mathematical reasoning, making it suitable for multi-faceted development tasks.

Through our systematic exploration of the IQuest-Coder-V1 training pipeline, we identified several pivotal findings that offer a deeper understanding of how logical intelligence and agentic capabilities emerge within language models. These insights, derived from extensive ablations of our **`code-flow`** data and mid-training strategies, challenge several conventional assumptions in code LLM development:

- **Finding 1**: The repository transition data (the flow of commits) provides a superior signal for task planning compared to training on usual static snapshot files alone.

[1] https://www.anthropic.com/claude/sonnet

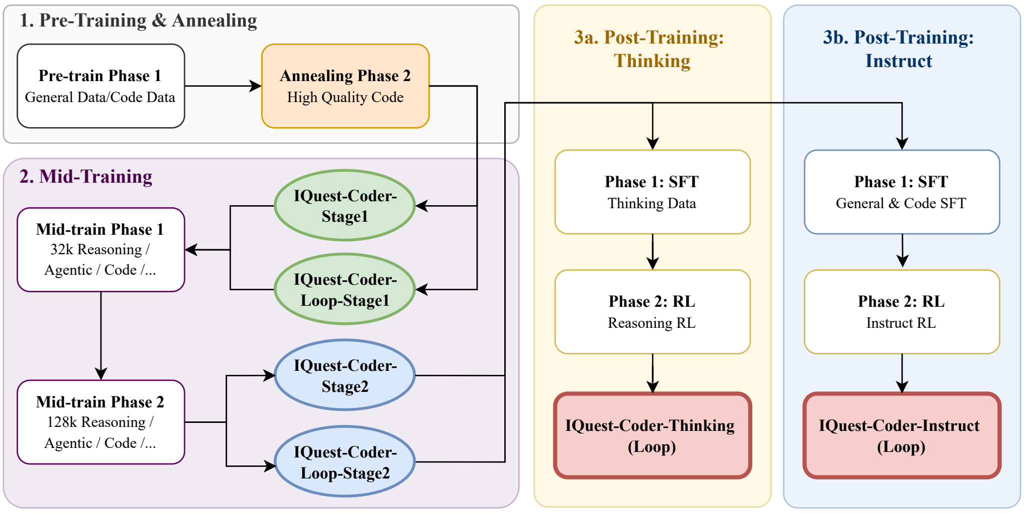

Figure 2. `Code-Flow` Training pipeline of IQuest-Coder-V1.

- **Finding 2**: Injecting 32k reasoning and agentic trajectories after high-quality code annealing—but before post-training—serves as a critical logical scaffold that stabilizes model performance under distribution shifts.
- **Finding 3**: The thinking path (utilizing RL) triggers an emergent ability for autonomous error-recovery in long-horizon tasks (e.g. SWE and code contest tasks) that is largely absent in standard Instruct SFT post-training paths.

Our post-training process leverages carefully curated datasets covering a wide spectrum of programming paradigms, languages, and real-world coding scenarios. This ensures that IQuest-Coder models can serve as effective coding assistants, capable of understanding complex requirements, generating robust solutions, and providing helpful explanations as revealed in Figure 1. We conduct extensive evaluations across popular benchmarks to validate the effectiveness of our approach, with results demonstrating significant improvements over existing open-source alternatives (*ref.* section 5). By releasing the complete evolutionary chain from stage 1 to the final post-training checkpoings, we provide a white-box resource for the community to study the forging of agentic code intelligence.

2. Model Architecture

2.1. LoopCoder

LoopCoder Architecture. The LoopCoder architecture employs a loop transformer design where transformer blocks with shared parameters are executed in two fixed iterations. In the first iteration, input embeddings are processed through transformer layers with position-shifted hidden states. During the second iteration, the model computes two types of attention: global attention (where queries from iteration 2 attend to all key-value pairs from iteration 1) and local attention (where queries attend only to preceding tokens within iteration 2 to maintain causality). These two attention outputs are combined using a learned gating mechanism based on query representations, with the gate controlling the weighted mixture of global context refinement and local causal dependencies. This approach differs from the original Parallel Loop Transformer by omitting token-shifting mechanisms and inference-specific optimizations.

LoopCoder Training. The training pipeline for LoopCoder consists of three main stages, as illustrated in Figure 2.

Model Size	Layers	Hidden Size	Intermediate Size	Attention	Max Context	Query Heads	KV Heads	Vocabulary
Base Models (Stage 1)								
IQuest-Coder-V1-7B-Stage1	14	5120	27648	GQA	131072	40	8	76800
IQuest-Coder-V1-14B-Stage1	28	5120	27648	GQA	131072	40	8	76800
IQuest-Coder-V1-40B-Stage1	80	5120	27648	GQA	131072	40	8	76800
IQuest-Coder-V1-40B-Loop-Stage1 (LoopCoder-Base-Stage1)	80	5120	27648	GQA	131072	40	8	76800
Base Models (Stage 2)								
IQuest-Coder-V1-7B-Stage2	14	5120	27648	GQA	131072	40	8	76800
IQuest-Coder-V1-14B-Stage2	28	5120	27648	GQA	131072	40	8	76800
IQuest-Coder-V1-40B-Stage2	80	5120	27648	GQA	131072	40	8	76800
IQuest-Coder-V1-40B-Loop-Stage2 (LoopCoder-Base-Stage2)	80	5120	27648	GQA	131072	40	8	76800
Instruct Models								
IQuest-Coder-V1-7B-Instruct	14	5120	27648	GQA	131072	40	8	76800
IQuest-Coder-V1-14B-Instruct	28	5120	27648	GQA	131072	40	8	76800
IQuest-Coder-V1-40B-Instruct	80	5120	27648	GQA	131072	40	8	76800
IQuest-Coder-V1-40B-Loop-Instruct (LoopCoder-Instruct)	80	5120	27648	GQA	131072	40	8	76800
Thinking Models								
IQuest-Coder-V1-7B-Thinking	14	5120	27648	GQA	131072	40	8	76800
IQuest-Coder-V1-14B-Thinking	28	5120	27648	GQA	131072	40	8	76800
IQuest-Coder-V1-40B-Thinking	80	5120	27648	GQA	131072	40	8	76800
IQuest-Coder-V1-40B-Loop-Thinking (LoopCoder-Thinking)	80	5120	27648	GQA	131072	40	8	76800

Table 1. Architecture of IQuest-Coder-V1.

Stage 1: Pre-Training & Annealing. The training begins with a pre-training phase using a mixture of general data and code data, followed by an annealing phase that focuses on high-quality code corpora. This stage establishes the foundational language understanding and code generation capabilities of the model.

Stage 2: Mid-Training. The mid-training stage is divided into two phases with progressively increasing context lengths. In Mid-train Phase 1, we train the model on 32k context data comprising reasoning, agentic, and code tasks, yielding IQuest-Coder-V1-Stage1 and LoopCoder-Base-Stage1. In Mid-train Phase 2, we further extend the context length to 128k and continue training on similar data distributions. This phase produces IQuest-Coder-V1-Stage2 and LoopCoder-Base-Stage2, which serve as the base models for subsequent post-training.

Stage 3: Post-Training. We develop two variants of LoopCoder through distinct post-training recipes:

- **Thinking Models:** We first perform supervised fine-tuning (SFT) on thinking data that contains explicit reasoning traces, followed by reinforcement learning (RL) optimized for reasoning capabilities. This yields LoopCoder-Thinking.
- **Instruct Models:** We apply SFT on general and code instruction-following data, then conduct RL training to enhance instruction-following abilities. This produces LoopCoder-Instruct.

2.2. Infra for LoopCoder

This document describes the three-stage training methodology and infrastructure from LoopCoder. The training progresses from (1) pre-training on general and code data with annealing on high-quality code, to (2) mid-training with progressively longer contexts (32k then 128k) on reasoning, agentic, and code tasks, and finally (3) post-training via two pathways—SFT and RL for either thinking models (with explicit reasoning) or instruct models (for instruction-following). Supporting this multi-million GPU-hour training effort, the infrastructure prioritizes computational efficiency through fused gated attention kernels that reduce memory bandwidth overhead, context parallelism that enables ultra-long context training via point-to-point KV

Language	Python	Java	JavaScript	TypeScript	C#	Go	Rust
Python	0.75	0.76 (↑**1.36%**)	0.77 (↓**1.12%**)	0.74 (↓**0.95%**)	0.77 (↓**1.69%**)	0.76 (↓**2.13%**)	0.77 (↓**2.72%**)
Java	0.85 (↑**6.02%**)	0.79	0.79 (↑**12.62%**)	0.90 (↑**12.08%**)	0.81 (↑**20.58%**)	0.79 (↑**10.68%**)	0.72 (↑**12.41%**)
JavaScript	0.51 (↑**5.49%**)	0.52 (↑**2.98%**)	0.53	0.53 (↑**4.69%**)	0.54 (↑**2.44%**)	0.54 (↑**1.34%**)	0.53 (↑**2.56%**)
TypeScript	0.51 (↑**4.17%**)	0.53 (↑**2.29%**)	0.53 (↑**3.34%**)	0.52	0.53 (↑**1.68%**)	0.52 (↑**1.39%**)	0.53 (↑**1.18%**)
C#	0.33 (↑**3.84%**)	0.33 (↑**1.93%**)	0.34 (↑**3.10%**)	0.34 (↑**3.71%**)	0.34	0.34 (↑**1.87%**)	0.35 (↑**1.98%**)
Go	0.41 (↑**4.77%**)	0.41 (↑**2.95%**)	0.42 (↑**2.70%**)	0.42 (↑**4.41%**)	0.43 (↑**2.95%**)	0.42	0.42 (↑**2.86%**)
Rust	0.38 (↑**3.87%**)	0.38 (↑**2.89%**)	0.40 (↑**4.20%**)	0.38 (↑**4.05%**)	0.38 (↑**2.81%**)	0.38 (↑**2.80%**)	0.38

Table 2. Synergy gain matrix (Reordered). Values indicate absolute performance, while parentheses show relative improvement vs baseline. **Bold numbers** indicate the percentage change. Background intensity indicates magnitude (Darker Red = Higher Gain).

shard transmission with reduced memory costs, and reliability through silent error detection using deterministic re-computation and tensor fingerprint validation to catch hardware failures that don't trigger explicit exceptions.

3. Pre-training

3.1. Data Mixtrue and Code Scaling Law

We adopt the pre-training guideline [24] for the code pre-training, which has direct implications for constructing multilingual code corpora. When training tokens are limited, prioritizing mixing syntactically-related PLs can further bring more improvement compared to naively upsampling a single PL. The positive synergy effects suggest that linguistic diversity, particularly when it spans across the code domain, acts as a form of data augmentation that improves model robustness. For realistic multilingual pre-training, a mixed-language training regime is superior to language-specific fine-tuning. Traditional scaling laws treat multilingual code as homogeneous, but PLs contribute differently to performance. We extend this by incorporating language proportions $p = (p_1, \ldots, p_K)$ explicitly:

$$\mathcal{L}(N, D; p) = A \cdot N^{-\alpha_N(p)} + B \cdot D_x^{-\alpha_D(p)} + L_\infty(p) \tag{1}$$

where $\alpha_N(p) = \sum_k p_k \alpha_N^k$, $\alpha_D(p) = \sum_k p_k \alpha_D^k$, and $L_\infty(p) = \sum_p pL_\infty^k$ are proportion-weighted averages of language-specific parameters from Figure 3. The effective data term captures the effects of the cross-lingual transfer:

$$D_x = D_{all}\left(1 + \gamma \sum_{L_i \neq L_j} p_{L_i} p_{L_j} \tau_{ij}\right) \tag{2}$$

where τ_{ij} is the transfer coefficient derived from Table 2. We can get the final scaling law:

$$\mathcal{L}^*(N, D) = A^* \cdot N^{-\alpha_N^*} + B^* \cdot D^{-\alpha_D^*} + L_\infty^* \tag{3}$$

where $\alpha_D^* = 0.6859$, $\alpha_N^* = 0.2186$, $L_\infty^* = 0.2025$ are the fitted parameters under the optimal multilingual allocation for the multilingual code generation and translation at the same time.

3.2. Stage1: General Pre-training

General Corpus Proccessing To construct the foundational corpus for IQuest-Coder, we curated a massive dataset primarily sourced from Common Crawl[2]. Our pre-processing pipeline begins with a rigorous cleaning stage utilizing regular expressions to remove low-quality noise and non-informative fragments. We ensure data integrity through a hierarchical deduplication

[2]https://commoncrawl.org/

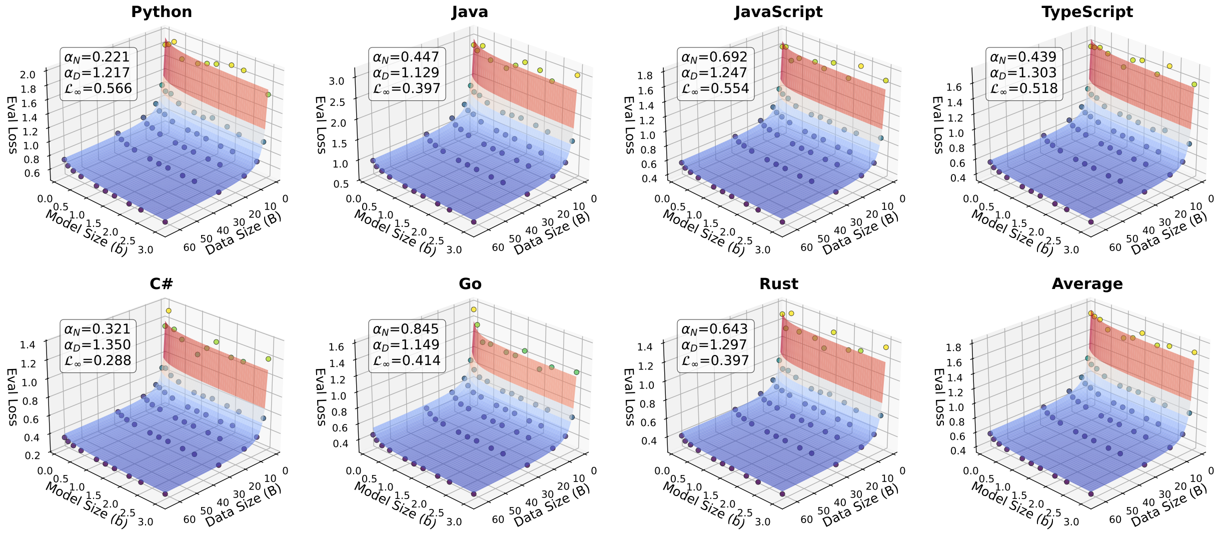

Figure 3. Scaling Laws for each PL independently. It shows a clear ordering of intrinsic predictability across PLs: C# < Java ≈ Rust < Go < TypeScript < JavaScript < Python.

strategy, combining exact match filtering with fuzzy deduplication driven by high-dimensional embedding models. To safeguard the validity of our evaluations, a comprehensive decontamination procedure is implemented to eliminate any overlaps with common benchmarks. For programming data retrieved from Common Crawl, we perform deep Abstract Syntax Tree (AST) analysis to verify syntactic structure and structural integrity, a critical step for our code-flow training paradigm. To scale quality control, we train a suite of domain-specific proxy classifiers specialized for general text, code, and mathematics. These proxies are designed to emulate the quality assessment capabilities of much larger models, which provide annotation samples across dimensions such as information density, educational value, and toxic content. Empirical results on validation sets confirm that these small proxy models outperform traditional FastText-based approaches, providing a far more precise signal for selecting high-utility tokens. To enhance the code-related factuality of LLM, we use CodeSimpleQA-Instruct [25], a large-scale instruction corpus with 66 million samples, into the pre-training stage. LLMs are adopted to automatically generate factual question-answer pairs from each cluster through a structured pipeline that incorporates explicit constraints to ensure questions are objective, unambiguous, and time-invariant with single correct answers. This approach produces high-quality, objective technical assessments suitable for knowledge evaluation platforms while ensuring time-invariant accuracy and requiring minimal ongoing maintenance.

To construct a dataset suitable for learning repository evolution patterns, we design a triplet construction strategy based on project lifecycle. For each code repository, the system constructs triplets of the form $(\mathcal{R}_{old}, \mathcal{P}, \mathcal{R}_{new})$, where $\mathcal{R}_{old}$ represents the project's code state at a stable development phase, $\mathcal{P}$ denotes the patch information capturing differences between two code states, and $\mathcal{R}_{new}$ represents the code state after a series of development iterations. The starting point selection follows a *project maturity principle*: commits are selected within the 40%-80% percentile range of the project lifecycle. This interval corresponds to the mature development phase of the project, where the codebase is relatively stable, avoiding both the uncertainty of early development and the fragmented changes typical of late-stage maintenance. This approach ensures that training data reflects authentic software development patterns. Based on the selected starting point, the system searches forward for appropriate endpoint commits to form complete triplets. The search strategy considers the quality and representativeness of code changes, ensuring that each triplet captures meaningful development iteration processes. This construction method generates training data that maintains the temporal continuity of code evolution while ensuring data diversity and information density, providing a theoretically sound foundational dataset for LLM to learn complex code transformation patterns.

Code Completion Code completion is a fundamental capability of code intelligence. This proficiency is primarily enhanced by training on data constructed in the Fill-In-the-Middle (FIM) [1] format. In the FIM paradigm, a code document is partitioned into three segments: prefix, middle, and suffix. The training objective is to predict the middle content based on the provided prefix and suffix. File-level FIM focuses on individual documents, where the segments are concatenated for training with Fill-In-the-Middle (FIM) pattern. Furthermore, Repo-level FIM extends this approach by incorporating semantically similar code snippets from the same repository as additional context to assist in predicting the middle segment. We primarily employ two strategies for code completion data construction: heuristic-based and multi-level syntax-based construction [23].

The heuristic-based approach consists of two techniques: random boundary splitting and random line splitting. Random boundary splitting partitions code documents at a character-level granularity, which enhances the model's generalization and improves its performance in generating large code blocks or continuing from specific characters. In contrast, random line splitting selects a specific line within the document as the target for completion, which better aligns with typical user interaction patterns. The syntax-based approach leverages the inherent structural properties of source code. By utilizing abstract syntax tree (AST) representations, we extract code segments from various nodes with different characteristics. This method ensures both the randomness of the training data and the structural integrity of the code. We implement several hierarchical levels, including expression-level, statement-level, and function-level. Based on these nodes, we construct multiple PLs and multi-level completion data for both file-level and repo-level tasks, significantly enhancing the diversity of the training samples.The task structure for file-level completion is `<|fim_prefix|>{code_pre}<|fim_suffix|>{code_suf}<|fim_middle|>{code_mid}<|im_end|>` and the task structure for repository-level completion is `<|repo_name|>{repo_name}` `<|file_sep|>{file_path1} {file_content1} <|file_sep|>{file_path2} {file_content2}` `<|file_sep|>{file_path3} <|fim_prefix|>{code_pre}<|fim_suffix|>{code_suf}` `<|fim_middle|>{code_fim}<|im_end|>`

3.3. Stage2: Mid-Training

This mid-training process uses a two-stage approach (Stage 2.1 at 32K context and Stage 2.2 at 128K context) to efficiently scale model capabilities while managing computational costs. Both stages train on the same core data categories: Reasoning QA (math, coding, logic), Agent trajectories, code commits, and file/repository-level fill-in-the-middle (FIM) data. The Reasoning QA component acts as a "reasoning runtime" that encourages structured problem decomposition and consistency checking rather than simple pattern matching, while Agent trajectory data teaches "closed-loop intelligence" by exposing the model to complete action-observation-revision cycles with dense environmental feedback (commands, logs, errors, test results). This combination provides both symbolic reasoning scaffolding and grounded "code world" experience, enabling the model to handle long-horizon tasks, recover from errors, and maintain coherent plans across extended contexts, with Stage 2.2 specifically extending these capabilities to repository-level reasoning by incorporating dedicated 128K sequence length samples.

4. Post-Training

Post-training transforms pre-trained models into specialized code intelligence systems through supervised fine-tuning and reinforcement learning. This phase uses instructional data spanning code engineering, mathematics, agentic capabilities, and conversation, employing model-in-the-loop synthesis with execution-based verification.

4.1. Data Construction

We employ a model-centric framework where frontier LLMs generate training data under rigorous automated verification, using deterministic execution-based validation for objective domains and ensemble mechanisms combining rule-based checks, reward models, and multi-agent debate for subjective domains. Our methodology spans API orchestration, full-stack engineering, competitive programming, code reasoning, text-to-SQL, code editing, terminal benchmarking, repository-scale engineering, tool use, and GUI agents, synthesizing data through techniques like stochastic perturbations, test-driven synthesis, reverse pipeline generation, and multi-stage filtering with automated environment construction. This is followed by large-scale supervised fine-tuning that processes token counts near pre-training scale to inject dense task-specific knowledge, utilizing optimization infrastructure such as aggressive sequence packing, conservative cosine annealing learning rates, and a three-phase curriculum that sequences data by difficulty to ensure stable convergence and superior performance on complex benchmarks.

4.2. Large-Scale Supervised Fine-Tuning

Post-training processes match pre-training scale to inject specialized knowledge through optimized infrastructure, including sequence packing with cross-sample masking, cosine learning rate schedules with extended low-rate phases, and three-phase curriculum learning progressing from basic instruction-following to adversarial examples. Quality control ensures only verified samples enter training through comprehensive sandboxed execution, capturing traces and metrics, symbolic mathematical verification, multi-agent debate for subjective evaluation, and aggressive contamination prevention via n-gram matching and MinHash LSH deduplication, prioritizing quality over quantity for improved generalization on complex benchmarks.

4.3. Multi-Objective Optimization

This section includes three main components: (1) Alignment tax mitigation through replay buffers, dynamic mixture adaptation, and compositional design to preserve general capabilities while specializing; (2) Reinforcement learning from verifiable feedback using GRPO algorithm with clip-Higher strategy on competition coding tasks, trained on test case pass rates without KL penalties; and (3) SWE-RL framework built on scalable cloud-based sandbox infrastructure that formulates real-world software engineering as interactive RL environments, where agents use tool-based actions across multiple steps and are trained via GRPO with rewards based on test suite passage plus regularization for efficiency, enabling parallel trajectory execution for stable long-horizon code reasoning and debugging capabilities—together yielding emergent capabilities like self-debugging, cross-language transfer, and improved uncertainty calibration.

5. Evaluation

5.1. Baselines

In our evaluation, we compare our model against a broad set of state-of-the-art code-focused language models covering instruction-tuned, base, and reasoning-enhanced variants. The baselines span leading closed-source and open-source systems known for strong performance on

Model	Python		Java		TypeScript		C#		Average	
	EM	ES	EM	ES	EM	ES	EM	ES	EM	ES
6B+ Models										
DeepSeek-Coder-6.7B-Base	41.1	79.2	39.9	80.1	46.3	82.4	55.0	86.9	45.6	82.1
DS-Coder-V2-Lite-Base	41.8	78.3	46.1	81.2	44.6	81.4	58.7	87.9	47.8	82.2
CodeQwen1.5-7B	40.7	77.8	47.0	81.6	45.8	82.2	59.7	87.6	48.3	82.3
Qwen2.5-Coder-7B	42.4	78.6	48.1	82.6	46.8	83.4	59.7	87.9	49.3	83.1
StarCoder2-7B	10.9	63.1	8.3	71.0	6.7	76.8	7.3	72.1	8.3	70.8
14B+ Models										
Qwen2.5-Coder-14B	47.7	81.7	54.7	85.7	52.9	86.0	66.4	91.1	55.4	86.1
StarCoder2-15B	28.2	70.5	26.7	71.0	24.7	76.3	25.2	74.2	26.2	73.0
20B+ Models										
DS-Coder-33B-Base	44.2	80.4	46.5	82.7	49.2	84.0	55.2	87.8	48.8	83.7
Qwen2.5-Coder-32B	49.2	82.1	56.4	**86.6**	54.9	87.0	**68.0**	**91.6**	57.1	**86.8**
CodeStral-22B	**49.3**	**82.7**	44.1	71.1	51.0	85.0	53.7	83.6	49.5	80.6
IQuest-Coder-V1-40B	49.0	81.7	**57.9**	86.2	**61.9**	**88.5**	63.4	85.5	**57.8**	85.7

Table 3. Performance comparison on CrossCodeEval Tasks.

Model	EvalPlus				BigCodeBench		FullStackBench
	HumanEval	HumanEval+	MBPP	MBPP+	Full	Hard	
6B+ Models							
DeepSeek-Coder-V2-Lite-Instruct	81.1	75.6	85.2	70.6	37.8	18.9	49.4
Qwen2.5-Coder-7B-Instruct	87.2	81.7	84.7	72.2	37.8	13.5	42.2
Seed-Coder-8B-Instruct	81.1	75.6	86.2	73.3	44.6	23.6	55.8
13B+ Models							
Qwen2.5-Coder-14B-Instruct	62.8	59.8	88.6	77.2	47.0	6.1	53.1
Qwen3-Coder-30B-A3B-Instruct	93.9	87.2	90.7	77.2	46.9	27.7	60.9
20B+ Models							
Deepseek-V3.2	93.9	88.4	93.4	77.2	48.1	27.0	64.9
Qwen2.5-Coder-32B-Instruct	93.3	86.6	90.2	77.8	48.0	24.3	57.4
Qwen3-235B-A22B-Instruct-2507	96.3	91.5	92.3	77.8	47.4	25.7	62.7
Qwen3-235B-A22B-Thinking-2507	**98.8**	**93.3**	95.5	81.5	44.1	23.0	-
Qwen3-Coder-480B-A35B-Instruct	97.6	92.7	94.2	80.2	49.4	27.7	66.4
Kimi-Dev-72B	93.3	86.0	79.6	68.8	45.4	31.8	38.6
Kimi-K2-Instruct-0905	94.5	89.6	91.8	74.1	49.8	30.4	63.5
Kimi-K2-Thinking	98.2	92.7	**97.4**	**82.3**	46.8	28.4	-
KAT-Dev	90.9	86.6	89.4	76.2	46.2	25.7	58.8
KAT-Dev-72B-Exp	88.4	81.7	85.2	69.3	48.3	26.4	52.9
GLM-4.7	87.2	79.9	90.5	75.7	45.7	26.4	70.2
IQuest-Coder-V1-40B-Instruct	96.3	90.2	91.8	77.8	**54.2**	**33.1**	**71.4**
IQuest-Coder-V1-40B-Loop-Instruct	97.6	91.5	92.9	77.2	49.9	27.7	68.3
Closed-APIs Models							
Gemini-3-Flash-preview	88.4	84.8	92.3	79.1	44.5	25.6	-
Gemini-3-Pro-preview	100.0	94.5	71.2	64.8	47.1	25.0	-
Claude-Opus-4.5	98.8	93.3	96.8	83.9	53.3	35.1	72.3
Claude-Sonnet-4.5	98.8	93.3	95.2	82.3	51.4	29.1	69.7
GPT-5.1	97.0	90.0	92.6	72.2	46.8	29.1	64.9

Table 4. Performance comparison on code generation tasks.

programming and reasoning tasks, including representative models from Anthropic (Claude 4.5), OpenAI (GPT-5.1), Google (Gemini 3), Alibaba (Qwen and Qwen-Coder series), DeepSeek (Coder and V3 series), Mistral (CodeStral), Moonshot (Kimi), ZhiPu (GLM), Kuaishou (Kwaipilot/KAT), and BigCode (StarCoder2). These models cover a wide parameter range and different tuning strategies, ensuring that our comparison reflects current capability boundaries in code generation, understanding, and complex task execution.

Model	**CruxEval**		**LiveCodeBench**	
	Input-COT	**Output-COT**	**V5**	**V6**
6B+ Models				
DeepSeek-Coder-V2-Lite-Instruct	57.1	56.2	13.2	19.4
Qwen2.5-Coder-7B-Instruct	66.9	66.0	14.4	18.9
Seed-Coder-8B-Instruct	62.0	66.6	19.2	22.3
13B+ Models				
Qwen2.5-Coder-14B-Instruct	75.6	79.2	22.8	24.6
Qwen3-Coder-30B-A3B-Instruct	76.9	80.5	43.1	36.0
20B+ Models				
DeepSeek-v3.2	82.1	94.2	-	83.3
Qwen2.5-Coder-32B-Instruct	78.8	84.0	30.5	27.4
Qwen3-235B-A22B-Instruct-2507	62.0	89.5	53.9	51.8
Qwen3-235B-A22B-Thinking-2507	15.2	46.9	80.2	74.1
Qwen3-Coder-480B-A35B-Instruct	87.1	90.4	48.6	53.9
Kimi-Dev-72B	33.0	64.2	46.1	40.0
Kimi-K2-Instruct-0905	86.8	89.5	52.1	53.7
Kimi-K2-Thinking	92.2	86.2	-	83.1
KAT-Dev	42.5	65.1	32.9	32.6
KAT-Dev-72B-Exp	71.4	81.1	13.8	16.0
GLM-4.7	65.6	81.2	-	**84.9**
IQuest-Coder-V1-40B-Instruct	93.5	87.0	55.7	46.9
IQuest-Coder-V1-40B-Thinking	97.9	98.9	83.8	80.5
IQuest-Coder-V1-40B-Loop-Instruct	91.1	85.5	48.6	48.5
IQuest-Coder-V1-40B-Loop-Thinking	**98.5**	**99.4**	**86.2**	81.1
Closed-APIs Models				
Gemini-3-Flash-preview	96.5	97.6	-	90.8
Gemini-3-Pro-preview	98.8	99.1	-	91.7
Claude-Opus-4.5	98.4	98.0	-	87.1
Claude-Sonnet-4.5	96.2	96.2	-	73.0
GPT-5.1	70.8	71.1	-	87.0

Table 5. Performance comparison on Code Reasoning Evaluation.

Model	**Aider-Polyglot-diff** **Pass@2**	**Aider-Polyglot-whole** **Pass@2**
6B+ Models		
DeepSeek-Coder-V2-Lite-Instruct	1.3	2.2
Qwen2.5-Coder-7B-Instruct	1.8	4.9
Seed-Coder-8B-Instruct	6.2	5.3
13B+ Models		
Qwen2.5-Coder-14B-Instruct	8.0	8.0
Qwen3-Coder-30B-A3B-Instruct	28.4	29.3
20B+ Models		
Qwen2.5-Coder-32B-Instruct	8.4	14.7
Qwen3-235B-A22B-Instruct-2507	53.3	57.3
Qwen3-235B-A22B-Thinking-2507	25.3	-
Qwen3-Coder-480B-A35B-Instruct	57.8	61.8
Kimi-Dev-72B	12.0	20.0
Kimi-K2-Instruct-0905	60.0	50.7
KAT-Dev	8.9	34.2
KAT-Dev-72B-Exp	16.4	15.6
IQuest-Coder-V1-40B-Instruct	**82.7**	**80.8**
IQuest-Coder-V1-40B-Loop-Instruct	68.9	62.3
Closed-APIs Models		
Gemini-3-Pro-preview	91.9	92.9
Claude-Opus-4.5	89.4	87.1
Claude-Sonnet-4.5	78.8	-
GPT-5.1	63.1	65.3

Table 6. Performance comparison on code editing task.

5.2. Experiments on Base Models

5.2.1. Code Completion

We evaluate cross-file code completion on CrossCodeEval [3], a multilingual benchmark encompassing Python, Java, TypeScript, and C#. This benchmark explicitly targets repository-level completion scenarios, serving as a core metric for assessing the fundamental capabilities of code LLMs in leveraging cross-file context.

5.3. Evaluation on Instruct Models and Reasoning model

5.3.1. Code Generation

Across a wide range of code-generation evaluations, our model achieves consistently strong performance. We validate functional correctness and robustness using EvalPlus [13] (including HumanEval+ and MBPP+ with substantially expanded test suites), and measure compositional, library-intensive problem solving on BigCodeBench [29]. We further demonstrate broad full-stack capability on FullStackBench [14], and strong results under contamination-aware, continuously refreshed testing on LiveCodeBench [8].

5.3.2. *Code Reasoning*

We further evaluate code reasoning with CRUXEval [5], which tests both forward execution (Input-to-Output, I2O) and inverse inference (Output-to-Input, O2I) over 800 concise Python functions. Our model performs strongly on I2O and also shows clear gains on the more challenging O2I setting, indicating improved ability to reason about code behavior beyond surface-level execution and to solve inverse constraints implied by a target return value.

Model	**Mercury**	
	Beyond@1	**Pass@1**
6B+ Models		
DeepSeek-Coder-V2-Lite-Instruct	76.8	91.4
Qwen2.5-Coder-7B-Instruct	69.9	84.8
Seed-Coder-8B-Instruct	78.5	93.8
13B+ Models		
Qwen2.5-Coder-14B-Instruct	76.7	88.3
Qwen3-Coder-30B-A3B-Instruct	81.1	95.3
20B+ Models		
DeepSeek-v3.2	81.6	**96.9**
Qwen2.5-Coder-32B-Instruct	79.1	96.1
Qwen3-235B-A22B-Instruct-2507	80.4	**96.9**
Qwen3-235B-A22B-Thinking-2507	61.2	70.3
Qwen3-Coder-480B-A35B-Instruct	80.2	96.1
Kimi-Dev-72B	59.1	69.5
Kimi-K2-Instruct-0905	76.1	90.6
Kimi-K2-Thinking	73.0	85.2
KAT-Dev	75.1	89.1
KAT-Dev-72B-Exp	79.0	94.5
GLM-4.7	74.1	86.7
IQuest-Coder-V1-40B-Instruct	**83.6**	95.3
IQuest-Coder-V1-40B-Loop-Instruct	82.2	94.1
Closed-APIs Models		
Gemini-3-Flash-preview	78.4	89.5
Gemini-3-Pro-preview	83.1	96.1
Claude-Opus-4.5	82.9	96.9
Claude-Sonnet-4.5	82.5	97.7
GPT-5.1	81.9	96.1

Table 7. Performance comparison on code efficiency task.

5.3.3. *Code Editing*

We evaluate code editing on Aider's Polyglot benchmark [17], which extends Aider's original editing setup from Python-only to a multilingual suite built on Exercism exercises. It covers C++, Go, Java, JavaScript, Python, and Rust, and concentrates on the hardest 225 problems selected from 697 available exercises across these languages, providing a challenging test of multi-language patch generation and iterative code refinement.

5.3.4. *Code Efficiency*

We assess code efficiency with Mercury [4], which evaluates Code LLMs beyond functional correctness by measuring runtime on natural-language-to-code tasks. Mercury contains 256 Python problems across multiple difficulty levels, each with a test-case generator and a set of real-world reference solutions that together define an empirical runtime distribution per task. The benchmark further proposes the percentile-based *Beyond* metric, which reweights Pass by relative runtime to jointly capture correctness and efficiency. Our model achieves strong Mercury results, indicating that it can produce solutions that are not only correct but also competitive in runtime under this distribution-based evaluation.

Model	Bird Execution Accuracy	Spider Execution Accuracy
6B+ Models		
DeepSeek-Coder-V2-Lite-Instruct	41.6	72.4
Qwen2.5-Coder-7B-Instruct	53.1	79.8
Seed-Coder-8B-Instruct	44.7	72.7
13B+ Models		
Qwen2.5-Coder-14B-Instruct	59.1	81.3
Qwen3-Coder-30B-A3B-Instruct	59.0	80.9
20B+ Models		
DeepSeek-v3.2	52.6	77.9
Qwen2.5-Coder-32B-Instruct	62.1	83.9
Qwen3-235B-A22B-Instruct-2507	62.8	81.1
Qwen3-235B-A22B-Thinking-2507	35.2	42.6
Qwen3-Coder-480B-A35B-Instruct	61.3	81.2
Kimi-K2-Instruct-0905	60.4	81.1
Kimi-K2-Thinking	40.6	49.6
KAT-Dev	52.2	77.6
KAT-Dev-72B-Exp	35.2	60.3
GLM-4.7	46.5	62.4
IQuest-Coder-V1-40B-Instruct	**70.5**	**92.2**
IQuest-Coder-V1-40B-Loop-Instruct	69.9	84.0
Closed-APIs Models		
Gemini-3-Flash-preview	66.6	87.2
Gemini-3-Pro-preview	67.5	87.0
Claude-Opus-4.5	66.0	76.0
Claude-Sonnet-4.5	62.5	80.1
GPT-5.1	53.3	77.6

Table 8. Performance comparison on Text2SQL Tasks.

5.3.5. *Text to SQL*

Our model also performs strongly on cross-domain Text-to-SQL benchmarks that stress generalization to unseen schemas and realistic database settings. On Spider [28], which uses a database-level train–test split to evaluate schema linking and structurally correct SQL generation with complex constructs, and on BIRD [12], which further emphasizes value grounding from database contents, real-world database scale, and execution-related practicality, our model achieves competitive results, indicating robust semantic parsing and reliable query generation in both schema-centric and content-grounded scenarios.

5.3.6. *Agentic Coding Tasks*

We further evaluate our model in agentic, end-to-end software workflows where success depends on correct tool use, long-horizon planning, and tight interaction with the execution environment. Terminal-Bench [21] measures whether an agent can reliably complete realistic terminal workflows (for example, building software from source, configuring services, managing dependencies, and debugging) inside containerized sandboxes with automated verification, while also standardizing execution via its runner for reproducible leaderboard evaluation. In parallel, SWE-bench [10] targets real-world software engineering by requiring models to produce patches from issue descriptions that turn failing repositories into passing ones under unit-test verification; SWE-bench Verified further improves reliability with 500 curated instances evaluated in a standardized Docker environment, where our model achieves a score of 81.4.

5.3.7. *Other Agentic Tasks*

Beyond coding-centric agents, we additionally evaluate general tool-use and interactive decision making across web, API, and conversational-agent settings. Mind2Web [2] targets generalist web agents that must follow natural-language instructions to complete open-ended tasks on real

Model	Agentic Coding			General Tool Use	
	Terminal-Bench	Terminal-Bench (2.0)	SWE-Verified	Mind2Web	BFCL V3
6B+ Models					
DeepSeek-Coder-V2-Lite-Instruct	5.0	0.0	-	26.7	-
Qwen2.5-Coder-7B-Instruct	6.3	0.0	-	38.4	54.2
Seed-Coder-8B-Instruct	7.5	2.5	-	38.2	-
13B+ Models					
Qwen2.5-Coder-14B-Instruct	8.8	0.0	-	42.7	59.9
Qwen3-Coder-30B-A3B-Instruct	23.8	23.8	51.9	36.1	63.4
20B+ Models					
DeepSeek-v3.2	23.8	**46.4**	73.1	47.2	68.8
Qwen2.5-Coder-32B-Instruct	5.0	4.5	-	32.5	62.3
Qwen3-235B-A22B-Instruct-2507	15.0	13.5	45.2	49.0	71.2
Qwen3-235B-A22B-Thinking-2507	8.8	3.4	44.6	43.2	71.9
Qwen3-Coder-480B-A35B-Instruct	37.5	23.6	67.0	54.0	68.7
Kimi-Dev-72B	-	2.3	60.4	-	55.5
Kimi-K2-Instruct-0905	44.5	27.8	69.2	53.4	70.3
Kimi-K2-Thinking	47.1	33.7	71.3	55.7	-
KAT-Dev	17.5	10.1	62.4	33.7	64.7
KAT-Dev-72B-Exp	21.3	7.9	74.6	-	-
GLM-4.7	36.3	41.0	73.8	53.7	64.8
IQuest-Coder-V1-40B-Instruct	**52.5**	33.0	75.2	**64.3**	51.7
IQuest-Coder-V1-40B-Loop-Instruct	51.3	33.0	**81.4**	62.5	**73.9**
Closed-APIs Models					
Gemini-3-Flash-preview	53.8	47.6	78.0	60.6	-
Gemini-3-Pro-preview	46.3	54.2	76.2	60.3	78.2
Claude-Opus-4.5	47.5	59.3	80.9	57.9	78.9
Claude-Sonnet-4.5	51.0	50.0	77.2	58.6	77.7
GPT-5.1	35.0	47.6	76.3	55.1	64.4

Table 9. Combined performance on agentic coding tasks (Terminal-Bench, Terminal-Bench 2.0, SWE-Verified) and general tool-use tasks (Mind2Web, BFCL V3).

websites, stressing cross-site generalization and long-horizon UI interaction. BFCL [16] tests tool-use across heterogeneous programming and API settings (for example, Java, JavaScript, Python, SQL, and REST APIs), with successive versions increasing realism from broad coverage (v1) to real tool execution (v2), multi-turn multi-step function calling (v3), and holistic agent evaluation that emphasizes autonomous planning and sequential decision making (v4). Finally, τ-bench [27] evaluates conversational agents that must interact naturally with users while following policy constraints, and its extension τ^2-bench further introduces dual-control environments where both the agent and the user can act on a shared world via tools, enabling fine-grained diagnosis of failures in reasoning versus coordination.

5.3.8. Safety Evaluation

We adopt the Tulu 3 benchmarking suite [11] to evaluate safety boundaries, balancing two objectives: maximizing refusals on harmful prompts while minimizing over-refusal on benign inputs in XSTest [18] and WildGuardTest [6]. Response validity is adjudicated by the WildGuard model [6], and we report macro-averaged accuracy across all benchmarks, where higher scores indicate better overall safety behavior. Concretely, we evaluate refusals on BeaverTails [9] (1,483 harmful prompts), HarmBench [15] (300 examples from Standard, Contextual, and Copyright subsets), Do-Anything-Now [19] (300 DAN-templated malicious prompts), Do-not-Answer [22] (939 harmful prompts), TrustLLM [7] (1,400 jailbreak prompts), and WildGuardTest [6] (780 harmful prompts within 1,725 items), reporting Refusal Rate (RTA) based on whether WildGuard classifies the response as a refusal; for XSTest [18], we report aggregate accuracy by requiring refusals on unsafe prompts and compliance on adversarial benign prompts.

Conclusion

In this work, we present IQuest-Coder-V1, a family of code LLMs that advance the state-of-the-art in autonomous software engineering through the code-flow pre-training paradigm and multi-phase evolutionary training. By capturing dynamic repository transitions and integrating extensive reasoning trajectories with repository-scale context during mid-training, our models establish robust logical foundations for complex code intelligence tasks. IQuest-Coder-V1 demonstrates exceptional performance across diverse benchmarks spanning agentic software engineering, competitive programming, and tool use, validating the effectiveness of our training methodology. The IQuest-Coder-V1-Loop variant further addresses practical deployment challenges through recurrent architectural innovations that optimize the capacity-efficiency trade-off. Through our bifurcated post-training approach—combining reasoning-driven reinforcement learning with instruction-tuning—we deliver specialized models tailored for both deep analytical reasoning and general assistance scenarios. By open-sourcing the complete training pipeline and model checkpoints, we aim to catalyze further research in code intelligence and accelerate the development of production-ready agentic systems capable of tackling real-world software engineering challenges.

6. Contributions and Acknowledgements

The authors of this paper are listed in order as follows:
Jian Yang, Wei Zhang, Shawn Guo, Zhengmao Ye, Lin Jing, Shark Liu, Yizhi Li, Jiajun Wu, Cening Liu, X. Ma, Yuyang Song, Siwei Wu, Yuwen Li, L. Liao, T. Zheng, Ziling Huang, Zelong Huang, Che Liu, Yan Xing, Renyuan Li, Qingsong Cai, Hanxu Yan, Siyue Wang, Shikai Li, Jason Klein Liu, An Huang, Yongsheng Kang, Chuan Hao, Haowen Wang, Weicheng Gu, Ran Tao, Mingjie Tang, Peihao Wu, Jianzhou Wang, Xianglong Liu, Weifeng Lv, Bryan Dai.

Core Contributors: Jian Yang, Wei Zhang, Shawn Guo, Zhengmao Ye, Lin Jing, Shark Liu, Yizhi Li, Jiajun Wu.

Contributors Cening Liu, X. Ma, Yuyang Song, Siwei Wu, Yuwen Li, L. Liao, T. Zheng, Ziling Huang, Zelong Huang, Che Liu, Yan Xing, Renyuan Li, Qingsong Cai, Hanxu Yan, Siyue Wang, Shikai Li, Jason Klein Liu, An Huang, Yongsheng Kang, Jinxing Zhang, Chuan Hao, Jing Yang, Haowen Wang, Weicheng Gu, IQuest Coder.

Leadership and Senior Advisory Committee: Ran Tao, Mingjie Tang, Peihao Wu, Jianzhou Wang, Xianglong Liu, Weifeng Lv.

Corresponding Authors: Bryan Dai.

References

1 Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models to fill in the middle, 2022. URL https://arxiv.org/abs/2207.14255.

2 Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36:28091–28114, 2023.

3 Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion, 2023. URL https://arxiv.org/abs/2310.11248.

4 Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. Mercury: A code efficiency benchmark for code large language models, 2024. URL https://arxiv.org/abs/2402.07844.

5 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.

6 Seungju Han, Kavel Rao, Allyson Ettinger, Liwei Jiang, Bill Yuchen Lin, Nathan Lambert, Yejin Choi, and Nouha Dziri. Wildguard: Open one-stop moderation tools for safety risks, jailbreaks, and refusals of llms. *Advances in Neural Information Processing Systems*, 37:8093–8131, 2024.

7 Yue Huang, Lichao Sun, Haoran Wang, Siyuan Wu, Qihui Zhang, Yuan Li, Chujie Gao, Yixin Huang, Wenhan Lyu, Yixuan Zhang, et al. Trustllm: Trustworthiness in large language models. *arXiv preprint arXiv:2401.05561*, 2024.

8 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. *arXiv preprint arXiv:2403.07974*, 2024.

9 Jiaming Ji, Mickel Liu, Josef Dai, Xuehai Pan, Chi Zhang, Ce Bian, Boyuan Chen, Ruiyang Sun, Yizhou Wang, and Yaodong Yang. Beavertails: Towards improved safety alignment of llm via a human-preference dataset. *Advances in Neural Information Processing Systems*, 36: 24678–24704, 2023.

10 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, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

11 Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, et al. Tulu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124*, 2024.

12 Jinyang Li, Binyuan Hui, Ge Qu, Binhua Li, Jiaxi Yang, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, et al. Can llm already serve as a database interface. *A big bench for large-scale database grounded text-to-sqls. CoRR, abs/2305.03111*, 2023.

13 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 *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

14 Siyao Liu, Ge Zhang, Boyuan Chen, Jialiang Xue, and Zhendong Su. FullStack Bench: Evaluating llms as full stack coders. *arXiv preprint arXiv:2412.00535*, 2024. URL https://arxiv.org/abs/2412.00535.

15 Mantas Mazeika, Long Phan, Xuwang Yin, Andy Zou, Zifan Wang, Norman Mu, Elham Sakhaee, Nathaniel Li, Steven Basart, Bo Li, et al. Harmbench: A standardized evaluation framework for automated red teaming and robust refusal. *arXiv preprint arXiv:2402.04249*, 2024.

16 Shishir G. Patil, Huanzhi Mao, Charlie Cheng-Jie Ji, Fanjia Yan, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Advances in Neural Information Processing Systems*, 2024.

17 polyglot-benchmark. polyglot-benchmark, 2025. URL https://aider.chat/2024/12/21/polyglot.html#the-polyglot-benchmark.

18 Paul Röttger, Hannah Kirk, Bertie Vidgen, Giuseppe Attanasio, Federico Bianchi, and Dirk Hovy. Xstest: A test suite for identifying exaggerated safety behaviours in large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 5377–5400, 2024.

19 Xinyue Shen, Zeyuan Chen, Michael Backes, Yun Shen, and Yang Zhang. " do anything now": Characterizing and evaluating in-the-wild jailbreak prompts on large language models. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 1671–1685, 2024.

20 swebench. swebench, 2025. URL https://www.swebench.com/original.html.

21 The Terminal-Bench Team. Terminal-bench: A benchmark for ai agents in terminal environments, Apr 2025. URL https://github.com/laude-institute/terminal-bench.

22 Yuxia Wang, Haonan Li, Xudong Han, Preslav Nakov, and Timothy Baldwin. Do-not-answer: A dataset for evaluating safeguards in llms. *arXiv preprint arXiv:2308.13387*, 2023.

23 Jian Yang, Jiajun Zhang, Jiaxi Yang, Ke Jin, Lei Zhang, Qiyao Peng, Ken Deng, Yibo Miao, Tianyu Liu, Zeyu Cui, et al. Execrepobench: Multi-level executable code completion evaluation. *arXiv preprint arXiv:2412.11990*, 2024.

24 Jian Yang, Shawn Guo, Lin Jing, Wei Zhang, Aishan Liu, Chuan Hao, Zhoujun Li, Wayne Xin Zhao, Xianglong Liu, Weifeng Lv, et al. Scaling laws for code: Every programming language matters. *arXiv preprint arXiv:2512.13472*, 2025.

25 Jian Yang, Wei Zhang, Yizhi Li, Shawn Guo, Haowen Wang, Aishan Liu, Ge Zhang, Zili Wang, Zhoujun Li, Xianglong Liu, et al. Codesimpleqa: Scaling factuality in code large language models. *arXiv preprint arXiv:2512.19424*, 2025.

26 Jian Yang, Wei Zhang, Shark Liu, Jiajun Wu, Shawn Guo, and Yizhi Li. From code foundation models to agents and applications: A practical guide to code intelligence. *arXiv preprint arXiv:2511.18538*, 2025.

27 Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. taubench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

28 Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*, 2018.

29 Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.