LoopCoder : Scaling Code Intelligence via Looped Language Models

Jian Yang[1] Wei Zhang[2] Shawn Guo[2] Yizhi Li[2] Lin Jing[2] Zhengmao Ye[2] Shark Liu[2] Yuyang Song[2] Jiajun Wu[2] Che Liu[2] T. Zheng[2] Siwei Wu[2] L. Liao[2] X. Ma[2] Chuan Hao[2] Ran Tao[2] Yan Xing[2] Jianzhou Wang[2] Mingjie Tang[2] Aishan Liu[1] Zhoujun Li[1] Xianglong Liu[1†] Weifeng Lv[1†] Bryan Dai[2†]

[1]Beihang University [2]IQuest Research

[†]Corresponding Authors. Email: {xlliu, lwf}@buaa.edu.cn

Abstract

While large language models (LLMs) have mastered syntax-level code generation, complex algorithmic reasoning remains a challenge, typically addressed by scaling model depth and parameter count. Universal Transformers (UT) offer a compelling alternative by introducing a recurrent inductive bias that aligns physically with the recursive nature of programming logic. However, training looped architectures at scale has historically been hindered by severe instability and optimization difficulties associated with backpropagation through time (BPTT). In this work, we present LoopCoder (40B-A80B) pre-trained on a corpus of 12T+ code and general tokens and the corresponding thinking and instruct version LoopCoder-Thinking and LoopCoder-Instruct, the first large-scale looped transformer for code that achieves superior convergence and downstream performance compared to standard dense architectures. Unlike prior approaches that restrict recurrence to small-scale tasks or specific fine-tuning stages, we implement a comprehensive Looped Training Protocol that spans both pre-training and post-training phases. We initiate the model via a dense-to-loop Transformation, folding a pre-trained dense checkpoint to initialize a recurrent block. This is followed by a rigorous looped pre-training stage that aligns general code representations with the recurrent architecture, and a specialized post-training phase targeting instruction following and thinking models (latent reasoning). We demonstrate that this full-stack integration of recurrence is key to unlocking the model's reasoning potential. LoopCoder exhibits faster convergence than dense baselines after mitigating BPTT issues by stabilizing the recurrence, and enables emergent over-thinking capabilities during inference. Our results establish a robust recipe for scaling coding intelligence via recurrent computation, proving that dense checkpoints serve as an optimal foundation for evolving into dynamic, looped reasoners.

1. Introduction

The ability to generate and reason about computer code is a cornerstone of modern Large Language Models (LLMs). The dominant recipe for improving this capability has been the dense scaling law [22, 36], stacking identical Transformer layers in depth to increase the model's capacity for hierarchical abstraction. While effective, this approach rigidly couples reasoning depth with memory footprint, resulting in static computation graphs that cannot dynamically adapt to problem complexity.

Theoretical frameworks like the universal Transformer (UT) [9] propose a more elegant solution: recurrence over depth. By sharing weights across layers, UTs introduce a recurrent inductive bias

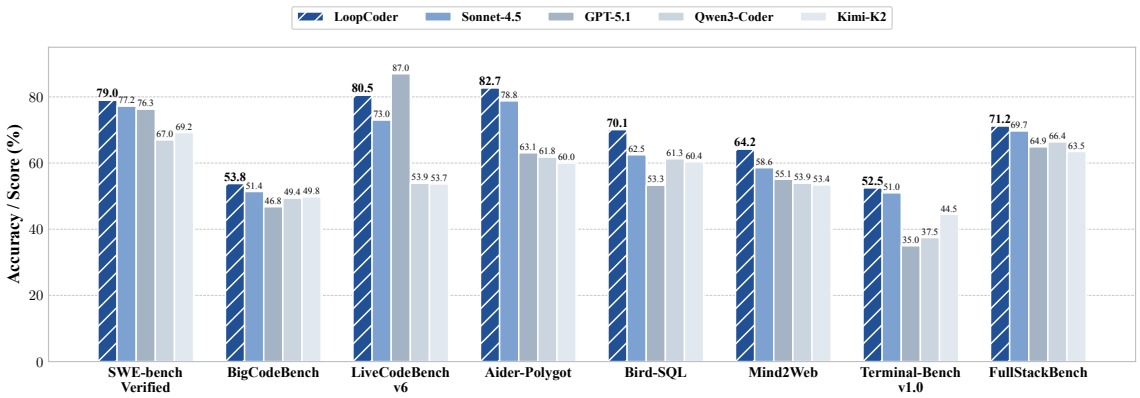

Figure 1. Benchmark performance of LoopCoder and its counterparts.

that naturally mirrors the iterative and recursive control flows (e.g., loops, recursion) inherent in programming languages. Recent studies, such as the Universal Reasoning Model (URM) [13], have validated this potential on abstract reasoning benchmarks like ARC-AGI [7], showing that looped models can achieve high parameter efficiency through iterative refinement.

However, scaling these benefits to real-world software engineering tasks has remained elusive. Training deep looped networks at scale is notoriously unstable, suffering from vanishing or exploding gradients inherent to Backpropagation Through Time (BPTT). Consequently, prior attempts have struggled to match the performance of well-optimized dense baselines, often failing to stabilize the architecture during the critical heavy-lifting phases of training.

In this work, we bridge this gap with LoopCoder, a large-scale looped coding model that not only matches but surpasses the convergence efficiency of equivalent dense models. We depart from the traditional view of recurrence as merely a fine-tuning trick. Instead, we propose a holistic full-lifecycle looped protocol:

- Dense-to-Loop Transformation: We bypass the cold start instability by strategically initializing our recurrent block from a folded pre-trained dense checkpoint. This preserves prior syntactic knowledge while establishing the necessary weight-sharing constraints.
- Looped Pre-training: We integrate the looped architecture directly into the continued pre-training phase. This critical step allows the model to fundamentally adapt its latent representations to the recurrent inductive bias on massive unsupervised code corpora, resolving the alignment issues that plague direct fine-tuning approaches.
- Looped Post-training: We specialize the model through a stabilized post-training recipe covering both Instruction Following (SFT) and Thinking Models (CoT), where the physical loop structure acts as a natural carrier for the latent thinking process.

Our contributions are as follows:

- **Large-Scale Viability**: We report the first successful training of a large-scale looped transformer for code using a continuous pre-training and post-training protocol, achieving better convergence rates than continuing to train dense baselines.
- **Mechanism Analysis**: We investigate why our pre-training and post-training Loop strategy succeeds where others failed, providing empirical evidence that early adaptation in the pre-training phase is essential for mitigating BPTT instability.
- **Emergent Reasoning**: We demonstrate that LoopCoder, trained as a thinking model, exhibits emergent capabilities to verify and refine code logic through dynamic recurrence, outperforming static dense models on complex algorithmic tasks.

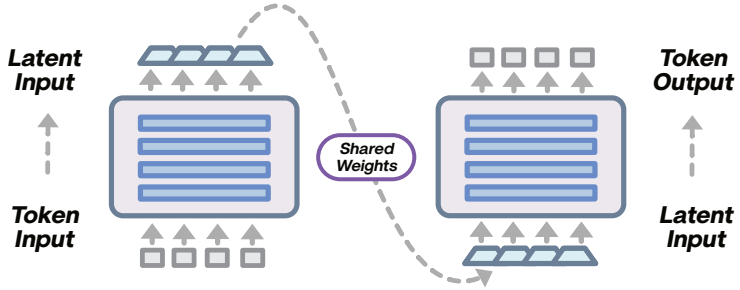

Figure 2. LoopCoder mechanism. During training and inference, the model recurrently predict output tokens with shared weights, using the latent representations produced by the previous iteration.

2. LoopCoder

2.1. Model Architecture

For the LoopCoder series (i.e., LoopCoder-40B-A80B-Loop), we adopt a loop transformer architecture inspired by the Parallel Loop Transformer [47, 48]. Specifically, given input embeddings $\boldsymbol{E} = (e_1, e_2, ..., e_n)$. Let $\boldsymbol{H}^{(l)}$ denotes the hidden states from the l-th iterations, with the initial hidden states given by $\boldsymbol{H}^{(1)} = f_\theta(\boldsymbol{E})$. The transformer layers are executed in two fixed iterations (i.e., $L = 2$ loops):

$$\boldsymbol{H}^{(l+1)} = f_\theta(\boldsymbol{E} + Shift(\boldsymbol{H}^{(l)}, l)), \quad l = 1, ..., L-1. \tag{1}$$

where f_θ denotes the transformer block with shared parameters θ across loops and $Shift(\boldsymbol{H}^l, l)$ indicates shifting $\boldsymbol{H}^l$ by l positions. In the second iteration, for each layer, we compute the query, key, and value projections as $\boldsymbol{Q}^{(2)} = \boldsymbol{H}^{(2)}\boldsymbol{W}_Q$, $\boldsymbol{K}^{(2)} = \boldsymbol{H}^{(2)}\boldsymbol{W}_K$, and $\boldsymbol{V}^{(2)} = \boldsymbol{H}^{(2)}\boldsymbol{W}_V$. Similarly, we cache the key-value pairs from the first iteration as $\boldsymbol{K}^{(1)}$ and $\boldsymbol{V}^{(1)}$.

We then compute two types of attention outputs:

- **Global Attention:** The queries from the second iteration attend to the key-value pairs from the first iteration, enabling the model to refine its representation based on the complete sequence context:

$$\boldsymbol{O}_{\text{global}} = \text{Attention}(\boldsymbol{Q}^{(2)}, \boldsymbol{K}^{(1)}, \boldsymbol{V}^{(1)}) \tag{2}$$

- **Local Attention:** The queries attend to the key-value pairs of preceding tokens within the second iteration, preserving causal dependencies:

$$\boldsymbol{O}_{\text{local}} = \text{Attention}(\boldsymbol{Q}^{(2)}, \boldsymbol{K}^{(2)}_{<t}, \boldsymbol{V}^{(2)}_{<t}) \tag{3}$$

The final output is obtained by a gated combination of these two attention outputs, where the gating weights are derived from the query representations:

$$\boldsymbol{g} = \sigma(\boldsymbol{Q}^{(2)}\boldsymbol{W}_g) \tag{4}$$

$$\boldsymbol{O} = \boldsymbol{g} \odot \boldsymbol{O}_{\text{global}} + (1 - \boldsymbol{g}) \odot \boldsymbol{O}_{\text{local}} \tag{5}$$

where $\sigma(\cdot)$ denotes the sigmoid function, $\boldsymbol{W}_g$ is a learnable projection matrix, and $\odot$ represents element-wise multiplication.

Unlike the original Parallel Loop Transformer, our implementation does not incorporate the token-shifting mechanism and does not include optimizations specifically designed for downstream inference efficiency.

Model Size	Layers	Hidden Size	Intermediate Size	Attention	Max Context	Query Heads	KV Heads	Vocabulary	Activated Params
Dense Models									
DenseCoder-40B-Base	80	5120	27648	GQA	131072	40	8	76800	80B
DenseCoder-40B-Instruct	80	5120	27648	GQA	131072	40	8	76800	80B
DenseCoder-40B-Thinking	80	5120	27648	GQA	131072	40	8	76800	80B
Loop Models									
LoopCoder-40B-Base	80	5120	27648	GQA	131072	40	8	76800	80B
LoopCoder-40B-Instruct	80	5120	27648	GQA	131072	40	8	76800	80B
LoopCoder-40B-Thinking	80	5120	27648	GQA	131072	40	8	76800	80B

Table 1. Architecture of LoopCoder.

2.2. Stabilizing Recurrence: Adaptation and Gradient Control

While the recurrent inductive bias of LoopCoder offers theoretical advantages, realizing these benefits at scale presents a formidable optimization challenge. A prominent obstacle is the instability associated with BPTT. As the network is unrolled over multiple iterations, the deep computational graph often induces chaotic gradient dynamics. In our preliminary experiments, we observed severe gradient explosion, where the accumulation of error signals across loop steps destabilized the shared weights, impeding effective convergence. To address these optimization difficulties, we propose a stabilization framework that combines structural transformation with dynamic gradient control.

Structual Stablization: Dense-to-Loop Adaptation. One of the strategy to address the optimization hurdle is through strategic initialization, *i.e.*, adapting the model into loop mode from continual pre-training stage. Training a looped model from scratch requires the shared parameters to simultaneously learn syntactic structures and recursive logic, a cold start scenario that significantly exacerbates instability. We propose a dense-to-loop adaption, grounded in the structural isomorphism between residual networks [12] and looped Transformers (unrolled RNNs). Formally, a standard dense Transformer models updates hidden states via the mapping $\boldsymbol{H} = f_\theta(\boldsymbol{E})$. This formulation can be viewed as a discretized approximation of a continuous dynamical system, where distinct layers parameterized by θ drive the state evolution. We hypothesize that the vector fields learned by these distinct dense layers share a common functional manifold. Based on this insight, we "fold" the pre-trainied dense checkpoints by aggregating weights from representative blocks to initialize the LoopCoder. This strategy provides a robust starting point where the spectral radius of the transformation is well-behaved, allowing the optimization process to focus on adapting to the recurrent constraint rather than learning representations *ab initio*.

2.3. Infrastructure Design of LoopCoder

The training of LoopCoder necessitated a total of over million GPU hours. Given this immense computational scale, our infrastructure design focuses on two core pillars: maximizing computational efficiency and ensuring system reliability. To achieve high efficiency, we minimize memory bandwidth and communication latency through fused kernel optimizations. Concurrently, to guarantee reliability, we deploy a rigorous silent error detection strategy to safeguard training correctness against hardware instabilities.

Fused Gated Attention Kernel. Equations 2-5 as independent kernels incurs redundant data transmission of intermediate results between HBM and on-chip SRAM. To address this, we implement a Fused Gated Attention Kernel to integrate these computation steps. This design not only eliminates unnecessary memory bandwidth consumption associated with intermediate results, but also significantly reduces kernel launch overhead.

Context Parallelism. Ring Attention [32] enables the training of ultra-long context models by

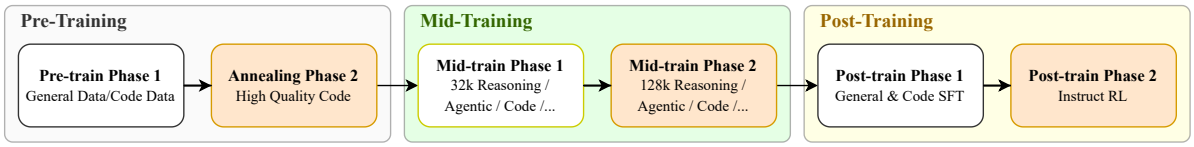

Figure 3. Training pipeline of LoopCoder .

achieving global attention through the circulation of Key-Value (KV) shards. However, existing implementations face limitations: TransformerEngine [38] lacks support for local attention mechanisms, while AllGather-based approaches [18] achieve this functionality by collecting global KV shards, incurring additional memory overhead. To address these challenges, we integrate shared-memory communication primitives within the Fused Gated Attention kernel to enable point-to-point transmission of the KV shards required by context parallel ranks. This design realizes the gated attention operator with reduced memory overhead, and allows for more fine-grained hiding of communication latency.

Silent Error Detection. Silent errors, which compromise training correctness without triggering explicit system exceptions, are identified using a two-fold detection strategy. First, we employ deterministic re-computation to isolate faulty nodes by verifying the consistency of results across repeated executions. Second, we extract tensor *fingerprints* from GPU memory to validate invariant conditions. These invariants represent strict logical constraints derived from the training recipe, exemplified by the requirement that parameter replicas within a data parallel group remain numerically identical prior to the forward pass.

2.4. 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[1]. 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 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 create a large-scale instruction corpus with 66 million samples for 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.

[1]https://commoncrawl.org/

Repository Transition 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) [5] 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 as illustrated in Figure 4. 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. The structure of this task is shown in Figure 5. We primarily employ two strategies for code completion data construction: heuristic-based and multi-level syntax-based construction [49].

File-level Completion

```
<|fim_prefix|>{code_pre}<|fim_suffix|>{code_suf}<|fim_middle|>{code_mid}<|im_end|>
```

Figure 4. Illustration of the File-Level Fill-In-the-Middle (FIM) input format.

Repository-level Completion

```
<|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|>
```

Figure 5. Illustration of the Repo-Level Fill-In-the-Middle (FIM) input format.

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 multi-language and multi-level completion data for both file-level and repo-level tasks, significantly enhancing the diversity of the training samples.

2.5. Stage2: Mid-Training

Dataset	Stage2.1 (32K)	Stage2.2 (128K)	ALL
	Tokens (B)		
Reasoning QA	175	65	240
Agent Trajectory	10	110	120
Commit	5	35	40
File-Level (50% FIM)	35	5	40
Repo-Level (50% FIM)	5	35	40
DownSampling Stage1	70	50	120
ALL	**300**	**300**	**600**

Table 2. **Token statistics across Stage2.1 and Stage2.2.**

We design the mid-training process in two stages to improve capability growth while controlling computational cost. Both stages use the same core data categories: **Reasoning QA** (including coding, math, and logic), **Agent trajectory data**, **Commit data**, **File-level and Repository-level FIM data**, and filtered Stage 1 samples to maintain distribution continuity. **Stage 2.1 (32K)** selects representative samples from each category and focuses on stabilizing behavior at moderate sequence lengths, allowing the model to absorb new signals efficiently. **Stage 2.2 (128K)** introduces a new set of 32K samples (non-overlapping with Stage 2.1 and accounting for roughly one-tenth of this stage) alongside dedicated 128K data across the same categories, extending the model's effective context range for repository-level reasoning and tool-interaction workflows. This staged structure enables long-context adaptation without excessive compute demands. Each stage contains approximately **300B** tokens, resulting in **600B** tokens in total for mid-training.

Math and Code Reasoning QA for math and code functions like a "reasoning runtime" installed into the model: it rewards explicit decomposition, persistent state tracking, and internal consistency checks, so multi-step solutions behave less like pattern replay and more like controlled execution. This direction matches recent open reasoning-data and verifiable-answer training that aim to push general reasoning rather than narrow task tricks [20]. When pretraining or continual pretraining is also seeded with explicit or latent-thought traces, the model gains a stronger internal scaffold for long-form derivations, making difficult math proofs and code synthesis more stable under distribution shift [41, 44, 53].

Agent Agent trajectory data is training for "closed-loop intelligence": the model learns to act, observe, revise, and continue, while keeping its plan consistent with earlier signals such as tool outputs, edited files, and partial failures. More importantly, trajectories carry dense environment state and rich exchange data with that environment (commands, logs, file diffs, test results, error traces), which can be seen as early "code world" grounding: a concrete substrate that links symbols to executable consequences. This turns tool use from a single-shot call into a coherent control process, improving long-horizon completion, recovery after wrong turns, and goal

retention across many steps. It also makes agentic synthetic data a scalable fuel source: agentic pipelines can generate large volumes of trajectories that are directly usable for pretraining or continual pretraining.

3. Post-Training

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

3.1. Data Construction

Synthesis Philosophy We use a model-centric framework where frontier LLMs generate training data under rigorous automated verification. Deterministic domains use execution-based validation; subjective domains employ ensemble mechanisms combining rule-based checks, reward models, and multi-agent debate.

API Orchestration We mine API patterns from production repositories with mature test coverage. Frontier models apply stochastic perturbations including cross-library dependencies, performance constraints, and edge-case amplification. Three-stage verification ensures quality: static analysis, sandboxed execution, and LLM-generated adversarial tests. We synthesize execution trace annotations capturing intermediate states, distilled into natural language for debugging and code understanding.

Full-Stack Engineering Test-driven synthesis harvests test suites from open-source projects, challenging models to generate satisfying implementations iteratively. We preserve complete trajectories including failures and error messages to teach debugging strategies. Validation deploys synthesized applications in headless browsers, executing interaction scripts and verifying DOM states and database contents.

Competitive programming We curated and collected competition problems from major open-source programming platforms, and used an internal model to filter them by difficulty. We retained the more challenging problems along with their corresponding verifiers, resulting in approximately 8,000 pairs. A subset of this data was selected to construct high-quality reasoning data for cold-start initialization, while the remaining data was used for reinforcement learning training.

Code Reasoning For CRUXEval, we develop specialized training for constraint satisfaction. The O2I task requires deducing valid inputs from outputs—an ill-posed inverse problem. We synthesize solution set annotations with diverse valid inputs and employ set-based losses. Chain-of-thought traces articulate constraint derivation: parsing control flow, backward-propagating outputs, solving constraint systems, and verifying through forward execution.

Text-to-SQL We reverse the traditional pipeline: generate valid SQL from schemas, then synthesize natural language questions. This exploits SQL's structural rigidity to minimize hallucination. Round-trip verification feeds generated questions back to SQL models, confirming semantic equivalence. Each sample includes reasoning traces decomposing SQL generation into interpretable steps.

Code Editing Code editing requires surgical precision, contextual understanding, and regression prevention. We curate editing tasks from commit diffs, synthetic bugs, and refactoring challenges. Sophisticated filtering prioritizes commits with clear messages, comprehensive tests, localized changes, and best practices. Edit-aware instruction tuning structures inputs with objectives, line-numbered code, context, and expected unified diffs. Repository-level refactoring captures coordinated multi-file changes. Evaluation assesses edit precision, recall, and code quality delta.

Terminal Bench We curate repository-grounded datasets with quality scoring across code quality, documentation, dependency hygiene, and CI/CD maturity. For high-quality repositories, we construct Docker environments replicating production conditions. Task synthesis extracts commands from scripts, mines tutorials, and employs LLM-based generation. Each task pairs with executable validation. We deploy environments and run multiple LLMs, retaining only validated trajectories. This eliminates subtle command errors critical for training.

Repository-Scale Engineering We construct training data from GitHub Issue-PR pairs with automated environment construction. Multi-stage filtering combines heuristics and LLM-based semantic analysis. The automated agent clones repositories, resolves dependencies, initializes databases, and configures environments. LLM-based debugging agents improve success rates from 30-40% to over 70%. Validation requires Fail-to-Pass tests demonstrating genuine issue resolution and zero Pass-to-Fail tests ensuring no regressions. We deploy code agents performing interactive problem-solving, recording complete trajectories. Sophisticated context management handles repositories exceeding context limits through semantic summarization, dynamic retrieval, and hierarchical representation.

Tool Use We build tool-use capabilities through four stages from simple invocation to complex coordination. Tool curation aggregates APIs with two-level clustering to identify duplicates while preserving diversity. Multi-agent simulation deploys role-playing LLM [30] instances simulating users, agents, and servers. Quality control combines hallucination detection and multi-agent voting. Cold-start SFT uses trajectories augmented with a distilled chain-of-thought. Iterative RL refines behavior through GRPO cycles with composite rewards balancing format compliance, tool correctness, and reasoning quality.

GUI Agent We develop grounded web interaction for Mind2Web through multi-modal understanding. Training data includes task descriptions, screenshots, HTML DOM, and action sequences. Data augmentation generates diversity through synthetic tasks, cross-website transfer, and hard negatives. Multi-modal encoders process visual and structural information jointly. The action space includes element selection, navigation, and task verification. Training combines behavioral cloning, contrastive learning, and trajectory-level rewards.

3.2. Large-Scale Supervised Fine-Tuning

Scale and Objectives Post-training processes token counts approaching pre-training scale, injecting dense task-specific knowledge underrepresented in pre-training: rare API patterns, specialized algorithms, and nuanced reasoning strategies.

Optimization Infrastructure **Sequence Packing.** Aggressive packing concatenates samples into extended sequences with cross-sample attention masking, substantially improving efficiency while potentially improving task-switching. **Learning Rate Dynamics.** Cosine annealing uses conservative peak rates decaying to minimal terminal rates. Extended low-rate annealing

phases prove critical for stable convergence with consistent behavior. **Curriculum Learning.** Three-phase curriculum sequences data by difficulty. Phase one focuses on format alignment and basic instruction-following. Phase two introduces specialized knowledge with difficulty-aware sampling. Phase three emphasizes frontier challenges with adversarial examples. This substantially outperforms random sampling on complex benchmarks.

Quality Control and Verification **Execution Infrastructure.** Comprehensive sandboxes support extensive language coverage with compilation, test execution, and profiling. Only zero-error samples enter training. Extended semantics capture execution traces, memory profiles, performance metrics, and coverage statistics. Mathematical verification employs symbolic computation engines and formal proof assistants. **Subjective Quality Assessment.** Multi-agent debate protocols evaluate domains lacking objective ground truth through independent scoring, peer review, and weighted voting calibrated against human preferences. **Contamination Prevention.** Multi-level detection combines exact n-gram matching, fuzzy string matching, and semantic similarity. Aggressive deduplication via MinHash LSH consistently improves generalization, demonstrating quality dominates quantity.

3.3. Multi-Objective Optimization

Alignment Tax Mitigation Specialized improvements risk degrading general performance due to finite parameter capacity and distribution shift. **Replay Buffer Regularization.** Replay buffers containing high-quality pre-training samples prevent representation collapse through quality-first sampling. **Dynamic Mixture Adaptation.** Real-time monitoring evaluates comprehensive benchmarks. Performance degradation triggers increased sampling weights; plateaus trigger reductions. This navigates Pareto frontiers efficiently. **Compositional Mixture Design.** Final mixtures reflect empirical optimization with hierarchical structure enabling precise capability control.

Reinforcement Learning from Verifiable Feedback For reinforcement learning on competition-style coding tasks, under a maximum context length of 96K, we adopt the GRPO [43] algorithm, remove the KL penalty, and incorporate the Clip-Higher strategy inspired by DAPO [51] to expand the exploration space of GRPO. For each problem, we select 20 high-quality test cases and compute the problem pass rate as the reward signal. Training is conducted for approximately 500 steps with a batch size of 64 and 16 rollouts per prompt.

SWE-RL For SWE tasks, we formulate problems as executable and interactive RL environments and build an end-to-end SWE-RL training and evaluation framework on isolated, containerized cloud sandboxes. Agents interact through tool-based actions over multiple steps. Rewards follow a rule-based design, where the primary reward is determined by whether the task passes the test cases, complemented by lightweight regularization penalties on excessive context usage, redundant tool invocations, and unproductive code edits, encouraging compact decision-making under a bounded context budget. Policy optimization is performed using GRPO [43], which updates the policy based on the relative performance of groups of sampled interaction trajectories, enabling stable learning from sparse and delayed rewards without an explicit value function. Each iteration generates 512 trajectories in parallel and incorporates curriculum learning to progressively expand the exploration space. All trajectories are executed concurrently in secure and reproducible sandbox instances, enabling high-throughput and stable experience collection and improving long-horizon code reasoning and repair capabilities.

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
LoopCoder-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.

Safety Alignment Comprehensive frameworks operate at the curation, training, and inference levels. Specialized classifiers detect harmful patterns. Safety-aware rewriting pairs demonstrations with warnings. Constitutional AI encodes safety desiderata as explicit rules for DPO. Extensive red-teaming employs human experts and automated adversarial generation, driving targeted augmentation and gradient-based adversarial training.

Emergent Capabilities Beyond quantitative improvements, we observe qualitative emergence: self-debugging with systematic error analysis, cross-language transfer from shared algorithmic patterns, compositional tool use decomposing complex goals, and improved uncertainty calibration from DPO training and process reward modeling.

4. Evaluation

4.1. 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 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.

4.2. Experiments on Base Models

4.2.1. Code Completion

We evaluate cross-file code completion on CrossCodeEval [10], 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.

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
LoopCoder-40B-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.

4.3. Evaluation on Instruct Models and Reasoning model

4.3.1. Code Generation

Across a wide range of code-generation evaluations, our model achieves consistently strong performance. We validate functional correctness and robustness using EvalPlus [33] (including HumanEval+ and MBPP+ with substantially expanded test suites), and measure compositional, library-intensive problem solving on BigCodeBench [57]. We further demonstrate broad full-stack capability on FullStackBench [34], and strong results under contamination-aware, continuously refreshed testing on LiveCodeBench [24].

4.3.2. Code Reasoning

We further evaluate code reasoning with CRUXEval [19], 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.

4.3.3. Code Editing

We evaluate code editing on Aider's Polyglot benchmark [39], 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.

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**
LoopCoder-40B-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	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**
LoopCoder-40B-Instruct	91.1	85.5	48.6	48.5
LoopCoder-40B-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 6. Performance comparison on Code Reasoning Evaluation.

4.3.4. Code Efficiency

We assess code efficiency with Mercury [11], 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.

4.3.5. 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 [52], which uses a

Model	Aider-Polyglot		Mercury		Text2SQL	
	Diff Pass@2	Whole Pass@2	Beyond@1	Pass@1	Bird	Spider
6B+ Models						
DeepSeek-Coder-V2-Lite-Instruct	1.3	2.2	76.8	91.4	41.6	72.4
Qwen2.5-Coder-7B-Instruct	1.8	4.9	69.9	84.8	53.1	79.8
Seed-Coder-8B-Instruct	6.2	5.3	78.5	93.8	44.7	72.7
13B+ Models						
Qwen2.5-Coder-14B-Instruct	8.0	8.0	76.7	88.3	59.1	81.3
Qwen3-Coder-30B-A3B-Instruct	28.4	29.3	81.1	95.3	59.0	80.9
20B+ Models						
DeepSeek-v3.2	-	-	81.6	**96.9**	52.6	77.9
Qwen2.5-Coder-32B-Instruct	8.4	14.7	79.1	96.1	62.1	83.9
Qwen3-235B-A22B-Instruct-2507	53.3	57.3	80.4	96.9	62.8	81.1
Qwen3-235B-A22B-Thinking-2507	25.3	-	61.2	70.3	35.2	42.6
Qwen3-Coder-480B-A35B-Instruct	57.8	61.8	80.2	96.1	61.3	81.2
Kimi-Dev-72B	12.0	20.0	59.1	69.5	-	-
Kimi-K2-Instruct-0905	60.0	50.7	76.1	90.6	60.4	81.1
Kimi-K2-Thinking	-	-	73.0	85.2	40.6	49.6
KAT-Dev	8.9	34.2	75.1	89.1	52.2	77.6
KAT-Dev-72B-Exp	16.4	15.6	79.0	94.5	35.2	60.3
GLM-4.7	-	-	74.1	86.7	46.5	62.4
LoopCoder-40B-Instruct	**68.9**	**62.3**	**82.2**	94.1	**69.9**	**84.0**
Closed-APIs Models						
Gemini-3-Flash-preview	-	-	78.4	89.5	66.6	87.2
Gemini-3-Pro-preview	91.9	92.9	83.1	96.1	67.5	87.0
Claude-Opus-4.5	89.4	87.1	82.9	96.9	66.0	76.0
Claude-Sonnet-4.5	78.8	-	82.5	97.7	62.5	80.1
GPT-5.1	63.1	65.3	81.9	96.1	53.3	77.6

Table 7. Combined performance across code editing (Aider-Polyglot), code efficiency (Mercury), and Text2SQL (Bird/Spider).

database-level train–test split to evaluate schema linking and structurally correct SQL generation with complex constructs, and on BIRD [28], 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.

4.3.6. 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 [46] 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 [25] 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.

We further evaluate our model in agentic, end-to-end software workflows where success de-

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
LoopCoder-40B-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 8. Combined performance on agentic coding tasks (Terminal-Bench, Terminal-Bench 2.0, SWE-Verified) and general tool-use tasks (Mind2Web, BFCL V3).

pends on correct tool use, long-horizon planning, and tight interaction with the execution environment. Terminal-Bench [46] 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 [25] 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. Related Work

Code Foundation Models. Code foundation models have evolved from code-pretraining on large, permissively licensed corpora into full-stack systems that combine strong generation with editing, repair, and long-horizon software workflows. A major early driver was the open-data and open-training line exemplified by BigCode, where large-scale source and software-adjacent text (issues, PRs, notebooks, documentation) are curated and used to train code-capable LMs with systematic evaluation across coding tasks [29, 35]. Subsequent development emphasized (i) scaling and specialization of the pretraining mixture for code, and (ii) stronger post-training for instruction following and tool-driven programming, leading to code-focused series that

publish detailed technical reports on data cleaning, synthetic data generation, and code-centric evaluation (for example, Qwen-Coder [23]) and large-scale MoE families that report stable training recipes and competitive code performance (for example, DeepSeek-V3 [31],GLM4.7 [54]). In parallel, frontier general models with strong coding behavior increasingly release public system cards that document capability profiles and deployment constraints, reflecting the shift from standalone code completion to interactive, agentic software development settings [1, 40]. Overall, the field is moving toward unified code foundation models where progress is driven jointly by data pipelines, scalable training, and post-training that supports reliable multi-step development workflows.

5.1. Looped Language Models.

Looped (weight-tied) Transformers reuse the same block across depth, trading parameters for recurrent computation. This design traces back to Universal Transformers [9], and has since been explored both as (i) an algorithmic substrate and (ii) a route to stronger reasoning per parameter. On the algorithmic side, prior work argues that looping encourages learning structured procedures and learned algorithms [17, 50]; complementary studies analyze how looped computation can implement multi-step optimization dynamics for in-context learning [14, 15]. On the reasoning side, looped architectures have been connected to iterative latent refinement [42], while later systems scale or adapt effective depth through recurrent-depth or dynamic-compute mechanisms [3, 6, 16, 55]. In parallel, parameter sharing has long been used as an efficiency strategy [8, 26, 45], and recent work revisits it for modern LLMs via lightweight adaptation or large-scale shared-depth training [2, 27].

Looped LMs are often viewed from two angles: (i) compression and efficiency via parameter reuse, and (ii) latent reasoning via iterative refinement of hidden states rather than longer explicit outputs. Recent variants make refinement more explicit by feeding intermediate activations back into subsequent steps [21, 37], while other work studies compute–length tradeoffs [48] or dynamic halting objectives [4]. We build on this foundation but target a different bottleneck: scaling code intelligence with recurrent computation under stable optimization. We introduce LoopCoder and a Looped Training Protocol spanning dense-to-loop initialization, looped pre-training, and post-training for instruction following and latent thinking, explicitly mitigating BPTT instabilities via gradient-scale reduction. We also draw on large-scale LoopLM pre-training results [56], but focus on an end-to-end recipe that makes recurrence reliable for code and yields over-thinking behaviors at inference time.

6. Conclusion

In this work, we introduce LoopCoder, the first large-scale looped transformer architecture for code that successfully overcomes the historical challenges of training recurrent models at scale. Through our comprehensive Looped Training Protocol, which encompasses dense-to-loop transformation, looped pre-training on 12T+ tokens, and specialized post-training for instruction following and latent reasoning, we demonstrate that recurrent architectures can achieve superior convergence and downstream performance compared to traditional dense models. Our approach establishes that the recursive nature of Universal Transformers provides a natural inductive bias for algorithmic reasoning, enabling emergent over-thinking capabilities during inference without the optimization instabilities traditionally associated with backpropagation through time. By proving that pre-trained dense checkpoints serve as an optimal foundation for evolving into dynamic looped reasoners, we provide a robust and scalable recipe for advancing coding intelligence through recurrent computation. These findings open new pathways for building more capable code generation models that align architectural design with the funda-

mental recursive structure of programming logic, offering a compelling alternative to simply scaling model depth and parameter count.

Limitations

While LoopCoder demonstrates promising results in code generation through recurrent computation, several limitations warrant consideration. First, the looped architecture introduces additional computational overhead during both training and inference due to the iterative refinement process, which may limit practical deployment in latency-sensitive applications despite the parameter efficiency gains. Second, our gradient-scale reduction techniques for BPTT stabilization, while effective, required extensive hyperparameter tuning and may not generalize seamlessly to other domains or model scales without careful adaptation. Third, the dense-to-loop initialization strategy creates a dependency on high-quality pre-trained dense checkpoints, potentially limiting accessibility for researchers without access to such foundations. Fourth, while we demonstrate emergent over-thinking capabilities, the interpretability and controllability of the iterative reasoning process remain challenging, making it difficult to predict or guarantee the number of refinement steps needed for optimal performance on novel tasks. Finally, our evaluation focuses primarily on code-related benchmarks, and the extent to which looped architectures provide similar advantages for broader language understanding tasks beyond algorithmic reasoning remains an open question requiring further investigation.

References

1 Anthropic. Introducing Claude, 2023. URL https://www.anthropic.com/index/introducing-claude.

2 Sangmin Bae, Adam Fisch, Hrayr Harutyunyan, Ziwei Ji, Seungyeon Kim, and Tal Schuster. Relaxed recursive transformers: Effective parameter sharing with layer-wise lora. *arXiv preprint arXiv:2410.20672*, 2024.

3 Sangmin Bae, Yujin Kim, Reza Bayat, Sungnyun Kim, Jiyoun Ha, Tal Schuster, Adam Fisch, Hrayr Harutyunyan, Ziwei Ji, Aaron Courville, et al. Mixture-of-recursions: Learning dynamic recursive depths for adaptive token-level computation. *arXiv preprint arXiv:2507.10524*, 2025.

4 Andrea Banino, Jan Balaguer, and Charles Blundell. Pondernet: Learning to ponder. *arXiv preprint arXiv:2107.05407*, 2021.

5 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.

6 Yilong Chen, Junyuan Shang, Zhenyu Zhang, Yanxi Xie, Jiawei Sheng, Tingwen Liu, Shuohuan Wang, Yu Sun, Hua Wu, and Haifeng Wang. Inner thinking transformer: Leveraging dynamic depth scaling to foster adaptive internal thinking. *arXiv preprint arXiv:2502.13842*, 2025.

7 Francois Chollet, Mike Knoop, Gregory Kamradt, and Bryan Landers. Arc prize 2024: Technical report, 2025. URL https://arxiv.org/abs/2412.04604.

8 Raj Dabre and Atsushi Fujita. Recurrent stacking of layers for compact neural machine translation models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 6292–6299, 2019.

9 Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers, 2019. URL https://arxiv.org/abs/1807.03819.

10 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.

11 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.

12 Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

13 Zitian Gao, Lynx Chen, Yihao Xiao, He Xing, Ran Tao, Haoming Luo, Joey Zhou, and Bryan Dai. Universal reasoning model, 2025. URL https://arxiv.org/abs/2512.14693.

14 Khashayar Gatmiry, Nikunj Saunshi, Sashank J Reddi, Stefanie Jegelka, and Sanjiv Kumar. Can looped transformers learn to implement multi-step gradient descent for in-context learning? *arXiv preprint arXiv:2410.08292*, 2024.

15 Khashayar Gatmiry, Nikunj Saunshi, Sashank J Reddi, Stefanie Jegelka, and Sanjiv Kumar. On the role of depth and looping for in-context learning with task diversity. *arXiv preprint arXiv:2410.21698*, 2024.

16 Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian R Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. Scaling up test-time compute with latent reasoning: A recurrent depth approach. *arXiv preprint arXiv:2502.05171*, 2025.

17 Angeliki Giannou, Shashank Rajput, Jy-yong Sohn, Kangwook Lee, Jason D Lee, and Dimitris Papailiopoulos. Looped transformers as programmable computers. In *International Conference on Machine Learning*, pages 11398–11442. PMLR, 2023.

18 Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengx-

ing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi,

Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

19 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.

20 Etash Guha, Ryan Marten, Sedrick Keh, Negin Raoof, Georgios Smyrnis, Hritik Bansal, Marianna Nezhurina, Jean Mercat, Trung Vu, Zayne Sprague, et al. Openthoughts: Data recipes for reasoning models. *arXiv preprint arXiv:2506.04178*, 2025.

21 Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason Weston, and Yuandong Tian. Training large language models to reason in a continuous latent space. *arXiv preprint arXiv:2412.06769*, 2024.

22 Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

23 Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

24 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.

25 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.

26 Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

27 Boxun Li, Yadong Li, Zhiyuan Li, Congyi Liu, Weilin Liu, Guowei Niu, Zheyue Tan, Haiyang Xu, Zhuyu Yao, Tao Yuan, et al. Megrez2 technical report. *arXiv preprint arXiv:2507.17728*, 2025.

28 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.

29 Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii,

Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. StarCoder: May the source be with you! *arXiv preprint arXiv:2305.06161*, abs/2305.06161, 2023. doi: 10.48550/arXiv.2305.06161. URL https://doi.org/10.48550/arXiv.2305.06161.

30 Yuwen Li, Wei Zhang, Zelong Huang, Mason Yang, Jiajun Wu, Shawn Guo, Huahao Hu, Lingyi Sun, Jian Yang, Mingjie Tang, and Byran Dai. Close the loop: Synthesizing infinite tool-use data via multi-agent role-playing. *arXiv preprint arXiv:2512.23611*, 2025.

31 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*, 2024.

32 Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context, 2023. URL https://arxiv.org/abs/2310.01889.

33 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. *arXiv preprint arXiv:2305.01210*, abs/2305.01210, 2023. URL https://arxiv.org/abs/2305.01210.

34 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.

35 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.

36 Xianzhen Luo, Wenzhen Zheng, Qingfu Zhu, Rongyi Zhang, Houyi Li, Siming Huang, YuanTao Fan, and Wanxiang Che. Scaling laws for code: A more data-hungry regime. *arXiv preprint arXiv:2510.08702*, 2025.

37 Amirkeivan Mohtashami, Matteo Pagliardini, and Martin Jaggi. Cotformer: More tokens with attention make up for less depth. In *Workshop on Advancing Neural Network Training: Computational Efficiency, Scalability, and Resource Optimization (WANT@ NeurIPS 2023)*, 2023.

38 NVIDIA. Transformerengine, 2025. URL https://github.com/NVIDIA/TransformerEngine.

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

40 Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. *OpenAI blog*, 2018. URL https://openai.com/research/language-unsupervised.

41 Yangjun Ruan, Neil Band, Chris J Maddison, and Tatsunori Hashimoto. Reasoning to learn from latent thoughts. *arXiv preprint arXiv:2503.18866*, 2025.

42 Nikunj Saunshi, Nishanth Dikkala, Zhiyuan Li, Sanjiv Kumar, and Sashank J Reddi. Reasoning with latent thoughts: On the power of looped transformers. *arXiv preprint arXiv:2502.17416*, 2025.

43 Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, abs/2402.03300, 2024. doi: 10.48550/ARXIV.2402.03300. URL https://doi.org/10.48550/arXiv.2402.03300.

44 Liangcai Su, Zhen Zhang, Guangyu Li, Zhuo Chen, Chenxi Wang, Maojia Song, Xinyu Wang, Kuan Li, Jialong Wu, Xuanzhong Chen, et al. Scaling agents via continual pre-training. *arXiv preprint arXiv:2509.13310*, 2025.

45 Sho Takase and Shun Kiyono. Lessons on parameter sharing across layers in transformers. In *Proceedings of The Fourth Workshop on Simple and Efficient Natural Language Processing (SustaiNLP)*, pages 78–90, 2023.

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

47 Bohong Wu, Mengzhao Chen, Xiang Luo, Shen Yan, Qifan Yu, Fan Xia, Tianqi Zhang, Hongrui Zhan, Zheng Zhong, Xun Zhou, Siyuan Qiao, and Xingyan Bin. Parallel loop transformer for efficient test-time computation scaling, 2025. URL https://arxiv.org/abs/2510.24824.

48 Bohong Wu, Shen Yan, Sijun Zhang, Jianqiao Lu, Yutao Zeng, Ya Wang, and Xun Zhou. Efficient pretraining length scaling. *arXiv preprint arXiv:2504.14992*, 2025.

49 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.

50 Liu Yang, Kangwook Lee, Robert Nowak, and Dimitris Papailiopoulos. Looped transformers are better at learning learning algorithms. *arXiv preprint arXiv:2311.12424*, 2023.

51 Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

52 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.

53 Eric Zelikman, Georges Harik, Yijia Shao, Varuna Jayasiri, Nick Haber, and Noah D Goodman. Quiet-star: Language models can teach themselves to think before speaking. *arXiv preprint arXiv:2403.09629*, 2024.

54 Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, et al. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*, 2025.

55 Boyi Zeng, Shixiang Song, Siyuan Huang, Yixuan Wang, He Li, Ziwei He, Xinbing Wang, Zhiyu Li, and Zhouhan Lin. Pretraining language models to ponder in continuous space. *arXiv preprint arXiv:2505.20674*, 2025.

56 Rui-Jie Zhu, Zixuan Wang, Kai Hua, Tianyu Zhang, Ziniu Li, Haoran Que, Boyi Wei, Zixin Wen, Fan Yin, He Xing, et al. Scaling latent reasoning via looped language models. *arXiv preprint arXiv:2510.25741*, 2025.

57 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.