LoopCoder : Scaling Code Intelligence via Looped Language Models

Jian Yang[1], Wei Zhang[1], Shawn Guo[2], Yizhi Li[1], Lin Jing[1], Zhengmao Ye[2], Shark Liu[2], Yuyang Song[2], Jiajun Wu[1], Che Liu[2], Tunny Zheng[2], Leo L[2], Xi Lin[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;
{jiayang}@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 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). We present LoopCoder (40B-A80B) pre-trained on 12T+ code and general tokens, along with LoopCoder-Thinking and LoopCoder-Instruct variants, the first large-scale looped transformer for code, achieving comparable performance to standard dense architectures with more parameters. Unlike prior approaches that restrict recurrence to small-scale tasks, we implement a comprehensive looped training protocol spanning both pre-training and post-training phases. We initiate the model via dense-to-loop transformation, folding a pre-trained dense checkpoint to initialize a recurrent block, followed by rigorous looped pre-training and specialized post-training for instruction following and reasoning. 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) (Yang et al., 2025). The dominant recipe for improving this capability has been the dense scaling law (Hoffmann et al., 2022; Luo et al., 2025), 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.

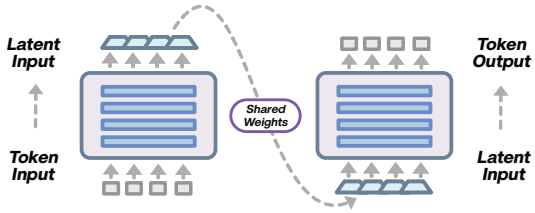

Figure 1: LoopCoder mechanism. During training and inference, the model recurrently predict output tokens with shared weights, using the latent representations produced by the previous iteration.

Theoretical frameworks like the universal Transformer (UT) (Dehghani et al., 2019) propose a more elegant solution: recurrence over depth. By sharing weights across layers, UTs introduce a recurrent inductive bias 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) (Gao et al., 2025), have validated this potential on abstract reasoning benchmarks like ARC-AGI (Chollet et al., 2025), 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 as shown in Figure 2, a large-scale looped coding model that not only matches but surpasses the convergence efficiency of equivalent dense mod-

* Corresponding Author.

Findings of the Association for Computational Linguistics: ACL 2026, pages 16209–16223

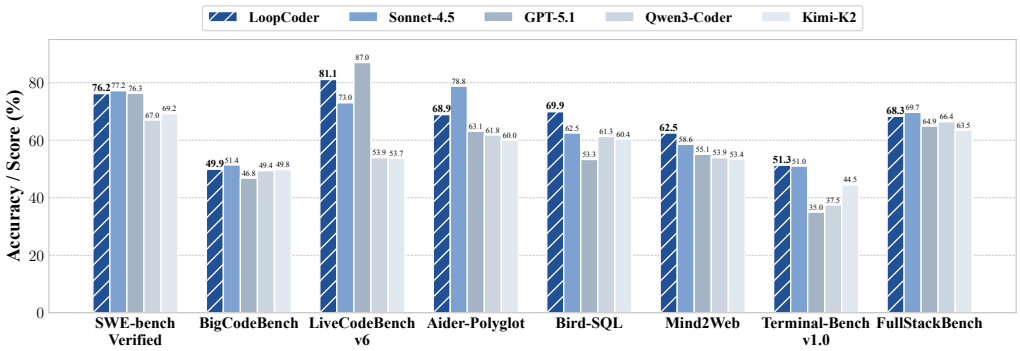

Figure 2: Benchmark performance of LoopCoder and its counterparts. We refer the LoopCoder-Thinking performance for LiveCodeBench v6 as its reasoning nature, and the performance of rest point to LoopCoder-Instruct.

els. We depart from the traditional view of recurrence as merely a fine-tuning trick. Instead, we propose a holistic full-lifecycle looped protocol: (1) Dense-to-Loop Initialization: Initialize recurrent blocks from folded pre-trained dense checkpoints to preserve syntactic knowledge while enabling weight-sharing. (2) Looped Pre-training: Continue pre-training with looped architecture on large code corpora, allowing latent representations to adapt to recurrent inductive bias. (3) Looped Post-training: Fine-tune through SFT and CoT, leveraging the loop structure as a natural carrier for reasoning processes.

Our contributions are as follows: (1) **Large-Scale Viability**: The first successful training of a large-scale looped transformer for code using a continuous pre-training and post-training protocol, achieving advanced performances on multiple coding tasks. (2) **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. (3) **Emergent Reasoning**: We show that LoopCoder, trained as a thinking model, can verify and refine code logic through dynamic recurrence, outperforming static dense models on complex algorithmic tasks.

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 (Wu et al., 2025b,a). 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):

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

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: (1) **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:

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

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

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

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

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

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

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.

Our 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 (Elman, 1990) 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 incur 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 (Liu et al., 2023a) enables the training of ultra-long context models by achieving global attention through the circulation of Key-Value (KV) shards. However, existing implementations face limitations: TransformerEngine (NVIDIA, 2025) lacks support for local attention mechanisms, while AllGather-based approaches (Grattafiori et al., 2024) 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 We construct a corpus from Common Crawl using a multi-stage pipeline: (1) regex-based cleaning to remove noise; (2) hierarchical deduplication via exact matching and embedding-based fuzzy methods; (3) benchmark decontamination; (4) AST analysis for code syntax validation. We train domain-specific quality classifiers for text, code, and math that outperform FastText by emulating larger models' assessments of information density, educational value, and toxicity. For code-related factuality, we generate 66M instruction samples during pre-training using LLMs to create objective, unambiguous, time-invariant QA pairs with single correct answers.

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) (Bavarian et al., 2022) 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 (Yang et al., 2024).

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

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

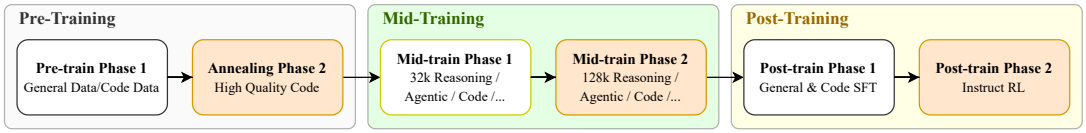

Figure 3: Training pipeline of LoopCoder .

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.

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 (Guha et al., 2025). 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

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.**

math proofs and code synthesis more stable under distribution shift (Zelikman et al., 2024; Su et al., 2025; Ruan et al., 2025).

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

A model-centric AI code training framework where LLMs generate verified training data across 10 domains: API orchestration (3-stage verification), full-stack engineering (TDD), competitive pro-

Model	Agentic Coding			General Tool Use		Aider-Polyglot		Mercury		Text2SQL	
	TB	TB 2.0	SWE-V	M2W	BFCL V3	Diff P@2	Whole P@2	Beyond@1	Pass@1	Bird	Spider
6B+ Models											
DeepSeek-Coder-V2-Lite-Instruct	5.0	0.0	-	26.7	-	1.3	2.2	76.8	91.4	41.6	72.4
Qwen2.5-Coder-7B-Instruct	6.3	0.0	-	38.4	54.2	1.8	4.9	69.9	84.8	53.1	79.8
Seed-Coder-8B-Instruct	7.5	2.5	-	38.2	-	6.2	5.3	78.5	93.8	44.7	72.7
13B+ Models											
Qwen2.5-Coder-14B-Instruct	8.8	0.0	-	42.7	59.9	8.0	8.0	76.7	88.3	59.1	81.3
Qwen3-Coder-30B-A3B-Instruct	23.8	23.8	51.9	36.1	63.4	28.4	29.3	81.1	95.3	59.0	80.9
20B+ Models											
DeepSeek-v3.2	23.8	**46.4**	73.1	47.2	68.8	-	-	81.6	**96.9**	52.6	77.9
Qwen2.5-Coder-32B-Instruct	5.0	4.5	-	32.5	62.3	8.4	14.7	79.1	96.1	62.1	83.9
Qwen3-235B-A22B-Instruct-2507	15.0	13.5	45.2	49.0	71.2	53.3	57.3	80.4	96.9	62.8	81.1
Qwen3-235B-A22B-Thinking-2507	8.8	3.4	44.6	43.2	71.9	25.3	-	61.2	70.3	35.2	42.6
Qwen3-Coder-480B-A35B-Instruct	37.5	23.6	67.0	54.0	68.7	57.8	61.8	80.2	96.1	61.3	81.2
Kimi-Dev-72B	-	2.3	60.4	-	55.5	12.0	20.0	59.1	69.5	-	-
Kimi-K2-Instruct-0905	44.5	27.8	69.2	53.4	70.3	60.0	50.7	76.1	90.6	60.4	81.1
Kimi-K2-Thinking	47.1	33.7	71.3	55.7	-	-	-	73.0	85.2	40.6	49.6
KAT-Dev	17.5	10.1	62.4	33.7	64.7	8.9	34.2	75.1	89.1	52.2	77.6
KAT-Dev-72B-Exp	21.3	7.9	74.6	-	-	16.4	15.6	79.0	94.5	35.2	60.3
GLM-4.7	36.3	41.0	73.8	53.7	64.8	-	-	74.1	86.7	46.5	62.4
DenseCoder-40B-Instruct	48.2	25.0	72.4	61.5	70.9	62.4	60.5	81.2	91.2	65.4	80.0
LoopCoder-40B-Instruct	51.3	33.0	**76.2**	62.5	**73.9**	**68.9**	**62.3**	**82.2**	94.1	**69.9**	**84.0**
Closed-APIs Models											
Gemini-3-Flash-preview	53.8	47.6	78.0	60.6	-	-	-	78.4	89.5	66.6	87.2
Gemini-3-Pro-preview	46.3	54.2	76.2	60.3	78.2	91.9	92.9	83.1	96.1	67.5	87.0
Claude-Opus-4.5	47.5	59.3	80.9	57.9	78.9	89.4	87.1	82.9	96.9	66.0	76.0
Claude-Sonnet-4.5	51.0	50.0	77.2	58.6	77.7	78.8	-	82.5	97.7	62.5	80.1
GPT-5.1	35.0	47.6	76.3	55.1	64.4	63.1	65.3	81.9	96.1	53.3	77.6

Table 3: Comprehensive evaluation across agentic coding (Terminal-Bench, Terminal-Bench 2.0, SWE-Verified), general tool use (Mind2Web, BFCL V3), code editing (Aider-Polyglot), code efficiency (Mercury), and Text2SQL (Bird/Spider). TB=Terminal-Bench, SWE-V=SWE-Verified, M2W=Mind2Web, P@2=Pass@2.

gramming (8K problems), code reasoning (constraint satisfaction), Text-to-SQL (inverse generation), code editing (commit diffs), terminal ops (Docker verification), repo-level engineering (Issue-PR matching, 30% to 70% build success), tool usage (4-stage progressive), and GUI agents (multimodal web). Features execution verification, subjective domain integration, complete dev trajectories with failures, multi-agent collaboration, and hybrid SL+RL training. More Details can seed in Appendix.

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 (Shao et al., 2024) algorithm, remove the KL penalty, and incorporate the Clip-Higher strategy inspired by DAPO (Yu et al., 2025) 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 (Shao et al., 2024), 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.

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 Evaluation Strategy

To comprehensively assess the capabilities of our framework, we conduct extensive evaluations on both the base and instruct/reasoning model variants across a diverse spectrum of coding tasks. Our testing protocol spans repository-level code completion, functional generation, code reasoning, multilingual editing, execution efficiency, Text-to-SQL, and agentic software engineering workflows. As

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
DenseCoder-40B-Instruct	46.2	78.4	56.4	84.1	56.0	84.2	61.2	83.1	54.5	82.4
LoopCoder-40B-Instruct	49.0	81.7	**57.9**	86.2	**61.9**	**88.5**	63.4	85.5	**57.8**	85.7

Table 4: Performance on CrossCodeEval Tasks.

Model	EvalPlus		BigCodeBench		FullStackBench
	HumanEval (+)	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	96.3 (91.5)	92.3 (77.8)	47.4	25.7	62.7
Qwen3-235B-A22B-Thinking	**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
DenseCoder-40B-Instruct	94.2 (88.4)	89.4 (74.2)	44.9	25.7	62.3
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 5: Performance on Code Generation tasks.

demonstrated in Tables 3, 4, 5, 6, as well as Figure 2, our models achieve consistently superior performance across these benchmarks, highlighting their robustness in handling both fundamental programming problems and complex, long-horizon software development scenarios. Details of each benchmark and additional experimental results are provided in the Appendix.

5 Related Work

Code foundation models have evolved from pre-training to full-stack systems (Lozhkov et al., 2024; Li et al., 2023b; Hui et al., 2024; Liu et al., 2024a; Zeng et al., 2025a; Anthropic, 2023; Radford et al., 2018). Looped Transformers reuse blocks across depth, trading parameters for recurrent computation (Dehghani et al., 2019), explored for structured procedures (Giannou et al., 2023; Yang et al., 2023; Gatmiry et al., 2024a,b), iterative refinement (Saunshi et al., 2025; Geiping et al., 2025; Zeng et al., 2025b; Chen et al., 2025; Bae et al., 2025), and efficiency (Lan et al., 2019; Dabre and Fujita, 2019;

Model	CruxEval		LiveCodeBench	
	I-COT	O-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**
DenseCoder-40B-Instruct	89.2	81.3	44.2	43.5
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 on Code Reasoning Evaluation.

Takase and Kiyono, 2023; Bae et al., 2024; Li et al., 2025a). Recent work enhances refinement (Hao et al., 2024; Mohtashami et al., 2023) and studies compute tradeoffs (Wu et al., 2025b; Banino et al., 2021). We introduce LoopCoder with dense-to-loop initialization, looped pre-training, and post-training, mitigating BPTT instabilities via gradient-scale reduction. Building on (Zhu et al., 2025), we provide an end-to-end recipe for reliable code recurrence with inference-time over-thinking.

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 and specialized post-training for instruction following and latent reasoning, we demonstrate that recurrent architectures can achieve superior convergence than traditional dense models. Our approach establishes that the recursive nature of UT provides a natural inductive bias for algorithmic reasoning, enabling emergent over-thinking capabilities during inference without the optimization instabilities traditionally associated with back propagation through time.

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.

Ethical Considerations

Our primary objective is to prove the scaling potential of the looped transformer architecture mainly in the complex code and reasoning domain. The system produces dialogue artifacts and code snippets, enabling automation for code tasks. The system should be used as a supplementary tool to aid, rather than substitute independent intellectual efforts.

References

Anthropic. 2023. Introducing Claude.

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

Sangmin Bae, Yujin Kim, Reza Bayat, Sungnyun Kim, Jiyoun Ha, Tal Schuster, Adam Fisch, Hrayr Harutyunyan, Ziwei Ji, Aaron Courville, and 1 others. 2025. Mixture-of-recursions: Learning dynamic recursive depths for adaptive token-level computation. *arXiv preprint arXiv:2507.10524*.

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

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *Preprint*, arXiv:2207.14255.

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

Francois Chollet, Mike Knoop, Gregory Kamradt, and Bryan Landers. 2025. Arc prize 2024: Technical report. *Preprint*, arXiv:2412.04604.

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

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. 2019. Universal transformers. *Preprint*, arXiv:1807.03819.

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. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Preprint*, arXiv:2310.11248.

Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. 2024. Mercury: A code efficiency benchmark for code large language models. *Preprint*, arXiv:2402.07844.

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

Zitian Gao, Lynx Chen, Yihao Xiao, He Xing, Ran Tao, Haoming Luo, Joey Zhou, and Bryan Dai. 2025. Universal reasoning model. *Preprint*, arXiv:2512.14693.

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

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

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

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

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, and 542 others. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

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

Etash Guha, Ryan Marten, Sedrick Keh, Negin Raoof, Georgios Smyrnis, Hritik Bansal, Marianna Nezhurina, Jean Mercat, Trung Vu, Zayne Sprague, and 1 others. 2025. Openthoughts: Data recipes for reasoning models. *arXiv preprint arXiv:2506.04178*.

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

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, and 1 others. 2022. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. 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.

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

Boxun Li, Yadong Li, Zhiyuan Li, Congyi Liu, Weilin Liu, Guowei Niu, Zheyue Tan, Haiyang Xu, Zhuyu Yao, Tao Yuan, and 1 others. 2025a. Megrez2 technical report. *arXiv preprint arXiv:2507.17728*.

Jinyang Li, Binyuan Hui, Ge Qu, Binhua Li, Jiaxi Yang, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, and 1 others. 2023a. Can llm already serve as a database interface. *A big bench for large-scale database grounded text-to-sqls. CoRR, abs/2305.03111*.

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, and 48 others. 2023b. StarCoder: May the source be with you! *arXiv preprint arXiv:2305.06161*, abs/2305.06161.

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

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024a. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Hao Liu, Matei Zaharia, and Pieter Abbeel. 2023a. Ring attention with blockwise transformers for near-infinite context. *Preprint*, arXiv:2310.01889.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023b. 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.

Siyao Liu, Ge Zhang, Boyuan Chen, Jialiang Xue, and Zhendong Su. 2024b. FullStack Bench: Evaluating llms as full stack coders. *arXiv preprint arXiv:2412.00535*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

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

Amirkeivan Mohtashami, Matteo Pagliardini, and Martin Jaggi. 2023. 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)*.

NVIDIA. 2025. Transformerengine.

polyglot-benchmark. 2025. polyglot-benchmark.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, and 1 others. 2018. Improving language understanding by generative pre-training. *OpenAI blog*.

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

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

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

Liangcai Su, Zhen Zhang, Guangyu Li, Zhuo Chen, Chenxi Wang, Maojia Song, Xinyu Wang, Kuan Li, Jialong Wu, Xuanzhong Chen, and 1 others. 2025. Scaling agents via continual pre-training. *arXiv preprint arXiv:2509.13310*.

Sho Takase and Shun Kiyono. 2023. 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.

The Terminal-Bench Team. 2025. Terminal-bench: A benchmark for ai agents in terminal environments.

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. 2025a. Parallel loop transformer for efficient test-time computation scaling. *Preprint*, arXiv:2510.24824.

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

Jian Yang, Xianglong Liu, Weifeng Lv, Ken Deng, Shawn Guo, Lin Jing, Yizhi Li, Shark Liu, Xianzhen Luo, Yuyu Luo, and 1 others. 2025. From code foundation models to agents and applications: A comprehensive survey and practical guide to code intelligence. *arXiv preprint arXiv:2511.18538*.

Jian Yang, Jiajun Zhang, Jiaxi Yang, Ke Jin, Lei Zhang, Qiyao Peng, Ken Deng, Yibo Miao, Tianyu Liu, Zeyu Cui, and 1 others. 2024. Execrepobench: Multi-level executable code completion evaluation. *arXiv preprint arXiv:2412.11990*.

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

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, and 1 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, and 1 others. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*.

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

Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, and 1 others. 2025a. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*.

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

Rui-Jie Zhu, Zixuan Wang, Kai Hua, Tianyu Zhang, Ziniu Li, Haoran Que, Boyi Wei, Zixin Wen, Fan Yin, He Xing, and 1 others. 2025. Scaling latent reasoning via looped language models. *arXiv preprint arXiv:2510.25741*.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

A 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 (Lozhkov et al., 2024; Li et al., 2023b). 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 (Hui et al., 2024)) and large-scale MoE families that report stable training recipes and competitive code performance (for example, DeepSeek-V3 (Liu et al., 2024a),GLM4.7 (Zeng et al., 2025a)). 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 (Anthropic, 2023; Radford et al., 2018). 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.

A.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 (Dehghani et al., 2019), 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 (Giannou et al., 2023; Yang et al., 2023); complementary studies analyze how looped computation can implement multi-step optimization dynamics for in-context learning (Gatmiry et al., 2024a,b). On the reasoning side, looped architectures have been connected to iterative latent refinement (Saunshi et al., 2025), while later systems scale or adapt effective depth through recurrent-depth or dynamic-compute mechanisms (Geiping et al., 2025; Zeng et al., 2025b; Chen et al., 2025; Bae et al., 2025). In parallel, parameter sharing has long been used as an efficiency strategy (Lan et al., 2019; Dabre and Fujita, 2019; Takase and Kiyono, 2023), and recent work revisits it for modern LLMs via lightweight adaptation or large-scale shared-depth training (Bae et al., 2024; Li et al., 2025a).

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 (Hao et al., 2024; Mohtashami et al., 2023), while other work studies compute–length tradeoffs (Wu et al., 2025b) or dynamic halting objectives (Banino et al., 2021). 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 (Zhu et al., 2025), but focus on an end-to-end recipe that makes recurrence reliable for code and yields over-thinking behaviors at inference time.

B 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 anno-

tations 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 (Li et al., 2025b) 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 syn-

thetic 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.

C Evaluation

C.1 Experiments on Base Models

C.1.1 Code Completion

We evaluate cross-file code completion on CrossCodeEval (Ding et al., 2023), 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.

C.2 Evaluation on Instruct Models and Reasoning model

C.2.1 Code Generation

Across a wide range of code-generation evaluations, our model achieves consistently strong performance. We validate functional correctness and robustness using EvalPlus (Liu et al., 2023b) (including HumanEval+ and MBPP+ with substantially expanded test suites), and measure compositional, library-intensive problem solving on BigCodeBench (Zhuo et al., 2024). We further demonstrate broad full-stack capability on FullStackBench (Liu et al., 2024b), and strong results under contamination-aware, continuously refreshed testing on LiveCodeBench (Jain et al., 2024).

C.2.2 Code Reasoning

We further evaluate code reasoning with CRUXEval (Gu et al., 2024), 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.

C.2.3 Code Editing

We evaluate code editing on Aider's Polyglot benchmark (polyglot-benchmark, 2025), 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.

C.2.4 Code Efficiency

We assess code efficiency with Mercury (Du et al., 2024), 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.

C.2.5 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 (Yu et al., 2018), which uses a database-level train–test split to evaluate schema linking and structurally correct SQL generation with complex constructs, and on BIRD (Li et al., 2023a), 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.

C.2.6 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 (Team, 2025) 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 (Jimenez et al., 2024) 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 depends on correct tool use, long-horizon planning, and tight interaction with the execution environment. Terminal-Bench (Team, 2025) 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 (Jimenez et al., 2024) 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.