ByteDance | Seed

# Stable-DiffCoder: Pushing the Frontier of Code Diffusion Large Language Model

**Chenghao Fan**[1,2], **Wen Heng**[2], **Bo Li**[2], **Sichen Liu**[1], **Yuxuan Song**[2], **Jing Su**[2], **Xiaoye Qu**[1], **Kai Shen**[2], **Wei Wei**[1]

[1]Huazhong University of Science and Technology, [2]ByteDance Seed

## Abstract

Diffusion-based language models (DLLMs) offer non-sequential, block-wise generation and richer data reuse compared to autoregressive (AR) models, but existing code DLLMs still lag behind strong AR baselines under comparable budgets. We revisit this setting in a controlled study and introduce Stable-DiffCoder, a block diffusion code model that reuses the Seed-Coder architecture, data, and training pipeline. To enable efficient knowledge learning and stable training, we incorporate a block diffusion continual pretraining (CPT) stage enhanced by a tailored warmup and block-wise clipped noise schedule. Under the same data and architecture, Stable-DiffCoder overall outperforms its AR counterpart on a broad suite of code benchmarks. Moreover, relying only on the CPT and supervised fine-tuning stages, Stable-DiffCoder achieves stronger performance than a wide range of ∼8B ARs and DLLMs, demonstrating that diffusion-based training can improve code modeling quality beyond AR training alone. Moreover, diffusion-based any-order modeling improves structured code modeling for editing and reasoning, and through data augmentation, benefits low-resource coding languages.

**Date:** January 12, 2026
**Correspondence:** Wei Wei at weiw@hust.edu.cn
**Project Page:** https://github.com/ByteDance-Seed/Stable-DiffCoder
**Model Page:** https://huggingface.co/collections/ByteDance-Seed/stable-diffcoder

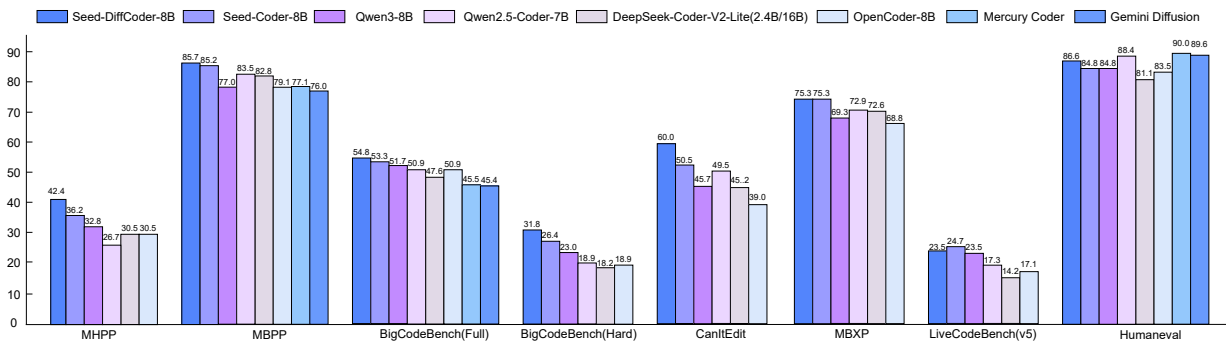



**Figure 1** Benchmark performance of Stable-DiffCoder-8B-Instruct.

# 1 Introduction

Autoregressive (AR) language models have achieved strong results in natural language and code by modeling sequences left-to-right [13, 21, 22, 31, 49, 61, 67], but their strictly sequential decoding underutilizes the inherently non-autoregressive nature of code, where developers routinely infill missing spans, iterate from a scratch toward the final version, revise earlier segments using later context, and generate independent blocks in parallel [23, 29, 58]. To better align with this paradigm, diffusion-based language models (DLLMs) [4, 43] provide a complementary view by treating generation as iterative denoising: they corrupt and reconstruct sequences under random masking, enabling non-sequential, block-wise decoding and implicitly augmenting each clean example with many arbitrary corruption patterns, which is especially appealing for rare high-quality or long-tail code samples [14, 41]. Recently, mask-based DLLMs that instantiate this corruption-and-denoising view have gained attention for their potential in fast parallel generation and high inference speed ceilings [30, 38, 40, 51, 57, 60]. In this work, we therefore focus on such mask-based DLLMs and use "training modes" to refer to the different corruption or masking patterns sampled for each example. Despite this focus on efficiency, their impact on modeling quality remains unclear, leaving open whether diffusion can systematically improve model capability. ln principle, this stochastic training should help models absorb scarce supervision and generalize beyond specific surface forms, yet existing diffusion-based code models still lag behind strong AR baselines in overall accuracy [25, 51, 62], and prior work often changes data, architecture, and training pipeline simultaneously [56, 59], leaving open a central question: *under a fixed data and compute budget, can the additional training modes introduced by diffusion actually improve model capability?*

In this work, we systematically study how to enable diffusion models to learn new knowledge more efficiently, by conducting controlled experiments on a 2.5B model size. Building on the observations, we propose Stable-DiffCoder, a diffusion language model that reuses the autoregressive Seed-Coder [49] pipeline and data but achieves stronger performance. Starting from the AR Seed-Coder checkpoint before annealing, we perform continual pretraining on 1.3T tokens using small-block diffusion (block size 4), preserving representation plasticity while introducing multi-token prediction patterns. To make the training process more suitable for DLLM, we design (i) the tailored warmup training process, (ii) a block-wise clipped noise schedule that guarantees non-trivial supervision within each block. Finally, we carry out a comprehensive evaluation on a broad suite of code benchmarks. Across both base and instruction-tuned settings, Stable-DiffCoder almost uniformly surpasses its AR counterpart Seed-Coder of the same size, and establishes new state-of-the-art results among 8B-scale diffusion code models on many metrics. These results suggest that, when equipped with an appropriate curriculum and training design, diffusion language models can not only match but exceed autoregressive models in code understanding and generation quality, even before factoring in any potential inference-time speed advantages.

# 2 Preliminary

## 2.1 Autoregressive Language Models

Transformer-based [54] AR models have been the dominant paradigm in language modeling, powering many state-of-the-art code and text generation systems [13, 21, 22, 31, 49, 61, 67]. Let $\mathcal{V}$ be a finite vocabulary and $\mathbf{x}_{1:T} = (x_1, \ldots, x_T) \in \mathcal{V}^T$ a token sequence. In an AR language model, it is assumed that each token is conditioned on all preceding tokens, so that the joint distribution is factorized in a left-to-right manner as

$$p_\theta(\mathbf{x}_{1:N}) = p_\theta(\mathbf{x}_1) \prod_{i=2}^{N} p_\theta(\mathbf{x}_i \mid \mathbf{x}_{<i}) \tag{1}$$

where $x_t$ is the $t$-th token in the sequence. We typically parameterize $p_\theta$ with a Transformer, and train by maximizing token-level cross-entropy. Concretely, given data $x \sim p_{\text{data}}$, the training loss is:

$$\mathcal{L}_{\text{AR}}(\theta) = -\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}} \Big[ \log p_\theta(\mathbf{x}_1) + \sum_{i=2}^{N} \log p_\theta(\mathbf{x}_i \mid \mathbf{x}_{<i}) \Big] \tag{2}$$

This is the standard next-token prediction objective widely used in mainstream AR models. Notable AR transformers like GPT-series [6, 31, 61] and code-specific LLMs [22, 49] have achieved high accuracy on

programming benchmarks, but their token-by-token generation can be slow and lacks a mechanism for global planning or simultaneous reasoning over multiple parts of the sequence. This limitation has motivated exploration of non-AR generation alternatives.

### 2.2 Masked Diffusion Language Models

Masked diffusion language models (DLLMs) have recently emerged as a promising non-AR alternative for text and code generation [5, 10, 18, 28, 43, 51, 62, 65]. Inspired by continuous diffusion models in vision, discrete diffusion LMs optimize a variational evidence lower bound by corrupting observed sequences (e.g., by randomly masking tokens) and training a denoiser to reconstruct the clean data [4, 36, 39]. This framework provides a principled likelihood-based foundation for parallel or partially parallel text generation, which earlier heuristic non-AR methods often lacked [26, 33, 58].

Instead of factorizing the distribution autoregressively, DLLMs define a sequence of latent variables $\mathbf{x}^0_{1:N}, \mathbf{x}^1_{1:N}, \ldots, \mathbf{x}^T_{1:N}$, where $\mathbf{x}^0_{1:N}$ denotes the clean text and $\mathbf{x}^T_{1:N}$ denotes a fully corrupted state. In DLLMs, the corruption is instantiated via a dedicated [MASK] token. The forward noising process progressively replaces tokens with [MASK], while the reverse denoising process iteratively reconstructs the clean text step by step.

$$p_\theta(\mathbf{x}_{1:N}) = p_\theta(\mathbf{x}^T_{1:N}) \prod_{t=1}^{T} p_\theta(\mathbf{x}^{t-1}_{1:N} \mid \mathbf{x}^t_{1:N}) \tag{3}$$

where $p_\theta(\mathbf{x}^T_{1:N})$ is typically a uniform prior over noise states, and $p_\theta(\mathbf{x}^{t-1}_{1:T} \mid \mathbf{x}^t_{1:T})$ is parameterized by a neural network.

In particular, for DLLMs, prior theoretical analyses [44, 47, 50] have shown that this variational denoising objective can be equivalently reduced to a weighted cross-entropy loss, which enables efficient training in practice:

$$\mathcal{L}_{\text{DLLM}}(\theta) = -\mathbb{E}_{\mathbf{x}^0 \sim p_{\text{data}},\, t \sim \mathcal{U}(0,1),\, \mathbf{x}^t \sim q(\mathbf{x}^t \mid \mathbf{x}^0)} \left[ w(t) \sum_{i=1}^{N} 1[\mathbf{x}^t_i = \text{MASK}] \, \log p_\theta(\mathbf{x}^0_i \mid \mathbf{x}^t_{1:N}) \right] \tag{4}$$

where $q(\mathbf{x}^t \mid \mathbf{x}^0)$ denotes the corruption process, $1[\mathbf{x}^t_i = \text{[MASK]}]$ is the indicator function specifying whether the $i$-th token is masked at timestep $t$, and $w(t)$ is a weighting coefficient that depends on the corruption level. For example, under a linear noise schedule one may set $w(t) = \frac{1}{t}$. This formulation allows DLLMs to leverage a wide range of advanced model architectures, requiring only minor modifications to the original implementation in order to support diffusion language model's training. Recently, several works combine AR structure with diffusion to form block diffusion variants of DLLMs [1, 5, 10, 33, 34, 53, 56, 57]. Instead of corrupting the whole sequence with random masks, these models only diffuse a contiguous block of tokens at each step while keeping the surrounding context clean, so that the denoiser learns to generate a span conditioned on a mostly uncorrupted prefix.

Beyond offering a higher ceiling for parallel decoding speed [18, 28, 38, 51, 56], diffusion-style training also enables repeated reuse of the same underlying examples under diverse corruption trajectories, which can extract more information from rare high-quality and long-tail samples and potentially improve overall model capability [14, 41]. Our work aims to probe this question in the code domain under controlled conditions: given the same data, architecture, and training pipeline as a strong AR baseline, we introduce a block diffusion stage and investigate whether the additional training modes provided by diffusion can translate into tangible gains in code understanding and generation accuracy, rather than merely offering a different decoding mechanism.

## 3 Approach

### 3.1 Efficient Knowledge Compression and Training-Inference Alignment in DLLMs

**Motivation.** Although random masking can dramatically improve data reusability for diffusion language models [14, 41] by repeatedly revisiting the same original examples with different mask patterns applied, it imposes a substantial computational cost. Moreover, some mask patterns only demonstrate negligible training utility, as

shown in recent work [26]. In particular, many masked tokens are trained in contexts where the correct answer is only weakly constrained, and the resulting gradients are dominated by noisy token co-occurrence rather than a sharp reasoning signal. Moreover, even when the model compresses knowledge efficiently under some training contexts, this does not automatically translate into good performance under the inference contexts, where a mismatch between the training and inference paradigms has been placed. We next formalize this effect and use it to motivate practical training curricula.

### 3.1.1 Token Reasoning Knowledge under Random Masking

In the RADD [44] formulation, the concrete score at time $t$ for coordinate $i$ takes the below form:

$$s^*(x_t, t)_{(i,\hat{x}_i)} \;=\; \alpha(t)\, p_0\big(\hat{x}_i \mid x_t^{\mathrm{UM}}\big), \tag{5}$$

where $x_t^{\mathrm{UM}}$ denotes all unmasked tokens at time $t$, and $\alpha(t)$ depends only on $t$ and the forward kernel. The time dependence is thus factored out, and what remains is the clean-data conditional $p_0(\hat{x}_i \mid c)$ for various contexts $c$ induced by random masks.

However, not every masked context can lead to a faithful reasoning unmask order towards the final answer. Consider a clean sequence:

$$\texttt{a = 1, b = 2, a + b = 3;a = 3, b = 4, a + b = 7.} \tag{6}$$

If we heavily mask the middle and final parts to obtain:

$$\texttt{a = 1, b = 2, [MASK}_1\texttt{] ...[MASK}_2\texttt{] a + b = [MASK}_n\texttt{]}, \tag{7}$$

the last $\texttt{[MASK}_n\texttt{]}$ cannot by itself teach the rule that `a + b` equals the sum of the preceding `a` and `b`, because the model never sees clean evidence for the pair `a = 3, b = 4`. From this particular corrupted view, the model mainly learns that some number tends to appear after `a + b =`, and that certain pairs like `a = 3, b = 4` and `7` co-occur. From the clean evidence the model has access to, the correct answer should be 3, yet the masked training signal biases the model toward predicting 7, creating a contradictory and misleading supervision signal. By contrast, in a sample like below:

$$\texttt{a = 1, b = 2, a + b = 3} \quad \texttt{masked as} \quad \texttt{a = 1, b = 2, a + b = [MASK].} \tag{8}$$

the context strongly constrains the answer. In this way, across many such examples, the model can discover a stable mapping from evidence to output, which corresponds to an arithmetic reasoning rule rather than mere memorization.

**Definition 3.1 (Token Reasoning Knowledge)** *Let $x$ be a clean training sequence sampled from the real data distribution $p_0$, and $c$ denote a context extracted from $x$ that is used to predict a single token $x_i$ (similarly for multiple tokens). For autoregressive training, $c_{\mathrm{AR}} = x_{<i}$, and for diffusion with random masks, $c_t^i = x_t^{\mathrm{UM}}$. Given a fixed context $c$, the clean-data conditional $p_0(\cdot \mid c)$ induces a candidate set:*

$$\mathcal{C}(c) = \big\{v \in \mathcal{V} \;:\; p_0(v \mid c) \geq \varepsilon\big\}, \qquad K(c) = |\mathcal{C}(c)|, \quad \varepsilon > 0. \tag{9}$$

*We define the token reasoning knowledge contained in a training example with context $c$ as the conditional distribution of the next token restricted to its candidate set, denoted compactly by $p_0\big(\mathcal{C}(c) \mid c\big)$*

The model's objective is to recover this token reasoning knowledge by learning conditionals $p_\theta(x_i^0 \mid c)$ such that, $p_\theta\big(\mathcal{C}(c) \mid c\big) \;\approx\; p_0\big(\mathcal{C}(c) \mid c\big)$. Concretely, training provides randomly sampled pairs $(c, x_i^0)$ obtained from $(x, i)$ under $p_0$, and the model adjusts $p_\theta(\cdot \mid c)$ based on the empirical distribution of these pairs. The difficulty of learning this knowledge is analogous to a multi-class classification problem [12, 27]: (1) the size of the candidate set $K(c)$; (2) how often each context–label pair $(c, x_i^0)$ appears in the training process.

With respect to $K(c)$, we can distinguish three qualitative regimes:

- **Reasoning regime.** $K(c)$ is small and the ground-truth token $x_i^0$ has large probability under $p_0(\cdot \mid c)$. The mapping $c \mapsto x_i^0$ is almost deterministic, so repeated samples with similar $c$ provide highly aligned gradients and quickly reinforce a stable reasoning rule.

- **Correlation regime.** $K(c)$ is moderate or large, and $x_i^0$ is one of many plausible candidates with comparable probability. The model still learns that $x_i^0$ correlates with patterns in $c$, but samples with similar $c$ often yield different targets, so gradients partially cancel and updates primarily fit noisy co-occurrence rather than a sharp rule.
- **Noise regime.** $K(c)$ is very large (on the order of $|\mathcal{V}|$) and $p_0(\cdot \mid c)$ is nearly flat. The context contains almost no information about $x_i^0$, and the model can only memorize idiosyncratic pairs $(c, x_i^0)$.

Because $p_0(\cdot \mid c)$ is defined with respect to the real data distribution, extremely unnatural or noisy contexts $c$ have negligible probability and can be ignored in expectation, as clarified in previous works [15, 64]. As more relevant evidence is revealed in $c$ (for example, longer and cleaner context), the conditional entropy $H(X_i \mid c)$ cannot increase, and the candidate set size $K(c)$ typically shrinks.

For autoregressive architectures and block diffusion with small block sizes, the context $c$ usually contains long, contiguous, and clean left-side evidence, which also matches the inherently autoregressive structure of natural language [48]. As a result, sampled contexts tend to have relatively small $K(c)$ and allow token reasoning knowledge to be compressed efficiently. In contrast, fully bidirectional or large-block diffusion readily produces contexts similar to Eq. 7 that fall into the correlation or noise regimes, dramatically reducing the efficiency of knowledge compression.

**Context–label Pair $(c, x_i^0)$.** To reliably learn the token reasoning knowledge associated with a context $c$, the model needs a sufficient number of informative context–label pairs $(c, x_i^0)$. DLLMs based on random masking generate a wide variety of contexts $c$, which greatly increases the number of context–label pairs $(c, x_i^0)$ that the model must learn [55]. If the total training data is fixed, this diversification reduces the number of effective pairs corresponding to any specific piece of knowledge, lowering its learning efficiency and forcing the model to relearn the same underlying knowledge many more times than an AR model would require [42, 52, 55]. Although such random contexts can be viewed as a form of data augmentation, our analysis of the size of $K(c)$ shows that many of these contexts fall into the correlation or noise regimes and therefore cannot be mapped to a clear reasoning rule. In some cases, such as the pattern in Eq. 7, the masked context $c$ even encourages learning a wrong association: the model is asked to predict a target under incomplete or misleading evidence. Such problematic contexts appear frequently when clean evidence is heavily disrupted, for example in purely bidirectional or large-block diffusion settings.

Moreover, the knowledge compressed during training is only useful at inference if it is associated with contexts that actually occur at test time. Let $\mathcal{C}_{\text{train}}$ denote the set (or distribution) of contexts $c$ sampled during training, and let $\mathcal{C}_{\text{infer}}$ denote the contexts encountered along inference trajectories. Good performance requires $\mathcal{C}_{\text{train}}$ and $\mathcal{C}_{\text{infer}}$ to be as close as possible. For instance, when training with block diffusion of block size $B$ and using left-to-right block-wise decoding of size $B$ at inference, the inference contexts closely match the training contexts, which makes the learned token reasoning knowledge directly applicable at test time.

In summary, to efficiently learn new knowledge while ensuring that the data augmentation induced by DLLM masking remains effective, two conditions should be satisfied: (1) the model should be exposed to clean and reliable reasoning evidence so that clear reasoning rules can be learned, and (2) the number of distinct sampled contexts $c$ should not grow excessively, and their form should align as closely as possible with the contexts encountered during inference. Based on these principles, we next design experiments to explore more suitable training pipelines.

### 3.1.2 Curriculum Design and Empirical Results at 2.5B Scale

Since initializing from an AR checkpoint has become a standard and effective practice, we also start from a 2.5B AR model trained on general-domain data, and use code data as new CPT data. Starting from this AR checkpoint, we consider the following curricula for learning new knowledge under the same compute budget, and evaluate them using block-wise decoding in the style of LLaDA [43]:

(1) **AR → BiDLLM**: continue pure AR training on the new data, then perform CPT to a bidirectional DLLM;

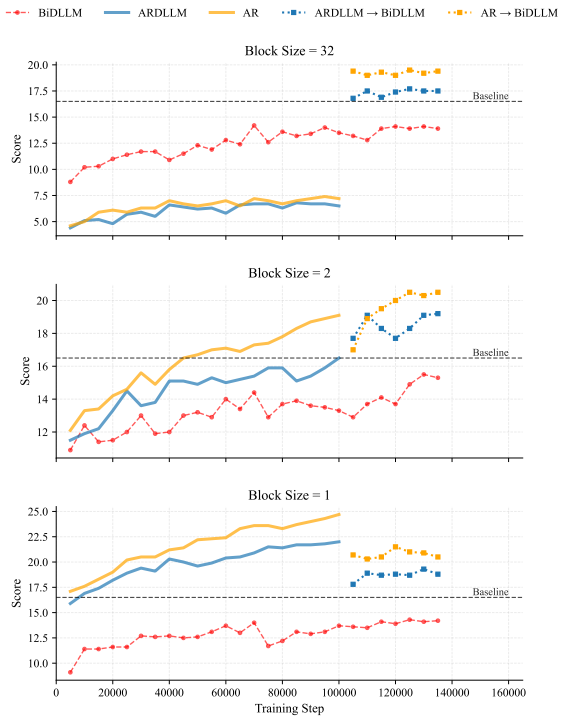



**Figure 2** Training dynamics under different block sizes. We compare **ARDLLM** and **AR** against a **BiDLLM** baseline for block sizes 32, 2, and 1 (top to bottom). Solid lines indicate training up to 100k steps, while dotted lines denote continued training after switching to BiDLLM. The horizontal dashed line marks the fixed baseline reference (0-step AR).

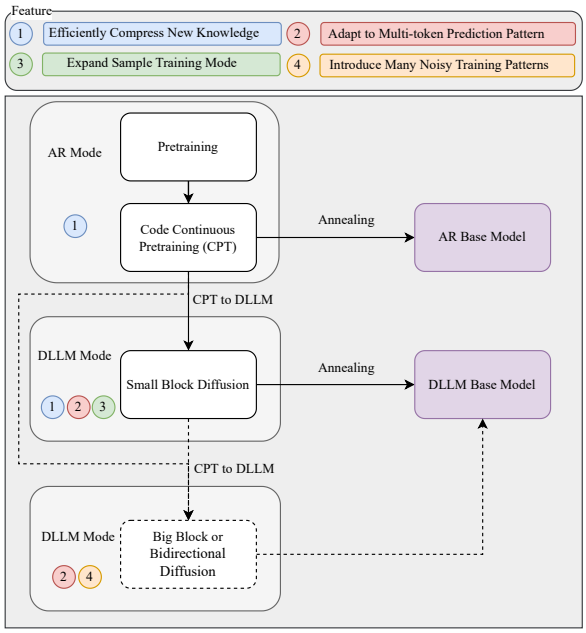



**Figure 3** As shown by the black solid line, we initialize from the pre-annealing checkpoint of Seed-Coder and perform CPT with small-block DLLM to obtain Stable-DiffCoder-Base, aiming to study efficient knowledge acquisition in diffusion-based models. The dashed line denotes an alternative pipeline for larger blocks, where new knowledge is first compressed using an AR model or a small-block DLLM before being transferred to the large-block diffusion setting.

(2) **ARDLLM** $\rightarrow$ **BiDLLM**: continue training with a causal-structured DLLM (AR-style diffusion), then CPT to a bidirectional DLLM;

(3) **BiDLLM**: directly CPT the AR checkpoint into a bidirectional DLLM and train new knowledge in that regime.

As shown in Fig. 2, we report the average performance across multiple code benchmarks. Under the same compute budget and after continued pre-training (CPT), the overall performance follows the ranking (1) $>$ (2) $>$ (3) under the same compute.

Scheme (1) preserves a purely AR training structure before CPT. Under small-block decoding (block size 1 or 2), its $\mathcal{C}_{\text{infer}}$ are almost identical to its $\mathcal{C}_{\text{train}}$ and satisfy the two principles from the previous section: the model sees clean reasoning evidence, and the number of distinct contexts remains controlled and well aligned with small-block inference. Consequently, before CPT to BiDLLM, scheme (1) consistently achieves the best performance for small blocks. For large-block decoding (block size 32), however, scheme (1) underperforms scheme (3), because the training context distribution $\mathcal{C}_{\text{train}}$ of pure AR does not cover the bidirectional patterns required by large-block decoding. Formally, with causal attention, tokens later in the block cannot influence earlier ones, so when decoding an entire block at once, the generation process becomes less stable.

After CPT into a bidirectional DLLM, scheme (1) performs well across all block sizes. This suggests that the

AR phase has already compressed the new token reasoning knowledge effectively in a well-aligned context family for small blocks, and the subsequent BiDLLM CPT primarily adapts the model to the larger-block context distribution.

It is worth noting that CPT into a fully bidirectional model introduces some degradation for block-1 decoding: both pure AR and AR→BiDLLM lose a similar amount of performance between 0 and 100k CPT steps at block size 1. However, the results also show that scheme (1) achieves higher learning efficiency during the AR phase, so even after the CPT-induced drop, it still attains better final performance.

Interestingly, scheme (2) also achieves strong performance after CPT. In particular, after converting to BiDLLM, it outperforms scheme (3) under block-32 decoding, despite having seen a similar amount (or even less) clean evidence during training. One explanation is that the training context distribution $\mathcal{C}_{\text{train}}$ for AR-style diffusion in scheme (2) more closely matches the prompt–response pattern commonly used at inference, where most reasoning evidence lies on the left. Thus scheme (2) enjoys better training–inference alignment than scheme (3), and this advantage persists even when both are eventually converted to a fully bidirectional DLLM.

Finally, based on the above theoretical analysis and empirical results, we recommend the following training procedure:

1. For new knowledge, first use AR training to efficiently compress it;
2. Then perform CPT with a small-block diffusion objective, leveraging its data-augmentation properties to further improve model quality;
3. if one wishes to explore larger block diffusion, additional CPT can be applied starting from the model obtained in step (2).

In our final system, we adopt steps (1) and (2) to train Stable-DiffCoder, as illustrated by the solid path leading to the "DLLM Base Model" in Fig. 3.

### 3.2 Warmup for Stable DLLM Continued Pretraining

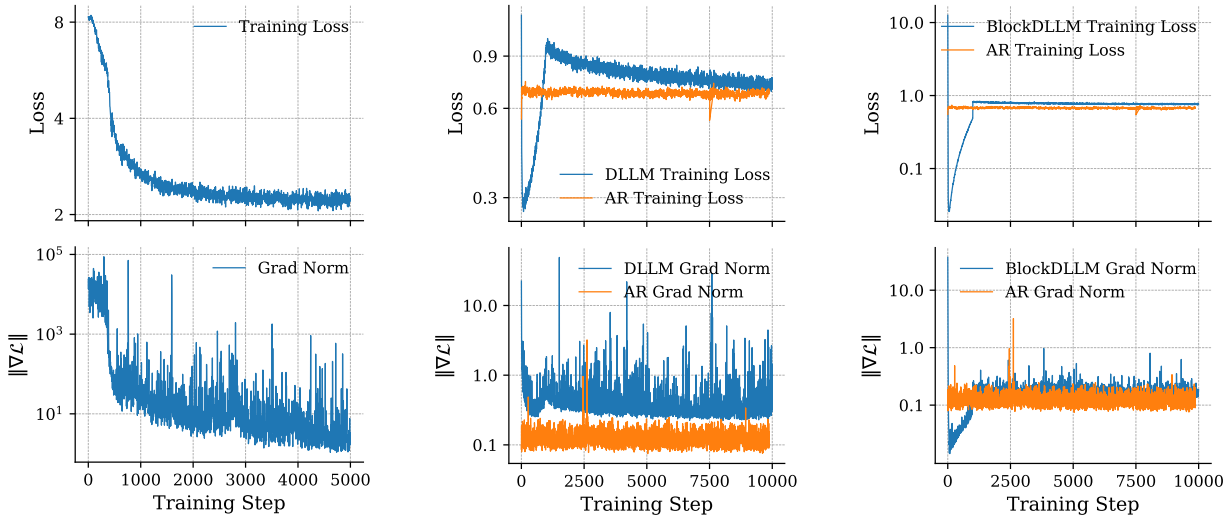
Previous work [16] has reported that CPT of mask diffusion language models (DLLMs) is highly sensitive to the learning rate. In practice, the stability of the loss and gradient norm is also affected by architecture (e.g., attention pattern, logit parametrization), numerical precision, and infrastructure details. This motivates a more robust warmup procedure for DLLM CPT.

To reduce the architectural and objective gap between AR and DLLM, we reuse the AR token head and logit-shift parametrization, and only change the attention pattern from a causal lower-triangular mask to a bidirectional, full attention mask. Even under this minimal change, naive AR→DLLM CPT shows a high initial loss and large gradient-norm spikes.

We attribute the instability mainly to three factors: (i) the change of attention mask, which induces a structural distribution shift in internal representations; (ii) the higher task difficulty when the corruption process masks a large fraction of tokens, compared to AR next-token prediction; (iii) the ELBO-motivated loss weight $w(t)$ in the DLLM objective (Eq. 4), where $w(t)$ can be large at low masking ratios (e.g., under a linear noise schedule, masking 10% of tokens yields $w(t) \approx 10$). This effectively acts as loss scaling that amplifies gradient norms, making training less stable.

Rather than annealing the attention mask [16] (which is inconvenient for highly optimized kernels such as FlashAttention that assume a fixed mask), we warm up only the corruption process. In standard DLLM, the corruption level $t$ is sampled uniformly from $[0, 1]$, and the masking ratio ranges from 0 to 1. During warmup, we cap the maximum corruption level: $t \sim \mathcal{U}(0, u_{\max})$, and linearly increase $q_{\max}$ from a small value $q_{\text{init}}$ (e.g., $10^{-3}$) to 1 over $S_{\text{warmup}}$ steps:

$$u_{\max}(s) = u_{\text{init}} + (1 - u_{\text{init}}) \cdot \frac{s}{S_{\text{warmup}}}, \quad s = 0, \ldots, S_{\text{warmup}}.$$




BiDLLM Without Warmup | BiDLLM With Warmup | BlockDLLM With Warmup (no-shift)

**Figure 4 Comparison of training stability before and after applying warmup.** The left figure shows the behavior without warmup, and the right figure shows the behavior with warmup. When warmup is used, both the training loss and gradient norm become significantly more stable, and they quickly decrease to a level comparable to that of the AR continual pretraining stage. BiDLLM refers to a purely bidirectional masked diffusion model, BlockDLLM to a block-masked diffusion model, and no-shift indicates that token logit shifting is disabled, which is the configuration adopted in our final model.

This implements a curriculum from easy (low mask ratio, almost AR-like) reconstruction to the full DLLM regime. To further suppress gradient spikes, we drop $w(t)$ in the warmup phase and optimize

$$\mathcal{L}_{\text{DLLM}}^{\text{warmup}}(\theta) = -\mathbb{E}_{\mathbf{x}^0\sim p_{\text{data}},\, t\sim\mathcal{U}(0,u_{\max}),\, \mathbf{x}^t\sim q(\mathbf{x}^t|\mathbf{x}^0)}\Big[\sum_{i=1}^{N} 1[\mathbf{x}_i^t = \text{MASK}]\, \log p_\theta(\mathbf{x}_i^0 \mid \mathbf{x}_{1:N}^t)\Big]. \tag{10}$$

After warmup, we revert to the original loss in Eq. (4) with $t \sim \mathcal{U}(0,1)$.

This warmup produces a much smoother CPT trajectory: the gradient norm spike at the AR→DLLM boundary is strongly reduced, and the loss follows a characteristic "$\sqrt{\cdot}$-shaped" curve (first decreasing on easy tasks, then increasing as the mask ratio grows, and finally decreasing again). The peak loss during warmup is significantly lower than the naive AR→DLLM baseline, and the final loss is comparable to AR→AR CPT. We observe similarly stable behavior when we remove the logit shift and train a block-diffusion DLLM with the same warmup (see Fig. 4).

### 3.3 Block-wise Clipped Noise Scheduling for Masked Block Diffusion

*Motivation.* In masked block diffusion training, at each step we corrupt only a single contiguous block of tokens rather than the full sequence. If one reuses a global continuous-time schedule $t\in[0,1] \mapsto u(t)$ (with $q$ the corruption/mask rate) that was designed for whole-sequence diffusion, a significant portion of training steps produce weak or even zero learning signal when the block length $B$ is small. Concretely, when the corruption is applied only inside a block $\mathcal{B}$ of size $B$, the expected number of corrupted tokens at time $t$ is $\mathbb{E}[m \mid t] = B\,u(t)$ and the probability of observing no corrupted token is $\Pr[m=0 \mid t] = (1-u(t))^B$.. Under a standard global linear schedule $u(t) = 1-t$ with $t \sim \text{Unif}[0,1]$, the fraction of steps with $m=0$ equals

$$\mathbb{E}_t[(1-t)^B] \;=\; \int_0^1 (1-t)^B\, dt \;=\; \frac{1}{B+1}, \tag{11}$$

which is non-negligible for typical block sizes (e.g., 1/3 for $B$=2, 1/5 for $B$=4, 1/9 for $B$=8).

Instead of redesigning the global schedule, we adopt a simple block-aware sampling rule. At each training step, after sampling $t$, we define a block-specific mask rate by clipping $u_{\text{blk}}(t) = \min\big(1, \max(u(t), 1/B)\big)$, so that $q_{\text{blk}}(t) \in [1/B, 1]$ for all $t$. This guarantees that the expected number of masked tokens in the block satisfies $\mathbb{E}[m \mid t] = B\, u_{\text{blk}}(t) \geq 1$. In addition, it prevents the loss weight $w(t) = \frac{1}{u_{\text{blk}}(t)}$ for those tokens from becoming excessively large, thereby promoting more stable training.

Given $u_{\text{blk}}(t)$, we independently mask each token in the chosen block $\mathcal{B}$ with probability $u_{\text{blk}}(t)$, while tokens outside $\mathcal{B}$ remain clean and provide context. To further ensure that every block contributes to the loss, we apply the following fallback rule: if, after sampling, no token in $\mathcal{B}$ is masked (i.e., $m = 0$), we uniformly sample one position in $\mathcal{B}$ and force it to be masked. In this way, every training step contains at least one supervised token inside the block, while still preserving the overall shape of the original schedule $u(t)$.

# 4 Experiments

We conduct extensive experiments to compare the base model and its instruction-tuned (instruction) variant across a diverse set of code-centric benchmarks.

## 4.1 Experiments Setting

To preserve representation plasticity and enable structural adaptation to the target data distribution, we perform continuous pretraining from a Seed-Coder pre-annealing checkpoint, using a context length of 8192 with packed sequences. We reuse the Seed-Coder training pipeline and compress its multi-stage continuous pretraining data to a total of 1.3T tokens via subsampling. The supervised fine-tuning (SFT) stage fully reuses the original Seed-Coder SFT dataset. We adopt block diffusion with a block size of 4 for this continuous pretraining stage. Since the no-logit-shift formulation is more consistent with the absorbing diffusion paradigm, where each masked position predicts itself and the input and prediction targets are aligned at both the token and sentence levels, we follow a no-logit-shift design similar to that used in LLADA and SDAR.

To prevent repeated compilation of flex attention, we make the attention between packed samples mutually visible, so that the same attention mask and operator can be reused in every forward pass. To maintain training efficiency, we adopt the same packing strategy during SFT. However, after each sample we randomly append 1–4 '<eos>' tokens, enabling the model to preserve the ability to generate variable-length outputs within each packed block.

In our experiments, we compare our method against a broad set of strong AR models and DLLM that have demonstrated competitive performance on code generation tasks.

The AR baselines include StarCoder2 [37], DeepSeek-Coder [20], CodeQwen1.5 [45], OpenCoder [21], Qwen2.5-Coder [22], Seed-Coder [49], CodeLlama [46], and Llama3.1 [35].

The DLLM baselines include LLaDA [43], Dream [62], DiffuCoder [17], Dream-Coder [59], LLaDA-MoE [66], Fast-dLLMv2 [56], SDAR [10], Seed-Diffusion-Preview, LLaDA2.0 [5], SDLM [34], WeDLM [30], Mercury Coder [28], and Gemini Diffusion [18].

## 4.2 Benchmarks

We evaluate the base and instruction models on a diverse suite of coding benchmarks spanning function-level code generation, execution-centric code reasoning, multilingual generalization, and instruction-following code editing:

- **HumanEval / HumanEval+**: HumanEval [9] contains 164 Python function-completion tasks evaluated by unit tests. HumanEval+ from EvalPlus [32] substantially expands the test suites (about 80×) for stricter functional correctness. We use EvalPlus to report results on both HumanEval and HumanEval+.
- **MBPP / MBPP+**: MBPP [3] includes 974 crowd-sourced Python programming problems with tests. We adopt the human-verified EvalPlus subset (399 well-formed tasks) and additionally evaluate on MBPP+, which augments the tests (about 35×) for more reliable pass@1 estimation [32].

| Model | Type | Size | HumanEval | | MBPP | |
|---|---|---|---|---|---|---|
| | | | *HE* | *HE*$^+$ | *MBPP* | *MBPP*$^+$ |
| ~8B Models | | | | | | |
| StarCoder2-7B | AR | 7B | 35.4 | 29.9 | 54.4 | 45.6 |
| DeepSeek-Coder-6.7B-Base | AR | 6.7B | 47.6 | 39.6 | 70.2 | 56.6 |
| CodeQwen1.5-7B | AR | 7B | 51.8 | 45.7 | 72.2 | 60.2 |
| OpenCoder-8B-Base | AR | 8B | 66.5 | 63.4 | 79.9 | **70.4** |
| Qwen2.5-Coder-7B | AR | 7B | 72.0 | 67.1 | 79.4 | 68.3 |
| Seed-Coder-8B-Base | AR | 8B | 77.4 | 68.3 | 82.0 | 69.0 |
| LLaDA-8B-Base | DLLM | 8B | 35.4 | 30.5 | 50.1 | 42.1 |
| Dream-7B-Base | DLLM | 7B | 56.7 | 50.0 | 68.7 | 57.4 |
| DiffuCoder-7B-Base | DLLM | 7B | 67.1 | 60.4 | 74.2 | 60.9 |
| Dream-Coder-7B-Base | DLLM | 7B | 66.5 | 60.4 | 75.9 | 61.6 |
| LLaDA-MoE-7B-A1B-Base | DLLM | 1B/7B | 45.7 | - | 52.4 | - |
| WeDLM-8B-Base | DLLM | 8B | 75.0 | 68.9 | 67.0 | - |
| Stable-DiffCoder-8B-Base | DLLM | 8B | **79.3** | **73.8** | **83.6** | 67.7 |
| 13B+ Models | | | | | | |
| StarCoder2-15B | AR | 15B | 46.3 | 37.8 | 66.2 | 53.1 |
| CodeLlama-70B-Base | AR | 70B | 52.4 | 50.6 | 71.0 | 65.6 |
| Llama-3.1-70B-Base | AR | 70B | 54.9 | 51.2 | 81.4 | 67.2 |
| DeepSeek-Coder-33B-Base | AR | 33B | 54.9 | 47.6 | 74.2 | 60.7 |
| DeepSeek-Coder-V2-Lite-Base | AR | 2.4B/16B | 40.9 | 34.1 | 71.9 | 59.4 |
| Qwen2.5-Coder-14B | AR | 14B | **83.5** | **75.6** | **83.6** | **69.8** |

**Table 1** Performance of various base models on HumanEval($^+$) and MBPP($^+$).

- **CRUXEval**: Small token-level changes can drastically alter program behavior, making code reasoning sensitive and precise. CRUXEval [19] provides 800 Python functions with I/O examples and evaluates two tasks: CRUXEval-I (predict inputs given outputs) and CRUXEval-O (predict outputs given inputs).
- **MultiPL-E**: Beyond Python, we evaluated base models on code generation across multiple programming languages using MultiPL-E [7], which extends HumanEval to 18 languages. Following Qwen2.5-Coder, we report results on eight representative mainstream languages to assess cross-language coding performance and language-specific effects.
- **MHPP**: To better separate strong models beyond saturated benchmarks (e.g., HumanEval/MBPP), we include MHPP [11], which targets harder Python problems with more complex specifications and reasoning requirements.
- **BigCodeBench**: BigCodeBench [68] focuses on challenging, realistic programming by requiring tool-like function calls from 139 libraries across 7 domains, covering 1,140 Python tasks with rich context. Each task has multiple tests (5.6 on average) with high branch coverage, and we report results on both the full set and the Hard split of BigCodeBench-Completion.
- **LiveCodeBench**: To mitigate contamination and overfitting to static benchmarks, LiveCodeBench [24] continuously collects time-stamped problems from competitive programming platforms (e.g., LeetCode, AtCoder, Codeforces), enabling evaluation within recent, user-specified time windows. We follow the v5 evaluation benchmark used by Seed-Coder for comparison.
- **MBXP**: MBXP [2] translates MBPP problems and unit tests into 10+ programming languages, enabling execution-based multilingual evaluation. We report results across 13 widely used languages.
- **NaturalCodeBench**: NaturalCodeBench [63] contains 402 high-quality problems curated from real user queries (Python and Java) across six practical domains (e.g., software engineering, data science, system administration), with more diverse and complex inputs than classic algorithmic benchmarks.

| Model | Size | Python | C++ | Java | PHP | TS | C# | Bash | JS | Average |
|---|---|---|---|---|---|---|---|---|---|---|
| ~8B Models | | | | | | | | | | |
| StarCoder2-7B | 7B | 35.4 | 40.4 | 38.0 | 30.4 | 34.0 | 46.2 | 13.9 | 36.0 | 34.3 |
| DeepSeek-Coder-6.7B-Base | 6.7B | 49.4 | 50.3 | 43.0 | 38.5 | 49.7 | 50.0 | 28.5 | 48.4 | 44.7 |
| CodeQwen1.5-7B | 7B | 51.8 | 52.2 | 42.4 | 46.6 | 52.2 | 55.7 | 36.7 | 49.7 | 48.4 |
| OpenCoder-8B-Base | 8B | 66.5 | 63.4 | 63.9 | 61.5 | 68.6 | 54.3 | 44.3 | 65.8 | 61.0 |
| Qwen2.5-Coder-7B | 7B | 72.0 | 62.1 | 53.2 | 59.0 | 64.2 | **60.8** | 38.6 | 60.3 | 58.8 |
| Seed-Coder-8B-Base | 8B | 77.4 | **69.6** | 72.8 | 63.9 | **77.4** | 53.8 | 48.1 | **77.6** | 67.6 |
| Stable-DiffCoder-8B-Base | 8B | **80.5** | 69.4 | **74.1** | **74.4** | 74.8 | **70.3** | **53.2** | 73.1 | **71.2** |
| 13B+ Models | | | | | | | | | | |
| StarCoder2-15B | 15B | 46.3 | 47.2 | 46.2 | 39.1 | 42.1 | 53.2 | 15.8 | 43.5 | 41.7 |
| CodeLlama-70B-Base | 70B | 52.4 | 49.7 | 44.7 | 46.6 | 57.2 | 46.7 | 31.6 | 56.5 | 48.2 |
| Llama-3.1-70B-Base | 70B | 54.9 | 41.0 | 41.1 | 48.4 | 57.9 | 44.2 | 29.1 | 55.3 | 46.5 |
| DeepSeek-Coder-33B-Base | 33B | 56.1 | 58.4 | **51.9** | 44.1 | 52.8 | 51.3 | 32.3 | 55.3 | 50.3 |
| DeepSeek-Coder-V2-Lite-Base | 2.4B/16B | 40.9 | 45.9 | 34.8 | 47.2 | 48.4 | 41.7 | 19.6 | 44.7 | 40.4 |
| Qwen2.5-Coder-14B | 14B | **83.5** | **69.6** | 46.8 | **64.6** | **69.2** | **63.3** | **39.9** | **61.5** | **62.3** |

**Table 2** Performance of base models on MultiPL-E.

| Model | Size | CRUXEval | |
|---|---|---|---|
| | | *Input-CoT* | *Output-CoT* |
| ~8B Models | | | |
| StarCoder2-7B | 7B | 39.5 | 35.1 |
| DeepSeek-Coder-6.7B-Base | 6.7B | 39.0 | 41.0 |
| OpenCoder-8B-Base | 8B | 43.3 | 43.9 |
| Qwen2.5-Coder-7B | 7B | **56.5** | 56.0 |
| Seed-Coder-8B-Base | 8B | 52.0 | 54.8 |
| Stable-DiffCoder-8B-Base | 8B | 53.8 | **60.0** |
| 13B+ Models | | | |
| StarCoder2-15B | 15B | 46.1 | 47.6 |
| CodeLlama-34B-Base | 34B | 49.4 | 43.9 |
| DeepSeek-Coder-33B-Base | 33B | 50.6 | 48.8 |
| DeepSeek-Coder-V2-Lite-Base | 2.4B/16B | 53.4 | 46.1 |
| Qwen2.5-Coder-14B | 14B | **60.6** | **66.4** |

**Table 3** Performance of base models CRUXEval.

- **Aider**: We use Aider's [1] code editing benchmark to assess instruction-following edits on existing codebases, based on 133 Exercism exercises, requiring models to produce changes that can be applied automatically and pass tests.
- **CanItEdit**: CanItEdit [8] evaluates instructional code editing with 105 hand-crafted problems covering both detailed and underspecified ("lazy") instructions, testing robustness across diverse edit scenarios.

## 4.3 Evaluation of Base Models

### 4.3.1 Code Generation

**Humaneval and MBPP** As shown in Table 1, we evaluate Stable-DiffCoder-8B-Base using the EvalPlus chat-style prompting template. Among diffusion language models (DLLMs) of comparable scale (approximately 8B parameters), our base model achieves the best overall performance on both HumanEval(+) and MBPP(+). When compared with similarly sized autoregressive (AR) baselines, Stable-DiffCoder consistently outperforms

[1] https://aider.chat/docs/leaderboards/edit.html

| Model | Type | Size | HumanEval | | MBPP | |
|---|---|---|---|---|---|---|
| | | | *HE* | *HE*$^+$ | *MBPP* | *MBPP*$^+$ |
| ~8B Models | | | | | | |
| CodeLlama-7B-Instruct | AR | 7B | 40.9 | 33.5 | 54.0 | 44.4 |
| DeepSeek-Coder-6.7B-Instruct | AR | 6.7B | 74.4 | 71.3 | 74.9 | 65.6 |
| CodeQwen1.5-7B-Chat | AR | 7B | 83.5 | 78.7 | 77.7 | 67.2 |
| Yi-Coder-9B-Chat | AR | 9B | 82.3 | 74.4 | 82.0 | 69.0 |
| Llama-3.1-8B-Instruct | AR | 8B | 68.3 | 59.8 | 70.1 | 59.0 |
| OpenCoder-8B-Instruct | AR | 8B | 83.5 | 78.7 | 79.1 | 69.0 |
| Qwen2.5-Coder-7B-Instruct | AR | 7B | **88.4** | **84.1** | 83.5 | 71.7 |
| Qwen3-8B | AR | 8B | 84.8 | 80.5 | 77.0 | 67.2 |
| Seed-Coder-8B-Instruct | AR | 8B | 84.8 | 78.7 | 85.2 | 71.2 |
| LLaDA-8B-Instruct | DLLM | 8B | 49.4 | - | 41.0 | - |
| Dream-7B-Instruct | DLLM | 7B | 63.4 | - | 68.3 | - |
| LLaDA-MoE-7B-Instruct | DLLM | 1B/7B | 61.6 | - | 70.0 | - |
| Fast-dLLMv2 | DLLM | 7B | 43.9 | 40.2 | 50.0 | 41.3 |
| DiffuCoder-7B-Instruct | DLLM | 7B | 72.0 | 65.2 | 75.1 | 61.9 |
| Dream-Coder-7B-Instruct | DLLM | 7B | 82.9 | - | 79.6 | - |
| SDAR-8B-Chat | DLLM | 8B | 78.7 | - | 72.0 | - |
| WeDLM-8B-Chat | DLLM | 8B | 80.5 | 73.8 | 70.5 | - |
| Stable-DiffCoder-8B-Instruct | DLLM | 8B | 86.6 | 82.3 | **85.7** | **72.8** |
| 13B+ Models and External API | | | | | | |
| StarCoder2-15B-Instruct | AR | 15B | 67.7 | 60.4 | 78.0 | 65.1 |
| Codestral-22B | AR | 22B | 81.1 | 73.2 | 78.2 | 62.2 |
| CodeLlama-70B-Instruct | AR | 70B | 72.0 | 65.9 | 77.8 | 64.6 |
| DeepSeek-Coder-33B-Instruct | AR | 33B | 81.1 | 75.0 | 80.4 | 70.1 |
| DeepSeek-Coder-V2-Lite-Istruct | AR | 2.4B/16B | 81.1 | 75.6 | 82.8 | 70.4 |
| DeepSeek-Coder-V2-Instruct | AR | 21B/236B | 85.4 | 82.3 | 89.4 | 75.1 |
| Qwen2.5-Coder-14B-Instruct | AR | 14B | 89.6 | 87.2 | 86.2 | 72.8 |
| Qwen2.5-Coder-32B-Instruct | AR | 32B | 92.7 | **87.2** | **90.2** | 75.1 |
| SDAR-30B-Chat | DLLM | 3B/30B | 87.2 | - | 71.6 | - |
| LLaDA2.0-mini | DLLM | 1B/16B | 86.6 | 79.9 | 81.5 | 74.1 |
| LLaDA2.0-flash | DLLM | 6B/100B | **94.5** | **87.8** | 88.3 | **79.6** |
| SDLM-32B | DLLM | 32B | 81.1 | 73.8 | 80.9 | 57.1 |
| Seed-Diffusion-Preview(0705) | DLLM | - | 82.8 | - | 79.4 | - |
| Mercury Coder | DLLM | - | 90.0 | - | 77.1 | - |
| Gemini Diffusion | DLLM | - | 89.6 | - | 76.0 | - |

**Table 4** Performance of various base models on HumanEval(+) and MBPP(+).

them on HumanEval and HumanEval+, as well as on MBPP, while being only slightly inferior on MBPP+. Notably, Stable-DiffCoder exhibits a substantial improvement over our AR baseline, Seed-Coder-8B-Base, across all evaluated HumanEval and MBPP settings.

**MBXP** Beyond Python-centric benchmarks, we further evaluate multilingual code generation on MultiPL-E (Table 2). Stable-DiffCoder yields particularly large gains in languages such as C# and PHP. Since these languages are sparsely represented in our training corpus, we hypothesize that diffusion-style stochastic sampling can effectively amplify learning signals from low-resource code by exposing the model to multiple corrupted-and-denoised views of the same underlying example, thereby improving generalization in data-scarce languages.

### 4.3.2 Code Reasoning

**CRUXEval** We additionally assess code reasoning on CRUXEval (Table 3), which consists of Python-only problems. We observe that Stable-DiffCoder outperforms Seed-Coder-Base on both reasoning inputs (Input-CoT) and reasoning outputs (Output-CoT). This suggests that incorporating a moderate degree of random

masking objectives can effectively enhance the model's reasoning capability. Moreover, the inputs and outputs in CRUXEval are inherently structured rather than strictly following left-to-right causal logic. As a result, DLLMs benefit from any-order modeling, which enables them to more comprehensively capture the relationships among these structured components.

## 4.4 Evaluation of Instruction Models

| Model | Size | MHPP *pass@1* | BigCodeBench *Full* | BigCodeBench *Hard* | LiveCodeBench *pass@1* |
|---|---|---|---|---|---|
| ~8B Models | | | | | |
| CodeLlama-7B-Instruct | 7B | 6.7 | 25.7 | 4.1 | 3.6 |
| DeepSeek-Coder-6.7B-Instruct | 6.7B | 20.0 | 43.8 | 15.5 | 9.6 |
| CodeQwen1.5-7B-Chat | 7B | 17.6 | 43.6 | 15.5 | 3.0 |
| Yi-Coder-9B-Chat | 9B | 26.7 | 49.0 | 17.6 | 17.5 |
| Llama-3.1-8B-Instruct | 8B | 17.1 | 40.5 | 13.5 | 11.5 |
| OpenCoder-8B-Instruct | 8B | 30.5 | 50.9 | 18.9 | 17.1 |
| Qwen2.5-Coder-7B-Instruct | 7B | 26.7 | 48.8 | 20.3 | 17.3 |
| Qwen3-8B | 8B | 32.8 | 51.7 | 23.0 | 23.5 |
| Seed-Coder-8B-Instruct | 8B | 36.2 | 53.3 | 26.4 | **24.7** |
| Stable-DiffCoder-8B-Instruct | 8B | **42.4** | **54.8** | **31.8** | 23.5 |
| 13B+ Models and External API | | | | | |
| StarCoder2-15B-Instruct | 15B | 19.0 | 45.1 | 14.9 | 5.3 |
| Codestral-22B | 22B | 25.2 | 52.5 | 24.3 | 20.5 |
| CodeLlama-70B-Instruct | 70B | 19.5 | 49.6 | 15.5 | 14.5 |
| DeepSeek-Coder-33B-Instruct | 33B | 32.9 | 51.1 | 20.9 | 14.5 |
| DeepSeek-Coder-V2-Lite-Instruct | 2.4B/16B | 30.5 | 47.6 | 18.2 | 14.2 |
| DeepSeek-Coder-V2-Instruct | 21B/236B | 31.9 | 59.7 | 33.1 | 28.9 |
| Qwen2.5-Coder-14B-Instruct | 14B | 36.7 | 52.2 | 16.2 | 19.3 |
| Qwen2.5-Coder-32B-Instruct | 32B | **42.4** | **52.3** | **20.9** | **30.7** |
| LLaDA2.0-flash | 6B/100B | - | 41.6 | - | - |
| Seed-Diffusion-Preview(0705) | - | - | 53.2 | - | - |
| Mercury Coder | - | - | 45.5 | - | - |
| Gemini Diffusion | - | - | 45.4 | - | - |

**Table 5** Performance of instruct models on MHPP, BigCodeBench-Completion and LiveCodeBench (v5).

| Model | Size | Python | Java | C++ | C# | TS | JS | PHP | Go | Kotlin | Perl | Ruby | Scala | Swift | Average |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ~8B Models | | | | | | | | | | | | | | | |
| CodeLlama-7B-Instruct | 7B | 54.0 | 38.8 | 32.9 | 50.0 | 42.3 | 45.5 | 36.6 | 48.8 | 47.2 | 50.1 | 36.9 | 40.2 | 33.2 | 42.8 |
| DeepSeek-Coder-6.7B-Instruct | 6.7B | 74.9 | 52.2 | 30.9 | 55.9 | 64.8 | 64.7 | 25.8 | 93.8 | 59.6 | 3.3 | 65.9 | 54.8 | 47.4 | 53.4 |
| CodeQwen1.5-7B-Chat | 7B | 77.7 | 66.6 | 66.8 | 64.4 | 66.7 | 67.5 | 67.3 | 55.1 | 60.9 | 61.1 | 65.9 | 60.0 | 54.7 | 64.2 |
| Yi-Coder-9B-Chat | 9B | 82.0 | 73.4 | 79.1 | 70.3 | 74.1 | 73.3 | 76.4 | 90.9 | 64.4 | 60.9 | 67.3 | 63.5 | 57.3 | 71.8 |
| Llama-3.1-8B-Instruct | 8B | 70.1 | 59.8 | 59.1 | 56.6 | 59.1 | 59.1 | 62.5 | 85.7 | 52.2 | 42.6 | 55.9 | 44.5 | 31.8 | 56.8 |
| OpenCoder-8B-Instruct | 8B | 79.1 | 68.1 | 71.3 | 71.0 | 67.6 | 61.4 | 68.1 | 94.4 | 66.4 | 56.1 | 70.5 | 63.1 | 56.7 | 68.8 |
| Qwen2.5-Coder-7B-Instruct | 7B | 83.5 | 70.5 | 74.1 | 71.5 | 72.2 | 74.1 | 74.2 | 96.0 | 65.5 | 64.4 | 75.5 | 64.2 | **62.0** | 72.9 |
| Qwen3-8B | 8B | 77.0 | 69.0 | 72.8 | 68.9 | 73.0 | 73.8 | 72.3 | 92.9 | 62.0 | 64.6 | 69.0 | 63.1 | 42.2 | 69.3 |
| Seed-Coder-8B-Instruct | 8B | 85.2 | 72.7 | 77.0 | **74.2** | 72.8 | **78.8** | **74.7** | 95.5 | **73.4** | **72.5** | **78.0** | 70.3 | 54.2 | **75.3** |
| Stable-DiffCoder-8B-instruct | 8B | **85.7** | **75.3** | **77.8** | 71.2 | **73.0** | 76.9 | 73.8 | **98.7** | 72.5 | 70.5 | 77.1 | **71.4** | 54.2 | **75.3** |
| 13B+ Models and External API | | | | | | | | | | | | | | | |
| StarCoder2-15B-Instruct | 15B | 78.0 | 25.1 | 25.9 | 21.7 | 20.7 | 59.8 | 53.5 | 90.4 | 46.7 | 31.9 | 56.1 | 43.2 | 42.0 | 45.8 |
| Codestral-22B | 22B | 78.2 | 73.6 | 77.3 | 70.1 | 71.7 | 68.5 | 74.9 | 97.1 | 71.0 | 66.6 | 74.2 | 64.4 | 50.1 | 72.1 |
| CodeLlama-70B-Instruct | 70B | 77.8 | 66.6 | 68.6 | 69.2 | 47.8 | 62.5 | 70.5 | 77.7 | 57.2 | 51.1 | 67.0 | 51.3 | 48.7 | 62.8 |
| DeepSeek-Coder-33B-Instruct | 33B | 80.4 | 71.8 | 76.8 | 69.9 | 72.4 | 69.8 | 75.1 | 96.4 | 70.1 | 66.6 | 75.1 | 64.6 | 54.3 | 72.6 |
| DeepSeek-Coder-V2-Lite-Instruct | 2.4B/16B | 82.8 | 73.3 | 75.3 | 72.4 | 72.4 | 73.1 | 75.1 | 95.1 | 69.9 | 61.6 | 74.5 | 63.5 | 55.0 | 72.6 |
| DeepSeek-Coder-V2-Instruct | 21B/236B | 89.4 | 78.2 | 77.6 | 72.6 | 74.8 | **80.5** | 75.8 | 89.1 | 74.5 | 70.7 | 80.2 | 67.9 | 59.0 | 76.2 |
| Qwen2.5-Coder-14B-Instruct | 14B | 86.2 | 77.5 | 84.8 | **80.1** | 77.6 | 77.7 | 79.7 | **97.1** | 75.3 | **76.2** | 79.3 | **73.1** | **67.2** | 79.4 |
| Qwen2.5-Coder-32B-Instruct | 32B | **90.2** | **80.4** | **86.3** | 73.5 | **78.3** | 79.3 | **87.6** | 96.4 | **75.6** | 74.7 | **83.4** | 63.3 | 66.7 | **79.7** |
| Seed-Diffusion-Preview(0705) | - | 79.4 | 67.7 | 72.6 | 70.3 | 73.0 | 76.6 | 74.7 | 92.9 | 71.2 | 71.2 | 72.5 | 67.0 | 54.2 | 72.6 |

**Table 6** Performance of instruct models on MBXP.

| Model | Size | NCB (zh) | | | NCB (en) | | | Total |
|---|---|---|---|---|---|---|---|---|
| | | *Python* | *Java* | *Total* | *Python* | *Java* | *Total* | |
| ~8B Models | | | | | | | | |
| CodeLlama-7B-Instruct | 7B | 18.6 | 8.6 | 13.6 | 17.1 | 14.3 | 15.7 | 14.6 |
| DeepSeek-Coder-6.7B-Instruct | 6.7B | 38.6 | 31.4 | 35.0 | 32.9 | 32.9 | 32.9 | 33.9 |
| CodeQwen1.5-7B-Chat | 7B | 30.0 | 28.6 | 29.3 | 30.0 | 27.1 | 28.6 | 25.7 |
| Yi-Coder-9B-Chat | 9B | 41.4 | **45.7** | 43.6 | 38.6 | 44.3 | 41.5 | 42.5 |
| Llama-3.1-8B-Instruct | 8B | 27.1 | 24.3 | 25.7 | 22.9 | 22.9 | 22.9 | 24.3 |
| OpenCoder-8B-Instruct | 8B | 40.0 | 30.0 | 35.0 | 35.7 | 24.3 | 30.0 | 32.5 |
| Qwen2.5-Coder-7B-Instruct | 7B | 34.3 | 37.1 | 35.7 | 34.3 | 35.7 | 35.0 | 35.4 |
| Qwen3-8B | 8B | 37.1 | 32.9 | 35.0 | 34.3 | 38.6 | 36.5 | 35.7 |
| Seed-Coder-8B-Instruct | 8B | **55.7** | **45.7** | **50.7** | **50.0** | **47.1** | **48.6** | **49.6** |
| Stable-DiffCoder-8B-Instruct | 8B | 51.4 | **45.7** | 48.6 | **50.0** | **47.1** | **48.6** | 48.6 |
| 13B+ Models and External API | | | | | | | | |
| StarCoder2-15B-Instruct | 15B | 44.3 | 30.0 | 37.2 | 38.6 | 42.9 | 40.8 | 39.0 |
| Codestral-22B | 22B | 40.0 | 44.3 | 42.2 | 41.4 | **45.7** | 43.6 | 42.9 |
| CodeLlama-70B-Instruct | 70B | 35.1 | 32.1 | 33.6 | 32.8 | 30.5 | 31.7 | 32.6 |
| DeepSeek-Coder-33B-Instruct | 33B | 44.3 | 38.9 | 41.6 | **44.3** | 44.3 | **44.3** | 43.0 |
| DeepSeek-Coder-V2-Lite-Instruct | 2.4B/16B | 41.4 | 47.1 | 44.3 | 41.4 | 37.1 | 39.3 | 41.8 |
| Qwen2.5-Coder-14B-Instruct | 14B | **48.6** | **48.6** | **48.6** | 42.9 | **45.7** | **44.3** | **46.4** |
| Seed-Diffusion-Preview(0705) | - | 52.9 | 38.6 | 45.8 | 45.7 | 38.6 | 42.2 | 43.9 |

**Table 7** Performance of instruct models on NaturalCodeBench.

| Model | Size | Input-CoT | Output-CoT |
|---|---|---|---|
| ~8B Models | | | |
| CodeLlama-7B-Instruct | 7B | 36.1 | 36.2 |
| DeepSeek-Coder-6.7B-Instruct | 6.7B | 42.6 | 45.1 |
| CodeQwen1.5-7B-Chat | 7B | 44.0 | 38.8 |
| Yi-Coder-9B-Chat | 9B | 47.5 | 55.6 |
| Llama-3.1-8B-Instruct | 8B | 35.6 | 37.8 |
| OpenCoder-8B-Instruct | 8B | 39.9 | 43.0 |
| Qwen2.5-Coder-7B-Instruct | 7B | 65.8 | 65.9 |
| Qwen3-8B | 8B | **73.8** | **76.9** |
| Seed-Coder-8B-Instruct | 8B | 63.3 | 67.1 |
| Stable-DiffCoder-8B-Instruct | 8B | 62.1 | 69.0 |
| 13B+ Models | | | |
| StarCoder2-15B-Instruct | 15B | 45.5 | 60.9 |
| Codestral-22B | 22B | 61.3 | 63.5 |
| CodeLlama-70B-Instruct | 70B | 56.5 | 57.8 |
| DeepSeek-Coder-33B-Instruct | 33B | 47.3 | 50.6 |
| DeepSeek-Coder-V2-Lite-Instruct | 2.4B/16B | 53.0 | 52.9 |
| Qwen2.5-Coder-14B-Instruct | 14B | **69.5** | **79.5** |

**Table 8** Performance of instruct models on CRUXEval.

### 4.4.1 Code Generation

**Humaneval and MBPP.** On standard function-level code generation benchmarks, Stable-DiffCoder-8B-Instruct achieves strong gains over the autoregressive baseline. As shown in Table 4, Stable-DiffCoder-8B-Instruct significantly improves upon Seed-Coder-8B-Instruct on both HumanEval(+) and MBPP(+), and on MBPP it outperforms all other instruction models while clearly surpassing all ∼8B-scale diffusion models.

**MHPP.** On the more challenging MHPP benchmark, which only reveals scores through an official submission interface to mitigate data contamination and ensure fair comparison, Stable-DiffCoder-8B-Instruct attains the best performance among all compared models and reaches the level of Qwen2.5-Coder-32B-Instruct (Table 5).

| Model | Size | Aider *tries=2* | CanItEdit *pass@1* |
|---|---|---|---|
| ~8B Models | | | |
| CodeLlama-7B-Instruct | 7B | 1.5 | 25.7 |
| DeepSeek-Coder-6.7B-Instruct | 6.7B | 44.4 | 36.9 |
| CodeQwen1.5-7B-Chat | 7B | 38.3 | 34.8 |
| Yi-Coder-9B-Chat | 9B | 54.1 | 50.5 |
| Llama-3.1-8B-Instruct | 8B | 33.1 | 39.5 |
| OpenCoder-8B-Instruct | 8B | 30.8 | 39.0 |
| Qwen2.5-Coder-7B-Instruct | 7B | **57.9** | 49.5 |
| Qwen3-8B | 8B | 55.6 | 45.7 |
| Seed-Coder-8B-Instruct | 8B | 57.1 | 50.5 |
| Stable-DiffCoder-8B-Instruct | 8B | 54.9 | **60.0** |
| 13B+ Models and External API | | | |
| StarCoder2-15B-Instruct | 15B | 38.2 | 31.4 |
| Codestral-22B | 22B | 51.1 | 52.4 |
| CodeLlama-70B-Instruct | 70B | 15.0 | 40.5 |
| DeepSeek-Coder-33B-Instruct | 33B | 54.5 | 46.2 |
| DeepSeek-Coder-V2-Lite-Instruct | 2.4B/16B | 52.6 | 45.2 |
| Qwen2.5-Coder-14B-Instruct | 14B | **69.2** | 52.9 |
| Seed-Diffusion-Preview(0705) | - | 44.4 | 54.3 |

**Table 9** Performance of instruct models on Aider ("whole" format) and CanItEdit.

**BigCodeBench.** For more realistic programming tasks, BigCodeBench measures the ability to solve challenging, real-world coding problems with rich context and tool-like function calls. As shown in Table 5, Stable-DiffCoder-8B-Instruct delivers substantial improvements over Seed-Coder-8B-Instruct and, among all models, is only surpassed by the much larger DeepSeek-Coder-V2-Instruct (21B/236B), demonstrating the strong practical code generation capabilities of Stable-DiffCoder-8B-Instruct.

**LiveCodeBench.** To ensure a fair comparison under identical data and training settings, we adopt exactly the same evaluation configuration as Seed-Coder-8B-Instruct on the LiveCodeBench (v5, 2024.10-2025.02). Here Stable-DiffCoder-8B-Instruct (23.5%) is slightly behind Seed-Coder-8B-Instruct (24.7%), but matches the performance of Qwen3-8B and remains stronger than other ∼8B-scale models (Table 5).

**MBXP.** We further evaluate multilingual code generation with MBXP. As summarized in Table 6, under the 10+ programming language setting Stable-DiffCoder-8B-Instruct achieves an overall average comparable to Seed-Coder-8B-Instruct, while attaining the highest scores in most languages among ∼8B instruct models. Due to the need to extensively supplement the scarce data such as C# and PHP that are lacking in pretraining during SFT, the advantage in multilingual coding capabilities has been reduced.

**NaturalCodeBench.** On NaturalCodeBench, which targets practical software engineering problems in Python and Java, Stable-DiffCoder-8B-Instruct is slightly weaker than Seed-Coder-8B-Instruct on Chinese Python queries but is otherwise on par overall (Table 7). Both models, however, are clearly ahead of other ∼8B models and even competitive with many 13B+ and external API systems.

### 4.4.2 Code Reasoning

**CRUXEval.** For execution-centric code reasoning, we evaluate on CRUXEval. Consistent with the trends observed for base models, Table 8 shows that Stable-DiffCoder-8B-Instruct achieves stronger performance than Seed-Coder-8B-Instruct on the Output-CoT setting and slightly better average performance across

Input-CoT and Output-CoT. Nonetheless, Qwen3-8B remains ahead on CRUXEval, indicating that there is still considerable headroom for specialized small-code models on fine-grained reasoning tasks.

### 4.4.3 Code editing

**CanItEdit.** On CanItEdit, Stable-DiffCoder-8B-Instruct substantially outperforms all other models (Table 9). We hypothesize that this gain benefits from the denoising nature of DLLMs: random masking and reconstruction inherently train the model on edit- and infill-like patterns, enabling it to better exploit editing supervision and extract more editing-related knowledge from the same data.

**Aider.** On Aider, which evaluates multi-turn editing and reasoning over entire codebases, Stable-DiffCoder-8B-Instruct remains slightly weaker than Seed-Coder-8B-Instruct under the tries=2 setting (Table 9). This task requires concatenating outputs across turns, often yielding very long contexts that exceed the 8192-token window used during training, and we observe a mild performance drop in this regime. Even so, Stable-DiffCoder-8B-Instruct reaches performance comparable to Qwen3-8B and surpasses larger models such as DeepSeek-Coder-33B-Instruct, indicating that it still offers strong practical code editing abilities despite this limitation.

## 5 Conclusion, Limitation, and Future Work

In this report, we present Stable-DiffCoder, which serves as a practical best-practice attempt demonstrating that the training paradigm of diffusion language models can provide effective data augmentation and lead to improved model performance. By analyzing how to efficiently enhance knowledge learning in DLLMs, designing a well-aligned training pipeline, and incorporating techniques that stabilize and optimize the learning process, we show that—while keeping the architecture and data identical to Seed-Coder—the diffusion-based training procedure yields consistently better results. These findings further indicate that the sampling process of text diffusion models can function as a principled and effective form of data augmentation for model training. On comprehensive code evaluation benchmarks, Stable-DiffCoder achieves state-of-the-art results among$\sim$8B AR-based or diffusion-based code models.

Since Stable-DiffCoder is primarily focused on the code domain and lacks large-scale training data from other areas, its performance on mathematical reasoning and general-purpose text tasks may be relatively limited. Whether text diffusion sampling can provide even greater benefits in broader domains remains an open question, requiring future model iterations and deeper empirical exploration.

## 6 Contributions

**Project Lead**
Chenghao Fan[1,2]

**Core Contributor**
Chenghao Fan[1,2]

**Contributor***
Wen Heng[2], Bo Li[2], Sichen Liu[1], Yuxuan Song[2] Jing Su[2],

**Supervision***
Wen Heng[2], Xiaoye Qu[1], Kai Shen[2], Wei Wei[1],

(*Sorted Alphabetically)

**Affiliation**
[1]School of Computer Science & Technology, Huazhong University of Science and Technology
[2]ByteDance Seed

## 7 Acknowledgments

We gratefully acknowledge and thank every Seed-LLM team member not explicitly mentioned above. We also acknowledge and thank every Seed team member for the valuable support. We thank Shulin Xin, Qi Liu, Yirong Chen, Zhexi Zhang, Ziwen Xu, Shen Nie, Hongrui Zhan, Shen Zheng for insightful technical discussions.

## References


[1] Marianne Arriola, Aaron Gokaslan, Justin T Chiu, Zhihan Yang, Zhixuan Qi, Jiaqi Han, Subham Sekhar Sahoo, and Volodymyr Kuleshov. Block diffusion: Interpolating between autoregressive and diffusion language models. arXiv preprint arXiv:2503.09573, 2025.

[2] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. Multi-lingual evaluation of code generation models. In The Eleventh International Conference on Learning Representations, 2023. URL https://openreview.net/forum?id=Bo7eeXm6An8.

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. ArXiv preprint, abs/2108.07732, 2021. URL https://arxiv.org/abs/2108.07732.

[4] Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces, 2023. URL https://arxiv.org/abs/2107.03006.

[5] Tiwei Bie, Maosong Cao, Kun Chen, Lun Du, Mingliang Gong, Zhuochen Gong, Yanmei Gu, Jiaqi Hu, Zenan Huang, Zhenzhong Lan, et al. Llada2. 0: Scaling up diffusion language models to 100b. arXiv preprint arXiv:2512.15745, 2025.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. Advances in neural information processing systems, 33:1877–1901, 2020.

[7] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. IEEE Transactions on Software Engineering, 49(7):3675–3691, 2023. doi: 10.1109/TSE.2023.3267446. URL https://www.computer.org/csdl/journal/ts/2023/07/10103177/1MpWUtj7Rwk.

[8] Federico Cassano, Luisa Li, Akul Sethi, Noah Shinn, Abby Brennan-Jones, Jacob Ginesin, Edward Berman, George Chakhnashvili, Anton Lozhkov, Carolyn Jane Anderson, and Arjun Guha. Can it edit? evaluating the ability of large language models to follow code editing instructions. In First Conference on Language Modeling, 2024. URL https://openreview.net/forum?id=D06yk3DBas.

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

[10] Shuang Cheng, Yihan Bian, Dawei Liu, Linfeng Zhang, Qian Yao, Zhongbo Tian, Wenhai Wang, Qipeng Guo, Kai Chen, Biqing Qi, et al. Sdar: A synergistic diffusion-autoregression paradigm for scalable sequence generation. arXiv preprint arXiv:2510.06303, 2025.

[11] Jianbo Dai, Jianqiao Lu, Yunlong Feng, Dong Huang, Guangtao Zeng, Rongju Ruan, Ming Cheng, Haochen Tan, and Zhijiang Guo. MHPP: Exploring the capabilities and limitations of language models beyond basic code generation, 2024. URL https://arxiv.org/abs/2405.11430.

[12] Arpan Dasgupta, Preeti Lamba, Ankita Kushwaha, Kiran Ravish, Siddhant Katyan, Shrutimoy Das, and Pawan Kumar. Review of extreme multilabel classification, 2025. URL https://arxiv.org/abs/2302.05971.

[13] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. arXiv e-prints, pages arXiv–2407, 2024.

[14] Zitian Gao, Haoming Luo, Lynx Chen, Jason Klein Liu, Ran Tao, Joey Zhou, and Bryan Dai. What makes diffusion language models super data learners? arXiv preprint arXiv:2510.04071, 2025.

[15] Aritra Ghosh, Naresh Manwani, and P. Shanti Sastry. Making risk minimization tolerant to label noise. ArXiv, abs/1403.3610, 2014. URL https://api.semanticscholar.org/CorpusID:8774197.

[16] Shansan Gong, Shivam Agarwal, Yizhe Zhang, Jiacheng Ye, Lin Zheng, Mukai Li, Chenxin An, Peilin Zhao, Wei Bi, Jiawei Han, et al. Scaling diffusion language models via adaptation from autoregressive models. arXiv preprint arXiv:2410.17891, 2024.

[17] Shansan Gong, Ruixiang Zhang, Huangjie Zheng, Jiatao Gu, Navdeep Jaitly, Lingpeng Kong, and Yizhe Zhang. Diffucoder: Understanding and improving masked diffusion models for code generation. arXiv preprint arXiv:2506.20639, 2025.

[18] Google DeepMind. Gemini diffusion. https://deepmind.google/models/gemini-diffusion, 2025.

[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, 2024. URL https://arxiv.org/abs/2401.03065.

[20] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence, 2024. URL https://arxiv.org/abs/2401.14196.

[21] Siming Huang, Tianhao Cheng, Jason Klein Liu, Weidi Xu, Jiaran Hao, Liuyihan Song, Yang Xu, Jian Yang, Jiaheng Liu, Chenchen Zhang, et al. Opencoder: The open cookbook for top-tier code large language models. In Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 33167–33193, 2025.

[22] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. arXiv preprint arXiv:2409.12186, 2024.

[23] Daniel Israel, Tian Jin, Ellie Cheng, Guy Van den Broeck, Aditya Grover, Suvinay Subramanian, and Michael Carbin. Planned diffusion. arXiv preprint arXiv:2510.18087, 2025.

[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. In The Thirteenth International Conference on Learning Representations, 2025. URL https://openreview.net/forum?id=chfJJYC3iL.

[25] Wonjun Kang, Kevin Galim, Seunghyuk Oh, Minjae Lee, Yuchen Zeng, Shuibai Zhang, Coleman Hooper, Yuezhou Hu, Hyung Il Koo, Nam Ik Cho, et al. Parallelbench: Understanding the trade-offs of parallel decoding in diffusion llms. arXiv preprint arXiv:2510.04767, 2025.

[26] Jaeyeon Kim, Kulin Shah, Vasilis Kontonis, Sham Kakade, and Sitan Chen. Train for the worst, plan for the best: Understanding token ordering in masked diffusions. arXiv preprint arXiv:2502.06768, 2025.

[27] Frederik Kunstner, Robin Yadav, Alan Milligan, Mark Schmidt, and Alberto Bietti. Heavy-tailed class imbalance and why adam outperforms gradient descent on language models, 2024. URL https://arxiv.org/abs/2402.19449.

[28] Inception Labs. Introducing mercury: A new paradigm for fast language diffusion, 2025. URL https://www.inceptionlabs.ai/introducing-mercury. Accessed: 2025-04-05.

[29] Chengze Li, Yitong Zhang, Jia Li, Liyi Cai, and Ge Li. Beyond autoregression: An empirical study of diffusion large language models for code generation. arXiv preprint arXiv:2509.11252, 2025.

[30] Aiwei Liu, Minghua He, Shaoxun Zeng, Sijun Zhang, Linhao Zhang, Chuhan Wu, Wei Jia, Yuan Liu, Xiao Zhou, and Jie Zhou. Wedlm: Reconciling diffusion language models with standard causal attention for fast inference, 2025. URL https://arxiv.org/abs/2512.22737.

[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] 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. Advances in Neural Information Processing Systems, 36:21558–21572, 2023.

[33] Jingyu Liu, Xin Dong, Zhifan Ye, Rishabh Mehta, Yonggan Fu, Vartika Singh, Jan Kautz, Ce Zhang, and Pavlo Molchanov. Tidar: Think in diffusion, talk in autoregression. arXiv preprint arXiv:2511.08923, 2025.

[34] Yangzhou Liu, Yue Cao, Hao Li, Gen Luo, Zhe Chen, Weiyun Wang, Xiaobo Liang, Biqing Qi, Lijun Wu, Changyao Tian, et al. Sequential diffusion language models. arXiv preprint arXiv:2509.24007, 2025.

[35] Llama Team. Introducing llama 3.1: Our most capable models to date. https://ai.meta.com/blog/meta-llama-3-1/, July 2024.

[36] Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete diffusion modeling by estimating the ratios of the data distribution. In Forty-first International Conference on Machine Learning, 2024.

[37] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. StarCoder 2 and The Stack v2: The next generation, 2024. URL https://arxiv.org/abs/2402.19173.

[38] Yuxin Ma, Lun Du, Lanning Wei, Kun Chen, Qian Xu, Kangyu Wang, Guofeng Feng, Guoshan Lu, Lin Liu, Xiaojing Qi, et al. dinfer: An efficient inference framework for diffusion language models. arXiv preprint arXiv:2510.08666, 2025.

[39] Chenlin Meng, Kristy Choi, Jiaming Song, and Stefano Ermon. Concrete score matching: Generalized score matching for discrete data. Advances in Neural Information Processing Systems, 35:34532–34545, 2022.

[40] Amin Karimi Monsefi, Nikhil Bhendawade, Manuel Rafael Ciosici, Dominic Culver, Yizhe Zhang, and Irina Belousova. Fs-dfm: Fast and accurate long text generation with few-step diffusion language models. arXiv preprint arXiv:2509.20624, 2025.

[41] Jinjie Ni, Qian Liu, Longxu Dou, Chao Du, Zili Wang, Hang Yan, Tianyu Pang, and Michael Qizhe Shieh. Diffusion language models are super data learners. arXiv preprint arXiv:2511.03276, 2025.

[42] Jinjie Ni, Qian Liu, Chao Du, Longxu Dou, Hang Yan, Zili Wang, Tianyu Pang, and Michael Qizhe Shieh. Training optimal large diffusion language models. arXiv preprint arXiv:2510.03280, 2025.

[43] Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large language diffusion models, 2025. URL https://arxiv.org/abs/2502.09992.

[44] Jingyang Ou, Shen Nie, Kaiwen Xue, Fengqi Zhu, Jiacheng Sun, Zhenguo Li, and Chongxuan Li. Your absorbing discrete diffusion secretly models the conditional distributions of clean data, 2025. URL https://arxiv.org/abs/2406.03736.

[45] Qwen Team. Code with CodeQwen1.5, April 2024. URL https://qwenlm.github.io/blog/codeqwen1.5/.

[46] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950.

[47] Subham Sekhar Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and effective masked diffusion language models, 2024. URL https://arxiv.org/abs/2406.07524.

[48] Dale Schuurmans, Hanjun Dai, and Francesco Zanini. Autoregressive large language models are computationally universal. arXiv preprint arXiv:2410.03170, 2024.

[49] ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, et al. Seed-coder: Let the code model curate data for itself. arXiv preprint arXiv:2506.03524, 2025.

[50] Jiaxin Shi, Kehang Han, Zhe Wang, Arnaud Doucet, and Michalis Titsias. Simplified and generalized masked diffusion for discrete data. Advances in neural information processing systems, 37:103131–103167, 2024.

[51] Yuxuan Song, Zheng Zhang, Cheng Luo, Pengyang Gao, Fan Xia, Hao Luo, Zheng Li, Yuehang Yang, Hongli Yu, Xingwei Qu, et al. Seed diffusion: A large-scale diffusion language model with high-speed inference. arXiv preprint arXiv:2508.02193, 2025.

[52] Haocheng Sun, Cynthia Xin Wen, and Edward Hong Wang. Why mask diffusion does not work. arXiv preprint arXiv:2510.03289, 2025.

[53] Yuchuan Tian, Yuchen Liang, Jiacheng Sun, Shuo Zhang, Guangwen Yang, Yingte Shu, Sibo Fang, Tianyu Guo, Kai Han, Chao Xu, et al. From next-token to next-block: A principled adaptation path for diffusion llms. arXiv preprint arXiv:2512.06776, 2025.

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. Advances in neural information processing systems, 30, 2017.

[55] Dimitri von Rütte, Janis Fluri, Omead Pooladzandi, Bernhard Schölkopf, Thomas Hofmann, and Antonio Orvieto. Scaling behavior of discrete diffusion language models. arXiv preprint arXiv:2512.10858, 2025.

[56] Chengyue Wu, Hao Zhang, Shuchen Xue, Shizhe Diao, Yonggan Fu, Zhijian Liu, Pavlo Molchanov, Ping Luo, Song Han, and Enze Xie. Fast-dllm v2: Efficient block-diffusion llm. arXiv preprint arXiv:2509.26328, 2025.

[57] Chengyue Wu, Hao Zhang, Shuchen Xue, Zhijian Liu, Shizhe Diao, Ligeng Zhu, Ping Luo, Song Han, and Enze Xie. Fast-dllm: Training-free acceleration of diffusion llm by enabling kv cache and parallel decoding, 2025. URL https://arxiv.org/abs/2505.22618.

[58] Zirui Wu, Lin Zheng, Zhihui Xie, Jiacheng Ye, Jiahui Gao, Yansong Feng, Zhenguo Li, Victoria W., Guorui Zhou, and Lingpeng Kong. Dreamon: Diffusion language models for code infilling beyond fixed-size canvas, 2025. URL https://hkunlp.github.io/blog/2025/dreamon.

[59] Zhihui Xie, Jiacheng Ye, Lin Zheng, Jiahui Gao, Jingwei Dong, Zirui Wu, Xueliang Zhao, Shansan Gong, Xin Jiang, Zhenguo Li, et al. Dream-coder 7b: An open diffusion language model for code. arXiv preprint arXiv:2509.01142, 2025.

[60] Chenkai Xu, Yijie Jin, Jiajun Li, Yi Tu, Guoping Long, Dandan Tu, Tianqi Hou, Junchi Yan, and Zhijie Deng. Lopa: Scaling dllm inference via lookahead parallel decoding. arXiv preprint arXiv:2512.16229, 2025.

[61] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. arXiv preprint arXiv:2505.09388, 2025.

[62] Jiacheng Ye, Zhihui Xie, Lin Zheng, Jiahui Gao, Zirui Wu, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Dream 7b: Diffusion large language models. arXiv preprint arXiv:2508.15487, 2025.

[63] Shudan Zhang, Hanlin Zhao, Xiao Liu, Qinkai Zheng, Zehan Qi, Xiaotao Gu, Xiaohan Zhang, Yuxiao Dong, and Jie Tang. NaturalCodeBench: Examining coding performance mismatch on humaneval and natural user prompts, 2024. URL https://arxiv.org/abs/2405.04520.

[64] Xiong Zhou, Xianming Liu, Deming Zhai, Junjun Jiang, and Xiangyang Ji. Asymmetric loss functions for noise-tolerant learning: Theory and applications. IEEE Transactions on Pattern Analysis and Machine Intelligence, 45: 8094–8109, 2023. URL https://api.semanticscholar.org/CorpusID:256687692.

[65] Fengqi Zhu, Rongzhen Wang, Shen Nie, Xiaolu Zhang, Chunwei Wu, Jun Hu, Jun Zhou, Jianfei Chen, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Llada 1.5: Variance-reduced preference optimization for large language diffusion models, 2025. URL https://arxiv.org/abs/2505.19223.

[66] Fengqi Zhu, Zebin You, Yipeng Xing, Zenan Huang, Lin Liu, Yihong Zhuang, Guoshan Lu, Kangyu Wang, Xudong Wang, Lanning Wei, et al. Llada-moe: A sparse moe diffusion language model. arXiv preprint arXiv:2509.24389, 2025.

[67] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. arXiv preprint arXiv:2406.11931, 2024.

[68] Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen GONG, James Hoang, Armel Randy

Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In The Thirteenth International Conference on Learning Representations, 2025. URL https://openreview.net/forum?id=YrycTjllL0.