Can Mamba Be Better? An Experimental Evaluation of Mamba in Code Intelligence

Shuo Liu[†], Jacky Keung[†], Zhen Yang[‡]*, Zhenyu Mao[†], and Yicheng Sun[†]
[†]Department of Computer Science, City University of Hong Kong, Hong Kong, China,
{sliu273-c, zhenyumao2-c, yicsun2-c}@my.cityu.edu.hk, jacky.keung@cityu.edu.hk
[‡] School of Computer Science and Technology, Shandong University, Qingdao, China, zhenyang@sdu.edu.cn

***Abstract*—The Transformer architecture and its core attention mechanism form the foundation of Pre-trained Language Models (PLMs) and have driven their remarkable progress across a wide range of code intelligence tasks. However, the quadratic complexity inherent in the attention mechanism poses scalability challenges. Recently, sub-quadratic architectures such as Mamba and Mamba-2 have emerged as compelling alternatives to the Transformer. While they have shown promising results and attracted increasing academic interest, their effectiveness in code intelligence tasks has not yet been fully explored.**

To fill this gap, we present the first systematic empirical study of Mamba-based PLMs on three typical code tasks (i.e., code completion, code generation, and code clone detection), covering both the code comprehension and generation categories to delve into their effectiveness and efficiency. We first pre-train two Mamba-based PLMs on code based on Mamba and Mamba-2, respectively. Subsequently, we evaluate these four PLMs against typical Transformer-based PLMs (e.g., CodeGPT) with Full fine-Tuning (FT) and Parameter-Efficient Fine-Tuning (PEFT) settings, demonstrating the overall superiority of Mamba-based PLMs across all code tasks. Subsequent experiments involve the architecture analysis via pre-training from scratch to isolate the influence of the training corpora and low-resource analysis via deliberately limiting the fine-tuning data volume. All demonstrate the superiority of Mamba-based PLMs in both efficacy and efficiency. Finally, we also extend the sizes of PLMs to larger scales (7B at most) and make comparisons with more diverse PLMs/LLMs. Experimental results demonstrate that pre-training corpora and tasks also heavily affect the code modeling performance, apart from architectures. This work provides a comprehensive investigation into Mamba-based PLMs in the context of code intelligence, uncovering their strengths, limitations, and potential for future applications.

***Index Terms*—Code Language Models, Transformer, Mamba, Parameter-Efficient Fine-Tuning**

I. Introduction

In modern software engineering, Pre-trained Language Models (PLMs) on code have become essential tools for assisting developers in programming [1], and have achieved remarkable success in a variety of code comprehension and generation tasks [2]–[4]. Based on the Transformer architecture and its core attention mechanism, code PLMs are first pre-trained on large-scale corpora to learn general-purpose code representations. Subsequently, they are fine-tuned on specific downstream tasks to adapt to the target domain. Autoregressive modeling is the most widely adopted paradigm in training code PLMs, which generates code sequences in a left-to-right, token-by-token fashion [5].

Despite the advancements achieved by code PLMs, their core attention module encounters an intrinsic drawback: it scales quadratically with sequence length [6]. Due to the attention mechanism computing attention weights for every token pair, it requires a two-dimensional matrix for storage. As the input length increases and the number of tokens grows, the computational complexity rises quadratically. This is why Transformer-based PLMs usually impose a maximum input length to ensure computational feasibility and model effectiveness. Numerous studies have emerged to tackle these issues [7]. Among them, Mamba [8] and Mamba-2 [9] are the most influential and representative works.

Mamba is a deep sequence model architecture. Similar to the Transformer, it consists of stacked blocks that integrate selective State Space Models (SSMs) and Multi-Layer Perceptrons (MLPs). The SSMs are regarded as a powerful alternative to the attention mechanism, transforming inputs to corresponding outputs through an implicit latent state. SSM is conceptually related to Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs), making it well-suited for modeling sequential data. Based on SSMs, Mamba extends them to selective SSMs, which enable input-conditioned parameterization and context-aware modeling [8]. To ensure practical applicability, Mamba also designs a hardware-aware algorithm that facilitates efficient training on modern parallelized hardware like GPUs. Moreover, building upon the advantages of both Mamba and the Transformer, Mamba-2 is introduced with several architectural refinements. These changes involve eliminating redundant sequential linear projections and incorporating normalization layers [9]. Mamba and Mamba-2 have achieved state-of-the-art performance in comparison with Transformer-based models of similar size, spanning diverse domains such as language [10], vision [11], and graphs [12]. However, in the domain of code intelligence, related studies are restricted to efficiency analysis of Mamba-generated code [13] or limited code tasks (e.g., test automation) [14], lacking systematic exploration of Mamba-based PLMs on code comprehension and generation tasks.

To the best of our knowledge, this work is the first systematic empirical study of Mamba-based PLMs in the code intelligence field. Focusing primarily on Mamba and Mamba-2 with 130M parameters, we conduct experiments to evaluate their

*Corresponding author.

performance on three widely-studied code tasks: (1) code completion, (2) code generation, and (3) code clone detection. We select CodeGPT and CodeGPT-adapted [15] as representative Transformer-based code PLMs for comparison, due to their similar model size and the use of an autoregressive modeling approach that aligns with Mamba. Considering that Mamba and Mamba-2 are not pre-trained on code corpora, we further pre-train them on monolingual Java and Python datasets from CodeSearchNet [16], respectively, obtaining Mamba$_{code}$ and Mamba-2$_{code}$. We begin by assessing the performance of the above PLMs under the Full fine-Tuning (FT) setting. Given the growing importance of Parameter-Efficient Fine-Tuning (PEFT) as a viable substitute for FT, we also incorporate two PEFT methods, LoRA [17] and $(IA)^3$ [18], into our evaluation. To control for the confounding factor of different pre-training datasets, we pre-train CodeGPT, Mamba, and Mamba-2 from scratch using the same code corpora, aiming to investigate whether the performance advantage originates from architectural differences. We also analyze the training efficiency of these PLMs during the fine-tuning stage. In addition, we conduct experiments in the low-resource scenario, motivated by the frequent occurrence of data scarcity in real-world applications. Finally, we also extend Mamba's size to 370M and 7B, thereby making comparison with more diverse PLMs and even LLMs, as well as exploring the scaling law of Mamba-based PLMs.

Our experimental results demonstrate that: (1) In the FT setting, Mamba-based PLMs outperform CodeGPT while performing neck-to-neck with CodeGPT-adapted. With further pre-training on code corpora, their performance increases and surpasses the above CodeGPT variants, indicating their high adaptability to code modeling. (2) The performance advantage of Mamba-based PLMs becomes more pronounced in the PEFT setting, highlighting their practical value in limited computational resources. (3) After uniform pre-training from scratch, Mamba-2 demonstrates superior performance and memory efficiency in the downstream task against CodeGPT, suggesting that its architectural innovations contribute to better capability. (4) In the low-resource scenario, Mamba-based PLMs demonstrate strong stability overall against CodeGPT variants across all code tasks. In particular, when the training data is reduced to 0.1%, Mamba-based PLMs suffer a performance degradation of only 3.69% in the Java code completion task. (5) Scaling up Mamba-based PLMs' sizes still obtains performance gains, but compared with other counterparts, pre-training databases and tasks also heavily affect the code modeling performance, apart from model architectures.

The contributions of this paper are fourfold:

1. To the best of our knowledge, this paper serves as the first systematic empirical study to investigate Mamba-based PLMs in the context of code intelligence.
2. We perform an in-depth comparison between Mamba-based and Transformer-based PLMs across multiple fine-tuning methods, including FT, LoRA, and $(IA)^3$, exploring the performance advantages and computational efficiency of the Mamba architecture.
3. We also evaluate Mamba-based PLMs in low-resource scenarios and explore their performance at larger scales, providing practical suggestions and exposing critical weaknesses of Mamba-based PLMs for reference of both academia and industry.
4. We release all source code and pre-trained models at [19].

II. Preliminary

A. Transformer

The Transformer [20] is the cornerstone for current code LMs. By leveraging its self-attention mechanism, it is capable of capturing long-range dependencies, thereby supporting rich contextual understanding. Each Transformer block contains two main sub-layers: Multi-Head Attention (MHA) and fully connected Feed-Forward Network (FFN). The MHA enables models to selectively focus on important information and capture complex patterns, thereby significantly enhancing their capacity. The FFN layer follows the MHA, consisting of two linear transformations with a ReLU activation in between.

B. State Space Models, Mamba, and Mamba-2

State Space Models (SSMs) [21] are a recent class of sequence models for deep learning, inspired by a particular continuous system. Its primary advantages lie in the linear time and memory complexity with respect to sequence length, as opposed to the quadratic scaling of attention mechanisms. This property facilitates effective and efficient modeling of long-range sequential dependencies, making it particularly suitable for tasks involving extended sequences such as natural language and code. SSMs take an input x_t, mapping it to y_t through an implicit latent state h_t. The transformation can be expressed in two stages:

$$h'_t = Ah_t + Bx_t, \quad y_t = Ch_t \tag{1}$$

As this model describes the relationship between continuous quantities, it requires discretization before further processing. Additionally, an important property of SSMs is Linear Time Invariance (LTI), which means the model's parameters remain constant across all time steps, thereby decreasing the performance when representing changing inputs [22]. To tackle this issue, Mamba [8] introduces the selective SSMs, which formulate parameters as functions of specific inputs. Given an input sequence of L tokens $I \in \mathbb{R}^{L\times d_m}$, where d_m is the dimension of input tokens, a Mamba block transforms it into the output sequence $O \in \mathbb{R}^{L\times d_m}$ through the following steps:

$$X = \sigma\left(Conv1D\left(Linear_{in}\left(I\right)\right)\right), \quad Y = SSM\left(X\right) \tag{2}$$

$$O = Linear_{out}\left(\sigma\left(Linear_{gate}\left(I\right)\right) \odot Y\right) \tag{3}$$

where $Linear_{in}$, $Linear_{gate}$, and $Linear_{out}$ are linear projections. σ is an activation function, and $\odot$ represents the element-wise multiplication. Conv1d is an abbreviation for the one-dimensional convolution operation. Prior to the B and C matrices, a linear projection is also applied, which we denote as $Linear_x$.

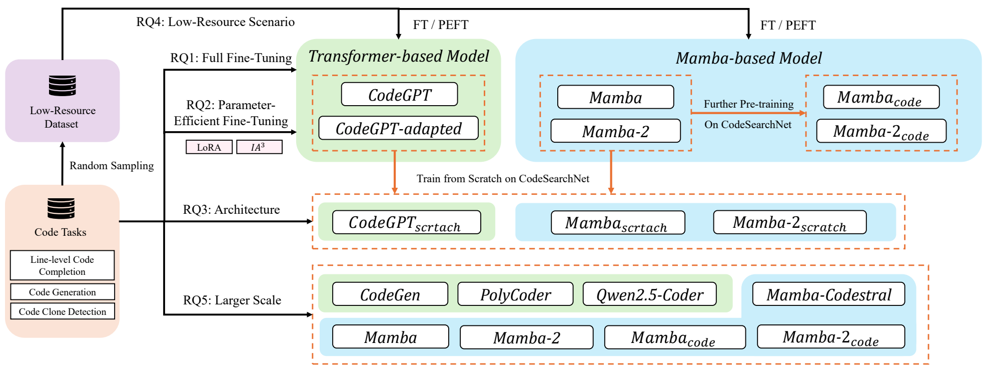


Fig. 1. The overall workflow. There are a total of five research questions.

Inspired by the connection between SSMs and the Transformer, Mamba-2 [9] refines the Mamba architecture by removing sequential linear projections. Alternatively, the parameters A, B, and C are produced in parallel with a single projection at the beginning of the block. Furthermore, an additional normalization layer is introduced to enhance stability.

C. LoRA and $(IA)^3$

LoRA [17], short for Low-Rank Adaptation, is a parameter-efficient fine-tuning method based on the hypothesis that updates to weight matrices have a low intrinsic rank. For a pre-trained weight matrix $W_0 \in \mathbb{R}^{d\times k}$, instead of directly learning the update ΔW, LoRA decomposes it into two low-rank matrices $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times k}$, where the rank satisfies $r \ll min(d,k)$. Therefore, the update of the weight matrix becomes a low-rank approximation: $W_0 + \Delta W = W_0 + BA$. To ensure that the initial update is zero, A uses a random Gaussian initialization while B is initialized with zeros. During tuning, W_0 is frozen, and only A and B are trainable. In the Transformer architecture, LoRA is usually applied to the *query* and *value* matrices. However, as research on PEFT methods for Mamba remains limited, based on prior work [23], we apply LoRA to the *x*, *in*, and *out* projection matrices in Mamba, along with the *in* and *out* projection matrices in Mamba-2.

$(IA)^3$ [18] stands for "Infused Adapter by Inhibiting and Amplifying Inner Activations", proposing an adaptation method of the form $l \odot x$, where l is a learnable vector and $\odot$ denotes the element-wise multiplication. In comparison to LoRA, it is much more parameter-efficient, requiring only a small portion of trainable parameters. In the Transformer architecture, $(IA)^3$ introduces three such learnable vectors, $l_k \in \mathbb{R}^{d_k}$, $l_v \in \mathbb{R}^{d_v}$, and $l_{ff} \in \mathbb{R}^{d_{ff}}$. These vectors respectively rescale the *key* and *value* matrices in the attention module and the intermediate activations within the feed-forward network. Given the similarity in the target module selection between $(IA)^3$ and LoRA, in the Mamba architecture, we specify to deploy $(IA)^3$ on the same modules as LoRA.

III. Experimental Setup

A. Research Questions

This paper investigates the following Research Questions (RQs), intending to explore the performance of Mamba in code intelligence and make comparisons with the Transformer counterparts. Figure 1 shows the overall workflow. Below experiments concerning PLMs tuning are repeated 5 times with different initialization seeds, while LLMs are also required to repeat 5 times of inference for each experiment.

RQ1: What is the effectiveness of Mamba in the Full fine-Tuning setting? Full fine-Tuning (FT) is the most straightforward approach to adapting PLMs to downstream tasks. In this RQ, we chose the 130M parameter-sized models of Mamba and Mamba-2 to conduct experiments. Accordingly, we use CodeGPT [15] as the representative of the Transformer counterpart, given its similar model size. Considering that Mamba and Mamba-2 are not particularly pre-trained on code corpora, to strengthen their code capacity, we perform further pre-training on them with autoregressive language modeling, using the Java and Python portions of CodeSearchNet [16], respectively. Thus, obtaining Mamba$_{code}$ and Mamba-2$_{code}$. We conducted a manual inspection of the used CodeSearchNet dataset and all associated downstream datasets and found no evidence of data overlap. All the PLMs above are involved in FT experiments on both code comprehension and generation tasks for evaluation.

RQ2: What is the effectiveness of Mamba in the parameter-efficient fine-tuning setting? Parameter-Efficient Fine-Tuning (PEFT) is increasingly recognized as a crucial approach for adapting PLMs, particularly because of its computational efficiency [24], [25]. Hence, it is worthwhile to explore the effectiveness of Mamba-based PLMs under PEFT settings. For this RQ, we adopt two representative PEFT methods, namely LoRA and $(IA)^3$. According to prior work [23], we specify target modules in the Mamba architecture to inject two PEFT methods. All PLMs and code tasks in RQ1 are experimented with in this RQ for a comprehensive evaluation.

RQ3: What are the advantages of Mamba architecture? Although previous RQs have made a series of comparisons between Transformer and Mamba-based PLMs, we still cannot know whether the performance discrepancies are derived from their architecture, as their pre-training corpora are not identical. Hence, we pre-train CodeGPT, Mamba, and Mamba-2 from scratch with autoregressive language modeling, thereby obtaining CodeGPT$_{scratch}$, Mamba$_{scratch}$, and Mamba-2$_{scratch}$. Subsequently, we analyze their performance

TABLE I
DATASET STATISTICS.

Tasks	Datasets	#Train	#Valid	#Test	|Code|	|NL|	Lang.
LCC	JavaCorpus	12,934	7,189	3,000	1320.11	–	Java
	PY150	95,000	5,000	10,000	1344.13	–	Python
CG	CONCODE	100,000	2,000	2,000	35.71	163.79	Java
CCD	BigCloneBench	901,028	415,416	415,416	1455.04	–	Java

across all code tasks from the aspects of efficacy and efficiency.

RQ4: How capable is Mamba in the low-resource scenario? Since low-resource scenarios are frequently encountered in real-world software development and maintenance, such as adaptation to new technical stacks or software tasks [24], [26]. Therefore, it is also essential to explore whether Mamba can perform well in such a situation. To be specific, we trained PLMs using samples randomly selected from the original training set, while keeping the validation and test sets unchanged. Seven different sampling ratios are set, ranging from 0.1% to 30%, and experiments are conducted under both FT and PEFT settings on all code tasks across two CodeGPT variants, Mamba$_{code}$ and Mamba-2$_{code}$.

RQ5: How does the performance of Mamba as its scale increases? The scaling law is a notable feature of the Transformer architecture [27], describing that the performance of PLMs improves with increases in scale. In this RQ, we extend four Mamba-based PLMs, i.e., Mamba, Mamba-2, and their code versions, at larger scales of 370M. Besides, we also include Codestral Mamba [28], a Mamba-based Large Language Model (LLM) with 7B parameters. To make a fair comparison between Transformer-based counterparts, CodeGen [29], PolyCoder [30], and Qwen2.5-Coder [31] of similar sizes to Mamba-based PLMs/LLMs are chosen for evaluation. In this RQ, PLMs are tuned with FT. As for LLMs around 7B, considering their most prevalent usage convention [32]–[34] and limitation of computational resources, we perform zero-shot learning for experiments. Specifically, for code completion and generation tasks, we feed LLMs with code prefixes or NL descriptions and ask them to complete the follow-up code. As for code clone detection, we formulate a basic instruction for guidance, see [19] for specific examples. All three code tasks are evaluated in this RQ.

B. Tasks and Datasets

We evaluate the effectiveness of Mamba on three widely studied typical code tasks covering both code comprehension and generation categories [35]–[39]: (1) code completion task, (2) code generation task, and (3) code clone detection task, thereby ensuring a comprehensive assessment.

1) Line-level Code Completion (LCC): Code completion has become an indispensable feature in modern Integrated Development Environments (IDEs) [5]. It contributes to higher developer productivity by providing context-driven code suggestions [40]. As in prior studies [15], [41], the task can be categorized into two scenarios: token-level code completion and line-level code completion. While the former aims at predicting the next single token, the latter evaluates the quality of the entire generated line. Due to its broader scope, line-level code completion is considered a more challenging task. We conduct experiments on two high-quality datasets, JavaCorpus [42] and PY150 [43]. Their statistical information is presented in Table I, where |Code| and |NL| denote the average lengths of code snippets and natural language descriptions after tokenization. Since natural language is not involved in the code completion task, the corresponding entry is left blank.

2) Code Generation (CG): Code generation is a particularly intriguing task, involving producing source code from natural language descriptions [3]. It holds significant promise for reducing development costs and reshaping programming paradigms, thereby attracting substantial attention from both industry and academia [44]. To carry out this task, we choose a widely used dataset, CONCODE [45]. It collects about 33,000 Java repositories from GitHub and splits them into training, validation, and test sets at the repository level, ensuring that the domains in the test set are distinct from those in the training set. Its input summary includes the relevant environment variables and methods. Detailed statistics are also shown in Table I.

3) Code Clone Detection (CCD): Detecting code clones, i.e., duplicated or highly similar code snippets, is critical in software engineering. It is helpful for code refactoring and bug detection [46]. It can be viewed as a binary classification task that takes a pair of code snippets as input and determines whether a code clone exists between them. We choose one well-known dataset, BigCloneBench [47], for evaluation. Its detailed statistics are shown in Table I. In this paper, due to computational resource constraints and to maintain a comparable scale with other code tasks, we sample the dataset at a ratio of 15%. During sampling, we preserved the original positive-to-negative class ratio to ensure distributional consistency. The final training, validation, and test sets contain 135,154, 62,312, and 62,312 samples, respectively.

C. Pre-trained Language Models

Our evaluation is mainly conducted on CodeGPT, Mamba, and Mamba-2. Furthermore, we incorporate additional large-scale code PLMs that are widely adopted to enable a more thorough assessment.

- **CodeGPT** [15] is a code-oriented extension of GPT-2 [48], retaining the same model architecture and training objectives. It is pre-trained monolingually on Java and Python corpora, consisting of 1.6 million Java code samples and 1.1 million Python code samples. Moreover, it can be grouped into two categories: CodeGPT and CodeGPT-adapted. The former is pre-trained from scratch, while the latter uses the GPT-2 as the starting point. It has 125M parameters.
- **Mamba** [8] and **Mamba-2** [9] are designed for autoregressive language modeling, pre-trained on the Pile dataset [49]. Their scales are from 130M to 1.4B, and we use their 130M-sized and 370M-sized models in this paper. In addition to their original models, our experiments also pre-trained Mamba$_{code}$, Mamba-2$_{code}$, Mamba$_{scratch}$, and Mamba-2$_{scratch}$ for evaluation as mentioned in RQ1 and RQ3 of Section III-A.

• **CodeGen** [29] is a family of code PLMs designed to enhance its capacities through conversational program synthesis. In this paper, we use the CodeGen-Multi model for Java and the CodeGen-Mono model for Python. Both 350M and 6B versions are included.

• **PolyCoder** [30] is also built upon the GPT-2 architecture and is pre-trained on a large-scale corpus of over 24 million code files covering 12 different Programming Languages (PLs). We use the 400M parameter-sized model in our experiments.

• **Qwen2.5-Coder** [31] comprises a range of open-source models, having achieved significant advancements when compared with leading code LMs. In this paper, we adopt its 500M-sized and 7B-sized models.

• **Codestral Mamba** [28] is a large-scale, Mamba-based code LLM released by Mistral AI [50]. Built upon the Mamba-2 architecture, it contains 7 billion parameters.

D. Evaluation Metrics

Following previous studies [51]–[54], we adopt the Exact Match (EM) and Edit Similarity (ES) as the evaluation metrics for line-level code completion, while using EM, BLEU [55], and CodeBLEU [56] for code generation. EM measures the accuracy of generated outputs by token-by-token comparing them with the ground truth, thereby serving as a restrictive metric. ES quantifies the resemblance between the predicted output $\hat{Y}$ and the ground truth Y by calculating the minimum number of edit operations. BLEU compares n-grams of the generated output with n-grams of the ground truth, counting the number of matches and penalizing overly short outputs. CodeBLEU builds upon BLEU, enhancing it with code-specific insights like code syntax via Abstract Syntax Tree (AST) and code semantics via data-flow. For code clone detection, we use the F1 score as the evaluation metric, consistent with previous studies [46], [57]. It is the harmonic mean of precision and recall.

E. Implementation Details

The code PLMs we used are from Huggingface [58], and we keep their default hyperparameter settings. In the line-level code completion task, following the prior work [15], we set a maximum number of training epochs to 5, along with a maximum input sequence length of 1024. The learning rate is 8e-5 for all code PLMs in the FT setting, but is adjusted in the range of {1e-2, 5e-3, 2e-3, 1e-3, 5e-4} in the PEFT setting. In the code generation task, we fixed the maximum epochs to 5 and the maximum input length to 512, ensuring consistency across all experiments. For FT, the learning rate is searched within {2e-4, 1e-4, 8e-5}, while for PEFT, the search range is the same as that used in the code completion task. In the code clone detection task, we set the maximum number of training epochs to 5 for both FT and LoRA, and 10 for $(IA)^3$. We use a learning rate of 5e-5 and limit the maximum input sequence length to 1024 tokens. For a fair comparison between Transformer and Mamba architectures, we adopt greedy search with a beam size equals 1, and we keep the top-1 generated sequence as the final output. The training paradigm of autoregression modeling is adopted in code completion and generation tasks. As for code clone detection, following [59], [60], two code samples are concatenated as model input, and we fetch the last token's representation of the last model layer for binary classification. All experiments were conducted on two GPU servers running Ubuntu: One with eight RTX 4090 24GB GPUs and another with ten L20 48GB GPUs.

IV. Experimental Results

A. RQ1: Performance of Full Fine-Tuning

Since our objective is to explore the effectiveness of Mamba in code intelligence, we start with an evaluation in the FT setting. The detailed results are presented in Table II. In the comparison among CodeGPT, CodeGPT-adapted, Mamba, and Mamba-2, we can observe that Mamba-based PLMs outperform CodeGPT overall, while being on par with CodeGPT-adapted. In particular, on the CONCODE dataset, the performance of Mamba surpasses CodeGPT and CodeGPT-adapted by 5.25%–14.77% in terms of EM, by 2.13%–5.11% in terms of BLEU, and by 2.56%–4.76% in terms of CodeBLEU, respectively. To validate the significance of the performance difference, we perform Wilcoxon Signed-Rank Tests (WSRT) [61] with a confidence level of 95% between Mamba-based PLMs and two CodeGPT variants, respectively. As can be seen from Table II, the superiority of Mamba-based PLMs against CodeGPT is consistently significant across almost all studied code tasks. As for CodeGPT-adapted, the performance differences are reduced, where only Mamba-2 surpasses on BigCodeBench, and Mamba excels on CONCODE. Nonetheless, given that Mamba-based models are not specifically pre-trained on large-scale code corpora, their exhibited performance still highlights the potential of the Mamba architecture in both code comprehension and generation tasks.

To enhance the code modeling capabilities of Mamba and Mamba-2, we further pre-train them on monolingual corpora of Java and Python, respectively. The pre-training corpora consist of approximately 500 thousand code programs for each PL. As shown in Table II, a performance gain can be observed in Mamba-based PLMs (i.e., Mamba$_{code}$ and Mamba-2$_{code}$), making their performance superiority more apparent against the two CodeGPT variants. To be specific, except for PY150 with a neck-to-neck performance, Mamba$_{code}$ and Mamba-2$_{code}$ significantly excel CodeGPT on all other code tasks, where Mamba-2$_{code}$ even manifests significant advantages against CodeGPT-adapted, according to the WSRT. For instance, on JavaCorpus, Mamba-2$_{code}$ outperforms two CodeGPT variants by 2.47%–12.1% in terms of EM, and by 1.2%–4.28% in terms of ES. As for CONCODE, Mamba-2$_{code}$ surpasses them by 4.57%–14% in terms of EM, by 3.62%–6.38% in terms of BLEU, and by 2.89%–5.10% in terms of CodeBLEU. Moreover, Mamba-2$_{code}$ also keeps its superiority on BigCodeBench, excelling by 1.94%–2.16% in terms of F1 score. Despite little flaws, the evaluation results suggest that overall Mamba-based PLMs can demonstrate

TABLE II
Evaluation results in the FT and parameter-efficient fine-tuning settings. The Mamba and Mamba-2 used here are 130M-sized.

Method	Models	JavaCorpus		PY150		CONCODE			BigCloneBench
		EM	ES	EM	ES	EM	BLEU	CodeBLEU	F1
FT	CodeGPT	27.43$_{\pm0.21}$	63.79$_{\pm0.30}$	37.81$_{\pm0.27}$	70.26$_{\pm0.43}$	16.25$_{\pm0.54}$	31.50$_{\pm0.32}$	32.56$_{\pm0.32}$	0.927$_{\pm0.005}$
	CodeGPT-adapted	30.01$_{\pm0.10}$	65.73$_{\pm0.24}$	37.77$_{\pm0.18}$	70.39$_{\pm0.25}$	17.72$_{\pm0.48}$	32.42$_{\pm0.84}$	33.26$_{\pm0.92}$	0.925$_{\pm0.002}$
	Mamba	28.31$_{\pm0.38}$	64.91$_{\pm0.12}$	36.80$_{\pm0.15}$	70.08$_{\pm0.10}$	18.65$_{\pm0.33}$	33.11$_{\pm0.52}$	34.11$_{\pm0.72}$	0.919$_{\pm0.007}$
	Mamba-2	29.16$_{\pm0.10}$	65.68$_{\pm0.32}$	**37.93**$_{\pm0.17}$	70.45$_{\pm0.17}$	18.47$_{\pm0.23}$	32.58$_{\pm0.51}$	33.34$_{\pm0.68}$	0.944$_{\pm0.003}$
	Mamba$_{code}$	29.39$_{\pm0.26}$	65.91$_{\pm0.10}$	37.25$_{\pm0.14}$	**70.53**$_{\pm0.11}$	**19.02**$_{\pm0.16}$	33.08$_{\pm0.08}$	33.50$_{\pm0.24}$	0.943$_{\pm0.001}$
	Mamba-2$_{code}$	**30.75**$_{\pm0.11}$	**66.52**$_{\pm0.26}$	37.68$_{\pm0.09}$	70.44$_{\pm0.03}$	18.53$_{\pm0.21}$	**33.51**$_{\pm0.33}$	**34.22**$_{\pm0.45}$	**0.945**$_{\pm0.003}$
LoRA	CodeGPT	25.42$_{\pm0.31}$	61.88$_{\pm0.47}$	31.99$_{\pm0.30}$	66.22$_{\pm0.27}$	15.65$_{\pm0.38}$	27.05$_{\pm0.30}$	27.96$_{\pm0.32}$	0.901$_{\pm0.003}$
	CodeGPT-adapted	28.71$_{\pm0.21}$	64.61$_{\pm0.18}$	33.53$_{\pm0.19}$	67.19$_{\pm0.10}$	17.78$_{\pm0.30}$	27.95$_{\pm0.61}$	28.80$_{\pm0.58}$	0.893$_{\pm0.006}$
	Mamba	27.72$_{\pm0.35}$	64.77$_{\pm0.19}$	34.56$_{\pm0.17}$	68.47$_{\pm0.08}$	18.12$_{\pm0.88}$	30.98$_{\pm0.45}$	31.62$_{\pm0.35}$	0.924$_{\pm0.003}$
	Mamba-2	28.59$_{\pm0.11}$	65.38$_{\pm0.36}$	35.11$_{\pm0.25}$	68.79$_{\pm0.06}$	17.83$_{\pm0.21}$	31.23$_{\pm0.25}$	31.91$_{\pm0.26}$	0.932$_{\pm0.004}$
	Mamba$_{code}$	29.23$_{\pm0.24}$	66.03$_{\pm0.17}$	35.23$_{\pm0.10}$	68.98$_{\pm0.10}$	**18.15**$_{\pm0.09}$	31.25$_{\pm0.20}$	31.93$_{\pm0.11}$	0.933$_{\pm0.001}$
	Mamba-2$_{code}$	**30.00**$_{\pm0.12}$	**66.48**$_{\pm0.16}$	**35.87**$_{\pm0.07}$	**69.24**$_{\pm0.07}$	17.67$_{\pm0.30}$	**31.73**$_{\pm0.44}$	**32.38**$_{\pm0.56}$	**0.935**$_{\pm0.003}$
$(IA)^3$	CodeGPT	21.62$_{\pm0.27}$	58.03$_{\pm0.32}$	23.42$_{\pm0.30}$	59.21$_{\pm0.14}$	10.57$_{\pm0.37}$	21.40$_{\pm0.81}$	22.99$_{\pm0.97}$	0.728$_{\pm0.007}$
	CodeGPT-adapted	25.52$_{\pm0.27}$	62.07$_{\pm0.25}$	26.41$_{\pm0.15}$	62.26$_{\pm0.27}$	13.90$_{\pm0.50}$	23.33$_{\pm0.53}$	24.47$_{\pm0.58}$	0.724$_{\pm0.060}$
	Mamba	25.07$_{\pm0.10}$	62.99$_{\pm0.05}$	25.88$_{\pm0.17}$	61.84$_{\pm0.16}$	15.03$_{\pm0.31}$	23.61$_{\pm0.12}$	24.69$_{\pm0.13}$	0.768$_{\pm0.020}$
	Mamba-2	25.67$_{\pm0.04}$	63.30$_{\pm0.16}$	26.44$_{\pm0.20}$	62.32$_{\pm0.16}$	15.33$_{\pm0.23}$	23.79$_{\pm0.67}$	24.86$_{\pm0.75}$	0.744$_{\pm0.017}$
	Mamba$_{code}$	26.56$_{\pm0.14}$	64.25$_{\pm0.13}$	28.15$_{\pm0.07}$	64.00$_{\pm0.13}$	15.63$_{\pm0.10}$	23.97$_{\pm0.57}$	25.08$_{\pm0.55}$	**0.798**$_{\pm0.005}$
	Mamba-2$_{code}$	**27.36**$_{\pm0.22}$	**64.67**$_{\pm0.09}$	**28.73**$_{\pm0.09}$	**64.20**$_{\pm0.13}$	**15.65**$_{\pm0.30}$	**24.52**$_{\pm0.68}$	**25.60**$_{\pm0.62}$	0.795$_{\pm0.005}$

* Table cells with a light green background indicate statistically significant improvements over CodeGPT, while those with a light red background indicate statistically significant improvements over both CodeGPT and CodeGPT-adapted.

superior performance when compared with Transformer-based counterparts in both code comprehension and generation tasks.

Finding 1: When using FT, Mamba and Mamba-2 achieve superior or at least comparable performance relative to CodeGPT variants across all code tasks. After further pre-training on codes, their performances are enhanced significantly.

B. *RQ2: Performance of Parameter-Efficient Fine-Tuning*

As tuning PLMs with FT usually incurs high computational and memory costs, alternative approaches such as Parameter-Efficient Fine-Tuning (PEFT) have become increasingly popular in practice. In this section, we employ LoRA and $(IA)^3$ to evaluate the effectiveness of Mamba-based PLMs, presenting the results in Table II. As can be seen, with the application of PEFT methods, the performance of Mamba and Mamba-2 falls short of that achieved through FT. This trend is similar to that observed in CodeGPT, indicating that only updating a small portion of parameters normally cannot realize PLMs' adequate learning. Focusing on experiments with LoRA, the four Mamba-based PLMs surpass the two CodeGPT variants a lot compared with the FT setting in RQ1. Statistical tests with WSRT also demonstrate the statistically significant superiority of Mamba-based PLMs across most code tasks. In particular, for the code completion task, Mamba-2$_{code}$ surpasses two CodeGPT variants by 4.49%–18.02% in terms of EM and by 1.2%–4.28% in terms of ES, on JavaCorpus. On PY150, it also outperforms them by 6.98%–12.13% and by 3.05%–4.56% in terms of each metric in order. As for the code generation task, except for the EM metric, Mamba-2$_{code}$ obtains a significant lead by 13.52%–17.3% and by 12.43%–15.81% in terms of BLEU and CodeBLEU, respectively. We manually examined the code samples and found that Mamba-based PLMs tend to generate longer outputs, resulting in their relatively high BLEU and CodeBLEU scores. But very few of them exactly match the ground truths. Regarding the code clone detection task, Mamba-2$_{code}$ still showcases its advantage by 3.77% in terms of F1 score.

When using $(IA)^3$, the performance of all PLMs is inferior to that of LoRA, which can be attributed to the significantly smaller trainable parameter size. Specifically, LoRA updates on average 1.86% of parameters for CodeGPT variants and 7.83% for Mamba-based counterparts. However, on average, approximately only 0.05% for CodeGPT variants and around 0.09% Mamba-based PLMs are tuned with $(IA)^3$. As for the performance comparison within PLMs tuned by $(IA)^3$, Mamba-based PLMs still obtain the lead position overall, where Mamba-2$_{code}$ retains the best performance across all six PLMs. In the code completion task, Mamba-2$_{code}$ outperforms both CodeGPT variants by 7.21%–26.55% in terms of EM and by 4.19%–11.44% in terms of ES on JavaCorpus, while outperforming them by 8.78%–22.67% and 3.12%–8.43% in terms of each evaluation metric in order on PY150. Towards the code generation task, Mamba-2$_{code}$ surpasses two CodeGPT variants by 12.59%–48.06% in terms of EM, by 5.1%–14.58% in terms of BLEU, and by 4.62%–11.35% in terms of CodeBLEU. Regarding the code clone detection task, Mamba-2$_{code}$ keeps excelling by 9.2%–9.81% in terms of F1 score. Vertically compare the performance gain of Mamba families over CodeGPT variants under FT and PEFT settings from RQ1-RQ2, we notice that PLMs of Mamba families obtain more advantages from fine-tuning with PEFT methods. The reason is that Mamba-based PLMs use a customized CUDA kernel that stores all time-varying SSM parameters, leading to stronger regularization and generalization ability than Transformer-based counterparts [62].

Finding 2: In the PEFT setting, although Mamba-based PLMs fall short of that achieved through FT, their performance gain over CodeGPT variants is more significant, highlighting their practical value in limited computational resources.

C. *RQ3: Advantages of Mamba Architecture*

Although Mamba-based PLMs show superior performance compared to Transformer-based ones in RQ1 and RQ2, the

TABLE III
MODEL PERFORMANCE WHEN PRE-TRAINED FROM SCRATCH. THE MAMBA AND MAMBA-2 USED HERE ARE 130M-SIZED.

Method	**Models**	**JavaCorpus**		**PY150**		**CONCODE**			**BigCloneBench**
		EM	ES	EM	ES	EM	BLEU	CodeBLEU	F1
FT	CodeGPT$_{scratch}$	12.11$_{\pm 0.97}$	46.09$_{\pm 1.49}$	24.02$_{\pm 0.60}$	59.94$_{\pm 0.59}$	5.48$_{\pm 0.39}$	22.16$_{\pm 0.85}$	24.25$_{\pm 0.96}$	0.897$_{\pm 0.010}$
	Mamba$_{scratch}$	12.99$_{\pm 0.14}$	48.83$_{\pm 0.09}$	22.75$_{\pm 0.31}$	59.83$_{\pm 0.16}$	6.42$_{\pm 0.36}$	25.44$_{\pm 0.40}$	27.58$_{\pm 0.41}$	0.917$_{\pm 0.004}$
	Mamba-2$_{scratch}$	**18.20**$_{\pm 0.32}$	**54.78**$_{\pm 0.50}$	**29.21**$_{\pm 0.35}$	**64.43**$_{\pm 0.21}$	**9.83**$_{\pm 0.23}$	**28.02**$_{\pm 0.51}$	**29.53**$_{\pm 0.69}$	**0.930**$_{\pm 0.006}$
LoRA	CodeGPT$_{scratch}$	12.89$_{\pm 0.10}$	47.15$_{\pm 0.42}$	19.00$_{\pm 0.28}$	54.20$_{\pm 0.46}$	5.32$_{\pm 0.36}$	20.34$_{\pm 0.24}$	22.04$_{\pm 0.35}$	0.848$_{\pm 0.004}$
	Mamba$_{scratch}$	11.67$_{\pm 0.58}$	47.06$_{\pm 0.68}$	18.25$_{\pm 0.10}$	55.16$_{\pm 0.08}$	5.42$_{\pm 0.29}$	22.04$_{\pm 0.14}$	23.83$_{\pm 0.28}$	0.898$_{\pm 0.011}$
	Mamba-2$_{scratch}$	**16.12**$_{\pm 0.21}$	**52.72**$_{\pm 0.07}$	**25.20**$_{\pm 0.13}$	**61.41**$_{\pm 0.20}$	**9.08**$_{\pm 0.32}$	**26.38**$_{\pm 0.14}$	**27.93**$_{\pm 0.49}$	**0.920**$_{\pm 0.006}$
$(IA)^3$	CodeGPT$_{scratch}$	8.40$_{\pm 0.09}$	36.41$_{\pm 0.44}$	7.31$_{\pm 0.15}$	38.94$_{\pm 0.15}$	1.70$_{\pm 0.20}$	16.23$_{\pm 0.73}$	17.54$_{\pm 0.76}$	0.474$_{\pm 0.003}$
	Mamba$_{scratch}$	8.89$_{\pm 0.10}$	42.07$_{\pm 0.20}$	9.06$_{\pm 0.06}$	43.06$_{\pm 0.28}$	1.25$_{\pm 0.05}$	12.51$_{\pm 0.49}$	14.15$_{\pm 0.18}$	0.494$_{\pm 0.015}$
	Mamba-2$_{scratch}$	**11.76**$_{\pm 0.15}$	**47.00**$_{\pm 0.29}$	**13.63**$_{\pm 0.11}$	**50.20**$_{\pm 0.12}$	**4.83**$_{\pm 0.13}$	**18.04**$_{\pm 0.03}$	**19.42**$_{\pm 0.08}$	**0.723**$_{\pm 0.007}$

* Table cells with a light green background indicate statistically significant improvements over CodeGPT$_{scratch}$.

discrepancy in their pre-training databases makes it challenging to attribute the gains solely to the Mamba architecture. To overcome this limitation, we pre-train CodeGPT, Mamba, and Mamba-2 from scratch using the same pre-training corpora as RQ1, and then evaluate their performance across all code tasks studied in this work. As shown in Table III, Mamba-2$_{scratch}$ consistently achieves the best performance across diverse tuning methods, with CodeGPT$_{scratch}$ and Mamba$_{scratch}$ yielding similar results. Specifically, in the code completion task, Mamba-2$_{scratch}$ outperforms CodeGPT$_{scratch}$ by 25.06%–50.29% in terms of EM and by 11.81%–29.09% in terms of ES on JavaCorpus regardless of FT or PEFT. As for PY150, it achieves a more substantial advantage by 21.61%–86.46% on the EM metric. Concerning the code generation task, Mamba-2$_{scratch}$ surpasses CodeGPT$_{scratch}$ by 70.68%–184.12% in terms of EM, by 11.15%–29.70% in terms of BLEU, and by 10.72%–26.72% in terms of CodeBLEU. Regarding the code clone detection task, Mamba-2$_{scratch}$ maintains its superiority by 3.68%–52.53% in terms of F1 score. By combining the strengths of selective SSMs and the attention mechanism, along with an additional normalization layer, the Mamba-2 block offers advantages in modeling long sequences such as code snippets, allowing it to outperform Transformer-based PLMs with similar parameter size [63]. It is worth noting that, although Mamba-2 only introduces minor architectural modifications compared to Mamba, there is a significant difference in their performance on code tasks. Across the results of this RQ and the previous two, Mamba-2 consistently demonstrates superior capability over Mamba in code modeling.

In addition to evaluating their performance, we further evaluate their efficiency. In the JavaCorpus dataset, we compare the training time and memory consumption of each PLM, focusing on their computational resource utilization. The results are illustrated in Figure 2. We report the average training time of each epoch and the total occupied memory size. Since the resource consumption across multiple repeated experiments was nearly consistent, we do not report the variance here. It can be observed that: (i) No matter CodeGPT$_{scratch}$, Mamba$_{scratch}$, or Mamba-2$_{scratch}$, their resource consumption under PEFT methods is consistently lower than that under FT. (ii) Mamba-based models require more training time but consume less memory. The first observation is expected, as PEFT methods involve only a small portion of trainable parameters, indicating that the efficiency of PEFT methods benefits PLMs of both categories. Regarding the second observation, the increased

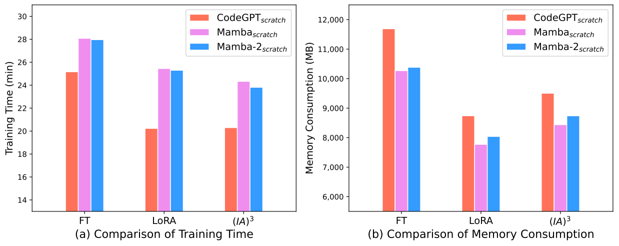


Fig. 2. Utilization of computational resources.

training time can be attributed to the complex computation process of selective SSMs, while the reduced memory consumption is due to their linear scaling with sequence length. Unlike the attention mechanism that requires storing quadratic attention weights, Mamba-based PLMs maintain a compact state representation that grows linearly. Since memory constraints are typically more limiting than computational time, the results highlight the potential for Mamba-based PLMs to be applied more broadly in resource-constrained environments.

Finding 3: After uniform pre-training from scratch, Mamba-2$_{scratch}$ achieves the best performance, demonstrating the advantage of the Mamba-2 block in code modeling. Moreover, Mamba-based models also show superior memory efficiency.

D. RQ4: Evaluation in the Low-Resource Scenario

This section evaluates two Mamba PLMs (i.e., Mamba$_{code}$ and Mamba-2$_{code}$) and two CodeGPT variants (i.e., CodeGPT and CodeGPT-adapted) under both FT and PEFT methods across all studied code tasks in low-resource scenarios. As illustrated in Figure 3, overall Mamba-based PLMs consistently keep the best performance across all data sampling rates, code tasks, and tuning methods. In particular, they preserve their relatively high performance on JavaCorpus in the code completion task. When the training data is reduced to 30%, Mamba$_{code}$ and Mamba-2$_{code}$ exhibit slight performance degradation with an average drop of 0.51% and 0.64% in terms of ES across diverse tuning methods. Even with as little as 0.1% of the full training data, their performance declines by only 5.09% and 3.96% on ES, respectively. We attribute this advantage to its selective SSMs mechanisms, which dynamically choose how to propagate or forget information based on the current input token, thereby learning more efficiently even when the training data is limited.

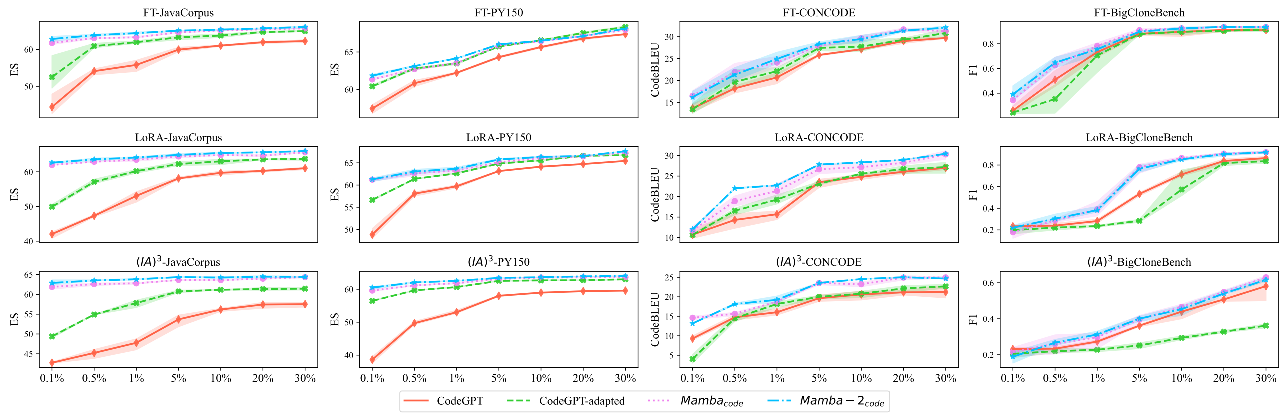


Fig. 3. Evaluation results in the low-resource scenario.

As for comparison between CodeGPT variants and Mamba-based PLMs, we find that the significance of the discrepancies is heterogeneous across diverse code tasks. For example, in the code completion task, the performance advantage of Mamba-based PLMs becomes more pronounced as the training data reduces. This phenomenon occurs on both JavaCorpus and PY150, regardless of the tuning method of FT or PEFT. As for the code generation task, the performance advantage of Mamba-based PLMs is almost consistent regardless of the changes of the training volume and tuning methods. Nonetheless, for the code clone detection task, the situation is completely different. Firstly, Mamba-based PLMs widened their lead over CodeGPT variants as the amount of training data increased when tuning with $(IA)^3$. In contrast, when it comes to LoRA, Mamba-based PLMs manifest a relatively prominent superiority when the training data is reduced to a medium position (around 5%-10%). In addition, once PLMs are tuned with FT, their performance discrepancies return to a stable situation. But all in all, although the performance of both Mamba and CodeGPT PLMs declines, the decrease of Mamba-based PLMs is relatively mild, showing a greater robustness when only limited training data is available.

Finding 4: Compared to Transformer-based PLMs, Mamba-based ones demonstrate superiority in the low-resource scenario, owing to the efficient learning ability derived from Mamba's selective SSM mechanism. Moreover, the performance advantages are heterogeneous across code tasks and tuning methods.

E. RQ5: Evaluation with Larger Scale

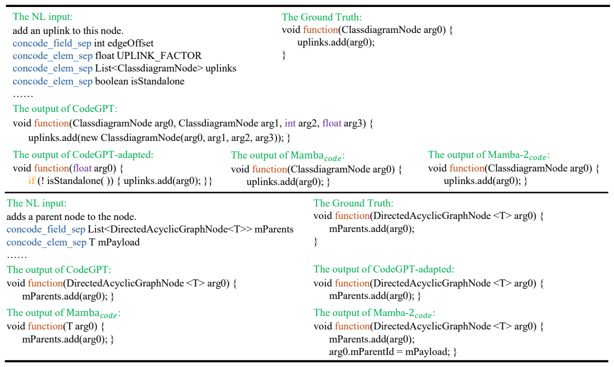
The experiments in the preceding RQs were conducted on the CodeGPT with 125M parameters, as well as Mamba and Mamba-2 with 130M parameters. In this section, we evaluate the performance of Mamba-based PLMs when their parameter sizes are scaled up to 370M or even 7B, and make comparisons with similarly sized Transformer-based models across all code tasks, where FT is adopted for PLMs while zero-shot learning is used for LLMs. Experimental results are shown in Table IV. As can be seen, scaling Mamba-based PLMs from 130M to 370M leads to a significant performance lift across almost all code tasks. However, compared with recent PLMs, such as CodeGen, PolyCoder, and Qwen2.5-Coder, our pre-trained Mamba-based PLMs perform slightly lackluster in the code completion task. Specifically, the best-performing model, i.e., CodeGen, outperforms Mamba-based models by 3.49%–8.17% in terms of EM and by 1.48%–2.57% in terms of ES on JavaCorpus. As for PY150, CodeGen still keeps its advantages by 7.06%–10.49% and 2.87%–3.86% in terms of each metric in order. A potential explanation is that recent PLMs were pre-trained with relatively larger training bases on code, making them more powerful on coding tasks. For example, CodeGen was pre-trained with 150.8B code tokens covering six PLs, which significantly surpasses Mamba-based PLMs pre-training on either the Java or Python portion of CodeSearchNet with at most 0.08B tokens. Surprisingly, in the code generation and code clone detection tasks, they instead manifest superiority against other Transformer-based models overall. Specifically, the best-performing Mamba-based model (Mamba-2_{code}) performs better than Transformer-based counterparts by 1.56%–5.16% in terms of EM, by 2.6%–10.79% in terms of BLEU, and 3%–10.45% in terms of CodeBLEU on CONCODE. Regarding the BigCloneBench, the best-performing Mamba-based PLM (Mamba) excels them by 1.27%–2.68% in terms of F1 score. For Mamba-based models that outperform all three Transformer-based models, we conduct WSRT to examine the statistical significance. According to the WSRT shown in Table IV, most outperforming Mamba-based models also demonstrate statistical significance in performance differences. A potential explanation is that recent Transformer-based models have been pre-trained on massive code samples with completion-analogous tasks, such as autoregression modeling, making their fine-tuning more conflicted with new task paradigms such as NL-guided code generation and binary-predictive code clone detection.

Furthermore, we also extend parameter sizes to 7B on the Large Language Model (LLM) level to evaluate the code generation and comprehension abilities of Transformer- and Mamba-based models. As can be seen from Table IV, Codestral Mamba lags behind Transformer-based models a lot, by 36.06%–45.93% in terms of EM and by 13.03%–19.12%

TABLE IV
EVALUATION RESULTS OF LARGER-SCALE MODELS USING FT DURING ADAPTATION.

Methods	Models	Params	JavaCorpus		PY150		CONCODE			BigCloneBench
			EM	ES	EM	ES	EM	BLEU	CodeBLEU	F1
FT	CodeGen	350M	**36.16**$_{\pm 0.38}$	**70.72**$_{\pm 0.33}$	**42.13**$_{\pm 0.09}$	**73.74**$_{\pm 0.07}$	20.15$_{\pm 0.22}$	35.34$_{\pm 0.24}$	35.49$_{\pm 0.17}$	0.927$_{\pm 0.006}$
	PolyCoder	400M	35.80$_{\pm 0.46}$	70.35$_{\pm 0.49}$	40.01$_{\pm 0.21}$	71.74$_{\pm 0.14}$	20.45$_{\pm 0.20}$	35.73$_{\pm 0.24}$	36.02$_{\pm 0.39}$	0.933$_{\pm 0.004}$
	Qwen2.5-Coder	500M	34.96$_{\pm 0.16}$	69.66$_{\pm 0.37}$	41.84$_{\pm 0.50}$	73.33$_{\pm 0.37}$	19.75$_{\pm 0.13}$	33.09$_{\pm 0.57}$	33.59$_{\pm 0.41}$	0.921$_{\pm 0.016}$
	Mamba	370M	33.43$_{\pm 0.01}$	68.95$_{\pm 0.06}$	38.13$_{\pm 0.13}$	71.00$_{\pm 0.06}$	20.75$_{\pm 0.09}$	36.40$_{\pm 0.70}$	36.85$_{\pm 0.69}$	🟩 **0.945**$_{\pm 0.004}$
	Mamba-2	370M	34.12$_{\pm 0.38}$	69.08$_{\pm 0.38}$	38.64$_{\pm 0.21}$	71.04$_{\pm 0.16}$	20.37$_{\pm 0.10}$	35.09$_{\pm 0.55}$	35.48$_{\pm 0.56}$	🟩 0.942$_{\pm 0.002}$
	Mamba$_{code}$	370M	34.01$_{\pm 0.07}$	69.26$_{\pm 0.12}$	38.58$_{\pm 0.19}$	71.36$_{\pm 0.12}$	19.59$_{\pm 0.13}$	34.53$_{\pm 0.05}$	34.98$_{\pm 0.09}$	🟩 0.941$_{\pm 0.001}$
	Mamba-2$_{code}$	370M	34.94$_{\pm 0.10}$	69.69$_{\pm 0.22}$	39.35$_{\pm 0.18}$	71.68$_{\pm 0.19}$	🟩 **20.77**$_{\pm 0.11}$	🟩 **36.66**$_{\pm 0.30}$	🟩 **37.10**$_{\pm 0.59}$	🟩 0.944$_{\pm 0.004}$
Zero-Shot	CodeGen	6B	31.61$_{\pm 0.64}$	67.32$_{\pm 0.16}$	22.95$_{\pm 0.18}$	57.21$_{\pm 0.26}$	0.00$_{\pm 0.00}$	0.13$_{\pm 0.02}$	10.04$_{\pm 0.34}$	0.000$_{\pm 0.000}$
	Qwen2.5-Coder	7B	**37.38**$_{\pm 0.27}$	**72.39**$_{\pm 0.38}$	**28.53**$_{\pm 0.17}$	**56.83**$_{\pm 0.10}$	0.00$_{\pm 0.00}$	**0.19**$_{\pm 0.04}$	**13.52**$_{\pm 0.32}$	0.000$_{\pm 0.000}$
	Codestral Mamba	7B	20.21$_{\pm 0.20}$	58.55$_{\pm 0.85}$	18.66$_{\pm 0.45}$	50.46$_{\pm 0.72}$	0.00$_{\pm 0.00}$	0.13$_{\pm 0.03}$	12.25$_{\pm 0.32}$	🟩 **0.219**$_{\pm 0.000}$

* Table cells with a light green background indicate statistically significant improvements over all Transformer-based counterparts.



The NL input:
add an uplink to this node.
concode_field_sep int edgeOffset
concode_elem_sep float UPLINK_FACTOR
concode_elem_sep List<ClassdiagramNode> uplinks
concode_elem_sep boolean isStandalone
……

The Ground Truth:
```
void function(ClassdiagramNode arg0) {
    uplinks.add(arg0);
}
```

The output of CodeGPT:
```
void function(ClassdiagramNode arg0, ClassdiagramNode arg1, int arg2, float arg3) {
    uplinks.add(new ClassdiagramNode(arg0, arg1, arg2, arg3)); }
```

The output of CodeGPT-adapted:
```
void function(float arg0) {
    if (! isStandalone( )) { uplinks.add(arg0); }}
```

The output of Mamba$_{code}$:
```
void function(ClassdiagramNode arg0) {
    uplinks.add(arg0); }
```

The output of Mamba-2$_{code}$:
```
void function(ClassdiagramNode arg0) {
    uplinks.add(arg0); }
```

The NL input:
adds a parent node to the node.
concode_field_sep List<DirectedAcyclicGraphNode<T>> mParents
concode_elem_sep T mPayload
……

The Ground Truth:
```
void function(DirectedAcyclicGraphNode <T> arg0) {
    mParents.add(arg0);
}
```

The output of CodeGPT:
```
void function(DirectedAcyclicGraphNode <T> arg0) {
    mParents.add(arg0); }
```

The output of CodeGPT-adapted:
```
void function(DirectedAcyclicGraphNode <T> arg0) {
    mParents.add(arg0); }
```

The output of Mamba$_{code}$:
```
void function(T arg0) {
    mParents.add(arg0); }
```

The output of Mamba-2$_{code}$:
```
void function(DirectedAcyclicGraphNode <T> arg0) {
    mParents.add(arg0);
    arg0.mParentId = mPayload; }
```

Fig. 4. Case study in the CONCODE dataset of the code generation task.

in terms of ES on JavaCorpus. This lagging performance trend continues extending to PY150, by 18.69%–34.6% and 11.21%–11.8% in terms of each metric in order. Similar results are also shown in the code generation task, although Codestral Mamba outperforms CodeGen, it still lags behind Qwen2.5-Coder by 31.58% and 9.39% in terms of BLEU and CodeBLEU. As Mistral AI [50] has not disclosed any technical report for Codestral Mamba, it is hard to infer the reason for their performance discrepancies. But as for the code clone detection task, Codestral Mamba demonstrates much better performance against Transformer-based counterparts, where none of the latter's generated contents are useful for detecting cloned code. The reason is that both CodeGen and Qwen2.5-Coder are completion LLMs without instruction tuning that Codestral Mamba has equipped [28], leading to their inability to understand new task paradigms other than generating code. The above experimental results demonstrate the importance of pre-training tasks and databases on the model performance, apart from the model architectures mentioned in RQ3.

Finding 5: Scaling up Mamba-based PLMs' sizes indeed obtains performance gains, but compared with other counterparts, pre-training databases and tasks also heavily affect the code modeling performance, apart from model architectures. This is the same case in the Mamba-based LLM and its counterparts.

V. DISCUSSION

A. Case Study

To thoroughly evaluate the performance of Mamba-based models, we conduct a case study in the CONCODE dataset of the code generation task. As shown in Figure 4, we present two illustrative instances, one correctly addressed by Mamba-based models and the other by Transformer-based models. In each instance, the input is formulated as a natural language description that specifies the objective of the desired code program, with additional variables separated by pre-defined special tokens. The ground truth is also provided.

The first illustrative case reveals that Mamba-based models are able to generate outputs fully aligned with the ground truth, while CodeGPT and CodeGPT-adapted produce some unrelated tokens. The code generated by CodeGPT exhibits lower quality, with excessive formal parameters and redundant logic. Despite the presence of an existing ClassdiagramNode instance "arg", the model still creates a new instance via "new ClassdiagramNode". The output generated by CodeGPT-adapted contains two notable errors: an incorrect type assignment for "arg0", and the inclusion of an extraneous "if" condition. It can be observed that the extra tokens they generate come from the variables in the input, such as "isStandalone", suggesting a limited ability to fully comprehend the input during the decoding stage. In the second instance, where Mamba-based models show inferior performance, although their outputs are not identical to the ground truth, the deviation is not substantial. Mamba fails to accurately output the complete parameter type for "arg0", while Mamba-2 erroneously adds an extra assignment statement for "mParentId". The main logic of the objective, adding a parent node, has been successfully implemented. In contrast to the poor-quality output from CodeGPT in the first instance, this demonstrates the effectiveness of Mamba-based models in code modeling.

In addition, we conduct an analysis of statistical characteristics of the inputs and outputs. Among the instances correctly answered by different code LMs, we observe that Mamba-2$_{code}$ is capable of handling longer input sequences. The longest input sequence accurately handled by Mamba-2 contains 333 tokens, while the other three models could only handle up to 249 tokens. This finding highlights the effectiveness of the Mamba architecture in processing long sequences. Unlike the attention module, which has quadratic complexity, Mamba employs a linear SSM that scales linearly with sequence length, thereby facilitating more efficient and scalable long-sequence modeling. We also compute the average output length of different code LMs. Similarly, Mamba-2$_{code}$ exhibits performance that is closer to the ground truth.

The ground truth has an average token number of 38.70, while for CodeGPT, CodeGPT-adapted, Mamba$_{code}$, and Mamba-2$_{code}$, they are 33.02, 28.22, 33.39, and 35.04, respectively. The results indicate that CodeGPT-adapted tends to generate shorter outputs, whereas Mamba-2$_{code}$ produces outputs that are more closely aligned with the ground truth.

B. Human Evaluation

To investigate whether the statistically significant outperforming also implies an obvious usage difference for practitioners, we conduct a human evaluation to measure programs generated by 130M-sized Mamba-2$_{code}$ and its Transformer-based counterparts based on experimental results of RQ1-2. Programs are manually evaluated along two dimensions: Correctness (whether the code fulfills the specified requirement) and Maintainability (whether the implementation adheres to coding standards and exhibits good readability). Each dimension is scored on an integer scale from 0 to 2, with 0 indicating poor quality and 2 indicating high quality. We evaluate across four distinct groups, formed by combining two datasets, JavaCorpus and CONCODE, with two fine-tuning strategies, FT and LoRA. For each group, we randomly select 10 samples per model, resulting in 30 samples per group. In total, we obtain 120 samples for evaluation.

We recruit 8 evaluators with 3–5 years of development experience, assigning two people to each group. The final score is the average of the two evaluators' ratings. As shown in Table V, Mamba-2$_{code}$ consistently achieves the highest scores across all groups. Furthermore, we compute the Cohen's Kappa coefficient for all evaluated items and find it consistently exceeds 0.6, indicating substantial agreement between evaluators and demonstrating the practical value of Mamba-based PLMs.

TABLE V
MODEL PERFORMANCE IN HUMAN EVALUATION.

Dataset	Method	Metric	CodeGPT	CodeGPT-A	Mamba-2$_c$	Kappa Score
JavaCorpus	FT	Corr.	0.55	0.75	**1.70**	0.682
		Maint.	1.05	1.35	**1.90**	0.884
	LoRA	Corr.	0.50	0.50	**1.60**	0.750
		Maint.	0.45	0.60	**1.65**	0.686
CONCODE	FT	Corr.	0.80	1.00	**1.80**	0.631
		Maint.	0.90	1.00	**1.80**	0.718
	LoRA	Corr.	0.40	0.50	**1.65**	0.749
		Maint.	0.60	0.70	**1.55**	0.714

* "Corr." and "Maint." are abbreviations for "Correctness" and "Maintainability". "CodeGPT-A" stands for "CodeGPT-adapted". "Mamba-2$_c$" denotes "Mamba-2$_{code}$".

C. Threats to validity

External Validity One threat to the external validity stems from the scale of the Mamba-based models. Given that our experiments span three different model sizes, i.e., 130M, 370M, and 7B parameters, covering both PLMs and LLMs, the results exhibit generalizability, rendering this limitation only a minor threat to validity. Another potential threat arises from the limited selection of code tasks and associated datasets. To address this concern, we selected three representative code tasks, covering both generation and comprehension categories, along with their widely adopted datasets. However, beyond the tasks considered in this work, numerous additional tasks, such as code translation and program repair, are also applicable to Mamba-based models. We could add more tasks in future work. As for PLMs under comparison with Mamba, only Transformer-based ones are selected because they are the most prevalent and mainstream LMs to date [31], [35], [36], [39]. Although we also noticed other kinds of neural networks, such as Spiking neural network [64] and Generative Adversarial Networks [65], they have not reached even a neck-to-neck performance with Transformer-based models in any mainstream research area. Therefore, selecting Transformer Code LMs for comparison can make our study more meaningful, representative, and practical. Thus, for models of other structures, we leave them for future work.

Internal Validity One potential threat lies in the hyperparameter settings, encompassing both those employed in PLMs and those applied in the adaptation process to downstream tasks. To mitigate this issue, all PLMs used in this work are obtained from Huggingface with their default configurations. We further tuned the learning rate of each model within a reasonable range to obtain the best results. Despite these efforts, it is still uncertain whether the optimal hyperparameter settings have been identified, and further assessment is still needed. In addition, internal validity is also influenced by the choice of target modules for PEFT methods within the Mamba architecture. Due to the absence of a widely accepted consensus on which components are most suitable for PEFT, we apply them specifically to the linear projection matrices, inspired by the prior work [23]. Whether targeting alternative modules could yield better performance remains an open question. Nonetheless, our results can still suggest the effectiveness of Mamba-based models in the PEFT setting.

D. Implications

Implications for developers. This study reveals several advantages of Mamba-based models, especially under PEFT settings and in low-resource scenarios, demonstrating that Mamba-based models offer a viable and effective solution in these scenarios. Moreover, the memory efficiency improvements achieved by Mamba underscore its enhanced feasibility for practical deployment. However, Mamba-based models also come with some limitations. First, they typically require longer training times compared to their Transformer-based counterparts. Second, large-scale Mamba models remain scarce, limiting their applicability in tasks demanding high capacity. Given these trade-offs, we recommend that developers carefully evaluate their specific use scenarios and choose a model with an appropriate architecture.

Implications for researchers. Being the first systematic study to evaluate Mamba in code intelligence tasks, this work uncovers several promising findings. Nevertheless, several open problems persist that require further investigation and may guide new directions for future research. First, Mamba-based code PLMs remain scarce. Unlike Transformer-based code PLMs, which have been extensively developed across diverse datasets and pre-training tasks, the in-depth advancement of Mamba-based code PLMs still requires greater attention from the research community. Besides, our findings indicate that pre-training databases and tasks affect the code modeling performance. Since the original Mamba models were not pre-trained on code corpora, they underperform relative to state-of-the-art Transformer-based models in some cases. This observation contrasts with the consistently positive gains Mamba has demonstrated in NLP and CV domains [8], [11], [63]. Although Codestral Mamba is specifically designed for code intelligence, it also underperforms in some cases. We advocate for more research on pre-training Mamba-based code PLMs. Moreover, it is worth exploring whether Mamba possesses superior capabilities in capturing various code properties, such as lexical, syntactic, and structural properties. Such exploration merits further investigation in the context of Mamba-based models.

VI. Related Work

A. Transformer-based Code Language Models

Recently, by virtue of their effectiveness and versatility, code language models are widely utilized in code intelligence tasks and substantially improve the performance. They are overwhelmingly based on the Transformer architecture. CodeBERT [35], CodeGPT [15], and CodeT5 [36] are three pioneering works, and they just correspond to three different architectures: Encoder-only, Decoder-only, and Encoder-Decoder [37], [38]. Appeared after them, UniXcoder [39] leverages cross-modal contents like ASTs and code comments to enhance code representation and unifies the architecture of the encoder and decoder, enabling it to support both understanding and generation tasks. CodeGen [29] contains a family of conversational code LMs. It utilizes three-stage training to produce three distinct models, trained sequentially on a natural language dataset, a multi-lingual dataset, and a Python-only dataset. PolyCoder [30] is based on the GPT-2 [48] architecture and is pre-trained on 249GB of code across 12 programming languages, sizing up to 2.7B parameters. In addition, several emerging code LMs like Deepseek-Coder [66] and Qwen2.5-Coder [31] have also achieved promising performance.

With the continuous increase in the scale of code LMs, directly FT code LMs becomes computationally expensive and memory-intensive. Therefore, researchers resort to Parameter-Efficient Fine-Tuning (PEFT) to seek a balance between effectiveness and efficiency. Wang et al. [67] investigate the performance of adapter tuning [68] for code search and code summarization, and experimentally show its advantages in low-resource scenarios. Liu et al. [25] cover four well-known PEFT methods and conduct a comprehensive empirical study across multiple scenarios. This work [24] delves into the effectiveness of adapter tuning and LoRA in code-change-related tasks, employing probing tasks to explain the efficacy of PEFT methods. Li et al. [69] concentrate on the automated program repair task, and they find that $(IA)^3$ [18] improves the creativity of code LMs more effectively. In this paper, we evaluate various models under both FT and PEFT settings.

B. Mamba and Mamba-based Models

Although the Transformer [20] has ushered in the era of LLM, its core attention layer suffers from the drawback of computational inefficiency because it scales quadratically with sequence length [6]. Recently, Mamba [8] has emerged as a powerful alternative to the Transformer and thus attracted the attention of researchers. It leverages selective State Space Models (SSMs) to obtain the context-aware ability, dynamically parameterizes its components based on input, and designs a hardware-aware algorithm to make the training of selective SSMs efficient on modern hardware (GPUs). Mamba has shown impressive performance in a wide range of domains [10]–[12]. As a subsequent follow-up, Mamba-2 [9] is inspired by the connection between SSMs and the Transformer, refining the architecture of Mamba and demonstrating better performance. Within the domain of code LMs, Codestral Mamba [28] stands as the only mamba-based LLM to date that is specifically optimized for code. Islam et al. [13] explore the running efficiency of its generated code, but without any correctness analysis. Iznaga [14] leverages Codestral Mamba and LoRA for exclusively automated software testing. Therefore, Mamba-related code intelligence studies are rare to a great extent. To fill this gap, we systematically explore three code tasks, covering both code comprehension and generation categories, with diverse tuning methods, i.e., FT, LoRA, and $(IA)^3$, practical scenarios, and key aspects.

VII. Conclusion

This paper presents a systematic empirical study of Mamba-based models across three representative code tasks covering both code comprehension and generation categories. Our experimental results demonstrate that Mamba-based PLMs perform either better or comparable to similarly sized Transformer-based ones with FT. In the PEFT setting, Mamba-based PLMs demonstrate more substantial advantages. Afterwards, we also empirically testify to the superiority of Mamba architecture on code and its efficient learning ability in low-resource scenarios. Besides, both Mamba and Transformer-based models are further investigated on large scales to 370M and 7B. In the end, we also present a qualitative case study and a human evaluation, followed by a discussion of the broader implications of this work. As a systematic empirical study of Mamba-based models, this work sheds light on their pros/cons and application prospects.

Acknowledgment

This work is supported in part by the General Research Fund of the Research Grants Council of Hong Kong and the research funds of the City University of Hong Kong (Grant No. 6000796, 9229109, 9229098, 9220103, 9229029), also by the National Natural Science Foundation of China (Grant Nos. U24B20149 and 62502283), the Natural Science Foundation of Shandong Province (Grant No. ZR2024QF093), the Young Talent of Lifting Engineering for Science and Technology in Shandong, China.

References

[1] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–79, 2024.

[2] M. Izadi, J. Katzy, T. Van Dam, M. Otten, R. M. Popescu, and A. Van Deursen, "Language models for code completion: A practical evaluation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[3] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *arXiv preprint arXiv:2406.00515*, 2024.

[4] X. Yu, L. Liu, X. Hu, J. W. Keung, J. Liu, and X. Xia, "Fight fire with fire: How much can we trust chatgpt on source code-related tasks?" *IEEE Transactions on Software Engineering*, 2024.

[5] F. Liu, Z. Fu, G. Li, Z. Jin, H. Liu, Y. Hao, and L. Zhang, "Non-autoregressive line-level code completion," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 5, pp. 1–34, 2024.

[6] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, "Transformers are rnns: Fast autoregressive transformers with linear attention," in *International conference on machine learning*. PMLR, 2020, pp. 5156–5165.

[7] Y. Tay, M. Dehghani, D. Bahri, and D. Metzler, "Efficient transformers: A survey," *ACM Computing Surveys*, vol. 55, no. 6, pp. 1–28, 2022.

[8] A. Gu and T. Dao, "Mamba: Linear-time sequence modeling with selective state spaces," in *First Conference on Language Modeling*, 2024.

[9] T. Dao and A. Gu, "Transformers are ssms: generalized models and efficient algorithms through structured state space duality," in *Proceedings of the 41st International Conference on Machine Learning*, 2024, pp. 10 041–10 071.

[10] S. Azizi, S. Kundu, M. E. Sadeghi, and M. Pedram, "Mambaextend: A training-free approach to improve long context extension of mamba," in *The Thirteenth International Conference on Learning Representations*, 2025.

[11] L. Zhu, B. Liao, Q. Zhang, X. Wang, W. Liu, and X. Wang, "Vision mamba: Efficient visual representation learning with bidirectional state space model," in *Forty-first International Conference on Machine Learning*, 2024.

[12] A. Behrouz and F. Hashemi, "Graph mamba: Towards learning on graphs with state space models," in *Proceedings of the 30th ACM SIGKDD conference on knowledge discovery and data mining*, 2024, pp. 119–130.

[13] M. A. Islam, D. V. Jonnala, R. Rekhi, P. Pokharel, S. Cilamkoti, A. Imran, T. Kosar, and B. Turkkan, "Evaluating the energy-efficiency of the code generated by llms," *arXiv preprint arXiv:2505.20324*, 2025.

[14] Y. S. Iznaga, "Enhancing software testing automation through large language models," Master's thesis, Universidade de Évora, 2025.

[15] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021.

[16] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[17] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen *et al.*, "Lora: Low-rank adaptation of large language models." *ICLR*, 2022.

[18] H. Liu, D. Tam, M. Muqeeth, J. Mohta, T. Huang, M. Bansal, and C. A. Raffel, "Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 1950–1965, 2022.

[19] S. Liu, J. Keung, Z. Yang, Z. Mao, and Y. Sun, "Can mamba be better? an experimental evaluation of mamba in code intelligence," 9 2025. [Online]. Available: https://github.com/ishuoliu/ASE-CMamba

[20] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[21] A. Gu, I. Johnson, K. Goel, K. Saab, T. Dao, A. Rudra, and C. Ré, "Combining recurrent, convolutional, and continuous-time models with linear state space layers," *Advances in neural information processing systems*, vol. 34, pp. 572–585, 2021.

[22] Z. Xu, Y. Yue, X. Hu, D. Yang, Z. Yuan, Z. Jiang, Z. Chen, JiangyongYu, XUCHEN, and S. Zhou, "Mambaquant: Quantizing the mamba family with variance aligned rotation methods," in *The Thirteenth International Conference on Learning Representations*, 2025.

[23] M. Yoshimura, T. Hayashi, and Y. Maeda, "MambaPEFT: Exploring parameter-efficient fine-tuning for mamba," in *The Thirteenth International Conference on Learning Representations*, 2025.

[24] S. Liu, J. Keung, Z. Yang, F. Liu, Q. Zhou, and Y. Liao, "Delving into parameter-efficient fine-tuning in code change learning: An empirical study," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2024, pp. 465–476.

[25] J. Liu, C. Sha, and X. Peng, "An empirical study of parameter-efficient fine-tuning methods for pre-trained code models," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 397–408.

[26] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence," in *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering*, 2022, pp. 382–394.

[27] T. Henighan, J. Kaplan, M. Katz, M. Chen, C. Hesse, J. Jackson, H. Jun, T. B. Brown, P. Dhariwal, S. Gray *et al.*, "Scaling laws for autoregressive generative modeling," *arXiv preprint arXiv:2010.14701*, 2020.

[28] "Codestral mamba - mistral ai," https://mistral.ai/news/codestral-mamba.

[29] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," in *The Eleventh International Conference on Learning Representations*, 2023.

[30] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN international symposium on machine programming*, 2022, pp. 1–10.

[31] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu *et al.*, "Qwen2. 5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[32] Z. Yang, F. Liu, Z. Yu, J. W. Keung, J. Li, S. Liu, Y. Hong, X. Ma, Z. Jin, and G. Li, "Exploring and unleashing the power of large language models in automated code translation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1585–1608, 2024.

[33] P. Xue, L. Wu, Z. Yu, Z. Jin, Z. Yang, X. Li, Z. Yang, and Y. Tan, "Automated commit message generation with large language models: An empirical study and beyond," *IEEE Transactions on Software Engineering*, 2024.

[34] J. Li, Y. Zhao, Y. Li, G. Li, and Z. Jin, "Acecoder: An effective prompting technique specialized in code generation," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–26, 2024.

[35] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[36] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708.

[37] C. Niu, C. Li, V. Ng, D. Chen, J. Ge, and B. Luo, "An empirical comparison of pre-trained models of source code," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2136–2148.

[38] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, 2022, pp. 39–51.

[39] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2022, pp. 7212–7225.

[40] D. Guo, C. Xu, N. Duan, J. Yin, and J. McAuley, "Longcoder: A long-range pre-trained language model for code completion," in *International Conference on Machine Learning*. PMLR, 2023, pp. 12 098–12 107.

[41] S. Lu, N. Duan, H. Han, D. Guo, S. W. Hwang, and A. Svyatkovskiy, "Reacc: A retrieval-augmented code completion framework," in *60th Annual Meeting of the Association for Computational Linguistics, ACL*, 2022, pp. 6227–6240.

[42] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *2013 10th working conference on mining software repositories (MSR)*. IEEE, 2013, pp. 207–216.

[43] V. Raychev, P. Bielik, and M. Vechev, "Probabilistic model for code with decision trees," *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 731–747, 2016.

[44] Y. Dong, J. Ding, X. Jiang, G. Li, Z. Li, and Z. Jin, "Codescore: Evaluating code generation by learning code execution," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 3, pp. 1–22, 2025.

[45] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping language to code in programmatic context," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018, pp. 1643–1652.

[46] Z. Xu, S. Qiang, D. Song, M. Zhou, H. Wan, X. Zhao, P. Luo, and H. Zhang, "Dsfm: Enhancing functional code clone detection with deep subtree interactions," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[47] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *2014 IEEE international conference on software maintenance and evolution*. IEEE, 2014, pp. 476–480.

[48] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[49] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima *et al.*, "The pile: An 800gb dataset of diverse text for language modeling," *arXiv preprint arXiv:2101.00027*, 2020.

[50] "About us - mistral ai," https://mistral.ai/about.

[51] Q. Guo, X. Li, X. Xie, S. Liu, Z. Tang, R. Feng, J. Wang, J. Ge, and L. Bu, "Ft2ra: A fine-tuning-inspired approach to retrieval-augmented code completion," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 313–324.

[52] T. Zhu, Z. Liu, T. Xu, Z. Tang, T. Zhang, M. Pan, and X. Xia, "Exploring and improving code completion for test code," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 2024, pp. 137–148.

[53] X. Gao, Y. Xiong, D. Wang, Z. Guan, Z. Shi, H. Wang, and S. Li, "Preference-guided refactored tuning for retrieval augmented code generation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 65–77.

[54] Z. Yang, S. Chen, C. Gao, Z. Li, X. Hu, K. Liu, and X. Xia, "An empirical study of retrieval-augmented code generation: Challenges and opportunities," *ACM Transactions on Software Engineering and Methodology*, 2025.

[55] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[56] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[57] S. Dou, Y. Wu, H. Jia, Y. Zhou, Y. Liu, and Y. Liu, "Cc2vec: Combining typed tokens with contrastive learning for effective code clone detection," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1564–1584, 2024.

[58] "Models – hugging face," 9 2025. [Online]. Available: https://huggingface.co/models

[59] J. Martinez-Gil, "Evaluating small-scale code models for code clone detection," *arXiv preprint arXiv:2506.10995*, 2025.

[60] J. Lin, Y. Xie, Y. Yu, Y. Yang, and L. Zhang, "Toward exploring the code understanding capabilities of pre-trained code generation models," *arXiv preprint arXiv:2406.12326*, 2024.

[61] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in Statistics: Methodology and Distribution*. Springer, 1992, pp. 196–202.

[62] J. T. Halloran, M. Gulati, and P. F. Roysdon, "Mamba state-space models can be strong downstream learners," *arXiv e-prints*, pp. arXiv–2406, 2024.

[63] W. Huang, J. Zhang, G. Li, L. Zhang, S. Wang, F. Dong, J. Jin, T. Ogawa, and M. Haseyama, "Manta: Enhancing mamba for few-shot action recognition of long sub-sequence," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, no. 4, 2025, pp. 3751–3759.

[64] S. Ghosh-Dastidar and H. Adeli, "Spiking neural networks," *International journal of neural systems*, vol. 19, no. 04, pp. 295–308, 2009.

[65] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," *Communications of the ACM*, vol. 63, no. 11, pp. 139–144, 2020.

[66] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[67] D. Wang, B. Chen, S. Li, W. Luo, S. Peng, W. Dong, and X. Liao, "One adapter for all programming languages? adapter tuning for code search and summarization," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 5–16.

[68] N. Houlsby, A. Giurgiu, S. Jastrzebski, B. Morrone, Q. De Laroussilhe, A. Gesmundo, M. Attariyan, and S. Gelly, "Parameter-efficient transfer learning for nlp," in *International conference on machine learning*. PMLR, 2019, pp. 2790–2799.

[69] G. Li, C. Zhi, J. Chen, J. Han, and S. Deng, "Exploring parameter-efficient fine-tuning of large language model on automated program repair," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 719–731.