MPCODER: Multi-user Personalized Code Generator with Explicit and Implicit Style Representation Learning

Zhenlong Dai[1], **Chang Yao**[1], **Wenkang Han**[1], **Ying Yuan**[2], **Zhipeng Gao**[1*], **Jingyuan Chen**[1*]
[1]Zhejiang University
[2]Zhejiang Police College
{daizhenlong, changy, tracy1108, zhipenggao, jingyuanchen}@zju.edu.cn

Abstract

Large Language Models (LLMs) have demonstrated great potential for assisting developers in their daily development. However, most research focuses on generating correct code, how to use LLMs to generate personalized code has seldom been investigated. To bridge this gap, we proposed **MPCODER** (**M**ulti-user **P**ersonalized **Code** Generato**r**) to generate personalized code for multiple users. To better learn coding style features, we utilize explicit coding style residual learning to capture the syntax code style standards and implicit style learning to capture the semantic code style conventions. We train a multi-user style adapter to better differentiate the implicit feature representations of different users through contrastive learning, ultimately enabling personalized code generation for multiple users. We further propose a novel evaluation metric for estimating similarities between codes of different coding styles. The experimental results show the effectiveness of our approach for this novel task. The code and dataset are available at https://github.com/455849940/MPCoder.

1 Introduction

Nowadays, LLMs have been successfully used to support developers' daily development, such as code generation, test generation, etc. However, existing Code LLMs are usually general models trained with large programming corpus (Zheng et al., 2023; Chen et al., 2022), therefore the generated code is difficult to adapt to personalized and/or customized requests. Consider the following practical scenarios: Alice is a software developer. To improve programmers' daily efficiency, her company provided the base LLMs that can be used for code generation. Nonetheless, different developers/projects have their own coding standards and specifications. If Alice needs to generate code satisfying specific conventions, the base LLMs may fail

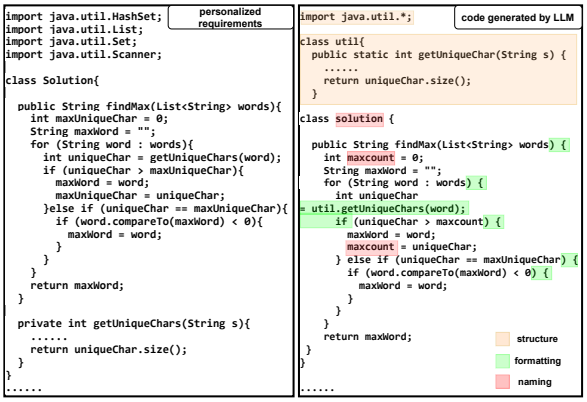

Figure 1: Example of code generated by LLMs and the corresponding personalized code that is expected, with areas inconsistent with the expectations marked in different colors within the model-generated code.

to capture these nuanced differences. As shown in Fig. 1, Alice has to painstakingly revise and review the generated code. If the custom style of the generated code conforms to the standard style of different developers/projects and is correct, it can greatly increase developer productivity (Kropp and Meier, 2013; Cheng et al., 2022; Song et al., 2023) and reduce code maintenance costs (Alkhatib, 1992; Tu et al., 2014).

Recent researchers have explored code generation task by using LLMs; however, most studies (Li et al., 2023b, 2022a; Ahmad et al., 2021; Hu et al., 2021) focus on generating "correct" code. There is limited research investigating how to generate "personalized" code, especially for multi-user personalization, with no research conducted yet. Automatically generating code according to developers' preferences or projects' consistency is a challenging task: (i) Considering different programmers have their own coding styles, **it is too expensive to fine-tune an LLM for each user** (Guo et al., 2021). Therefore, how to build an efficient model to generate personalized code for multiple users poses a significant challenge. (ii) **Coding styles are hard to learn and capture.** Coding styles include different aspects of the code, such as code

*Co-corresponding authors.

Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 3765–3780

naming, formatting, and structures. How to distinguish coding style differences between different users and obtain good style representations is another challenge. (iii) **Coding styles are hard to evaluate.** Unlike code correctness, which can be evaluated by executing test cases, there is no evaluation metric to estimate coding styles of different code fragments. How to evaluate coding styles quantitatively becomes another challenge for our study (Husain et al., 2019; Lu et al., 2021).

To tackle the above challenges, we propose a novel approach named **MPCODER**, which is designed to generate personalized code for multiple users according to their individual coding styles. After training, our model can be easily queried with the ID of the user and generate personalized code consistent with his/her desired coding styles. To better capture styles within the raw code, we encode the **explicit style features** and **implicit style features** to obtain an effective coding style representation. Regarding the explicit style features, we apply the coding style checking tool (Checkstyle[1] in our study) to detect different coding style attributes explicitly, then the model learn and encode these style attributes by residual learning, which guides the model in identifying the coding style attribute by contrasting two sets of coding style attributes (Section 2.2). For the implicit style features, to capture the subtle and unnoticed style differences between different users, we design a multi-user style adapter to further distinguish coding style differences among different users by using contrastive learning (Section 2.3). Finally, by combining the explicit coding style features and user-specific implicit style features, we can generate the user's personalized code that both contain the syntax and semantic styles of the code (Section 2.4). Due to the limited prior work on exploring personalized code generation, there is currently no effective way to estimate whether two pieces of code have similar coding styles. In this paper, we propose a novel evaluation metric, **Coding Style Score** (CSS), to quantitatively estimate the coding styles between two given codes.

In summary, our paper makes the following contributions: Firstly, current research mainly focuses on generating correct code, to the best of our knowledge, no prior work explored how to generate personalized code for multiple users. Secondly, we develop a novel model, named MPCODER, to learn multi-users' coding style features and generate personalized code. Thirdly, we propose a novel evaluation metric for estimating coding styles quantitatively, and additionally, we also release our dataset which contains source code written by multiple users. The experimental results show the effectiveness of our model over a set of baselines, showing its ability to generate personalized code while minimizing the degradation of code correctness. We hope our study can lay the foundations for this research topic and provide valuable insights into the potential for personalized generation capabilities of general LLMs.

2 Methodology

The coding style can be categorized into *syntax style* and *semantic style*, according to the structure and meaning of the code. Syntax style refers to the formatting rules of the code (*e.g.*, indentation, spacing, capitalization of variables) which can be easily defined (Allamanis et al., 2014; Markovtsev et al., 2019). On the other hand, semantic style refers to the use of language features and constructs to convey intent (*e.g.*, design patterns, and meaningful names of code) which is hard to precisely define in language (Parr and Vinju, 2016; Ogura et al., 2018). Both syntax and semantic style are important for making the code more readable and maintainable. Details of the syntax and semantic coding style differences can be found in Appendix B.3.

To capture these two types of coding styles, as illustrated in Fig. 2, we propose a novel approach MPCODER, which utilizes explicit coding style learning (Section 2.2) to capture the syntax style standards pre-defined by industry and implicit coding style learning (Section 2.3) to capture the semantic style that is learned from the code itself. Moreover, a multi-user style adapter (Section 2.3) is trained to estimate the style probability distribution for multiple users. After two stages of training , MPCODER can generate personalized code for multiple users simultaneously(Section 2.4).

2.1 Task Definition

Given a specific programming question q, the task of multi-user personalized code generation is to generate the corresponding code c for a particular user $u \in U$ based on his/her historical programming records r, where the generated code c should be consistent with the user's historical coding styles. It is important to note that the model should be able to generate personalized code for dif-

[1] http://checkstyle.sourceforge.net

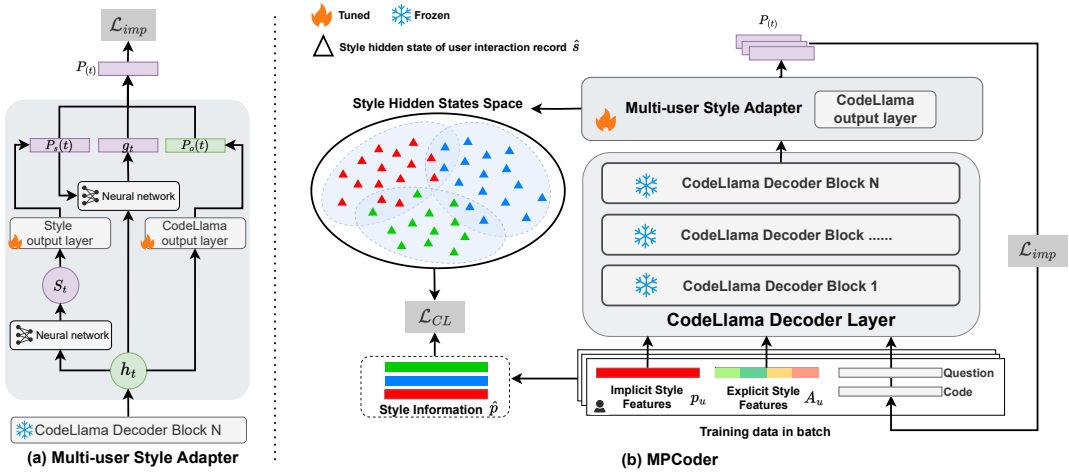

Figure 2: Overview of MPCODER. (a) illustrates the structure of the multi-user style adapter. (b) is the second training stage of MPCODER at the decoding step t.

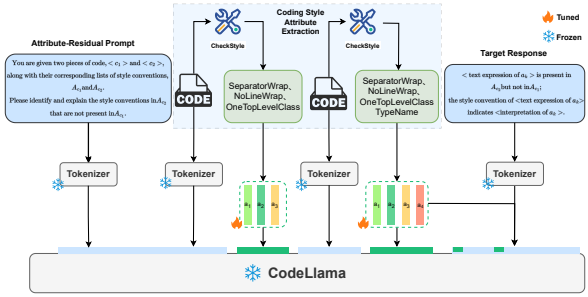

Figure 3: Explicit coding style residual learning.

ferent users simultaneously. More formally, the objective of personalized code generation is to learn the underlying conditional probability distribution $P_\theta(c|r, q, u)$ parameterized by θ. In other words, the goal is to train a model θ such that the probability $P_\theta(c|r, q, u)$ is maximized over the training dataset in order to generate personalized responses.

2.2 Explicit Coding Style Learning

Coding Style Attribute Extraction. The syntax style of code refers to a set of guidelines on how code is organized, such as indentation, spacing, capitalization of variables. These coding styles can be explicitly identified using a standard coding style checking tool. In this study, we use the Checkstyle[1] tool to explicitly examine 25 coding style attributes (*e.g.*, `SeparatorWrap`, `NoLineWrap`, etc.), with the detailed information provided in Appendix A.3.

Specifically, a piece of code c is identified by the Checkstyle tool to contain multiple coding style attributes. These attributes are vectorized as learnable representations, denoted as $A_c = \{a_1, a_2, ..., a_k\}$. Here, k is the number of attributes, $a_i \in \mathbb{R}^H$ denotes the i-th attribute where each a_i belongs to the 25 coding style attributes defined by Checkstyle, and H represents the dimension of the representation which is the same as the word embedding dimension of LLMs.

Coding Style Residual Learning. It is challenging for LLMs to directly acquire and distinguish the representations of all coding style attributes from the corresponding code. Inspired by previous study (Alayrac et al., 2022), which suggests that providing LLMs with two similar images along with their differences simultaneously can enhance their learning of image representations, we propose a novel residual learning mechanism to aid in the explicit recognition of each coding style attribute. Given two code fragments with similar coding style attributes, we guide the LLM in identifying the residual attribute to learn more precise and distinguishable attribute representations, as shown in Fig. 3.

Specifically, two pieces of code c_1 and c_2, exhibit similar coding style attributes denoted by $A_{c_1} = \{a_1, a_2, ..., a_{k-1}\}$ and $A_{c_2} = \{a_1, a_2, ..., a_k\}$, where a_k represents the residual attribute. Here we design attribute-residual prompt R, which inspires LLMs to identify the residual attribute a_k by comparing the representations of A_{c_1} and A_{c_2} as:

> • **Attribute-Residual Prompt.** You are given two pieces of code, <c_1> and <c_2>, along with their corresponding lists of style conventions, A_{c_1} and A_{c_2}. Please identify and explain the style conventions in A_{c_2} that are not present in A_{c_1}.
> • **Target Response.** <text expression of a_k> is present in A_{c_2} but not in A_{c_1}; the style convention of <text expression of a_k> indicates <interpretation of a_k>.

Here, <text expression of a_k> represents the name of attribute a_k, and <interpretation of a_k> denotes the detailed explanation. Through residual training, the learned coding style attribute representations

are consistent with the corresponding syntax styles.

Training Objective. In this training stage, the objective is to minimize the negative log-likelihood by utilizing the attribute-residual prompt as:

$$\mathcal{L}_{\text{exp}} = -\sum_{t=1}^{l} \log P(x_t|x_{<t}; A), \tag{1}$$

where l is the number of tokens in the attribute-residual prompt and the target response, $P \in \mathbb{R}^{|\mathcal{V}|}$ is the probability distribution on LLM's vocabulary, and A denotes the representations of coding style attributes that are to be learned in this stage.

2.3 Implicit Coding Style Learning

Explicit coding style learning mainly focuses on capturing the syntax style features that are predefined by industry standards and are considered to be user-independent. However, it is important to note that there is another type of coding style features that are more difficult to precisely define using language. These features are user-specific and may vary from one individual to another. We define this feature as semantic style feature, which refers to the use of language features and constructs to convey intent. For example, some user prefer to use i, k, j to denote variables, while others may want to use more meaningful variable names.

To obtain semantic features of coding styles, we utilize implicit coding style learning to learn each user's style features from their historical coding records. Since detecting the coding styles of various users can be intricate and challenging, we propose a multi-user style adapter to better differentiate the implicit feature representations of different users through contrastive learning. Subsequently, the multi-user style adapter estimates the probability distribution of styles over the entire vocabulary for multiple users. Additionally, it incorporates personalized fine-tuning based on implicit features, thereby minimizing the necessity to fine-tune and store multiple copies of LLM for different users.

Implicit Style Features. Inspired by personalized lightweight fine-tuning (Lester et al., 2021; Li et al., 2023a; Zlotchevski et al., 2022), we guide the personalized generation of LLMs for each user $u \in U$ with a set of pre-trained continuous vector representations, namely implicit style features. Specifically, we aim to learn implicit style features $p_u = \{p_1, p_2, ..., p_m\}$ for user u using their historical coding records. Here, $p_i \in \mathbb{R}^H$ is the i-th learnable representation for user u and m is the number of learnable representations. We keep most of the LLMs' parameters fixed, and only tune the output layer to accelerate convergence. Given a programming question q and its corresponding code c, represented as a token sequence $x = \{x_1, x_2..., x_n\}$ of length n, we obtain a sequence of token embeddings e through the embedding layer of LLMs. User-specific semantic style features p_u are then concatenated with token embeddings as the input of decoder layer in LLMs as:

$$e = \texttt{EmbeddingLayer}(\{x_0, x_1..., x_n\}), \tag{2}$$

$$h = \texttt{DecoderLayer}([p_u; e]), \tag{3}$$

where $e = \{e_1, e_2, ..., e_n\}$ and the hidden states of the decoder layer are denoted as $h = \{h_1, h_2, ..., h_{n+m}\}$. Each token embedding e_i and implicit style feature p_i share the same dimension H. Implicit semantic style features p_u are learned based on the following prompt template:

> • **Prompt template.** <p_u> Give you a programming question <q> and corresponding user coding style conventions <A_u>, please give the corresponding style of the answer in Java.
> • **Target Code.** <c>

where A_u denotes the learned coding style attribute representations of user u in the first stage. After the training process, these user-specific semantic style features p_u can be learned from the user-generated code itself, allowing for the implicit expression of the users' coding style.

Multi-user Style Adapter. The generic nature of the output layer in LLMs entails that it does not take into account personalized generation requirements. Therefore, even if LLMs receive user-specific semantic style features, they are unable to effectively translate these features into personalized outputs. To tackle this issue, we propose a multi-user style adapter aimed at bridging the gap between generic outputs and user-specific personalized outputs, ultimately enabling personalized generation for multiple users.

Specifically, as illustrated in Fig. 2(a), at each decoding step t, the style hidden states are extracted by a simple neural network consisting of two fully connected layers as:

$$s_t = W_c(W_h h_t + b_h) + b_c, \tag{4}$$

where s_t represents the style hidden states at the t-th step of the output sequence. W_* and b_* denote the trainable parameters in this section. The obtained style hidden states s_t are then passed through

a feed-forward layer with Softmax function to estimate the style probability distribution over the entire vocabulary $\mathcal{V}$ for user u as:

$$P_{\mathrm{s}}(x_t|x_{<t}, A_u; p_u) = \texttt{Softmax}(W_s s_t + b_s), \quad (5)$$

where A_u denotes the learned coding style attribute representations of user u which appear in his code records in the training data. For simplicity, we omit the other trainable parameters. To merge the style probability distribution and the generic probability distribution of LLMs, we incorporate a dynamic gate vector g_t, which indicates the weight between the two distributions. The gate vector g_t is derived by combining the hidden state of the decoder layer h_t with the style distribution $P_{\mathrm{s}}(x_t|x_{<t}, A_u; p_u)$ as:

$$s_t^{'} = \texttt{Relu}(W_g P_{\mathrm{s}}(x_t|x_{<t}, A_u; p_u) + b_g), \quad (6)$$

$$h_t^{'} = W_k h_t + b_k, \quad (7)$$

$$g_t = \texttt{Sigmoid}(W_r(s_t^{'} + h_t^{'}) + b_r), \quad (8)$$

where $g_t \in \mathbb{R}^{|\mathcal{V}|}$ represents the dynamic gating vector for step t. The final probability distribution $P(x_t|x_{<t}, A_u; p_u)$ is then derived as:

$$\begin{aligned} P(x_t|x_{<t}, A_u; p_u) = g_t \cdot P_{\mathrm{s}}(x_t|x_{<t}, A_u; p_u) \\ +(1 - g_t) \cdot P_{\mathrm{o}}(x_t|x_{<t}, A_u; p_u), \end{aligned} \quad (9)$$

where $P_{\mathrm{o}} \in \mathbb{R}^{|\mathcal{V}|}$ denotes the generic distribution without the style adapter.

Contrastive Learning. In our approach, the use of a shared style adapter by multiple users necessitates a clear distinction in style hidden states among users. Therefore, we incorporate a contrastive learning strategy into our model to aid in learning style hidden states based on global style features, which includes both syntax and semantic style features. Specifically, we define the global style features $\hat{p}$ and global style hidden states $\hat{s}$ respectively as:

$$\hat{p} = \frac{1}{m}\sum_{i=1}^{m} p_i + \frac{1}{k}\sum_{i=1}^{k} a_i, \quad (10)$$

$$\hat{s} = \frac{1}{m+n}\sum_{t=1}^{m+n} s_t. \quad (11)$$

We aim to maximize the correlation between $\hat{p}$ and $\hat{s}$ of the same user while minimize the correlation with other non-matching pairs. Specifically, we formulate the contrastive objective as:

$$\mathcal{L}_{\mathrm{CL}} = -\log \frac{\exp(corr(\hat{p}_u, \hat{s}_u)/\tau)}{\sum_{u^-} \exp(corr(\hat{p}_u, \hat{s}_{u^-})/\tau)}, \quad (12)$$

where τ is the temperature parameter. By minimizing this loss, the style hidden states can be optimized to enhance personalized expressiveness.

Training Objective. In this training stage, the generation loss is defined as:

$$\mathcal{L}_{\mathrm{imp}} = -\sum_{t=1}^{n} \log P(x_t|x_{<t}, A_u; p_u). \quad (13)$$

We train implicit style features, the output layer, and the multi-user style adapter with the contrastive loss jointly as:

$$\mathcal{L} = \mathcal{L}_{\mathrm{imp}} + \alpha\mathcal{L}_{\mathrm{CL}}, \quad (14)$$

where α is a hyper-parameter.

2.4 Inference

During the inference stage, when presented with a programming question q for a specific user u, we are able to gather the coding style attributes A_u and implicit style features p_u. We then utilize the same prompt template as the second training stage to generate personalized code c for user u.

3 Experiments

3.1 Experimental Setup

Dataset. Until now, no public dataset is available for personalized code generation task. In this study, we first build two datasets for this novel task. Our datasets are collected from PTA[2](Programming Teaching Assistant), for enhancing students' programming skills and will be made public upon acceptance of this work. The platform has recorded different users' problem-solving histories for different programming problems. For each user, we can collect all his/her written code for solving different problems. After our empirical exploration, most platform users have a few problem-solving records, while only a small subset have extensive records. Thus we construct two datasets **PCISparse** (Personalized Code Interaction Sparse) and **PCIDense** (Personalized Code Interaction Dense), based on problem sets with average user interaction records below and above 100. Overall statistics of the two dataset are given in Table 1. After data collection, we split the dataset into training, validation and testing set by the ratio of 8:1:1. Further details can be found in Appendix A.1.

[2]https://pintia.cn/

Dataset	records	users	records per user		
			max	min	avg
PCIDense	5794	50	178	101	115
PCISparse	34642	1121	113	17	31

Table 1: Dataset Statistics.

Baselines. We compare our model with state-of-the-arts methods as follows: (1) **CodeLlama**: CodeLlama-Instruct-7B (Roziere et al., 2023) can be seen as a benchmark without personalization; (2) **DAPT**: Domain Adaptive Pre-Training (Gururangan et al., 2020); (3) **L-LDB**: Customization for a specific software project (Zlotchevski et al., 2022); (4) **Adapter**: Adapter is used to fine-tune the LLM (Houlsby et al., 2019).Further details can be found in Appendix A.2.

Implementation Details. We choose CodeLlama (Roziere et al., 2023) as the base LLM. For training, the model is optimized with AdamW (Loshchilov and Hutter, 2017) and Fully Sharded Data Parallel (Ott et al., 2021). During decoding, code is generated using greedy decoding. Further details can be found in Appendix A.3.

3.2 Evaluation Metrics

CSS Evaluation Metrics. The personalized code generation task is more concerned with generated code styles, the traditional evaluation metrics such as BLEU (Papineni et al., 2002) and Rouge (Lin and Och, 2004) scores neglect important syntactic and semantic features of code and are not suitable for evaluating coding styles. Inspired by the empirical research on coding styles (Zou et al., 2019), we first propose an evaluation metric, namely Coding Style Score (CSS), for evaluating coding styles between different codes.

Specifically, we characterized the coding style with 24 style criteria (detail in Appendix B.2) for Java programming language. These criteria are associated with three aspects of coding styles: code structure, formatting, and naming. These style criteria also comply with Google's Java coding style[3]. For a given Java code file, it can be parsed with a 24-dimensional coding style criteria vector. Each dimension signifies the percentage of a specific criteria violation.

More formally, the coding style criteria vector of a Java file could be defined as $c = \langle c_1, c_2, ..., c_{24} \rangle$, where $c_i \in [0, 1]$ describes the extent of the i-th coding style criteria has been violated. We define the CSS metric between the generated code c_{gen} and the reference code c_{ref} as follows:

$$\text{css}(c_{\text{gen}}, c_{\text{ref}}) = 1 - D_{JS}(c_{\text{gen}}, c_{\text{ref}}), \quad (15)$$

where $\text{css} \in [0, 1]$, and D_{JS} is JS (Jensen-Shannon) divergence: it measures the similarity of two probability distributions as:

$$D_{JS}(p, q) = \frac{1}{2}[D_{kl}(p \| \frac{p+q}{2}) + D_{kl}(q \| \frac{p+q}{2})], \quad (16)$$

where D_{kl} is Kullback-Leibler Divergence as:

$$D_{kl}(p \| q) = \sum_{i=1}^{n} p(x_i) \log(\frac{p(x_i)}{q(x_i)}). \quad (17)$$

With CSS evaluation metric, we can provide a quantitative way to measure the coding style between different codes. The larger CSS metric is, the more similar the coding styles of the generated code and the reference code are.

Correctness Evaluation Metrics. Regarding our task, we hope to not only generate personalized code, but also maintaining the correctness of the generated code at the same time. We used HumanEval-X[4] to evaluate the correctness of the generated code. HumanEval-X contains 164 Java programming problems and their corresponding test cases, a code is considered as correct if it passes all test cases for a specific programming problem.

3.3 Experimental Results

Coding Style Evaluation. Table 2 shows the results of different models on PCIDense and PCISparse datasets respectively. It is obvious that: (1) MPCODER outperforms all other baselines in terms of the CSS evaluation metric, which verifies the advantage of our approach in learning code styles from code. (2) We include three variants of MPCODER, namely $\text{MPCODER}_{\text{ISF}}$, $\text{MPCODER}_{\text{ESF}}$ and $\text{MPCODER}_{\text{IES}}$, as baselines. The $\text{MPCODER}_{\text{ISF}}$ only uses the implicit style features as input; The $\text{MPCODER}_{\text{ESF}}$ only uses the explicit style features as input; The $\text{MPCODER}_{\text{IES}}$ uses the implicit and explicit style features without changing the structure of the CodeLlama (*i.e.*, not adding Multi-user Style Adapter and Contrastive Learning). The experimental results show that incorporating implicit features can notably enhance CodeLlama's performance in terms of textual similarities (*i.e.*, BLEU

[3]http://google.github.io/styleguide/javaguide.html

[4]https://github.com/THUDM/CodeGeeX2

Number of users	Model	PCIDense				PCISparse			
		CSS	BLEU	Rouge-1	Rouge-2	CSS	BLEU	Rouge-1	Rouge-2
Single	CodeLlama	48.73	48.09	39.23	25.11	46.29	50.28	37.49	24.20
	DAPT	27.38	30.78	40.01	28.77	34.29	39.21	49.17	38.30
	L-LDB	**64.06**	54.93	46.35	34.12	**61.09**	57.89	44.18	31.53
	Adapter	49.44	49.72	43.95	31.12	39.78	42.16	25.16	13.84
Multiple	MPCODER$_{ISF}$	56.68	56.35	42.67	29.06	60.34	58.10	41.53	27.56
	MPCODER$_{EFS}$	57.22	56.17	41.63	27.02	62.52	55.71	40.50	26.73
	MPCODER$_{IES}$	63.26	55.72	42.73	29.17	65.61	56.67	40.91	27.26
	MPCODER	**64.50**	55.73	41.56	28.25	**66.18**	57.50	41.74	28.11

Table 2: Evaluation results on the Java personalized code generation dataset PCIDense and PCISparse. All results in the table are reported in percentage (%). "Single" represents the model can only be used for a single user. We report the BLEU and Rouge scores for reference by calculating BLEU-4 and Rouge-1/2.

Model	multi-user	Parameters	
		Total	Trained
DAPT	✗	6.7B*N	100%*N
L-LDB	✗	6.7B*N	3.00%*N
Adapter	✗	(6.7B+1.2M)*N	0.02%*N
MPCODER$_{ISF}$	✓	20K*N+6.7B	3e−4% * N + 1.95%
MPCODER$_{ESF}$	✓	96K+6.7B	1.95%
MPCODER$_{IES}$	✓	20K*N+96K+6.7B	3e−4% * N+1.95%
MPCODER	✓	20K*N+96K+6.9B	3e−4% * N+4.82%

Table 3: Comparison of the total size and trainable parameters. N denotes the number of users and 'multi-user' indicates whether the model supports multi-user. 96K is the number of parameters for 25 explicit coding style attributes.

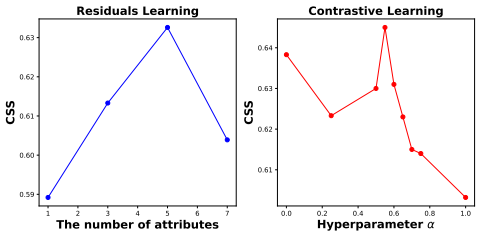

Figure 4: Residual Learning (left) and Contrastive Learning Hyperparameter (right) Effects on CSS.

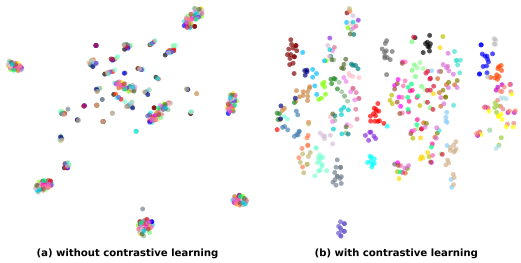

Figure 5: t-SNE visualization results: style hidden states of user interaction records on PCIDense. Different colors denote different users.

and Rouge) and coding style similarities (*i.e.*, CSS). After adding explicit features (MPCODER$_{IES}$), the textual similarities almost stay the same while the CSS has been further significantly improved. This confirms the effectiveness of incorporating explicit and implicit features learning in our study. (3) The L-LDB is a customized baseline tailored to individual users, which has its advantage compared with other baselines. However, as it is specifically designed for single users, it requires the model to be retrained for each user, making it cost-prohibitive for multi-user code generation. Qualitative examples of our model and other models can be found in Appendix B.4.

Cost Analysis. Table 3 shows the total size and trainable parameters of each model. The results show that MPCODER can greatly reduce training and storage costs for multiple users. The cost of adding new users to our model can be neglected (3e-4% in parameters per user).

Residual and Contrastive Learning Analysis. There are two key hyperparameters, which are the number of attributes in residual learning and the weight parameter α in the loss function. In this section, we evaluate the the influence of these two parameters. The number of attributes means the maximum number of style attributes in the attribute-residual prompt. As shown in Fig. 4(a), when setting the number of attributes to 1, it equals to not utilizing residual learning. The experimental results show that employing residual learning with 5 attributes can effectively guide LLM to understand code attributes. For hyperparameters α, as shown in Fig. 4(b), we vary its value from 0 to 1.0 with a step size of 0.25. Then we further adjust it from 0.50 to 0.75 with a step size of 0.05. The experimental results show that setting α to 0.55 can achieve optimal performance.

Furthermore, we visually represent the style hidden states of 50 users to demonstrate the effectiveness of contrastive learning in Fig. 5. The hidden states of records from the same user are clustered together, while those from different users are effectively separated.

Ablation Study. As illustrated in Table 4, we performed an ablation study by removing key com-

Model	CSS	
	PCIDense	PCISparse
MPCODER	**64.64**	**66.18**
w/o Coding Style Attributes	58.19	64.13
w/o Contrastive Learning	63.83	63.36
w/o Multi-user Style Adapter	63.26	65.61

Table 4: Ablation study.

Numbers of user	Model	PCIDense	PCISparse
Single	CodeLlama	30.49%	30.49%
	DAPT	-	-
	L-LDB	**28.62**%	22.56%
	Adapter	27.32%	**29.73**%
Multiple	$\text{MPCODER}_{\text{ISF}}$	29.57%	25.76%
	$\text{MPCODER}_{\text{ESF}}$	30.24%	**28.53**%
	$\text{MPCODER}_{\text{IES}}$	27.88%	26.60%
	MPCODER	**31.25**%	26.18%

Table 5: Correctness evaluation.

ponents (*i.e.*, Coding Style Attributes, Contrastive Learning and Muti-user Style Adapter) of MPCODER separately. The experimental results show that: (1) No matter which component we drop, it hurts the overall performance of our model, which signals the importance and effectiveness of all three components. (2) The CSS drops most significantly on PCIDense and PCISparse datasets when the Coding Style Attributes and Contrastive Learning component are removed, showing these two components can complement each other under different situations, which justifies the importance and necessity of the above two components.

Correctness Evaluation. We aim to generate personalized code while still keeping the code correct. We further evaluate the correctness of each baseline method on HumanEval-X dataset. We report the average accuracy of the generated code based on three prompts. Further details can be found in Appendix B.1. As shown in Table 5, DAPT is unable to generate compilable code due to an excessive focus on word overlap, resulting in repeated instances such as 'doublee' for data types. From the table, we can observe that: (1) Although L-LDB performs relatively well in personalized code generation, its code correctness has been greatly affected compared with CodeLlama. (2) Adapter maintains the generated code correctness with CodeLlama. However, it is not suitable for generating personalized code. The code generated by MPCODER achieves a good balance between correctness and personalization compared to all baselines.

Human Study. To verify the effectiveness of our CSS metric, we conduct a human study to compare the results of CSS with human results. In particular, we compare MPCODER with L-LDB and Adapter

Numbers of user	Model	Rate of the best models	
		human evaluation	CSS
Single	L-LDB	37%	41%
Single	Adapter	12%	13%
Multiple	MPCODER	43%	46%
-	Undecided	8%	-

Table 6: Human study.

by conducting a user study on PCIDense. We asked 5 users to answer questionnaires of 60 comparative questions, totaling 300 answers. All questions present them with a choice between two options. Users are asked to answer the question: "Which of the two code copies is closer in coding style to the reference code?", and every user is provided with three options (i.e., A is Better, B is Better, Cannot Determine/Both Equally). Further detail can be found in Appendix A.4. We calculate the CSS values of the two codes and the reference code for different models respectively. The model with the highest CSS value is regarded as the best model. Table 6 shows the results of the human study.

The results obtained by CSS are consistent with human evaluation(MPCODER > L-LDB > Adapter), which verifies the effectiveness of our proposed CSS evaluation metric for estimating coding styles. By comparing the results of different models, MPCODER outperforms the L-LDB and Adapter significantly and consistently. The experimental results show our model's superiority in both automatic evaluation (CSS) and human evaluation.

Adaptation For New Users. Introducing a new user to our model can be categorized into two scenarios: (1) the new user's historical coding records are available; (2) the new user's historical coding records are unavailable. In Appendix A.5, we discuss in detail how MPCODER effectively adapts to these two types of new users.

4 Related Work

Code Pre-trained Language Models. With the latest developments in the Transformer-based model (Vaswani et al., 2017), recent work has attempted to apply LLM to code to advance software engineering and code intelligence. CodeBERT (Feng et al., 2020) pretrains the NL-PL data. CodeT5 (Wang et al., 2021) leverages the T5 (Raffel et al., 2020) architecture to leverage code semantics through identifier tokens. LLM (Radford et al., 2019; Brown et al., 2020) has made many achievements in the field of natural language processing. the OpenAI Codex (Chen et al., 2021)

model with 12B parameters pioneered and demonstrated the potential of large code generation models pre-trained on billions of lines of public code. Subsequently, models dedicated to code generation emerged, such as CodeLlama (Roziere et al., 2023) and CodeGeex (Zheng et al., 2023).

Personalized Generation. Most of the existing work in personalized generation focuses on attribute-based controlled text generation (Keskar et al., 2019), such as emotions and topics (Dathathri et al., 2019; Kong et al., 2021; Xu et al., 2022). The text-description-based approach (Song et al., 2021) focuses on promoting character consistency through pre-trained language models. The embedding-based utilizes user ID information (Al-Rfou et al., 2016) or embedded user dialogue (Ma et al., 2021) history as an implicit profile. Within the domain of personalized code generation, existing approaches (Zlotchevski et al., 2022) involves fine-tuning for specific software projects, providing Java unit tests for a single coding style, while we focus on providing coding styles for multiple users.

5 Conclusions

This research aims to generate personalized code for multiple users to satisfy the coding styles of different developers or projects. To perform this novel task, we propose an approach MPCODER which utilizes explicit coding style residual learning to capture the syntax style standards and implicit coding style learning to capture the semantic style of each user. We propose a multi-user style adapter to bridge the gap between the generic outputs and user-specific personalized outputs. MPCODER can ultimately generate personalized code for multiple users simultaneously. The experimental results show the effectiveness of our for this task. We hope our study can lay the foundations for this new research and provide valuable insights into the potential for personalized generation capabilities of general LLMs.

6 Limitations

Several limitations are concerned with our work. Firstly, our study is based on Java, which is one of the most popular programming languages used by developers. However, our approach is language-independent, we believe our approach can be easily adapted to other programming languages such as Python or Javascript. Secondly, the correctness of the generated code has been affected when our model was applied to the PCISparse dataset. Exploring effective ways to generate personalized code while maintaining its correctness with a limited number of data samples is an interesting research topic for our future work.

7 Ethics Statement

To prevent privacy leaks, we have removed personal and sensitive information from our dataset, utilizing anonymous IDs as individual identifiers. Specifically, the raw data underwent an initial preprocessing step to transform it into structured data. We manually identified labels that may pose privacy risks (e.g., ID, user name, email address, age, gender), and then anonymized the corresponding information by either deleting it or mapping it to new values. We explore the feasibility of using LLMs to perform personalized code generation. However, LLMs such as CodeLlama may have some ethical biases, and these ethical concerns inevitably affect our proposed approach Ethical guidelines and the deployment of such techniques should be considered to mitigate potential negative consequences. We hope our work will stimulate further investigation and advancement in this novel research area of personalized code generation and the general personalized generation abilities of LLMs.

8 Acknowledgement

This work was supported by the National Natural Science Foundation of China (No.62037001, No.62307032, No.62293555) and the Shanghai Rising-Star Program (23QA1409000).

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*.

Rami Al-Rfou, Marc Pickett, Javier Snaider, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. 2016. Conversational contextual cues: The case of personalization and history for response ranking. *arXiv preprint arXiv:1606.00372*.

Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, et al. 2022. Flamingo: a visual language model for few-shot learning. *Advances in Neural Information Processing Systems*, 35:23716–23736.

Ghazi Alkhatib. 1992. The maintenance problem of application software: An empirical analysis. *Journal*

of Software Maintenance: Research and Practice, 4(2):83–104.

Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering*, pages 281–293.

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Nuo Chen, Qiushi Sun, Renyu Zhu, Xiang Li, Xuesong Lu, and Ming Gao. 2022. Cat-probing: A metric-based approach to interpret how pre-trained models for programming language attend code structure. *arXiv preprint arXiv:2210.04633*.

Lan Cheng, Emerson Murphy-Hill, Mark Canning, Ciera Jaspan, Collin Green, Andrea Knight, Nan Zhang, and Elizabeth Kammer. 2022. What improves developer productivity at google? code quality. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1302–1313.

Sumanth Dathathri, Andrea Madotto, Janice Lan, Jane Hung, Eric Frank, Piero Molino, Jason Yosinski, and Rosanne Liu. 2019. Plug and play language models: A simple approach to controlled text generation. *arXiv preprint arXiv:1912.02164*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Han Guo, Bowen Tan, Zhengzhong Liu, Eric P Xing, and Zhiting Hu. 2021. Text generation with efficient (soft) q-learning. *arXiv e-prints*, pages arXiv–2106.

Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A Smith. 2020. Don't stop pretraining: Adapt language models to domains and tasks. *arXiv preprint arXiv:2004.10964*.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pages 2790–2799. PMLR.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Nitish Shirish Keskar, Bryan McCann, Lav R Varshney, Caiming Xiong, and Richard Socher. 2019. Ctrl: A conditional transformer language model for controllable generation. *arXiv preprint arXiv:1909.05858*.

Xiangzhe Kong, Jialiang Huang, Ziquan Tung, Jian Guan, and Minlie Huang. 2021. Stylized story generation with style-guided planning. *arXiv preprint arXiv:2105.08625*.

Martin Kropp and Andreas Meier. 2013. Teaching agile software development at university level: Values, management, and craftsmanship. In *2013 26th International Conference on Software Engineering Education and Training (CSEE&T)*, pages 179–188. IEEE.

Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*.

Lei Li, Yongfeng Zhang, and Li Chen. 2023a. Personalized prompt learning for explainable recommendation. *ACM Transactions on Information Systems*, 41(4):1–26.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023b. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022a. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Zhen Li, Guenevere Chen, Chen Chen, Yayi Zou, and Shouhuai Xu. 2022b. Ropgen: Towards robust code authorship attribution via automatic coding style transformation. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1906–1918.

Chin-Yew Lin and Franz Josef Och. 2004. Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*, pages 605–612.

Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Zhengyi Ma, Zhicheng Dou, Yutao Zhu, Hanxun Zhong, and Ji-Rong Wen. 2021. One chatbot per person: Creating personalized chatbots based on implicit user profiles. In *Proceedings of the 44th international ACM SIGIR conference on research and development in information retrieval*, pages 555–564.

Vadim Markovtsev, Waren Long, Hugo Mougard, Konstantin Slavnov, and Egor Bulychev. 2019. Style-analyzer: fixing code style inconsistencies with interpretable unsupervised algorithms. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 468–478. IEEE.

Naoto Ogura, Shinsuke Matsumoto, Hideaki Hata, and Shinji Kusumoto. 2018. Bring your own coding style. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 527–531. IEEE.

Myle Ott, Sam Shleifer, Min Xu, Priya Goyal, Quentin Duval, and Vittorio Caggiano. 2021. Fully sharded data parallel: faster ai training with fewer gpus.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Terence Parr and Jurgen Vinju. 2016. Towards a universal code formatter through machine learning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 137–151.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Andrew H Song, Guillaume Jaume, Drew FK Williamson, Ming Y Lu, Anurag Vaidya, Tiffany R Miller, and Faisal Mahmood. 2023. Artificial intelligence for digital and computational pathology. *Nature Reviews Bioengineering*, 1(12):930–949.

Haoyu Song, Yan Wang, Kaiyan Zhang, Wei-Nan Zhang, and Ting Liu. 2021. Bob: Bert over bert for training persona-based dialogue models from limited personalized data. *arXiv preprint arXiv:2106.06169*.

Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280.

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

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Xinchao Xu, Zhibin Gou, Wenquan Wu, Zheng-Yu Niu, Hua Wu, Haifeng Wang, and Shihang Wang. 2022. Long time no see! open-domain conversation with long-term persona memory. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 2639–2650.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. In *KDD*.

Andrei Zlotchevski, Dawn Drain, Alexey Svyatkovskiy, Colin B Clement, Neel Sundaresan, and Michele Tufano. 2022. Exploring and evaluating personalized models for code generation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1500–1508.

Weiqin Zou, Jifeng Xuan, Xiaoyuan Xie, Zhenyu Chen, and Baowen Xu. 2019. How does code style inconsistency affect pull request integration? an exploratory study on 117 github projects. *Empirical Software Engineering*, 24:3871–3903.

A Experimental setup details

A.1 Dataset Construction Details

PCIDense(Personalized Code Interaction Dense) have 382 programming problems, PCISparse (Personalized Code Interaction Sparse) have 662 programming problems. Since codeLlama mainly supports English, we use Chatgpt-3.5[5] to translate non-English problems into English. We only keep the record that the user can pass all test samples for

[5]https://chat.openai.com/

the same problem no more than three times, and choose one at random as the user's answer to the problem. We combined interaction records from problem sets with average user interaction records both below and above 100 to form the PCISparse and PCIDense datasets, respectively. Then we conduct a random split of 8/1/1 by programming problem, validation, and test to avoid data leakage. To prevent privacy disclosure, we have excluded personal and sensitive data such as user names and email addresses, retaining only the unique student IDs as individual identifiers.

A.2 Baseline Setup Details

We compare our model with mainstream models as follows:**(1)CodeLlama**: CodeLlama Instruct 7B version can be seen as a benchmark for code generation without personalization, we can observe the degree to which each method succeeds in personalizing the LLM for the target task; **(2)DAPT**: We perform domain Adaptive Pre-Training (Gururangan et al., 2020) by finetuning on user-specific data; **(3)L-LDB**: it personalizes to a specific software project for personalized unit test generation of Java methods (Zlotchevski et al., 2022). This is achieved by freezing most parameters in the baseline model and only performing lightweight fine-tuning on the last decoder block; **(4)Adapter**: It is used to fine-tune the LLM, and the parameters of the original network remain unchanged, achieving a high degree of parameter sharing (Houlsby et al., 2019). We split the data in the dataset according to the user, and train a corresponding single-user model with each user's data in turn. The result is calculated by averaging the sum of all users' metrics. For parameter Settings of Adapter, refer to llama-recipes[6]. The llama-recipes repository provides fine-tuning code for CodeLlama. In the training stage of baselines, the prompt template for baselines is shown in Fig. 6.

- **Prompt template.** Give you a programming problem <q>, please provide answers in Java.
- **Target Response.** <c>

Figure 6: The prompt template for baselines.

A.3 Training and Decoding Details

we set τ as 0.5 and α as 0.55. In the first training stage, we set the batch size to 8. In the second training stage, we set the batch size to 4. In both training stages, we used four A800 graphics cards to train the model and truncate the total length of the input statement and output target to a maximum of 2048 tokens, the learning rate is set to 1e-4.

In the first training of explicit coding style attributes, we choose 25 style attributes, of which 24 attributes are shown in Fig. 7. Some style attributes for CheckStyle checking are shared by all users. These feature differences of users cannot be reflected in the evaluation of personalized code generation. However, during the process of acquiring coding style attributes, leveraging features shared among all users' code assists the LLM in gaining a deeper understanding of these attributes. Therefore, such features are preserved in the residual learning dataset to facilitate the acquisition of coding style attributes. Consequently, we retain "Indentation" as a fundamental style convention, which means: "Control indentation between comments and surrounding code.". Since some semantic coding style features can be explicitly defined by CheckStyle, we also put them into the explicitly coding attributes. Therefore, explicit coding style attributes focus more on learning syntax style, while implicit style features are more concerned with learning semantic style.

For coding style attributes training, we construct a record with the number of style attributes less than or equal to 5 and balance the attribute of each style in the training. The number of records for each style as a residual attribute does not exceed 600 pieces. No more than 75 pieces of data were evaluated for each style attribute. In the experiment for studying the number of attributes, setting the number of attributes to 1 means without using residual learning, we use prompt as shown in Fig. 7. For implicit style features training, the number of vector representations of implicit style features for each user is set to 5.

- **Attribute Prompt without Residual.** You are given one piece of code <c> along with their corresponding style convention <A_u>. Please identify and explain the style convention.
- **Target Response.** <text expression of a_k> is present in code; the style convention of <text expression of a_k> indicates <interpretation of a_k>.

Figure 7: The prompt template without residual learning.

[6]https://github.com/facebookresearch/llama-recipes

aspect	criteria	Coding Style	Description
Structure	NoLineWrap	Syntax	Do not put '}' on its own line.
	AvoidStarImport	Syntax	Do not break after ',' but before '.'.
	OneTopLevelClass	Syntax	break import and package lines.
	EmptyLineSeparator	Semantic	import statements that use the * notation.
Formatting	RightCurly	Syntax	Do not put '}' on its own line.
	SeparatorWrap	Syntax	Do not break after ',' but before '.'.
	WhitespaceAround	Syntax	Do not use a space between a reserved word and its follow-up bracket,*e.g.*, if(.
	GenericWhitespace	Syntax	Use a space before the definition of generic type, *e.g.*, List <.
	OperatorWrap	Syntax	Break after '=' but after other binary operators.
	LineLength	Semantic	The line length exceeds 100 characters.
	LeftCurly	Syntax	Do not put '{' on the same line of code.
	EmptyBlock	Syntax	Have empty block for control statements.
	NeedBraces	Syntax	Do not use braces for single control statements.
	MultipleVariableDeclarations	Syntax	Not every variable declaration is in its own statement and on its own line.
	OneStatementPerLine	Syntax	there is not only one statement per line.
	UpperEll	Syntax	long constants are defined with an upper ell. That is 'l' and not 'L'.
	ModifierOrder	Syntax	Do not follow the order: public, protected, private, abstract, default, static, final, transient, volatile, synchronized, native, strictfp.
	FallThrough	Semantic	Do not put a fall-through comment in a switch If a 'case' has no break, return, throw, or continue.
	MissingSwitchDefault	Semantic	switch statement does not has a default clause.
Naming	TypeName	Syntax	Type name is not in UpperCamelCase.
	MethodName	Syntax	Method name is not in lowerCamelCase.
	MemberName	Syntax	Member name is not in lowerCamelCase.
	ParameterName	Syntax	Parameter name is not in lowerCamelCase.
	LocalVariableName	Syntax	Local variable name is not in lowerCamelCase.

Table 7: The 24 criteria for characterizing the coding style.

• **Prompt template 1.** Here is an incomplete code <*q*>, you need to complete. Wrap your code answer using ```, your code must include a complete implementation of the 'Solution' class with exactly one function in the class.
• **Prompt template 2.** Give you a piece of Java code, please continue to write the unfinished function <*q*>.
• **Prompt template 3.** Give you a programming question <*q*>, please provide answers in Java.

Figure 8: The prompt templates for the correctness test.

A.4 Details of Human Study

We conduct a user study to verify the effectiveness of our CSS metric for evaluating coding styles compared with humans. In particular, we compare MPCoder with L-LDB and Adapter by conducting a user study on PCIDense dataset. We asked 5 users to answer questionnaires of 60 comparative questions, totaling 300 answers. All of the users are majored in Computer Science and/or Software Engineering and with more than 4 years of Java programming experience.

Prior to the study, users are informed in the definition of syntax and semantic coding style. Syntactic style includes common formats and structures; semantic style includes aspects such as data flow and meaningful naming. Samples are randomly selected from a pool of test data. To simplify the decision-making process for users, all questions present them with a choice between two options. Users are asked to answer the question: "Which of the two code copies is closer in coding style to the reference code?", and every user is provided with three options (i.e., A is Better, B is Better, Cannot Determine/Both Equally). It is worth mentioning that the users do not know which code snippet is generated by which method. we provide an example of the question as shown in Fig 9.

A.5 Details for Adapting to New Users

Scenario 1. In this scenario, we can utilize both full training and incremental training to update the

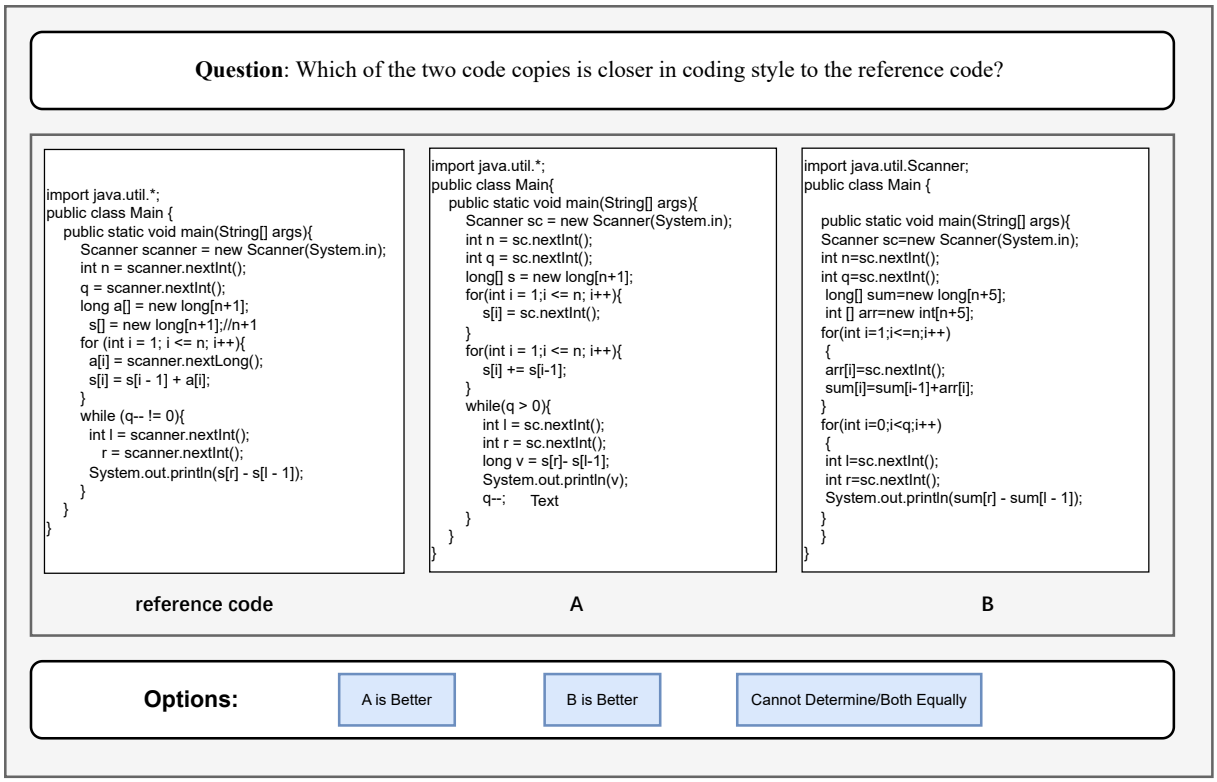

Figure 9: An example of the human study.

model. For the incremental training approach, we freeze the model parameters and only train the implicit style features of the new user based on his historical records. Due to time constraints, we have conducted a preliminary verification comparing the results of full training versus incremental training for a single new user in Table 8. Although the effectiveness of incremental training is slightly inferior to that of full training, this approach avoids the need to retrain the entire model, thus showcasing the feasibility and efficiency of our proposed approach for adding new users.

Method	CSS	BLUE	Rouge-1	Rouge-2
Full training	60.92	63.39	52.21	39.25
Incremental training	58.21	63.35	51.65	38.28

Table 8: Incremental training on PCIDense.

Scenario 2. In this scenario, we can use MPCODER$_{\text{ESF}}$ which only utilizes explicit style attributes for inference. Since explicit attributes do not depend on specific user data, the user only needs to specify the corresponding explicit coding style attribute or opt for a default setting. Consequently, the model can generate code that aligns with the user's syntax style. The experimental results are shown in the table 2. Although MPCODER$_{\text{ESF}}$ may not be as efficient as MPCODER and is focused primarily on syntax style, it does not require any training for new users.

B Evaluation Statement

B.1 Correctness Evaluation

Models trained on PCISparse and PCIDense are evaluated on a dataset of human-x code correctness tests. The correctness of the first reply code of the model is tested by greedy decoding. On PCIDense dataset, we fully test all problems in HumanEval-X for each user and report the average values based on three prompt templates. Because there are too many users in PCISparse data, we randomly select 50 users as the object of correctness verification. The prompt template for the correctness test is shown in Fig. 8.

B.2 Java Style Criteria

As shown in Table 7, we select 24 code criteria that can reflect the coding style of Java programming problems from three aspects: structure, naming, and formatting. It is important to note that the proposed CSS metric incorporates 20 style criteria for identifying syntax style and 4 style criteria for identifying semantic style. For example, the attributes "FallThrough" and "MissingSwitchDefault" are utilized to detect differences in the code

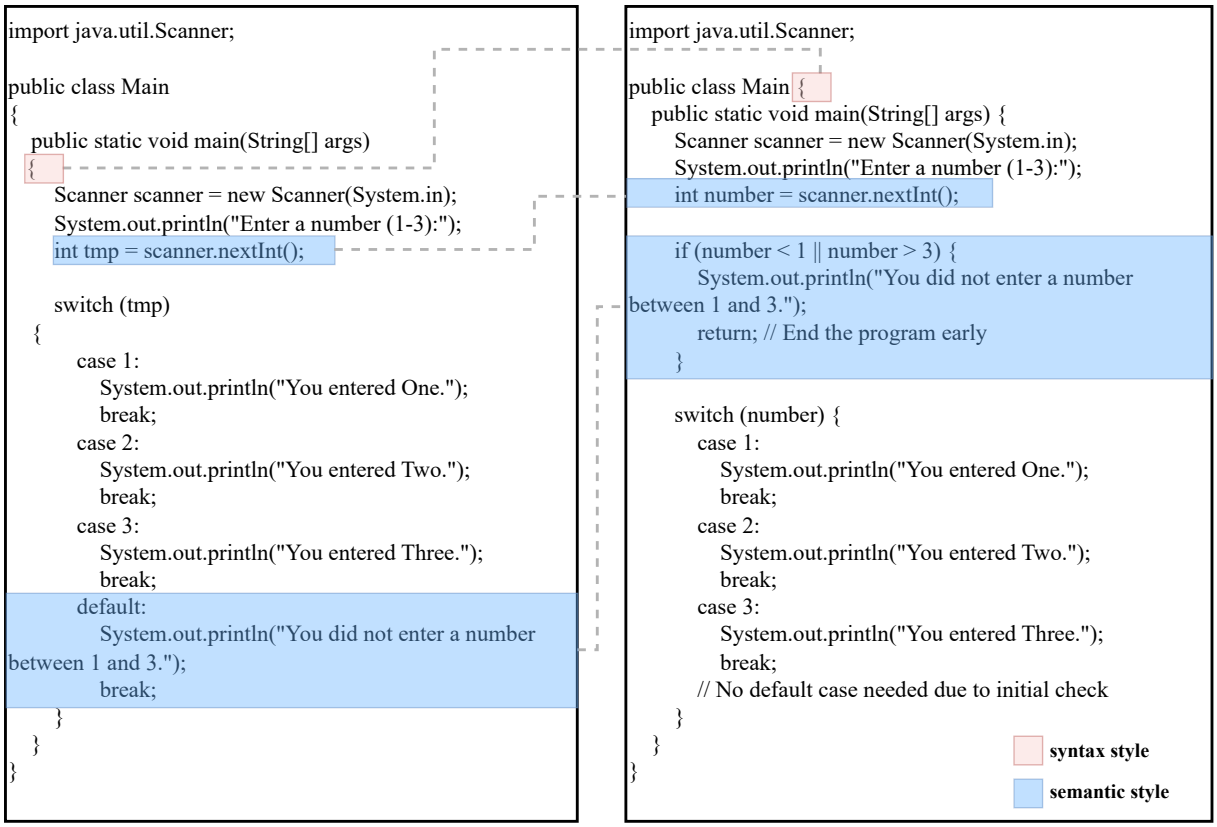

Figure 10: Syntax and Semantic Coding Styles.

execution order. Both of which pertain to the semantic style of control flow and data flow (Li et al., 2022b). Specifically, “FallThrough” means “Do not put a fall-through comment in a switch If a ’case’ has no break, return, throw, or continue”; “MissingSwitchDefault” means “switch statement does not have a default clause”. In other words, the code style attributes checked by CheckStyle contain both syntax features and semantic features, and our CSS evaluation metric using these features can estimate model performance from both syntactic and semantic perspectives.

B.3 Syntax and Semantic Coding Styles

Fig. 10 shows an example of the syntax and semantic differences in coding styles. Both copies of the code solve the same problem, but the code reflects different syntax and semantic styles.

Syntax style differences. The curly braket "{" in the left code copy is placed on the same line as the preceding statement, while the right code copy places it on a separate new line, which reflects the layout style difference in the format.

Semantic style differences. The left code copy uses "tmp" as a temporary variable name, while the right code copy uses "number" as a numerical variable name, reflecting different semantic styles in naming conventions. The left code copy uses default in the Switch statement, while the right code copy does not use default. Due to the different code control flows, the order of actual program execution may also be different. The left code may run to the end of the default statement, and the right code could output and terminate the program at the beginning. The two pieces of code represent different data flows, or design patterns, that reflect the semantic coding styles of different developers.

B.4 Example of Model Performance

Figure 11 shows the personalized code generation results of the same question for the same user regarding different models. Significant coding styles are highlighted in blue, showing that the generation result of our model aligns more closely with the overall style (including syntax style and semantic style) of the reference code than those of Adapter and L-LDB.

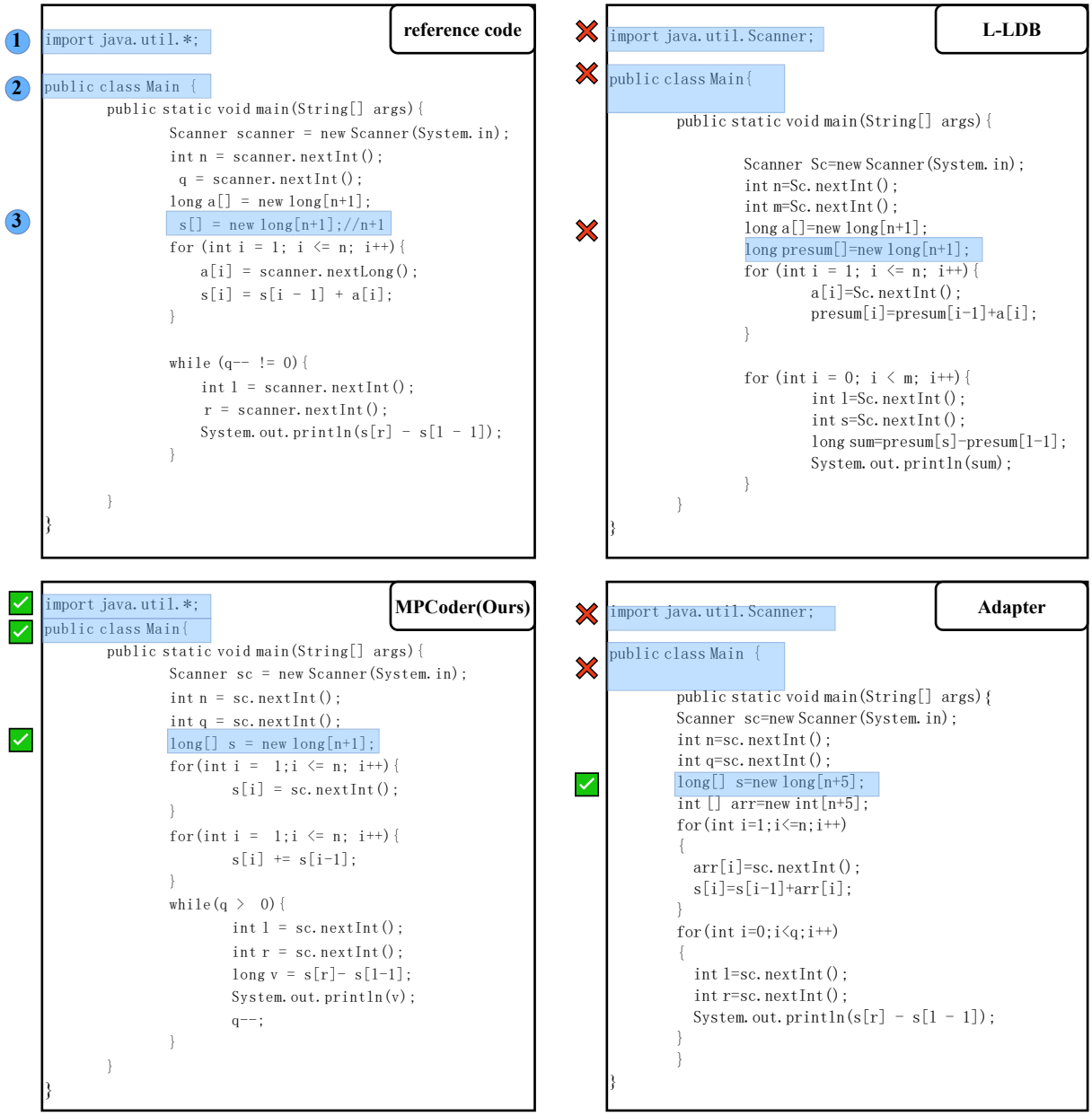

Figure 11: Syntax and Semantic Coding Styles.