



Run Run Shaw Library

香港城市大學
City University of Hong Kong

Copyright Warning

Use of this thesis/dissertation/project is for the purpose of private study or scholarly research only. ***Users must comply with the Copyright Ordinance.***

Anyone who consults this thesis/dissertation/project is understood to recognise that its copyright rests with its author and that no part of it may be reproduced without the author's prior written consent.

CITY UNIVERSITY OF HONG KONG

香港城市大學

Advancing Code Intelligence via Efficiency
and Adaptability: Insights, Methods, and
Applications

通過效率與適應性推進代碼智能：洞察、方
法與應用研究

Submitted to

Department of Computer Science

計算機科學系

in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy

哲學博士學位

by

LIU Shuo

劉碩

AUGUST 2026

二零二六年八月

Declaration

I declare that the thesis entitled “*Advancing Code Intelligence via Efficiency and Adaptability: Insights, Methods, and Applications*” represents my own work, except where due acknowledgments are made, and that it has not been previously included in a thesis, dissertation, or report submitted to this university or any other institution for a degree, diploma, or other qualifications.

LIU Shuo
AUGUST 2026

Abstract

Code intelligence refers to the capability to comprehend, analyze, and generate source code within the domain of intelligent software engineering. Its primary objective is to enhance both programming productivity and software quality. Benefiting from the robust representation capabilities, Code Language Models (Code LMs) can capture intricate semantic and structural knowledge, enabling high performance across various downstream tasks. Nevertheless, while code intelligence has achieved significant milestones, computational efficiency and scenario-specific adaptability remain two pivotal hurdles to its further application in real-world scenarios. On one hand, state-of-the-art code LMs predominantly follow the “pre-training and fine-tuning” paradigm. This approach, however, demands prohibitive computational overhead, creating a high barrier to entry for many researchers. On the other hand, given the sophisticated nature of downstream scenarios, how to seamlessly adapt code LMs to diverse scenarios, such as dynamic code changes and continual learning, remains a paramount challenge.

In this thesis, centered around the two pivotal dimensions of efficiency and adaptability, we systematically investigate the optimization and application of code LMs from four perspectives: (1) high-efficiency fine-tuning paradigms, (2) exploration of efficient model architectures, (3) dynamic adaptive learning strategies, and (4) structure-aware adaptive learning approaches. Based on experimental insights obtained, we propose a variety of innovative methods to bridge the gap between theoretical model capabilities and practical application requirements.

Specifically, to mitigate the high computational overhead associated with Full-Model Fine-Tuning (FMFT), we first investigate Parameter-Efficient Fine-Tuning (PEFT) in the context of code-change-related tasks. Unlike static code comprehension that naturally aligns with pre-training paradigms, thereby facilitating seamless knowledge transfer, dynamic code change introduces unique complexity as it involves reasoning over modified code pairs, leaving it unclear whether PEFT can offer superior performance

over FMFT in such specialized domains. To address this question, we perform a rigorous comparative analysis between FMFT and four prominent PEFT techniques (Adapter Tuning, LoRA, Prompt Tuning, and Prefix Tuning) across two widely studied code-change-related tasks. Our evaluation spans diverse experimental settings, including common, cross-lingual, and low-resource scenarios. Furthermore, we conduct multi-dimensional probing tasks from both static and dynamic perspectives to provide a granular explanation of their respective strengths. Inspired by the layer-wise insights gained from these probing tasks, we propose Pasta_K, a self-adaptive framework for efficient, layer-specific tuning in code change learning. It harmonizes the distinct advantages of FMFT and PEFT, delivering robust performance while maintaining high parameter efficiency.

While the Transformer and its core attention mechanism have driven the remarkable success of Pre-trained Language Models (PLMs) in code intelligence, the quadratic complexity of self-attention poses a substantial scalability barrier. This limitation has catalyzed the emergence of sub-quadratic alternatives, such as Mamba and Mamba-2. Consequently, beyond exploring fine-tuning paradigms, we further investigate efficient model architectures and present the first systematic empirical study of Mamba-based PLMs in the field of code intelligence. By pre-training two Mamba-based models on a large-scale code corpus and benchmarking them against representative Transformer-based baselines under both FMFT and PEFT settings, we demonstrate their overall superiority in efficacy and efficiency across tasks such as code completion, code generation, and code clone detection. Subsequent experiments involve architectural analysis of the Transformer, Mamba, and Mamba-2 by employing from-scratch pre-training to isolate the influence of different training corpora. Additionally, we assess the models’ robustness through low-resource scenarios by deliberately constraining the fine-tuning data volume. We further extend our experiments to larger scales (up to 7B parameters), providing a comprehensive comparison with a broader range of state-of-the-art Transformer-based baselines. By identifying the specific advantages and limitations of Mamba-based PLMs, we offer critical insights into their potential for future deployment in code intelligence.

Regarding adaptability in evolving environments, we investigate the Continual Learning (CL) scenario, as software programs undergo constant iteration in practice and trigger frequent domain shifts. In contrast to traditional offline fine-tuning, which operates on static, consolidated datasets, CL involves learning from a sequence of datasets arriving incrementally. We criticize that existing rehearsal-based CL methods

rely heavily on the storage and retraining of historical samples, which introduces extra training overhead and the risk of data disclosure. To address these issues, we leverage the superiority of prompts in eliciting pre-trained knowledge to realize a rehearsal-free method. Our preliminary exploration of vanilla prompt tuning in CL scenarios reveals that inheritance of PLM parameters from previous stages is beneficial, and prompt stability should be emphasized. Therefore, we propose an effective method named Prompt Tuning with Domain-wise Distillation (PTDD), which can distill prompts and optimize PLMs with a two-sided learning objective, thus improving PLMs’ performance in diverse domains. To the best of our knowledge, we conduct the first systematic experimental analysis of prompt tuning’s performance within the CL setting in the field of code intelligence.

Finally, we focus on the termination analysis task, which is a critical component for ensuring software reliability and availability. We argue that existing methods are hindered by their reliance on complex sandbox environments for execution or a lack of fine-grained analysis. Therefore, we propose TerGEC, a graph-enhanced contrastive approach for precise, execution-free termination detection. TerGEC leverages the structural and semantic information from Abstract Syntax Trees (ASTs) of programs, capturing intra-class semantics by analyzing program behaviors at both the syntactic and lexical levels. Simultaneously, it employs contrastive learning to distinguish the boundaries between termination and non-termination, thereby mastering inter-class semantics. To bolster the robustness, we incorporate graph augmentation strategies alongside weighted contrastive loss and focal loss, specifically designed to alleviate the class imbalance problem inherent in non-termination detection. Consequently, TerGEC achieves a more fine-grained analysis while entirely obviating the need for execution.

Overall, this thesis presents a series of novel methodologies and comprehensive empirical evaluations dedicated to advancing code intelligence through the dual lenses of efficiency and adaptability. By synthesizing our proposed frameworks with deep experimental insights, we aim to make meaningful contributions to the advancement of code LMs and facilitate their efficient application and robust performance in practical software engineering scenarios.

Keywords: Code Intelligence, Code Change Learning, Parameter-Efficient Fine-Tuning, Mamba, Continual Learning, Termination Analysis

CITY UNIVERSITY OF HONG KONG
Qualifying Panel and Examination Panel

Surname:	LIU
First Name:	Shuo
Degree:	Doctor of Philosophy
College/Department:	Department of Computer Science

The Qualifying Panel of the above student is composed of:

Supervisor(s)

Prof. KEUNG Wai Jacky	Department of Computer Science City University of Hong Kong
-----------------------	--

Qualifying Panel Member(s)

Prof. WANG Shiqi	Department of Computer Science City University of Hong Kong
------------------	--

Prof. SONG Linqi	Department of Computer Science City University of Hong Kong
------------------	--

This thesis has been examined and approved by the following examiners:

Prof. KEUNG Wai Jacky	Department of Computer Science City University of Hong Kong
-----------------------	--

Prof. WONG Hau San Raymond	Department of Computer Science City University of Hong Kong
----------------------------	--

Prof. CHAN Wing Kwong Ricky	Department of Computer Science City University of Hong Kong
-----------------------------	--

Prof. Kla Tantithamthavorn	Department of Software Systems & Cybersecurity Monash University
----------------------------	---

Acknowledgments

Looking back on my PhD journey, I am grateful to everyone who has offered their unwavering support along the way. First and foremost, I would like to express my deepest and most sincere gratitude to my supervisor, Prof. Jacky Keung, for his invaluable guidance and support. His profound academic insights and visionary perspectives have not only steered my doctoral research away from pitfalls but also left an indelible imprint on my intellectual and personal maturity. Additionally, his meticulous supervision and genuine care have been a constant source of strength, empowering me to bring my endeavor to a successful and confident completion. I will forever treasure that summer four years ago when Prof. Jacky Keung offered me the opportunity to join his research group. It is under his inspiring guidance that I have cultivated the academic vision to explore the frontiers of software engineering and artificial intelligence.

I would also like to extend my sincere thanks to my PhD qualifying panel members, Prof. Shiqi Wang and Prof. Linqi Song, for their valuable time and rigorous evaluation. Their insightful feedback and constructive comments have significantly improved the quality of my work.

Special appreciation goes to my collaborators for their dedication and expertise. Specifically, I am deeply grateful to Dr. Zhen Yang for the countless hours spent refining our papers, a process enriched by our stimulating discussions and frequent brainstorming sessions. His sharp insights and rigorous attitude toward research have set an exemplary standard for me.

I am also immensely grateful to the seniors I met in the research group, Dr. Xiao Yu, Dr. Yan Xiao, Dr. Shuo Feng, Dr. Xiaoxue Ma, Dr. Yishu Li, and Dr. Jingyu Zhang, for their invaluable advice and support throughout my journey. My sincere thanks also go to my colleagues and friends in the lab: Yicheng Sun, Fengji Zhang, Guancheng Lin, Zhenyu Mao, Yihan Liao, and many others. Beyond our academic collaborations, I deeply cherish the laughter and unforgettable memories we shared

outside the laboratory. I look forward to a lifelong friendship as we all step into our bright futures.

Last but certainly not least, my ultimate gratitude belongs to my family. To my parents, thank you for your unconditional love, constant encouragement, and for always believing in my potential. Your profound understanding and boundless support have been my greatest pillar of strength. A special and heartfelt note of gratitude goes to my girlfriend, Chenning Du, whose love, companionship, and care have brought me immense comfort and peace of mind.

*The past is prologue,
And the future awaits exploration.*

Publications

Published Journal Papers

1. **Shuo Liu**, Jacky Keung, Zhi Jin, Zhen Yang, Fang Liu, Hao Zhang. An Empirical Study of Parameter-Efficient Fine-Tuning in Code Change Learning and Beyond. In *IEEE Transactions on Software Engineering (TSE)*, 52(1), 3-21, 2025.
2. **Shuo Liu**, Jacky Keung, Zhen Yang, Fang Liu, Fengji Zhang, Yicheng Sun. Exploring Continual Learning in Code Intelligence with Domain-Wise Distilled Prompts. In *Information and Software Technology (IST)*, 185, 2025.
3. **Shuo Liu**, Jacky Keung, Zhen Yang, Yihan Liao, Yishu Li. TerGEC: A Graph Enhanced Contrastive Approach for Program Termination Analysis. In *Science of Computer Programming (SCP)*, 237, 2024.
4. Zhen Yang, Hongyi Lin, Xiao Yu, Jacky Keung, **Shuo Liu***, Pak Yuen Patrick Chan, Yicheng Sun, Fengji Zhang. *R²ComSync*: Improving Code-Comment Synchronization with In-Context Learning and Reranking. In *Empirical Software Engineering (EMSE)*, 31(4), 2026.
5. Yicheng Sun, Jacky Keung, Zhen Yang, **Shuo Liu**, Hi Kuen Yu. SemiRALD: A Semi-Supervised Hybrid Language Model for Robust Anomalous Log Detection. In *Information and Software Technology (IST)*, 183, 2025.
6. Zhen Yang, Jacky Keung, Zeyu Sun, Yunfei Zhao, Ge Li, Zhi Jin, **Shuo Liu**, Yishu Li. Improving Domain-Specific Neural Code Generation with Few-Shot Meta-Learning. In *Information and Software Technology (IST)*, 166, 2024.
7. Yicheng Sun, Jacky Keung, Zhen Yang, **Shuo Liu**, Yihan Liao. SemiSMAC: A Semi-Supervised Framework for Log Anomaly Detection with Automated Hyperparameter Tuning. In *Information and Software Technology (IST)*, 187, 2025.

8. Yishu Li, Jacky Keung, Zhen Yang, Xiaoxue Ma, Jingyu Zhang, **Shuo Liu**. SimAC: Simulating Agile Collaboration to Generate Acceptance Criteria in User Story Elaboration. In *Automated Software Engineering (ASE)*, 31(2), 2024.
9. Yicheng Sun, Jacky Keung, Zhen Yang, **Shuo Liu**, Hi Kuen Yu. Improving Anomaly Detection in Software Logs Through Hybrid Language Modeling and Reduced Reliance on Parser. In *Automated Software Engineering (ASE)*, 33(1), 2026.
10. Yihan Liao, Jacky Keung, Jingyu Zhang, Yurou Dai, **Shuo Liu**. Evaluate Inference Attacks: Attack and Defense against 2D Semantic Segmentation Models. In *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 2025.

Published Conference Papers

1. **Shuo Liu**, Jacky Keung, Zhen Yang, Zhenyu Mao, Yicheng Sun. Can Mamba Be Better? An Experimental Evaluation of Mamba in Code Intelligence. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov, 2025.
2. **Shuo Liu**, Jacky Keung, Zhen Yang, Fang Liu, Qilin Zhou, Yihan Liao. Delving into Parameter-Efficient Fine-Tuning in Code Change Learning: An Empirical Study. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Mar, 2024.
3. Xinrui He, **Shuo Liu**, Jacky Keung, Jingrui He. Co-Clustering for Federated Recommender System. In *International World Wide Web Conference (WWW)*, May, 2024.
4. Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Keung, Jia Li, **Shuo Liu**, Yifan Hong, Xiaoxue Ma, Zhi Jin, Ge Li. Exploring and Unleashing the Power of Large Language Models in Automated Code Translation. In *ACM International Conference on the Foundations of Software Engineering (FSE)*, Jul, 2024.
5. Zhen Yang, Hongyi Lin, Yifan He, Junqi Wang, Zeyu Sun, **Shuo Liu**, Jie Xu, Pengpeng Wang, Zhongxing Yu, Qingyuan Liang. Contamination Means Over-estimation? A Fine-grained Empirical Study in Code Intelligence. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Oct, 2026.

6. Yicheng Sun, Jacky Keung, Hi Kuen Yu, **Shuo Liu**, Yihan Liao, Jingyu Zhang. Beyond Log Parsers: A Scalable AI-Driven Framework for Efficient Log Anomaly Detection in Software Engineering. In *IEEE International Conference on Computers, Software, and Applications (COMPSAC)*, Jul, 2025.
7. Yicheng Sun, Jacky Keung, Jingyu Zhang, Hi Kuen Yu, Wenqiang Luo, **Shuo Liu**. Unveiling Hidden Anomalies: Leveraging SMAC-LSTM for Enhanced Software Log Analysis. In *IEEE International Conference on Computers, Software, and Applications (COMPSAC)*, Jul, 2024.
8. Zhenyu Mao, Jacky Keung, Fengji Zhang, **Shuo Liu**, Yifei Wang, Jialong Li. Towards Engineering Multi-Agent LLMs: A Protocol-Driven Approach. In *Asia-Pacific Software Engineering Conference (APSEC)*, Dec, 2025.

Under-Reviewed Journal Papers

1. **Shuo Liu**, Hao Ning, Jacky Keung, Zhen Yang, Xiao Yu, Xing Hu, Yicheng Sun. SynMOE: A Syntax-Aware MOE-Guided Approach for PEFT-Based Code Intelligence Learning. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*. [Under Review]

Contents

Abstract	iii
Qualifying Panel and Examination Panel	vi
Publications	ix
List of figures	xvii
List of tables	xix
1 Introduction	1
1.1 Research Background	1
1.2 Parameter-Efficient Fine-Tuning in Code Change Learning	2
1.3 Mamba in Code Intelligence	5
1.4 Continual Learning in Code Intelligence	7
1.5 Program Termination Analysis	10
1.6 Contributions of the Thesis	13
1.7 Organization of the Thesis	14
2 Literature Review	16
2.1 Code Change Learning	16
2.2 Termination Analysis	17
2.3 Transformer-based Code Language Models	19

2.4	Mamba and Mamba-based Models	19
2.5	Continual Learning	20
2.6	GNN and Graph Contrastive Learning	22
3	Delving into Parameter-Efficient Fine-Tuning in Code Change Learning	24
3.1	Preliminary	25
3.1.1	Full-Model Fine-Tuning	25
3.1.2	Adapter Tuning (AT)	25
3.1.3	Low-Rank Adaptation (LoRA)	27
3.1.4	Prompt Tuning (PT)	28
3.1.5	Prefix Tuning (PreT)	28
3.1.6	Training-free Methods	29
3.2	Experimental Setup	29
3.2.1	Research Questions	29
3.2.2	Tasks and Datasets	31
3.2.3	Pre-trained Language Models and Baselines	34
3.2.4	Evaluation Metrics	36
3.2.5	Implementation Details	36
3.3	Experimental Results	37
3.3.1	RQ1: Quantitative Evaluation	37
3.3.2	RQ2: Performance in the Cross-lingual Scenario	45
3.3.3	RQ3: Performance in the Low-resource Scenario	47
3.3.4	RQ4: Probing Tasks	51
3.3.5	RQ5: Self-adaptive Efficient Layer-specific Tuning	54
3.4	Discussion	58
3.4.1	Implications	58
3.4.2	Threats to Validity	59

3.5	Chapter Summary	60
4	Unveiling the Efficacy and Efficiency of Mamba in Code Intelligence	61
4.1	Preliminary	62
4.1.1	Transformer	62
4.1.2	State Space Models, Mamba, and Mamba-2	62
4.1.3	LoRA and $(IA)^3$	64
4.2	Experimental Setup	65
4.2.1	Research Questions	65
4.2.2	Tasks and Datasets	67
4.2.3	Pre-trained Language Models	68
4.2.4	Evaluation Metrics	69
4.2.5	Implementation Details	70
4.3	Experimental Results	70
4.3.1	RQ1: Performance of Full Fine-Tuning	70
4.3.2	RQ2: Performance of Parameter-Efficient Fine-Tuning	73
4.3.3	RQ3: Advantages of Mamba Architecture	74
4.3.4	RQ4: Evaluation in the Low-Resource Scenario	77
4.3.5	RQ5: Evaluation with Larger Scale	79
4.4	Discussion	80
4.4.1	Case Study	80
4.4.2	Human Evaluation	82
4.4.3	Threats to validity	83
4.5	Chapter Summary	84
5	Exploring Continual Learning in Code Intelligence with Domain-wise Dis- tilled Prompts	85
5.1	Preliminary	86

5.1.1	Continual Learning Protocols	86
5.1.2	Continual Learning with Fine-Tuning and Prompt Tuning	87
5.2	Methodology	88
5.2.1	Prompt Embedding	89
5.2.2	Distilled Prompt Learning	89
5.3	Experimental Setup	91
5.3.1	Tasks and Datasets	91
5.3.2	Baselines	92
5.3.3	Evaluation Metrics	94
5.3.4	Implementation Details	94
5.4	Experimental Results	95
5.4.1	RQ1: Vanilla Prompt Tuning Performance	95
5.4.2	RQ2: Comparison with Baselines	96
5.4.3	RQ3: Ablation Study	99
5.4.4	RQ4: Parameter Analysis	101
5.4.5	RQ5: Performance in the Low-resource Scenario	102
5.5	Discussion	104
5.5.1	Visualization for PTDD	104
5.5.2	Case Study	105
5.5.3	Training Cost Analysis	106
5.5.4	Implications	106
5.5.5	Threats to Validity	108
5.6	Chapter Summary	109
6	TerGEC: A Graph Enhanced Contrastive Approach for Program Termination Analysis	110
6.1	Preliminary	111
6.1.1	Notations	111

6.1.2	GNN layers	111
6.2	Research Methods	113
6.2.1	Graph Augmentation	113
6.2.2	Graph Encoder	115
6.2.3	Intra-class Learning	116
6.2.4	Inter-class Learning	117
6.3	Data Preparation	119
6.3.1	Preprocessing	119
6.3.2	Data Transformation	120
6.4	Evaluation	121
6.4.1	Experiment Setting	121
6.4.2	RQ1: Quantitative Evaluation	124
6.4.3	RQ2: Role of Graph Augmentation	125
6.4.4	RQ3: Backbone Analysis	126
6.4.5	RQ4: Effectiveness of Balance Factors	128
6.5	Discussion	129
6.5.1	Limitation	129
6.5.2	Threats to Validity	129
6.6	Chapter Summary	130
7	Conclusion and Future Work	131
7.1	Conclusion	131
7.2	Future Work	133
	References	136

List of figures

1.1	Representative code intelligence tasks related to static and dynamic code snippets	3
1.2	An illustration of the continual learning scenario. X marks indicate the domain ID that test samples derive from is unknown when testing . . .	8
3.1	Illustrations on FMFT, AT, LoRA, PT, and PreT	26
3.2	Illustrations of probing tasks	33
3.3	Comparison of training time and memory size in the CMG task, using CodeT5 as the backbone	44
3.4	Hyperparameter analysis of the LoRA rank, the intermediate dimension for AT, and the prompt length for PT and PreT	45
3.5	Accuracy of probing tasks by layers	53
4.1	Illustrations of the State Space Model, Mamba, and Mamba-2	63
4.2	The overall workflow. There are a total of five research questions . . .	66
4.3	Utilization of computational resources	75
4.4	Evaluation results in the low-resource scenario	78
4.5	Case study in the CONCODE dataset of the code generation task . . .	82
5.1	Overview of the proposed Prompt Tuning with Domain-wise Distillation (PTDD)	88
5.2	Performance recorded by the dataset order	98
5.3	Parameter analysis of the prompt length and λ value	101

5.4	Prompt visualization in a representation space. We mark out the order of each prompt in its upper right corner	104
5.5	Examples of summarizations outputted by PTDD and Fine-Tuning . .	107
6.1	The overall framework of TerGEC	114
6.2	An example of AST for Python	114

List of tables

3.1	Dataset Statistics	31
3.2	Evaluation results of JIT-DP on the JIT-Defects4J dataset	39
3.3	Evaluation results of CMG on the MCMD dataset. M is short for Meteor. R is short for Rouge-L. Digits in bold denote the best value among all results. Table cells in a light grey background denote the best result among different approaches under the same backbone model	43
3.4	Evaluation results of CMG in the cross-lingual scenario. CodeT5 is used as the backbone model for FMFT, LoRA, AT, PT, and PreT	46
3.5	Evaluation results of CMG in the low-resource scenario. M is short for Meteor. R is short for Rouge-L. CodeT5 is used as the backbone model for FMFT, LoRA, AT, PT, and PreT	48
3.6	Evaluation results of JIT-DP in the low-resource scenario. CodeBERT is used as the backbone model	50
3.7	Evaluation results of $Pasta_K$ in the CMG task. We use the Java dataset for evaluation. The improving ratios of $Pasta_K$ on performance are compared with corresponding PEFT methods, while on consumption are compared with FMFT	55
3.8	Evaluation results of $Pasta_K$ in the JIT-DP task	57
4.1	Dataset Statistics	67
4.2	Evaluation results in the FT and PEFT settings. The Mamba and Mamba-2 used here are 130M-sized	72

4.3	Model performance when pre-trained from scratch. The Mamba and Mamba-2 used here are 130M-sized	76
4.4	Evaluation results of larger-scale models using FT during adaptation .	76
4.5	Model performance in human evaluation	83
5.1	Evaluation results of code summarization. M and R are short for Meteor and Rouge-L, respectively	97
5.2	Evaluation results of code vulnerability detection and code clone detection	98
5.3	Evaluation results in the low-resource scenario. M and R are short for Meteor and Rouge-L, respectively	103
5.4	Training cost analysis of PTDD and other representative CL methods .	106
6.1	Statistics for datasets	122
6.2	Experimental results for the baselines and TerGEC	122
6.3	Experimental results about graph augmentation	127
6.4	Experimental results with different backbones	127
6.5	Experimental results about balance factors	127

Chapter 1

Introduction

1.1 Research Background

In the contemporary era of software engineering, highly automated and intelligent development workflows have become indispensable for enhancing programmer productivity and ensuring software reliability. Consequently, code intelligence has emerged as a critical cornerstone, dedicated to programmatically comprehending, analyzing, and generating source code. It has demonstrated profound efficacy across various downstream tasks, such as code completion [77, 79, 99], code summarization [61, 180, 241], and vulnerability detection [153, 172, 178]. The recent breakthroughs in code intelligence are primarily propelled by the rapid evolution of Code Language Models (Code LMs). Through the classical “pre-training and fine-tuning” paradigm, code LMs demonstrate powerful representation capabilities, effectively capturing the intricate semantic and structural knowledge inherent in programs, thereby establishing superior performance benchmarks.

Nevertheless, despite these remarkable successes, critical bottlenecks still hinder code LMs from seamless integration into real-world application environments. This research gap is primarily underscored by two pivotal hurdles: computational efficiency and scenario-specific adaptability. Regarding efficiency, since state-of-the-art (SOTA) code LMs predominantly adhere to the “pre-training and fine-tuning” paradigm, full parameter fine-tuning demands prohibitive computational overhead and massive memory footprints. These intensive resource requirements create a daunting barrier to entry for independent researchers and resource-constrained practitioners. Regarding adaptability, the inherently dynamic, sophisticated, and evolving nature of actual software

scenarios causes static code LMs to frequently struggle with seamlessly adapting to diverse target domains. Consequently, how to elegantly infuse domain-specific knowledge into code LMs to achieve agile, context-aware adaptation remains a paramount yet open challenge.

To bridge these gaps, this thesis centers on the twin dimensions of efficiency and adaptability to pioneer advanced frameworks for code intelligence. Specifically, the optimization and practical application of code LMs are systematically explored across four perspectives:

1. High-Efficiency Fine-Tuning Paradigms: Investigating parameter-efficient fine-tuning approaches to enable superior downstream adaptation.
2. Exploration of Efficient Model Architectures: Going beyond the traditional Transformer to comprehensively evaluate the novel Mamba architecture.
3. Dynamic Adaptive Learning Strategies: Evaluating the capabilities of code LMs in the continual learning scenario to mitigate catastrophic forgetting.
4. Structure-Aware Adaptive Learning Approaches: Leveraging the structural and semantic information embedded in Abstract Syntax Trees (ASTs) to formulate fine-grained code representations.

Through deep dissections of these crucial topics, we propose a suite of innovative methods to render the application of code LMs highly practical and adaptive within the realm of software engineering.

1.2 Parameter-Efficient Fine-Tuning in Code Change Learning

Pre-trained Language Models (PLMs), e.g., CodeBERT [55] and CodeT5 [207], have been extensively utilized in various software engineering tasks, including code generation [142, 222] and code comprehension [76, 110], having achieved notable advancements [219]. These models follow a prevalent paradigm where they are initially pre-trained on large-scale corpora using Masked Language Modeling (MLM) or Next Token Prediction (NTP) [43], and subsequently fine-tuned for specific downstream tasks. Nonetheless, Full-Model Fine-Tuning (FMFT) for PLMs or even LLMs prohibitively relies on enormous computational resources, rendering it unsuitable for most users.

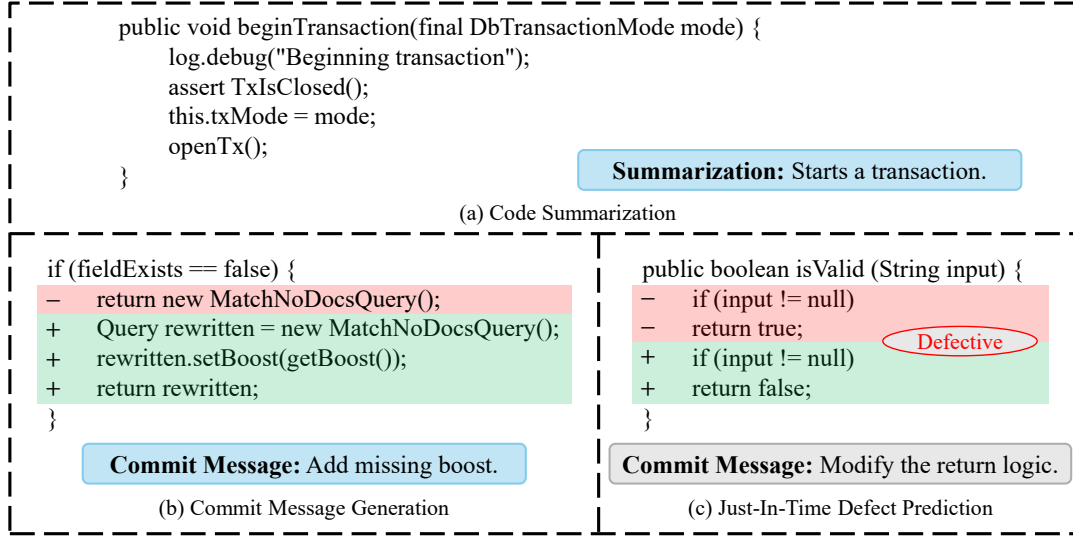


Figure 1.1. Representative code intelligence tasks related to static and dynamic code snippets.

Therefore, a series of Parameter-Efficient Fine-Tuning (PEFT) methods [94, 95, 120, 122], updating only 0.01%-5% of the whole parameters while keeping the others frozen, have been proposed in recent years and gained much attention [44, 59, 211].

Previous studies [170, 199] revealed the superiority of PEFT against FMFT in code search, code clone, and code summarization tasks. Because PEFT greatly alleviates the catastrophic forgetting problem, it can efficiently harness the pre-trained knowledge for the downstream tasks [68]. However, it is noted that most of the investigation objects of the above literature are static code comprehension tasks, where code snippets are learned in only a single status. For example, code summarization, shown in Figure 1.1(a), is a typical task of this kind, whose input is a code snippet of a single status while producing a code comment as an output. Apparently, this is consistent with the paradigm that PLMs learned during their pre-training phase. As such, we conjecture that adapting PLMs to static code comprehension tasks inherently needs fewer parameter tuning. In contrast, dynamic code comprehension tasks (e.g., Figure 1.1(b) and (c)), such as Commit Message Generation (CMG) and Just-In-Time Defect Prediction (JIT-DP), differ significantly from the above static ones. They contain code changes (i.e., code *diffs*) within the model inputs, where both old and new versions of code snippets co-occur, inadvertently enlarging the transferring gap between the pre-training and adaptation phases [127, 197]. The above code change-related tasks are ubiquitous in developers' daily work and are important for project maintenance, as developers need

to debug with code changing and review code *diff*s submitted by other developers for collaboration. As such, it is necessary to explore the effect of PEFT on code change learning, thereby seeking the potential for further improvement, and probing its application discrepancies against static code learning.

In this work, we choose four mainstream PEFT methods for experiments, namely Adapter Tuning (AT) [85, 94, 160], Low-Rank Adaptation (LoRA) [95], Prompt Tuning (PT) [120], and Prefix Tuning (PreT) [122]. AT introduces adapter modules, each of which is a bottleneck structure, and inserts them between layers of PLMs. In the fine-tuning stage, the parameters of the original PLMs are fixed, and only adapter modules are adjusted. LoRA injects trainable rank decomposition matrices into PLMs to substitute the original pre-trained weights, which can amplify some hidden features encoded during pre-training [95]. PT and PreT leverage the power of prompting in eliciting PLMs' pre-trained knowledge [22]. The former adds trainable vectors as prompts to the original input, but the latter introduces extra trainable parameters to the hidden states. To evaluate their performance on code change learning, we conduct experiments for JIT-DP [239] and CMG [40]. The former is a classification task aiming to predict whether a code change is defect-prone or not, and the latter is a generation task targeting to automatically generate a commit message given changed code snippets.

Specifically, (i) we first compare the performance of PEFT methods to FMFT and State-Of-The-Art (SOTA) approaches with full-data settings, namely the common scenario. Experimental results show that, in the JIT-DP task, all four PEFT methods can surpass FMFT and even SOTA approaches when incorporating extra expert features, with AT and LoRA obtaining improvements of 5.08% and 9.75% in F1 over SOTA baselines. In the CMG task, AT and LoRA achieve comparable performance to FMFT and SOTA baselines, while PT and PreT perform worse. Given their lower training time and memory overhead, AT and LoRA remain practical and acceptable. (ii) Besides, to evaluate how well PEFT methods transfer knowledge and generalize across Programming Languages (PLs), we explore their capabilities in the cross-lingual scenario, where PLMs are fine-tuned in one PL but are tested in another. Evaluation results show that PEFT methods perform either better or comparable to FMFT. (iii) To understand how well PEFT performs in low-resource scenarios, we scaled down the fine-tuning datasets while keeping the testing samples the same for experiments. Evaluation results reveal that almost all PEFT methods fall short of FMFT on the CMG task, whereas they exhibit overall superiority or are on par with FMFT on the JIT-DP task in the w/ EF setting. (iv) Furthermore, we utilize three probing tasks, namely invalid

TYPE detection (TYP), Code Change Match (CCM), and Line Type Prediction (LTP), to investigate the layer-wise encoded code semantics, expecting to make an explanation for the performance of PEFT and FMFT. TYP relates to static code semantics, while CCM and LTP are associated with dynamic code semantics from global and local perspectives, respectively. The average probing results are consistent with previous findings and explain the efficacy of PEFT techniques and FMFT. However, PEFT and FMFT demonstrate distinctive code-change learning abilities among different layers of PLMs.

Inspired by the distinctive advantages of PEFT and FMFT in their layer-wise probing results, we propose **Pasta_K**, which stands for self-adaptive efficient layer-specific tuning for pre-trained models in code change learning, to combine FMFT and PEFT during the domain adaptation. Since the combination and assignment of PEFT and FMFT layers are based on the results of probing tasks, this reflects the self-adaptability of Pasta_K to experimental data. Here, K is a hyperparameter representing the number of FMFT layers, i.e., the resource constraint. In experiments, K is set to 0, 3, 6, 9, and 12 for evaluation. According to the layer-wise probing results, Pasta_K allocates PEFT and FMFT across different layers of PLMs for tuning, thereby integrating their respective strengths. We conduct experiments using all four PEFT methods on the CMG task to verify their effectiveness. Experimental results show that Pasta_K significantly outperforms diverse PEFT methods. Even compared with FMFT, Pasta_K can surpass it by 1.48%, 3.21%, and 1.87% at most in terms of BLEU, Meteor, and Rouge-L, respectively. Meanwhile, Pasta_K remains efficient, saving 26.26% of training time and 20.65% of computational memory compared with FMFT, indicating the practical value of Pasta_K.

1.3 Mamba in Code Intelligence

In the field of software engineering, the success and prosperity of Pre-trained Language Models (PLMs) are deeply rooted in the Transformer architecture and its core attention mechanism. These models have evolved into indispensable assets for developers nowadays, significantly boosting efficiency in a variety of code comprehension and generation tasks [92, 103, 106, 236]. Despite the advancements achieved by code PLMs, their core attention module encounters an intrinsic drawback: it scales quadratically with sequence length [111]. 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 [186]. Among them, Mamba [72] and Mamba-2 [41] 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 [72]. 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 [41]. 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 [13], vision [249], and graphs [16]. However, in the domain of code intelligence, related studies are restricted to efficiency analysis of Mamba-generated code [101] or limited code tasks (e.g., test automation) [104], 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 performance on three widely-studied code tasks: (1) code completion, (2) code generation, and (3) code clone detection. We select CodeGPT and CodeGPT-adapted [142] 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 [98], 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 [95] and $(IA)^3$ [132], 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.

1.4 Continual Learning in Code Intelligence

Recent years have seen a tendency to leverage Pre-trained Language Models (PLMs), such as CodeBERT [55] and CodeT5 [207], for various downstream code intelligence tasks, including code summarization [3, 61], code vulnerability detection [178, 242], and code clone detection [51, 183]. Typically, these works mostly follow a prevalent paradigm named pre-training and fine-tuning, which first optimizes language models

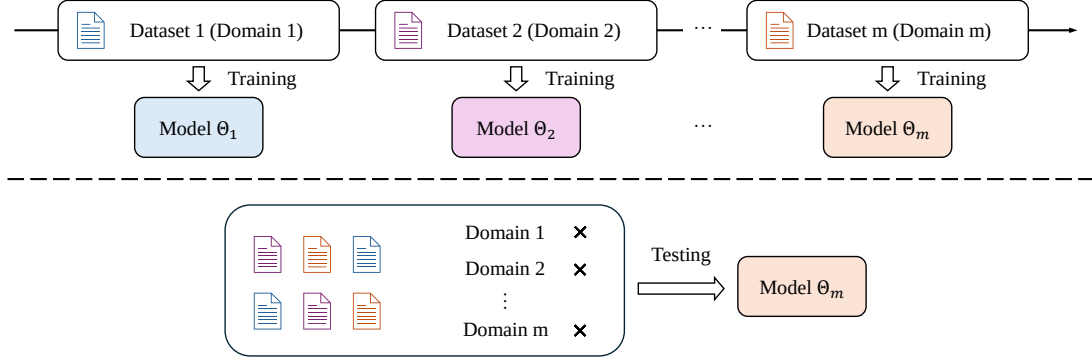


Figure 1.2. An illustration of the continual learning scenario. X marks indicate the domain ID that test samples derive from is unknown when testing.

on large-scale code corpora to obtain general knowledge, and then adapts models to acquire task-specific knowledge. Benefiting from the powerful representation capability of PLMs, they can obtain superior performance.

However, previous works primarily focus on an offline setting, tuning a PLM on a pre-collected dataset. The PLM is an expert in the domain of the pre-collected dataset, but performs worse if domain shifting. The domain shift [245], also known as the distributional shift, occurs regularly in the practical application of deep learning models and describes the variation between different datasets. In the field of code intelligence, domain shifts are more common as programs evolve constantly. New repositories and new APIs are created over time [156, 171]. If ignoring the dynamic nature of software codebases, the practicability of code intelligence models can be affected. Therefore, it is significant to explore the performance of PLMs in a Continual Learning (CL) setting, where models are trained on a sequence of datasets one by one and are expected to perform well on all introduced domains. As shown in Figure 1.2, in the training process, PLMs are adapted to a new domain once inputting a new dataset. The model finally tuned will be evaluated with data samples from all previously learned domains, but which domain a data sample belongs to is not known at test time. This setting is also called domain-incremental continual learning [193]. As far as we know, only a few works are dedicated to the CL scenario in the field of code intelligence.

Continual learning was first proposed in the field of machine learning, and some representative methods have emerged. They can mainly fall into three categories, namely regularization-based [116, 138], rehearsal-based [167, 200], and representation-based

methods [208, 209]. The major challenge in the CL scenario is the catastrophic forgetting problem [147], which indicates models’ performance on previous domains can be affected when feeding datasets of new domains. To mitigate the problem, in the code intelligence field, Gao et al. [62] resort to storing and retraining some carefully selected previous data, as well as adding an adaptive parameter regularization module. Their method can be seen as a combination of the regularization-based and rehearsal-based approaches. Similarly, Weyssow et al. [212] evaluate the two approaches on API-related tasks. Although it is intuitive to retrain models with previous data and new data to alleviate forgetting, we criticize that previous data samples are not always available because of storage or privacy constraints [203]. Consider a scenario in which several research and development teams share one continual-learned PLM to assist programming, and this PLM needs to evolve based on different teams’ development requests. However, in modern software companies, it is prevalent that historical coding samples from different teams are not allowed to be shared. In this case, the above rehearsal-based approach is impracticable. Besides, additional computational and storage resources are needed in the approach. Therefore, to avoid the risk of data disclosure and pursue practicality, we focus on a rehearsal-free method.

In this work, we resort to the superiority of prompts in eliciting pre-trained knowledge to mitigate the catastrophic forgetting problem. By adding several tunable tokens, prompt tuning can reduce the disparity in the training form between pre-training and downstream tasks, thus providing instructions when tuning PLMs [198]. In the CL scenario, Yadav et al. [225] propose prompt pools, containing diverse prompts for learning domain-specific knowledge. Besides, they also introduce an additional prompt selection and combination module to construct domain-sharing knowledge. We criticize that prompt pools lack the inheritance of previous knowledge and may lead to bias in prompt selection given an example of an unknown domain. Therefore, we expect a method with certain prompts that can instruct both domain-specific and domain-sharing knowledge. We first conduct an experimental study to explore the performance of vanilla Prompt Tuning (PT) and Parameter-Efficient Prompt Tuning (PEPT) in the CL scenario when compared with Fine-Tuning (FT). Experimental results show that PT can surpass FT while PEPT performs worse, enlightening us that it is necessary to tune PLMs when applying PT. Besides, although PT outperforms FT, they still can not balance plasticity and stability during continuous domain adaptation.

Based on the above findings, we propose an effective method called Prompt Tuning with Domain-wise Distillation (PTDD), which only involves two prompts during training, one for previous domains and one for the current domain, and only the current prompt is utilized when evaluating. Specifically, considering there are correlations between different domains in code intelligence, PTDD first initializes the prompt as its previous one and adds a Multi-Layer Perceptron (MLP) to improve models’ representation ability [122]. To distill prompts and optimize the backbone model, PTDD introduces an additional stability learning objective to maintain the learned knowledge in previous domains. PTDD can be tuned with all parameters or just the prompt parameters. We denote the former as PTDD and the latter as parameter-efficient PTDD (PE-PTDD). It is worth noting that the prompt distillation here describes a process that limits the variation of previous prompts in cross-domain adaptations, getting the current prompt that can furthest keep the previous instruction ability and adapt it to the current domain. It is a different term from the knowledge distillation that is often used in the machine learning field.

We conduct experiments on three widely-used code intelligence tasks where domain shifts occur regularly, including code summarization (CS), code vulnerability detection (CVD), and code clone detection (CCD). We evaluate PTDD in comparison with various baselines, and PTDD can obtain consistent improvements among all rehearsal-free methods. For instance, in the CS task, PTDD averagely outperforms FT by 2.55% in terms of the BLEU metric on six datasets. In the CVD and CCD tasks, PTDD outperforms FT by 11.12% and 2.25% in terms of the F1 score. Even when comparing PE-PTDD with other parameter-efficient baselines, our method still shows effectiveness. We also conduct an ablation study and parameter analysis to investigate the performance of PTDD from different perspectives. Furthermore, we visualize the prompts learned by PTDD and PT to give an explanation and discuss the capability of PTDD in the low-resource scenario. In the CS task, when decreasing the number of training samples to 10 in each dataset, the improvement of PTDD can reach up to 69.09% in terms of BLEU. The extensive experiments indicate the superiority of our method in the CL scenario.

1.5 Program Termination Analysis

Termination analysis aims to discover non-terminating programs that are induced by improper code structures or faulty statements. These programs may execute infinitely

and incur various bugs, such as denial-of-service vulnerability and memory exhaustion. Therefore, many researchers have devoted themselves to this field. Effective termination analysis tools have become a pivotal component in ensuring system availability, serving as an indispensable role in code auditing and automated formal verification workflows.

Traditional methods of termination analysis employ formal symbolic reasoning. For example, some studies developed various methods to prove the existence of ranking functions, thereby identifying terminating programs [17, 18, 20, 69, 162, 192]. However, since these methods cannot discover non-terminating programs, their detection results always contain lots of false positives. Accordingly, subsequent studies proposed to adopt the recurrence sets to prove the existence of non-terminating programs [30, 39, 58, 117]. Whereas recurrence sets usually need to be singletons, making it difficult to find a recurrence set for a program. Moreover, all of them can not conduct their detection without program execution. Hence, the detection process can be very time-consuming, and the configuration of the sandbox environments for execution is also cumbersome.

In recent years, machine learning-based methods gradually emerged [66, 182, 217], where they adopt support vector machine and neural networks to simulate ranking functions and recurrence sets during the termination detection, thus they still can not exempt the execution. Different from the above methods, Alon et al. [10] adopted Graph Neural Network (GNN) to directly analyze program semantics on their Abstract Syntax Trees (ASTs). Hence, they can detect non-terminating programs effectively without program execution. Although this approach overcame deficiencies of previous studies, their methodologies are coarse-grained. On the one hand, their modeling process is one-sided. Because they only consider intra-class semantics without learning inter-class semantics when termination detection, the discrepancy between terminating and non-terminating programs will be ignored. On the other hand, their approach also neglects the class-imbalance problem. As a kind of program defect, programs tend to have more terminating instances than non-terminating ones [53] in practice.

To address the aforementioned problems, we propose a graph-enhanced contrastive approach for program termination analysis (TerGEC) in this work. First of all, TerGEC analyzes program semantics via ASTs, thereby ensuring the comprehension of code both syntactically and lexically. Besides, both intra-class and inter-class learning are considered in TerGEC, which is our main innovation and makes the analysis to be more

fine-grained. Moreover, we also proposed a series of solutions in TerGEC to overcome the class-imbalance problem.

In detail, TerGEC first transforms programs into their corresponding ASTs. Subsequently, we propose two graph augmentation methods, namely feature masking and feature projection, which can increase the diversity of training samples, improve the robustness of TerGEC, and is the prerequisite for the following inter-class learning. Afterward, both the original ASTs and the augmented ASTs are encoded by GNNs, where three GNN models are experimented in this work, including GCN [115], GAT [196], and GIN [216]. Towards the intra-class semantics, we follow the processes that Alon et al. [10] used and introduce focal loss during the training. As for inter-class semantics, TerGEC utilizes contrastive learning to learn the discrepancy between terminating and non-terminating programs. Finally, we bridge the learning of both intra-class and inter-class perspectives by a hyperparameter and formulate the final optimization target. Thereby a more comprehensive analysis can be accomplished. In addition, during inter-class and intra-class learning, weighted contrastive loss and focal loss are also equipped in TerGEC to alleviate the classes-imbalance problem.

In experiments, we construct four custom datasets with a total of 20k Python programs for termination analysis, which contain much more samples than the previous dataset and reflect the classes-imbalance problem that the previous dataset overlooked. To assess the generalization ability of TerGEC, we also use an existing dataset TPDB [67] with 206 C programs for extra experiments. Subsequently, we compare TerGEC with six state-of-the-art baselines on the aforementioned five datasets. Experimental results show that TerGEC outperforms baselines by 4.59% - 15.56% in terms of mAP on dataset 125M_HumanEval. And similarly, the increments of mAp can up to 2.56% - 17.79% on dataset 125M_MBPP, 0.18% - 11.38% on dataset 1_3B_HumanEval, 1.00% - 10.60% on dataset 1_3B_MBPP, and 3.17% - 15.18% on dataset TPDB. As for the AUC metric, TerGEC achieves the best on datasets 125M_MBPP, 1_3B_MBPP, and TPDB, where the improvements are 5.10% - 27.55%, 1.25% - 42.38%, and 2.69% - 34.87% respectively, although it underperforms a little on the other two datasets. Besides, we design a series of ablation experiments to analyze the effectiveness of the graph augmentation module and the balance factors used in the weighted contrastive loss and the overall loss. Extensive experiments demonstrate that the graph-enhanced contrastive approach that combines intra-class and inter-class learning, together with our proposed class-imbalance solutions, can significantly improve the performance of non-terminating program detection in practice.

1.6 Contributions of the Thesis

The primary contributions of this thesis are summarized as follows.

Within the landscape of high-efficiency fine-tuning paradigms, we pioneer a systematic investigation into the efficacy and efficiency of PEFT on code-change-related tasks. Specifically, we conduct extensive experimental evaluations encompassing four PEFT methods and seven PLMs across two downstream tasks, while thoroughly exploring their performance in the cross-lingual and low-resource scenarios. Furthermore, we propose two specialized code-change-related probing tasks along with curated datasets to dissect the encoded dynamic code semantics. Inspired by our probing insights, we propose Pasta_K , a self-adaptive tuning approach that synergistically combines FMFT and PEFT during domain adaptation, achieving an optimal trade-off between adaptation performance and computational overheads, demonstrating its practical value. Finally, the complete replication package of this work is publicly available to facilitate future research¹.

Focusing on the exploration of efficient model architectures, we present the first comprehensive empirical evaluation of Mamba-based PLMs in the context of code intelligence. In particular, we conduct an in-depth comparison between Mamba-based and Transformer-based PLMs across multiple fine-tuning paradigms, including FT, LoRA, and $(IA)^3$, dissecting both performance advantage and computational efficiency of the Mamba architecture. Furthermore, we evaluate Mamba-based PLMs under challenging low-resource scenarios and explore their performance at larger parameter scales, providing practical suggestions and exposing critical weaknesses of Mamba-based PLMs for reference of both academia and industry. All source code, specialized configurations, and pre-trained model weights are made open-source to the community².

When it comes to dynamic adaptive learning strategies, we conduct the first experimental study to explore the performance of PT in the CL scenario. Our preliminary findings reveal two insights: on one hand, relying solely on frozen PLMs under vanilla PT is insufficient for sequential tasks; on the other hand, maintaining prompt stability is a critical requirement for successful adaptation. Consequently, we propose a rehearsal-free framework named PTDD, which can distill and preserve previous knowledge by introducing a stability learning objective, thereby mitigating catastrophic forgetting

¹<https://github.com/ishuoliu/PeftCC>

²<https://github.com/ishuoliu/ASE-CMamba>

across previously learned domains. We conduct extensive evaluations on three widely-studied code intelligence tasks, where experimental results validate the effectiveness of PTDD. Furthermore, we provide interpretations of our framework via prompt visualization and discuss its capabilities in low-resource scenarios. To facilitate reproduction and inspire future research, our source code has been made publicly accessible³.

Regarding structure-aware adaptive learning approaches, we introduce TerGEC, a graph-enhanced contrastive approach that simultaneously captures the intra-class and inter-class semantics of programs via ASTs, thereby enabling a more fine-grained termination analysis. In addition, we propose two graph augmentation strategies to supplement samples and increase their diversity, and incorporate weighted contrastive loss and focal loss to alleviate the class-imbalance problem. Critically, we construct four custom termination detection datasets with a total of 20K samples, which constitute the first large-scale dataset for termination analysis. We also release our replication package⁴, containing the custom datasets and the complete implementation of TerGEC. Extensive evaluations across both Python and C programs demonstrate the superior predictive efficacy and robust generalizability of the proposed TerGEC framework.

1.7 Organization of the Thesis

The remainder of this thesis is organized as follows.

Chapter 2 presents a comprehensive literature review to synthesize the foundational works underpinning this thesis. Specifically, it reviews specialized tasks such as code change learning and termination analysis, discusses diverse model architectures from Transformer-based to Mamba-based code language models, and details advanced learning paradigms encompassing continual learning and graph contrastive learning.

Chapter 3 delves into PEFT within the field of code change learning and introduces Pasta_K, a self-adaptive layer-specific tuning framework. Inspired by the distinctive advantages of PEFT and FMFT revealed through layer-wise probing results, Pasta_K can dynamically combine both tuning paradigms during domain adaptation.

Chapter 4 investigates the efficacy and efficiency of Mamba in code intelligence via rigorous empirical assessment. Specifically, from-scratch pre-training experiments are designed to uncover the intrinsic architectural advantages of Mamba. Extensive

³<https://github.com/ishuoliu/IST-25-PTDD>

⁴<https://zenodo.org/records/10703451>

evaluations provide critical insights into both the strengths and boundaries of Mamba-based code intelligence.

Chapter 5 introduces PTDD, a novel framework that can distill prompts and optimize PLMs via a two-sided learning objective within the continual learning scenario. We conduct comprehensive experiments to thoroughly evaluate its performance, complemented by prompt visualization to elucidate its underlying mechanism.

Chapter 6 presents TerGEC, an execution-free, graph-enhanced contrastive approach tailored for termination analysis. It captures fine-grained intra-class semantics by analyzing programs from ASTs, and uncovers inter-class semantics through a dedicated contrastive learning objective. Extensive evaluations across five Python and C datasets demonstrate that TerGEC achieves the best performance in general.

Finally, Chapter 7 concludes this thesis with a comprehensive synthesis of the core findings and foundational contributions established herein, while outlining promising research directions for future endeavors within the realm of code intelligence.

Chapter 2

Literature Review

This chapter provides a systematic survey of the literature, structured around three primary dimensions: specialized tasks, model architectures, and learning paradigms. We begin by reviewing the research on code change learning and termination analysis. Subsequently, the discussion delves into diverse architectures for code language models, encompassing both the well-established Transformer and the emerging Mamba framework. Finally, we explore continual learning strategies alongside the advancements in GNN and graph contrastive learning.

2.1 Code Change Learning

Code changes are closely associated with software development and exist extensively. It appears when adding new features, fixing bugs, or refactoring current code [23, 231]. Along with code changes, developers have to undertake considerable workloads for tasks like writing high-quality commit messages [107] and identifying whether software changes are defect-inducing [108]. Therefore, techniques aiming at automating such code-change-related tasks are emerging and showing their effectiveness in enhancing development efficiency. Most of the works follow the representation learning approach, which converts code changes into feature vectors primarily, and then retrieves them in the feature space to obtain expected results.

Concentrating on concrete downstream tasks, several studies have proposed task-specific methods to learn code change representations. For commit message generation, CoDiSum [218] extracts code structure and code semantics by separating bidirectional

GRU and aligns the two parts by an attention layer. FIRA [45] describes code change operations in fine-grained graphs and utilizes a graph-neural-network-based encoder to learn representations. RACE [174], based on the Transformer architecture, retrieves instructive code changes firstly by the cosine similarity of representation vectors, using them to guide the generation of commit messages. For just-in-time defect prediction, DeepJIT [88] leverages two discrete CNNs to extract features from code changes and corresponding commit messages, and employs a fully connected network for feature fusion. JITLine [163] extracts bag-of-tokens features and conducts five well-known classification techniques, like Support Vector Machine (SVM), to build commit-level prediction models. JIT-Fine [154] combines 14 change-level expert features [108] with extracted semantic features, obtaining integrated representations.

Beyond task-specific optimizations, a vast wealth of research focuses on general-purpose representations for code changes that can be applied across various downstream tasks. CC2Vec [89] inputs the removed code and added code separately to a hierarchical attention network, extracts their features, and uses multiple comparison functions to fuse the two parts. CCRep [140] proposes a novel mechanism called query back, which can emphasize the core status of changed code and adaptively learn representations from the context. CodeReviewer [125] leverages a large-scale dataset of real-world code changes to pre-train PLMs using four specialized objectives, including code change quality estimation, review comment generation, and code refinement. These tasks enhance models' representative power in code review scenarios. Similarly, CCT5 [127] proposes four pre-training tasks to build the semantic connection between code changes and corresponding commit messages. Furthermore, it incorporates an additional pre-training task to make the model aware of the program structure. CCS2VEC [240] adopts a slice-based approach to code-change representation, extending its scope to include unchanged code that has control or data dependencies to the changed code. Specifically, it performs forward and backward slicing on the Program Dependency Graph (PDG) of the code before the change, centered around the removed code.

2.2 Termination Analysis

Termination is associated with the liveness and safety properties of programs, which means termination is crucial to state the correctness of programs [9]. Proving termination is vital for specific domains like term rewrite systems [226] and GPU programming [112], and therefore, research about it has emerged constantly. To prove termination,

many existing methods intend to search for ranking functions, which map program states to certain values [66]. In addition, these functions are monotonic decreasing after each loop iteration, and are lower bounded. Due to the existence of a ranking function being equivalent to the termination behavior of given programs, various approaches to synthesize ranking functions are proposed, such as linear [17, 162], lexicographic linear [20, 69], multiphase [18], and piecewise [192]. However, these ranking function-based methods mostly are incomplete. The denial of proving termination does not imply non-termination, and the program still may or may not terminate.

Correspondingly, proving non-termination is also a challenging task. The general approach is based on the recurrence set [80], which is a set of non-terminal program states in which the program can stay indefinitely [27]. Chen et al. [30] further proposed a concept, i.e., closed recurrence set, simplified the non-termination proving problem, and solved it with safety provers. Cook et al. [39] introduced live abstractions to soundly disprove termination. Frohn et al. [58] designed a loop acceleration technique, which can generate a non-termination criterion to promote the proving process. Le et al. [117] utilized dynamic invariant inference to learn recurrent sets.

In recent years, approaches integrating machine learning and deep learning have gained significant traction. Diverging from previous methods that require predefining a linear or polynomial ranking function template, Tan et al. [182] proposed a data-driven strategy to synthesize ranking functions via Deep Neural Networks (DNNs). The framework begins with the initial construction of a function template, where the specific coefficients are subsequently optimized using a collected training set. The resulting candidate ranking function is then subjected to a formal verification process to confirm its validity. A similar thought was implemented by Giacobbe et al. [66]. They train a neural network using sampled execution traces, and the network is optimized to ensure its output decreases along the traces. Subsequently, symbolic reasoning is employed to formally verify that the learned property generalizes to all potential executions. If the verification is successful, a formal certificate of termination is established for the program. Xu et al. [217] employ a reinforced black-box learning approach, consisting of a loop bound learner and a validation checker. It integrates a quick bound checking algorithm and a two-way data sharing mechanism to enhance overall efficiency. Building on the strengths of GNNs in representation learning, Alon et al. [10] design and implement classifiers for program termination based on graph convolutional networks and graph attention networks. They also introduce a semantic segmentation GNN to precisely localize AST nodes that are likely to trigger non-termination.

2.3 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 [55], CodeGPT [142], and CodeT5 [207] are three pioneering works, and they just correspond to three different architectures: Encoder-only, Decoder-only, and Encoder-Decoder [157, 238]. Appeared after them, UniXcoder [75] 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 [155] 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 [215] is based on the GPT-2 [164] architecture and is pre-trained on 249GB of code across 12 programming languages, sizing up to 2.7B parameters.

Beyond models explicitly engineered for code intelligence, a prevalent trend in the open-source community involves adapting general-purpose Large Language Models (LLMs) for code-related tasks through continual pre-training or fine-tuning. High-profile examples include Meta AI’s Code Llama [169], derived from the foundational LLaMA model [191], and DeepSeek-Coder [78], an extension of the baseline DeepSeek LLM [19]. Similarly, the Qwen team and Google have followed this trajectory, introducing Qwen2.5-Coder [97] and CodeGemma [187] as code-specific variants of Qwen [14] and Gemma [188], respectively. Collectively, these developments underscore a growing consensus and concerted effort toward repurposing foundational LLMs to excel in software engineering tasks.

2.4 Mamba and Mamba-based Models

Although the Transformer [195] 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 [111]. Recently, Mamba [72] 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, such as natural language processing [13], graph learning [16], and visual representation [249]. As a subsequent advancement, Mamba-2 [41] exploits the theoretical convergence between SSMs and Transformers. By refining the core architecture of its predecessor, it introduces Structured State Space Duality (SSD) and enhances hardware efficiency, thereby achieving substantially faster training speeds.

Within the domain of code LMs, Codestral Mamba [37] stands as the only mamba-based LLM to date that is specifically optimized for code. Its development benefits from the guidance of Gu et al., the original authors of Mamba. Built upon the Mamba-2 architecture with 7 billion parameters, the model supports a native context window of up to 256k tokens. It is fully open-sourced under the Apache 2.0 license, with all model weights publicly accessible to facilitate seamless modification and deployment. A few preliminary studies have explored the capability of Codestral Mamba. Islam et al. [101] investigate the energy efficiency and environmental impact of the code generation task, evaluating Codestral Mamba alongside 19 other popular code language models. Their work effectively bridges the gap left by prior studies, which predominantly focused on the functional correctness of generated code. Iznaga [104] pioneers the integration of Codestral Mamba and LoRA for automated software testing, revealing significant improvements in generating high-quality, context-aware test scripts. Given that research on the Mamba architecture is still in its embryonic stage, there remains a noticeable paucity of such Mamba-based studies in the wider code intelligence domain.

2.5 Continual Learning

Continual learning is set to learn a sequence of datasets one by one and behave on them simultaneously [203]. The major challenge is known as catastrophic forgetting [147], where models' performance on previous datasets is prone to decline when adapting the model to new datasets. Existing CL methods are generally categorized into three primary strains: regularization-based, rehearsal-based, and representation-based methods. Regularization-based methods [116, 138] selectively constrain the variation of models' parameters, thus alleviating forgetting of learned knowledge. They usually add penalty terms to the loss function, and through the backpropagation process, those important parameters will be preserved. Rehearsal-based methods [167, 200] store a fraction of data samples and retrain them when new datasets come, to ensure models still have the previous knowledge. Representation-based methods [208, 209] primarily

utilize prompts to instruct the representation of pre-trained models. By accommodating prompts in different datasets, they can elicit task-sharing and task-specific knowledge to relieve forgetting.

Due to the prosperity of prompt-based methods in recent years, representation-based methods have become popular with simplicity, practicality, and good performance. A representative strategy, L2P [209], learns to dynamically prompt a pre-trained model for sequential task learning. It conceptualizes a shared prompt pool that archives task-independent prompts, from which the most relevant ones are adaptively selected based on different inputs during inference. In this manner, L2P effectively manages knowledge retention and transfer across varying tasks. Building upon the foundations of L2P, DualPrompt [208] draws inspiration from the human Complementary Learning Systems (CLS), which facilitate continual learning through the synergy of two distinct memory mechanisms. Specifically, DualPrompt introduces general prompts to capture task-invariant common features, and expert prompts to encode task-specific unique knowledge. By directly decoupling the higher-level prompt space, this method proves to be both more effective and memory-efficient. Another typical method, Progressive Prompts [166], allocates a dedicated prompt for each incoming task and sequentially concatenates it with previously learned prompts to systematically guide the representation learning. Additionally, it reparameterizes the prompt embedding matrix utilizing a Multi-Layer Perceptron (MLP) to enhance stability and model capacity.

Within code intelligence, software repositories and programming paradigms constantly evolve. To seamlessly augment models with fresh knowledge, investigating their adaptation capabilities within CL scenarios has become both practically imperative and strategically vital. Despite its profound practical significance, scholarly dedication to this intersection remains conspicuously scarce. Gao et al. [62] pioneer the investigation into this problem by proposing REPEAT, a novel framework that mitigates catastrophic forgetting through a synergy of representative exemplar replay and adaptive parameter regularization. The former strategically selects both informative and diverse samples from historical datasets for periodic rehearsal. Concurrently, the latter identifies critical parameters and adaptively penalizes their updates, thereby insulating historical knowledge from consolidation overwrites. Weyssow et al. [212] formulate a dynamic scenario where models are required to sequentially learn from a stream of software programs characterized by novel, unseen APIs. Under this setup, they systematically evaluated five regularization-based and rehearsal-based baselines on the

API call and API usage prediction tasks. Yadav et al. [225] construct CodeTask-CL, a comprehensive benchmark spanning a diverse spectrum of tasks, including code generation, translation, summarization, and refinement. They also introduce a methodology termed Prompt Pooling with Teacher Forcing (PP-TF), which enforces constraints on prompt selection during the training phase by assigning certain prompts to fixed tasks.

2.6 GNN and Graph Contrastive Learning

Inspired by the remarkable success of Deep Neural Networks (DNNs), Graph Neural Networks (GNNs) were introduced to effectively operate on graph-structured data, rapidly evolving into a highly prominent and proliferating research domain [71]. Kipf et al. [115] propose the Graph Convolutional Network (GCN), which employs an efficient architecture motivated by a localized first-order approximation of spectral graph convolutions. By scaling linearly with the number of graph edges, GCN seamlessly encodes both local graph topologies and node features within its hidden layers. Velickovic et al. [196] design the Graph Attention Network (GAT), which incorporates masked self-attentional layers to direct the model towards the most relevant regions. By stacking these attention layers, the model implicitly assigns distinct weights to different nodes within a specific neighborhood, and operates effectively without requiring prior knowledge of the global graph topology upfront. Xu et al. [216] emphasize the critical importance of capturing structural discriminability across graphs, leading to the development of the Graph Isomorphism Network (GIN). They also empirically prove that GIN has high representational power, showing a near-perfect fit on training data. Beyond these, a diverse lineage of specialized variants has emerged to accommodate heterogeneous graph topologies. Notable exemplars include EdgeConv [206] for point clouds and GraphSAGE [82] for large-scale graphs. Collectively, these models serve as robust graph encoders to transform graph-structured data into vector representations.

Based on graph neural networks, researchers proposed graph contrastive learning to improve the performance of graph tasks. Graph contrastive learning is an approach that aims to utilize those label-invariant properties to gather samples with the same label and separate samples with different labels. The properties are usually learned from the original data and the augmented auxiliary data. You et al. [234] introduce a taxonomy of four graph augmentation mechanisms: node dropping, edge perturbation, attribute masking, and subgraph extraction. Subsequently, they conduct comprehensive studies to evaluate how diverse combinations of these augmentations influence

model performance. Departing from the above topological augmentation, Zhang et al. [248] propose COSTA, a framework that executes augmentation directly on the hidden feature vectors. They introduce three distinct feature-level augmentation variants manifested through matrix sketching techniques: singular value decomposition, random row selection, and random projection. Zhu et al. [252] demonstrate that augmentation strategies should adaptively calibrate according to the intrinsic characteristics of the input graphs. Specifically, on the topology level, they design adaptive schemes utilizing node centrality measures to highlight pivotal connective structures. On the node attribute level, they corrupt node features by injecting heavier noise into non-essential features, thereby forcing the model to robustly discern underlying semantic information. Distinct from the aforementioned self-supervised paradigms, Khosla et al. [113] and Zhang et al. [244] extend graph contrastive learning into a supervised setting, pioneering its application within image classification.

In the realm of program analysis, various representation methods [6, 204, 243] have been established to formulate source code as graphs. These graph-structured methodologies can be seamlessly adapted to a diverse array of critical downstream software engineering tasks, such as type inference [5], bug detection [7], and code summarization [56, 229]. The profound advancements achieved by these GNN-based paradigms consistently underscore the inherent superiority of graph representations in analyzing programs.

Chapter 3

Delving into Parameter-Efficient Fine-Tuning in Code Change Learning

In comparison with Full-Model Fine-Tuning (FMFT), Parameter-Efficient Fine-Tuning (PEFT) effectively mitigates catastrophic forgetting in code intelligence by updating only a fraction of model parameters. While existing works validate the viability of PEFT in static code comprehension, owing to its alignment with PLM pre-training paradigms for efficient knowledge transfer, whether PEFT can outperform FMFT in dynamic, code-change-related scenarios remains an open question. To bridge this gap, we conduct a comprehensive evaluation comparing four prominent PEFT methods against FMFT across seven representative PLMs on two pivotal tasks: Just-In-Time Defect Prediction (JIT-DP) and Commit Message Generation (CMG). Our benchmarks reveal that while PEFT systematically excels over FMFT on JIT-DP, it merely achieves comparable results on CMG, albeit maintaining a clear advantage in cross-lingual and low-resource settings. To explain the efficacy of PEFT and FMFT, we perform extensive layer-wise probing tasks from both static and dynamic perspectives. The diagnostic insights inspire Pasta_K , a self-adaptive layer-specific tuning framework that harmoniously orchestrates FMFT and PEFT under the guidance of probing results. Empirical results on CMG demonstrate that Pasta_K consistently surpasses all PEFT baselines. Remarkably, it outperforms FMFT by up to 1.48% in BLEU, 3.21% in Meteor, and 1.87% in Rouge-L, while simultaneously reducing training time by 26.26% and memory consumption by 20.65%.

The rest of this chapter is organized as follows. Section 3.1 presents the preliminary and theoretical foundations of FMFT and diverse PEFT methods. Section 3.2

establishes the experimental setup, detailing the research questions, core tasks, baselines, and implementation specifications. Section 3.3 discusses the experimental results and proposes the Pasta_K method. Section 3.4 elaborates on the implications and threats to the validity of this work. Finally, section 3.5 concludes this chapter by providing a concise summary.

3.1 Preliminary

3.1.1 Full-Model Fine-Tuning

Pre-training and fine-tuning have become the dominant paradigm for applying PLMs to specific downstream tasks, and have shown promising results on many code-related tasks [109, 119, 214, 230]. In the pre-training stage, PLMs are trained from scratch to learn general-purpose representations, and in the fine-tuning stage, PLMs utilize the pre-trained parameters as initialization to adapt them to downstream tasks. Formally, given a downstream task-specific dataset X and corresponding labels Y , FMFT initializes PLMs to pre-trained weights Θ_0 and updates them to Θ in a supervised manner, which can be formulated as:

$$\Theta = \Theta_0 + \Delta\Theta = \arg \max_{\Theta} Pr(Y|X; \Theta) \quad (3.1)$$

where Pr represents the likelihood of Y and $\Delta\Theta$ represents the adjustment of parameters. Considering that PLMs are Transformer-based architecture [195], FMFT is illustrated in Figure 3.1(a). The parameters of the core block of the Transformer, including multi-head attention, feed-forward layer, and layer normalization, as well as the projection matrices W_Q , W_K , and W_V , all are part of Θ and are trainable in FMFT for PLMs.

3.1.2 Adapter Tuning (AT)

AT is a representative method of PEFT, which adds extra compact modules (adapters) to every transformer layer [85, 94, 160]. Figure 3.1(b) shows the standard architecture of AT [94], where adapter modules are added twice after the multi-head attention and the feed-forward layer. Supposing that adapter modules are fed d -dimensional features,

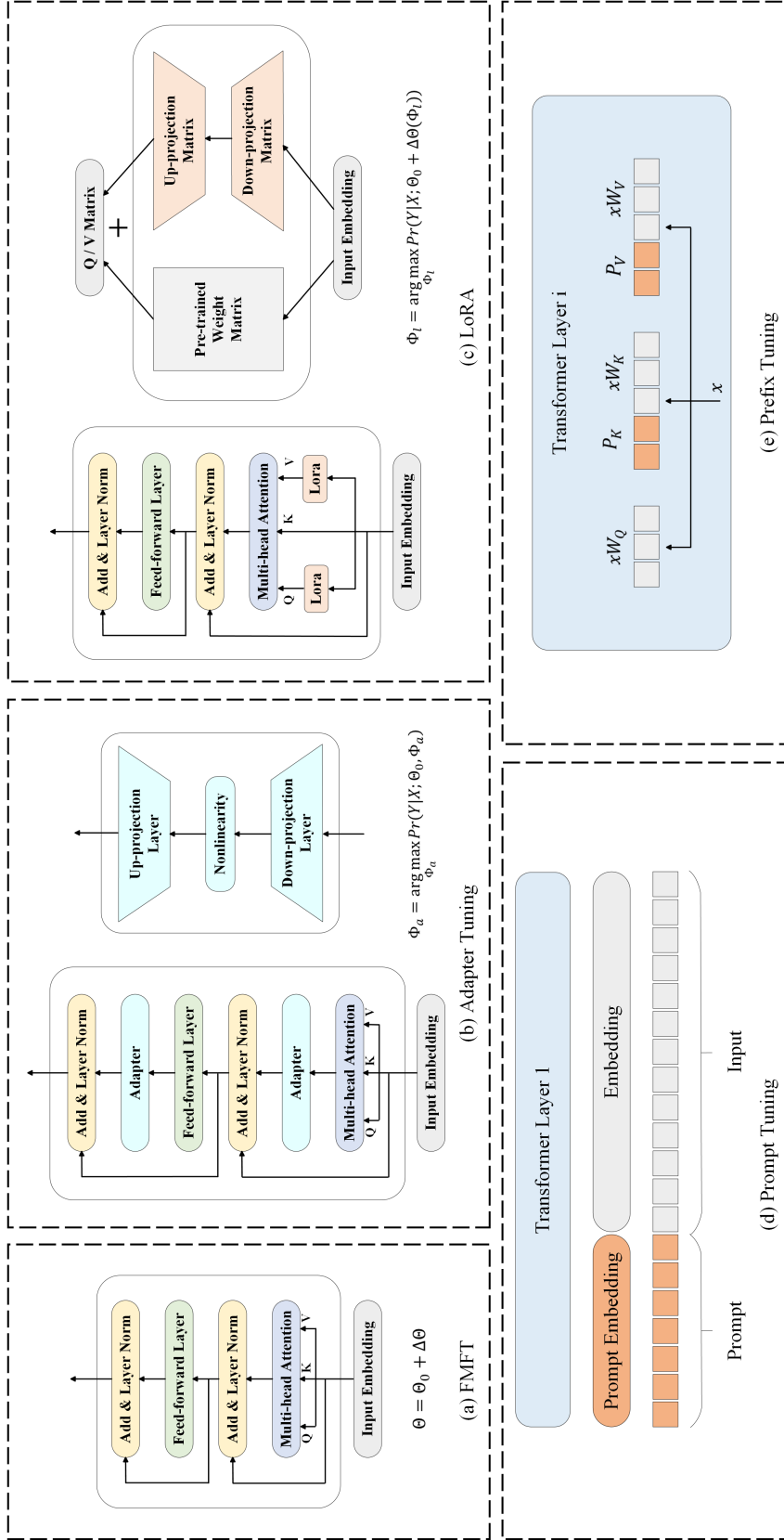


Figure 3.1. Illustrations on FMFT, AT, LoRA, PT, and PreT.

they first project features into dimension m , $m \ll d$, then apply a nonlinearity, and finally project back to dimension d . The two projection layers construct a bottleneck structure. Specifically, adapter modules are initialized randomly, and the target of AT is to train the added modules and the corresponding following layer normalizations. Its tuning objective can be defined as:

$$\Phi_a = \arg \max_{\Phi_a} Pr(Y|X; \Theta_0, \Phi_a) \quad (3.2)$$

where Φ_a is the adapter parameter, and Pr represents the likelihood. X and Y are a task-specific dataset and corresponding outputs. PLMs' pre-trained weights Θ_0 are frozen here. During AT, around 3% - 5% of the whole parameters are trainable, which is considerably fewer than FMFT.

3.1.3 Low-Rank Adaptation (LoRA)

Different from AT, which adds extra trainable parameters, LoRA utilizes the idea of reparameterization to learn the parameters' updation by low-rank matrices [126]. As illustrated in Figure 3.1(c), when applying LoRA to the transformer architecture, it is limited to only adapting the attention modules, and specifically, adapting weight matrices W_Q and W_V [95]. For a pre-trained weight matrix $W \in \mathbb{R}^{d \times d}$, LoRA decomposes its updation into two low-rank matrices that satisfy $\Delta W = BA$, where $B \in \mathbb{R}^{d \times r}$, $A \in \mathbb{R}^{r \times d}$, $r \ll d$. Supposing $\Delta\Theta$ is a task-specific parameter modification, it can be further encoded by the LoRA parameter Φ_l as $\Delta\Theta(\Phi_l)$. The tuning objective is then transferred to optimize Φ_l as:

$$\Phi_l = \arg \max_{\Phi_l} Pr(Y|X; \Theta_0 + \Delta\Theta(\Phi_l)) \quad (3.3)$$

where X and Y are the dataset and corresponding outputs. Pr represents the likelihood of Y . Θ_0 is also fixed here. Due to LoRA being only inserted into weight matrices W_Q and W_V , as well as the dimension $r \ll d$, the trainable parameters of LoRA can be reduced to 1%.

3.1.4 Prompt Tuning (PT)

Rather than inserting tiny modules inside PLMs as the above two PEFT methods, PT prepends a series of prompt tokens P to the input X , formulating a new input $[P; X]$ for model training [120]. The strength of PT lies in its ability to steer PLMs toward effectively leveraging the rich knowledge acquired during the pre-training phase simply by optimizing prompts, while also demonstrating robustness even under data-scarce scenarios. It has been investigated in lots of static code comprehension tasks [52, 198], and we extend its exploration to the realm of dynamic code comprehension tasks. As shown in Figure 3.1(d), prompt tokens are inserted into the first Transformer layer only. When optimizing prompt tokens P , PLMs’ pre-trained weights Θ_0 are fixed, and the trainable parameters are only about 0.03%. The tuning objective can be formulated to maximize the likelihood of Y , which represents the expected outputs of X :

$$P = \arg \max_P Pr(Y|[P; X]; \Theta_0) \quad (3.4)$$

Considering that PT is an effective approach to elicit PLMs’ pre-trained knowledge [22] and has been widely studied in recent literature [74, 197, 198, 211], we include it for exploration on code change learning.

3.1.5 Prefix Tuning (PreT)

Similar to PT, PreT [122] is also a prevalent prompt-based method that has been widely studied [44, 133, 197, 211]. Unlike PT, PreT inserts trainable prefix vectors to each Transformer layer, thereby enhancing the expressiveness of PLMs when activating their previously acquired knowledge. The trainable vectors are introduced to the original key vector and value vector, which are used for the calculation of multi-head attention. We include PreT in our exploration since it is another comparable counterpart to AT and LoRA, both of which inject trainable modules into PLMs. Figure 3.1(e) shows an illustration of PreT. In each transformer layer, there are two sets of prefix vectors P_K and P_V that are concatenated with xW_K and xW_V , respectively. Here x is the input vector. W_K and W_V are the original weight matrices. xW_K and xW_V are the original key vector and value vector. Only the prefix vectors are trainable, which are about 0.5%, and PLMs’ pre-trained weights Θ_0 are fixed. The tuning objective can be defined to maximize the likelihood of the input X ’s corresponding output Y :

$$[P_K; P_V] = \arg \max_{P_K, P_V} Pr(Y|X; \Theta_0, P_K, P_V) \quad (3.5)$$

3.1.6 Training-free Methods

The above-introduced methods require parameter updates to PLMs for domain adaptation. Here, we present two training-free methods that PLMs also use for baseline comparison. The first one is Zero-shot Learning (ZL). It bypasses the fine-tuning process and directly guides PLMs/Large Language Models (LLMs) to perform prediction or generation on downstream tasks without learning from any training samples. ZL is more pervasive for LLMs owing to their powerful emerging ability [54, 232]. The second one is In-Context Learning (ICL), which also does not require fine-tuning; instead, it guides PLMs/LLMs to perform specific downstream tasks by providing examples within the input. Recently, more and more studies have proved its effectiveness, especially when the given examples resemble the target problem [50, 109].

3.2 Experimental Setup

3.2.1 Research Questions

This chapter investigates the following research questions about four widely used PEFT methods, exploring their efficacy and efficiency.

RQ1: How do the four PEFT methods perform compared with FMFT and SOTA approaches? In this RQ, we compare the performance of FMFT, AT, LoRA, PT, and PreT on two widely studied code-change-related tasks, namely Just-In-Time Defect Prediction (JIT-DP) and Commit Message Generation (CMG). The former is a classification task, and the latter is a generation task. For JIT-DP, we apply the FMFT and four PEFT methods to six popular PLMs, namely CodeBERT [55], GraphCodeBERT [76], PLBART [4], UniXcoder [75], CodeT5 [207], and CodeT5+ [205], to conduct comprehensive comparisons and explore the generalization abilities of PEFT on diverse PLMs. As a common workaround, classification tasks normally rely on encoders appended with a linear layer for category prediction [33, 125, 127]. Thus, we select only encoder-only and encoder-decoder PLMs for experiments, where the latter’s encoder parts are decomposed for usage. Besides, various task-specific SOTA

approaches are also involved for comparison [89, 127, 154, 163]. For CMG, we apply FMFT and PEFT methods on CodeT5, CodeT5+, and Qwen2.5-Coder [97] for their strong capabilities in generation tasks. Here, we only select encoder-decoder and decoder-only PLMs for experiments, because encoder-only PLMs do not have generation ability due to their pre-training paradigm. Additionally, we incorporate ZL and ICL as baseline methods in the two tasks to facilitate a more comprehensive analysis.

RQ2: How capable are the four PEFT methods in the cross-lingual scenario compared with FMFT? In the cross-lingual scenario, practitioners normally fine-tune PLMs in one PL of rich samples and apply them in another scarce one for inference. This is a common but resultful approach to relieve the problem of data scarcity [250]. Regardless of static or dynamic code comprehension tasks, cross-lingual scenarios are pervasive with widespread studies [34, 159, 197, 198]. Because there are massive domain-specific PLs that contain very few available training samples in practice. At the time the studies [161, 230] were released, 8,945 PLs have been invented, while Meyerovich et al. [148] found that the top 20 PLs account for 95% of the project repositories, showing that most of the PLs are niche. Therefore, exploring the performance of PEFT methods in the cross-lingual scenario is valuable for practical guidance. To do this, we carry out experiments with FMFT, AT, LoRA, PT, and PreT on CodeT5 and select the CMG task for evaluation, as it involves code change samples from five PLs.

RQ3: How capable are the four PEFT methods in the low-resource scenario compared with FMFT? The performance of FMFT prohibitively relies on the scale of downstream data [74, 120, 246]. However, large-scale datasets are usually rare. In addition, fine-tuning in the low-resource scenario means that PLMs can learn task-specific knowledge quickly, which meets practitioners’ usual expectations. Considering the difference in dataset sizes between the two tasks, we randomly sample 100, 500, 1,000, 2,500, and 5,000 training data for JIT-DP, and 1,000, 5,000, and 10,000 per programming language for CMG. To mitigate the impact of data sampling variability, we conduct three independent experiments using different random seeds and report the average results of the three runs. We still evaluate the models on the original validation sets and testing sets.

RQ4: What kind of knowledge is encoded by the four PEFT methods? To explore what benefits PEFT methods can bring about, we utilize probing tasks, which have been extensively used in the NLP field [189, 190], to understand the encoded semantic information. We employ three probing tasks, invalid TYPE detection (TYP)

for static semantic-level information [110], as well as Code Change Match (CCM) and Line Type Prediction (LTP), which are proposed by us to probe the dynamic semantic-level information from the global and local perspectives, respectively. The three probing tasks are introduced in detail in section 3.2.2. We mainly investigate what kind of knowledge is encoded by the four PEFT methods, as well as making comparisons with FMFT and ZL, thereby providing explanations for their performance.

RQ5: How to combine the superiority of PEFT and FMFT for better domain adaptation? Based on the experimental results of the aforementioned research questions, we investigate more effective alternatives to the PEFT methods. With much fewer trainable parameters, it is difficult for PEFT to surpass FMFT too much, even worse in some complicated tasks, such as the CMG task. It is observed by no matter previous studies [95, 133] or our experimental evaluation in Section 3.3.1. Therefore, inspired by our layer-wise probing results, we explore whether there is a brand-new domain adaptation method that can integrate the advantages of PEFT and FMFT together, improving the performance while keeping efficiency.

3.2.2 Tasks and Datasets

Table 3.1. Dataset Statistics.

Tasks	Datasets	Training	Validation	Test
JIT-DP	JIT-Defects4J	16,374	5,465	5,480
CMG	Java	160,018	19,825	20,159
	C#	149,907	18,688	18,702
	C++	160,948	20,000	20,141
	Python	206,777	25,912	25,837
	JavaScript	197,529	24,899	24,773
Probing Tasks	TYP	600	200	200
	CCM	600	200	200
	LTP	600	200	200

3.2.2.1 Code-Change-Related Tasks

To evaluate the performance of PEFT methods in code change learning, we consider two emerging code-change-related tasks, just-in-time defect prediction and commit

message generation, which have been popularly researched in recent years [40, 50, 223, 239].

Just-In-Time Defect Prediction: The change of code may damage software quality, so it is crucial to discover defects as early as possible [239]. JIT-DP aims to identify defective code changes when they are just committed, and return judgments. It can be seen as a binary classification task since the outputs are two classes, namely defective or not.

The experimental dataset we used in JIT-DP is JIT-Defects4J [154], which is extended from LLTC4J [87] with extra buggy commits. In addition, for each code change, JIT-Defects4J extracts the 14 change-level Expert Features (EF) proposed by Kamei et al. [108], which can measure the code change from five dimensions, i.e., diffusion, size, purpose, history, and experience. It is widely adopted in previous studies [1, 33, 127, 154]. Following their setup, we evaluate different methods in two settings, one for directly using the learned code change representations to predict, and another for incorporating the expert features.

Commit Message Generation: Commit message summarizes the intent and content of a code change, which is helpful for developers to understand programs quickly in software maintenance [40, 223]. However, writing high-quality commit messages is time-consuming, and thus many are left empty [48]. Therefore, taking changed codes as inputs, the CMG task aims to generate commit messages automatically and has become a hot topic in the software engineering domain.

In the CMG task, we choose the Multi-language Commit Message Dataset (MCMD) for experiments [184]. MCMD contains five PLs, including Java, C#, C++, Python, and JavaScript. For each PL, it collects the top 100 starred projects from GitHub and randomly retains 450,000 commits. It has been a widely used large-scale dataset for the CMG task and has been adopted by numerous prior studies [127, 174, 185, 223]. Following the common operation, noisy data containing multiple files or files that cannot be parsed are filtered out. The statistics of MCMD are shown in Table 3.1.

3.2.2.2 Probing Tasks

Probing tasks are auxiliary diagnostic tasks that are used to understand what specific property is encoded in the language model weights. It has been widely utilized in NLP research [105]. In this chapter, we employ three probing tasks that are related to code

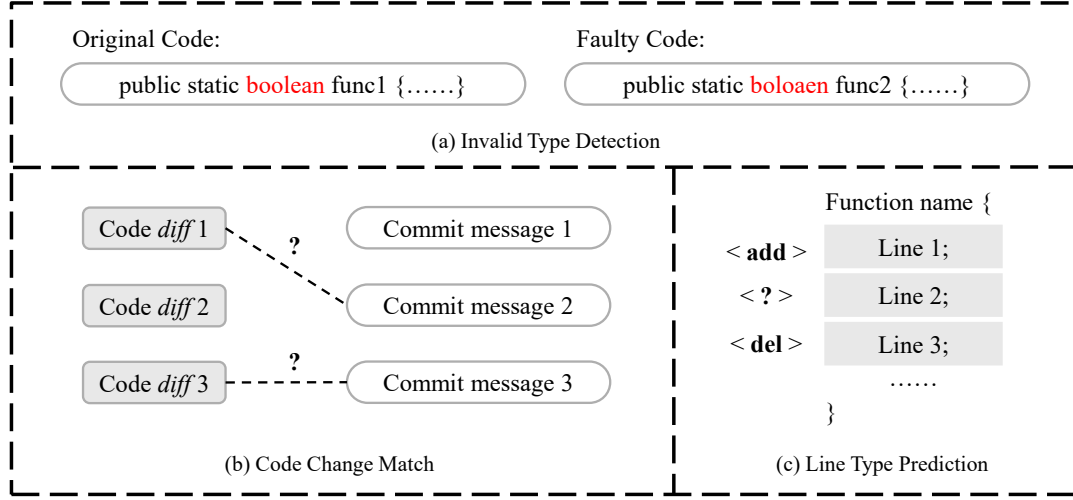


Figure 3.2. Illustrations of probing tasks.

properties from different aspects. For their datasets, we split them into the training set, validation set, and testing set by the ratio of 6/2/2 uniformly.

Invalid TYPE detection (TYP), proposed by Karmakar et al. [110], aims to measure the static code semantics understood by PLMs. As Figure 3.2(a) shows, it falsifies the data types of some code snippets deliberately, and expects PLMs to distinguish the invalid samples from the others. We introduce the probing task here to explore whether PLMs still encode static code semantics when they are fine-tuned in code-change-related tasks. We also adopt the dataset containing 1,000 samples from Karmakar et al. [110]. It is gathered from 50K-C [146], a dataset of compilable Java projects, and balances the valid and invalid classes.

Code Change Match (CCM), probing the dynamic semantic-level information, is proposed by us. Different from TYP, designed for code semantics in one code snippet, CCM takes a pair of changed codes and a commit message as inputs, inferring whether the commit message corresponds to the given code change. It is shown in Figure 3.2(b). If the judgment is correct, it means PLMs can identify the code change semantics from a global perspective. We construct a dataset for CCM from FIRA [45], which collects commits from the top 1,000 popular Java projects in GitHub. Similar to the dataset of TYP, we keep 500 correct matches and randomly produce 500 false matches.

Line Type Prediction (LTP), proposed by us as well, is also to determine whether PLMs can encode dynamic semantic-level information. As code changes are composed of added, kept, and deleted lines, LTP aims to predict the line type when given the

surrounding context, as illustrated in Figure 3.2(c). Considering LTP is designed for predicting specific line types, it measures PLMs’ ability to comprehend the semantics of code changes from a local perspective. We also build the dataset for LTP from FIRA [45]. For each commit, to measure the perception of code changes, we only retain the added and deleted lines, then randomly mask a line type for prediction. The dataset also incorporates 1,000 samples, and Table 3.1 demonstrates its statistics.

Although probing tasks could be designed with finer granularity, for example, by evaluating whether PLMs can distinguish between adding a variable declaration, modifying a return value, or deleting an unnecessary function call, however, such complicated classification tasks might intensify interpretability problems [38], deviating from the original intent of probing tasks, i.e., focusing on simple PL properties of code snippets [110]. An alternative solution is to deconstruct the above multi-category classification into several different probing tasks for more fine-grained exploration. As such, we leave them for future work and begin with the simple probing tasks proposed in this section for analysis.

3.2.3 Pre-trained Language Models and Baselines

There are a total of seven PLMs that we use to evaluate the performance of FMFT and the four PEFT methods. In the JIT-DP task, we conduct experiments on six PLMs, where only encoders are used for the purpose of code change representation. The selected PLMs have all been widely used and extensively studied in code-related tasks [133, 199], which facilitates the generalization and comparability of our experimental findings. We include CodeT5+ here because it is a more recent model whose encoder can be used independently, enabling fair and consistent experimentation with other PLMs in the JIT-DP task. For the CMG task, we choose CodeT5, CodeT5+, and Qwen2.5-Coder as the backbone model. The selection of CodeT5 and CodeT5+ is motivated by the need to align with the existing SOTA approach [127], while Qwen2.5-Coder is included due to its more recent release, strong performance in generation tasks, and a parameter scale comparable to that of the other PLMs in our study. The specific introduction for each model is listed below.

- **CodeBERT** [55]: It is an encoder-only PLM with 125M parameters, pre-trained on CodeSearchNet [98] which contains 2M bimodal data.

- **GraphCodeBERT** [76]: It has the same architecture and parameter size as CodeBERT, while pre-trained with edge prediction and node alignment to learn from data flow.
- **PLBART** [4]: It is pre-trained on Java and Python datasets, as well as natural language descriptions. It is an encoder-decoder architecture that has 140M parameters.
- **UniXcoder** [75]: It is a unified encoder-decoder PLM with 125M parameters, and can be flexibly modified to an encoder-only or decoder-only architecture.
- **CodeT5** [207]: It contains 220M parameters, and is a representative PLM of the encoder-decoder architecture for its good performance in generation tasks.
- **CodeT5+** [205]: It is based on CodeT5 and enhanced with a mixture of pre-training objectives to flexibly adapt to various downstream tasks. We use the 220M-sized variant to maintain consistency with CodeT5.
- **Qwen2.5-Coder** [97]: It is a new series of decoder-only language models demonstrating top-tier performance in coding tasks. We employ the 500M-parameter model in this chapter.

We also compare the performance of FMFT and PEFT methods with the following baselines.

- **JITLine** [163]: It is a baseline of JIT-DP, which represents code changes by bag-of-tokens features and constructs classifiers like SVM to predict defective commits.
- **JITFine** [154]: It uses CodeBERT as an encoder, and incorporates the encoded code change representations with extra expert features, achieving a substantial improvement in the JIT-DP task.
- **CC2Vec** [89]: It is a baseline of JIT-DP, which utilizes a hierarchical attention network and emphasizes modeling the correlation between removed codes and added codes.
- **CodeReviewer** [125]: It is proposed for code review activities, but can be extended to the CMG task. It proposes four pre-training tasks to improve CodeT5’s [207] capacity for understanding code changes.
- **CCT5** [127]: It is the state-of-the-art baseline for both JIT-DP and CMG. It also pre-trains CodeT5 [207] with code-change-related corpus and objectives.

3.2.4 Evaluation Metrics

3.2.4.1 Just-In-Time Defect Prediction

Owing to the imbalance learning nature of the JIT-DP task, prior works [127, 154] normally use the F1-score and the Area Under the receiver operating characteristic Curve (AUC) as the evaluation metrics. Hence, we follow their settings for evaluation. F1-score combines the precision and recall scores of a model, and AUC reflects the distinguishing ability of models between positive and negative classes. In the JIT-DP task, we treat defect-prone code changes as positive samples.

3.2.4.2 Commit Message Generation

For the CMG task, we evaluate the generated commit messages using three metrics: BLEU [158], Meteor [15], and Rouge-L [128]. BLEU calculates the n-gram precision between the generated text and ground truth text. Meteor takes into account the precision, recall, and fluency of the generated text. Rouge-L focuses on the longest common subsequence so that it evaluates more about the word order. Specifically, for the BLEU metric, we choose a smoothed BLEU-4 score for evaluations in this chapter.

3.2.4.3 Probing Tasks

Due to probing tasks being classification tasks and their class distributions being balanced, we use accuracy as the evaluation metric, which measures the correctness of predictions.

3.2.5 Implementation Details

The experiments were conducted on an Ubuntu GPU server with four RTX 3090 24GB GPUs. Our code is implemented by the deep learning framework PyTorch¹. The PLMs we used are from Huggingface², and we keep their default hyperparameter settings. For the AT, we implement it with the OpenDelta Library³, and we set the intermediate dimension m to 128 because it strikes an effective balance between performance and

¹<https://pytorch.org/>

²<https://huggingface.co/models>

³<https://opendelta.readthedocs.io/en/latest/>

parameter efficiency. LoRA, PT, and PreT are implemented by the PEFT Library⁴. Based on prior findings on optimal LoRA rank [95], we set the LoRA rank $r = 8$. After sweeping over prompt lengths 10, 20, 30, 40, 50, we find the optimal prompt length for the latter two methods to be 50. An elaborate analysis of these PEFT-related parameters will be discussed in section 3.3.1.4. In the JIT-DP task, we adjust the learning rate from $5e-5$ to $5e-2$ for all PLMs. In the CMG task, we set the learning rate from $5e-5$ to $1e-2$ for different methods after a grid search. Besides, we set the maximum training epoch to 10, and adopt early stopping with a patience of 5.

3.3 Experimental Results

3.3.1 RQ1: Quantitative Evaluation

3.3.1.1 Just-In-Time Defect Prediction

We first evaluate the performance of the four PEFT methods in the JIT-DP task. We employ six prevalent PLMs to exhibit comprehensive comparisons. Following previous studies [127, 154], we also conduct experiments on two settings. In the first setting (w/o EF), encoded code change representations are extracted and then fed into a linear classifier, predicting defectiveness or not. The second setting (w/ EF) combines the encoded representations with extra expert features [108], and the concatenated feature vectors are used to predict. Table 3.2 shows the detailed evaluation results. Digits in bold denote the best value among all results. Table cells in a light grey background denote the best result among different backbone models under the same approach. Because JITLine and CC2Vec do not utilize expert features in their original papers, their performances in the second setting are omitted.

When comparing the performance of FMFT with AT and LoRA, it can be noticed that no matter in which setting, their average performances are always better than FMFT. To be specific, in the first setting (w/o EF), AT obtains an improvement of 3.17% and 1.53% in terms of F1 and AUC, while LoRA increases by 3.17% and 1.29%. When incorporating extra expert features, the improvements become significant. AT surpasses fine-tuning by 16.43% and 2.06% for F1 and AUC, respectively. LoRA increases 21.60% and 2.75% accordingly. The larger performance gains reflect a complementary synergy between expert features and code-change representation learning.

⁴<https://huggingface.co/docs/peft/index>

Specifically, expert features act as explicit anchors that guide the lightweight PEFT parameters, enabling PLMs to focus more effectively on aligning high-level software patterns with fine-grained semantic modifications within code diffs. However, as for the PT and PreT, they underperform FMFT in the first setting. As Table 3.2 indicates, when excluding expert features, PT decreases 11.11% and 7.63% in terms of F1 and AUC, while PreT declines 6.88% and 5.99% accordingly compared with FMFT. In the second setting, which incorporates extra expert features, it can be observed that CodeBERT, GraphCodeBERT, and UniXcoder perform better than FMFT when using PT and PreT, while the other three encoder-decoder models perform worse. Since only the encoder is utilized in these encoder-decoder models, they have fewer trainable parameters and limited representational capacity, which results in inferior performance when applying PT and PreT. Meanwhile, this highlights that the parameter scale of PT and PreT is insufficient to fully handle the complexities of dynamic code comprehension tasks. However, their performance can improve when augmented with additional expert features to aid PLMs’ understanding of the tasks.

The above results indicate that AT and LoRA show their effectiveness in the JIT-DP task, but PT and PreT may underperform FMFT as their representation abilities degrade when only updating quite a few parameters. Besides, when introducing extra expert features, the results also show that the straightforward features are prone to being utilized by PEFT methods, leading to noteworthy improvements. A potential explanation is that PEFT methods with very few parameter tuning are more capable of learning straightforward features like expert features, thereby obtaining substantial improvement. In addition, it can be observed that AT and LoRA consistently show their improvements in diverse PLMs, which manifests their generalization abilities.

When compared with other baselines, AT and LoRA achieve state-of-the-art results that reach 0.496 and 0.518 in terms of F1 in the setting with extra expert features. They surpass CCT5 [127], the current SOTA method, by 5.08% and 9.75%, respectively. Although the average performances of PT and PreT are lower than CCT5, they still can achieve surprising results when employing UniXcoder, reaching 0.517 and 0.566 for the F1 metric, and also performing better than CCT5. Considering that CCT5 has been pre-trained on code-change-related corpus with various objectives, it indicates the prominent advantage of the four PEFT methods because they only need to fine-tune a very small portion of the parameters. For ZL and ICL, they show substantially worse performance compared to PEFT methods and FMFT, indicating that fine-tuning remains essential for the effective application of PLMs.

Table 3.2. Evaluation results of JIT-DP on the JIT-Defects4J dataset.

Models		w/o EF		w/ EF	
		F1	AUC	F1	AUC
End-to-end	JITLine	0.261	0.802	-	-
	JITFine	0.375	0.856	0.431	0.881
	CC2Vec	0.248	0.791	-	-
Pre-trained	CCT5	0.451	0.871	0.472	0.882
FMFT	CodeBERT	0.382	0.849	0.419	0.867
	GraphCodeBERT	0.390	0.869	0.361	0.861
	PLBART	0.329	0.834	0.412	0.869
	UniXcoder	0.401	0.847	0.480	0.889
	CodeT5	0.398	0.859	0.433	0.878
	CodeT5+	0.366	0.851	0.452	0.878
	Avg.	0.378	0.852	0.426	0.874
LoRA	CodeBERT	0.380	0.860	0.512	0.899
	GraphCodeBERT	0.379	0.869	0.523	0.898
	PLBART	0.393	0.862	0.512	0.890
	UniXcoder	0.388	0.852	0.527	0.907
	CodeT5	0.376	0.867	0.519	0.895
	CodeT5+	0.425	0.867	0.515	0.898
	Avg.	0.390	0.863	0.518	0.898
AT	CodeBERT	0.408	0.864	0.502	0.894
	GraphCodeBERT	0.377	0.866	0.525	0.897
	PLBART	0.377	0.858	0.494	0.890
	UniXcoder	0.402	0.871	0.515	0.901
	CodeT5	0.404	0.866	0.522	0.892
	CodeT5+	0.371	0.862	0.418	0.875
	Avg.	0.390	0.865	0.496	0.892

Continued on next page...

Models		w/o EF		w/ EF	
		F1	AUC	F1	AUC
PT	CodeBERT	0.370	0.829	0.493	0.880
	GraphCodeBERT	0.333	0.815	0.483	0.886
	PLBART	0.332	0.773	0.403	0.812
	UniXcoder	0.371	0.852	0.517	0.898
	CodeT5	0.310	0.717	0.382	0.768
	CodeT5+	0.297	0.737	0.346	0.779
	Avg.	0.336	0.787	0.437	0.837
PreT	CodeBERT	0.370	0.846	0.510	0.890
	GraphCodeBERT	0.398	0.866	0.526	0.900
	PLBART	0.348	0.791	0.410	0.798
	UniXcoder	0.422	0.858	0.566	0.904
	CodeT5	0.299	0.726	0.336	0.778
	CodeT5+	0.277	0.721	0.360	0.782
	Avg.	0.352	0.801	0.451	0.842
ZL	CodeBERT	0.000	0.443	0.148	0.453
	GraphCodeBERT	0.090	0.485	0.141	0.486
	PLBART	0.053	0.476	0.143	0.461
	UniXcoder	0.134	0.498	0.148	0.520
	CodeT5	0.131	0.525	0.134	0.500
	CodeT5+	0.159	0.510	0.156	0.528
	Avg.	0.095	0.489	0.145	0.491
ICL	CodeBERT	0.026	0.514	0.152	0.486
	GraphCodeBERT	0.038	0.483	0.170	0.540
	PLBART	0.064	0.435	0.140	0.480
	UniXcoder	0.125	0.480	0.160	0.561
	CodeT5	0.128	0.501	0.136	0.526
	CodeT5+	0.159	0.493	0.158	0.512
	Avg.	0.090	0.484	0.153	0.517

Finding 1: In the JIT-DP task, all four PEFT methods can surpass FMFT when incorporating extra expert features. Notably, AT and LoRA even obtain improvements of 5.08% and 9.75% in terms of F1 when compared with SOTA baselines.

3.3.1.2 Commit Message Generation

For the commit message generation task, we employ CodeT5, CodeT5+, and Qwen2.5-Coder as the backbone PLMs to explore the performance of PEFT methods. Considering there are five sub-datasets containing diverse PLs, namely Java, C#, C++, Python, and JavaScript, we fine-tune PLMs for each PL separately and calculate the average performance among the five PLs. The evaluation results are shown in Table 3.3. For CodeReviewer and CCT5, since their performance on Meteor and Rouge-L is not reported in the original papers, we retrained them using their source code and report the full evaluation results.

From Table 3.3, when using CodeT5 and CodeT5+ as backbones, we could observe mild degradation with AT and LoRA compared to FMFT. The degradation is consistent across the five PLs. For example, when applied to CodeT5, AT and LoRA achieve average BLEU scores of 21.60 and 19.85, respectively, both of which are slightly below the FMFT’s result of 22.16. However, PT and PreT significantly underperform FMFT in the CMG task, which is in agreement with the findings in previous studies [95, 133]. For the decrements of PEFT methods, we conjecture the reasons mainly contain: (i) In the architecture of CodeT5 and CodeT5+, AT, LoRA, PT, and PreT only modify 4.10%, 0.40%, 0.03%, and 0.41% of the full parameters, which are quite a small portion of parameters. (ii) Compared with classification tasks, e.g., the JIT-DP task, generation tasks, such as the CMG task, carry a high complexity and difficulty. As such, the CMG task needs more parameter tuning for knowledge adaptation. (iii) As for the significant underperformances of PT and PreT, because the two methods only optimize the representation of prompt tokens but ignore other input tokens, and PT only involves parameters in the first layer, we argue that it is not enough to transfer pre-trained knowledge to the CMG task, and thus leads to the degradation. Their underperformance has also been observed in prior studies [133, 135], underscoring both the inherent complexity of dynamic code comprehension tasks and the inadequacy of these methods’ limited parameter budgets in capturing the sophisticated knowledge adaptation required by such tasks.

When applying FMFT, AT, and LoRA to Qwen2.5-Coder, their performance is observed to be lower than that achieved with CodeT5 and CodeT5+. Since Qwen2.5-Coder was pre-trained with next token prediction and fill-in-the-middle without any code diff semantic learning, making them less capable of modeling the transformation from code changes to commit messages. In contrast, CodeT5 and CodeT5+ were pre-trained with identifier-aware denoising, simulating code snippet modifications, thereby making them learn to restore masked identifiers or code blocks, which is directly associated with code change scenarios. Nevertheless, we can still observe similar conclusions to those of CodeT5 and CodeT5+, namely, AT and LoRA achieve performance on par with FMFT. For PT and PreT, they perform poorly and fail to generate high-quality commit messages, further revealing their limitations when applied to PLMs.

When compared with CodeReviewer and CCT5, AT and LoRA perform neck-to-neck when using CodeT5 and CodeT5+ as backbones, especially dominating on sub-datasets of Java, C++, and JavaScript. By contrast, PT and PreT perform weakly. It is well known that pre-training on large-scale datasets is not practical for everyone, but AT and LoRA can obtain similar results by modifying only a small portion of the parameters, indicating their practical value. Additionally, in line with the findings from the JIT-DP task, ZL and ICL continue to perform significantly worse in the CMG task, highlighting their limitations without fine-tuning.

Finding 2: In the CMG task, PEFT methods perform lower than or on par (at best) with FMFT. Except for FMFT, AT and LoRA still perform the best among other domain adaptation approaches. Besides, using CodeT5 and CodeT5+ as backbones, AT and LoRA perform neck-to-neck with SOTA baselines.

3.3.1.3 Efficiency

In addition to evaluating the performance, we also evaluate the efficiency of PEFT methods. Taking CodeT5 in the CMG task as a case study, we analyze the training time and memory consumption of FMFT and the four PEFT methods. The results are illustrated in Figure 3.3. We report the average training time of each epoch and the occupied memory size in total. The devices we use are elaborated in Section 3.2.5. It can be observed that AT, LoRA, and PT can save approximately 15-30 minutes each epoch, and decrease memory consumption of about 4,000-5,000 MB. PreT can save more training time and computational resources, which are about 1 hour and 16,000 MB of memory size.

Table 3.3. Evaluation results of CMG on the MCMD dataset. M is short for Meteor. R is short for Rouge-L. Digits in bold denote the best value among all results. Table cells in a light grey background denote the best result among different approaches under the same backbone model.

Models	Java			C#			C++			Python			JavaScript			Avg.			
	BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R	
CodeReviewer	21.13	12.15	26.69	21.85	13.82	28.84	15.51	10.55	20.50	18.42	15.72	29.95	20.16	13.57	26.32	19.41	13.16	26.46	
CCT5	23.80	14.27	30.52	23.64	15.80	29.76	17.22	12.17	23.22	20.25	14.10	27.55	24.42	16.30	31.78	21.87	14.53	28.57	
FMFT	24.34	15.28	32.15	22.11	13.96	28.93	18.78	13.38	25.70	20.55	14.97	28.96	25.03	17.30	32.99	22.16	14.98	29.75	
LoRA	21.65	14.14	29.17	18.39	11.89	24.84	16.97	12.60	23.36	18.76	14.02	26.57	23.48	16.18	31.17	19.85	13.77	27.02	
AT	23.95	15.25	31.60	20.69	13.36	27.49	18.33	13.39	25.31	20.12	14.75	28.44	24.92	17.07	32.75	21.60	14.76	29.12	
CodeT5	PT	11.69	7.89	16.22	10.67	6.71	14.71	9.93	7.53	13.61	10.97	8.17	14.98	13.02	9.28	18.41	11.26	7.92	15.59
PreT	16.10	10.07	21.18	13.40	8.08	17.92	12.74	9.35	17.27	14.60	11.02	20.70	17.66	12.16	23.73	14.90	10.14	20.16	
ZL	0.79	1.25	1.26	0.84	1.37	1.05	1.14	1.91	1.34	1.22	2.16	1.43	0.86	1.45	1.57	0.97	1.63	1.33	
ICL	0.73	1.20	2.43	0.87	1.03	2.14	0.94	1.14	1.54	1.05	1.04	3.63	0.67	1.20	3.02	0.85	1.12	2.55	
FMFT	24.82	15.87	32.88	22.51	14.37	29.76	19.28	14.11	26.55	21.11	15.65	29.74	26.08	17.67	34.10	22.76	15.53	30.61	
LoRA	21.74	14.09	29.25	18.40	11.73	25.03	17.05	12.46	23.13	18.60	10.97	26.86	22.88	15.76	30.42	19.73	13.00	26.94	
AT	23.87	15.42	31.88	21.04	13.43	27.83	18.14	13.28	25.15	19.95	15.08	28.44	25.23	17.47	33.53	21.65	14.94	29.37	
CodeT5+	PT	8.49	4.51	11.63	7.86	3.79	10.57	8.84	5.94	11.41	10.35	6.68	12.95	10.17	7.47	15.41	9.14	5.68	12.39
PreT	16.62	9.40	22.17	13.21	8.45	17.93	12.66	7.21	17.72	13.66	6.94	19.53	17.06	9.81	23.28	14.64	8.36	20.13	
ZL	0.99	1.15	0.53	1.12	1.13	0.52	1.28	1.13	0.54	1.29	1.14	0.62	1.24	1.14	0.56	1.18	1.14	0.55	
ICL	0.76	1.05	0.27	1.06	1.05	0.63	1.03	1.05	0.20	1.06	1.07	0.25	1.10	1.05	0.24	1.00	1.05	0.32	
FMFT	13.40	7.57	17.72	12.10	6.97	16.17	10.03	5.01	13.74	9.94	6.54	14.76	13.58	8.48	18.76	11.81	6.91	16.23	
LoRA	12.72	6.52	16.41	11.32	5.05	14.26	9.65	4.93	12.72	9.28	4.78	12.76	12.77	5.80	16.73	11.15	5.14	14.58	
AT	14.63	7.64	19.05	13.75	7.03	17.71	11.51	5.98	15.63	11.27	6.91	16.17	14.49	7.76	19.32	13.13	7.06	17.58	
Qwen2.5	PT	1.13	3.68	1.52	1.14	3.02	1.41	1.22	3.03	1.47	1.31	1.77	1.21	2.96	1.38	1.20	3.16	1.51	
-Coder	PreT	1.12	3.65	1.53	1.13	2.90	1.39	1.22	3.02	1.47	1.31	1.76	1.21	2.97	1.37	1.20	3.13	1.50	
ZL	1.13	3.68	1.52	1.14	3.02	1.41	1.22	3.03	1.47	1.31	1.77	1.21	1.21	2.96	1.38	1.20	3.16	1.51	
ICL	1.05	3.02	1.45	1.05	3.26	1.70	1.21	2.65	1.42	1.28	3.13	1.93	1.02	3.20	1.61	1.12	3.05	1.62	

**Qwen2.5
-Coder**

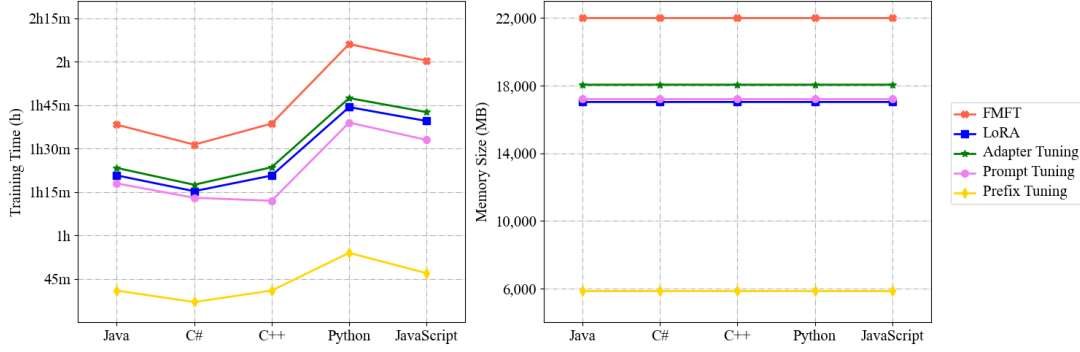


Figure 3.3. Comparison of training time and memory size in the CMG task, using CodeT5 as the backbone.

Finding 3: When compared with FMFT, PEFT methods consume less training time and memory size, which can save up to 1 hour and 16,000 MB at most in code-change-related tasks.

3.3.1.4 Hyperparameter Analysis

To conduct a thorough analysis of the impact of PEFT-related hyperparameters, specifically, the LoRA rank, intermediate dimension of AT, and prompt length for PT and PreT, we adjust these critical hyperparameters within appropriate ranges and observe the resulting performance variations. For the JIT-DP task, we employ CodeBERT as the backbone model under the w/ EF setting, while for the CMG task, we use CodeT5 on the Java dataset. As shown in Figure 3.4, the trends in performance with respect to the LoRA rank and the intermediate dimension of AT vary across tasks. In the JIT-DP task, the best performance for LoRA is achieved at a rank of 8. As the rank increases from 8 to 128, performance first declines and then improves. Similarly, AT exhibits fluctuating performance as the dimension changes, achieving its peak at a dimension of 128, consistent with our chosen hyperparameter configuration. In the CMG task, the performance of respective PEFT methods consistently improves as the LoRA rank and the intermediate dimension of AT increase. But it is worth noting that while larger values yield better performance, they also lead to a higher number of trainable parameters. Regarding the prompt length for PT and PreT, within our explored range, performance consistently shows an upward trend as the prompt length increases across both tasks, achieving the best results at a length of 50. The above analysis demonstrates that the hyperparameter settings of diverse PEFT methods are well optimized overall, thereby ensuring the reliability of this work.

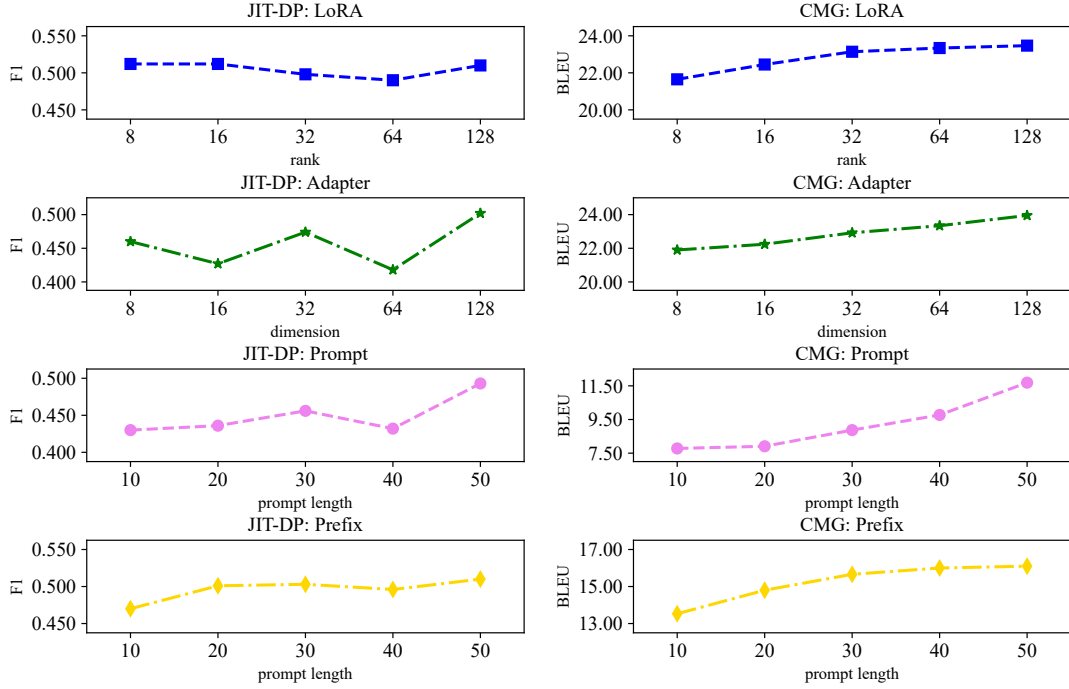


Figure 3.4. Hyperparameter analysis of the LoRA rank, the intermediate dimension for AT, and the prompt length for PT and PreT.

Finding 4: The impact of the LoRA rank and the intermediate dimension of AT varies across tasks, while within our explored range, the performance of PT and PreT consistently exhibits an upward trend as the prompt length increases.

3.3.2 RQ2: Performance in the Cross-lingual Scenario

In this section, we investigate the performance of PEFT methods in the cross-lingual scenario, comparing them with FMFT as well. Using CodeT5 as the backbone, we apply these methods to the CMG task as its dataset contains five PLs. In the cross-lingual scenario, PLMs are expected to generate commit messages about the target PL but are fine-tuned in a different source PL. It can be seen as a combination task of monolingual CMG and program translation [201], and is useful when the training data is limited or developers are only familiar with specific PLs. Table 3.4 illustrates the evaluation results, where rows represent the source PLs while columns represent the target PLs. We only report the BLEU metric due to the page limitation.

From Table 3.4, eliminating the diagonal values, i.e., the source PL and target PL are the same, we could observe consistent improvements over FMFT brought by AT

Table 3.4. Evaluation results of CMG in the cross-lingual scenario. CodeT5 is used as the backbone model for FMFT, LoRA, AT, PT, and PreT.

Models		Java	C#	C++	Python	JavaScript	Avg.
Java	FMFT	-	9.94	10.06	10.95	12.10	10.76
	LoRA	-	10.52	11.00	11.70	13.30	11.63
	AT	-	10.47	10.69	11.31	12.90	11.34
	PT	-	8.41	8.61	9.84	10.49	9.34
	PreT	-	8.72	9.16	9.97	10.57	9.61
C#	FMFT	10.23	-	9.97	10.68	13.18	11.02
	LoRA	11.01	-	11.04	11.57	13.21	11.71
	AT	10.65	-	10.89	11.44	13.34	11.58
	PT	9.39	-	9.77	10.54	11.40	10.28
	PreT	9.16	-	9.60	10.65	10.76	10.04
C++	FMFT	10.64	10.42	-	12.08	13.44	11.65
	LoRA	11.26	10.93	-	12.64	14.20	12.26
	AT	11.00	10.31	-	12.34	13.52	11.79
	PT	8.08	8.87	-	10.02	10.56	9.38
	PreT	8.10	7.91	-	10.20	9.87	9.02
Python	FMFT	9.78	10.14	10.47	-	13.24	10.91
	LoRA	10.20	10.54	10.99	-	13.79	11.38
	AT	10.00	10.55	11.03	-	13.69	11.32
	PT	8.24	8.42	9.60	-	10.19	9.11
	PreT	9.17	9.30	10.20	-	11.49	10.04
JavaScript	FMFT	9.62	9.90	9.75	10.93	-	10.05
	LoRA	11.37	10.73	11.20	12.30	-	11.40
	AT	10.65	9.96	10.63	11.88	-	10.78
	PT	8.95	8.56	8.81	9.88	-	9.05
	PreT	9.48	8.93	9.51	10.73	-	9.66

and LoRA in the cross-lingual scenario. Specifically, the average promotion of AT and LoRA can be up to 4.62% and 7.56%, respectively. A potential explanation is that FMFT, after updating all parameters, overfits to the training samples (i.e., the source PL). In contrast, LoRA and AT, which adjust only a small portion of parameters, preserve the multilingual general knowledge acquired during PLMs’ pre-training and further learn new PL-invariant knowledge from the training samples, enabling them to apply the above general knowledge to the testing set (i.e., the target PL) and outperform FMFT in cross-lingual scenarios. As for PT and PreT, their average performance

is still weaker than FMFT, but the declines are slight compared with the results in RQ1. Because, although they also maintain the previous PL-invariant knowledge from their backbone PLMs, their trainable parameters are extremely limited, making it insufficient to learn PL-invariant features from the training samples (i.e., the source PL). Therefore, PreT and PT do not reach LoRA and AT’s performance in the cross-lingual scenario, but still manifest a comparable performance to FMFT. Given the much lower computational resources required and the comparable or even better performance of PEFT methods, we strongly recommend using PEFT techniques for cross-lingual code change learning. The above findings bring unique insights. PEFT is particularly suitable for cross-lingual scenarios because it updates only a small portion of parameters, thereby substantially mitigating catastrophic forgetting of previously acquired general knowledge in backbone PLMs. Besides, partially trainable parameters also leave PEFT methods room to further acquire PL-invariant knowledge for follow-up transfer to unseen PLs, while this process remains effective for code change learning.

Finding 5: PEFT techniques perform either better or comparable to FMFT in cross-lingual scenarios. Considering their much lower computational resource consumption, PEFT is highly recommended.

3.3.3 RQ3: Performance in the Low-resource Scenario

In this section, we investigate how well the PEFT methods perform in the low-resource scenario. Considering the disparity in dataset sizes between the two tasks, we randomly sample 1,000, 5,000, and 10,000 training instances for each PL in the CMG task, and 100, 500, and 1,000, 2500, and 5000 samples for JIT-DP. We repeat experiments three times with different random seeds to mitigate the bias in data sampling on the results. In addition, we employ the Wilcoxon Signed-Rank Test (WSRT) [213] with a confidence level of 95% to conduct pairwise comparisons between FMFT and PEFT methods. The evaluation results are presented in Table 3.5 and Table 3.6, respectively. Owing to the page limit, we only report standard deviations under the Avg. column of Table 3.5. In contrast, in Table 3.6, standard deviations are listed across all columns. For PEFT methods that outperform FMFT with a statistically significant difference on certain metrics, we highlight them in red under those metrics. In contrast, if FMFT statistically significantly outperforms PEFT methods, we highlight them in green accordingly. For the remaining ones without statistically significant differences, we leave them intact (i.e., white background).

Table 3.5. Evaluation results of CMG in the low-resource scenario. M is short for Meteor. R is short for Rouge-L. CodeT5 is used as the backbone model for FMFT, LoRA, AT, PT, and PreT.

Samples	Models	Java			C#			C++			Python			JavaScript			Avg.		
		BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R	M	M	R
1,000	FMFT	10.32	7.19	14.23	9.81	6.33	13.56	9.04	7.02	12.42	10.73	7.99	14.57	10.71	8.25	15.57	10.12 $\pm_{0.79}$	7.36 $\pm_{0.19}$	14.07 $\pm_{0.38}$
	LoRA	9.47	6.88	13.34	9.22	5.59	12.45	8.93	6.89	12.11	10.31	7.81	13.53	11.09	8.07	16.08	9.80 $\pm_{0.24}$	7.05 $\pm_{0.10}$	13.50 $\pm_{0.17}$
	AT	10.08	6.80	13.90	9.43	6.14	13.19	8.64	7.15	11.99	9.19	6.88	12.25	9.78	7.93	14.77	9.42 $\pm_{0.12}$	6.98 $\pm_{0.20}$	13.22 $\pm_{0.31}$
	PT	5.97	3.53	7.92	6.80	3.86	8.99	7.65	5.44	10.27	8.17	5.52	10.52	6.98	4.59	9.61	7.11 $\pm_{0.17}$	4.59 $\pm_{0.24}$	9.46 $\pm_{0.42}$
	PreT	4.95	2.88	7.11	5.82	3.09	7.71	6.84	4.75	8.49	6.68	4.47	7.98	6.29	3.98	8.95	6.11 $\pm_{0.17}$	3.83 $\pm_{0.06}$	8.05 $\pm_{0.22}$
5,000	FMFT	14.32	9.57	19.44	12.84	7.59	17.03	11.84	8.85	16.03	12.77	9.89	18.16	16.03	11.08	21.82	13.56 $\pm_{0.34}$	9.39 $\pm_{0.42}$	18.49 $\pm_{0.55}$
	LoRA	14.22	9.51	19.27	12.36	7.76	16.80	11.16	8.40	15.48	12.72	9.80	17.73	15.25	10.75	20.96	13.14 $\pm_{0.15}$	9.25 $\pm_{0.17}$	18.05 $\pm_{0.24}$
	AT	14.33	9.48	19.50	12.90	8.10	17.59	11.47	8.82	16.04	13.00	9.88	18.15	15.50	11.02	21.50	13.44 $\pm_{0.28}$	9.46 $\pm_{0.28}$	18.56 $\pm_{0.35}$
	PT	7.35	5.06	10.50	7.24	4.47	9.94	8.13	6.35	11.28	9.10	6.80	12.11	9.03	6.56	12.90	8.17 $\pm_{0.22}$	5.85 $\pm_{0.20}$	11.35 $\pm_{0.36}$
	PreT	6.83	4.91	9.61	6.82	4.42	9.30	7.39	5.57	9.65	8.07	5.83	10.36	8.61	6.19	12.51	7.54 $\pm_{0.13}$	5.38 $\pm_{0.17}$	10.29 $\pm_{0.09}$
10,000	FMFT	17.19	11.25	23.15	14.15	8.63	19.22	13.05	9.65	18.19	14.76	11.28	21.22	18.05	12.75	24.78	15.44 $\pm_{0.02}$	10.71 $\pm_{0.04}$	21.31 $\pm_{0.13}$
	LoRA	16.29	10.86	22.01	13.80	9.11	18.82	12.73	9.63	17.56	14.28	11.07	20.43	17.51	12.44	23.95	14.92 $\pm_{0.04}$	10.62 $\pm_{0.05}$	20.55 $\pm_{0.08}$
	AT	16.77	10.85	22.50	14.28	9.13	19.07	12.70	9.55	17.31	14.91	11.40	21.86	17.66	12.29	24.08	15.26 $\pm_{0.12}$	10.64 $\pm_{0.18}$	20.96 $\pm_{0.05}$
	PT	7.62	4.82	10.47	7.42	4.81	10.41	8.16	6.74	11.86	9.59	6.95	12.55	9.56	7.07	13.98	8.47 $\pm_{0.38}$	6.08 $\pm_{0.11}$	11.85 $\pm_{0.42}$
	PreT	9.44	6.61	13.09	8.83	5.69	12.17	8.07	6.17	11.03	9.35	6.91	12.56	10.97	7.38	15.17	9.33 $\pm_{0.28}$	6.55 $\pm_{0.19}$	12.80 $\pm_{0.38}$

* A light green background indicates that FMFT outperforms the corresponding PEFT methods with statistical significance, while the remaining ones indicate no statistically significant differences.

As shown in Table 3.5, it can be observed that when reducing the training samples to 1,000, 5,000, and 10,000 for each PL ($<1/100$, $<1/20$, and $<1/10$ of the full-data setting), AT and LoRA still slightly lag behind FMFT on the CMG task. Moreover, the WSRT shows that the advantage of FMFT is statistically significant across almost all metrics. However, compared with the full-data setting, where FMFT always outperforms AT and LoRA (see RQ1), they exhibit relative superiority in the low-resource setting, with distinct strengths across different PLs. In particular, in the 5,000-sample setting, AT, on average, outperforms FMFT on the Meteor and Rouge-L metrics, although the WSRT does not demonstrate a statistically significant difference. As such, we conclude that AT and LoRA can achieve comparable or slightly inferior performance in low-resource settings. We speculate that CMG is a challenging code-change task that requires more parameter tuning for adaptation. Even though AT and LoRA inherently exhibit superiority in low-resource settings [133, 199], they still cannot achieve a better understanding of code changes with very limited training samples in such generative tasks. As for PT and PreT, they still show poor performance compared with FMFT, and the gaps narrow as the number of training samples decreases. Given their efficiency during tuning, they can be viable alternatives when training samples are very limited. We also compare the above methods with CodeT5 without a fine-tuned version, i.e., in ZL and ICL settings. Since the testing sets are fixed to the same as those in RQ1, and we only change the random seeds to sample from the training set, the results for the ZL and ICL settings can be found in Table 3.3. We find that both settings perform worse than PEFT/FMFT methods and lag behind them significantly, as the parameter size of CodeT5 is limited and lacks emerging abilities [210].

For the JIT-DP task, we define five settings with varying training samples, the largest of which (5,000 samples) is still less than one-third of the full-data setting. To reduce the influence of data sampling variability, we perform three independent experiments with distinct random seeds and report the average results. From Table 3.6, we can observe that most PEFT methods achieve comparable performance against FMFT under the w/o EF setting. PT performs the worst in most low-resource settings. When incorporating expert features (i.e., the w/ EF setting), both FMFT and PEFT show improvements, with PEFT’s gain being more substantial, and many PEFT methods start to surpass FMFT with a statistically significant margin (highlighted in red). This phenomenon is the same as that in RQ1, shown in Table 3.2. In addition, we find that these methods perform inconsistently when the training sample size is small. For example, LoRA, AT, PreT, and FMFT all have a chance to achieve the best performance

Table 3.6. Evaluation results of JIT-DP in the low-resource scenario. CodeBERT is used as the backbone model.

Samples	Models	w/o EF		w/ EF	
		F1	AUC	F1	AUC
100	FMFT	0.123 $\pm$ 0.118	0.691 $\pm$ 0.073	0.117 $\pm$ 0.131	0.681 $\pm$ 0.077
	LoRA	0.015 $\pm$ 0.014	0.671 $\pm$ 0.010	0.079 $\pm$ 0.076	0.667 $\pm$ 0.080
	AT	0.138 $\pm$ 0.124	0.708 $\pm$ 0.053	0.195 $\pm$ 0.152	0.708 $\pm$ 0.049
	PT	0.022 $\pm$ 0.015	0.672 $\pm$ 0.002	0.203 $\pm$ 0.110	0.664 $\pm$ 0.073
	PreT	0.130 $\pm$ 0.115	0.648 $\pm$ 0.047	0.209 $\pm$ 0.078	0.668 $\pm$ 0.067
500	FMFT	0.269 $\pm$ 0.108	0.795 $\pm$ 0.014	0.296 $\pm$ 0.073	0.795 $\pm$ 0.023
	LoRA	0.271 $\pm$ 0.076	0.797 $\pm$ 0.016	0.313 $\pm$ 0.104	0.818 $\pm$ 0.021
	AT	0.301 $\pm$ 0.047	0.801 $\pm$ 0.015	0.308 $\pm$ 0.073	0.807 $\pm$ 0.021
	PT	0.117 $\pm$ 0.090	0.751 $\pm$ 0.018	0.307 $\pm$ 0.058	0.779 $\pm$ 0.030
	PreT	0.254 $\pm$ 0.092	0.777 $\pm$ 0.033	0.326 $\pm$ 0.033	0.796 $\pm$ 0.035
1,000	FMFT	0.284 $\pm$ 0.054	0.818 $\pm$ 0.019	0.345 $\pm$ 0.022	0.813 $\pm$ 0.011
	LoRA	0.281 $\pm$ 0.062	0.810 $\pm$ 0.010	0.358 $\pm$ 0.049	0.837 $\pm$ 0.009
	AT	0.305 $\pm$ 0.069	0.809 $\pm$ 0.032	0.259 $\pm$ 0.080	0.828 $\pm$ 0.011
	PT	0.110 $\pm$ 0.065	0.764 $\pm$ 0.021	0.292 $\pm$ 0.077	0.804 $\pm$ 0.028
	PreT	0.326 $\pm$ 0.046	0.814 $\pm$ 0.018	0.356 $\pm$ 0.086	0.831 $\pm$ 0.029
2,500	FMFT	0.279 $\pm$ 0.038	0.820 $\pm$ 0.010	0.307 $\pm$ 0.055	0.831 $\pm$ 0.021
	LoRA	0.290 $\pm$ 0.035	0.826 $\pm$ 0.005	0.384 $\pm$ 0.012	0.857 $\pm$ 0.015
	AT	0.281 $\pm$ 0.056	0.817 $\pm$ 0.010	0.266 $\pm$ 0.028	0.827 $\pm$ 0.004
	PT	0.094 $\pm$ 0.012	0.765 $\pm$ 0.022	0.289 $\pm$ 0.022	0.801 $\pm$ 0.002
	PreT	0.268 $\pm$ 0.039	0.800 $\pm$ 0.006	0.255 $\pm$ 0.023	0.812 $\pm$ 0.026
5,000	FMFT	0.311 $\pm$ 0.010	0.843 $\pm$ 0.003	0.349 $\pm$ 0.046	0.850 $\pm$ 0.010
	LoRA	0.323 $\pm$ 0.024	0.847 $\pm$ 0.004	0.463 $\pm$ 0.020	0.873 $\pm$ 0.009
	AT	0.297 $\pm$ 0.029	0.828 $\pm$ 0.006	0.291 $\pm$ 0.010	0.841 $\pm$ 0.012
	PT	0.105 $\pm$ 0.026	0.814 $\pm$ 0.007	0.396 $\pm$ 0.030	0.864 $\pm$ 0.003
	PreT	0.323 $\pm$ 0.037	0.833 $\pm$ 0.002	0.390 $\pm$ 0.029	0.853 $\pm$ 0.008

* A light green background indicates that FMFT outperforms the corresponding PEFT method with statistical significance, while a light red background indicates the reverse, i.e., the PEFT method significantly outperforms FMFT. The remaining ones indicate no statistically significant differences.

on certain metrics when the training sample size is limited to 100, 500, or 1,000. Besides, their standard deviations are relatively larger than those in 2,500 or 5,000 training samples, suggesting that the training is possibly insufficient and their performance is highly affected by data sampling. When it comes to 2,500 and 5,000 training samples,

we find that LoRA consistently maintains its advantage over diverse counterparts, including FMFT. To be specific, LoRA surpasses them on average by 55.06%–55.97% in F1 and 2.13%–3.26% in AUC with the w/o EF setting. As for the w/ EF setting, LoRA still maintains its superiority by 31.85%–38.23% in F1 and 2.47%–4.82% in AUC on average. This indicates that LoRA indeed outperforms other PEFT methods and FMFT on the JIT-DP task, provided it is sufficiently trained. Similarly, the ZL and ICL settings still perform the worst and lag behind the above methods a lot, as shown in Table 3.2.

Finding 6: In the CMG task, almost all PEFT methods fall short of FMFT in the low-resource scenario. In contrast, for the JIT-DP task in the w/ EF setting, PEFT methods can overall achieve superior or at least comparable performance to FMFT when training data is limited.

3.3.4 RQ4: Probing Tasks

In this section, we utilize three probing tasks, namely TYP, CCM, and LTP tasks, to explore what code properties are encoded in PLMs, thereby making an explanation for the above experimental results. TYP task relates to static semantic information, while CCM and LTP tasks are associated with dynamic semantic information from global and local perspectives. We reused CodeBERT and CodeT5 trained with FMFT, four PEFT methods, and ZL settings for JIT-DP and CMG tasks in RQ1 for layer-wise probing experiments. Considering the PLM’s parameters in ZL and ICL are the same, as they are all fixed without training, we only consider the former setting for probing experiments. For each probing task, after tuning on their respective training set, we extract the encoded features from each transformer layer and then feed them into a simple linear classifier to predict their categories [110] on their testing sets. The classifier has no hidden units, so the performance of probing tasks prohibitively depends on the feature vectors. The probing results are illustrated in Figure 3.5, where the dashed lines indicate the average performance of all layers.

From the first row of Figure 3.5, we could derive several insightful findings for the JIT-DP task: (i) The average performances of PEFT methods can always outperform FMFT in all three probing tasks, except for AT and PT that slightly underperform in the LTP task. (ii) Comparing with the advantages obtained by the four PEFT methods in the TYP task, the promotions on CCM and LTP are not dramatic. (iii) The ZL approach achieves the best performance in TYP, but it performs worse than FMFT and PEFT

methods on CCM and LTP in most cases. The first finding is expected and explains that the four PEFT methods learn more about code-change-related semantics from both global and local perspectives while keeping the original understanding ability of static code semantics. The second finding indicates that learning dynamic code semantics is more difficult than static semantics. Considering that code-change-related tasks are more complex, the result is reasonable. The third finding is also expected. Since TYP is a probing task related to static code semantics, fine-tuning on code-change-related downstream tasks tends to reduce the model’s capacity for understanding static code properties. In contrast, CCM and LTP probe dynamic code semantics, which are less directly captured by pre-training. As a result, the ZL approach performs poorly in these aspects, while FMFT and PEFT methods demonstrate superior adaptability in most cases. It can be concluded that the probing results are overall consistent with previous findings and explain the efficacy of PEFT methods.

For the CMG task, as shown in the second row of Figure 3.5, we found that no one approach can dominate all three probing tasks. The ZL setting still performs best in the TYP task, as PLMs’ parameters were not adapted to either JIT-DP or CMG task under this approach, thereby keeping its original performance on static code understanding. As for probing tasks of dynamic perspective, AT dominates on the CCM task, while FMFT excels on the LTP task. Hence, it is hard to say which method leads PLMs to learn more about dynamic code semantics during adaptation. These probing results also reflect the experimental evidence in RQ1 that, although AT and LoRA underperform FMFT slightly, their performance gaps are not significant. However, as for PreT and PT, they always perform the worst in both RQ1 and probing experiments on average. Although the average performances of FMFT and PEFT methods are similar, their layer-wise probing results show distinctive advantages. For example, in the CCM task, AT and LoRA perform much better than FMFT in the top layers (layers 6-12) but significantly decrease in the bottom layers (layers 1-5). This finding enlightens us to combine the layer-wise advantages of different tuning methods to further boost domain adaptation performance from a fine-grained perspective in the following sections.

Finding 7: Probing tasks show that PEFT methods can benefit the understanding of dynamic code semantics in the JIT-DP task. In the CMG task, the dynamic understanding gap between AT, LoRA, and FMFT is not evident on average. Nonetheless, PEFT methods carry distinctive advantages against FMFT in certain layers, making their combination in domain adaptation a reasonable attempt.

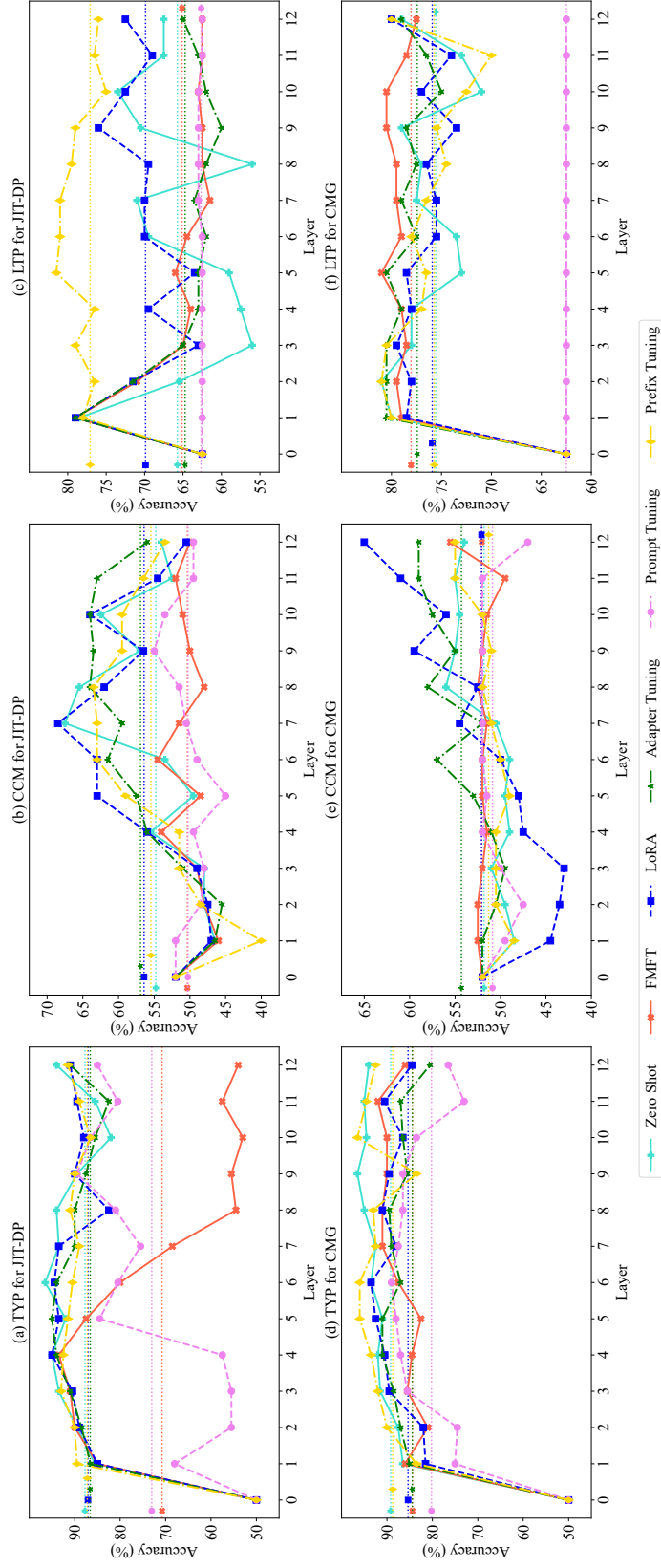


Figure 3.5. Accuracy of probing tasks by layers.

3.3.5 RQ5: Self-adaptive Efficient Layer-specific Tuning

In RQ1, the evaluation results reveal that although efficient, PEFT methods perform either comparable or worse than FMFT in the CMG task. Inspired by the layer-wise probing results in RQ4, we attempt to combine the distinctive advantages of PEFT and FMFT by extending PLMs’ parameters of certain layers to be trainable in PEFT methods, consequently achieving both high performance and efficiency. To that end, we propose **Pasta_K**, which stands for self-adaptive efficient layer-specific tuning for pre-trained models in code change learning. K is a hyperparameter that represents the number of FMFT layers. Given a specific PEFT method, Pasta_K first determines the location of the PEFT layers and FMFT layers by the layer-wise probing results of either CCM or LTP. For example, to combine LoRA and FMFT in the CMG task, we use CCM (see Figure 3.5(e)) for location determination, as it presents more obvious layer-wise pros and cons of LoRA and FMFT than LTP (see Figure 3.5(f)), where LoRA performs better at top layers ($\text{layer\#} \geq 5$), while FMFT excels at the bottom layers ($\text{layer\#} \leq 4$). Given the pre-defined FMFT layer number (K), Pasta_K assigns the bottom K layers to be trainable with FMFT, while tuning the remaining layers with LoRA. As for other PEFT methods, we follow the above principle to implement Pasta_K . Since the layer assignments for FMFT and PEFT in Pasta_K are determined from probing results, which can be automatically conducted in practice, the system is self-adaptive. In experiments, since the backbone models we used have 12 layers in total, we uniformly set K to 0, 3, 6, 9, and 12 for evaluation, where 0 and 12 denote **Pasta_K** degenerate to its corresponding PEFT and FMFT, respectively, thereby using K to simulate the constraint on the number of full-parameter trainable layers, i.e., the resource constraint. Our experiments in the CMG task are conducted on the Java dataset. The experimental results are shown in Table 3.7.

For PT, as its poor performance in LTP (see Figure 3.5(f)), we can only summarize its layer-wise advantages from CCM (see Figure 3.5(e)). As can be seen, PT generally performs neck-to-neck with FMFT at the top layers ($\text{layer\#} \geq 4$), while underperforming FMFT at the bottom layers ($\text{layer\#} < 4$). Thus, Pasta_K extends the bottom K layers to be trainable with FMFT to improve the ability of domain adaptation. Results are shown in Table 3.7. Pasta_K significantly improves the performance of PT, achieving a maximum enhancement of 108.47%, 99.37%, and 100.31% in terms of BLEU, Meteor, and Rouge-L, respectively, and also surpassing FMFT, when K is set to 6. By integrating FMFT-learned code change knowledge with PEFT, Pasta_K prevents catastrophic

Table 3.7. Evaluation results of $Pasta_K$ in the CMG task. We use the Java dataset for evaluation. The improving ratios of $Pasta_K$ on performance are compared with corresponding PEFT methods, while on consumption are compared with FMFT.

Models	Performance			Consumption		
	BLEU	M	R	Training Time	Memory Size	
FMFT	24.34	15.28	32.15	1h39m	22,005M	
LoRA	PEFT	21.65	14.14	29.17	1h20m	17,027M
	$Pasta_3$	23.11	14.94	30.89	1h25m	17,961M
		↑ 6.74%	↑ 5.66%	↑ 5.90%	↓ 14.14%	↓ 18.38%
	$Pasta_6$	24.09	15.50	32.14	1h31m	19,345M
		↑ 11.27%	↑ 9.62%	↑ 10.18%	↓ 8.08%	↓ 12.09%
	$Pasta_9$	24.47	15.73	32.57	1h38m	20,741M
		↑ 13.03%	↑ 11.24%	↑ 11.66%	↓ 1.01%	↓ 5.74%
AT	PEFT	23.95	15.25	31.60	1h23m	18,049M
	$Pasta_3$	23.79	15.46	31.90	1h26m	18,771M
		↓ 0.67%	↑ 1.38%	↑ 0.95%	↓ 13.13%	↓ 14.70%
	$Pasta_6$	24.19	15.54	32.10	1h31m	19,959M
		↑ 1.00%	↑ 1.90%	↑ 1.58%	↓ 8.08%	↓ 9.30%
	$Pasta_9$	24.49	15.69	32.41	1h35m	21,151M
		↑ 2.25%	↑ 2.89%	↑ 2.56%	↓ 4.04%	↓ 3.88%
PT	PEFT	11.69	7.89	16.22	1h18m	17,221M
	$Pasta_3$	23.22	15.01	31.02	1h27m	18,765M
		↑ 98.63%	↑ 90.24%	↑ 91.25%	↓ 12.12%	↓ 14.72%
	$Pasta_6$	24.37	15.73	32.49	1h31m	20,247M
		↑ 108.47%	↑ 99.37%	↑ 100.31%	↓ 8.08%	↓ 7.99%
	$Pasta_9$	23.95	15.44	31.94	1h36m	21,771M
		↑ 104.88%	↑ 95.69%	↑ 96.92%	↓ 3.03%	↓ 1.06%
PreT	PEFT	16.10	10.07	21.18	41m	5,867M
	$Pasta_3$	23.33	14.81	30.84	57m	10,293M
		↑ 44.91%	↑ 47.07%	↑ 45.61%	↓ 42.42%	↓ 53.22%
	$Pasta_6$	24.42	15.63	32.27	1h05m	13,869M
		↑ 51.68%	↑ 55.21%	↑ 52.36%	↓ 34.34%	↓ 36.97%
	$Pasta_9$	24.70	15.77	32.75	1h13m	17,461M
		↑ 53.41%	↑ 56.60%	↑ 54.63%	↓ 26.26%	↓ 20.65%

^Φ FMFT equals $Pasta_{12}$ while PEFT equals $Pasta_0$.

forgetting and preserves foundational code knowledge acquired during pre-training, ultimately yielding superior overall performance. However, this design comes with an implicit trade-off. It inevitably restricts the fine-grained plasticity of PLMs, thereby undermining their capacity to capture subtle or intricate semantics in code changes. As for the consumption of computational resources, although there are increases compared with the vanilla PT, the costs of Pasta_K are significantly lower than FMFT. The reduction can reach up to 12.12% and 14.72% in terms of the consumption of training time and memory size, showing the effectiveness and efficiency of Pasta_K .

For PreT, Figure 3.5(f) demonstrates its pros and cons more apparent, where FMFT significantly surpasses PreT in the top 9 layers, indicating that Pasta_K should switch the top K layers to be trainable while keeping others frozen, showing the opposite phenomenon of PT. Table 3.7 shows that Pasta_K can enhance the performance of PreT by 53.41%, 56.60%, and 54.63% at most in terms of BLEU, Meteor, and Rouge-L, respectively, even exceeding the performance of FMFT by 1.48%, 3.21%, and 1.87% in terms of each metric in order, when K is set to 9. With comparable or better results, the consumption of Pasta_K in training time and memory size is quite fewer than FMFT. For example, Pasta_3 can save 42.42% of training time and 53.22% of computational memory, showing better practical values.

For AT and LoRA, they show a similar trend with PT in Figure 3.5(e), performing worse in the bottom layers while better in the top layers. It enlightens Pasta_K to place FMFT layers at the bottom but PEFT layers on the top. From Table 3.7, we can observe that the maximum improvement is 2.25% and 13.03% in terms of BLEU for AT and LoRA, respectively. The best performance of Pasta_K also surpasses FMFT. Meanwhile, the consumption of Pasta_K is still consistently lower than that of FMFT, showing that Pasta_K can achieve better performance with higher efficiency.

Apart from the experiments on the CMG task, we also conduct extra experiments on the JIT-DP task to explore the performance of Pasta_K , where the evaluation results are shown in Table 3.8. We still use the setting with expert features and CodeBERT as the backbone model, following the guidance from probing results. Similar to the strategy used in the CMG task, Pasta_K first allocates PEFT and FMFT layers based on layer-wise probing results. As shown in Figure 3.5(b), AT, LoRA, and PreT perform either much better or comparable to FMFT, excelling on upper layers (layer#>4). However, PT performs neck-and-neck with FMFT on the CCM task. As for the LTP task, shown in Figure 3.5(c), PreT shows advantages against FMFT from layer#2 to #12, while neck-to-neck for the rest of the layers. LoRA generally outperforms FMFT from layer#4 to

Table 3.8. Evaluation results of $Pasta_K$ in the JIT-DP task.

Models		Performance		Consumption	
		F1	AUC	Training Time	Memory Size
	FMFT	0.419	0.867	6m02s	23,921M
LoRA	PEFT	0.512	0.899	4m46s	19,284M
	$Pasta_3$	0.415	0.885	5m05s	20,276M
	$Pasta_6$	0.448	0.880	5m23s	21,296M
	$Pasta_9$	0.462	0.882	5m41s	22,662M
AT	PEFT	0.502	0.894	5m05s	20,064M
	$Pasta_3$	0.415	0.890	5m23s	21,038M
	$Pasta_6$	0.386	0.885	5m35s	22,058M
	$Pasta_9$	0.398	0.887	6m00s	23,846M
PT	PEFT	0.493	0.880	4m29s	17,902M
	$Pasta_3$	0.432	0.877	4m47s	18,862M
	$Pasta_6$	0.421	0.866	5m05s	20,254M
	$Pasta_9$	0.422	0.874	5m24s	21,646M
PreT	PEFT	0.510	0.890	4m03s	17,054M
	$Pasta_3$	0.429	0.872	4m21s	17,824M
	$Pasta_6$	0.463	0.886	4m38s	18,510M
	$Pasta_9$	0.414	0.887	4m55s	20,022M

^Φ FMFT equals $Pasta_{12}$ while PEFT equals $Pasta_0$.

#12 and ties on the remaining layers. AT performs neck-to-neck against FMFT across all layers. PT underperforms FMFT in the bottom layers (layer#<7), while showing a comparable trend in the remaining layers. Consequently, results from both probing tasks direct us to first extend the bottom layers with FMFT, then gradually unfreeze the upper layers. Similarly, K is still set to 0, 3, 6, 9, and 12 for evaluation, where 0 and 12 represent $Pasta_0$ and $Pasta_{12}$ respectively.

As shown in Table 3.8, PEFT (i.e., $Pasta_0$) always performs the best, even outperforming FMFT (i.e., $Pasta_{12}$). Our probing results explain this phenomenon: for AT, LoRA, and PreT, FMFT does not exhibit superior capability for capturing dynamic code semantics at either the top or bottom layers. Consequently, applying PEFT to all layers is optimal, leading $Pasta_K$ to degenerate into pure PEFT, i.e., $Pasta_0$. This does not imply that $Pasta_K$ has failed. Instead, it means that $Pasta_K$ is explainable and strictly follows our proposed layer-wise tuning rules, thus it is still working on the JIT-DP task. It should be noted that $Pasta_K$ is not a method that necessarily combines

FMFT and PEFT layers. Instead, it adaptively selects between them based on their performance on probing tasks.

An exception is Pasta_K with PT. Theoretically, Pasta_K should have surpassed PT with the bottom K layers becoming FMFT. But the result is not the case. A potential explanation is that the probing results in CCM are somewhat contradictory to those in LTP when comparing PT and FMFT. For example, in CCM, FMFT’s main advantage is in layers#4–#6, while in LTP, its advantage is concentrated below layer#3. Owing to this reason, Pasta_K does not achieve the expected performance gain.

Finding 8: By analyzing the layer-wise probing results of CCM and LTP, our proposed Pasta_K can significantly improve the performance of PEFT methods in the CMG task, even surpassing FMFT. Meanwhile, the consumption of Pasta_K in training time and memory size is consistently lower than FMFT, showing the practical values of Pasta_K . In the JIT-DP task, the performance of Pasta_K also aligns well with the outcomes of our probing experiments overall, underscoring its rationality and interpretability.

3.4 Discussion

3.4.1 Implications

This chapter presents the first empirical study that investigates the performance of PEFT on code-change-related tasks and proposes a brand-new, effective, and efficient domain adaptation method, namely Pasta_K . In this section, we discuss the implications of this work from the perspectives of developers and researchers.

Implications for developers. This study indicates that PEFT can outperform FMFT on the JIT-DP task, as well as AT and LoRA can achieve comparable results on the CMG task with less training time and memory consumption. The results enlighten developers that can leverage PEFT to replace the original FMFT methods in such code-change-related classification tasks, and for generation tasks, when the computational resources are limited, PEFT can be an effective alternative. In addition, the superiority that PEFT shows in the cross-lingual and low-resource scenarios indicates that PEFT can be used in practice when meeting the data scarcity problem. Different from previous studies that show consistent improvements brought by PEFT in code search, code clone, and code summarization tasks [170, 199], our work guides when and how to

apply PEFT to code-change-related tasks. In the end, combining PEFT and FMFT, we propose Pasta_K, providing practitioners with a brand-new domain adaptation method with high efficacy and efficiency simultaneously for code change learning. With limited training resources, the new proposed method gives developers more flexibility to strike a balance between efficiency and performance, depending on the actual needs.

Implications for researchers. As the first empirical study to explore the performance of PEFT in code change learning, this work demonstrates that although there is an inconsistency between the pre-training tasks used by PLMs and the code-change-related tasks, PEFT methods can still perform better or comparably to FMFT in certain adaptation cases, such as cross-lingual and low-resource scenarios. Moreover, it can be seen that in the JIT-DP task, the performance of PEFT increases when adding the expert features, demonstrating that PLMs with PEFT methods are more skilled in learning straightforward semantics, owing to their limited parameter tuning. This finding deserves to be investigated in other tasks to pursue high performance. In addition, we design two code-change-related probing tasks for investigating the encoded dynamic code semantics from global and local perspectives. However, probing tasks for code-change learning from other angles are also necessary, such as syntax and lexical. Thus, we call for more effective probing tasks related to the code change. Furthermore, we propose a new approach named Pasta_K to explore domain adaptation techniques in a fine-grained manner. As the first attempt, this work combines FMFT and PEFT by their layer-wise probing results and testifies that a balance between efficiency and performance is achievable. Given that probing tasks are lightweight classification problems, the Pasta_K method remains highly feasible even as the backbone model scales up (e.g., to 24 layers or deeper) without introducing prohibitive computational overhead. To this end, this work can be seen as a footstone, and in the future, whether the combination can be fully automated or at a neuron level is still worth exploring.

3.4.2 Threats to Validity

In this section, we identify threats to the validity of this work from two aspects.

External validity relates to the generalizability of our findings. In this chapter, we conduct experiments on two widely studied code-change-related tasks, including JIT-DP and CMG. Although there are other code-change-related tasks, such as patch identification [90] and comment updating [231], JIT-DP, a classification task, and CMG, a generation task, are representative and cover two main categories. In the future, we

can include more code-change-related tasks for assessment. Moreover, due to computational resource constraints, although we did not exhaust all PLMs for experiments, we selected seven representative and widely studied code PLMs with diverse architectures and families. Thus, we believe the above threat is minimal.

Internal validity relates to the aspects that could impact the experimental results. A main threat to the internal validity lies in the hyperparameter settings, especially for those PEFT methods’ hyperparameters. In this study, we implement the standard architecture of AT and LoRA and follow their default hyperparameter settings. We conduct specific hyperparameter analysis to explore their impact across both tasks. For the PT and PreT, we also search for a good prompt length among many alternatives. As finding optimal hyperparameters is always a difficult task, it needs to be explored persistently.

3.5 Chapter Summary

Unlike static code comprehension, code-change-related tasks exhibit a pronounced discrepancy between pre-training objectives and fine-tuning requirements, leaving the relative efficacy of PEFT versus FMFT an open question. In this chapter, we systematically evaluate four prominent PEFT methods across two pivotal code-change tasks: JIT-DP and CMG. Our empirical investigation reveals that while PEFT methods consistently outperform FMFT on JIT-DP (with AT and LoRA achieving state-of-the-art results), they merely exhibit comparable performance on CMG despite lower resource consumption. Crucially, PEFT demonstrates relative superiority in both cross-lingual and low-resource scenarios. To make an explanation for the performance of PEFT and FMFT, we conduct three probing tasks spanning both static and dynamic perspectives. Leveraging the complementary layer-wise insights gleaned from probing results, we introduce Pasta_K , a novel framework that adaptively orchestrates FMFT and PEFT blocks to optimize domain adaptation. Extensive experiments confirm that Pasta_K substantially elevates PEFT baseline performance while consuming fewer computational resources than FMFT, thereby offering practical utility for real-world deployment.

Chapter 4

Unveiling the Efficacy and Efficiency of Mamba in Code Intelligence

While Transformer-based code language models have dominated code intelligence, their inherent quadratic attention complexity poses severe scalability bottlenecks. In this chapter, we explore Mamba, an emerging sub-quadratic architecture, to determine whether it can break through the limitations and offer a more scalable alternative. To bridge this gap, we present a systematic empirical evaluation across three representative tasks: code completion, code generation, and code clone detection. Specifically, we first pre-train two Mamba-based PLMs on a large-scale code corpus to establish rigorous baselines. Comprehensive evaluations under Full fine-Tuning (FT) and Parameter-Efficient Fine-Tuning (PEFT) configurations demonstrate the systematic superiority of Mamba over traditional Transformer baselines. Subsequently, from-scratch pre-training experiments are designed to decouple the intrinsic architectural capacities from corpus-specific influences, complemented by low-resource evaluations that deliberately restrict the available fine-tuning data. Both results firmly validate the dual advantages of Mamba in both efficacy and efficiency. Finally, scaling experiments up to 7B parameters reveal that code language models' performance is subject to the combined effects of model architecture, pre-training corpora, and specific task characteristics. This chapter provides critical insights into the strengths and boundaries of Mamba-based code intelligence.

The rest of this chapter is organized as follows. Section 4.1 presents the preliminary concepts of advanced architectures and tuning paradigms, followed by section 4.2 that

details our experimental setup and evaluation benchmarks. In section 4.3, we comprehensively analyze the experimental results, answering the five proposed research questions. Section 4.4 provides a deeper discussion through a case study, a human evaluation, and an analysis of threats to validity. Finally, section 4.5 encapsulates this chapter with a brief summary.

4.1 Preliminary

4.1.1 Transformer

The Transformer [195] 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.

4.1.2 State Space Models, Mamba, and Mamba-2

State Space Models (SSMs) [73] 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:

$$h'_t = Ah_t + Bx_t, \quad y_t = Ch_t \quad (4.1)$$

As this model describes the relationship between continuous quantities, it requires discretization before further processing. Additionally, an important property of SSMs

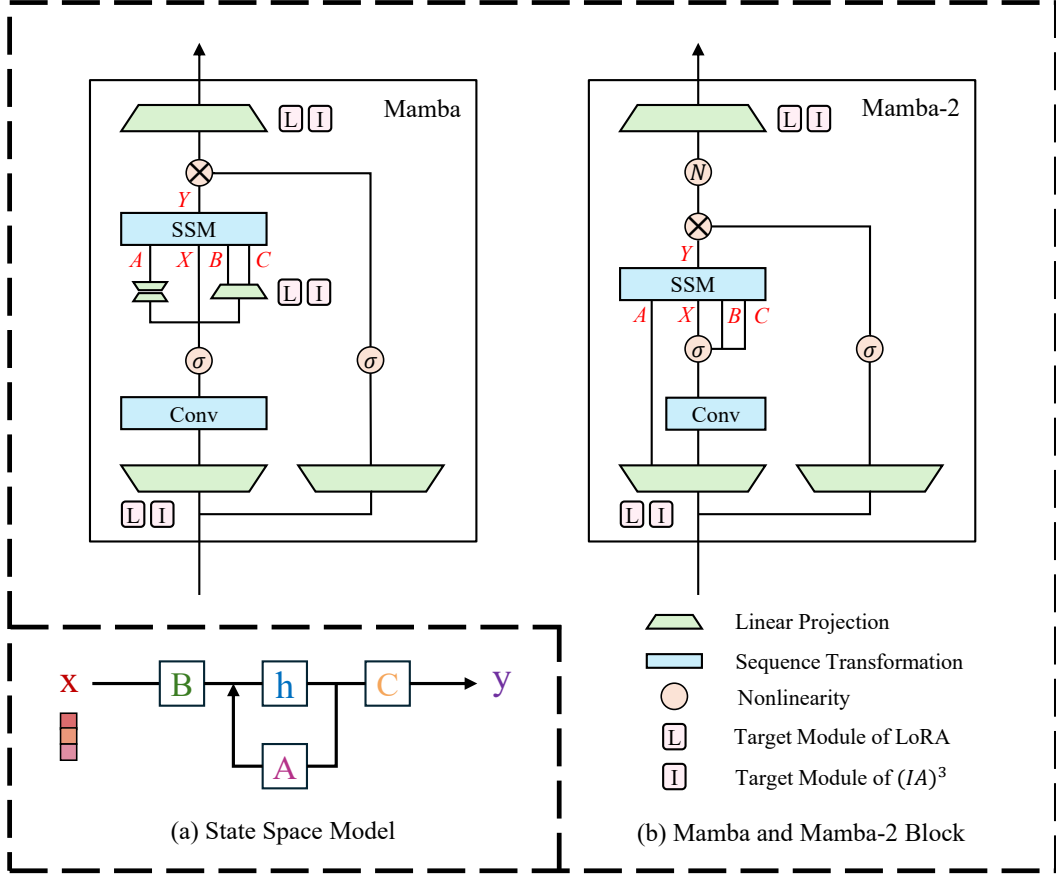


Figure 4.1. Illustrations of the State Space Model, Mamba, and Mamba-2.

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 [221]. To tackle this issue, Mamba [72] 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:

$$X = \sigma(\text{Conv1D}(\text{Linear}_{in}(I))), \quad Y = \text{SSM}(X) \quad (4.2)$$

$$O = \text{Linear}_{out}(\sigma(\text{Linear}_{gate}(I)) \odot Y) \quad (4.3)$$

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 $Linear_x$.

Inspired by the connection between SSMS and the Transformer, Mamba-2 [41] 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.

4.1.3 LoRA and $(IA)^3$

LoRA [95], 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 [233], we apply LoRA to the x , *in*, and *out* projection matrices in Mamba, along with the *in* and *out* projection matrices in Mamba-2.

$(IA)^3$ [132] 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.

4.2 Experimental Setup

4.2.1 Research Questions

This chapter investigates the following Research Questions (RQs), intending to explore the performance of Mamba in code intelligence and make comparisons with the Transformer counterparts. Figure 4.2 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 [142] 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 [98], 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 [133, 136]. 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 [233], 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, learning objectives, and hyperparameter configurations are not identical. Hence, we pre-train CodeGPT, Mamba, and Mamba-2 from scratch with autoregressive language modeling, thereby obtaining CodeGPT_{scratch},

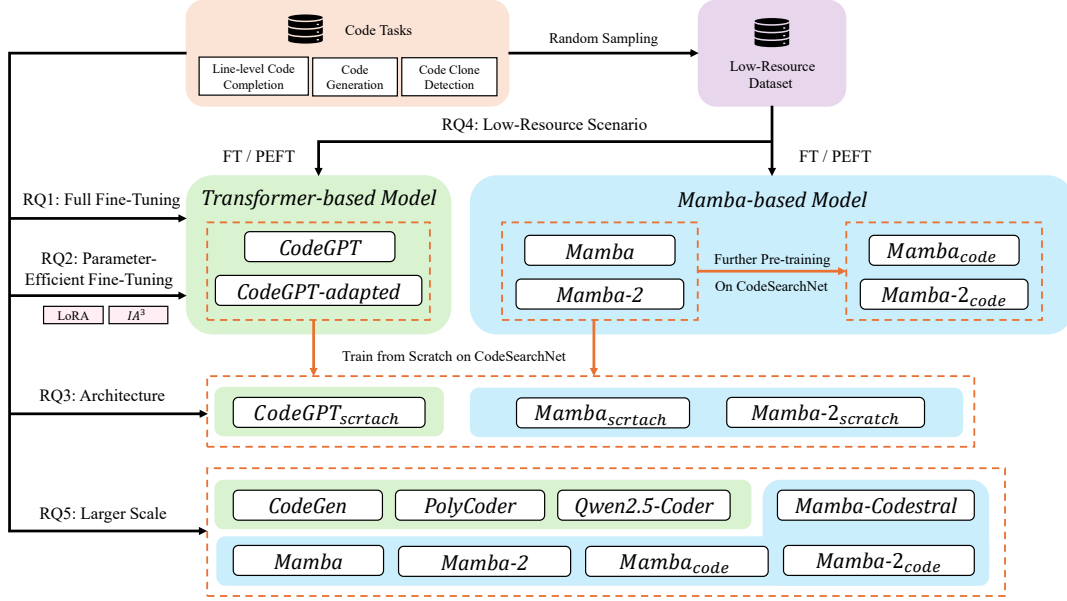


Figure 4.2. The overall workflow. There are a total of five research questions.

Mamba_{scratch}, and Mamba-2_{scratch}. Subsequently, we analyze their performance 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 [136, 197]. 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 [86], 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 [37], a Mamba-based Large Language Model (LLM) with 7B parameters. To make a fair comparison between Transformer-based counterparts, CodeGen [155], PolyCoder [215], and Qwen2.5-Coder [97] 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 [121, 223, 232] 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 [137] for specific examples. All three code tasks are evaluated in this RQ.

4.2.2 Tasks and Datasets

We evaluate the effectiveness of Mamba on three widely studied typical code tasks covering both code comprehension and generation categories [55, 75, 157, 207, 238]: (1) code completion task, (2) code generation task, and (3) code clone detection task, thereby ensuring a comprehensive assessment.

Table 4.1. 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

4.2.2.1 Line-level Code Completion (LCC)

Code completion has become an indispensable feature in modern Integrated Development Environments (IDEs) [131]. It contributes to higher developer productivity by providing context-driven code suggestions [77]. As in prior studies [141, 142], 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 [8] and PY150 [165]. Their statistical information is presented in Table 4.1, 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.

4.2.2.2 Code Generation (CG)

Code generation is a particularly intriguing task, involving producing source code from natural language descriptions [106]. It holds significant promise for reducing development costs and reshaping programming paradigms, thereby attracting substantial attention from both industry and academia [46]. To carry out this task, we choose a widely used dataset, CONCODE [102]. 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 4.1.

4.2.2.3 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 [220]. 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 [181], for evaluation. Its detailed statistics are shown in Table 4.1. In this chapter, 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.

4.2.3 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** [142] is a code-oriented extension of GPT-2 [164], 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 types: CodeGPT and CodeGPT-adapted. The former is pre-trained from scratch, while the latter uses the GPT-2 as the starting point. They both have 125M parameters.

- **Mamba** [72] and **Mamba-2** [41] are designed for autoregressive language modeling, pre-trained on the Pile dataset [60]. Their scales are from 130M to 1.4B, and we use their 130M-sized and 370M-sized models in this chapter. In addition to their original models, our experiments also pre-trained Mamba_{code} , Mamba-2_{code} , $\text{Mamba}_{scratch}$, and $\text{Mamba-2}_{scratch}$ for evaluation as mentioned in RQ1 and RQ3 of Section 4.2.1.

- **CodeGen** [155] is a family of code PLMs designed to enhance its capacities through conversational program synthesis. In this chapter, we use the CodeGen-Multi model for Java and the CodeGen-Mono model for Python. Both 350M and 6B versions are included.

- **PolyCoder** [215] 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** [97] comprises a range of open-source models, having achieved significant advancements when compared with leading code LMs. In this chapter, we adopt its 500M-sized and 7B-sized models.

- **Codestral Mamba** [37] is a large-scale, Mamba-based code LLM released by Mistral AI [2]. Built upon the Mamba-2 architecture, it contains 7 billion parameters.

4.2.4 Evaluation Metrics

Following previous studies [64, 79, 228, 251], we adopt the Exact Match (EM) and Edit Similarity (ES) as the evaluation metrics for line-level code completion, while using EM, BLEU [158], and CodeBLEU [168] 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 [47, 220]. It is the harmonic mean of precision and recall.

4.2.5 Implementation Details

The code PLMs we used are from Huggingface [150], and we keep their default hyperparameter settings. In the line-level code completion task, following the prior work [142], 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 [129, 145], 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.

4.3 Experimental Results

4.3.1 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 4.2. 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) [213] with a confidence level of 95% between Mamba-based PLMs and two CodeGPT variants, respectively. As can be seen from Table 4.2, 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 4.2, 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 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.

Table 4.2. Evaluation results in the FT and PEFT settings. The Mamba and Mamba-2 used here are 130M-sized.

Method	Models	JavaCorpus				PY150				CONCODE				BigCloneBench	
		EM	ES	EM	ES	EM	ES	EM	BLEU	CodeBLEU	F1				
FT	CodeGPT	27.43 $\pm$ 0.21	63.79 $\pm$ 0.30	37.81 $\pm$ 0.27	70.26 $\pm$ 0.43	16.25 $\pm$ 0.54	31.50 $\pm$ 0.32	32.56 $\pm$ 0.32	0.927 $\pm$ 0.005						
	CodeGPT-adapted	30.01 $\pm$ 0.10	65.73 $\pm$ 0.24	37.77 $\pm$ 0.18	70.39 $\pm$ 0.25	17.72 $\pm$ 0.48	32.42 $\pm$ 0.84	33.26 $\pm$ 0.92	0.925 $\pm$ 0.002						
	Mamba	28.31 $\pm$ 0.38	64.91 $\pm$ 0.12	36.80 $\pm$ 0.15	70.08 $\pm$ 0.10	18.65 $\pm$ 0.33	33.11 $\pm$ 0.52	34.11 $\pm$ 0.72	0.919 $\pm$ 0.007						
	Mamba-2	29.16 $\pm$ 0.10	65.68 $\pm$ 0.32	37.93 $\pm$ 0.17	70.45 $\pm$ 0.17	18.47 $\pm$ 0.23	32.58 $\pm$ 0.51	33.34 $\pm$ 0.68	0.944 $\pm$ 0.003						
	Mamba _{code}	29.39 $\pm$ 0.26	65.91 $\pm$ 0.10	37.25 $\pm$ 0.14	70.53 $\pm$ 0.11	19.02 $\pm$ 0.16	33.08 $\pm$ 0.08	33.50 $\pm$ 0.24	0.943 $\pm$ 0.001						
LoRA	Mamba-2 _{code}	30.75 $\pm$ 0.11	66.52 $\pm$ 0.26	37.68 $\pm$ 0.09	70.44 $\pm$ 0.03	18.53 $\pm$ 0.21	33.51 $\pm$ 0.33	34.22 $\pm$ 0.45	0.945 $\pm$ 0.003						
	CodeGPT	25.42 $\pm$ 0.31	61.88 $\pm$ 0.47	31.99 $\pm$ 0.30	66.22 $\pm$ 0.27	15.65 $\pm$ 0.38	27.05 $\pm$ 0.30	27.96 $\pm$ 0.32	0.901 $\pm$ 0.003						
	CodeGPT-adapted	28.71 $\pm$ 0.21	64.61 $\pm$ 0.18	33.53 $\pm$ 0.19	67.19 $\pm$ 0.10	17.78 $\pm$ 0.30	27.95 $\pm$ 0.61	28.80 $\pm$ 0.58	0.893 $\pm$ 0.006						
	Mamba	27.72 $\pm$ 0.35	64.77 $\pm$ 0.19	34.56 $\pm$ 0.17	68.47 $\pm$ 0.08	18.12 $\pm$ 0.88	30.98 $\pm$ 0.45	31.62 $\pm$ 0.35	0.924 $\pm$ 0.003						
	Mamba-2	28.59 $\pm$ 0.11	65.38 $\pm$ 0.36	35.11 $\pm$ 0.25	68.79 $\pm$ 0.06	17.83 $\pm$ 0.21	31.23 $\pm$ 0.25	31.91 $\pm$ 0.26	0.932 $\pm$ 0.004						
(IA) ³	Mamba _{code}	29.23 $\pm$ 0.24	66.03 $\pm$ 0.17	35.23 $\pm$ 0.10	68.98 $\pm$ 0.10	18.15 $\pm$ 0.09	31.25 $\pm$ 0.20	31.93 $\pm$ 0.11	0.933 $\pm$ 0.001						
	Mamba-2 _{code}	30.00 $\pm$ 0.12	66.48 $\pm$ 0.16	35.87 $\pm$ 0.07	69.24 $\pm$ 0.07	17.67 $\pm$ 0.30	31.73 $\pm$ 0.44	32.38 $\pm$ 0.56	0.935 $\pm$ 0.003						
	CodeGPT	21.62 $\pm$ 0.27	58.03 $\pm$ 0.32	23.42 $\pm$ 0.30	59.21 $\pm$ 0.14	10.57 $\pm$ 0.37	21.40 $\pm$ 0.81	22.99 $\pm$ 0.97	0.728 $\pm$ 0.007						
	CodeGPT-adapted	25.52 $\pm$ 0.27	62.07 $\pm$ 0.25	26.41 $\pm$ 0.15	62.26 $\pm$ 0.27	13.90 $\pm$ 0.50	23.33 $\pm$ 0.53	24.47 $\pm$ 0.58	0.724 $\pm$ 0.060						
	Mamba	25.07 $\pm$ 0.10	62.99 $\pm$ 0.05	25.88 $\pm$ 0.17	61.84 $\pm$ 0.16	15.03 $\pm$ 0.31	23.61 $\pm$ 0.12	24.69 $\pm$ 0.13	0.768 $\pm$ 0.020						
(IA) ³	Mamba-2	25.67 $\pm$ 0.04	63.30 $\pm$ 0.16	26.44 $\pm$ 0.20	62.32 $\pm$ 0.16	15.33 $\pm$ 0.23	23.79 $\pm$ 0.67	24.86 $\pm$ 0.75	0.744 $\pm$ 0.017						
	Mamba _{code}	26.56 $\pm$ 0.14	64.25 $\pm$ 0.13	28.15 $\pm$ 0.07	64.00 $\pm$ 0.13	15.63 $\pm$ 0.10	23.97 $\pm$ 0.57	25.08 $\pm$ 0.55	0.798 $\pm$ 0.005						
	Mamba-2 _{code}	27.36 $\pm$ 0.22	64.67 $\pm$ 0.09	28.73 $\pm$ 0.09	64.20 $\pm$ 0.13	15.65 $\pm$ 0.30	24.52 $\pm$ 0.68	25.60 $\pm$ 0.62	0.795 $\pm$ 0.005						

* 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.

4.3.2 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 4.2. 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 an average of 1.86% of parameters for CodeGPT variants and 7.83% for Mamba-based counterparts. In contrast, the adaptation of $(IA)^3$ is more lightweight, tuning only approximately 0.05% of parameters for CodeGPT variants and around 0.09% for Mamba-based PLMs. Regarding the performance comparison of PLMs tuned via $(IA)^3$, 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 [81].

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.

4.3.3 RQ3: Advantages of Mamba Architecture

Although Mamba-based PLMs show superior performance compared to Transformer-based ones in RQ1 and RQ2, the 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 4.3, 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 [96]. 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.

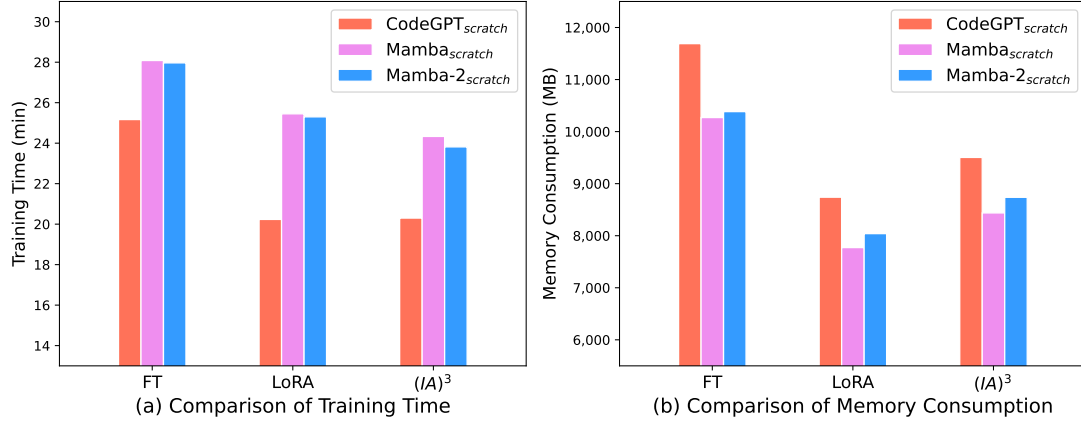


Figure 4.3. Utilization of computational resources.

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 4.3. 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 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.

Table 4.3. 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}.

Table 4.4. Evaluation results of larger-scale models using FT during adaptation.

Methods	Models	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	
	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	
Zero Shot	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.

4.3.4 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 4.4, 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.

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.

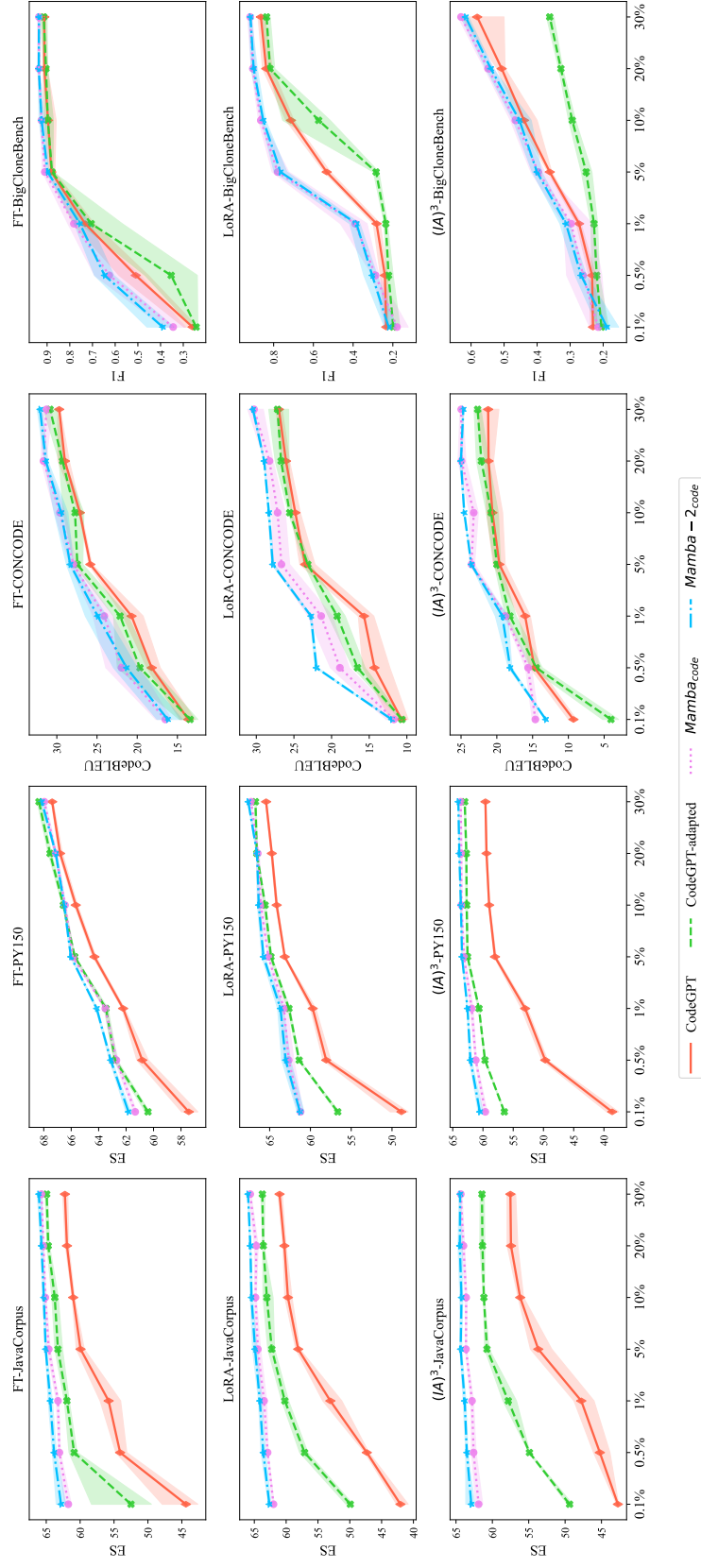


Figure 4.4. Evaluation results in the low-resource scenario.

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.

4.3.5 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 4.4. 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 ??, 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 4.4, Codestral Mamba lags behind Transformer-based models a lot, by 36.06%–45.93% in terms of EM and by 13.03%–19.12% 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 [2] 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 [37], 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.

4.4 Discussion

4.4.1 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 4.5, 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 extra “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.

<p>The NL input:</p> <pre>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</pre> <p>The output of CodeGPT:</p> <pre>void function(ClassdiagramNode arg0, ClassdiagramNode arg1, int arg2, float arg3) { uplinks.add(new ClassdiagramNode(arg0, arg1, arg2, arg3)); }</pre> <p>The output of CodeGPT-adapted:</p> <pre>void function(float arg0) { if (!isStandalone()) { uplinks.add(arg0); } }</pre>	<p>The Ground Truth:</p> <pre>void function(ClassdiagramNode arg0) { uplinks.add(arg0); }</pre> <p>The output of Mamba_{code}:</p> <pre>void function(ClassdiagramNode arg0) { uplinks.add(arg0); }</pre>	<p>The output of Mamba-2_{code}:</p> <pre>void function(ClassdiagramNode arg0) { uplinks.add(arg0); }</pre>
<p>The NL input:</p> <pre>adds a parent node to the node. concode_field_sep List<DirectedAcyclicGraphNode<T>> mParents concode_elem_sep T mPayload</pre> <p>The output of CodeGPT:</p> <pre>void function(DirectedAcyclicGraphNode<T> arg0) { mParents.add(arg0); }</pre> <p>The output of Mamba_{code}:</p> <pre>void function(T arg0) { mParents.add(arg0); }</pre>	<p>The Ground Truth:</p> <pre>void function(DirectedAcyclicGraphNode<T> arg0) { mParents.add(arg0); }</pre> <p>The output of CodeGPT-adapted:</p> <pre>void function(DirectedAcyclicGraphNode<T> arg0) { mParents.add(arg0); }</pre> <p>The output of Mamba-2_{code}:</p> <pre>void function(DirectedAcyclicGraphNode<T> arg0) { mParents.add(arg0); arg0.mParentId = mPayload; }</pre>	

Figure 4.5. Case study in the CONCODE dataset of the code generation task.

4.4.2 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 4.5, 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 that it consistently exceeds 0.6, indicating substantial agreement between evaluators and demonstrating the practical value of Mamba-based PLMs.

Table 4.5. 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}”.

4.4.3 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 [55, 75, 97, 207]. Although we also noticed other kinds of neural networks, such as Spiking Neural Network [65] and Generative Adversarial Networks [70], 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 [233]. 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.

4.5 Chapter Summary

This chapter presents a systematic empirical study of Mamba-based models across three representative code intelligence tasks, covering both code comprehension and generation. Our experimental results demonstrate that Mamba-based PLMs perform better than or comparably to similarly sized Transformer-based alternatives under full fine-tuning. Notably, in the PEFT setting, Mamba-based models exhibit even more substantial advantages. We further provide empirical evidence of the Mamba architecture’s inherent suitability for source code and its superior learning ability in low-resource scenarios. Additionally, we scale our investigation to 370M and 7B parameter ranges for both Mamba and Transformer-based models. Finally, we supplement our quantitative analysis with a qualitative case study, a human evaluation, and a discussion of broader implications, shedding light on the strengths, limitations, and future prospects of Mamba in the software engineering domain.

Chapter 5

Exploring Continual Learning in Code Intelligence with Domain-wise Distilled Prompts

In practical software engineering, continuous program evolution inevitably triggers severe domain shifts that challenge traditional offline training paradigms. In the context of the evolving environment, this chapter systematically investigates Continual Learning (CL) strategies tailored for code intelligence to enhance adaptability. While prevailing approaches attempt to mitigate domain shifts through historical data rehearsal, they inherently introduce prohibitive training overheads and severe risks of data privacy leakage. To circumvent these limitations, we leverage the superiority of prompts in eliciting pre-trained knowledge to realize a rehearsal-free method. Through a preliminary investigation into vanilla prompt tuning, we establish a crucial foundation that inheriting PLM weights underpins sequential adaptation, and maintaining prompt stability should be emphasized. Driven by these insights, we introduce Prompt Tuning with Domain-wise Distillation (PTDD), a framework that can distill prompts and optimize PLMs with a two-sided learning objective. Extensive experiments spanning three pivotal code intelligence tasks demonstrate that PTDD consistently eclipses competitive baselines, outperforming fine-tuning by 2.55%, 11.12%, and 2.25%, respectively. Moreover, we interpret PTDD’s effectiveness via prompt visualization and analyze its robustness in the low-resource scenario, where the improvement of PTDD becomes stark with fewer training samples, reaching up to 69.09%. Overall, this chapter validates the efficacy of PTDD and furnishes deeper insights into the capability of prompts.

The subsequent sections of this chapter proceed as follows. Section 5.1 presents the preliminary concepts, including continual learning protocols and their combination with fine-tuning and prompt tuning. Section 5.2 introduces the detailed methodology of our proposed PTDD framework. Section 5.3 establishes the experimental setup, and in section 5.4, we comprehensively analyze the experimental results to address the five proposed research questions. Section 5.5 provides a deeper discussion, incorporating a visualization analysis for PTDD, a qualitative case study, training cost analysis, implications, and threats to validity. Finally, section 5.6 encapsulates this chapter with a brief summary.

5.1 Preliminary

5.1.1 Continual Learning Protocols

Generally, in the CL scenario, the language model is presented with a sequence of non-overlapping datasets corresponding to various distinctive domain knowledge, and the goal is to incrementally learn new domain knowledge while not severely forgetting previously learned ones [177, 237]. Considering in the code intelligence field, we denote the sequence of datasets as $D = \{D_1, D_2, \dots, D_m\}$, where m is the total number of datasets. For the t -th dataset D_t , it contains data samples $(x_i^t, y_i^t)_{i=1}^{N_t}$, where x_i^t is an input code snippet or natural language description, and y_i^t is a corresponding label in classification tasks or an output text in generation tasks. N_t represents the dataset size of D_t . We assume that data samples in the same dataset are independent and identically distributed, but they have varied distributions in different datasets. The learning objective is to learn a probabilistic model $p(D|\Theta) = \prod_{t=1}^m p(D_t|\Theta)$ and maximize its performance on D . Θ is the model's parameters. Across all datasets, the learning objective can be defined as:

$$\Theta = \arg \max_{\Theta} \sum_{t=1}^m \sum_{x, y \in D_t} \log p_{\Theta}(y|x) \quad (5.1)$$

In this chapter, we deal with the domain-incremental continual learning setting, which learns to solve the same task in different datasets but does not know which dataset a data sample belongs to at test time [25, 193]. Specifically, D_t can be divided into a training set, a validation set, and a test set, denoted as $D_t = \{T_t, V_t, S_t\}$. When it is the

turn of D_t , the model will be trained and validated on T_t and V_t . Consequently, model Θ_t is obtained. It is worth noting that some existing works [62, 83, 237] allow the model to access training samples from previous datasets, but in this chapter, in order not to violate data privacy, only T_t is accessible. At test time, Θ_t will be evaluated on all learned datasets, namely $\{S_1, S_2, \dots, S_t\}$.

5.1.2 Continual Learning with Fine-Tuning and Prompt Tuning

Fine-tuning has become the predominant technique for adapting PLMs to specific downstream tasks [43, 247]. It initializes models with the parameters from PLMs, and all parameters Θ are updated to learn task-specific knowledge. In many code intelligence tasks, it has shown promising results [133, 136, 175]. Formally, given a downstream dataset D' with input text x_i and corresponding output y_i , FT updates all parameters Θ in a supervised manner as:

$$\Theta = \arg \max_{\Theta} \sum_{x_i, y_i \in D'} \log p_{\Theta}(y_i | x_i) \quad (5.2)$$

Prompt tuning [120] is an emerging technique and can be regarded as an alternative to fine-tuning. In contrast to FT, it introduces a series of virtual tokens, or soft prompts, to modify the input text. This makes PLMs obtain additional task-specific instructions, so that more pre-trained knowledge can be elicited [134]. In addition, PT can mitigate the gap in learning objectives between the pre-training and the fine-tuning process [198]. It achieves a similar form of Masked Language Modeling, which is a common pre-training objective. The above-mentioned advantages make PT popular in both NLP [143, 202] and code intelligence [123, 197]. Consider a dataset $D' = \{(x_i, y_i)\}_{i=1}^{N'}$, and denote the soft prompt as P . In prompt tuning, its parameters not only include the PLM parameters Θ , but also contain the extra prompt parameters Θ_P . The learning objective is to maximize the log-likelihood of y_i given the input x_i concatenated with the soft prompt P , which can be formulated as:

$$[\Theta, \Theta_P] = \arg \max_{\Theta, \Theta_P} \sum_{x_i, y_i \in D'} \log p_{\Theta, \Theta_P}(y_i | [P; x_i]) \quad (5.3)$$

Moreover, to pursue low computational overhead and high efficiency, there is also an approach called parameter-efficient prompt tuning (PEPT) [29, 122]. It tunes only

the extra prompt parameters Θ_P on downstream tasks, but keeps the PLM parameters Θ frozen to reduce the computational cost and memory requirements. Although the final performance may be impacted due to the small number of tuned parameters, PEPT is practical in real-world scenarios, especially when training data is scarce [198].

5.2 Methodology

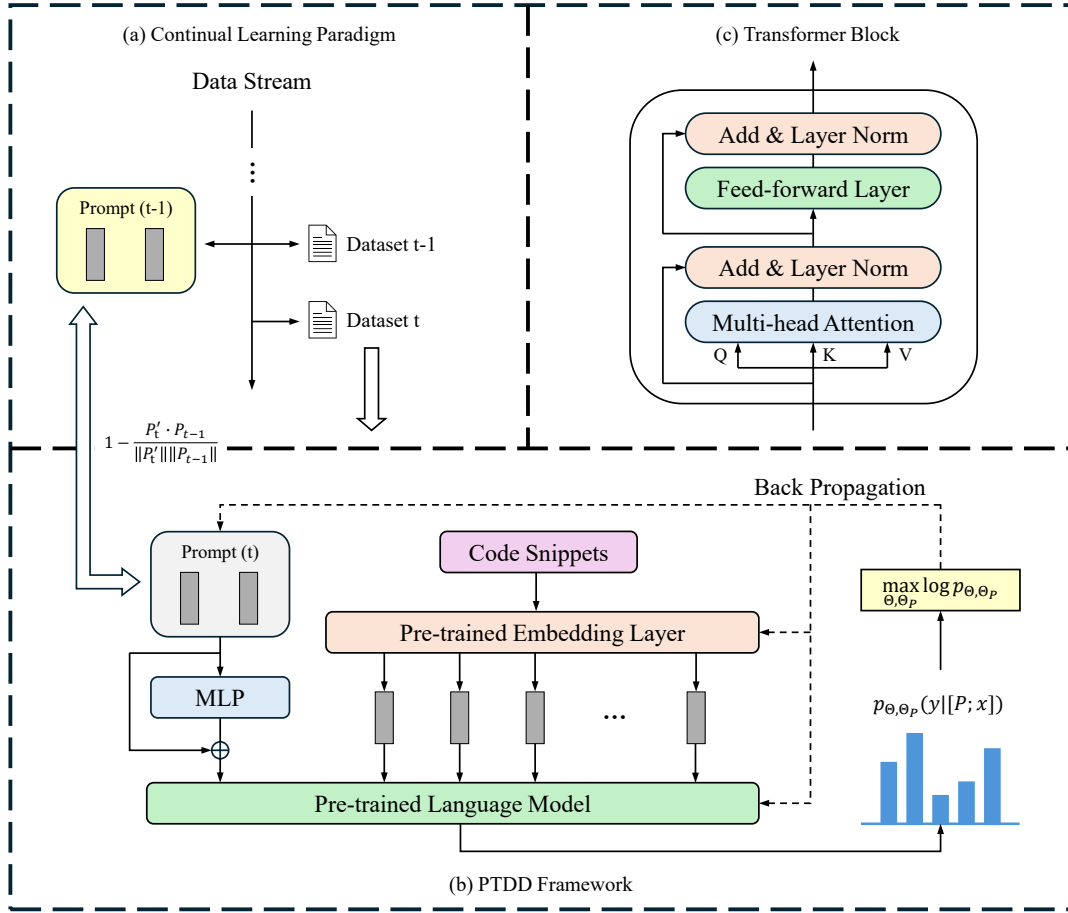


Figure 5.1. Overview of the proposed Prompt Tuning with Domain-wise Distillation (PTDD).

Figure 5.1 shows the overview of the proposed PTDD, which is well-designed to balance the plasticity and the stability learning objective. Receiving a sequence of datasets one by one, PTDD constantly updates the model and the prompt to distill previous knowledge, which can perform well on the current dataset while not forgetting much previously learned knowledge. As there is no overlap between datasets, here one

dataset represents one domain. For each dataset, PTDD first initializes the prompt and embeds it into the prompt space. Then, PTDD optimizes the model and the prompt to distill learned knowledge and integrate new knowledge.

5.2.1 Prompt Embedding

In the training process, consider that PTDD is dealing with the t -th dataset D_t , whose prompt is denoted as P_t . Different from previous works [209, 225] that randomly initialize a prompt pool and calculate the relationship between prompts and datasets, PTDD adopts a straightforward strategy that assigns one prompt for the current dataset and another prompt for all previous datasets. The two prompts alternate when fitting on new domains. Considering that the development experience gained previously significantly contributes to future development in practice, knowledge sharing across domains is vitally important [11]. To accomplish this, the prompt P_t is initialized as its previous prompt P_{t-1} to explicitly inherit previously learned knowledge, but P_0 is randomly initialized from the vocabulary. Formally, $P_t \in \mathbb{R}^{L_P \times D}$, where L_P is the prompt’s token length and D is the embedding dimension.

In addition, as some existing works [139, 166] find that directly passing the prompt embedding to language models may cause training instability and hinder model performance, PTDD inputs the embedding P_t into a Multi-Layer Perceptron (MLP) and utilizes a residual connection combining P_t and the MLP’s output. The residual connection improves language models’ representation ability and is beneficial to optimization. As a result, a new prompt embedding P'_t is obtained, which has the same size as P_t and can be formulated as:

$$P'_t = MLP(P_t) + P_t \quad (5.4)$$

5.2.2 Distilled Prompt Learning

To distill prompts containing previous knowledge, it is important to balance plasticity and stability. Plasticity refers to the adaptability to new domains, while stability prevents knowledge of past domains from forgetting. When training on dataset D_t , plasticity can be simply accommodated by optimizing the model and prompt on the current dataset. Specifically, denote the input code snippet as x_t , its target output as y_t , and $X = [P_t; x_t]$ is the language model’s input embedding. As shown in Figure 5.1(c),

input embedding X is first transformed to Q, K, V by multiplying learnable projection matrices W_Q, W_K, W_V , and then calculated in different parts, including multi-head attention, feed-forward layer, and layer normalization. The encoder output is further sent to a decoder or a classifier according to the task category to generate the final output. The optimizing objective can be defined to minimize the following loss function:

$$\mathcal{L}_1 = \mathcal{L}(f_{PLM}([P'_t; x_t]), y_t) \quad (5.5)$$

Where $\mathcal{L}$ is a loss function, which is the language modeling loss for generation tasks and the cross-entropy loss for classification tasks. Besides, $f_{PLM} : \mathcal{X} \rightarrow \mathcal{Y}$ mainly represents the transformation caused by the PLM.

Algorithm 1 Prompt Tuning with Domain-wise Distillation

Require: Dataset $D_{1:m}$, pre-trained language model Θ_0 , prompt length L_P , factor λ .

Ensure: Generated text or predicted label.

- 1: Initialize $\Theta \leftarrow \Theta_0$.
 - 2: Randomly select L_P tokens from the vocabulary to initialize P_1 .
 - 3: $P'_1 \leftarrow MLP(P_1) + P_1$.
 - 4: Minimize (5) to update P'_1 and Θ .
 - 5: **for** $t = 2, \dots, m$ **do**
 - 6: Embed $P_t \leftarrow P'_{t-1}$.
 - 7: $P'_t \leftarrow MLP(P_t) + P_t$.
 - 8: Calculate cosine similarity between P'_t and P'_{t-1} .
 - 9: Minimize (7) to update P'_t and Θ .
 - 10: **end for**
 - 11: Concatenate P'_m with test data to calculate outputs.
 - 12: **return** Generated text or predicted label.
-

As the findings in Section 5.4.1, besides plasticity, PTDD also introduces stability into the learning objective. It employs cosine similarity to measure the proximity between the currently updated prompt P'_t and the last prompt P'_{t-1} , and expects these two to be similar. In this way, P'_t can retain alike semantics to previous knowledge, thus stabilizing the prompt's representation. To achieve the learning objective, an additional loss function is defined as:

$$\mathcal{L}_2(P'_t, P'_{t-1}) = 1 - \frac{P'_t \cdot P'_{t-1}}{\|P'_t\| \|P'_{t-1}\|} \quad (5.6)$$

Finally, to combine the two objectives together, PTDD uses a factor λ as the regularization coefficient to balance their relationship. Thus, the overall loss function is:

$$\mathcal{L} = \mathcal{L}_1 + \lambda \cdot \mathcal{L}_2(P'_t, P_{t-1}) \quad (5.7)$$

Through backpropagation, PTDD updates the model and the prompt while preserving previous knowledge. Without accessing any past data, PTDD is fully rehearsal-free. It is worth noting that we realize stability from the perspective of prompt, but not like some previous works from the perspective of PLM [116]. As the two perspectives are orthogonal, they can be integrated to further investigate, but beyond the scope of this chapter. During training, if freezing the PLM parameters Θ and keeping only the prompt parameters Θ_P trainable, the training stage becomes efficient, and we name it as parameter-efficient PTDD (PE-PTDD). Moreover, in the inference stage, as the dataset from which test data comes is unknowable, we directly concatenate the finally learned prompt with input code snippets, and discard the MLP module so as not to interfere with the prompt representation. The overall approach of PTDD is manifested in the Algorithm 1.

5.3 Experimental Setup

5.3.1 Tasks and Datasets

To evaluate the effectiveness of our proposed PTDD, we adopt three downstream code intelligence tasks, namely code summarization, code vulnerability detection, and code clone detection. We utilize the project-level continual learning datasets open-sourced by Gao et al. [62]. In this section, we will describe the three downstream tasks and their continual learning datasets in detail.

5.3.1.1 Code Summarization (CS)

A high-quality summarization is essential to program comprehension and maintenance [173]. When given a code snippet, code summarization aims to automatically generate a natural language description containing the code’s functionality and purpose. Recent state-of-the-art approaches commonly resort to sequence-to-sequence deep learning models to achieve promising performance [3, 61]. CodeSearchNet [98] is a widely-used dataset for code summarization, which includes open-source code spanning six

programming languages. To simulate the continual learning scenario, on each programming language, Gao et al. randomly split the original dataset into five parts according to the project name that code snippets derive from. Their division makes code snippets in different parts belonging to different projects, ensuring project-level continual learning. Datasets for each part are divided into training set, validation set, and test set by the ratio of 8:1:1.

5.3.1.2 Code Vulnerability Detection (CVD)

Vulnerability detection aims to identify defective code in a timely manner and is crucial to software security [26]. Recent deep learning-based methods mostly represent it as a binary classification task, to predict whether the input code snippet contains vulnerabilities [242]. Fan et al. [49] curate a large C/C++ code vulnerability dataset, namely Big-Vul. In the same procedure as aforementioned, Gao et al. divide it into five parts and split the training set, validation set, and test set, constructing its project-level continual learning version, called Big-Vul-PC.

5.3.1.3 Code Clone Detection (CCD)

Code clone indicates two code snippets have the same functional semantics but with syntactic variations. It makes code error-prone and introduces risks on software maintenance and evolution [118]. Code clone detection aims to determine the functional equivalence of code snippet pairs through a classification paradigm. POJ [151] is a large-scale dataset for code clone detection containing 104 different algorithms, and for each algorithm, there are 500 programs written in the C language. Due to the enormous amount of data when pairing code snippets, following the previously used strategy [235], after dividing the 104 algorithms into five parts, Gao et al. randomly sample 40,000/5,000/5,000 pairs in each part to construct the training set, validation set, and test set. The continual learning dataset is represented as POJ-PC.

5.3.2 Baselines

To show the effectiveness of our proposed PTDD, we compare our methods with the following representative baselines. Their detailed information is described below. Specifically, in the comparison, we choose CodeT5 [207] as the backbone model for code

summarization and CodeBERT [55] for code vulnerability detection and code clone detection. The reason for choosing the two models is based on their popularity and representativeness in the three tasks. As pre-trained on large-scale code corpora, CodeT5 and CodeBERT learn universal representations for programs and have achieved outstanding results. They are widely used baselines in the research on the three tasks, and become fundamental models to achieve SOTA results [179, 183, 242]. In the meantime, CodeBERT is an encoder-only model and is good at classification tasks like code vulnerability detection and code clone detection, and CodeT5 builds on an encoder-decoder framework so that it excels at generation tasks like code summarization.

- **Fine-Tuning** [43]: FT directly updates all parameters of language models to downstream tasks. In the continual learning scenario, the tuned model in the previous dataset is used as the model’s initialization in the next dataset.
- **Prompt Tuning** [120]: For continual learning, PT and PEPT both add extra prompts, but the former tunes all parameters and the latter only updates prompts. Following previous works [209, 225], the initialization of prompts is randomly generated.
- **EWC** [116]: As the abbreviation of elastic weight consolidation, EWC slows down learning on certain weights to mitigate catastrophic forgetting. The selected parameters are important for previous tasks and are constrained to stay close to their old values.
- **EMR** [200]: EMR is short for episodic memory replay, a representative rehearsal-based method. It randomly retrains data samples from the previous datasets to ensure the learned knowledge will not be forgotten.
- **PP-TF** [225]: PP-TF, namely prompt pooling with teacher forcing, is a parameter-efficient continual learning approach for generation tasks. It enforces constraints on the prompt selection to make training stable.

It is worth noting that we choose EMR as the representative of rehearsal-based methods, but not REPEAT [62]. This is because, considering the advantages that rehearsal-based methods have, namely reutilizing previous data samples, there is no need to compare PTDD with the SOTA rehearsal-based method, i.e., REPEAT. As REPEAT is based on EMR and adds two extra components, including representative exemplars replay and adaptive parameter regularization, we choose the more fundamental baseline EMR in this chapter to demonstrate the effect of the rehearsal-based methods.

5.3.3 Evaluation Metrics

For code summarization, we use three metrics for evaluation, i.e., BLEU [158], Meteor [15], and Rouge-L [128]. BLEU compares n-grams of the generated text with n-grams of the ground truth text, counting the number of matches. We set n as 4 in this chapter (i.e., BLEU-4). Meteor focuses on the unigrams and takes into account the precision, recall, and fluency of the generated text. Rouge-L calculates the F-score based on the Longest Common Subsequence (LCS) of the generated text and the ground truth text. For code vulnerability detection and code clone detection, we adopt F1, Precision (P), and Recall (R), which are widely used in previous works [133, 175].

Specifically, in the CL scenario, a sequence of models needs to be evaluated on all previously learned datasets. To better assess the method’s performance, we calculate the average performance following the same strategy that former works used [62, 149]:

$$\Omega = \frac{1}{m} \sum_{i=1}^m \Omega_i, \quad \Omega_i = \frac{1}{i} \sum_{j=1}^i \Omega_{i,j} \quad (5.8)$$

where $\Omega_{i,j}$ represents the performance on the j -th test set after learning the i -th dataset. Ω_i is the average performance of $\Omega_{i,j}$ on all test set from $j = 1$ to $j = i$, and Ω is the overall performance on all m datasets. Here, Ω can be any metric introduced above.

5.3.4 Implementation Details

The experiments are conducted on an Ubuntu GPU server with four RTX 3090 24GB GPUs. Our code is implemented by the deep learning framework PyTorch¹. The PLMs we used, namely CodeBERT and CodeT5, are from Huggingface², and we keep their default hyperparameter settings. We reproduce the performance of EWC, EMR, and PP-TF, referring to previous works [62, 225]. For all three tasks, we set the training epoch to 10 and the batch size to 16. But considering the different complexity of the three tasks and their used PLMs, we set the learning rate to 5e-5 for CS but 2e-5 for CVD and CCD. For the parameter-efficient methods in the CS task, we set the learning rate to 1e-3 as the parameters that can be tuned decrease dramatically. The prompt length is adjusted in the range of {10, 20, 30, 40, 50}, and the factor λ is adjusted in

¹<https://pytorch.org/>

²<https://huggingface.co/models>

{0.1, 0.5, 1, 5, 10, 50}. An elaborate analysis of the two parameters will be discussed in section 5.4.4.

5.4 Experimental Results

In this section, we conduct extensive experiments to answer the following research questions:

RQ1: How is the performance of vanilla prompt tuning in the continual learning scenario?

RQ2: How effective is PTDD when compared to baselines?

RQ3: What are the contributions of different components in PTDD?

RQ4: How do varied parameter settings influence the performance of PTDD?

RQ5: How does PTDD perform in the low-resource scenario?

5.4.1 RQ1: Vanilla Prompt Tuning Performance

Prompt tuning has become a popular approach in applying PLMs to specific downstream tasks, and researches on it emerge constantly in the field of software engineering [197, 198]. In the CL scenario, prompt-based methods also show advantages [225]. However, we criticize that the existing method lacks inheritance of previous knowledge, and the additional prompt selection module may lead to bias and error. We prefer to explore the effectiveness of using limited prompts. In this section, we first conduct an experimental study to investigate the performance of PT and PEPT, and expect some inspiring findings. We evaluate PT on all three code intelligence tasks, but evaluate PEFT only on the CS task, as they cannot converge on other tasks with tuning a tiny fraction of parameters based on our experiments. Table 5.1 and Table 5.2 illustrate their performance.

From the tables, we can observe that PT obtains consistent improvement when compared with the predominant tuning paradigm, i.e., fine-tuning. This finding is in accordance with previous works [197, 198]. For the CS task, Table 5.1 shows that PT’s average performance increases by 1.91%, 3.82%, and 1.83% in terms of BLEU, Meteor, and Rouge-L. The highest improvement is in the Go dataset, where the BLEU metric raises 4.99%. Besides, although the BLEU metric slightly declines in the Javascript

dataset, the other two metrics here are obviously improved. Thus, we argue that PT is still beneficial for those codes in Javascript. For the CVD and the CCD task, as shown in Table 5.3, the enhancement brought by PT can reach 3.33% and 2.18% in terms of the F1 score. The experimental results indicate that PT can achieve better performance in the CL scenario.

Due to the possible capability degradation issue, there are few works exploring the effectiveness of PEPT with PLMs in code intelligence tasks. In the experiment, we evaluate PEPT on the CS task. From Table 5.1, we can observe that the performance of PEPT is weaker than fine-tuning. The average decline is 1.66, 0.61, and 1.27 in terms of BLEU, Meteor, and Rouge-L, respectively. The observation shows that only depending on adjusting prompt parameters are difficult to adapt across domains, and it is necessary to tune PLMs when applying PT. Besides, although PT outperforms FT, they still lack a learning objective for stability. As shown in Figure 5.2, which illustrates the performance variation, their performance decreases when feeding new datasets. It inspires us to balance the plasticity and the stability objective when tuning.

Answer to RQ1: In the CL scenario, PT can consistently outperform FT on the studied three code intelligence tasks. However, the performance of PEPT is slightly weaker. The study inspires us that it is necessary to tune PLMs when applying PT, and we should balance the plasticity and the stability learning objective.

5.4.2 RQ2: Comparison with Baselines

In this section, we conduct comprehensive experiments to evaluate PTDD by comparing it with a series of baselines. We keep the backbone model used in all baselines the same as our method, namely CodeT5 for CS and CodeBERT for CVD and CCD. Table 5.1 and Table 5.2 show the detailed evaluation results, and we highlight the highest values among all rehearsal-free methods in bold. The values presented are the average performance calculated by Equation 5.8.

For the CS task, it can be observed from Table 5.1 that the average performance of PTDD on all six datasets can surpass FT by 2.55%, 4.37%, and 2.10% in terms of BLEU, Meteor, and Rouge-L. PTDD also obtains consistent improvements on all three metrics when compared with PT. For the regularization-based baseline EWC, PTDD outperforms it by 0.77% on average regarding the BLEU metric. Considering the additional computation required by EWC to select important parameters, PTDD omits this

Table 5.1. Evaluation results of code summarization. M and R are short for Meteor and Rouge-L, respectively.

Methods	Ruby			Javascript			Java			Go			PHP			Python		
	BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R
EMR	18.92	14.02	31.88	17.03	12.54	28.46	23.39	17.13	40.27	35.19	22.97	51.76	26.56	18.90	41.09	21.59	16.06	37.37
PEPT	15.89	12.05	28.53	15.14	11.62	26.95	20.85	15.48	38.16	31.67	20.97	49.11	24.62	17.79	39.88	20.49	15.04	36.55
PP-TF	17.06	12.49	29.71	15.65	11.58	27.41	20.14	14.76	37.02	28.65	19.38	46.33	24.28	17.19	39.13	20.22	14.75	35.95
FT	18.67	13.44	31.53	16.87	11.75	27.75	22.72	16.34	39.72	32.84	21.19	49.67	26.25	18.57	40.99	21.24	15.32	37.11
PT	18.75	13.56	31.69	16.61	12.34	28.66	23.28	16.96	40.33	34.48	22.75	51.22	26.49	18.79	41.39	21.63	15.90	37.63
EWC	18.76	13.71	31.77	16.93	12.35	28.91	23.05	16.79	40.08	34.35	22.47	51.20	26.44	18.65	41.16	21.52	15.92	37.54
PTDD	18.97	13.81	31.98	16.93	12.31	28.71	23.35	17.03	40.37	34.61	22.84	51.34	26.55	18.80	41.41	21.72	16.04	37.71
PE-PTDD	17.63	12.65	30.60	16.18	11.77	28.02	21.16	15.47	38.40	31.91	21.16	49.31	24.70	17.81	39.93	20.70	15.18	36.71
Ablation	18.93	13.72	31.88	16.89	12.35	28.79	23.26	16.94	40.31	34.48	22.75	51.27	26.53	18.82	41.40	21.67	16.02	37.64

Table 5.2. Evaluation results of code vulnerability detection and code clone detection.

Methods	CVD			CCD		
	F1	P	R	F1	P	R
EMR	32.55	42.90	27.50	85.01	89.11	81.45
FT	27.62	45.67	21.79	83.07	84.50	82.33
PT	28.54	43.64	23.44	84.88	84.92	85.45
EWC	28.61	46.71	24.07	83.34	83.97	83.43
PTDD	30.69	44.89	25.11	84.94	84.65	85.94
Ablation	29.20	42.94	24.08	84.76	83.82	86.28

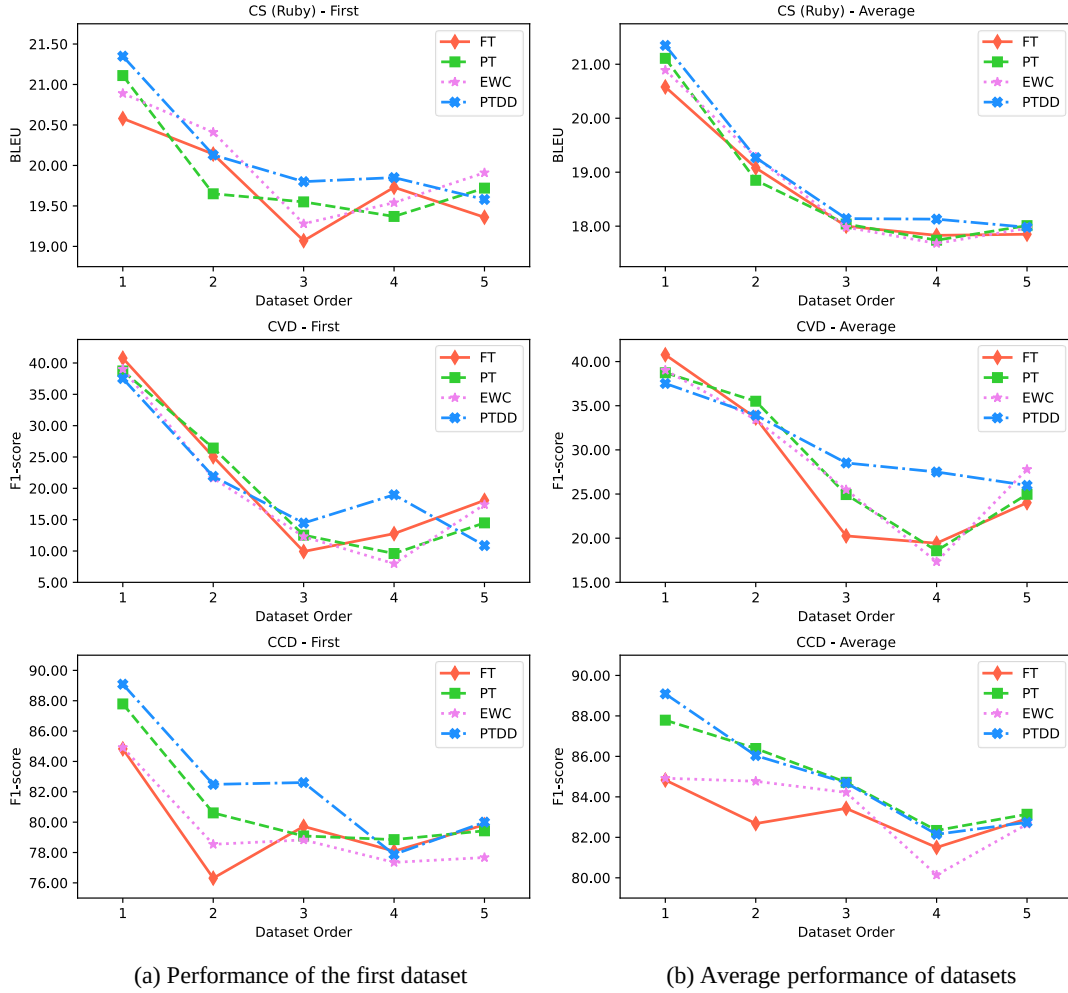


Figure 5.2. Performance recorded by the dataset order.

process, updates models and prompts in a backpropagation step, and can obtain a better performance, indicating its advantage in alleviating the catastrophic forgetting problem.

Even in comparison with the rehearsal-based baseline EMR, PTDD can achieve comparable performance. It surpasses EMR in the Ruby and Python datasets regarding the BLEU metric. Due to the need to store previous data, EMR becomes not applicable to data privacy sensitive scenarios. Our proposed PTDD gets rid of the shortcomings and realizes a rehearsal-free approach.

For the CVD task, as shown in Table 5.2, PTDD dramatically outperforms FT by 11.12% and 15.24% in terms of F1 and Recall, but mildly decreases in terms of Precision. When compared with PT, PTDD can obtain consistent superiority of 7.53%, 2.86%, and 7.13% regarding the F1 score, Precision, and Recall, respectively. Besides, PTDD performs better than EWC but weaker than EMR. This result shows that replaying a fraction of data samples is still effective in such a classification task. Whereas when considering the constraint of data privacy, PTDD performs best among all baselines. For the CCD task, PTDD can surpass FT by 2.25%, 0.18%, and 4.39% in terms of F1, Precision, and Recall. A similar advantage of PTDD can be observed in comparison with PT and EWC. Although it is slightly inferior to EMR, PTDD avoids the risk of data breaches and is practical in real-world scenarios.

Furthermore, we illustrate the performance variation when inputting a sequence of datasets in Figure 5.2. We select the Ruby dataset of the CS task and datasets used in the other two tasks as representatives. The performance trend of the first dataset and the average performance trend on previously learned datasets are presented separately. The horizontal axis is the input order of datasets, and the vertical axis is the value of the BLEU or F1 metric. From Figure 5.2, it can be observed that the catastrophic forgetting problem widely exists in the continual learning scenario. Our proposed PTDD can generally improve performance and slow decline trends, indicating it can alleviate forgetting of learned knowledge.

Answer to RQ2: The performance of PTDD can surpass FT, PT, and EWC generally in all three code intelligence tasks, indicating its superiority in the continual learning scenario. However, it is slightly inferior to the rehearsal-based baseline EMR as EMR stores and retrains a fraction of data samples, avoiding our rehearsal-free setting.

5.4.3 RQ3: Ablation Study

In this section, we conduct an ablation study from two aspects. The first is to investigate the contribution of the distilled prompt learning module to the performance of

PTDD. In addition, considering the prosperity of exploring parameter-efficient fine-tuning methods in code intelligence, we also detect how effective the PE-PTDD is, expecting advantages of our method in the parameter-efficient scenario. The ablation results are presented at the bottom of Table 5.1 and Table 5.2.

To validate the contribution of the distilled prompt learning module, we remove the stability learning objective and train models with the loss function in Equation 5.5. In the experiment, following the same experimental setting of RQ2, we use all datasets of the three tasks. For the CS task, as shown in Table 5.1, there is a consistent decline if discarding the module in terms of BLEU. Except for Javascript and PHP datasets, the same downward tendency occurs in terms of Meteor and Rouge-L. Although the drop is not dramatic, considering the reported values are the average on all five datasets and our prompt embedding strategy is still used, the consistent decline demonstrates the contribution of the module in improving the performance. For the CVD task, from Table 5.2, there is an obvious degradation of 1.49, 1.95, and 1.03 in terms of F1, Precision, and Recall. For the CCD task, performance drops in terms of F1 and Precision, but an inverse variation takes place in terms of Recall. As Precision and Recall sometimes conflict, the consistent drop in the F1 score also shows that the module can improve our method.

For the effectiveness of PE-PTDD, we experiment by freezing the backbone PLM and keeping the prompt parameters trainable. Therefore, we only update prompts through input datasets. We evaluate all six datasets of the code summarization task following the same setting of RQ1. As shown in Table 5.1, when compared with PTDD, PE-PTDD’s average performance decreases by 1.64, 1.13, and 1.43 in terms of BLEU, Meteor, and Rouge-L, respectively. Although PE-PTDD leads to degradations, we can observe that it still surpasses parameter-efficient baselines, namely PEPT and PP-TF. PE-PTDD outperforms them by 2.81% and 4.98% on average in terms of BLEU among all datasets (i.e., domains). The results indicate that our method is still effective if only tuning a fraction of the parameters.

Answer to RQ3: The distilled prompt learning module is essential for PTDD. In addition, our method still demonstrates robust effectiveness in the parameter-efficient scenario.

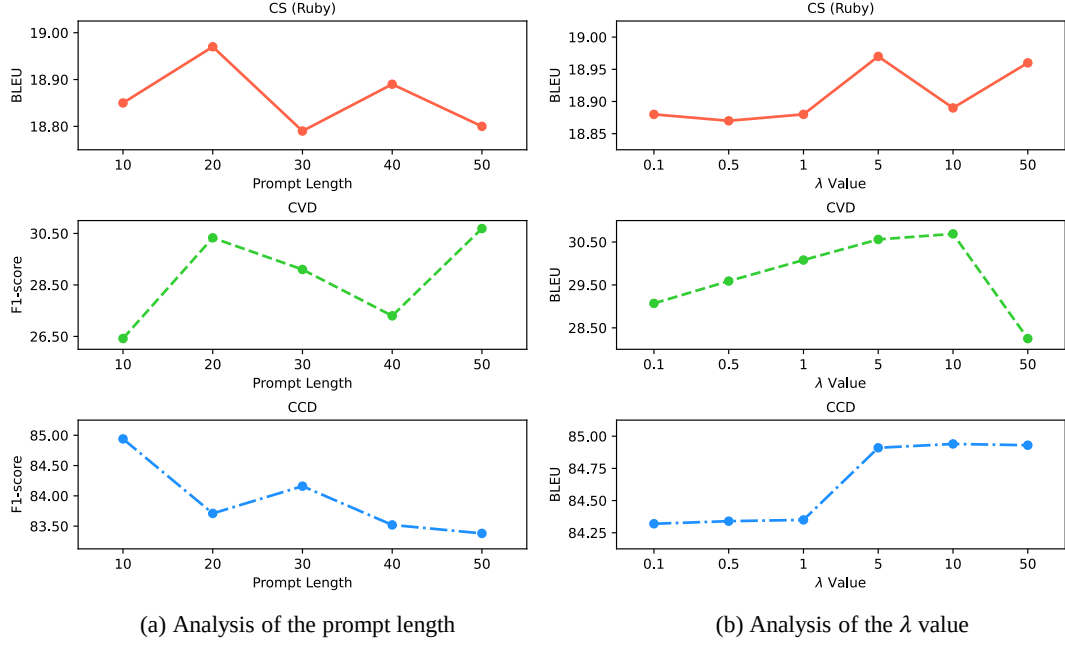


Figure 5.3. Parameter analysis of the prompt length and λ value.

5.4.4 RQ4: Parameter Analysis

In this section, we investigate the impact of the prompt length L_P and the factor λ on the performance of PTDD. Intuitively, a longer prompt means more trainable parameters and more powerful representation capability. However, excessively lengthy prompts may also tend to overfit the training data and be detrimental to models. Therefore, it is crucial to set an appropriate prompt length to uncover the attainable performance. Besides, the factor λ is used to balance the learning objectives of plasticity and stability. Exploring a proper value is beneficial when applying PTDD in practice. To do so, we consider the Ruby dataset of code summarization and datasets from the other two tasks for investigation, where we only vary the analyzed parameter and keep the others unchanged. The prompt length is set to 10, 20, 30, 40, and 50, and the factor λ is varied in the range of $\{0.1, 0.5, 1, 5, 10, 50\}$. Figure 5.3 illustrates the experimental results.

As shown in the figure, we can observe that the optimal prompt length for different datasets is distinguishing. For the Ruby dataset of CS, the best performance is achieved when L_P is set to 20. Larger or lower values do not give better results. For the CVD task, PTDD performs well when L_P equals 20 or 50, but the performance is slightly higher at 50. For the CCD task, setting L_P to 10 can obtain the best result. A performance drop occurs if the prompt length is continuously increased. We suspect that this

phenomenon may be related to the complexity of the tasks and the PLMs used, and we recommend adjusting an appropriate prompt length first if generalizing PTDD to other code intelligence tasks. As for the value of the factor λ , we can observe that there is a tendency that larger λ can obtain better results, and PTDD always achieves the best performance on the three datasets when λ is set to 5 or 10. Due to the more complex calculation requirements of the cross-entropy loss, λ needs to increase to emphasize the significance of keeping stability. The two losses become similar when λ is 5 or 10, thus improving PTDD’s performance. Through the parameter analysis, we reveal the influence of L_P and λ , as well as select an appropriate setting for our method.

Answer to RQ4: The prompt length L_P relates to the added trainable parameters and the factor λ concerns the significance of keeping stability. Based on the experimental results, we suggest adjusting an appropriate L_P first on new tasks, and adapting λ around 5 and 10.

5.4.5 RQ5: Performance in the Low-resource Scenario

Low resource is a common scenario for code intelligence tasks [28]. Especially for the continual learning scenario, consistently constructing large-scale datasets is difficult. In this section, we investigate the performance of PTDD when given only a small number of training samples, and we use the code summarization task as an example. For each dataset, we randomly select 10, 50, and 100 training samples, but keep the validation set and the test set unchanged. The training set size is about 0.02% - 2.3% for their original size. Table 5.3 presents the evaluation results.

From Table 5.3, it can be observed that PTDD can generally exceed FT, PT, and EWC, even surpassing EMR in the low-resource scenario. For instance, when compared with FT and the number of training samples is 10, 50, and 100, the BLEU score of PTDD can obtain average promotions of 44.76%, 53.10%, and 15.63%, respectively. Accordingly, the average promotions are 67.88%, 50.13%, and 7.43% when compared with EWC, and reach 69.09%, 45.70%, and 5.67% when compared with EMR. In comparison with PT, the improvements become not drastic and are 11.79%, 8.53%, and 1.43%, respectively. In addition, there is a tendency for the improvement of PTDD to be stark in fewer training samples. The result is explicable because PTDD adds extra prompts that are beneficial to elicit pre-trained knowledge. Thus, even with scarce training data, PTDD can achieve acceptable performance while FT and EWC suffer

Table 5.3. Evaluation results in the low-resource scenario. M and R are short for Meteor and Rouge-L, respectively.

Samples Methods	Ruby			Javascript			Java			Go			PHP			Python			
	BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R	
10 × 5	FT	5.92	5.31	4.61	3.51	2.47	3.99	3.32	1.96	3.25	4.24	2.78	5.53	2.24	1.98	1.63	7.93	6.19	6.93
	PT	9.47	8.32	14.33	4.51	4.60	8.12	4.12	4.28	7.13	6.34	8.98	12.96	3.03	2.64	3.24	6.94	6.30	11.00
	EWC	9.36	7.16	13.00	4.56	4.54	7.84	1.97	1.90	2.17	5.82	5.36	9.98	2.43	2.82	2.21	2.99	4.93	4.97
	EMR	9.19	7.20	13.85	4.41	4.30	7.27	2.03	1.81	2.23	3.73	3.48	6.15	1.97	2.31	1.79	4.34	5.42	5.86
	PTDD	9.57	8.69	14.70	4.79	5.24	9.09	5.38	5.23	9.55	6.83	8.43	12.72	3.19	2.71	3.42	8.32	7.47	14.27
50 × 5	FT	6.29	5.01	7.01	6.16	4.35	8.77	5.24	3.60	7.61	11.76	7.72	19.08	7.42	5.69	10.66	10.46	7.95	13.99
	PT	9.89	8.55	17.20	7.23	6.29	12.65	9.46	7.86	17.51	15.28	12.05	27.71	12.69	10.01	19.21	10.34	8.61	17.92
	EWC	9.93	7.84	14.43	8.49	6.58	14.34	3.99	3.65	5.90	10.94	6.84	16.97	6.60	5.29	8.47	10.66	8.12	14.70
	EMR	9.98	7.86	14.37	7.55	5.86	12.55	4.02	3.54	5.97	12.80	8.07	20.07	7.34	5.51	9.07	10.37	7.84	13.97
	PTDD	10.53	8.64	17.72	9.65	7.80	17.12	9.46	7.98	17.79	16.24	12.28	28.58	12.88	10.28	19.64	10.70	8.96	18.68
100 × 5	FT	12.14	9.03	18.96	11.75	8.72	20.17	13.92	10.80	26.19	18.15	12.10	29.84	14.78	11.01	23.52	15.90	11.73	27.39
	PT	14.36	11.23	25.16	12.47	9.81	22.33	15.22	12.01	29.00	22.85	16.22	38.84	18.52	14.01	30.40	15.73	12.07	28.53
	EWC	12.99	10.05	22.76	11.99	9.51	21.52	13.76	11.26	26.44	23.52	15.33	38.24	17.36	12.64	26.98	14.75	10.74	24.17
	EMR	13.38	10.14	23.33	12.12	9.59	21.75	13.64	11.05	26.11	23.57	15.40	38.18	17.97	13.16	28.55	15.21	11.43	25.73
	PTDD	14.59	11.25	25.27	12.51	9.67	21.77	15.35	12.05	29.08	23.59	16.28	39.37	18.60	14.05	30.56	16.07	12.80	29.36

from not having enough samples to learn task-specific knowledge. PT is affected because it does not mitigate the catastrophic forgetting problem. As for the EMR method, when reducing the training samples, the size of the stored samples is also reduced, and consequently, its performance deteriorates. The observations demonstrate that PTDD is more advantageous in the low-resource scenario and enlighten us on the practical value of PTDD.

Answer to RQ5: In the low-resource scenario, the performance of PTDD can generally surpass FT, PT, EWC, and even EMR. The average promotion can reach up to 69.09%. In addition, the improvement of PTDD is more stark when training samples are fewer.

5.5 Discussion

5.5.1 Visualization for PTDD

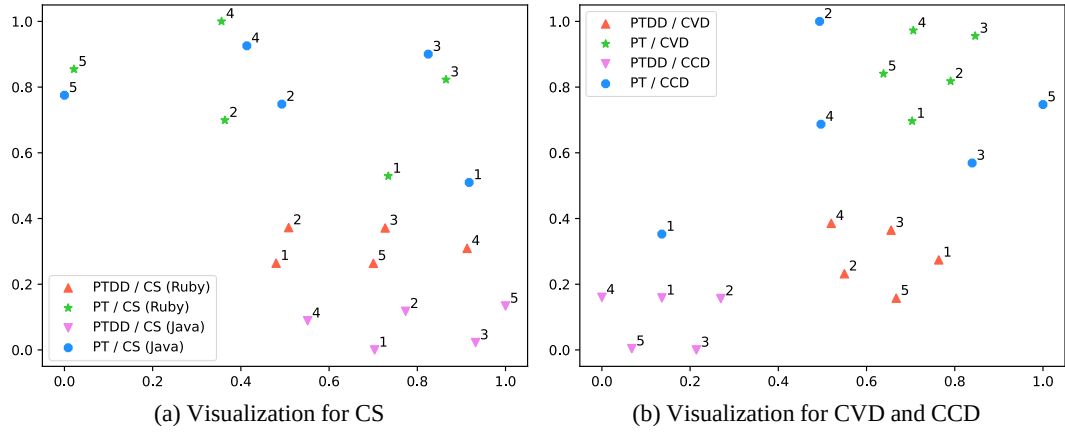


Figure 5.4. Prompt visualization in a representation space. We mark out the order of each prompt in its upper right corner.

To directly show the influence of PTDD in distilling prompts, we visualize the learned prompts in a two-dimensional representation space. Focusing on PT and our proposed PTDD, we carry out experiments on all three code intelligence tasks. In particular, regarding the CS task, Ruby and Java are selected for experiments. Subsequently, we use t-distributed Stochastic Neighbor Embedding (t-SNE) [194] on the learned prompts to obtain their two-dimensional representations. The t-SNE algorithm

is a non-linear dimensionality reduction technique for data exploration, giving an intuition on how data is arranged in higher dimensions. Figure 5.4 shows the visualization result. As there are five parts in each dataset, we mark out their order in the upper right corner.

As shown in Figure 5.4, we can observe that the prompt distribution of PTDD is compact while scattered for PT. The discrete distribution of PT indicates that different datasets belong to different domains, and there are significant diversities that can lead to catastrophic forgetting. However, because PTDD calculates the cosine similarity between current prompts and corresponding previous prompts during training and aims to minimize their differences, PTDD can restrict newly learned prompts not far from previous embeddings. Therefore, new prompts contain similar semantics to alleviate forgetting of learned knowledge. In addition, it can also be observed that the prompt distribution of PTDD is close to the first prompt embedding of PT. It demonstrates that when training on the first dataset, PTDD and PT can learn prompts with alike semantics. But when a sequence of datasets is inputted continually, PTDD can stabilize prompts' representations while PT can not. The visualization results directly explain the superiority of PTDD in the CL scenario.

5.5.2 Case Study

The advantage of PTDD comes from its utilization of prompts to instruct the representation of PLMs and thus mitigate the catastrophic forgetting of previously learned knowledge. In this section, as shown in Figure 5.5, we present two instances to intuitively show the strength. Evaluating on the Java dataset of the code summarization task, we inspect predictions of the first Java dataset. We compare the initial prediction with the final prediction, which are obtained after training on the first subdataset and all five subdatasets. For the first instance, it can be observed that the initial summarizations of fine-tuning and PTDD are similar and are aligned with the ground truth. But after training on the other four subdatasets, the output of PTDD does not change but fine-tuning varies, outputting "checks" instead of the original "returns true". For the second instance, after training on all five subdatasets, PTDD can still accurately output keywords "TypeConverter", but fine-tuning outputs some extra unnecessary words like "the list of registered converters". We find that many cases that do not work well on fine-tuning but do well on PTDD follow a similar pattern of the second case, i.e., the final fine-tuning summarization outputs some extra words that do not appear initially.

These cases indicate that simple fine-tuning tends to forget previously learned knowledge, thus producing low-quality summarization, while PTDD can catch the correct main idea of the source code because the learned knowledge is kept.

5.5.3 Training Cost Analysis

In this section, we evaluate the training cost of PTDD and compare it with other representative continual learning methods. Specifically, using the code summarization task as an example, we investigate the training time per epoch and the memory consumption of the fifth part of the Java dataset. The evaluation results are illustrated in Table 5.4. From Table 5.4 and the conclusion about the performance in Section 5.4.2, it can be observed that the training cost of PTDD is comparable to that of FT and PT, while it performs better. When compared with EWC, PTDD consumes fewer training resources and obtains better results. Because of the introduction of additional calculations to select important parameters, EWC needs more training time and memory consumption. In comparison with EMR, although the performance of PTDD is slightly inferior, it shows greater efficiency. Without retraining stored data samples, PTDD utilizes prompts to instruct the presentation of PLMs, which is lightweight and shows effectiveness and practicality.

Table 5.4. Training cost analysis of PTDD and other representative CL methods.

Methods	Training time	Memory Consumption
FT	11min22s	16,994M
PT	11min33s	17,784M
EWC	14min46s	20,392M
EMR	12min03s	18,084M
PTDD	11min35s	17,788M

5.5.4 Implications

This chapter presents the first experimental study to investigate the performance of prompt tuning set in the CL scenario in the code intelligence field, and proposes a rehearsal-free CL method named PTDD. In this section, we discuss some implications of this work from the perspectives of developers and researchers.

Code:

```
public boolean isToolSelected(int tool) {
    boolean selected = false;
    switch(tool) {
        case IBurpExtenderCallbacks.TOOL_PROXY :
            selected = jCheckBoxProxy.isSelected();
            break;
        case IBurpExtenderCallbacks.TOOL_REPEATER :
            selected = jCheckBoxRepeater.isSelected();
            break;
        case IBurpExtenderCallbacks.TOOL_SCANNER :
            selected = jCheckBoxScanner.isSelected();
            break;
        case IBurpExtenderCallbacks.TOOL_INTRUDER :
            selected = jCheckBoxIntruder.isSelected();
            break;
        case IBurpExtenderCallbacks.TOOL_SEQUENCER :
            selected = jCheckBoxSequencer.isSelected();
            break;
        case IBurpExtenderCallbacks.TOOL_TARGET :
            break;
        default :
            break;
    }
    return selected;
}
```

Initial Fine-Tuning Summarization: Returns true if the specified tool is currently selected.

Final Fine-Tuning Summarization: Checks if the specified tool is selected.

Initial PTDD Summarization: Returns true if the specified tool is selected.

Final PTDD Summarization: Returns true if the specified tool is selected.

Ground Truth: Returns true if the requested tool is selected in the GUI.

Code:

```
public void removeConverter(final TypeConverter tc) {
    if (tc.getSupportedTypes() == null) {
        untypedTypeEncoders.remove(tc);
        registeredConverterClasses.remove(tc.getClass());
    }
    else {
        for (final Entry<Class, List<TypeConverter>> entry : tcMap.entrySet()) {
            List<TypeConverter> list = entry.getValue();
            if (list.contains(tc)) {
                list.remove(tc);
            }
            if (list.isEmpty()) {
                tcMap.remove(entry.getKey());
            }
        }
        registeredConverterClasses.remove(tc.getClass());
    }
}
```

Initial Fine-Tuning Summarization: Remove a type converter.

Final Fine-Tuning Summarization: Remove a converter from the list of registered converters.

Initial PTDD Summarization: Remove a type converter.

Final PTDD Summarization: Remove a TypeConverter.

Ground Truth: Removes the type converter.

Figure 5.5. Examples of summarizations outputted by PTDD and Fine-Tuning.

Implications for developers. This work focuses on the continual learning scenario, where datasets are input sequentially to update PLMs so that they can be applied to wider domains. Considering the dynamic nature of software codebases, it is practical for developers in real-world scenarios. In addition, due to the potential restriction of data privacy in practice, this work discards the data selection and data storage module that previous works contain, and leverages the capability of prompt in eliciting pre-trained knowledge to realize a rehearsal-free approach, enlightening developers about how to apply prompt tuning in the CL scenario. Experimental results in the low-resource scenario show the superiority of PTDD with scarce data.

Implications for researchers. This work also obtains several interesting findings that might be instructive for researchers. In the experimental study of prompt tuning set in the CL scenario, we find that freezing the PLM parameters prominently damages models’ performance, especially for classification tasks such as code vulnerability detection and code clone detection. The results differ from findings in the NLP field [120], but are in accordance with previous studies in code intelligence [198]. We argue that this phenomenon is related to the size and quality of PLMs, as well as the complexity of downstream tasks. It deserves to be investigated in future works to better utilize prompt tuning. Besides, although PTDD is tailored for domain-incremental continual learning, its utility is not confined to domain-specific tasks. The core paradigm of leveraging prompts to preserve previously learned knowledge can readily extend to cross-domain continual learning. Nevertheless, given the pronounced distribution shifts in cross-domain scenarios, incorporating supplementary strategies, such as prompt pool, is beneficial for achieving superior performance and deserves deeper exploration.

5.5.5 Threats to Validity

External Validity mainly concerns the generalizability of our findings to other settings and datasets. One threat to the external validity comes from the limited used PLMs. Considering the computational resource constraints, our experiments are conducted on two widely used PLMs, namely CodeBERT and CodeT5. However, more PLMs should be considered due to their promising performance, such as DeepSeek-Coder [78] and CodeLlama [169]. As illustrated in the previous work [120], PT becomes more competitive with larger PLMs’ scale. It is deserved to explore our proposed PTDD on other PLMs in the future work. Besides, the limited choice of evaluation datasets is

another threat. In this work, we adopt the project-level continual learning datasets open-sourced by Gao et al. [62], which are large-scale and their original datasets are well-known. However, as the research about continual learning in code intelligence is just beginning, there is a lack of alternative datasets to enrich our experiments. For example, we follow the same setting to input five datasets sequentially, but it also deserves to be investigated if the total number of datasets becomes more. In the future, we will construct a more comprehensive dataset to expand the experiments.

Internal Validity mainly concerns the aspects that could impact the experimental results. The first threat to the internal validity lies in the hyperparameter settings. Finding the optimal hyperparameters is always a difficult task. In this work, besides some conventional hyperparameters like batch size and learning rate that follow the usual, we conduct specific parameter analysis to adjust the prompt length and the factor λ . Although our analysis does not include all possible parameter values, this aspect is not crucial to the study as we have already shown the performance of PTDD with reasonable hyperparameter values. In addition, the internal validity is also related to the replication of baseline models and evaluation metrics. To mitigate these threats, we strictly follow their designs to reproduce or directly utilize their source code for model construction. Therefore, we conclude that it is a lightweight threat to validity.

5.6 Chapter Summary

Concentrating on the CL scenario, this chapter thoroughly investigates the performance of vanilla prompt tuning and introduces PTDD, a novel rehearsal-free CL methodology. PTDD sequentially distills prompt representations and optimizes backbone models by incorporating a stability learning objective. From our first experimental study, we find that inheriting previous PLM parameters is beneficial in the CL scenario. For the effectiveness of our proposed method, extensive benchmarks across three representative code intelligence tasks demonstrate that PTDD can surpass a series of rehearsal-free baselines, even in the parameter-efficient setting. Furthermore, we unveil the underlying mechanism of PTDD via prompt visualization and evaluate its performance under data-sparse conditions. Finally, we summarize our findings and provide implications to inspire and scaffold subsequent endeavors in this domain.

Chapter 6

TerGEC: A Graph Enhanced Contrastive Approach for Program Termination Analysis

Programs exhibiting non-termination behavior can introduce severe vulnerabilities, such as denial-of-service and memory exhaustion, making early detection before deployment highly crucial. While existing execution-based and deep learning-based methods have made notable strides, they remain bottlenecked by complex sandbox environment dependencies and a pervasive deficit in fine-grained analysis. To overcome these limitations, this chapter introduces TerGEC, an execution-free, graph-enhanced contrastive approach. In detail, TerGEC analyzes behaviors of programs from Abstract Syntax Trees (ASTs), thereby capturing intra-class semantics both syntactically and lexically. Besides, it incorporates contrastive learning to learn the discrepancy between program behaviors of termination and non-termination, thereby acquiring inter-class semantics. Furthermore, we integrate graph augmentation to bolster robustness and employ a joint objective of weighted contrastive loss and focal loss to alleviate the inherent class-imbalance issue. Extensive evaluations across five Python and C datasets demonstrate that TerGEC achieves the best performance in general. Among all experimented datasets, TerGEC outperforms state-of-the-art baselines by 8.20% in terms of mAP and by 17.07% in terms of AUC on average, thereby validating the complementary synergy between intra-class and inter-class learning, as well as the effectiveness of our proposed class-imbalance solution.

The subsequent sections of this chapter proceed as follows. Section 6.1 presents the preliminary concepts, establishing essential notations and introducing GNN layers. Section 6.2 elaborates on the proposed TerGEC framework. Section 6.3 establishes the data preparation workflow, encompassing preprocessing and data transformation procedures. In section 6.4, we comprehensively evaluate the experimental results across four distinct research questions. Section 6.5 provides a deeper discussion, incorporating a critical analysis of limitations and potential threats to validity. Finally, section 6.6 encapsulates this chapter with a concise summary.

6.1 Preliminary

To formally illustrate our graph-enhanced contrastive approach, we first present some background knowledge in this section. It contains the notations of graphs and the calculation of some GNN layers that are used in TerGEC.

6.1.1 Notations

In this paper, we denote a graph as $G = (\mathcal{V}, \mathcal{E}, X)$, where $\mathcal{V}$ is the set of vertices and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of edges. $N = |\mathcal{V}|$ is the number of vertices. The graph structural and attributive features are represented as an adjacency matrix $A \in \{0, 1\}^{N \times N}$ and a feature matrix $X \in \mathbb{R}^{N \times F}$, where $A_{i,j} = 1$ iff $e_{i,j} \in \mathcal{E}$. F is the dimension of attribute vector, and $x_i \in \mathbb{R}^F$ is the feature of v_i .

6.1.2 GNN layers

Graph Convolutional Network (GCN), which is one of the most representative methods of GNN. Set H^l is the feature matrix in the l^{th} layer, that is $H^0 = X$ for original graphs and $H^0 = \tilde{X}$ for augmented graphs, the layer-wise propagation rule is:

$$H^{l+1} = \sigma \left(\tilde{D}^{-\frac{1}{2}} \hat{A} \tilde{D}^{-\frac{1}{2}} H^l W^l \right) \quad (6.1)$$

Here, $\hat{A} = A + I_N$ is the adjacency matrix A with the addition of the identity matrix I_N . This is actually adding self-connections on graphs to consider the influence from nodes themselves. $\tilde{D}$ is the degree matrix of $\hat{A}$, which is a diagonal matrix and $\tilde{D}_{ii} = \sum_j \hat{A}_{ij}$.

Adding the degree matrix $\tilde{D}$ can normalize $\hat{A}$ that can avoid neglecting the information from those nodes with few degrees. W^l is a layer-specific trainable weight matrix. $\sigma(\cdot)$ is an activation function, such as ELU [36], ReLU [152], PReLU [84].

Graph Attention Network (GAT), which implements the self-attention mechanism in the graph domain, emphasizing concentrating on the most relevant parts. In calculation, the preference is realized by attention coefficients, which is defined as:

$$e_{ij} = a([W^l h_i^l; W^l h_j^l]) \quad (6.2)$$

Where e_{ij} is the attention coefficient between node i and node j , indicating the importance of node j 's feature to node i . h_i^l is the input feature vector of node i in the l^{th} layer, and is also the i^{th} row of feature matrix H^l . h_j^l has a similar meaning. W^l is a trainable weight matrix for l^{th} layer. $[\cdot; \cdot]$ denotes the concatenation operator. The function $a(\cdot)$ maps the concatenation matrix to a real number. In the implementation, $a(\cdot)$ is a single-layer feedforward neural network, which is parameterized by a weight vector $\mathbf{a}$, and followed by an activation function LeakyReLU [144] to provide nonlinearity. Thus attention coefficients e_{ij} can be represented as:

$$e_{ij} = \text{LeakyReLU}(\mathbf{a}^T([W^l h_i^l; W^l h_j^l])) \quad (6.3)$$

Following the graph topology, only adjacency nodes are computed in the attention coefficient. It is necessary to normalize the coefficients so that makes them easily comparable. After normalization, the coefficients are expressed as:

$$\alpha_{ij} = \text{softmax}(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})} \quad (6.4)$$

Where $\mathcal{N}_i$ is the neighborhood of node i .

To calculate the output feature vector of node i in the l^{th} layer, namely $\hat{h}_i^l$, multi-head attention is applied that can concatenate attention coefficients from diverse representation spaces. Formally, with K independent attention mechanisms, $\hat{h}_i^l$ is defined as:

$$\hat{h}_i^l = \text{Concat}_{k=1}^K \left(\sigma \left(\sum_{j \in \mathcal{N}_i} \alpha_{ij}^k W^{lk} h_j^l \right) \right) \quad (6.5)$$

Here, $\sigma(\cdot)$ is an activation function. W^{lk} is the corresponding weight matrix W^l in the k^{th} attention mechanism.

Graph Isomorphism Network (GIN), which achieves maximum discriminative power for graph topology. It redesigns the aggregation process and combination process, and updates node representations by:

$$h_i^l = \text{MLP}^l \left((1 + \epsilon^l) \cdot h_i^{l-1} + \sum_{j \in \mathcal{N}_i} h_j^{l-1} \right) \quad (6.6)$$

Here, h_i^l denotes the feature of node i in the l^{th} layer. MLP^l is one multi-layer perceptron that is used in the l^{th} layer, to transform features to the output dimension. ϵ^l can be a trainable parameter or a fixed scalar, which can introduce a tiny difference in different GIN layers.

6.2 Research Methods

Figure 6.1 demonstrates the overall framework of TerGEC, which contains four components, i.e., graph augmentation, graph encoder, intra-class learning, and inter-class learning. Programs are first transformed into graphs, and then graphs are fed into the graph augmentation module. Graph augmentation is used to generate different views for input graphs, which is the foundation for the following inter-class learning and improves the robustness of the overall framework. Graph Encoder maps input graphs and augmented graphs to a feature space. Subsequently, a weighted contrastive loss and a focal loss are introduced in TerGEC to guide intra-class learning and inter-class learning, respectively. The above pipeline consequently forms our enhanced contrastive strategy.

6.2.1 Graph Augmentation

As for input programs, we first transform them into graph representations. The conversion process is a part of data preprocessing and is elaborated in section 6.3. Subsequently, samples are fed into the graph augmentation module. The objective of the augmentation module is to improve the sufficiency and diversity of samples [227], and is also the prerequisite for the following inter-class learning. In addition, augmentation has been shown to be effective in preventing deep networks from over-fitting and improving robustness [63, 224, 252]. To create rational data, graph augmentation usually follows certain methods to add perturbation on nodes, edges, or attributes.

In TerGEC, for input AST graphs, the structural information of programs is represented via the graph topology. To keep the topology and make ASTs meaningful, we propose two methods operating on attributive features (i.e., node representations) to augment graphs, i.e., feature masking and feature projection. It is flexible to use either of them in our framework.

Feature masking, which uses a randomly generated masking template to substitute a fraction of dimensions with zeros in attributive features. The masking template $M \in \{\mathbf{0}, \mathbf{1}\}^F$, where $\mathbf{0} \in \{0\}^{N \times 1}$ and $\mathbf{1} \in \{1\}^{N \times 1}$. The distribution for each column vector in M is a Bernoulli distribution, namely the column vector is $\mathbf{0}$ with probability p_m , and is $\mathbf{1}$ with probability $1 - p_m$. Thus, the augmented feature matrix $\tilde{X}$ is defined as:

$$\tilde{X} = M \circ X \in \mathbb{R}^{N \times F} \quad (6.7)$$

Where $\circ$ denotes the Hadamard product.

Feature projection, which aims to project the input attributive features to an approximate feature matrix. It has been proven that it can facilitate learning mixtures of Gaussians [42]. Let $P \in \mathbb{R}^{N \times N}$ be the projection matrix, whose element p_{ij} is randomly sampled and has a standard normal distribution, namely $p_{ij} \sim \mathcal{N}(0, 1)$. By multiplying P and X , the augmentation of feature matrix is computed by:

$$\tilde{X} = PX \in \mathbb{R}^{N \times F} \quad (6.8)$$

After graph augmentation, there are batches of augmented graphs that have the same topology as input graphs, and only attributive features are transformed. They all are sent to the same graph encoder for the following processes.

6.2.2 Graph Encoder

Graph encoder targets extracting graph-level representations, consisting of an embedding layer and a multi-layer GNN. The embedding layer converts features to an embedding space, where the dimension of features is F_e , and $F_e \ll F$. By embedding, sparse vectors are converted to dense vectors, which preserve the semantic information and reduce the dimensionality. It also decreases the number of parameters that can accelerate the calculation of TerGEC.

The multi-layer GNN is the backbone of encoding. Each GNN layer updates nodes' representations by aggregating the information from the neighbors of nodes. It can be represented formally as:

$$a_i^k = \text{AGGREGATE}^k (\{h_j^{k-1} : j \in \mathcal{N}_i\}) \quad (6.9)$$

$$h_i^k = \text{COMBINE}^k (h_i^{k-1}, a_i^k) \quad (6.10)$$

Here, a_i^k denotes the aggregation representation from the neighbors of node i in the k^{th} layer. h_i^k is the updated representation of node i in the k^{th} layer.

The number of layers indicates how many hops the aggregation can be executed. With the hop increasing, the central node can interact information with nodes from a wider scope. Figure 6.2 shows the interaction scope of the root node with 1 hop and 2 hops, respectively. Referring to the models utilized broadly in the graph classification tasks, we propose three models that can be used in the multi-layer GNN, i.e., GCN [115], GAT [196], GIN [216]. They use differentiated methods to implement the $\text{AGGREGATE}^k(\cdot)$ and $\text{COMBINE}^k(\cdot)$ functions. The details of corresponding calculations are demonstrated in section 6.1.2.

After multi-layer GNN, nodes' feature vectors are updated. As for ASTs, it means the syntactical and lexical information has been captured by the graph encoder. Moreover, to predict whether a program is terminated or not, we adopt a $\text{READOUT}(\cdot)$ function to aggregate nodes' feature vectors from each graph in the final GNN layer. It can be denoted as:

$$h_G = \text{READOUT} (\{h_i^k | i \in G\}) \quad (6.11)$$

Where h_G is the aggregated feature vector. Some usual methods for the $\text{READOUT}(\cdot)$ function are sum, mean, and max aggregators. In this paper, we use the mean aggregators to avoid influences from the number of nodes.

6.2.3 Intra-class Learning

In the intra-class learning module, we concentrate on original graphs G and their representations U that are extracted from the graph encoder. Therefore, we optimize TerGEC directly based on their respective labels. Similarly, an MLP as the projection layer $\theta^*(\cdot)$ is used for U firstly. To deal with the imbalance problem, instead

of traditional cross-entropy loss, we use focal loss [130] to optimize TerGEC during intra-class learning. Focal loss can be formally defined below:

$$L_{intra} = FL(p_t) = -\alpha_t (1 - p_t)^\gamma \log(p_t) \quad (6.12)$$

Here, p_t is the estimated probability of each class. If we use $y \in \{0, 1\}$ indicates labels, $p_t = p$ if $y = 1$, and $p_t = 1 - p$ otherwise. α_t has an analogous definition, which is a weighting factor. $(1 - p_t)^\gamma$ is a modulating factor, where γ is a focusing parameter. It is observed that focal loss is an extension of cross-entropy loss, which emphasizes adding weights for samples that are in the minority class and reducing the loss contribution from easy samples.

6.2.4 Inter-class Learning

Apart from learning from intra-classes, acquiring the discrepancies between programs of terminating and non-terminating from inter-classes is also crucial and conducive for termination detection. From the perspective of feature space, samples of different classes should be separated as much as possible, while samples of the same class should be as similar as possible, which can be seen as inter-class information. In TerGEC, following this motivation, we propose the inter-class learning module.

For a batch of graphs G and their augmentations $\tilde{G}$, their graph representations $U \in \mathbb{R}^{S \times D}$ and $\tilde{U} \in \mathbb{R}^{S \times D}$ are obtained after graph encoder, where S is the batch size and D is the dimension of graph feature vectors. Before calculating the difference, a nonlinear projection layer $\theta(\cdot)$ is added to transform graph representations to another hidden space that is used to compute the inter-class loss. It has been proven that a nonlinear projection can help improve the representation quality [32]. In the implementation, the projection layer is an MLP with a nonlinear activation function.

After projection, U and $\tilde{U}$ are transformed to U' and $\tilde{U}'$. We concatenate the two graph feature matrices and get $\hat{U} = [U'; \tilde{U}']$, computing inter-class loss based on $\hat{U}$. We set labels of augmented graphs as same as the corresponding original graphs. Thus, in comparison to U' , $\hat{U}$ contains more samples, whose quantity is $2S$, and introduces augmented positive samples that maintain the graph topology feature. In $\hat{U}$, each sample has its label, namely termination or non-termination, and samples can be separated according to their labels. What we target is to enforce the classifier to maximize the difference between samples of different classes. For each pair of samples in $\hat{U}$, i.e., $\hat{u}_i$

Algorithm 2 Graph Enhanced Contrastive Approach

Require: Batches of graphs G , which are transformed from programs, with their corresponding labels y that indicate termination or non-termination.

Ensure: Prediction results y' .

```
1: for epoch = 1, 2, ... do
2:   Generate the augmented graphs  $\tilde{G} = (\mathcal{V}, \mathcal{E}, \tilde{X})$ .
3:   Obtain graph representations  $U$  and  $\tilde{U}$  of input graphs  $G$  and augmented graphs  $\tilde{G}$  after the graph encoder.
4:   Project  $U$  that is obtained in step 3 by  $\theta^*(\cdot)$ .
5:   Calculate the intra-class loss  $L_{intra}$ .
6:   Project  $U$  and  $\tilde{U}$  by  $\theta(\cdot)$  to obtain  $U'$  and  $\tilde{U}'$ .
7:    $\hat{U} = [U'; \tilde{U}']$ .
8:   Calculate the inter-class loss  $L_{inter}$ .
9:   Combine the two optimization objectives and calculate  $L$ .
10:  Update parameters by gradient descent.
11: end for
12: Predict labels using the trained model.
13: return  $y'$ 
```

and $\hat{u}_j$, we utilize a normalized temperature-scaled cross-entropy loss function to capture their similarity, which is denoted as L_{inter}^{ij} . The difference is implicitly expressed in the other pairs of samples, i.e., $\hat{u}_i$ and $\hat{u}_k$, where $i \neq k$. Only pairs of samples that have the same label are accumulated for the inter-class loss L_{inter} , which can be seen as a weighted contrastive loss. It can be formally represented as:

$$L_{inter}^{ij} = -\log \frac{\exp(\varphi(\hat{u}_i, \hat{u}_j)/\tau)}{\sum_{k=1}^N \mathbf{1}_{i \neq k} \exp(\varphi(\hat{u}_i, \hat{u}_k)/\tau)} \quad (6.13)$$

$$L_{inter} = \sum_{i=1}^N \frac{\alpha}{N_{y_i} - 1} \sum_{j=1}^N \mathbf{1}_{i \neq j} \mathbf{1}_{y_i = y_j} L_{inter}^{ij} \quad (6.14)$$

Where $\varphi(\hat{u}_i, \hat{u}_j) = \hat{u}_i \cdot \hat{u}_j / \|\hat{u}_i\| \|\hat{u}_j\|$ is the cosine similarity function. τ is a temperature parameter. y_i is the label of sample $\hat{u}_i$, and N_{y_i} is the number of samples whose labels are as same as y_i in the batch. $\mathbf{1}_{i \neq j} \in \{0, 1\}$ and $\mathbf{1}_{y_i = y_j} \in \{0, 1\}$ are two indication functions. They equal to 1 iff $i \neq j$ and $y_i = y_j$ respectively. α is a weighted coefficient we set to alleviate the imbalance between termination samples and non-termination samples. According to their proportion in the batch, a higher α is assigned to samples that are in the minority class, namely non-termination. With the decreasing of L_{inter} ,

samples of the same class are gathered and samples of different classes are separated, which brings a brand-new optimization objective for high-quality termination analysis.

Finally, to combine the intra-class and inter-class semantic information together, we use a hyperparameter λ to balance their weights. The overall loss is defined as:

$$L = \lambda L_{inter} + (1 - \lambda) L_{intra} \quad (6.15)$$

Different from the training stage where the loss is produced by a weighted sum of intra-class and inter-class modules, during the inference stage, given a piece of code snippet, the predictive results of termination or not are produced by the intra-class learning module of TerGEC only.

The overall procedure of the proposed graph-enhanced contrastive approach is summarized in Algorithm 2.

6.3 Data Preparation

Existing termination detection datasets face some common problems. On the one hand, the number of data samples is relatively limited, usually several hundred or so, and some labels are missing. On the other hand, the proportion of termination samples and non-termination samples in these datasets is approximately balanced, which violates the practical situation that only a minority of programs are non-terminal. To address these limitations, we construct four custom datasets for termination detection.

6.3.1 Preprocessing

The raw datasets are derived from Large language models (LLMs)-generated programs. To be specific, it is based on two widely utilized code generation datasets, i.e., HumanEval [31] and MBPP [12]. The two datasets are widely acknowledged in the field of code generation, and they are equipped with test cases, which can be used to judge whether each program can terminate. To construct our datasets, we adopt the code generation results of both GPT-Neo 125M and GPT-Neo 1.3B [100], producing four distinct raw datasets, namely 125M_HumanEval, 125M_MBPP, 1_3B_HumanEval, and 1_3B_MBPP, respectively. GPT-Neo is an open-sourced model and is pre-trained on Pile dataset [60]. It has two versions whose parameters are 125M and 1.3B. We

chose GPT-Neo because there are existing results on HumanEval and MBPP, which has been published by [100]. Thus, it is convenient to construct the raw datasets. In addition, two code-generation models are involved which can guarantee the diversity of our four constructed datasets.

The raw constructed datasets include plentiful programs. HumanEval is a set of 164 hand-written programming problems, and MBPP contains 500 test problems. For each problem, there are about 100 generated programs. The raw number of each dataset is shown in Table 6.1. However, as are automatically generated, many of them may have grammatical errors. Therefore, we filter them and subsequently label the raw datasets with termination or not. Specifically, we use the test cases given by HumanEval and MBPP to evaluate each generated code. Programs without loops are first filtered out because a loop is a prerequisite for non-termination. When executing a program with equipped test cases, if it exceeds a predefined time limit, which is set as 10s in this work, it means the program is a “non-termination”. For the remaining programs, if they can pass all test cases, they will be annotated as “passed”. Or if they raise some error, they will be labeled as “failed”. There are many reasons that may cause a program failure. If the raised error is a logic error, which means the program can not pass all the test cases, but the program can still execute and terminate. To supplement our datasets, among all programs with logic errors, we pick up those containing loop iterations, and combine them with “passed” programs to constitute “termination”. Subsequently, programs labeled with termination and non-termination will be transformed into graph representations.

6.3.2 Data Transformation

To represent programs both syntactically and lexically, we transform them into AST graphs. AST is a homogeneous and undirected graph presenting program syntax via its hierarchy and non-leaf nodes while conserving the program lexical via leaf nodes, which is widely applied in compilers for static analysis. In this paper, those programs that have been preprocessed are converted to corresponding ASTs by an open-sourced parser generator tool, tree-sitter [24]. Figure 6.2 illustrates an example in 125M_HumanEval dataset. The values of leaf nodes in AST are shown in *italics*, such as “def” and “generate_integers”. Non-leaf nodes express different types in syntax, such as “module” and “parameters”, and they are organized hierarchically. Therefore, the content and structure of the original program can be fully represented in AST.

As for the initialization of attributive features, we maintain a dictionary for each dataset. The dictionary contains all types of non-leaf nodes and all values of leaf nodes that appear in the same dataset, and it maps each node to a one-hot vector. It is noted that we do not embed original node features in data preparation, but combine the embedding process with our methods. Table 6.1 manifests the statistics of our four custom datasets, including the number of graphs, the dimension of features, the average number of nodes and edges, as well as the number of different classes. As shown in the table, the positive samples merely take a small proportion.

6.4 Evaluation

To investigate the performance of TerGEC, we conduct plentiful experiments, including comparing with multiple state-of-the-art baselines and analyzing the improvement contributed by the components of TerGEC. Specifically, our evaluations follow four research questions below:

RQ1. Quantitative Evaluation. How does TerGec perform compared with the state-of-the-art baselines?

RQ2. Role of Graph Augmentation. Can our proposed augmentation strategies improve the performance in termination analysis?

RQ3. Backbone Analysis. Embedded with various backbone models, such as GCN, GAT, and GIN, how do they influence the performance of TerGEC?

RQ4. Effectiveness of Balance Factors. To what extent that the balance factors in TerGEC can improve performance in an imbalanced setting?

6.4.1 Experiment Setting

Baseline models. There are a total of eight state-of-the-art baselines that TerGEC compares with. Besides three fundamental methods, namely GCN [115], GAT [196], and GIN [216], we also compare TerGEC with some recent representative methods, concluding GGNN [124], GNN-FiLM [21], and graph transformer [176]. GGNN [124] extends the usage of gated recurrent units (GRU) to graph-structured problems so that it can update node and graph representation vectors. GNN-FiLM [21] introduces the

Table 6.1. Statistics for datasets.

Datasets	Raw Programs	Graphs	Features	Nodes	Edges	Termination (Negative)	Non-Termination (Positive)
125M_HumanEval	16400	3338	850	~70.1	~69.1	3250	88
125M_MBPP	47964	1336	921	~65.1	~64.1	1308	28
1_3B_HumanEval	15744	4562	886	~62.5	~61.5	4485	77
1_3B_MBPP	47978	10995	1695	~50.6	~49.6	10856	139
TPDB	206	206	220	~126.3	~125.3	158	48

Table 6.2. Experimental results for the baselines and TerGEC.

Model	125M_HumanEval		125M_MBPP		1_3B_HumanEval		1_3B_MBPP		TPDB	
	mAP(%)	AUC(%)	mAP(%)	AUC(%)	mAP(%)	AUC(%)	mAP(%)	AUC(%)	mAP(%)	AUC(%)
LSTM	54.64	68.27	62.30	64.77	50.00	50.06	50.01	49.95	53.17	56.71
GRU	54.49	62.52	56.31	71.50	51.09	75.26	50.32	67.28	57.14	58.33
GCN	60.08	81.56	71.49	85.17	58.51	88.59	58.84	90.29	61.02	84.18
GAT	64.51	83.54	71.54	87.22	59.33	91.23	59.61	92.08	63.50	88.89
GIN	63.18	86.26	66.06	85.10	60.90	90.52	59.31	91.87	65.18	87.21
GGNN	64.14	86.15	68.03	86.69	61.18	91.25	55.54	90.36	62.57	83.16
GNN-FiLM	64.92	85.21	65.90	87.22	58.67	89.56	54.71	88.74	63.38	82.49
Graph Transformer	65.46	87.91	67.57	86.24	61.20	89.33	58.33	88.96	63.20	83.16
TerGEC	70.05	84.11	74.10	92.32	61.38	90.20	60.61	93.33	68.35	91.58

graph hypernetworks and a feature-wise linear modulation to modify the message passing process that numerous GNNs used. Graph transformer [176] is inspired by the power of transformer in NLP, reforming it into graph learning tasks. In addition, two recurrent networks are also used to compare, namely Long Short-Term Memory (LSTM) [91] and Gated Recurrent Unit (GRU) [35]. It is noticed that there is also a branch of methods [66, 217] that require satisfiability modulo theories (SMT) solving to execute programs and check whether programs can terminate, but they have noteworthy drawbacks like needing complex configuration for complete sandbox environment and taking too much time in execution, leading to their unrealistic application in practice. Therefore, we exclude these methods from baselines and choose methods that do not need the execution process to compare.

Metrics. To evaluate the performance of TerGEC, we use mean Average Precision (mAP) and the Area Under the receiver operating characteristic Curve (AUC) as metrics. Average Precision (AP) summarizes the Precision-Recall curve to one scalar value, which can be seen as the weighted mean of precisions when setting different thresholds. Mean average precision is the average of APs for termination and non-termination instances. Receiver Operating characteristic Curve (ROC) shows the true positive rate and the false positive rate at each classification threshold, and AUC uses a value between 0 to 1 to manifest ROC. A higher AUC indicates better performance.

Datasets. We conduct extensive experiments on programs with Python and C language, which can evaluate the generalization ability of TerGEC. As for Python programs, we use four custom datasets, and their construction processes are introduced in section 6.3. As for C programs, we use an existing dataset, the Termination Problem DataBase (TPDB), which is from the TermComp termination competition [67]. Although it has been published by [67] as claimed in their paper, we cannot find the complete version. Therefore, we tried our best to collect a subset, which contains 206 C programs, with 158 termination programs and 48 non-termination programs. Its detailed statistics are shown in Table 6.1. All datasets are split into the training set, validation set, and testing set by the ratio of 7/1/2 for experiments.

Setup. The experiments were conducted on an Ubuntu GPU server with four RTX 3090 24GB GPUs. TerGec was implemented by the deep learning framework PyTorch¹ and a GNN package PyG [57]. Specifically, we set 4 layers for the GNN encoder and use Adam optimizer [114] whose learning rate is adjusted in $\{5e-4, 1e-4, 5e-5\}$

¹<https://pytorch.org/>

according to the datasets. Moreover, batch size is adjusted in $\{64, 128\}$ and the dimension of hidden vectors come within $\{128, 256\}$. A maximum training epoch is set to 200. As for the hyperparameters in the intra-class and inter-class learning module, we set $\alpha_t = 0.95$ and $\gamma = 2$ for the focal loss, which is a default setting in many previous studies [10, 130] and has been proved the effectiveness. We adjust the value of λ in the range of $(0, 0.5]$ and choose the value that performs best, which is $\lambda = 0.01$. An elaborated illustration of the role of λ has been discussed in section 6.4.5. In addition, we adjust the specific method used in the graph augmentation and the backbone. We consistently selected the best combination in each experimental dataset and recorded their performance. A detailed analysis of the two components is illustrated in section 6.4.3 and 6.4.4.

6.4.2 RQ1: Quantitative Evaluation

Comparing TerGEC with state-of-the-art baselines can directly indicate the superiority that TerGEC has. To achieve the best performance of TerGEC, we adjust some components like the graph augmentation method and the backbone used, reporting the best results. The effects of choosing different graph augmentation strategies and backbones will be further discussed in section 6.4.3 and 6.4.4. For the eight baselines, which follow the same setting of TerGEC, we implemented them by PyTorch and PyG as well. Table 6.2 shows the detailed results for the baselines and TerGEC.

As shown in the table, TerGEC achieves the best performance on almost all datasets and all evaluation metrics. It can be observed that the performance of TerGEC surpasses LSTM and GRU a lot, which indicates that the graph representation of programs benefits the termination analysis. Specifically, for the mAP metric, TerGEC outperforms baselines in all five datasets. In terms of 125M_HumanEval, there is an increment on mAP about 4.59% - 15.56%. In addition, the increments are 2.56% - 17.79% in terms of 125M_MBPP, 0.18% - 11.38% in terms of 1_3B_HumanEval, 1.00% - 10.60% in terms of 1_3B_MBPP, and 3.17% - 15.18% in terms of TPDB. The improvements indicate that TerGEC can accurately predict non-terminating programs among a mass of termination samples with a lower error rate. As for the AUC metric, TerGEC achieves the best on datasets 125M_MBPP, 1_3B_MBPP, and TPDB, where the improvements are 5.10% - 27.55%, 1.25% - 42.38%, and 2.69% - 34.87%, respectively. Although on the other two datasets, 125M_HumanEval and 1_3B_HumanEval, TerGEC does not perform best. We investigate this because TerGEC is equipped with

class-imbalance solutions; more non-terminating programs were found, but this also resulted in some misclassified termination samples, which leads to the decline of the AUC metric in the two datasets. Furthermore, besides Python program datasets, TerGEC outperforms all baselines in C program dataset TPDB, which manifests TerGEC has the generalization ability that is regardless of the programming language. It is worth noting that TerGEC and all baselines are evaluated under identical dataset configurations. Moreover, our evaluation across various dataset scales demonstrates consistent performance gains, confirming that the improvement inherently stems from TerGEC rather than variations in dataset size.

Answer to RQ1: When comparing with the state-of-the-art baselines, our proposed TerGEC can achieve better/comparable performance on both Python program datasets and C program datasets, which indicates that TerGEC can improve the prediction performance and has the generalization ability.

6.4.3 RQ2: Role of Graph Augmentation

The graph augmentation module is a key component of TerGEC that can increase the diversity of samples and improve the robustness of TerGEC. To investigate the impact of graph augmentation, we compare the performance of TerGEC using different graph augmentation strategies with a special version of TerGEC that removes the graph augmentation module. The different graph augmentation strategies are feature masking (FM) and feature projection (FP), as elaborated in section 6.2.1. They can yield a batch of new graphs to supplement the original graphs. When removing the graph augmentation module, no more new graphs are produced, and only the original graphs are sent to the following graph encoder to calculate the intra-class and inter-class semantic information. Table 6.3 illustrates the experimental results of graph augmentation. For the selection of backbones, to achieve the best results as that in RQ1, we choose GAT for dataset 125M_HumanEval and TPDB, and GIN for dataset 125M_MBPP, 1_3B_HumanEval, and 1_3B_MBPP.

It can be seen from Table 6.3 that, regardless of FM or FP, using graph augmentation can bring a better performance on all five datasets. The maximum increment can be 7.25% for mAP and 8.08% for AUC. The good performance not only verifies that graph augmentation is beneficial to non-termination prediction but also proves that FM and FP are two effective strategies that have the potential to be used in program analysis tasks. As for the performance of TerGEC with FM and FP, FP outperforms FM in

datasets 125M_MBPP and 1_3B_MBPP, while FM has a higher score in the other datasets. We suspect that it is because the programs generated by the MBPP dataset are relatively short, so simple feature masking may lose more semantic information. The experimental results show that FM and FP have their own merits and demerits, and choosing a proper graph augmentation strategy for different datasets is vital. FM is simple in calculation but has the probability of losing the crucial dimension. FP is realized by a randomly sampled projection matrix, which has a higher possibility of introducing noise.

Answer to RQ2: Graph augmentation is conducive to the prediction of non-termination and can improve the performance in termination analysis tasks. Feature masking and feature projection are two effective graph augmentation strategies in practice.

6.4.4 RQ3: Backbone Analysis

Graph encoder in TerGEC aims to capture the semantic information from ASTs. Hence, choosing an appropriate GNN model as the backbone of the graph encoder module influences the representation capability of TerGEC. Because of the difference in initial design purposes, GNN models have their preferences in dealing with different tasks. Generally speaking, GAT performs well in node feature representation, and GIN does better in graph-level applications [93]. To choose a proper backbone for termination analysis, we investigate three popular GNN models, namely GCN [115], GAT [196], and GIN [216], and evaluate the performance of their corresponding TerGEC. Table 6.4 shows the performance of TerGEC with different backbones.

From the results of Table 6.4, we can conclude that different backbones affect the performance of TerGEC. GAT achieves the best performance in dataset 125M_HumanEval and TPDB, and GIN wins in the other three. If we analyze the average node number of the five datasets, we can discover that dataset 125M_HumanEval and TPDB are higher than the others. This phenomenon confirms the advantages of GAT in node representation, which is more suitable for graphs with more nodes. On the contrary, when the node number descends, the superiority of GIN in graph representation appears. Meanwhile, we should note that the difference among different backbones might be minute. Thus, it is necessary to consider carefully when selecting specific backbones. Moreover, as can be seen from Table 6.2 and Table 6.4, when using the same backbone, TerGEC always has better performances than the original baselines. But there are also

Table 6.3. Experimental results about graph augmentation.

Model	125M_HumanEval		125M_MBPP		1_3B_HumanEval		1_3B_MBPP		TPDB	
	mAP(%)	AUC(%)	mAP(%)	AUC(%)	mAP(%)	AUC(%)	mAP(%)	AUC(%)	mAP(%)	AUC(%)
With FM	70.05	84.11	74.03	91.10	61.38	90.20	60.61	93.33	68.35	91.58
With FP	67.19	83.32	74.10	92.32	57.90	89.67	60.42	94.23	65.15	83.50
w/o Augmentation	62.80	83.08	73.90	90.87	57.80	88.23	57.09	92.95	63.33	83.50

Table 6.4. Experimental results with different backbones.

Model	125M_HumanEval		125M_MBPP		1_3B_HumanEval		1_3B_MBPP		TPDB	
	mAP(%)	AUC(%)	mAP(%)	AUC(%)	mAP(%)	AUC(%)	mAP(%)	AUC(%)	mAP(%)	AUC(%)
TerGEC-GCN	66.57	83.60	71.13	90.04	59.03	88.34	58.29	89.66	62.03	85.52
TerGEC-GAT	70.05	84.11	72.40	89.96	59.63	87.70	58.21	90.48	68.35	91.58
TerGEC-GIN	65.04	85.22	74.03	91.10	61.38	90.20	60.61	93.33	68.44	87.88

Table 6.5. Experimental results about balance factors.

Model	125M_HumanEval		125M_MBPP		1_3B_HumanEval		1_3B_MBPP		TPDB	
	mAP(%)	AUC(%)	mAP(%)	AUC(%)	mAP(%)	AUC(%)	mAP(%)	AUC(%)	mAP(%)	AUC(%)
TerGEC	70.05	84.11	74.03	91.10	61.38	90.20	60.61	93.33	68.35	91.58
$\lambda = 0.1$	60.20	83.04	68.31	87.53	57.63	90.42	61.06	93.25	64.72	82.49
$\lambda = 0.3$	58.97	84.04	68.04	88.06	58.04	90.97	60.29	93.45	63.95	80.47
w/o λ	58.33	82.12	67.38	89.89	57.51	88.99	56.21	93.18	63.97	79.80
w/o α	61.84	83.70	68.82	88.63	59.72	89.11	60.12	91.73	63.64	82.49

some counter-examples. For instance, in the 1_3B_MBPP dataset, when using GCN or GAT as the backbone, the performance decreases slightly if using TerGEC. Due to the difference is minute here, we conjecture it can be attributed to the random initialization of the model. On the whole, our proposed TerGEC performs better in most datasets and backbones, which proves the effectiveness of TerGEC.

Answer to RQ3: The choice of different backbones can affect the performance of TerGEC. Specifically, GAT is suitable for datasets with more nodes, like 125M_HumanEval and TPDB, and GIN does better in graph representation in the other three datasets.

6.4.5 RQ4: Effectiveness of Balance Factors

To cope with the imbalance classes problem in termination analysis, TerGEC introduces the inter-class learning module and corresponding balance factors. There are two kinds of balance factors in TerGEC. One of them is the coefficient α in equation 6.14, which aims to give a higher weight to non-terminating samples that are in the minority, and another is the hyperparameter λ in equation 6.15, which is used to balance the proportion between inter-class and intra-class semantic information. To evaluate the effectiveness of the two balance factors, we conduct ablation experiments of TerGEC by removing α and λ , respectively. That is to set $\alpha = 1$ in equation 6.14 and $\lambda = 0.5$ in equation 6.15. In addition, we also set $\lambda = 0.1$ and $\lambda = 0.3$ to observe the change of performance when λ is varied. Table 6.5 demonstrates the performances in detail. The other settings are also the same as those of RQ1.

Table 6.5 distinctly presents a drop in prediction performance if removing either α or λ , proving that balance factors are indispensable in TerGEC. The decline could be 3.87% - 11.72% for the mAP metric and 1.21% - 11.78% for the AUC metric in the five datasets. To be specific, the drop caused by the loss of λ is more obvious than that of α . In fact, due to the more complex calculation requirements of the inter-class loss, its value is usually higher than the intra-class loss. When increasing the value of λ , the importance of intra-class loss is weakened, and the performance decreases. Balance factor λ plays an important role here in balancing the inter-class loss and the intra-class loss, bringing TerGEC to the optimum of classification. Similarly, the balance factor α in equation 6.14 enlarges the effects from the non-terminating samples and conducts TerGEC to learn more semantic information from non-termination. Therefore, the two balance factors help TerGEC improve its performance in an imbalanced setting.

Answer to RQ4: Balance factors are effective when disposing of the class-imbalance problem in termination analysis. The coefficient α and hyperparameter λ boost the performance of TerGEC in the five experimental datasets.

6.5 Discussion

6.5.1 Limitation

Our work shows efficiency and effectiveness and enhances the performance of deep learning-based methods in termination analysis, but there are still limitations. To provide an estimation of a program’s termination behavior, our method aims to learn the discrepancies in characteristics between non-terminating programs and terminating ones. However, it cannot provide formal proof of termination, which means the prediction may consist of some false positives, and the explainability is also weak. To try to alleviate the above limitations inherent in deep learning-based approaches, we designed inter-class and intra-class joint learning methodologies to improve the prediction performance of neural networks. As for explainability, we will further design a localizer to specify the defective lines and use visualization methods to provide necessary hints in future work. In addition, with the rapid development and extensive studies of pre-trained language models [136, 230] and large language models [223, 232], their application and adaptation in termination analysis are also worthy of trying. In the future, we will incorporate them into our framework for a more comprehensive investigation.

6.5.2 Threats to Validity

External Validity mainly concerns the generalizability of our findings to other datasets. TerGEC takes four custom datasets and one existing dataset into account, including Python and C languages. It is the first large-scale dataset for termination analysis. However, as TerGEC learns the semantic information from the training set for termination analysis, the dataset quality directly influences the performance. As for the four custom datasets, since we predefined a maximum execution time to identify non-termination programs, some programs that can terminate but may exceed the predefined execution time might be mislabeled. But we have set the execution time to be large enough to

alleviate the threats to validity, which is set to 10s. Thus, we believe this threat is minimal. In addition, following previous studies [10, 66], we only consider programs with loops and analyze whether they can terminate, but exclude infinite recursion and corresponding non-termination issues. It is worth considering the infinite recursion problem in future work.

Interval Validity mainly concerns the aspects that may impact the experimental results. One potential aspect is the hyperparameter setting. Finding the optimal hyperparameters is always a difficult task. In TerGEC, besides conventional hyperparameters, such as batch size and embedding dimension, a proper graph augmentation strategy, a suitable backbone, and an opportune setting of balance factors are also needed to adjust. Although we have analyzed these in section 6.4.3, 6.4.4, and 6.4.5, there are still some settings that are not considered, like if the hyperparameter λ is set to other values, and we keep it for future work. In addition, threats are also related to the bias in the replication of baseline models. To replicate baselines on our datasets, we followed their designs and implemented them using PyTorch and PyG. We have double-checked and fully tested our replication code to reduce errors. Therefore, we believe that it is a little threat to validity.

6.6 Chapter Summary

This chapter introduces TerGEC, a graph-enhanced contrastive approach tailored for program termination analysis. By simultaneously capturing both intra-class and inter-class semantics of programs, TerGEC facilitates a more fine-grained representation. Additionally, graph augmentation is introduced to enhance robustness, while weighted contrastive loss and focal loss are incorporated to tackle the pervasive class-imbalance problem. To evaluate our approach, we construct four large-scale datasets that reflect real-world class-imbalance issues, pioneering the first comprehensive benchmark study for termination analysis in software engineering. Extensive evaluations demonstrate that TerGEC generally outperforms state-of-the-art baselines across both Python and C programs. Ablation studies further confirm that the graph-enhanced contrastive approach that combines the intra-class and inter-class semantic information, along with our proposed class-imbalance solutions, effectively improves the prediction performance for non-terminating programs.

Chapter 7

Conclusion and Future Work

This thesis presents a series of novel methodologies and comprehensive experimental evaluations dedicated to advancing code intelligence across the dual dimensions of efficiency and adaptability. Our investigation spans four core perspectives: high-efficiency fine-tuning paradigms, exploration of efficient model architectures, dynamic adaptive learning strategies, and structure-aware adaptive learning approaches. In this chapter, we conclude the findings established in these works and outline strategic directions for future research.

7.1 Conclusion

In the area of high-efficiency fine-tuning paradigms, we systematically investigate the competitive edge of PEFT relative to FMFT within code-change-related tasks. These tasks are of particular interest due to the inherent objective mismatch between pre-training and downstream fine-tuning requirements. Specifically, our empirical evaluation encompasses four prevalent PEFT methods (AT, LoRA, PT, and PreT) across two pivotal downstream tasks: JIT-DP and CMG. Our findings reveal that while PEFT variants consistently outperform FMFT on JIT-DP, with AT and LoRA achieving SOTA results, they only exhibit comparable performance on CMG, albeit with reduced resource consumption. Crucially, PEFT demonstrates relative superiority under cross-lingual and low-resource settings. To elucidate the performance variance between PEFT and FMFT, we conduct three probing tasks that evaluate their code semantic representations from both static and dynamic perspectives. Based on the probing insights, we

propose Pasta_K, a novel approach that harnesses the layer-wise complementary advantages of FMFT and PEFT to achieve superior domain adaptation. Experimental results show that Pasta_K significantly outperforms various PEFT baselines. Remarkably, even when compared to FMFT, Pasta_K achieves maximum performance gains of 1.48%, 3.21%, and 1.87% in terms of BLEU, Meteor, and Rouge-L, respectively. Concurrently, Pasta_K maintains exceptional efficiency, reducing training time by 26.26% and computational memory usage by 20.65% relative to FMFT, thereby underscoring its substantial practical value.

Focusing on the exploration of efficient model architectures, we evaluate Mamba-based models, which are sub-quadratic alternatives that bypass the quadratic complexity of the Transformer, across three representative code intelligence tasks. Our experimental results demonstrate that under standard full fine-tuning, Mamba-based PLMs exhibit performance parity or clear dominance relative to their similarly sized Transformer-based counterparts. Crucially, under PEFT settings, the superiority of Mamba-based PLMs becomes substantially more pronounced. Furthermore, we provide empirical evidence establishing the architectural advantages of Mamba for code intelligence, alongside its efficient learning ability in low-resource scenarios. In particular, when trained in an extremely data-scarce setting (just 0.1% of the dataset), Mamba-based PLMs exhibit remarkable robustness, with a performance degradation of only 3.69% in the Java code completion task. Finally, we extend our comparative inquiry to larger scales, specifically benchmarking models with 370M and 7B parameters. We observe that while scaling up parameters consistently yields performance gains for Mamba-based PLMs, their performance is inherently bounded by pre-training data and tasks, causing their competitive advantage over Transformer-based counterparts to diminish.

When it comes to dynamic adaptive learning strategies, we propose PTDD, a novel rehearsal-free continual learning framework tailored for code intelligence. Short for Prompt Tuning with Domain-wise Distillation, PTDD can sequentially distill prompts and optimize backbone models by introducing an auxiliary stability learning objective. Our preliminary experiments reveal that leveraging inherited PLM parameters conduces to preserving learned knowledge, yielding benefits for continual learning. Subsequently, comprehensive experiments across three code intelligence tasks demonstrate that PTDD significantly outperforms a series of rehearsal-free baselines. Specifically, it achieves performance gains of 2.55%, 11.12%, and 2.25% over full fine-tuning

across these tasks. Furthermore, we provide intuitive insights into PTDD’s effectiveness through prompt visualization and evaluate its performance in low-resource scenarios. Notably, the performance gains become even more pronounced with limited training data, exhibiting a remarkable improvement of up to 69.09%.

Regarding structure-aware adaptive learning approaches, we introduce TerGEC, a graph-enhanced contrastive approach for program termination analysis. It simultaneously captures the intra-class and inter-class semantics of programs, enabling a more fine-grained analysis. In addition, a series of graph augmentation mechanisms is designed to enhance robustness. To mitigate the class-imbalance problem, TerGEC incorporates both weighted contrastive loss and focal loss. Concurrently, we construct four large-scale datasets that faithfully reflect real-world class-imbalance issues to evaluate our method. Compared with SOTA baselines, TerGEC generally achieves the best performance across both Python and C programs. On average, it surpasses existing competitive methods by 8.20% in mAP and 17.07% in AUC across all benchmark datasets. Ablation experiments further demonstrate that the graph-enhanced contrastive approach that integrates both intra-class and inter-class semantic information, coupled with our proposed class-imbalance solutions, effectively improves the prediction performance for non-terminating programs.

7.2 Future Work

While our proposed approaches demonstrate substantial efficacy in advancing code intelligence across the twin dimensions of efficiency and adaptability, there remain potential directions for further refinement.

In the area of high-efficiency fine-tuning paradigms, we conduct experiments on two pivotal code-change-related tasks: Just-In-Time Defect Prediction (JIT-DP) and Commit Message Generation (CMG). While alternative tasks exist within the scope of code change learning, such as patch identification [90] and comment updating [231], we strategically prioritize JIT-DP and CMG as they form a representative dual-paradigm (classification vs. generation) that robustly generalizes to the core challenges of code intelligence. Moving forward, we plan to expand our evaluation by incorporating more relevant tasks to rigorously verify the generalizability of our findings and delineate their operational boundaries. Besides, the probing tasks currently deployed to dissect the dynamic semantic-level information can be designed with finer granularity. Future

investigations could incorporate advanced probing tasks for fine-grained exploration, such as evaluating whether PLMs can precisely detect code changes like the insertion of a variable declaration. Additionally, our proposed Pasta_K combines FMFT and PEFT based on their layer-wise probing results, successfully striking a desirable trade-off between efficiency and performance. In the future, we could explore automating this combination for diverse backbone architectures without relying on manual probing analysis.

Focusing on the exploration of efficient model architectures, we present the first systematic empirical study of Mamba-based PLMs across three representative code intelligence tasks. At the 130M parameter scale, we comprehensively evaluate the performance of Mamba-based PLMs against Transformer-based baselines across various fine-tuning paradigms, including FT and PEFT. Furthermore, we dissect their unique architectural advantages and explore their performance in low-resource scenarios. At larger scales, however, the prohibitive computational overhead and resource constraints inherent in foundational pre-training preclude us from training large-scale Mamba architectures from scratch. In the future, we plan to conduct a rigorous and unified pre-training process to establish a fair comparison between Mamba and Transformer architectures at larger scales. The advantage of Mamba-based PLMs in handling longer code snippets is also worth exploring. In addition, looking beyond Mamba, other non-Transformer architectures, including Spiking Neural Network [65] and Generative Adversarial Networks [70], are also worthy of investigation in future studies.

When it comes to dynamic adaptive learning strategies, our preliminary experimental investigation into prompt tuning within the CL paradigm reveals that freezing the backbone PLM parameters severely compromises model performance, particularly for classification tasks. While this phenomenon stands in contrast to established findings in the NLP domain [120], it aligns with recent empirical observations in code intelligence [198]. We attribute it to a combination of factors, including the parameter scale and pre-training quality of the PLMs, as well as the complexity of downstream tasks. Consequently, unraveling the underlying mechanisms of this phenomenon remains a critical future endeavor to unlock the full potential of prompt tuning. Furthermore, despite the surging research interest in continual learning for code intelligence, this field still suffers from a scarcity of comprehensive evaluation benchmarks. In the future, it is imperative to construct cross-domain or cross-lingual benchmarks comprising a broader number of datasets, thereby facilitating a more rigorous evaluation of diverse CL methods.

Regarding structure-aware adaptive learning approaches, we introduce an inter-class and intra-class joint learning framework to enhance the prediction performance for termination analysis. However, our current analysis focuses exclusively on program loops and their termination behaviors, following previous studies [10, 66]. Infinite recursion and its associated non-termination issues remain outside the scope of this study. Additionally, because our framework currently lacks the capacity to generate formal proofs of termination, its underlying decision-making process remains a black box. In the future, it is highly worthwhile to extend our scope to include the infinite recursion problem, incorporating a designed fault localization mechanism to pinpoint faulty program statements and leveraging visualization techniques to deliver essential explainability hints. It is also worth systematically investigating the optimal number of GNN layers for AST-based termination analysis. Moreover, beyond ASTs, other graph-structured code representations, such as Control Flow Graphs (CFGs) and Data Flow Graphs (DFGs), merit deep investigation to capture multi-faceted dependencies and yield highly interpretable predictions.

References

- [1] A. Abdu, Z. Zhai, H. A. Abdo, R. Algabri, M. A. Al-Masni, M. S. Muhammad, and Y. H. Gu, “Semantic and traditional feature fusion for software defect prediction using hybrid deep learning model,” *Scientific Reports*, vol. 14, no. 1, p. 14 771, 2024.
- [2] *About us - mistral ai*, <https://mistral.ai/about>.
- [3] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, “A transformer-based approach for source code summarization,” in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 4998–5007.
- [4] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, “Unified pre-training for program understanding and generation,” in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Jun. 2021, pp. 2655–2668. [Online]. Available: <https://www.aclweb.org/anthology/2021.naacl-main.211>.
- [5] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, “Typilus: Neural type hints,” in *Proceedings of the 41st acm sigplan conference on programming language design and implementation*, 2020, pp. 91–105.
- [6] M. Allamanis, M. Brockschmidt, and M. Khademi, “Learning to represent programs with graphs,” in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, OpenReview.net, 2018. [Online]. Available: <https://openreview.net/forum?id=BJOFETxR->.
- [7] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, “Self-supervised bug detection and repair,” *Advances in Neural Information Processing Systems*, vol. 34, pp. 27 865–27 876, 2021.

- [8] 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.
- [9] A. A. Almeida and M. Ayala-Rincón, “Formalizing the dependency pair criterion for innermost termination,” *Science of Computer Programming*, vol. 195, p. 102474, 2020.
- [10] Y. Alon and C. David, “Using graph neural networks for program termination,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 910–921.
- [11] A. Aurum, R. Jeffery, C. Wohlin, and M. Handzic, *Managing software engineering knowledge*. Springer Science & Business Media, 2013.
- [12] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al., “Program synthesis with large language models,” *arXiv preprint arXiv:2108.07732*, 2021.
- [13] 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.
- [14] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang, et al., “Qwen technical report,” *arXiv preprint arXiv:2309.16609*, 2023.
- [15] S. Banerjee and A. Lavie, “Meteor: An automatic metric for mt evaluation with improved correlation with human judgments,” in *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, 2005, pp. 65–72.
- [16] 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.
- [17] A. M. Ben-Amram and S. Genaim, “Ranking functions for linear-constraint loops,” *Journal of the ACM (JACM)*, vol. 61, no. 4, pp. 1–55, 2014.
- [18] A. M. Ben-Amram and S. Genaim, “On multiphase-linear ranking functions,” in *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, Springer, 2017, pp. 601–620.

- [19] X. Bi, D. Chen, G. Chen, S. Chen, D. Dai, C. Deng, H. Ding, K. Dong, Q. Du, Z. Fu, et al., “Deepseek llm: Scaling open-source language models with longtermism,” *arXiv preprint arXiv:2401.02954*, 2024.
- [20] A. R. Bradley, Z. Manna, and H. B. Sipma, “Linear ranking with reachability,” in *Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings 17*, Springer, 2005, pp. 491–504.
- [21] M. Brockschmidt, “Gnn-film: Graph neural networks with feature-wise linear modulation,” in *International Conference on Machine Learning*, PMLR, 2020, pp. 1144–1152.
- [22] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Nee-lakantan, P. Shyam, G. Sastry, A. Askell, et al., “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [23] I. I. Brudaru and A. Zeller, “What is the long-term impact of changes?” In *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering*, Atlanta, Georgia, 2008, pp. 30–32.
- [24] M. Brunsfeld, P. Thomson, A. Hlynskyi, et al., *Tree-sitter*, <https://github.com/tree-sitter/tree-sitter>.
- [25] Y. Cao, H.-R. Wei, B. Chen, and X. Wan, “Continual learning for neural machine translation,” in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2021, pp. 3964–3974.
- [26] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, “Deep learning based vulnerability detection: Are we there yet?” *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3280–3296, 2021.
- [27] K. Chatterjee, E. K. Goharshady, P. Novotný, and Đ. Žikelić, “Proving non-termination by program reversal,” in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 1033–1048.
- [28] F. Chen, F. H. Fard, D. Lo, and T. Bryksin, “On the transferability of pre-trained language models for low-resource programming languages,” in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 401–412.

- [29] G. Chen, F. Liu, Z. Meng, and S. Liang, “Revisiting parameter-efficient tuning: Are we really there yet?” In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, 2022, pp. 2612–2626.
- [30] H.-Y. Chen, B. Cook, C. Fuhs, K. Nimkar, and P. O’ Hearn, “Proving nontermination via safety,” in *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20*, Springer, 2014, pp. 156–171.
- [31] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.
- [32] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, “A simple framework for contrastive learning of visual representations,” in *International conference on machine learning*, PMLR, 2020, pp. 1597–1607.
- [33] X. Chen, F. Xu, Y. Huang, N. Zhang, and Z. Zheng, “Jit-smart: A multi-task learning framework for just-in-time defect prediction and localization,” *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1–23, 2024.
- [34] R. Chopra, S. Roy, and R. Malhotra, “Transductive instance transfer learning for cross-language defect prediction,” in *Proceedings of the 2022 4th Asia Pacific Information Technology Conference*, 2022, pp. 176–182.
- [35] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” *arXiv preprint arXiv:1412.3555*, 2014.
- [36] D. Clevert, T. Unterthiner, and S. Hochreiter, “Fast and accurate deep network learning by exponential linear units (elus),” in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: <http://arxiv.org/abs/1511.07289>.
- [37] *Codestral mamba - mistral ai*, <https://mistral.ai/news/codestral-mamba>.
- [38] A. Conneau, G. Kruszewski, G. Lample, L. Barrault, and M. Baroni, “What you can cram into a single $\mathbb{R}^d$ vector: Probing sentence embeddings for linguistic properties,” in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2018, pp. 2126–2136.

- [39] B. Cook, C. Fuhs, K. Nimkar, and P. O’Hearn, “Disproving termination with overapproximation,” in *2014 Formal Methods in Computer-Aided Design (FM-CAD)*, IEEE, 2014, pp. 67–74.
- [40] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk, “On automatically generating commit messages via summarization of source code changes,” in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, 2014, pp. 275–284.
- [41] 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.
- [42] S. Dasgupta, “Experiments with random projection,” *arXiv preprint arXiv:1301.3849*, 2013.
- [43] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019, pp. 4171–4186.
- [44] N. Ding, Y. Qin, G. Yang, F. Wei, Z. Yang, Y. Su, S. Hu, Y. Chen, C.-M. Chan, W. Chen, et al., “Parameter-efficient fine-tuning of large-scale pre-trained language models,” *Nature Machine Intelligence*, vol. 5, no. 3, pp. 220–235, 2023.
- [45] J. Dong, Y. Lou, Q. Zhu, Z. Sun, Z. Li, W. Zhang, and D. Hao, “Fira: Fine-grained graph-based code change representation for automated commit message generation,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 970–981.
- [46] 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.
- [47] 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.

- [48] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, “Boa: A language and infrastructure for analyzing ultra-large-scale software repositories,” in *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 422–431.
- [49] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, “Ac/c++ code vulnerability dataset with code changes and cve summaries,” in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 508–512.
- [50] L. Fan, J. Liu, Z. Liu, D. Lo, X. Xia, and S. Li, “Exploring the capabilities of llms for code-change-related tasks,” *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 6, pp. 1–36, 2025.
- [51] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, “Functional code clone detection with syntax and semantics fusion learning,” in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 516–527.
- [52] C. Feng, Y. Sun, K. Li, P. Zhou, J. Lv, and A. Lu, “Genetic auto-prompt learning for pre-trained code intelligence language models,” *arXiv preprint arXiv:2403.13588*, 2024.
- [53] S. Feng, J. Keung, X. Yu, Y. Xiao, and M. Zhang, “Investigation on the stability of smote-based oversampling techniques in software defect prediction,” *Information and Software Technology*, vol. 139, p. 106 662, 2021.
- [54] S. Feng and C. Chen, “Prompting is all you need: Automated android bug replay with large language models,” in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.
- [55] 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.
- [56] P. Fernandes, M. Allamanis, and M. Brockschmidt, “Structured neural summarization,” in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, OpenReview.net, 2019. [Online]. Available: <https://openreview.net/forum?id=H1ersoRqtm>.
- [57] M. Fey and J. E. Lenssen, “Fast graph representation learning with PyTorch Geometric,” in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

- [58] F. Frohn and J. Giesl, “Proving non-termination via loop acceleration,” in *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*, C. W. Barrett and J. Yang, Eds., IEEE, 2019, pp. 221–230. [Online]. Available: <https://doi.org/10.23919/FMCAD.2019.8894271>.
- [59] Z. Fu, H. Yang, A. M.-C. So, W. Lam, L. Bing, and N. Collier, “On the effectiveness of parameter-efficient fine-tuning,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, 2023, pp. 12 799–12 807.
- [60] 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.
- [61] S. Gao, C. Gao, Y. He, J. Zeng, L. Nie, X. Xia, and M. Lyu, “Code structure-guided transformer for source code summarization,” *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 1, pp. 1–32, 2023.
- [62] S. Gao, H. Zhang, C. Gao, and C. Wang, “Keeping pace with ever-increasing data: Towards continual learning of code intelligence models,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, IEEE, 2023, pp. 30–42.
- [63] X. Gao, R. K. Saha, M. R. Prasad, and A. Roychoudhury, “Fuzz testing based data augmentation to improve robustness of deep neural networks,” in *Proceedings of the acm/ieee 42nd international conference on software engineering*, 2020, pp. 1147–1158.
- [64] 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.
- [65] S. Ghosh-Dastidar and H. Adeli, “Spiking neural networks,” *International journal of neural systems*, vol. 19, no. 04, pp. 295–308, 2009.
- [66] M. Giacobbe, D. Kroening, and J. Parsert, “Neural termination analysis,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 633–645.
- [67] J. Giesl, A. Rubio, C. Sternagel, J. Waldmann, and A. Yamada, *The termination and complexity competition*. Springer, 2019.

- [68] D. Goel, R. Grover, and F. H. Fard, “On the cross-modal transfer from natural language to code through adapter modules,” in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 71–81.
- [69] L. Gonnord, D. Monniaux, and G. Radanne, “Synthesis of ranking functions using extremal counterexamples,” *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 608–618, 2015.
- [70] 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.
- [71] M. Gori, G. Monfardini, and F. Scarselli, “A new model for learning in graph domains,” in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, IEEE, vol. 2, 2005, pp. 729–734.
- [72] A. Gu and T. Dao, “Mamba: Linear-time sequence modeling with selective state spaces,” in *First Conference on Language Modeling*, 2024.
- [73] 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.
- [74] Y. Gu, X. Han, Z. Liu, and M. Huang, “PPT: Pre-trained prompt tuning for few-shot learning,” in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 8410–8423. [Online]. Available: <https://aclanthology.org/2022.acl-long.576>.
- [75] 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.
- [76] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, “Graphcodebert: Pre-training code representations with data flow,” in *9th International Conference on Learning Representations, ICLR 2021*. [Online]. Available: <https://openreview.net/forum?id=jLoC4ez43PZ>.

- [77] 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.
- [78] 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.
- [79] 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.
- [80] A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu, “Proving non-termination,” in *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2008, pp. 147–158.
- [81] J. T. Halloran, M. Gulati, and P. F. Roysdon, “Mamba state-space models can be strong downstream learners,” *arXiv e-prints*, arXiv–2406, 2024.
- [82] W. Hamilton, Z. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” *Advances in neural information processing systems*, vol. 30, 2017.
- [83] X. Han, Y. Dai, T. Gao, Y. Lin, Z. Liu, P. Li, M. Sun, and J. Zhou, “Continual relation learning via episodic memory activation and reconsolidation,” in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 6429–6440.
- [84] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification,” in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.
- [85] R. He, L. Liu, H. Ye, Q. Tan, B. Ding, L. Cheng, J. Low, L. Bing, and L. Si, “On the effectiveness of adapter-based tuning for pretrained language model adaptation,” in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, Online: Association for Computational Linguistics, Aug. 2021, pp. 2208–2222. [Online]. Available: <https://aclanthology.org/2021.acl-long.172>.

- [86] 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.
- [87] S. Herbold, A. Trautsch, B. Ledel, A. Aghamohammadi, T. A. Ghaleb, K. K. Chahal, T. Bossenmaier, B. Nagaria, P. Makedonski, M. N. Ahmadabadi, et al., “A fine-grained data set and analysis of tangling in bug fixing commits,” *Empirical Software Engineering*, vol. 27, no. 6, p. 125, 2022.
- [88] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, “Deepjit: An end-to-end deep learning framework for just-in-time defect prediction,” in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, IEEE, 2019, pp. 34–45.
- [89] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, “Cc2vec: Distributed representations of code changes,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 518–529.
- [90] T. Hoang, J. Lawall, Y. Tian, R. J. Oentaryo, and D. Lo, “Patchnet: Hierarchical deep learning-based stable patch identification for the linux kernel,” *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2471–2486, 2019.
- [91] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
- [92] 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.
- [93] Z. Hou, X. Liu, Y. Cen, Y. Dong, H. Yang, C. Wang, and J. Tang, “Graph-mae: Self-supervised masked graph autoencoders,” in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2022, pp. 594–604.
- [94] 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.
- [95] 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.

- [96] 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, 2025, pp. 3751–3759.
- [97] 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.
- [98] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “Code-searchnet challenge: Evaluating the state of semantic code search,” *arXiv preprint arXiv:1909.09436*, 2019.
- [99] R. A. Husein, H. Aburajouh, and C. Catal, “Large language models for code completion: A systematic literature review,” *Computer Standards & Interfaces*, vol. 92, p. 103 917, 2025.
- [100] J. P. Inala, C. Wang, M. Yang, A. Cudas, M. Encarnación, S. Lahiri, M. Musuvathi, and J. Gao, “Fault-aware neural code rankers,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 13 419–13 432, 2022.
- [101] 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.
- [102] 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.
- [103] 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.
- [104] Y. S. Iznaga, “Enhancing software testing automation through large language models,” M.S. thesis, Universidade de Évora, 2025.
- [105] G. Jawahar, B. Sagot, and D. Seddah, “What does bert learn about the structure of language?” In *ACL 2019-57th Annual Meeting of the Association for Computational Linguistics*, 2019.
- [106] 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.

- [107] S. Jiang, A. Armaly, and C. McMillan, “Automatically generating commit messages from diffs using neural machine translation,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2017, pp. 135–146.
- [108] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, “A large-scale empirical study of just-in-time quality assurance,” *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.
- [109] S. Kang, J. Yoon, and S. Yoo, “Large language models are few-shot testers: Exploring llm-based general bug reproduction,” in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE ’23, Melbourne, Victoria, Australia: IEEE Press, 2023, pp. 2312–2323. [Online]. Available: <https://doi.org/10.1109/ICSE48619.2023.00194>.
- [110] A. Karmakar and R. Robbes, “What do pre-trained code models know about code?” In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2021, pp. 1332–1336.
- [111] 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.
- [112] J. Ketema and A. F. Donaldson, “Termination analysis for gpu kernels,” *Science of Computer Programming*, vol. 148, pp. 107–122, 2017.
- [113] P. Khosla, P. Teterwak, C. Wang, A. Sarna, Y. Tian, P. Isola, A. Maschinot, C. Liu, and D. Krishnan, “Supervised contrastive learning,” *Advances in neural information processing systems*, vol. 33, pp. 18 661–18 673, 2020.
- [114] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: <http://arxiv.org/abs/1412.6980>.
- [115] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, OpenReview.net, 2017. [Online]. Available: <https://openreview.net/forum?id=SJU4ayYgl>.

- [116] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, et al., “Overcoming catastrophic forgetting in neural networks,” in *PNAS Proceedings of the National Academy of Sciences of the United States of America*, National Academy of Sciences, 2017.
- [117] T. C. Le, T. Antonopoulos, P. Fathololumi, E. Koskinen, and T. Nguyen, “Dynamite: Dynamic termination and non-termination proofs,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.
- [118] M. Lei, H. Li, J. Li, N. Aundhkar, and D.-K. Kim, “Deep learning application on code clone detection: A review of current knowledge,” *Journal of Systems and Software*, vol. 184, p. 111 141, 2022.
- [119] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, “Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 919–931.
- [120] B. Lester, R. Al-Rfou, and N. Constant, “The power of scale for parameter-efficient prompt tuning,” in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 3045–3059.
- [121] 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.
- [122] X. L. Li and P. Liang, “Prefix-tuning: Optimizing continuous prompts for generation,” in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2021, pp. 4582–4597.
- [123] X. Li, S. Yuan, X. Gu, Y. Chen, and B. Shen, “Few-shot code translation via task-adapted prompt learning,” *Journal of Systems and Software*, vol. 212, p. 112 002, 2024.
- [124] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, “Gated graph sequence neural networks,” in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: <http://arxiv.org/abs/1511.05493>.

- [125] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu, et al., “Automating code review activities by large-scale pre-training,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1035–1047.
- [126] V. Lialin, V. Deshpande, and A. Rumshisky, “Scaling down to scale up: A guide to parameter-efficient fine-tuning,” *arXiv preprint arXiv:2303.15647*, 2023.
- [127] B. Lin, S. Wang, Z. Liu, Y. Liu, X. Xia, and X. Mao, “Cct5: A code-change-oriented pre-trained model,” in *Proceedings of the 31th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE, 2023.
- [128] C.-Y. Lin, “Rouge: A package for automatic evaluation of summaries,” in *Text summarization branches out*, 2004, pp. 74–81.
- [129] 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.
- [130] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, “Focal loss for dense object detection,” in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2980–2988.
- [131] 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.
- [132] 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.
- [133] 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.
- [134] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, “Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing,” *arXiv preprint arXiv:2107.13586*, 2021.

- [135] S. Liu, D. Guo, J. Zhang, W. Ma, Y. Li, and Y. Liu, “An empirical study of exploring the capabilities of large language models in code learning,” *IEEE Transactions on Software Engineering*, 2025.
- [136] 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.
- [137] S. Liu, J. Keung, Z. Yang, Z. Mao, and Y. Sun, *Can mamba be better? an experimental evaluation of mamba in code intelligence*, Sep. 2025. [Online]. Available: <https://github.com/ishuoliu/ASE-CMamba>.
- [138] X. Liu, M. Masana, L. Herranz, J. Van de Weijer, A. M. Lopez, and A. D. Bagdanov, “Rotate your networks: Better weight consolidation and less catastrophic forgetting,” in *2018 24th International Conference on Pattern Recognition (ICPR)*, IEEE, 2018, pp. 2262–2268.
- [139] X. Liu, Y. Zheng, Z. Du, M. Ding, Y. Qian, Z. Yang, and J. Tang, “Gpt understands, too,” *AI Open*, 2023.
- [140] Z. Liu, Z. Tang, X. Xia, and X. Yang, “Ccrep: Learning code change representations via pre-trained code model and query back,” in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE ’23, Melbourne, Victoria, Australia, 2023, pp. 17–29. [Online]. Available: <https://doi.org/10.1109/ICSE48619.2023.00014>.
- [141] 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.
- [142] 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.
- [143] Y. Ma, N. Yan, J. Li, M. Mortazavi, and N. V. Chawla, “Hetgpt: Harnessing the power of prompt tuning in pre-trained heterogeneous graph neural networks,” in *Proceedings of the ACM on Web Conference 2024*, 2024, pp. 1015–1023.

- [144] A. L. Maas, A. Y. Hannun, A. Y. Ng, et al., “Rectifier nonlinearities improve neural network acoustic models,” in *Proc. icml*, Atlanta, Georgia, USA, vol. 30, 2013, p. 3.
- [145] J. Martinez-Gil, “Evaluating small-scale code models for code clone detection,” *arXiv preprint arXiv:2506.10995*, 2025.
- [146] P. Martins, R. Achar, and C. Lopes, “50k-c: A dataset of compilable, and compiled, java projects,” May 2018, pp. 1–5.
- [147] M. McCloskey and N. J. Cohen, “Catastrophic interference in connectionist networks: The sequential learning problem,” in *Psychology of learning and motivation*, vol. 24, Elsevier, 1989, pp. 109–165.
- [148] L. A. Meyerovich and A. S. Rabkin, “Empirical analysis of programming language adoption,” in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013, pp. 1–18.
- [149] F. Mi, L. Chen, M. Zhao, M. Huang, and B. Faltings, “Continual learning for natural language generation in task-oriented dialog systems,” in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 3461–3474.
- [150] *Models –hugging face*, Sep. 2025. [Online]. Available: <https://huggingface.co/models>.
- [151] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, “Convolutional neural networks over tree structures for programming language processing,” in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, 2016.
- [152] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.
- [153] K. Napier, T. Bhowmik, and S. Wang, “An empirical study of text-based machine learning models for vulnerability detection,” *Empirical Software Engineering*, vol. 28, no. 2, p. 38, 2023.
- [154] C. Ni, W. Wang, K. Yang, X. Xia, K. Liu, and D. Lo, “The best of both worlds: Integrating semantic features with expert features for defect prediction and localization,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 672–683.

- [155] 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.
- [156] M. Nita and D. Notkin, “Using twinning to adapt programs to alternative apis,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 205–214.
- [157] 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.
- [158] 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.
- [159] I. Paul, G. Glavaš, and I. Gurevych, “Ircoder: Intermediate representations make language models robust multilingual code generators,” *arXiv preprint arXiv:2403.03894*, 2024.
- [160] J. Pfeiffer, A. Rücklé, C. Poth, A. Kamath, I. Vulić, S. Ruder, K. Cho, and I. Gurevych, “AdapterHub: A framework for adapting transformers,” in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Online: Association for Computational Linguistics, Oct. 2020, pp. 46–54. [Online]. Available: <https://aclanthology.org/2020.emnlp-demos.7>.
- [161] D. Pigott, *Online historical encyclopaedia of programming languages*, 2020.
- [162] A. Podelski and A. Rybalchenko, “A complete method for the synthesis of linear ranking functions,” in *Verification, Model Checking, and Abstract Interpretation: 5th International Conference, VMCAI 2004 Venice, Italy, January 11-13, 2004 Proceedings 5*, Springer, 2004, pp. 239–251.
- [163] C. Pornprasit and C. K. Tantithamthavorn, “Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction,” in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, IEEE, 2021, pp. 369–379.
- [164] 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.

- [165] 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.
- [166] A. Razdaibiedina, Y. Mao, R. Hou, M. Khabsa, M. Lewis, and A. Almahairi, “Progressive prompts: Continual learning for language models,” in *International Conference on Learning Representations*, 2023.
- [167] S.-A. Rebuffi, A. Kolesnikov, G. Sperl, and C. H. Lampert, “Icarl: Incremental classifier and representation learning,” in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 5533–5542.
- [168] 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.
- [169] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, et al., “Code llama: Open foundation models for code,” *arXiv preprint arXiv:2308.12950*, 2023.
- [170] I. Saberi, F. Fard, and F. Chen, “Utilization of pre-trained language model for adapter-based knowledge transfer in software engineering,” *arXiv preprint arXiv:2307.08540*, 2023.
- [171] H. Shao, D. Sun, J. Wu, Z. Zhang, A. Zhang, S. Yao, S. Liu, T. Wang, C. Zhang, and T. Abdelzaher, “Paper2repo: Github repository recommendation for academic papers,” in *Proceedings of The Web Conference 2020*, 2020, pp. 629–639.
- [172] Z. Sheng, Z. Chen, S. Gu, H. Huang, G. Gu, and J. Huang, “Llms in software security: A survey of vulnerability detection techniques and insights,” *ACM Computing Surveys*, vol. 58, no. 5, pp. 1–35, 2025.
- [173] E. Shi, Y. Wang, L. Du, J. Chen, S. Han, H. Zhang, D. Zhang, and H. Sun, “On the evaluation of neural code summarization,” in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 1597–1608.
- [174] E. Shi, Y. Wang, W. Tao, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun, “RACE: Retrieval-augmented commit message generation,” in *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, Dec. 2022, pp. 5520–5530.

- [175] E. Shi, Y. Wang, H. Zhang, L. Du, S. Han, D. Zhang, and H. Sun, “Towards efficient fine-tuning of pre-trained code models: An experimental study and beyond,” in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 39–51.
- [176] Y. Shi, Z. Huang, S. Feng, H. Zhong, W. Wang, and Y. Sun, “Masked label prediction: Unified message passing model for semi-supervised classification,” in *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, Z. Zhou, Ed., ijcai.org, 2021, pp. 1548–1554. [Online]. Available: <https://doi.org/10.24963/ijcai.2021/214>.
- [177] J. S. Smith, L. Karlinsky, V. Gutta, P. Cascante-Bonilla, D. Kim, A. Arbelle, R. Panda, R. Feris, and Z. Kira, “Coda-prompt: Continual decomposed attention-based prompting for rehearsal-free continual learning,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 11 909–11 919.
- [178] B. Steenhoek, M. M. Rahman, R. Jiles, and W. Le, “An empirical study of deep learning models for vulnerability detection,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, IEEE, 2023, pp. 2237–2248.
- [179] W. Sun, C. Fang, Y. Chen, Q. Zhang, G. Tao, Y. You, T. Han, Y. Ge, Y. Hu, B. Luo, et al., “An extractive-and-abstractive framework for source code summarization,” *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 3, pp. 1–39, 2024.
- [180] W. Sun, Y. Miao, Y. Li, H. Zhang, C. Fang, Y. Liu, G. Deng, Y. Liu, and Z. Chen, “Source code summarization in the era of large language models,” in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, IEEE, 2025, pp. 1882–1894.
- [181] 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.
- [182] W. Tan and Y. Li, “Synthesis of ranking functions via dnn,” *Neural Computing and Applications*, vol. 33, pp. 9939–9959, 2021.

- [183] C. Tao, Q. Zhan, X. Hu, and X. Xia, “C4: Contrastive cross-language code clone detection,” in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 413–424.
- [184] W. Tao, Y. Wang, E. Shi, L. Du, S. Han, H. Zhang, D. Zhang, and W. Zhang, “On the evaluation of commit message generation models: An experimental study,” in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2021, pp. 126–136.
- [185] W. Tao, Y. Zhou, Y. Wang, H. Zhang, H. Wang, and W. Zhang, “Kadel: Knowledge-aware denoising learning for commit message generation,” *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 5, pp. 1–32, 2024.
- [186] Y. Tay, M. Dehghani, D. Bahri, and D. Metzler, “Efficient transformers: A survey,” *ACM Computing Surveys*, vol. 55, no. 6, pp. 1–28, 2022.
- [187] C. Team, H. Zhao, J. Hui, J. Howland, N. Nguyen, S. Zuo, A. Hu, C. A. Choquette-Choo, J. Shen, J. Kelley, et al., “Codegemma: Open code models based on gemma,” *arXiv preprint arXiv:2406.11409*, 2024.
- [188] G. Team, T. Mesnard, C. Hardin, R. Dadashi, S. Bhupatiraju, S. Pathak, L. Sifre, M. Rivière, M. S. Kale, J. Love, et al., “Gemma: Open models based on gemini research and technology,” *arXiv preprint arXiv:2403.08295*, 2024.
- [189] I. Tenney, D. Das, and E. Pavlick, “BERT rediscovers the classical NLP pipeline,” in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 4593–4601. [Online]. Available: <https://aclanthology.org/P19-1452>.
- [190] I. Tenney, P. Xia, B. Chen, A. Wang, A. Poliak, R. T. McCoy, N. Kim, B. V. Durme, S. Bowman, D. Das, and E. Pavlick, “What do you learn from context? probing for sentence structure in contextualized word representations,” in *International Conference on Learning Representations*, 2019. [Online]. Available: <https://openreview.net/forum?id=SJzSgnRcKX>.
- [191] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al., “Llama: Open and efficient foundation language models,” *arXiv preprint arXiv:2302.13971*, 2023.

- [192] C. Urban, A. Gurfinkel, and T. Kahsai, “Synthesizing ranking functions from bits and pieces,” in *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings 22*, Springer, 2016, pp. 54–70.
- [193] G. M. Van de Ven, T. Tuytelaars, and A. S. Tolias, “Three types of incremental learning,” *Nature Machine Intelligence*, vol. 4, no. 12, pp. 1185–1197, 2022.
- [194] L. Van der Maaten and G. Hinton, “Visualizing data using t-sne,” *Journal of machine learning research*, vol. 9, no. 11, 2008.
- [195] 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.
- [196] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, “Graph attention networks,” in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, OpenReview.net, 2018. [Online]. Available: <https://openreview.net/forum?id=rJXMpikCZ>.
- [197] 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.
- [198] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, “Prompt tuning in code intelligence: An experimental evaluation,” *IEEE Transactions on Software Engineering*, 2023.
- [199] 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.
- [200] H. Wang, W. Xiong, M. Yu, X. Guo, S. Chang, and W. Y. Wang, “Sentence embedding alignment for lifelong relation extraction,” in *Annual Conference of the North American Chapter of the Association for Computational Linguistics*, Association for Computational Linguistics (ACL), 2019.

- [201] J. Wang, F. Meng, D. Zheng, Y. Liang, Z. Li, J. Qu, and J. Zhou, “A survey on cross-lingual summarization,” *Transactions of the Association for Computational Linguistics*, vol. 10, pp. 1304–1323, 2022.
- [202] J. Wang, C. Wang, F. Luo, C. Tan, M. Qiu, F. Yang, Q. Shi, S. Huang, and M. Gao, “Towards unified prompt tuning for few-shot text classification,” in *Findings of the Association for Computational Linguistics: EMNLP 2022*, 2022, pp. 524–536.
- [203] L. Wang, X. Zhang, H. Su, and J. Zhu, “A comprehensive survey of continual learning: Theory, method and application,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024.
- [204] Y. Wang, K. Wang, F. Gao, and L. Wang, “Learning semantic program embeddings with graph interval neural network,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–27, 2020.
- [205] Y. Wang, H. Le, A. Gotmare, N. Bui, J. Li, and S. Hoi, “Codet5+: Open code large language models for code understanding and generation,” in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 1069–1088.
- [206] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, “Dynamic graph cnn for learning on point clouds,” *Acm Transactions On Graphics (tog)*, vol. 38, no. 5, pp. 1–12, 2019.
- [207] 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.
- [208] Z. Wang, Z. Zhang, S. Ebrahimi, R. Sun, H. Zhang, C.-Y. Lee, X. Ren, G. Su, V. Perot, J. Dy, et al., “Dualprompt: Complementary prompting for rehearsal-free continual learning,” in *European Conference on Computer Vision*, Springer, 2022, pp. 631–648.
- [209] Z. Wang, Z. Zhang, C.-Y. Lee, H. Zhang, R. Sun, X. Ren, G. Su, V. Perot, J. Dy, and T. Pfister, “Learning to prompt for continual learning,” in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 139–149.

- [210] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, et al., “Emergent abilities of large language models,” *arXiv preprint arXiv:2206.07682*, 2022.
- [211] M. Weyssow, X. Zhou, K. Kim, D. Lo, and H. Sahraoui, “Exploring parameter-efficient fine-tuning techniques for code generation with large language models,” *ACM Transactions on Software Engineering and Methodology*, 2023.
- [212] M. Weyssow, X. Zhou, K. Kim, D. Lo, and H. Sahraoui, “On the usage of continual learning for out-of-distribution generalization in pre-trained language models of code,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1470–1482.
- [213] F. Wilcoxon, “Individual comparisons by ranking methods,” in *Breakthroughs in Statistics: Methodology and Distribution*, Springer, 1992, pp. 196–202.
- [214] C. S. Xia, Y. Wei, and L. Zhang, “Automated program repair in the era of large pre-trained language models,” in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE ’23, Melbourne, Victoria, Australia: IEEE Press, 2023, pp. 1482–1494. [Online]. Available: <https://doi.org/10.1109/ICSE48619.2023.00129>.
- [215] 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.
- [216] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, “How powerful are graph neural networks?” In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, OpenReview.net, 2019. [Online]. Available: <https://openreview.net/forum?id=ryGs6iA5Km>.
- [217] R. Xu, J. Chen, and F. He, “Data-driven loop bound learning for termination analysis,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 499–510.
- [218] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, and J. Lu, “Commit message generation for source code changes,” in *IJCAI*, 2019.
- [219] Y. Xu and Y. Zhu, “A survey on pretrained language models for neural code intelligence,” *arXiv preprint arXiv:2212.10079*, 2022.

- [220] Z. Xu, S. Qiang, D. Song, M. Zhou, H. Wan, X. Zhao, P. Luo, and H. Zhang, “Dsfn: 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.
- [221] 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.
- [222] P. Xue, L. Wu, Z. Yang, Z. Yu, Z. Jin, G. Li, Y. Xiao, S. Liu, X. Li, H. Lin, et al., “Exploring and lifting the robustness of llm-powered automated program repair with metamorphic testing,” *arXiv preprint arXiv:2410.07516*, 2024.
- [223] 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.
- [224] Y. Xue, K. Whitecross, and B. Mirzasoleiman, “Investigating why contrastive learning benefits robustness against label noise,” in *International Conference on Machine Learning*, PMLR, 2022, pp. 24 851–24 871.
- [225] P. Yadav, Q. Sun, H. Ding, X. Li, D. Zhang, M. Tan, P. Bhatia, X. Ma, R. Nallapati, M. K. Ramanathan, et al., “Exploring continual learning for code generation models,” in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, 2023, pp. 782–792.
- [226] A. Yamada, K. Kusakari, and T. Sakabe, “A unified ordering for termination proving,” *Science of Computer Programming*, vol. 111, pp. 110–134, 2015.
- [227] S. Yang, W. Xiao, M. Zhang, S. Guo, J. Zhao, and F. Shen, “Image data augmentation for deep learning: A survey,” *arXiv preprint arXiv:2204.08610*, 2022.
- [228] 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.
- [229] Z. Yang, J. Keung, X. Yu, X. Gu, Z. Wei, X. Ma, and M. Zhang, “A multi-modal transformer-based code summarization approach for smart contracts,” in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, 2021, pp. 1–12.

- [230] Z. Yang, J. W. Keung, Z. Sun, Y. Zhao, G. Li, Z. Jin, S. Liu, and Y. Li, “Improving domain-specific neural code generation with few-shot meta-learning,” *Information and Software Technology*, vol. 166, p. 107365, 2024.
- [231] Z. Yang, J. W. Keung, X. Yu, Y. Xiao, Z. Jin, and J. Zhang, “On the significance of category prediction for code-comment synchronization,” *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 1–41, 2023.
- [232] 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.
- [233] M. Yoshimura, T. Hayashi, and Y. Maeda, “MambaPEFT: Exploring parameter-efficient fine-tuning for mamba,” in *The Thirteenth International Conference on Learning Representations*, 2025.
- [234] Y. You, T. Chen, Y. Sui, T. Chen, Z. Wang, and Y. Shen, “Graph contrastive learning with augmentations,” *Advances in neural information processing systems*, vol. 33, pp. 5812–5823, 2020.
- [235] H. Yu, X. Hu, G. Li, Y. Li, Q. Wang, and T. Xie, “Assessing and improving an evaluation dataset for detecting semantic code clones via deep learning,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 4, pp. 1–25, 2022.
- [236] 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.
- [237] W. Yuan, H. Yin, T. He, T. Chen, Q. Wang, and L. Cui, “Unified question generation with continual lifelong learning,” in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 871–881.
- [238] 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.
- [239] Z. Zeng, Y. Zhang, H. Zhang, and L. Zhang, “Deep just-in-time defect prediction: How far are we?” In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 427–438.

- [240] F. Zhang, B. Chen, Y. Zhao, and X. Peng, “Slice-based code change representation learning,” in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2023, pp. 319–330.
- [241] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, “Retrieval-based neural source code summarization,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1385–1397.
- [242] J. Zhang, Z. Liu, X. Hu, X. Xia, and S. Li, “Vulnerability detection by learning from syntax-based execution paths of code,” *IEEE Transactions on Software Engineering*, vol. 49, no. 8, pp. 4196–4212, 2023.
- [243] K. Zhang, W. Wang, H. Zhang, G. Li, and Z. Jin, “Learning to represent programs with heterogeneous graphs,” in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 378–389.
- [244] L. Zhang, X. Chen, J. Zhang, R. Dong, and K. Ma, “Contrastive deep supervision,” in *European Conference on Computer Vision*, Springer, 2022, pp. 1–19.
- [245] M. Zhang, H. Marklund, N. Dhawan, A. Gupta, S. Levine, and C. Finn, “Adaptive risk minimization: Learning to adapt to domain shift,” *Advances in Neural Information Processing Systems*, vol. 34, pp. 23 664–23 678, 2021.
- [246] N. Zhang, L. Li, X. Chen, S. Deng, Z. Bi, C. Tan, F. Huang, and H. Chen, “Differentiable prompt makes pre-trained language models better few-shot learners,” in *International Conference on Learning Representations*, 2022. [Online]. Available: <https://openreview.net/forum?id=ek9a0qIafW>.
- [247] T. Zhang, F. Wu, A. Katiyar, K. Q. Weinberger, and Y. Artzi, “Revisiting few-sample bert fine-tuning,” in *International Conference on Learning Representations*, 2021.
- [248] Y. Zhang, H. Zhu, Z. Song, P. Koniusz, and I. King, “Costa: Covariance-preserving feature augmentation for graph contrastive learning,” in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2022, pp. 2524–2534.
- [249] 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.

- [250] M. Zhu, A. Jain, K. Suresh, R. Ravindran, S. Tipirneni, and C. K. Reddy, “Xl-cost: A benchmark dataset for cross-lingual code intelligence,” *arXiv preprint arXiv:2206.08474*, 2022.
- [251] 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.
- [252] Y. Zhu, Y. Xu, F. Yu, Q. Liu, S. Wu, and L. Wang, “Graph contrastive learning with adaptive augmentation,” in *Proceedings of the Web Conference 2021*, 2021, pp. 2069–2080.