When Fine-Tuning LLMs Meets Data Privacy: An Empirical Study of Federated Learning in LLM-Based Program Repair

WENQIANG LUO, Department of Computer Science, City University of Hong Kong, China
JACKY WAI KEUNG, Department of Computer Science, City University of Hong Kong, China
BOYANG YANG, Jisuan Institute of Technology, Beijing JudaoYouda Network Technology Co. Ltd., China
HE YE, School of Computer Science, Carnegie Mellon University, USA
CLAIRE LE GOUES, School of Computer Science, Carnegie Mellon University, USA
TEGAWENDÉ F. BISSYANDÉ, SnT, University of Luxembourg, Luxembourg
HAOYE TIAN*, School of Computing and Information Systems, University of Melbourne, Australia
BACH LE, School of Computing and Information Systems, University of Melbourne, Australia

Software systems have been evolving rapidly and inevitably introducing bugs at an increasing rate, leading to significant maintenance costs. While large language models (LLMs) have demonstrated remarkable potential in enhancing software development and maintenance practices, particularly in automated program repair (APR), they rely heavily on high-quality code repositories. Most code repositories are proprietary assets that capture the diversity and nuances of real-world industry software practices, which public datasets cannot fully represent. However, obtaining such data from various industries is hindered by data privacy concerns, as companies are reluctant to share their proprietary codebases. There has also been no in-depth investigation of collaborative software development by learning from private and decentralized data while preserving data privacy for program repair.

To address the gap, we investigate federated learning as a privacy-preserving method for fine-tuning LLMs on proprietary and decentralized data to boost collaborative software development and maintenance. We use the private industrial dataset TutorCode for fine-tuning and the EvalRepair-Java benchmark for evaluation, and assess whether federated fine-tuning enhances program repair. We then further explore how code heterogeneity (i.e., variations in coding style, complexity, and embedding) and different federated learning algorithms affect bug fixing to provide practical implications for real-world software development collaboration. Our evaluation reveals that federated fine-tuning can significantly enhance program repair, achieving increases of up to 16.67% for Top@10 and 18.44% for Pass@10, even comparable to the bug-fixing capabilities of centralized learning. Moreover, the negligible impact of code heterogeneity implies that industries can effectively collaborate despite diverse data distributions. Different federated algorithms also demonstrate unique strengths across LLMs, suggesting that tailoring the optimization process to specific LLM characteristics can further improve program repair.

*Corresponding author.

Authors' addresses: Wenqiang Luo, Department of Computer Science, City University of Hong Kong, China, wenqialuo4-c@my.cityu.edu.hk; Jacky Wai Keung, Jacky.Keung@cityu.edu.hk, Department of Computer Science, City University of Hong Kong, China; Boyang Yang, Jisuan Institute of Technology, Beijing JudaoYouda Network Technology Co. Ltd., China, buaabarty@gmail.com; He Ye, School of Computer Science, Carnegie Mellon University, USA, hey@cs.cmu.edu; Claire Le Goues, School of Computer Science, Carnegie Mellon University, USA, clegoues@cs.cmu.edu; Tegawendé F. Bissyandé, SnT, University of Luxembourg, Luxembourg, tegawende.bissyande@uni.lu; Haoye Tian, School of Computing and Information Systems, University of Melbourne, Australia, tianhaoyemail@gmail.com; Bach Le, School of Computing and Information Systems, University of Melbourne, Australia, bach.le@unimelb.edu.au.

ACM 1557-7392/2025/4-ART
https://doi.org/10.1145/3733599

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Program Repair, Federated Learning, Large Language Models

1 INTRODUCTION

Learning-based automated program repair (APR) has attracted considerable attention with emerging deep learning (DL) techniques, significantly reducing the manual effort in software maintenance [126]. The continuous evolution of modern software systems inevitably leads to the emergence of software bugs at an increasing rate, which results in substantial software maintenance costs in terms of both time and financial resources [9, 10, 103]. Benefiting from the advancements of large language models (LLMs), there is a growing research interest in applying LLMs to the bug-fixing process of APR, which further improves the efficiency of automated bug repair [64, 77, 90, 92, 117]. Fine-tuning LLMs to satisfy specific downstream tasks is rather effective since they have been pre-trained with huge corpora. Recent studies have demonstrated the potential of fine-tuning LLMs for APR tasks, showing significant improvements in repair accuracy and generalization [32, 35]. Novel approaches have been explored to fine-tune LLMs for APR, including standard neural machine translation (NMT) fine-tuning [22, 36], multi-objective fine-tuning [118], incorporating additional information such as code review to enhance fine-tuning [75], augmenting fine-tuning datasets with syntactically diverse but semantically equivalent code [27], utilizing optimal code representations [78, 86] and parameter-efficient techniques are also adopted for effective and low-cost fine-tuning [86, 118, 133], each aiming to enhance program repair capabilities through LLM fine-tuning.

The access to extensive datasets has enabled LLMs to unleash their potential to learn complex patterns and nuances in different scenarios. Researchers for a long period have been mining public communities millions of code samples in the form of bug-fix pairs to enhance the repairing ability of their approaches [37, 97]. LLM-based APR is further improved by fine-tuning with carefully curated open-source data. Moreover, researchers and developers use synthetic data generated by models to enhance fine-tuning and satisfy specific needs [41, 101, 110]. By fine-tuning with a variety of data, the LLMs are capable of repairing bugs across different bug types, scenarios, and programming languages [32, 123, 127]. The fine-tuning paradigm, to a large extent, has enabled the LLMs to achieve better performance and adapt to specific downstream tasks.

However, LLM-based APR heavily relies on high-quality code repositories since fine-tuning effectiveness depends on the dataset size and quality [106, 125]. While millions of public repositories exist on open-source platforms such as GitHub [85], most remain private or proprietary assets from various industries[1][2]. Such data captures a diversity of real-world industrial software development practices that handle different unexpected requirements [66]. Therefore, simply relying on public data is insufficient to build robust and effective software systems. However, direct industrial collaboration is limited by data privacy concerns. On the other hand, with state-of-the-art LLMs such as GPT-4 [2] having a knowledge cut-off date of 2023, continuous updates of the data are required. Meanwhile, real-world deployments at Facebook and Google demonstrate the viability of integrating automated fixes into production workflows [68, 83], while specialized domains such as autonomous driving handle multi-location faults [1] and security tasks benefit from GPT-4's remarkable improvements [107]. Education scenarios also show higher repair rates in large-scale classroom settings [30], and more comprehensive evaluations of real-world bugs reveal the strengths of learning-based approaches [74]. On the other hand, while most academic research shows promising performance, transitioning these works into practical use for industries is challenging since they may underperform on unseen private industrial data [121]. Another critical problem pointed out by a group of researchers is that high-quality public data is expected to be depleted by 2026 [98]. Public data and synthetic data may no longer suffice for research and software development. Therefore, leveraging

[1]https://github.com/about

[2]https://web.archive.org/web/20240923210406/https://github.com/search?q=is%3Apublic&type=repositories

private industrial data while preserving privacy is essential to bridge the gap between academic research and practical software development.

Federated learning has emerged as a promising approach that facilitates private entities to collaborate on model training without exposing their raw data [70], benefiting industries in software development and maintenance. Advances such as differential privacy [120], secure aggregation [39], and secure multi-party computation [114] further safeguard data privacy. Federated learning has also demonstrated effectiveness in a few software engineering tasks such as code clone detection, defect prediction [121], fault localization [24], and commit classification [48, 85]. In addition, there are various recent works that focus on preserving privacy in LLM-based fine-tuning [15, 61, 99] and leverage federated learning for knowledge distillation [67], recommendation [130], legal intelligence [124], and security [131]. Efficient fine-tuning strategies have also been proposed to mitigate the huge cost of LLM parameters [14, 87].

This work. To address the gap in collaborative and decentralized software development, we conduct a comprehensive empirical study on fine-tuning LLMs for program repair using federated learning. Our study addresses three research questions: RQ1 examines the effectiveness of federated fine-tuning for program repair. RQ2 investigates the impact of heterogeneous code in federated learning. RQ3 explores how different federated algorithms influence LLM fine-tuning for program repair. We fine-tune selected LLMs using a private industrial dataset consisting of 1,239 real-world programs and evaluate them on a benchmark with up to 583 test cases per problem on average to mitigate patch overfitting. Specifically, we conduct a series of experiments and analyses for each RQ as follows:

(1) Firstly, we compare federated fine-tuning against standard fine-tuning methods including centralized learning, which is the ideal scenario that pools all data together, and the original LLMs across various model architectures to validate whether LLM-based federated fine-tuning is able to enhance program repair. We find that federated fine-tuning can substantially enhance the program repair capabilities of LLMs to achieve performance even comparable to the centralized fine-tuning approach, achieving maximal increases of up to 16.57% on $Top@10$ and 18.44% on $Pass@10$.

(2) Secondly, we construct different degrees of heterogeneous code based on various code features such as coding style, code complexity and code embedding to better align with real-world scenarios. We evaluate the performance of LLMs fine-tuned with different data distributions to investigate how data heterogeneity influences program repair. Notably, the results indicate that heterogeneous code has no significant impact on the fine-tuning of LLMs compared to the ideal homogeneous data distribution. On the contrary, it leads to a considerable improvement that achieves maximal increases of 18.41% and 21.46% on $Top@10$ and $Pass@10$ for program repair.

(3) Finally, we evaluate different federated algorithms optimized for different stages including client-side and server-side optimization. We also evaluate the personalized learning paradigm to assess its effectiveness in adapting LLMs to specific clients. Our analysis reveals that while different types of federated algorithms exhibit varying strengths and weaknesses across different LLMs, FedAvg demonstrates the best overall performance and the personalized learning approach remains a challenge for LLM fine-tuning.

Contributions. In summary, our study makes the following main contributions:

- We present an empirical study on fine-tuning LLMs for program repair with federated learning. Our study explores the feasibility and effectiveness of fine-tuning LLMs while **preserving data privacy** in decentralized and collaborative software development. Our results indicate that federated fine-tuning significantly enhances program repair capabilities and achieves performance comparable to the regular centralized learning approach.
- We explore the influence of federated fine-tuning on program repair where the objective of such task is to generate correctly fixed patches, in contrast to the majority of prior federated learning research that concentrates on typical discriminative tasks, unveiling novel insights into the application of federated learning to other forms of generative tasks. It is worth noting that our results on program repair reveal the

fact that the generative tasks may differ significantly compared to other types of tasks such as classification, regression, etc., in the context of decentralized LLM fine-tuning.
- We investigate the impact of different LLMs on the performance of federated program repair. By comparing various state-of-the-art LLMs of code that have varying architectures and pre-training strategies on multiple baselines, we provide insights into the suitability and practicality of different LLMs in federated learning.
- We explore the effect of heterogeneous code on the program repair capability of LLMs in federated fine-tuning. We construct feature-skewed Non-IID (see Section 2.2) data distribution to align with real-world scenarios where the collaborators have different code repositories with diverse software development practices and bug-fixing patterns. The results demonstrate that fine-tuning with different degrees of heterogeneous code causes negligible impact on the performance of LLMs and can still benefit program repair significantly. By evaluating the LLMs with different data distributions, we shed light on the robustness and adaptability of LLMs in handling Non-IID data in decentralized environments.
- We evaluate the impact of various federated learning algorithms that are optimized for different stages of federated learning on the bug fixing capability of LLM-based program repair. The results show that different optimizations exhibit varying strengths and weaknesses across different LLMs. Our analysis provides insights into the trade-offs and considerations when selecting federated learning algorithms for LLM-based program repair.

Availability. The artifact of this study is publicly available at: **https://github.com/stringing/Federated-LLM-Based-APR**.

The remainder of this paper is organized as follows: Section 2 describes the background knowledge of this study. Section 3 provides an overview of related work. Section 4 presents the study design, including the research questions and the overall methodology. Section 5 details the experiments conducted and analyze the results obtained. Section 6 discusses additional considerations and aspects of this study. Potential threats to validity are identified in Section 7. Finally, Section 8 concludes the paper by summarizing the main contributions and insights of this paper.

2 BACKGROUND

2.1 Parameter-Efficient Federated Fine-tuning

Federated learning is a distributed machine learning paradigm that enables multiple participants to collaboratively train a model without sharing their private data, thereby preserving privacy. Different from traditional distributed learning, where data collected from multiple sources are transferred to a server, and all data are placed together to train a centralized model, each device in federated learning trains a single model on its local dataset, and a central server is responsible for aggregating all models into a final global model.

Figure 1 presents the workflow of the parameter-efficient federated fine-tuning process. All clients are dispatched with the same randomly initialized model at first, and each participant then starts to fine-tune its model with its local dataset. Typically, the fine-tuned models are uploaded to the central server for further processing. However, the communication cost can be extremely high for LLMs since billions of parameters need to be transmitted. Therefore, benefit from lightweight fine-tuning techniques such as Low-Rank Adaption (LoRA) [29] and Quantized Low-Rank Adaption (QLoRA) [19], only the adapter that is a small fraction of parameters is supposed to be uploaded while the original pre-trained model weights are retained in the client's device. Once the central server has received all models from the clients, these models are aggregated into one global model, which is then dispatched to all clients for the next round. The global model attained in the final round is the final model that can be applied to downstream tasks. Note that the central server in federated learning acts as a

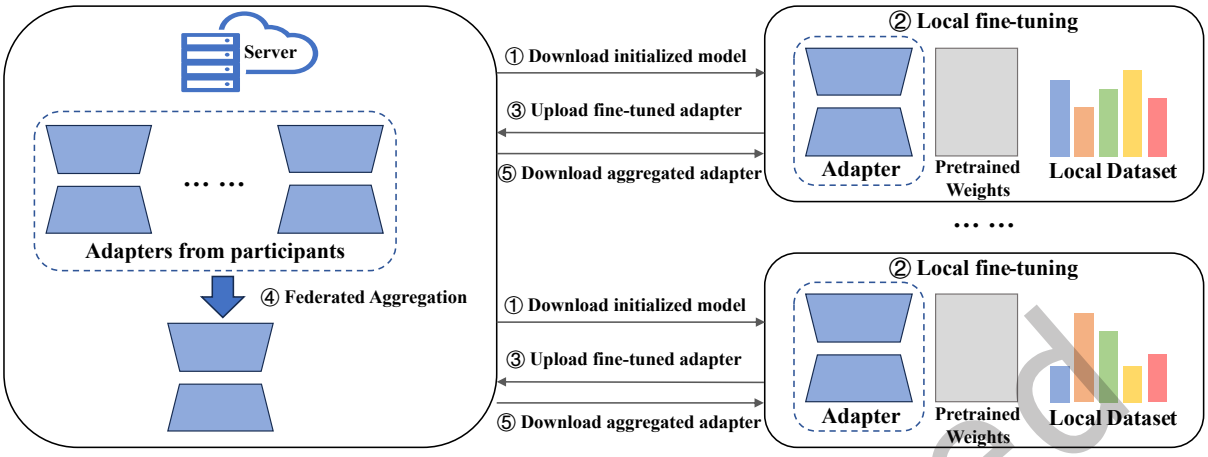


Fig. 1. The workflow of federated learning of parameter-efficient federated fine-tuning framework.

trusted third-party server. Throughout the entire federated learning process, all datasets are kept local and not shared with any other clients, thereby ensuring data privacy is protected.

In our study, we utilize the efficient fine-tuning approach QLoRA, which is an extended version of LoRA in order to alleviate resource overhead in terms of GPU memory and parameter transfer in federated learning. The work of QLoRA proposed a novel 4-bit normal float (NF4) data type to quantize the pre-trained weights and reduce additional storage cost of quantization constants by double quantization [19].

In detail, the workflow mainly consists of the following steps:

(1) Model Initialization: Initially, the pre-trained model weights $\boldsymbol{W_0} \in \mathbb{R}^{d_1 \times d_2}$ on each client are quantized to 4-bit precision using the NF4 quantization format:

$$\tilde{\boldsymbol{W}}_{\boldsymbol{0}} = Q_{NF4}(\boldsymbol{W_0}) \tag{1}$$

where Q_{NF4} denotes the quantization function and $\tilde{\boldsymbol{W}}_{\boldsymbol{0}}$ represents the quantized pre-trained model weights. The quantized weights $\tilde{\boldsymbol{W}}_{\boldsymbol{0}}$ remain frozen during fine-tuning while the trainable adapter $\boldsymbol{\Delta W} = \boldsymbol{BA}$, where $\boldsymbol{B} \in \mathbb{R}^{d_1 \times r}$ and $\boldsymbol{A} \in \mathbb{R}^{r \times d_2}$ are the low-rank decomposed matrices, is introduced into the model architecture. $r \ll min(d_1, d_2)$ is the rank of adaption and also a hyperparameter in our setting. In order to preserve model consistency in federated fine-tuning, the random Gaussian initialization for $\boldsymbol{A}$ is performed on the server and then it is downloaded by all clients as the initial parameters at the beginning of fine-tuning, except for B whose initial weights are zero.

(2) Local Fine-tuning: Consider a federated learning system composed of K clients, each holding a local fine-tuning dataset $\mathcal{D}_k$, where $k \in 1, 2, ..., K$. The adapted model $\boldsymbol{W_k}$ at client k is:

$$\boldsymbol{W_k} = \tilde{\boldsymbol{W}}_{\boldsymbol{0}} + \Delta \boldsymbol{W_k} = \tilde{\boldsymbol{W}}_{\boldsymbol{0}} + \boldsymbol{B_k A_k} \tag{2}$$

The fine-tuning objective is to minimize the local loss function $\mathcal{L}_k$ with respect to the adapter parameters $\boldsymbol{B_k}$ and $\boldsymbol{A_k}$:

$$\mathcal{L}_k \left(\tilde{\boldsymbol{W}}_{\boldsymbol{0}} + \boldsymbol{B_k A_k}; \mathcal{D}_k \right) = \frac{1}{n_k} \sum_{i=1}^{n_k} \ell \left(f \left(\tilde{\boldsymbol{W}}_{\boldsymbol{0}} + \boldsymbol{B_k A_k}; \boldsymbol{x_i^k} \right), \boldsymbol{y_i^k} \right) \tag{3}$$

where $(\boldsymbol{x_i^k}, \boldsymbol{y_i^k}) \in \mathcal{D}_k$ is one of the bug-fix code pairs in the fine-tuning dataset $\mathcal{D}_k$ and n_k is the size of $\mathcal{D}_k$. f is the output function of the model and we employ a cross-entropy loss function ℓ to measure the difference

between the probability distributions of the model output and actual code for each token in the sequences as follows:

$$\ell(\hat{\boldsymbol{y}}_{\boldsymbol{i}}^{\boldsymbol{k}}, \boldsymbol{y}_{\boldsymbol{i}}^{\boldsymbol{k}}; \boldsymbol{x}_{\boldsymbol{i}}^{\boldsymbol{k}}) = -\sum_{j} Q\left(\hat{y}_{i,j}^{k} \middle| \boldsymbol{x}_{\boldsymbol{i}}^{\boldsymbol{k}}, \hat{y}_{i,1}^{k}, ..., \hat{y}_{i,j-1}^{k}\right) \times \log P\left(y_{i,j}^{k} \middle| \boldsymbol{x}_{\boldsymbol{i}}^{\boldsymbol{k}}, y_{i,1}^{k}, ..., y_{i,j-1}^{k}\right) \tag{4}$$

where Q is the probability distribution for the predicted output $\hat{\boldsymbol{y}}_{\boldsymbol{i}}^{\boldsymbol{k}}$ and P is the actual distribution for $\boldsymbol{y}_{\boldsymbol{i}}^{\boldsymbol{k}}$. $\hat{y}_{i,j}^{k} \in \hat{\boldsymbol{y}}_{\boldsymbol{i}}^{\boldsymbol{k}}$ and $y_{i,j}^{k} \in \boldsymbol{y}_{\boldsymbol{i}}^{\boldsymbol{k}}$ are one of the tokens in the predicted output and actual code respectively. Then the optimized low-rank adapter parameters can be obtained by using various optimizers such as SGD:

$$\Delta \boldsymbol{W}_{\boldsymbol{k}}^{*} = \boldsymbol{B}_{\boldsymbol{k}}^{*} \boldsymbol{A}_{\boldsymbol{k}}^{*} = \Delta \boldsymbol{W}_{\boldsymbol{k}} - \eta \Delta \mathcal{L}_{k}\left(\tilde{\boldsymbol{W}}_{\boldsymbol{0}} + \boldsymbol{B}_{\boldsymbol{k}} \boldsymbol{A}_{\boldsymbol{k}}; \mathcal{D}_{k}\right) \tag{5}$$

where η is the learning rate. Note that in this framework the paged optimizers are utilized according to the setting of QLoRA, which are able to automatically transfer the information from GPU to CPU for error-free GPU processing when GPU encounters out-of-memory issues [19].

(3) Fine-tuned Model Upload: Instead of uploading the entire fine-tuned LLMs, clients upload only the fine-tuned adapters $\boldsymbol{B}_{\boldsymbol{k}}^{*} \boldsymbol{A}_{\boldsymbol{k}}^{*}$ to the central server. This significantly reduces the cost of transferring model parameters since the adapter constitutes only a small portion of the original model parameters.

(4) Federated Aggregation: In this step, the central server aggregates the adapter parameters received from all clients into a global adapter $\Delta \boldsymbol{W}_{G}$ using aggregation algorithms such as federated averaging (FedAvg) [70]:

$$\Delta \boldsymbol{W}_{\boldsymbol{g}} = \boldsymbol{B}_{\boldsymbol{g}} \boldsymbol{A}_{\boldsymbol{g}} = \frac{1}{n} \sum_{k=1}^{K} n_{k} \boldsymbol{B}_{\boldsymbol{k}}^{*} \boldsymbol{A}_{\boldsymbol{k}}^{*} \tag{6}$$

where $n = \sum_{k=1}^{K} n_{k}$.

(5) Aggregated Model Download: The aggregated adapter $\Delta \boldsymbol{W}_{\boldsymbol{g}}$ is dispatched back to all clients. Each client updates the local adapter with the downloaded global adapter:

$$\Delta \boldsymbol{W}_{\boldsymbol{k}}^{\prime} = \tilde{\boldsymbol{W}}_{\boldsymbol{0}} + \Delta \boldsymbol{W}_{\boldsymbol{g}} \tag{7}$$

The updated model $W_{k}^{\prime}$ serves as the starting point for the next round of federated fine-tuning. The global model obtained after a total of T rounds of fine-tuning is the final model, which can be further applied to corresponding downstream tasks.

Throughout the entire federated fine-tuning process, parameter efficiency is achieved by quantization, significantly reducing storage costs for each client and the overhead of parameter transfer in federated learning system. Data privacy is also protected by not exposing local data to each participating client. In this study, we investigate whether effective fine-tuning can be promoted by this framework to achieve comparable performance while protecting data privacy at the same time.

2.2 Data Heterogeneity in Federated Learning

Data heterogeneity has been a critical challenge that incurs performance degradation in federated learning [40]. In ideal situations where the data across all clients are independently and identically distributed (IID), federated learning can achieve comparable performance without centralizing the data [70]. However, real-world data distributions can be affected by various reasons. The datasets collected on each device tend to vary in their distributions due to a number of factors such as geographical differences, demographic differences and different user behaviors, which exhibit non-independently and identically distributed (Non-IID) characteristics. Therefore, each local model trains towards different objectives inconsistent with the global optimum and eventually deviate from the optimal solution under Non-IID scenarios [51]. According to the works of Kairouz et al. [40], there are mainly three basic types of Non-IID data. Consider the local data distribution $P(\boldsymbol{x}_{\boldsymbol{i}}^{\boldsymbol{k}}, \boldsymbol{y}_{\boldsymbol{i}}^{\boldsymbol{k}})$ at client k, the three fundamental Non-IID scenarios are as follows:

(1) Label Skew. Label skew arises when the distribution of the target labels differs across devices, i.e., the marginal distribution $P(\boldsymbol{y_i^k})$ varies in each client. Label skew happens when certain labels are more prevalent on some devices than others, leading to an imbalance in the label distribution. For example, rice is commonly grown in Southeast Asia, whereas in drier areas like North America, such type of crop is mainly wheat. Generally, such imbalances can significantly affect the performance of DL models in federated learning, which may become biased towards the more prevalent classes. However, different from typical supervised tasks where the data is accompanied with fixed and explicit labels, we do not focus on label skew in this study for program repair task where the objective is to generate a valid patch that correctly fixes a bug.
(2) Feature Skew. Feature skew occurs when the distribution of the input features varies across different devices, i.e., $P(\boldsymbol{x_i^k})$ differs across clients. For example, the same handwritten number from different people can present in different ways because of diverse writing habits. In the context of software engineering, feature skew can manifest in various ways due to the diverse nature of programming languages, frameworks and development practices. For instance, consider a scenario where client A primarily develops applications using Java and the Spring framework, while Client B focuses on Python and the Django framework. The code features, such as syntax, libraries and design patterns, will exhibit significant differences and variations in the feature representation between these two clients. Consequently, the local models trained on these distinct feature distributions may struggle to generalize effectively to the global model, which can lead to suboptimal performance in federated learning. We construct and explore various code features in this study to investigate the impact of heterogeneous code on federated learning.
(3) Quantity Skew. Quantity skew (i.e., n_k varies across clients) exists when different clients hold different amounts of data. In real-world scenarios, companies of varying sizes or from different industries often possess diverse quantities of data due to differences in business scale, data collection capabilities, etc. This disparity in data quantity can lead to an imbalance in the contributions of participating devices during federated learning, potentially introducing a bias in the global model. On the other hand, label-skewed or feature-skewed data is often inherently accompanied with quantity skew. Quantity skew typically does not exist in isolation for most scenarios, but coexists with the other types of skews instead. Thus, we mainly focus on feature skew to investigate the variations in code rather than quantity in our study.

Overall, each type of skew is able to cause performance loss in federated learning since each model trained on different distribution has its own bias against data, which leads to instability and inconsistency in the global model aggregated from these individual models. Furthermore, whether the final model aggregated from models fine-tuned with diverse data distributions is negatively affected and to what extent it is affected still remains unclear in terms of LLM fine-tuning, especially with the generative tasks that mainly involve feature skew.

Therefore, as program repair, which aims to generate correctly repaired code patches, lacks fixed and explicit labels and present diverse features in the code, we focus on feature skew in our study.

2.3 Problem Setting and Challenges in Federated Learning for Program Repair

To illustrate the challenges of program repair in federated learning, consider two IT companies focusing on distinct domains. Assume company A develops IoT firmware and Company B is a web application provider. Company A frequently encounters memory leaks in its resource-constrained embedded systems due to manual memory management. In contrast, Company B rarely faces memory issues in its web applications, where garbage collection automates memory processing. However, occasional unreleased resources may cause slight memory leaks. While Company A's memory management expertise could theoretically inform Company B's edge-case fixes, and vice versa, collaboration between real-world companies faces substantial barriers in practice. As illustrated in Figure 2, each company could learn from the codebase of the other to address their respective deficiencies. However, direct collaboration is infeasible due to data privacy concerns in which the raw data cannot

be exposed to each other. Even if privacy barriers were overcome, whether the code heterogeneity such as the differences in coding styles and code complexity between their codebases affects collaborative learning remains unclear, which underscores the challenges of privacy preservation and heterogeneous knowledge learning in federated program repair.

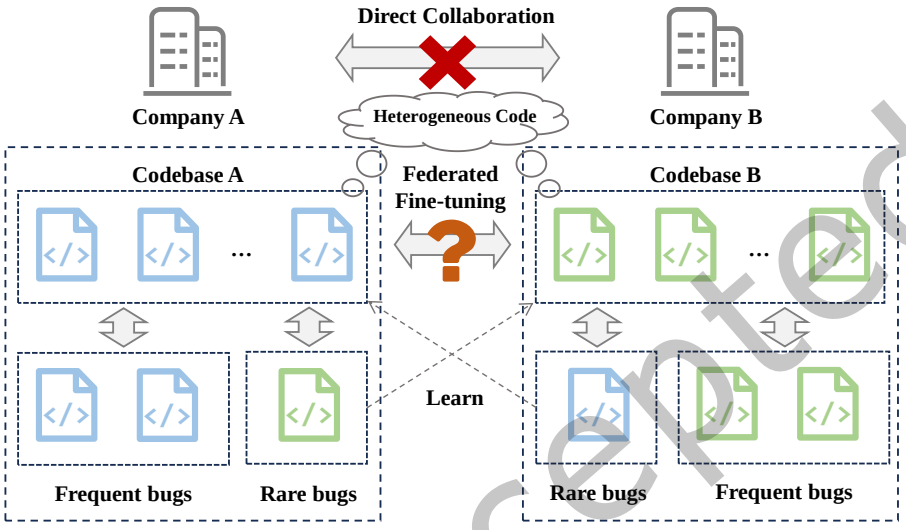


Fig. 2. An illustrative example of the challenges in federated learning for program repair.

Figure 3 presents an example of program repair across two different institutions with code heterogeneity. Specifically, the bug from company A can be fixed with a notably small cost by modifying only two hunks of code, where the original logic mishandles digit carry propagation in a big integer addition. The repair only adjusts loop boundaries and replaces incorrect arithmetic operations, addressing a subtle algorithmic problem. In contrast, Company B's bug involves a graph traversal flaw in a memoized deep first search, where the repair transitions from a global accumulator (i.e., ans) with void returns to a memoization-based approach (i.e., vis[x]), fundamentally altering control flow and state management. Beyond functional differences, the examples also diverge in coding style since company A's code relies on low-level I/O redirection (i.e., freopen) and array indexing, while Company B's repair introduces recursion with mixed imperative and stateful patterns. Such disparities in code structure, problem domains (i.e., numeric vs. graph), and error types (logical vs. architectural) underscore the challenges for federated learning. A global model must generalize the repair patterns across different clients with Non-IID codebases, varying in variable naming, loop idioms, and error patterns, while preserving local data privacy. Therefore, it is challenging for federated repair systems to balance adaptability to heterogeneous coding conventions with robustness to diverse bug patterns.

3 RELATED WORK

3.1 Federated Learning for Software Engineering

Existing Works. Despite the growing interest in federated learning, its application to software engineering tasks remains relatively unexplored, with only a limited number of studies addressing this area. Yang et al.

Bug from Company A

```
#include <cstdio>
#include <iostream>
using namespace std;

int main() {
    freopen("bigadd.in","r",stdin);
    freopen("bigadd.out","w",stdout);
    string s;
    cin >> s;
    int a[10005];
    for(int i = s.size()-1;i >= 0;i--) a[i] = s[i] - '0';
    a[s.size()-1] += 2018;
-   for(int i = 0;i < s.size();i++) {
+   for(int i = 2; i <= s.size(); i++) {
        a[s.size()-i] += a[s.size()-i+1] / 10;
-       a[s.size()-i+1] /= 10;
+       a[s.size()-i+1] %= 10;
        if(a[s.size()-i] < 10) break;
    }
    for(int i = 0;i < s.size();i++) cout << a[i];
    return 0;
}
```

Bug from Company B

```
.....
bool vis[10100];
- long long ans=0;
- void dfs(int u) {
-   if(vis[u]) return;
-   vis[u] = 1;

-   for(int i=0; i<scc[u].size(); i++) {
-     ans += a[scc[u][i]];
-   }

-   for(int i=new_head[u]; i; i=new_edge[i].next) {
-     dfs(new_edge[i].to);
-   }
- }
+ int dfs(int x) {
+   if(vis[x]) return vis[x];

+   int sum=0, temp=0;
+   for(int i=0; i<scc[x].size(); i++) {
+     sum += a[scc[x][i]];
+   }

+   for(int i=new_head[x]; i; i=new_edge[i].next) {
+     temp = max(temp, dfs(new_edge[i].to));
+   }

+   vis[x] = temp + sum;
+   return vis[x];
+ }
int main() {
    for(int i=1; i<=n; i++) {
        for(int j=1; j<=m; j++) {
-           a[i*m+j] = getchar()-'0';
-           if(a[i*m+j]=='*'-'0') {
-             a[i*m+j] = 0;
-             chuansong[pos++] = i*m+j;
-           }
-           else if(a[i*m+j]==' '-'0') a[i*m+j]=0;
-           else a[i*m+j]=-1;
+ a[i*m+j-m] = (getchar()-'0');
+         if(a[i*m+j-m]<=9 && a[i*m+j-m]>=1) {
+           continue;
+         }
+         else if(a[i*m+j-m]=='*'-'0') {
+           a[i*m+j-m] = 0;
+           chuansong[pos++] = i*m-m+j;
+         }
+         else a[i*m-m+j] = -1;
    }
    getchar();
  }
- for(int i=1; i<n; i++) {
-   for(int j=1; j<m; j++) {
-     if(a[i*m+j]!='#' && a[i*m+j+1]!='#') {
-       addEdge(i*m+j, i*m+j+1);
-     }
-     if(a[i*m+j]!='#' && a[i*m+j+m]!='#') {
-       addEdge(i*m+j, i*m+j+m);
-     }
-   }
- }
+ for(int i=1; i<=n; i++) {
+   for(int j=1; j<=m; j++) {
+     if(a[i*m+j-m]!=-1 && a[i*m-m+j+1]!=-1 && j+1<=m) {
+       addEdge(i*m-m+j, i*m-m+j+1);
+     }
+     if(a[i*m+j-m]!=-1 && a[i*m+j]!=-1 && i+1<=n) {
+       addEdge(i*m+j-m, i*m+j);
+     }
+   }
+ }
}
.....
```

Fig. 3. A program repair case in federated learning.

[121] proposed a novel federated learning framework ALMITY to enhance the models trained on private and heterogeneous datasets for code clone detection and defect prediction. ALMITY provided an approach to tackle data imbalance and improve minority class learnability. Gill et al. [24] integrated systematic debugging with federated learning to locate problematic clients with a novel replay strategy. In order to explore the value of non-open source projects for federated learning on software engineering, Shanbhag et al. [85] conducted a study on commit classification to examine whether federated learning achieved comparable performance to centralized learning. Kumar et al. [48] evaluated federated learning on code summarization and found that federated learning is as effective as centralized learning.

Limitations. Existing studies suffer from several limitations. Most of them lack further investigation into LLMs with only a limited range of architectures or sizes, also failing to explore the potential of state-of-the-art models. Additionally, there is a lack of exploration into data heterogeneity, which limits understanding of the impact of diverse data during software development. In the meanwhile, the focus has predominantly been on tasks with labeled data, neglecting the crucial challenges posed by generation tasks with feature-skewed data, which is prevalent in real-world applications. Lastly, the reliance on simulated data derived from public datasets, rather than incorporating real-world private industrial data, limits the generalizability and practical relevance of the findings.

Addressing the Gap. In this study, we evaluate the efficacy of federated learning for program repair using multiple state-of-the-art LLMs to investigate their fixing ability. We evaluate the impact of data heterogeneity by

employing a private industrial dataset to construct Non-IID scenarios. As program repair is a code generation task without fixed and explicit labels, we exploit the rich feature information embedded within the code. Consequently, we extract diverse code features to conduct a thorough analysis of the influence of heterogeneous code on federated learning.

3.2 Federated Learning with LLMs for Privacy Preservation

Existing Works. Since federated learning enables decentralized fine-tuning of LLMs without exposing raw data, it opens up possibilities for collaborative learning while maintaining data privacy. The use of private industrial data for LLM fine-tuning while preserving data privacy is a critical challenge. Considering the limited availability of public domain data and the need to protect private data, Chen et al. [15] conducted an analysis between different components of federated learning with LLMs, which include federated pre-training, federated fine-tuning and federated prompt engineering, to explore solutions to potential issues that might occur when applying federated learning to LLMs. In order to further promote the privacy-preserving capabilities of federated learning, Wang et al. [99] selected public data that has similar distribution to private data for LLM training and used LLMs as the teaching models in knowledge distillation to facilitate the training of private models on each device. Liu et al. [61] specifically designed a privacy-preserving approach for federated learning by employing Gaussian noise to the updates of LLMs.

Furthermore, some researchers apply federated learning with LLMs to various tasks since centralized training of LLMs fails to tackle privacy concerns. Zhao et al. [130] performed federated fine-tuning on users' behavior data for federated recommendation based on LLMs. Yue et al. [124] utilized a continual learning approach with federated learning for legal tasks on an LLM of size 7B. Zheng et al. [131] studied the attack performance of LLMs with federated learning. Given that there were few works on federated distillation with large-scale models, Ma et al. [67] conducted federated distillation with LLMs for both homogeneous and heterogeneous settings.

On the other hand, one of the major challenges of federated learning with LLMs is the huge number of parameters each update of an LLM consists of. To mitigate this issue, Che et al. [14] designed an approach that selected important layers of prompts for efficient prompt-tuning. Sun et al. [87] proposed an effective federated learning framework that enables prompt exchanging between servers and clients rather than parameters to improve the performance of LLMs. However, the performance of prompt-tuning can be suboptimal since the prompts require elaborate design with prior knowledge [23].

Limitations. Despite recent advancements on LLMs, there remains a significant research gap regarding the application of federated learning with LLMs to software engineering tasks, especially those pertaining to code. Given the inherent value of code repositories to enterprises, ensuring data privacy becomes a crucial concern. Current research does not sufficiently address how federated learning can be effectively utilized for code-related tasks while ensuring data privacy.

Addressing the Gap. Our focus lies on program repair, which is a crucial code-related task for software maintenance, by leveraging federated learning with LLMs. We aim to preserve the privacy of sensitive code repositories while effectively enhancing program repair capabilities.

3.3 Fine-tuning LLMs for Program Repair

Existing Works. Pre-trained on extensive corpora, LLMs can already achieve remarkable performance without additional training. For instance, the AlphaRepair framework, introduced by Xia et al. [112], employs a cloze-style technique to generate code patches leveraging the original CodeBERT [21] without fine-tuning, presenting superior performance compared to many state-of-the-art APR techniques. Furthermore, Xia et al. [111] performed an extensive investigation into the direct application of various LLM configurations for APR. Their findings suggested that LLMs offer considerable potential for enhancing APR effectiveness.

Given that the benefits of LLMs remain underexplored, there is a growing trend toward further enhancing LLMs with fine-tuning techniques. To validate the effectiveness of fine-tuning LLMs for APR, Jiang et al. [35] conducted the first study that specifically evaluated code LLMs on multiple APR benchmarks. This study found that fine-tuned LLMs were able to significantly outperform existing DL-based APR techniques, demonstrating great potential in LLM-based APR. To further evaluate the fine-tuned LLMs, Huang et al. [32] considered multiple LLM architectures, programming languages for a comprehensive investigation covering bugs, vulnerabilities and errors on 7 benchmarks, revealing that fine-tuned LLMs can significantly surpass the performance of previous state-of-the-art APR tools.

Moreover, building upon the promising results of fine-tuning LLMs for program repair, researchers have proposed various techniques and approaches to further enhance the performance of LLM-based APR for specific needs. Mashhadi et al. [69] made a preliminary attempt in fine-tuning CodeBERT [21] to fix simple bugs in Java. Silva et al. [86] fine-tuned LLMs with various code representations to improve the fine-tuning performance, highlighting the importance of domain-specific and expert code representations for APR. On the other hand, integrating the power of LLMs with existing tools can be another promising direction for enhancing LLM-based APR performance. Jin et al. [37] proposed a novel APR approach, InferFix, which utilized a static analyzer to detect bugs and fine-tuned an LLM to produce appropriate patches for the bugs. Wei et al. [102] combined a completion engine with the fine-tuned LLMs to mitigate the limitations of LLMs in generating syntactically and semantically correct codes in order to generate more valid fixes.

With further exploration, another remarkable capability of LLMs in APR has exhibited in cross-language repairing. The study conducted by Ahmed et al. [3] demonstrated that LLMs had substantial capacity to achieve multi-lingual program repair. This study revealed that LLMs fine-tuned across multiple programming languages are more likely to improve the repairing performance as opposed to a single programming language. Yuan et al. [123] proposed an APR framework CIRCLE to achieve multi-lingual program repair and address the need of increasing software requirements. CIRCLE fine-tuned LLMs in a continual learning manner to learn from multiple programming languages with a replay approach that mitigated the risk of forgetting previous languages to enable accurate and stable multi-lingual program repair.

Limitations. Current explorations of LLM-based program repair techniques have primarily focused on centralized environments, and program repair techniques in distributed environments require further investigation. Developers often work together on software development, especially for large-scale projects, collaborating across different locations and time zones. This collaborative nature of software development presents unique challenges and opportunities for decentralized program repair techniques.

Addressing the Gap. We focus on LLM-based program repair with a decentralized setting to explore the potential of collaborative learning while preserving data privacy. We simulate a number of distributed devices and datasets in the federated learning system to investigate the practicality of federated fine-tuning and provide insights that may benefit further research for decentralized program repair or other code-related tasks.

4 STUDY DESIGN

4.1 Research Questions

RQ1: Can federated learning enhance LLM fine-tuning in repairing programs while protecting data privacy? While federated learning preserves data privacy of each client, whether it can improve and to what extent it can enhance the bug fixing capability of the LLMs still remains underexplored for most code-related tasks. As a preliminary RQ, we want to understand the feasibility and effectiveness of federated fine-tuning. Therefore, we compare federated fine-tuning with other standard fine-tuning approaches across different LLMs to examine its potential to enhance program repair.

RQ2: How do data distributions in federated learning environments affect the repairing capabilities of LLMs? Software development practices vary significantly across real-world industries. The code repositories held by different organizations, such as software companies, can exhibit heterogeneity in code features such as coding style, code complexity, etc., whereas data heterogeneity has hindered further improvement for most traditional federated learning tasks. Therefore, in this RQ, we aim to explore the influence of heterogeneous code by constructing different degrees of Non-IID scenarios to fine-tune and evaluate the LLMs.
RQ3: How do different federated learning algorithms perform in fine-tuning LLMs for program repair? As key components of federated learning, the optimization strategies in different phases, including local training and federated aggregation, are crucial to enhance the federated learning process and adapt to various federated learning scenarios. This RQ aims to investigate how federated algorithms with different types of optimization influence the performance of federated fine-tuning. We evaluate federated algorithms covering client-side optimization, server-side optimization, and personalized learning to validate the impact of algorithms on program repair.

4.2 Fine-tuning Dataset and Evaluation Benchmark

Selection Criteria. To effectively select the fine-tuning dataset and evaluation benchmark, we take several critical criteria into consideration:

- Firstly, the overlap of pre-training data and evaluation benchmark can cause data leakage [32], which hinders the evaluation to reflect the real capability of the models since they tend to achieve excellent performance on data they have been trained in advance [94, 108]. On the other hand, patch overfitting [49, 122] is another major concern for the evaluation benchmark due to the limitations of the test cases. The generated patches can be incorrect even though they have passed the test cases in the benchmark if deficient test suites fail to validate key intentions in the code [58, 100]. Therefore, preventing data leakage and minimizing patch overfitting is paramount.
- Secondly, the use of prevalent programming languages ensures compatibility and ease of integration with tools commonly employed in both academia and industry. To that end, according to the findings by Zhang et al. [128] and the trends in the use of programming languages on GitHub[3], the programming languages widely adopted in current communities include Python, Java, C++, C and JavaScript. Therefore, we aim to choose datasets with programming languages that fall within this range.
- Thirdly, employing private industrial datasets in terms of LLM fine-tuning provides a practical source of real-world data, enhancing the model's applicability and robustness. Additionally, high-quality test cases are essential for the evaluation benchmark, offering a comprehensive assessment of model performance across diverse scenarios.
- Furthermore, providing detailed metadata with the datasets significantly improves the fine-tuning process by offering valuable information for data preprocessing and the models to fine-tune with.

Consequently, we select datasets for fine-tuning and evaluation based on these criteria collectively to facilitate effective model development and accurate evaluation.

- **Fine-tuning Dataset.** In order to investigate the performance of federated fine-tuning with LLMs using private industrial data in real-world scenarios, we employ a private industrial dataset TutorCode [116] as the fine-tuning dataset in our study to satisfy real-world conditions to validate the effectiveness of federated fine-tuning. TutorCode is a curated collection of programming submissions sourced from the online programming education platform by 20 experts, ensuring integrity and quality by manual verification. To prevent data leakage, the dataset is not crawled from public repositories but collected from the company's

[3]https://madnight.github.io/githut/#/pull_requests/2024/1

proprietary data. The usage license further safeguards its integrity and confidentiality by preventing potential unauthorized crawling. The TutorCode dataset comprises 1,239 buggy C++ programs developed by 427 programmers, addressing 35 distinct programming problems. The dataset covers a wide range of algorithms, which cater to both novice and advanced levels, resulting in 12 tiers of problem difficulty. It includes a variety of bugs spanning multiple functions, while the corresponding fixes also cover from one to more than ten modified code hunks. Furthermore, each buggy code is accompanied with extra metadata, such as problem description, to provide helpful information to improve the repairing process. Since buggy code originates from various programmers with different coding habits and programming skills, rich information lies within the code itself. By mining different code features, we can construct datasets of distinct distributions to further explore the impact of heterogeneous code on federated fine-tuning.

– **Evaluation Benchmark.** Despite that existing benchmarks such as Defects4J [38] and ManyBugs [50] have been widely used to evaluate the capability of bug fixing, they still exhibit certain limitations, including data leakage [128] and patch overfitting [126] issues. Therefore, we mainly aim to select high-quality evaluation benchmark that satisfy the above criteria and mitigate potential risks as much as possible. We adopt EvalRepair-Java [118] as the evaluation benchmark on federated fine-tuning. EvalRepair-Java consists of 163 unique bugs, each corresponding to a distinct programming problem. EvalRepair-Java is built on top of the foundation of HumanEval-Java [35]. Different from the Java section of HumanEval-X [132], HumanEval-Java was specifically designed to avoid data leakage by excluding itself from the pre-training datasets of existing LLMs up to the cut-off date of 2023. Moreover, HumanEval-Java[4] was released on the open-source platform GitHub later than HumanEval-X[5], thereby reducing the potential risk of data leakage. EvalRepair-Java provides enhanced test cases that more accurately reflect the performance of the fine-tuned LLMs. By integrating additional test cases from EvalPlus [60], EvalRepair-Java significantly increases the average number of test cases per problem from 7 to 583. This substantial expansion of test cases plays a vital role in mitigating the risk of patch overfitting caused by a limited number of test cases during the evaluation process.

While our fine-tuning dataset contains C++ programs, we use EvalRepair-Java as the evaluation benchmark rather than EvalRepair-C++ [118] to ensure rigorous evaluation that prioritizes leakage avoidance. EvalRepair-Java was built upon the new HumanEval-Java benchmark as we clarified previously. In contrast, EvalRepair-C++ was based on the C++ section of HumanEval-X, carrying a higher risk of data leakage due to its earlier release on the open-source platform. On the other hand, recent studies such as [3] have also demonstrated that cross-language training setups can achieve competitive or even superior performance compared to monolingual approaches. Moreover, this decoupling of fine-tuning and evaluation languages also helps reduce the risk of overfitting to language-specific characteristics while emphasizing the generalizability of the federated fine-tuning approach for program repair across different programming languages.

4.3 Evaluation Models

Selection Criteria. To thoroughly evaluate the effectiveness and generalizability of federated fine-tuning, we select a diverse set of LLMs based on several crucial criteria:

– Firstly, it is imperative that the chosen LLMs are explicitly code-targeted, as they are pre-trained with code-related corpora to understand and generate programming languages, thereby enhancing the accuracy and effectiveness of code-related tasks.

[4]https://github.com/lin-tan/clm/commits/main/humaneval-java
[5]https://github.com/THUDM/CodeGeeX/commits/main/codegeex/benchmark/humaneval-x/java/data

- Secondly, diversity in underlying model architectures and sizes is of vital importance, enabling exploration of the generalizability and practicality of existing LLMs. Due to the constraints imposed by computational resources, we select models with sizes ranging from 7B to 15B parameters.
- Thirdly, we select LLM variants that are specifically fine-tuned for instruction following since such tailored versions of LLMs can capture complex prompts more accurately during fine-tuning.
- Furthermore, we also take into consideration the prevalence of different LLMs, as widely recognized models benefit from extensive community support, facilitating better integration and troubleshooting. On the other hand, we can also obtain a more comprehensive understanding of popular LLMs. Therefore, we choose widely used LLMs of code based on the numbers of downloads from HuggingFace[6]. We obtain the overall numbers of historical downloads for each model through the official API[7] from HuggingFace.
- Additionally, we also consider LLMs that have demonstrated superior performance in the current community to determine the final selection of the model. In this way, we can investigate whether existing state-of-the-art models can still maintain exceptional performance in our study. To that end, we refer to the leaderboard[8] from EvalPlus [60], which provides a comprehensive evaluation of existing state-of-the-art LLMs of code to guide model selection for our study. Specifically, the EvalPlus leaderboard assesses the LLMs on its enhanced HumanEval [17] and MBPP [5] benchmarks with expanded test cases that are 80 times and 35 times more than the original benchmarks, respectively. All the LLMs are ranked according to the average *Pass*@1 scores on the leaderboard.

Consequently, we take all the criteria into account and six different LLMs of code are selected in our study. The overview of the selected models is presented in Table 1, including the model configurations, number of model downloads and average scores from the EvalPlus leaderboard.

- **CodeLlama.** CodeLlama [82] is a cutting-edge family of LLMs developed by Meta AI, specifically tailored for code generation and understanding tasks, and can generate code snippets based on surrounding context with infilling capabilities. Building on the Llama 2 [96] architecture, CodeLlama includes several variants, such as the base model, Python-specific model, and instruction-following model. CodeLlama is designed for both research and commercial applications, making it a versatile tool for advancing code-related tasks in various domains. Due to resource constraints and the need to better understand the prompt, we employ CodeLlama-13B-Instruct and CodeLlama-7B-Instruct in our study to evaluate federated fine-tuning.
- **DeepSeekCoder.** DeepSeekCoder [26] is a series of open-source models specifically designed for coding tasks. The pre-training of DeepSeekCoder employs a project-level code corpus and a complementary fill-in-the-blank objective, enabling the model to handle code completion and infilling tasks at the project scale. DeepSeekCoder models excel in handling complex coding scenarios, outperforming existing closed-source models across various benchmarks. In this study, we employ DeepSeekCoder-7B-Instruct-V1.5 to leverage its advanced capabilities in code-related tasks.
- **WizardCoder.** WizardCoder [65] is a state-of-the-art LLM of code that enhances code generation and comprehension through advanced instruction fine-tuning techniques. It utilizes the Evol-Instruct [113] method that enables fine-tuning with more complex and diverse instructions for better code-related instruction following. We use WizardCoder-15B-V1.0 in this study to evaluate its applicability in federated learning.
- **Mistral.** Mistral-7B [33] uses Mistral as the base model and has made progress in the attention mechanism, demonstrating the potential of auto-regressive models. Mistral-7B is able to outperform Llama 2 across all benchmarks. We employ the instruction-tuned version Mistral-7B-Instruct-v0.2, which provides a

[6]https://huggingface.co/models
[7]https://huggingface.co/api/models
[8]https://evalplus.github.io/leaderboard.html

Table 1. Overview of Selected LLMs of Code (* The instruction-tuned versions of CodeLlama-13B and CodeLlama-7B are not evaluated in the leaderboard, thus the scores of the base versions are provided as a reference.)

Model	Base Model	Model Size	# Download	Average Score
CodeLlama-13B-Instruct	Llama2	13B	601.0K	45.5*
CodeLlama-7B-Instruct	Llama2	7B	886.3K	41.1*
DeepseekCoder-7B-Instruct-V1.5	DeepSeek-LLM	7B	115.5K	66.8
WizardCoder-15B-V1.0	CodeLlama	15B	155.9K	52.4
Mistral-7B-Instruct-v0.2	Mistral	7B	11.9M	36.5
CodeQwen1.5-7B-Chat	QWen	7B	115.8K	73.8

larger context window and no sliding-window attention compared with previous versions, to validate its effectiveness and consistency in the federated learning setting.

– **CodeQwen.** CodeQWen [7] is a specialized language model developed as part of the QWen series, designed to excel in code-related tasks. It builds upon the foundational QWen model and is a code-specific version of QWen, leveraging advanced training techniques such as reinforcement learning from human feedback (RLHF) to enhance its performance in coding tasks. We use CodeQwen1.5-7B-Chat, which is also an instruction-following version of CodeQWen, demonstrating robust code generation abilities and competitive performance across various code-related tasks including program repair.

For brevity, we refer to CodeLlama-13B-Instruct, CodeLlama-7B-Instruct, DeepSeekCoder-7B-Instruct-V1.5, WizardCoder-15B-V1.0, Mistral-7B-Instruct-v0.2, CodeQwen1.5-7B-Chat as CodeLlama-13B, CodeLlama-7B, DeepSeekCoder-7B, WizardCoder-15B, Mitral-7B and CodeQWen-7B in the subsequent sections of this paper.

4.4 Evaluation Metrics

Top@k. To evaluate the performance of the fine-tuned LLMs, we employ the *Top@k* metric, which assesses the model's ability to generate correct solutions within a limited number of attempts. The value of k represents the number of samples generated for each problem. In the *Top@k* evaluation, a problem is considered solved if at least one of the k generated samples for that problem passes the corresponding test cases. The *Top@k* score is then calculated as the proportion of successfully solved problems within the given k attempts. Previous research has shown that developers are less likely to utilize automated repair tools if they fail to generate a viable solution within a limited number of attempts [43, 73]. A study by Kochhar et al. [43] found that developers tend not to use automated repair tools if they cannot find a suitable fix within five attempts. Similarly, Noller et al. [73] reported that the maximum number of patches developers are willing to review is typically around 10. Taking these findings into consideration, we adopt the *Top*@5 and *Top*@10 metrics to evaluate the performance of the fine-tuned LLMs. Furthermore, we use *Top*@1, which is also a representative of the *Top@k* family, in order to evaluate the LLMs in generating accurate solutions with minimal effort from the developers.

Pass@k. In addition, we leverage an unbiased metric *Pass@k* [17] to assess the repairing performance comprehensively. The unbiased estimator is calculated as follows:

$$Pass@k = \mathbb{E}_{\text{problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \tag{8}$$

where n is the total number of generated samples by the LLM, $k \leq n$ is the number of attempts, and $c \leq n$ is the number of correct samples. We obtain *Pass@k* for each problem and then the expectation is taken over all problems. This metric calculates the expected value of the proportion of problems for which at least one correct solution is generated within k attempts from the n generated samples. The *Pass@k* metric provides a more nuanced view of the model's performance, as it takes into account the distribution of correct solutions among the

generated samples, rather than simply considering the presence or absence of a correct solution within a fixed number of attempts. Together with *Top@k*, we use *Pass*@5 and *Pass*@10 to evaluate the capability of fine-tuned LLMs in bug fixing.

4.5 Prompt Design

Effective prompt engineering is critical to guiding LLMs toward generating accurate and context-aware code repairs. Previous research [13, 93] indicates that lengthy prompts negatively impact LLM performance, and precise intentions can enhance response quality. Therefore, we prioritize the clarity and simplicity of prompt designs for our methods. To mitigate the influence caused by different prompt designs, we adopt a structured prompt template from the work of Yang et al. [116], where the prompt format is simple and precise. As illustrated in the prompt structure in Figure 4, a problem description is first provided to specify the intention and context of the code. Following the problem description is the corresponding incorrect code and a task-specific instruction that defines the domain of software engineering and sets the objective of bug fixing. Specifically, Figure 4 also shows an example of a problem description in which detailed information is provided to ensure the LLM understands the functional requirements and the expected input and output behavior of the problem. The complete incorrect code will also be provided as presented in Figure 4. Note that the fault location is not included in the prompt, and the LLM is supposed to output the complete repaired version of the incorrect code. All methods and LLMs use the prompt structure uniformly to ensure consistency and mitigate potential biases.

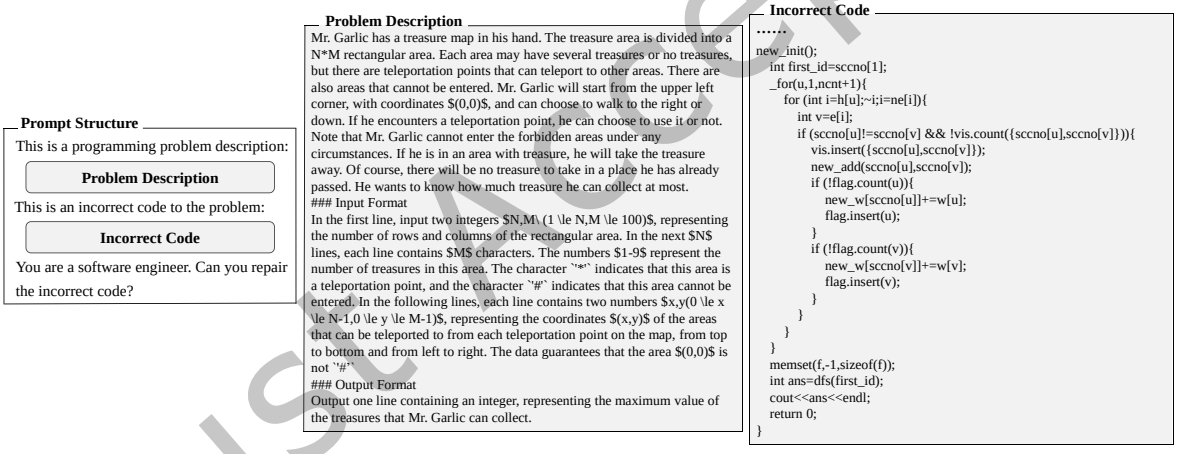


Fig. 4. Prompt design for program repair.

4.6 Experimental Setup

Hardware & Software Configuration. We conduct the experiments on 4 Nvidia RTX 4090 GPUs, each with 24GB of graphical memory. The system runs with Intel(R) Xeon(R) Platinum 8352V CPU @ 2.10GHz and 480GB of RAM on Ubuntu 20.04. We employ the FederatedScope-LLM (FS-LLM) [47] package as the fundamental framework for federated learning. FS-LLM is an open-source federated learning package developed by Alibaba, enabling large-scale and efficient federated fine-tuning for LLMs. We implement several key components, including the construction of diverse heterogeneous code scenarios, the parameter-efficient fine-tuning technique, and the

evaluation benchmark for the program repair task based on this framework to conduct a comprehensive study. All of our approaches are built based on Pytorch 1.10.0 and Cuda 11.3.

Hyperparameters. We refer to the settings for LLMs of code from the work of Jiang et al. [35] as the guidance of hyperparameter configuration in our study. We use the paged AdamW [62] optimizer, which is a variant of the AdamW optimizer designed to mitigate the occasionally out-of-memory issue on GPUs by allocating CPU RAM for the GPUs, with a learning rate of $1e^{-4}$ to update the weights. The batch size is set to one due to significant computational resources caused by large-scale models, and we set the max length of input tokens to 2048 to capture as much context as possible under the resource constraint. We fine-tune at most 30 epochs on each client and set the rounds of the global exchange of model parameters to 10. The fine-tuning process is regularized by an early stopping strategy that prevents excessive fine-tuning if the validation loss does not significantly decrease after 10 epochs. The rank and dropout for QLoRA are set to 32 and 0.05, respectively.

5 EXPERIMENTS & RESULTS

5.1 RQ1: Effectiveness of Federated Fine-tuning

[Experiment Goal for RQ1]: We aim to explore the effectiveness of federated learning in enhancing fine-tuning LLMs for program repair task. We compare the performance of federated learning with other fine-tuning scenarios and the original model, assessing its capability to improve model performance while protecting private data.

[Experiment Design for RQ1]: To validate the effectiveness of federated fine-tuning, we first compare federated learning with the original model, which is not fine-tuned to assess whether LLM fine-tuning can enhance program repair in the context of federated learning. We further investigate the efficacy of federated fine-tuning in improving performance while preserving data privacy by comparing it with other fine-tuning methods, i.e., local fine-tuning and centralized fine-tuning. In local fine-tuning, the models are fine-tuned within the local client, which has no access to datasets from other clients, and the model is updated locally without interacting with the central server. Therefore, we can validate to what extent federated learning can enhance performance with collaboration. We also compare it with centralized fine-tuning, which is regarded as the ideal situation for most traditional federated learning tasks since it pools all data together to fine-tune models regardless of data privacy or any other restrictions. It allows us to examine how close the performance of federated learning can be to centralized learning or whether it can outperform the centralized method. Comparing with the original model ensures that federated fine-tuning can indeed boost the LLM rather than introducing performance loss to the model. On the other hand, comparing with other fine-tuning methods provides insights into the potential of federated fine-tuning to enhance program repair through collaboration without compromising data privacy. In this RQ, we set the number of clients K to 100 to simulate the federated learning system where 100 clients collaboratively fine-tune the LLMs.

[Experimental Results for RQ1]: Table 2 presents the evaluation results of different fine-tuning methods on EvalRepair-Java. The best results for each metric are highlighted in bold and the second-best results are underlined.

Firstly, the results show that federated fine-tuning outperforms each original model on the benchmark. Specifically, for the *Top*@10 metric, the improvement is most significant on the Mistral-7B model, reaching an increase of 16.57%, while the smallest improvement is observed on the CodeLlama-7B model, with an increase of 6.75%. For the *Pass*@10 metric, the greatest improvement is on the DeepSeekCoder-7B model, achieving a boost of 18.44%, whereas the Codellama-7B model experiences the smallest increase of 7.77%. These results indicate that federated fine-tuning can indeed enhance bug-fixing capabilities with LLMs.

Secondly, the performance of federated fine-tuning surpasses local fine-tuning across all LLMs. We observe that local fine-tuning performs the worst among all fine-tuning methods, especially on DeepSeekCoder-7B where

Table 2. Experimental results of the original model (Original), local fine-tuning (Local), federated fine-tuning (FL) and centralized fine-tuning (Central) on EvalRepair-Java.

Scenario	CodeLlama-13B					CodeLlama-7B					DeepSeekCoder-7B				
	Top@5	Top@10	Pass@1	Pass@5	Pass@10	Top@5	Top@10	Pass@1	Pass@5	Pass@10	Top@5	Top@10	Pass@1	Pass@5	Pass@10
Original	56.44	66.87	20.76	52.36	65.59	53.37	64.42	25.72	54.31	63.44	39.26	53.37	10.10	34.85	50.62
Local	53.99	55.83	32.37	53.85	59.78	46.63	53.37	29.68	49.70	54.64	20.25	31.90	6.89	23.04	35.19
FL	63.80	**76.69**	**39.31**	**68.81**	**76.88**	**62.58**	**71.17**	**31.48**	**62.35**	**71.21**	48.46	68.10	16.61	51.62	69.06
Central	**66.87**	72.39	33.55	66.04	73.64	52.76	65.03	23.78	54.57	65.71	**62.58**	**82.21**	**26.33**	**66.45**	**79.84**
Scenario	WizardCoder-15B					Mistral-7B					CodeQWen-7B				
	Top@5	Top@10	Pass@1	Pass@5	Pass@10	Top@5	Top@10	Pass@1	Pass@5	Pass@10	Top@5	Top@10	Pass@1	Pass@5	Pass@10
Original	42.33	60.12	13.40	42.92	58.44	44.17	52.76	19.82	43.52	52.60	85.28	89.57	53.14	83.82	89.47
Local	41.72	47.85	21.99	43.14	49.78	26.38	36.81	12.76	25.68	35.52	71.17	79.14	50.97	73.82	79.22
FL	**56.44**	**68.10**	**26.10**	**57.94**	**68.27**	**60.12**	**69.33**	**29.31**	**58.14**	**69.21**	87.12	90.18	56.11	85.03	90.30
Central	50.31	60.12	16.80	48.10	60.93	42.94	53.37	14.20	40.06	52.23	**90.80**	**92.64**	**59.37**	**88.65**	**92.98**

the reduction on *Top*@10 is up to 21.47% compared to the original model. On the other hand, Mistral-7B shows the most significant decrease at 17.08% in terms of *Pass*@10, highlighting the necessity of collaborative learning.

The results also show that federated fine-tuning achieves the best scores in terms of both *Top@k* and *Pass@k* with CodeLlama-13B, CodeLlama-7B, WizardCoder-15B and Mistral-7B, which, to our surprise, even outperform those of centralized fine-tuning. Notably, the federated fine-tuning approach achieves a significant increase of 16.98% for *Pass*@10 and 15.96% for *Top*@10 on Mistral-7B compared with the centralized fine-tuning approach. The best performance is achieved by centralized fine-tuning for DeepSeekCoder-7B and CodeQWen-7B. However, federated fine-tuning achieves the second-best performance, approaching the central method, particularly for CodeQWen-7B. The differences between federated fine-tuning and centralized fine-tuning on CodeQWen-7B are only 2.46% and 2.68% for *Top*@10 and *Pass*@10, respectively.

To rigorously validate the equivalence between federated and centralized fine-tuning, we conduct statistical tests with a significance level of 0.05 , and calculate Cliff's Delta to verify the slight advantage of federated fine-tuning over centralized fine-tuning. The results in Table 3 show that there are no statistically significant differences (p-values > 0.05) between the two approaches across all LLMs, confirming that federated fine-tuning achieves performance statistically equivalent to centralized fine-tuning. In addition, positive Cliff's Delta values for CodeLlama-13B, CodeLlama-7B, WizardCoder-15B, and Mistral-7B indicate that federated fine-tuning slightly outperforms centralized fine-tuning for these models. Only DeepSeekCoder-7B and CodeQWen-7B exhibit marginally lower performance according to their Cliff's Delta values. These findings reinforce that federated fine-tuning not only matches centralized performance but can even surpass it for certain LLMs, revealing the potential of LLM-based federated fine-tuning for program repair.

Centralized learning is often regarded as the upper bound for traditional DL tasks in federated learning since it represents the ideal situation in real-world scenarios where all data are put together. Our results reveal that federated fine-tuning with LLMs can achieve comparable performance to centralized learning while protecting private data, whereas the central approach compromises data privacy. Besides, we also notice in a few previous research [34, 121, 124] that the LLM-based federated fine-tuning approach has the potential to outperform centralized fine-tuning, revealing the unsuitability of simply combining data from different distributions for centralized learning.

We also speculate that it is because in traditional federated learning tasks, models like deep neural networks have to learn from scratch and centralized training has direct access to all data, thus performing better than federated learning in general cases. However, our results demonstrate that this advantage of the central method does not apply to fine-tuning models like LLMs since they have already been pre-trained with massive corpora.

Table 3. Experimental results of Wilcoxon Signed-Rank Test and Cliff's Delta Measurement between Federated Fine-tuning and Centralized Fine-tuning

Model	p-value	Effect Size	Model	p-value	Effect Size	Overall	
						p-value	Effect Size
CodeLlama-13B	0.1875	0.2	WizardCoder-7B	0.0625	0.36		
CodeLlama-7B	0.0625	0.36	Mistral-7B	0.0625	0.68	0.0606	0.12
DeepSeekCoder-7B	0.0625	-0.36	CodeQWen-7B	0.0625	-0.52		

Finding 1: Federated fine-tuning can effectively enhance program repair capabilities compared to other fine-tuning methods, even rivaling the centralized fine-tuning approach. Federated fine-tuning exhibits the potential for industries to collaborate and improve performance without compromising data privacy.

We also note that while centralized fine-tuning generally performs well in most scenarios, a few cases still exist where the central approach even performs worse than the original models. For example, the *Top*@5 and *Pass*@1 of the centralized approach on CodeLlama-7B are 0.61% and 1.94% lower than the original model. In addition, centralized fine-tuning results in worse performance on Mistral-7B for all metrics except for *Top*@10 compared with the original model, with the *Pass*@1 metric notably decreasing by 5.62%. Moreover, the local fine-tuning approach on all LLMs presents the worst performance among all baselines. Since the central approach utilizes the full dataset whereas the local approach fine-tunes the LLMs with only a portion of all data, we speculate that the size of the fine-tuning dataset could influence the performance of LLM fine-tuning, which will be further discussed in detail in Section 6.

In summary, compared with the local and centralized fine-tuning approaches, federated fine-tuning enhances program repair by facilitating indirect client collaboration to utilize all datasets implicitly while protecting data privacy, thus combining the advantages and overcoming the disadvantages of both approaches. Furthermore, federated fine-tuning demonstrates promising and consistent performance across all LLMs, whereas the other two approaches exhibit performance degradation in some cases.

Finding 2: The local fine-tuning and centralized fine-tuning approaches demonstrate instability and inconsistency in fine-tuning LLMs for program repair. Despite that centralized fine-tuning achieves promising performance in most cases, there are still some LLMs fine-tuned by centralized learning that perform worse than the original models such as CodeLlama-7B and Mistral-7B. On the other hand, the local fine-tuning approach results in the worst performance compared to all baselines, which again highlights the advantage of federated fine-tuning and the necessity in collaboration.

The Venn diagram of unique fixes generated by different LLMs based on the top-10 fixes with federated fine-tuning is presented in Figure 5. By performing manual verification to eliminate the overfitted code, we only found one common overfitted bug for CodeLlama-13B, CodeLlama-7B and Mistral-7B, which is discussed in detail in Section 6. After eliminating the overfitted code, the numbers of bugs fixed by CodeLlama-7B, CodeQWen-7B, DeepSeekCoder-7B, Mistral-7B, WizardCoder-15B and CodeLlama-13B are 114, 147, 111, 112, 111 and 124. Despite the 72 bugs repaired by all LLMs, CodeQWen-7B is able to fix 7 unique bugs, followed by CodeLlama-13B, which can fix 2 unique bugs. However, CodeLlama-13B exhibits a narrower scope in terms of the bugs it can fix, with a relatively smaller overlap with other LLMs compared with CodeQWen-7B. This emphasizes the complementary

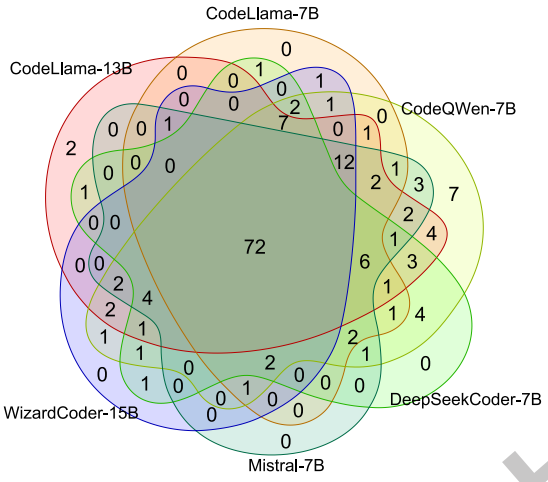


Fig. 5. Bug fix Venn diagram on EvalRepair-Java.

strengths of CodeQWen-7B since it presents larger overlaps of fixes with and without CodeLlama-13B. For example, CodeQWen-7B is able to fix 3 bugs with Mistral-7B, 4 bugs with DeepSeekCoder-7B, and 4 bugs with CodeLlama-13B, which demonstrates that CodeQWen-7B can be complementary to other LLMs of code, presenting the potential of CodeQWen-7B for further use.

Finding 3: CodeQWen-7B and CodeLlama-13B are able to fix 7 and 2 more unique bugs than others, respectively. It is noteworthy that CodeQWen-7B demonstrates its complementary strength in relation to CodeLlama-13B. Not only does CodeQWen-7B exhibit a larger overlap with CodeLlama-13B, but it also extends its bug-fixing capabilities beyond the scope covered by CodeLlama-13B, indicating that CodeQWen-7B complements CodeLlama-13B and other LLMs by addressing a broader range of bugs.

5.2 RQ2: Impact of data heterogeneity

[Experiment Goal for RQ2]: We aim to further investigate the impact of various data distributions on LLM-based federated fine-tuning for program repair. We construct heterogeneous code to explore whether fine-tuning LLMs under diverse Non-IID scenarios is able to enhance the performance and whether it incurs negative impacts on the performance compared to IID scenario.

[Experiment Design for RQ2]: In order to comprehensively analyse the impact of Non-IID data on fine-tuning LLMs for program repair, we construct different degrees of Non-IID scenarios along with the IID scenario to evaluate federated fine-tuning. Most previous research in federated learning and applying federated learning to SE mainly focuses on typical supervised tasks such as classification, regression, etc, which enable them to construct Non-IID data conforming to label skew based on the labeled datasets. However, few of them pay attention to generative tasks such as program repair and other code-related tasks that involve feature skew, which is also a critical challenge that may occur in real-world applications because of different software development practices across companies. Existing research constructs feature skew by collecting datasets from different data sources,

Table 4. A brief overview of coding style attributes.

Attribute Type	Description	Example
Layout	The organization of code	Usage of indention, empty lines, etc.
Lexical	The tokens used in the code	Usage of naming method, variable, global declaration, etc.
Syntactic	Tree structure and syntactic constructs of a program	Location of variables, variable assignment, etc.
Semantic	Control flows and data flows of a program	Usage of loop structures, conditional structures, etc.

domains, or individuals that inherently exhibit Non-IID properties [16, 54, 56, 89]. The code repositories from different companies can differ to varying degrees because they follow different coding standards, developers have different levels of programming skills, or they need to satisfy different business requirements. Therefore, we leverage three types of code features in our study to construct feature-skewed Non-IID data to study its impact in depth.

Coding Style. Firstly, considering program readability and maintainability, companies generally would enforce specific programming guidelines and requirements to restrict coding style for easier code modification or code refactoring in the future [95, 104]. Individuals can write the same program in various ways because of different programming habits. For example, the usage of Camel case, Pascal case, or underscores as identifier naming method, the preference to use *if* or *switch* for conditional structures, etc. Coding style is primarily composed of four different attributes, namely layout features, lexical features, syntactic features, and semantic features [12, 55]. A brief overview of the four types of coding style attributes is presented in Table 4. Previous works extract coding styles in source code authorship attribution to identify programmers, students, or other individuals [6, 12, 44]. We use coding style as the first type of code feature to construct feature-skewed Non-IID data. In order to obtain the coding style attributes from the dataset, we follow the work in [55] to extract the attributes from our dataset, which results in 23 specific attributes across lexical, syntactic, and semantic features.

Code Complexity. Secondly, code complexity is one of the major aspects that affects the internal quality of software during software development and complex code can increase the cognitive burden for code comprehending [4, 76]. However, developers from different enterprises can vary in their experience, problem-solving approaches, and programming skills, which generally end up with different complexity of code. Therefore, we consider code complexity as the second code feature to construct feature-skewed Non-IID data. According to Antinyan et al. [4], there are three direct sources of code complexity: (1) The elements and connections existing in the code. (2) Representational clarity of the code. (3) Intensity of evolution. The third source indicates that the frequency and magnitude of changes in the code can directly reflect the complexity level of the code since it influences the time cost of code maintenance. In view of this, to assess the code complexity of our bug-fix code pairs, we leverage the number of modified code hunks of each pair by comparing the buggy code with its fixed version to indicate the complexity levels of our dataset.

Code Embedding. Thirdly, we extract code embeddings to capture the underlying semantic and syntactic information of code. Due to the fact that different companies engage in diverse business activities, they inevitably have varied software development needs. This diversity leads to underlying differences in the information presented within their codebases. These distinct software requirements also reveal differing intentions within the code, reflecting the unique objectives of each company. Therefore, we extract the code embeddings from our dataset as the third code feature to construct feature-skewed Non-IID data since code embeddings provide an enhanced understanding of the semantic and syntactic meanings of code tokens as the pre-trained models are developed using extensive external datasets [20]. Furthermore, we utilize CodeBERT [21] as the pre-trained model for code embedding extraction. CodeBERT is capable of capturing the semantic relationship between natural language (NL) and programming language (PL) and can effectively support NL-PL understanding for

generative tasks. Given that our dataset provides natural language descriptions as metadata for each code, we use the NL-PL pair as the input for CodeBERT and obtain the contextual vector representation as the output.

Feature Clustering. Lastly, we aim to construct feature-skewed Non-IID scenarios based on the three code features to further explore the impacts of data heterogeneity. Existing studies construct feature-skewed Non-IID data mainly through feature probability distribution and code clustering based on embeddings [59, 72]. Similar to the idea of label-skewed scenarios where data samples are allocated to clients in a Non-IID manner according to the label distribution of the dataset, clustering unlabeled data into groups provides a form of implicit labels that can guide the Non-IID data allocation. Previous studies cluster the code by extracting code description features, contextual syntactic features, code change features, or using a code embedding model to encode the code [46, 57, 122]. Considering real-world scenarios as described previously, we perform code clustering based on the coding style and code embedding features except for code complexity, of which we use the complexity level as the label. We adopt Bisecting-K-means [71], which is a hierarchical clustering algorithm in order to determine a reasonable number of clusters. To that end, we analyse the sum of squared error (SSE) to determine the appropriate number of clusters. The SSE is defined as follows:

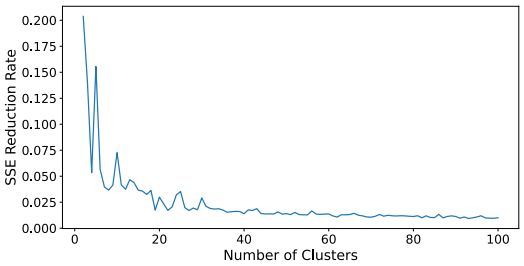
$$SSE = \sum_{k=1}^{K} \sum_{i \in C_k} (x_i - \mu_k)^2 \tag{9}$$

where K is the number of clusters and C_k is the set of elements in cluster k. x_i represents the i-th element in cluster k and μ_k is the centroid of cluster k. SSE measures the variance within the clusters, thus a smaller SSE indicates better clustering.

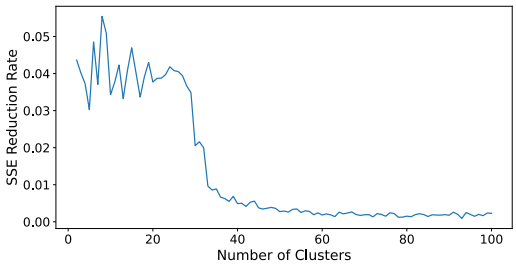
Optimizing Cluster Sizes. Figure 6a and Figure 6b illustrate the reduction rate of SSE according to different numbers of clusters in terms of coding style and code embedding features. We observe that the SSE no longer declines significantly after the number of 20 in terms of the coding style feature. Due to resource constraints caused by clustering on high-dimensional data with a large number of clusters, we set the number of clusters to 40 for coding style according to the SSE drop rates. As demonstrated in Figure 6b, the SSE drop rate decreases rapidly from approximately 25 to 40 clusters with a notably low rate of around 35 clusters, which is the number of distinct problems in TutorLLMCode. Since CodeBERT extracts the contextual information from both natural language description and code in the NL-PL pairs, underlying semantic information in the code can also be captured beside each unique problem. Similar purposes can exist in the code snippets that address different problems when they have common sub-problems, such as sorting and depth-first search, which results in lower SSE drop rates with more than 35 clusters. Taking into account the resource limitations, we set the number of clusters to 45 for the feature of code embedding.

Figure 7 presents the distribution of the 40 clustered coding styles and the distribution of different code complexity levels represented by the number of modified hunks. As depicted in Figure 7b, there are 13 levels of code complexity since there is no code with 12 modified hunks.

Construction of Heterogeneous Code. We allocate the datasets to each client in a Non-IID manner through the Dirichlet distribution [31], which is widely applied in previous studies [42, 51, 109]. Given a set of N clients and K clusters, the cluster assignments are assumed to follow a categorical distribution. Specifically, each client $n \in \{1, ..., N\}$ is assigned to a cluster according to a probability vector $\boldsymbol{q}_n$, where $\boldsymbol{q}_{n,k} > 0$ denotes the probability that client n belongs to cluster $k \in \{1, ..., K\}$ and $||\boldsymbol{q}_n||_1 = 1$. The probability vector $\boldsymbol{q}_n$ is drawn independently from a Dirichlet distribution, denoted as $\boldsymbol{q}_n \sim Dir(\alpha \cdot \boldsymbol{p})$ where $\alpha > 0$ is a concentration parameter controlling the degree of heterogeneity in the cluster assignments and $\boldsymbol{p}$ represents a prior uniform distribution over the K clusters. As $\alpha \to \infty$, the distributions of clusters assigned to each client are almost uniform, which also indicates the IID scenario. On the contrary, as $\alpha \to 0$, the distribution tends towards an extreme case, concentrating the probability mass on a single cluster for each client. Therefore, we construct different degrees of Non-IID scenarios

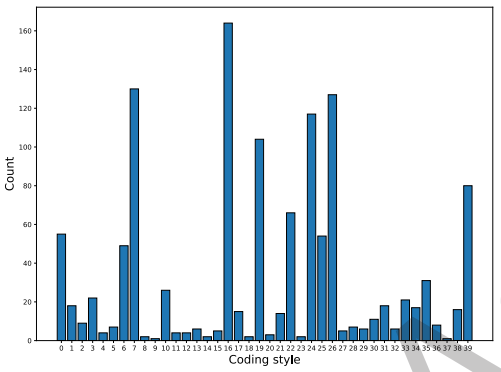


(a) Clustering of coding style.

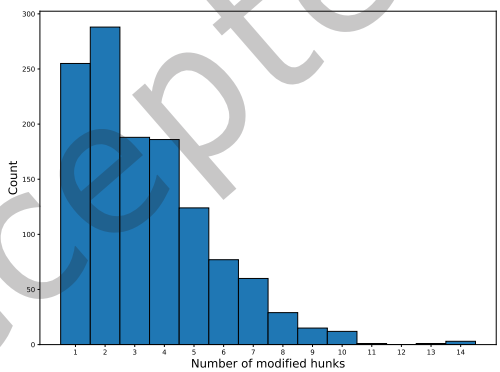


(b) Clustering of code embedding.

Fig. 6. SSE drop rate for different numbers of clusters.



(a) Distribution of coding styles.

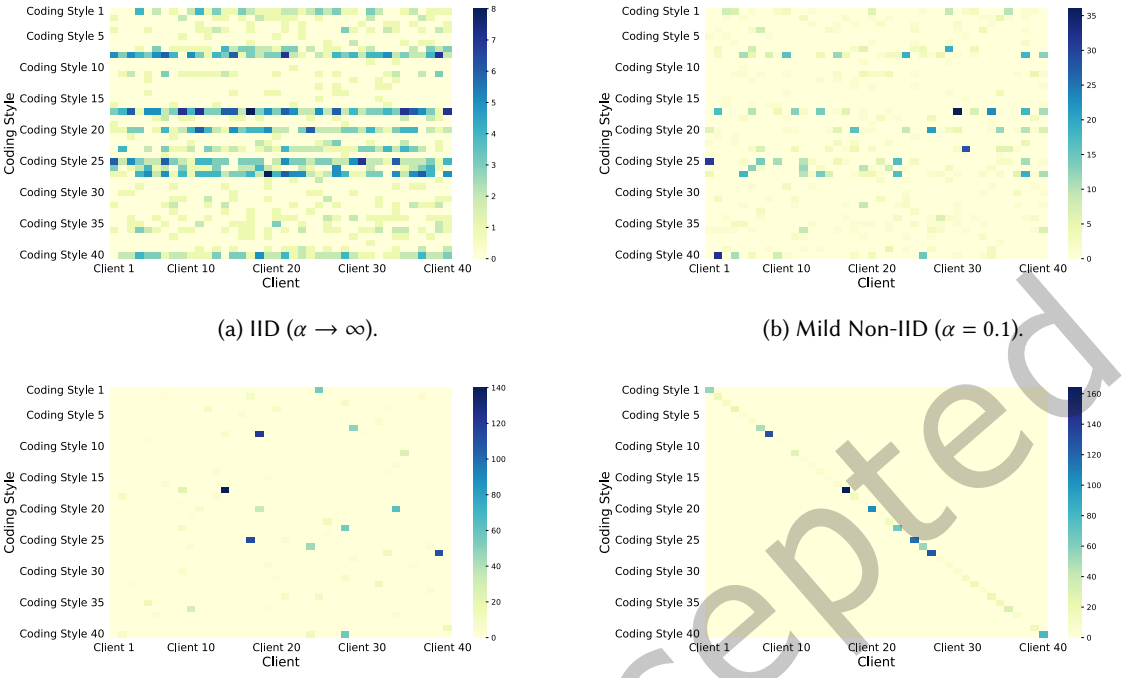


(b) Distribution of code complexity.

Fig. 7. Distribution of coding styles and code complexity.

by controlling α. Specifically, we set α to 0.1 and 0.01 for mild and medium degrees of heterogeneity, respectively. In addition, we construct an extreme Non-IID scenario where $\alpha \rightarrow 0$ and each client only holds data of a single cluster. Therefore, we set the numbers of clients to 40 and 45 for coding style and code embedding, respectively, to fit the extreme Non-IID scenario according to their numbers of clusters. We set the number of clients to 10 in terms of code complexity because there are only a few or no code samples with numbers of modified hunks greater than 10, as presented in Figure 7b. Therefore, we combine code samples with a number of modified hunks not less than 10 into a single cluster for better construction of the extreme Non-IID scenario.

Figure 8 presents as an example the four data distributions employed in our study based on the feature of coding style. As we can observe from Figure 8a, the data distributions across clients tend to be identical, whereas the distributions in Figure 8b, Figure 8c, and Figure 8d become more concentrated as the degree of heterogeneity increases. In particular, each client possesses only a single cluster of data in the extreme Non-IID scenario, as shown in Figure 8d. We compare the four scenarios of data distribution with the original model and centralized fine-tuning to explore the impact of data heterogeneity.



(a) IID ($\alpha \rightarrow \infty$).

(b) Mild Non-IID ($\alpha = 0.1$).

(c) Medium Non-IID ($\alpha = 0.01$).

(d) Extreme Non-IID ($\alpha \rightarrow 0$).

Fig. 8. Different degrees of Data Heterogeneity.

Examples of Heterogeneous Code. The examples of heterogeneous coding style, code complexity, and code embedding distributed in different clients are presented in Figure 9 and Figure 10. We interpret the three examples as follows: (1) As depicted in Figure 9, the coding style of the two code snippets on client i and client j where $i \neq j$ differs across lexical, syntactic, and semantic attributes. The code snippet on client i uses Pascal case as the naming method, whereas the method name is separated by an underscore in client j in terms of lexical differences. The two clients also differ in the locations of increment operators, which is one of the syntactic differences. For semantic differences, client i and client j import different header files and use different loop structures. In addition, client i uses static array allocation while client j uses dynamic memory allocation. Assuming that each client in the federated learning system represents a different company, developers in real-world settings are likely to have varying coding styles as aforementioned. Thus, we evaluate heterogeneous coding styles to study their impacts on our work. (2) Figure 10 demonstrates the heterogeneity in code complexity where client i only needs to fix a single-hunk bug and client j needs to fix four hunks of bugs. As a major factor affecting software maintenance, code complexity differs across code repositories from various industries, and the impact on LLMs learning with heterogeneous code complexity is to be evaluated in our study. (3) Since the contextual information is extracted by CodeBERT from the NL-PL pairs, the heterogeneity in code embedding, as also illustrated in Figure 10, is reflected in different code snippets within a client addressing the same problem, whereas different clients target different goals. Real-world industries focus on addressing different problems due to a diversity of business requirements and objectives. Therefore, the impact of heterogeneous code embedding will also be explored in this study.

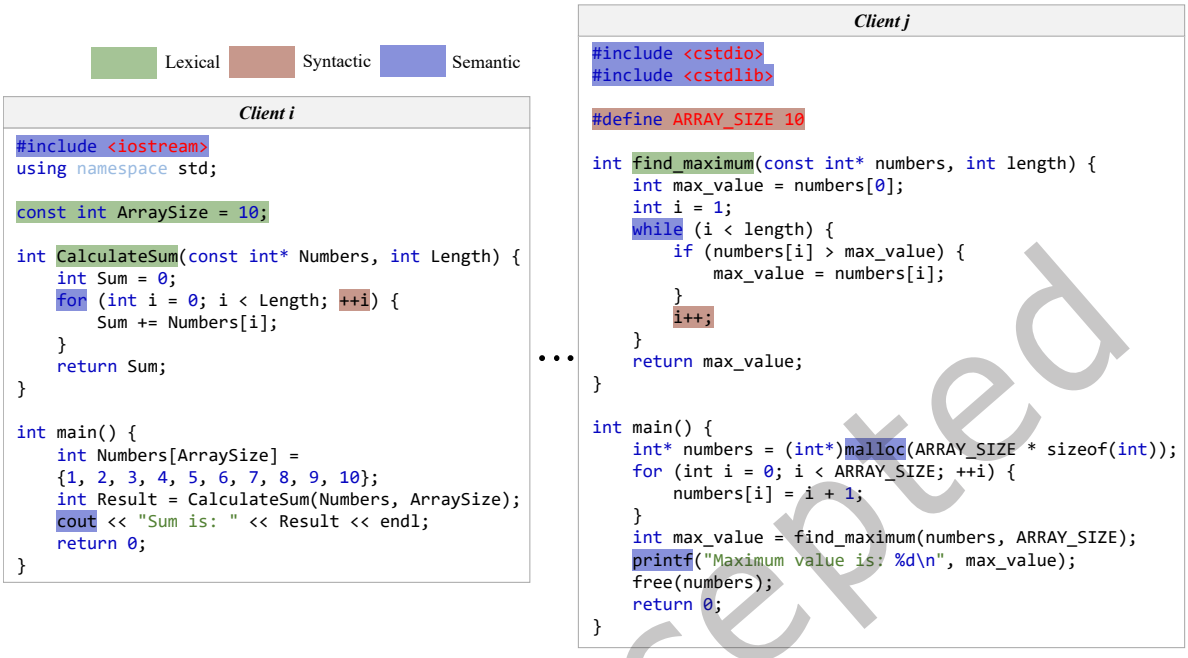


Fig. 9. An example of heterogeneous coding style.

Significance Test. Furthermore, we employ Wilcoxon signed-rank test [105] to evaluate the statistical significance of the performance differences between various scenarios, in which the null hypothesis is that there is no significant difference between two scenarios. This non-parametric test is widely used by previous studies [115, 121, 129] and is particularly suitable for non-normal distributions.

Effect Size. To further validate the significance of the observed differences and assess the magnitude of the effect, we adopt Cliff's delta [18], a non-parametric effect size measure. Cliff's delta has been widely used in recent studies [3, 84, 121] to quantify the size of differences. The value of Cliff's delta is confined to the range $[-1, 1]$, with 0 indicating no difference and values closer to -1 or 1 indicating a larger effect size. We follow the study of Yang et al. [121] to interpret the absolute value of Cliff's delta in different ranges as follows: (1) Negligible effect: $[0, 0.11)$. (2) Small effect: $[0.11, 0.28)$. (3) Medium effect: $[0.28, 0.43)$. (4) Large effect: $[0.43, 1]$.

Note that the numbers of clients set in this RQ are different from RQ1, which can produce different results for the same baselines.

[Experimental Results for RQ2]: The evaluation results of different data distributions based on different code features are presented in Table 5, Table 6, and Table 7. Mild, Medium, and Extreme represent three different degrees of Non-IID scenarios as previously designed. Note that the best and second-best performances are highlighted only within the IID, Mild, Medium, and Extreme scenarios since we aim to investigate the impact of different data heterogeneity. We find that fine-tuning LLMs with heterogeneous code can still enhance the repairing capability and achieve promising results.

For the mild Non-IID coding style scenario, as illustrated in Table 5, CodeLlama-7B and Mistral-7B exhibit the best repairing capabilities among all data distributions. DeepSeekCoder-7B achieves the best performance

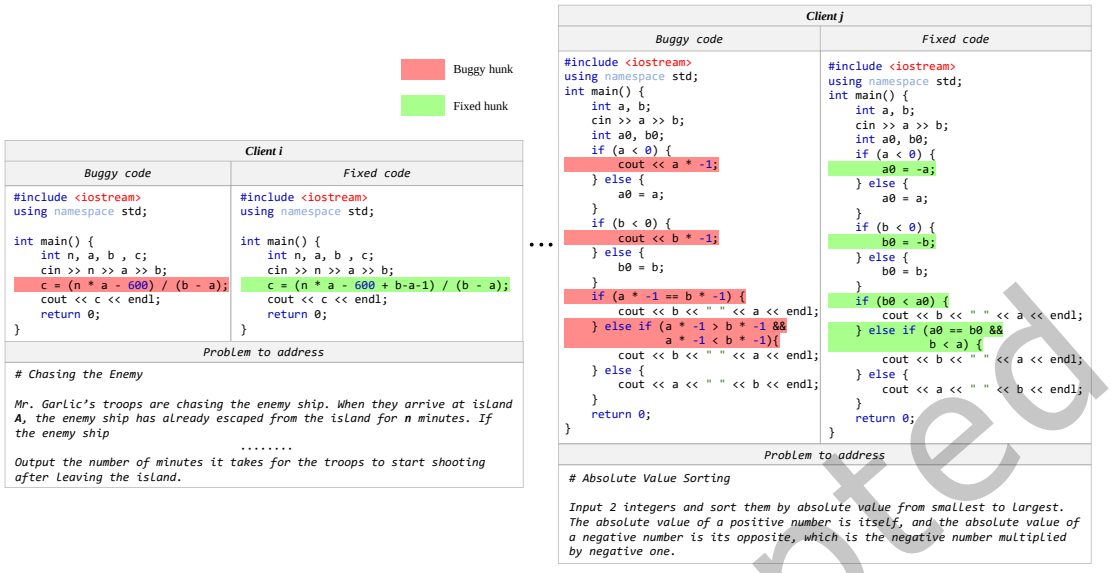


Fig. 10. An example of heterogeneous code complexity and code embedding.

Table 5. Experimental results of different coding style distributions on EvalRepair-Java.

Scenario	CodeLlama-13B					CodeLlama-7B					DeepSeekCoder-7B				
	Top@5	Top@10	Pass@1	Pass@5	Pass@10	Top@5	Top@10	Pass@1	Pass@5	Pass@10	Top@5	Top@10	Pass@1	Pass@5	Pass@10
Original	56.44	66.87	20.76	52.36	65.59	53.37	64.42	25.72	54.31	63.44	39.26	53.37	10.10	34.85	50.62
IID	70.55	80.37	36.01	70.78	79.26	57.06	64.42	27.51	58.76	67.43	43.56	60.12	11.84	42.10	61.72
Mild	70.55	78.53	37.94	69.24	76.80	**61.96**	**74.85**	29.78	**62.76**	**73.30**	52.76	68.10	15.95	50.04	67.46
Medium	69.33	74.23	38.18	70.48	80.72	60.74	69.33	37.05	62.30	69.14	**58.90**	**75.46**	**22.22**	**60.78**	**76.24**
Extreme	**71.78**	**80.98**	**43.61**	**73.75**	**80.73**	60.74	66.26	**37.14**	62.15	68.06	51.53	71.78	17.08	53.53	72.08
Central	66.87	72.39	33.55	66.04	73.64	52.76	65.03	23.78	54.57	65.71	62.58	82.21	26.33	66.45	79.84
Scenario	WizardCoder-15B					Mistral-7B					CodeQWen-7B				
	Top@5	Top@10	Pass@1	Pass@5	Pass@10	Top@5	Top@10	Pass@1	Pass@5	Pass@10	Top@5	Top@10	Pass@1	Pass@5	Pass@10
Original	42.33	60.12	13.40	42.92	58.44	44.17	52.76	19.82	43.52	52.60	85.28	89.57	53.14	83.82	89.47
IID	**63.80**	**76.07**	**31.52**	**67.65**	**77.83**	50.92	61.35	23.08	52.95	62.13	84.05	89.57	56.44	84.71	90.06
Mild	55.21	67.48	21.42	54.72	67.82	**53.99**	**65.64**	26.00	**56.26**	**67.03**	80.98	86.50	54.18	82.81	87.48
Medium	53.99	68.10	22.79	56.03	67.15	48.47	62.58	25.48	52.44	64.41	**85.89**	89.57	59.13	85.65	90.37
Extreme	52.15	68.10	24.07	56.67	66.31	52.76	65.03	**27.61**	52.76	63.26	85.28	**91.41**	**61.49**	**86.66**	**91.54**
Central	53.99	63.80	16.80	48.10	60.93	42.94	53.37	14.20	40.06	52.23	90.80	92.64	59.37	88.65	92.98

in terms of all metrics with *Top*@10 and *Pass*@10 at 76.07% and 75.42%, under the scenario of mild Non-IID code complexity as shown in Table 6. On the other hand, we observe that the DeepSeekCoder-7B model is enhanced the most with an increase of 14.73% and 16.84% on *Top*@10 and *Pass*@10 compared to the original model, respectively, as presented in Table 5. Remarkably, the DeepSeekCoder-7B model is improved by 22.27% and 24.80% on *Top*@10 and *Pass*@10, under the mildly heterogeneous code complexity scenario as presented in Table 6. The fine-tuned Mistral-7B shows the largest improvement under the scenario of mildly heterogeneous code embedding in Table 7, with an increase of 14.72% and 13.79% on *Top*@10 and *Pass*@10. The results demonstrate

Table 6. Experimental results of different code complexity distributions on EvalRepair-Java.

Scenario	CodeLlama-13B					CodeLlama-7B					DeepSeekCoder-7B				
	Top@5	Top@10	Pass@1	Pass@5	Pass@10	Top@5	Top@10	Pass@1	Pass@5	Pass@10	Top@5	Top@10	Pass@1	Pass@5	Pass@10
Original	56.44	66.87	20.76	52.36	65.59	53.37	64.42	25.72	54.31	63.44	39.26	53.37	10.10	34.85	50.62
IID	72.39	78.53	38.84	70.41	78.30	60.12	73.01	30.34	62.75	72.77	53.37	68.10	16.99	51.57	66.68
Mild	69.33	76.07	40.25	70.06	76.91	53.37	66.26	27.42	57.75	67.69	**63.80**	**76.07**	**26.24**	**62.47**	**75.42**
Medium	**73.01**	**80.98**	**43.51**	**73.22**	**79.50**	**63.80**	**75.46**	**33.51**	**64.94**	**75.04**	53.99	68.10	17.37	52.95	69.32
Extreme	68.71	76.07	37.33	69.42	76.80	60.74	69.94	30.34	63.27	71.36	46.63	67.48	17.08	52.30	68.76
Central	66.87	72.39	33.55	66.04	73.64	52.76	65.03	23.78	54.57	65.71	62.58	82.21	26.33	66.45	79.84
Scenario	WizardCoder-15B					Mistral-7B					CodeQWen-7B				
	Top@5	Top@10	Pass@1	Pass@5	Pass@10	Top@5	Top@10	Pass@1	Pass@5	Pass@10	Top@5	Top@10	Pass@1	Pass@5	Pass@10
Original	42.33	60.12	13.40	42.92	58.44	44.17	52.76	19.82	43.52	52.60	85.28	89.57	53.14	83.82	89.47
IID	**61.35**	**71.17**	**26.05**	**60.23**	**69.98**	47.85	64.42	21.05	51.84	64.78	84.66	90.80	56.77	85.67	90.40
Mild	55.21	63.80	21.19	52.50	63.16	53.99	64.42	**26.38**	53.74	63.47	82.21	90.80	55.40	84.94	90.15
Medium	49.08	64.42	17.27	49.76	64.35	52.76	**67.48**	24.87	**53.98**	**68.01**	86.50	**92.02**	56.91	85.88	**91.25**
Extreme	51.53	67.48	17.27	50.67	65.64	**54.60**	64.42	23.69	53.59	65.97	**88.96**	89.57	**59.70**	**86.59**	90.24
Central	53.99	63.80	16.80	48.10	60.93	42.94	53.37	14.20	40.06	52.23	90.80	92.64	59.37	88.65	92.98

Table 7. Experimental results of different code embedding distributions on EvalRepair-Java.

Scenario	CodeLlama-13B					CodeLlama-7B					DeepSeekCoder-7B				
	Top@5	Top@10	Pass@1	Pass@5	Pass@10	Top@5	Top@10	Pass@1	Pass@5	Pass@10	Top@5	Top@10	Pass@1	Pass@5	Pass@10
Original	56.44	66.87	20.76	52.36	65.59	53.37	64.42	25.72	54.31	63.44	39.26	53.37	10.10	34.85	50.62
IID	**75.46**	**80.37**	38.89	72.73	**82.28**	**61.96**	**69.94**	27.74	61.46	**71.09**	36.81	55.83	11.94	42.90	62.02
Mild	69.33	77.91	40.16	68.50	76.08	60.12	68.10	30.39	60.35	67.62	30.67	47.85	10.05	37.00	55.77
Medium	70.55	79.14	42.24	72.54	79.85	61.35	68.10	**33.03**	61.84	68.98	50.92	68.10	16.28	50.98	68.59
Extreme	71.17	77.30	**43.98**	**72.84**	79.93	60.12	69.33	32.42	**62.14**	69.89	**55.83**	**74.23**	**17.56**	**54.42**	**71.44**
Central	66.87	72.39	33.55	66.04	73.64	52.76	65.03	23.78	54.57	65.71	62.58	82.21	26.33	66.45	79.84
Scenario	WizardCoder-15B					Mistral-7B					CodeQWen-7B				
	Top@5	Top@10	Pass@1	Pass@5	Pass@10	Top@5	Top@10	Pass@1	Pass@5	Pass@10	Top@5	Top@10	Pass@1	Pass@5	Pass@10
Original	42.33	60.12	13.40	42.92	58.44	44.17	52.76	19.82	43.52	52.60	85.28	89.57	53.14	83.82	89.47
IID	**66.26**	**73.01**	**27.98**	**63.98**	**74.42**	51.53	66.87	23.12	52.98	65.60	**85.28**	**90.18**	58.47	85.34	**90.45**
Mild	55.21	65.64	24.26	57.66	69.25	50.92	**67.48**	23.83	**53.97**	**66.39**	81.60	88.34	53.37	83.79	89.38
Medium	59.51	66.26	22.89	56.55	68.80	**54.60**	66.87	**25.01**	52.88	64.35	84.05	89.57	53.80	84.43	90.38
Extreme	57.06	66.26	24.73	55.80	65.20	50.92	61.96	22.84	50.85	62.16	84.66	89.57	**61.68**	**86.04**	90.40
Central	53.99	63.80	16.80	48.10	60.93	42.94	53.37	14.20	40.06	52.23	90.80	92.64	59.37	88.65	92.98

that the federated fine-tuning approach is effective in enhancing the performance of LLMs for program repair under the mild degree of heterogeneity so far.

As the heterogeneity increases, we further investigate whether fine-tuning under the medium Non-IID scenario can enhance the repairing ability of LLMs. The results in Table 5 and Table 6 show that DeepSeekCoder-7B, CodeLlama-13B and CodeLlama-7B outperform the other data distributions under the medium Non-IID scenario, particularly with the DeepSeekCoder-7B model under the heterogeneous coding style scenario, achieving the largest increase of 22.09% and 25.62% on *Top*@10 and *Pass*@10 compared to the original model. We also notice from Table 7 that CodeLlama-7B experiences the smallest improvement of the bug fixing capability of 3.68% and 5.54% in terms of *Top*@10 and *Pass*@10. The results indicate that all of the LLMs can be enhanced except for CodeQWen-7B, which will be discussed in subsequent analysis, under the scenario of medium data heterogeneity.

In the extreme Non-IID scenario, each client only holds a single category of data. Different from previous findings that indicate the severity of Non-IID data correlates with increased impact on the performance [28, 51, 63], our study reveals that fine-tuning LLMs under even the most heterogeneous Non-IID scenario is able to enhance

Table 8. Experimental results of Wilcoxon Signed-Rank Test and Cliff's Delta Measurement across different data distributions

Code Feature	IID Original		Mild Original		Medium Original		Extreme Original		Mild IID		Medium IID		Extreme IID	
	p-value	Effect Size	p-value	Effect Size	p-value	Effect Size	p-value	Effect Size	p-value	Effect Size	p-value	Effect Size	p-value	Effect Size
Coding Style	5.25E-06	0.29	2.05E-07	0.34	2.56E-06	0.36	2.56E-09	0.34	0.37	0.01	0.14	0.05	0.03	0.06
Code Complexity	3.73E-09	0.34	5.32E-06	0.35	1.86E-09	0.33	2.56E-06	0.31	0.58	-0.03	0.07	0.02	0.16	-0.06
Code Embedding	4.80E-06	0.31	5.59E-05	0.24	3.90E-06	0.34	2.85E-06	0.35	9.22E-06	-0.11	0.58	-0.04	0.66	-0.02
Overall	1.41E-15	0.31	1.07E-13	0.31	4.74E-16	0.35	5.65E-16	0.33	0.05	-0.04	0.12	0.01	0.56	-2.72E-03

the program repair performance. For example, the CodeLlama-13B model in Table 5 achieves the *Top*@10 and *Pass*@10 at 80.98% and 80.73% and the DeepSeekCoder-7B model in Table 7 achieves the *Top*@10 and *Pass*@10 at 74.23% and 71.44%, both outperforming the other data distributions across all metrics under the extreme Non-IID scenario. We also notice that the DeepSeekCoder-7B model fine-tuned under the extreme scenario obtains the largest increment on both *Top*@10 and *Pass*@10 compared to other LLMs. For instance, Table 5 presents that the *Top*@10 and *Pass*@10 of the fine-tuned DeepSeekCoder-7B are increased by 18.41% and 21.46%. The results further demonstrate that heterogeneous code can benefit the fine-tuning of LLMs for program repair.

Similarly, it is also evident that fine-tuning LLMs under Non-IID scenarios can achieve better performance than centralized fine-tuning. We observe that except for DeepSeekCoder-7B and CodeQWen-7B, all of the other LLMs outperform the central approach across all data distributions, especially on *Top*@10 and *Pass*@10. For instance, as Table 6 presents, the Mistral-7B model under medium Non-IID scenario achieves an increase of 14.11% and 13.92% in terms of *Top*@10 and *Pass*@10 compared to the central approach. An increase of 13.36% on *Pass*@1 can also be observed on CodeLlama-7B under the extreme Non-IID scenario in Table 5.

In order to further confirm the effectiveness of federated fine-tuning under different data distributions, we also conduct statistical tests to validate whether and to what extent it can improve the repairing capability of LLMs. Table 8 reveals the significance of the improvement achieved by fine-tuning with different data distributions. Specifically, the results in Table 8 present the p-values and effect sizes of the tests between fine-tuning with all distributions and the original model, as well as between different Non-IID distributions and the IID distribution, across the LLMs for each code feature.

We observe that all of the differences between the performance of federated fine-tuning with different distributions and the original models (i.e., from *IID & Original* to *Extreme & Original*) are statistically significant (i.e., p-value<0.05). For instance, the LLMs fine-tuned under the scenario of extremely heterogeneous coding style yield a p-value of $2.56e^{-6}$ and an effect size of 0.34, indicating that the difference is statistically significant while the difference is also a positive improvement. These results demonstrate that the LLMs fine-tuned under various Non-IID scenarios, including even the extreme one, are confirmed to facilitate significant improvement. We also notice that all effect sizes except for the scenario of mildly heterogeneous code embedding, which yields an effect size of 0.24, fall into the range of medium effect size (i.e., $[0.28, 0.43)$), further validating that federated fine-tuning with either IID or Non-IID distributions leads to a promising increase in the repairing capability of LLMs across the evaluation metrics.

Finding 4: Federated fine-tuning with different data distributions, including the most extreme Non-IID scenario, is able to improve the repairing capabilities as significantly as the IID scenario, providing the insight that real-world industries can benefit from collaborative learning regardless of diverse data distributions.

On the other hand, contrary to most previous studies regarding federated learning on traditional DL tasks where Non-IID data mostly results in significant performance degradation compared to the ideal IID distribution [40, 51, 53], we find that fine-tuning LLMs with diverse Non-IID distributions is overall negligibly affected and in many cases even outperforming the IID scenario.

Firstly, we observe that fine-tuning under the Non-IID scenarios on specific LLMs exhibits a slight decline in performance compared to the IID scenario. For example, as presented in Table 5, we notice a maximal decrease of 6.14% for CodeLlama-13B on *Top*@10 in terms of fine-tuning with medium Non-IID distribution and a decrease of 2.46% on *Pass*@10 under the mild Non-IID scenario, compared to the IID distribution. In Table 6, we see that both CodeLlama-13B and CodeLlama-7B are slightly influenced by fine-tuning under some of the heterogeneous code complexity scenarios. There is a maximal decrease of only 3.68% on *Top*@5 while fine-tuning CodeLlama-13B under the extreme Non-IID scenario and a reduction of 6.75% on *Top*@10 while fine-tuning CodeLlama-7B with mild Non-IID distribution. The results in Table 7 show that the LLMs are more susceptible to heterogeneous distributions of code embedding, where all of the LLMs except DeepSeekCoder-7B exhibit slight declines in the performance across the three Non-IID scenarios.

However, the results also reveal that fine-tuning under Non-IID scenarios can outperform the IID scenario in many cases. In particular, Table 5 presents that the CodeLlama-7B model and DeepSeekCoder-7B model outperform the IID scenario under all Non-IID scenarios. Notably, in the best-performing scenario of DeepSeekCoder-7B (i.e., Medium), the model achieves a significant increase of 15.34% and 18.68% compared to the IID distribution. Mistral-7B and DeepSeekCoder-7B in the heterogeneous code complexity scenarios, as illustrated by Table 6, achieve the best repairing performance with all Non-IID distributions. The DeepSeekCoder-7B also achieves the largest increase at 19.02% and 18.40% on *Top*@5 and *Top*@10 under the extreme Non-IID scenario compared to the IID scenario from Table 7. These results indicate that while the LLMs can be slightly affected by some of the heterogeneous distributions, they still exhibit superior performance compared to the IID scenario in many cases.

To further confirm whether these effects are significant and to determine if they are overall positive or negative effects, as they have specific pros and cons over different scenarios, we also conduct significance tests between the Non-IID scenarios and the IID scenario. The p-values and effect sizes of the differences between the Non-IID distributions and the IID (i.e., from *Mild & IID* to *Extreme & IID*) are presented in Table 8.

Specifically, the results in Table 8 show that most of the differences between Non-IID scenarios and the IID scenario are not significant. The only exceptions are that the difference between the mildly heterogeneous code embedding scenario and the IID scenario yields a p-value of $9.22e^{-6}$ and the difference between the extremely heterogeneous coding style scenario and the IID scenario yields a p-value of 0.03. However, we notice their corresponding values of Cliff's delta are -0.11 and 0.06, which indicate negligible effects. Therefore, even the most significant differences have minimal practical impact on the repairing performance of the federated fine-tuning approach. In addition, the overall results across the three code features also yield p-values$\geq$0.05 with trivial effect sizes, further confirming that fine-tuning with heterogeneous code has negligible impact on the performance of bug fixing.

Finding 5: Different degrees of data heterogeneity from mild Non-IID to extreme Non-IID have negligible impact on the performance of LLM fine-tuning compared to IID data distribution, indicating that it is feasible for real-world industries holding datasets with either similar or distinct data distributions to collaborate in the context of LLM fine-tuning in federated learning.

Despite the fact that the overall impact of heterogeneous code is negligible, we find that WizardCoder-15B only achieves the best performance with the IID data distribution across all code features. While the WizardCoder-15 model can also benefit from fine-tuning with heterogeneous code, we notice that compared to the IID scenario, the Non-IID scenarios consistently cause degradation on the performance of WizardCoder-15B, particularly for the extremely heterogeneous coding style scenario where the *Pass*@10 decreases by up to 11.52%, as presented in Table 5. The results in Table 6 and Table 7 also demonstrate that WizardCoder-15B experiences a decline from over 70% to over 60% across the Non-IID scenarios, especially on *Top*@10 and *Pass*@10. These results indicate that WizardCoder-15B is more susceptible to heterogeneous code compared to the other LLMs.

It is also worth noting that CodeQWen-7B exhibits stable and consistent performance across all code features and data distributions. For example, we observe that CodeQWen-7B achieves a maximal improvement of 2.07% on *Pass*@10 under the extreme Non-IID scenario as presented by Table 5, and the greatest increase of 2.45% on *Top*@10 is achieved by fine-tuning with the medium Non-IID distribution as illustrated in Table 6. On the other hand, CodeQWen-7B is shown to be hardly affected by heterogeneous code compared to the IID distribution. In particular, we see that regardless of the increase or decrease in performance, the bug fixing capability of CodeQwen on the *top*@10 and *Pass*@10 metrics remains around 90% across all data distributions and code features, demonstrating remarkable stability.

Finding 6: Despite that the bug fixing capability of WizardCoder-15B can be enhanced through fine-tuning with various data distributions, it remains more susceptible to heterogeneous code compared to other models. In contrast, CodeQWen-7B demonstrates remarkable stability and consistency across all code features and data distributions, with minimal performance fluctuation compared to other LLMs.

5.3 RQ3: Impact of federated algorithms

[Experiment Goal for RQ3]: We investigate the impact of applying various types of federated algorithms optimized for different phases of federated learning on fine-tuning LLMs for program repair.
[Experiment Design for RQ3]: As described in the federated learning process in section 2.1, local fine-tuning and federated aggregation are the key components of federated learning. On the client side, each local device updates its model based on the local data towards a local objective, while on the server side, the updated parameters are further aggregated into a new global model according to a specific strategy. We employ various algorithms to comprehensively explore different types of federated algorithms based on the two sides. In addition, we extend our investigation to personalized learning, which is a different learning paradigm that takes into account the individual characteristics and preferences of each client.

Client-side Optimization. In the context of federated learning, client-side optimization plays a crucial role in enhancing the efficiency and effectiveness of local model training on participating devices. To establish a baseline for client-side optimization, we employ FedProx [53], a widely used federated algorithm specifically designed to optimize the local training process. FedProx introduces a proximal term in the local objective function, which serves to improve the quality of local models that may not have undergone sufficient training. We evaluate FedProx to investigate the role of client-side optimization during the federated fine-tuning process.

Server-side Optimization. Server-side optimization is crucial for aggregating updates from diverse client models into a stable global model. Therefore, we leverage FedSWA [11] as another baseline since it applies a stochastic weight averaging strategy for federated aggregation on the server side to promote better generalizability in federated learning. FedSWA enhances generalization by encouraging convergence toward flatter minima, which is beneficial for distributed data across clients. Extensive testing on various tasks validates its robustness across different applications. We evaluate it in the LLM-based federated fine-tuning setting to further investigate its effectiveness.

Optimization on Both Sides. We also evaluate the performance of FedOPT [80], which performs adaptive optimization on both the client side and server side. FedOPT employs adaptive optimization methods on the server and clients for faster convergence compared to traditional methods, robust to noisy gradients, and provides theoretical convergence guarantees that ensure reliable performance across scenarios. We evaluate FedOPT as another type of federated algorithm to assess its performance in comparison to other types of algorithms.

Personalized Learning. Furthermore, we take personalized federated learning into consideration since it has emerged as an alternative federated learning approach to further improve model performance in recent studies [52, 79, 88, 91]. The main difference from traditional federated learning is that personalized federated learning is able to handle client-specific characteristics. While federated learning focuses on training a single global model that performs well on average across all clients, personalized federated learning takes a step further by creating personalized models, such as retaining parts of the updated model parameters in each local client [52, 79], that adapt to the unique characteristics of each client. However, it still remains unclear whether LLM fine-tuning can benefit from personalized learning. Therefore, we leverage a typical and widely used algorithm pFedMe [88] as the personalization technique to evaluate the performance of LLMs fine-tuned for program repair. pFedMe utilizes Moreau envelopes as regularized loss functions to decouple personalized model updates from global model learning, allowing each client to optimize its personalized model based on local data [88].

We compare FedAvg [70], the foundational federated learning algorithm that aggregates locally trained models by averaging their parameters as described in Section 2.1, with the other four baselines to investigate the impact of different types of federated algorithms on LLM fine-tuning.

Algorithm Ranking. Given that different algorithms may exhibit varied performance across diverse LLMs and metrics, we use the Borda count method [81], which is one of the typical and most important voting rules, to evaluate the overall performance of the algorithms. With the comprehensive nature of Borda count in capturing the strength accurately of each candidate [8], we aim to obtain the aggregate ranking that best reflects the overall strengths of the algorithms since several metrics are used to assess each algorithm. Initially, each algorithm receives a Borda score based on their rankings across all voters (i.e., metrics). For example, in a selection with m candidates, the top-ranked candidate is assigned m points while the second-ranked gets $m-1$ points and 1 point for the last candidate. By summing these points across all metrics, the Borda count identifies a winner with broad support, rather than simply based on a majority of the number of first-place performance.

[Experimental Results for RQ3]: Table 9 presents the evaluation results of fine-tuning with different federated algorithms. We notice that FedAvg barely achieves the best performance but performs the second best on most of the metrics across all LLMs. As Table 9 shows, FedAvg only achieves the highest *Top*@10 of 79.75% on CodeLlama-13B and the highest *Top*@10 and *Pass*@10 of 71.78% and 68.54% on DeepSeekCoder-7B. Other than that, however, FedAvg reaches the second best performance for most of the other cases. For example, FedAvg performs only slightly inferior to FedSWA on WizardCoder-15B, and it achieves the second-best performance on *Top*@5, *Pass*@1, and *Pass*@5 on both CodeLlama-7B and Mistral-7B.

To further validate the effectiveness of the algorithms, we perform the typical Borda count voting algorithm to capture the overall strength of each candidate algorithm. The results of the algorithm ranking is presented in Figure 11. The voting results rank the federated algorithms as: **FedAvg > FedSWA > FedOPT > FedProx > pFedMe**. The results illustrate that FedAvg achieves the highest overall Borda count of 117 points among all

algorithms. This is also intuitive since FedAvg consistently achieves the second-best performances under most scenarios while others present varying performances on different LLMs. Therefore, despite that FedAvg does not achieve the best performance across each LLM, the results demonstrate that the overall performance of FedAvg remains promising, demonstrating its effectiveness as the foundational federated algorithm for aggregating knowledge from diverse clients for program repair.

Table 9. Experimental results of different federated learning algorithms on EvalRepair-Java.

Scenario	CodeLlama-13B					CodeLlama-7B					DeepSeekCoder-7B				
	Top@5	Top@10	Pass@1	Pass@5	Pass@10	Top@5	Top@10	Pass@1	Pass@5	Pass@10	Top@5	Top@10	Pass@1	Pass@5	Pass@10
FedAvg	65.64	**79.75**	31.76	67.49	77.71	57.67	66.87	28.41	60.35	**70.10**	48.47	**71.78**	15.24	49.12	**68.54**
FedProx	55.83	63.80	19.49	52.28	65.76	56.44	67.48	21.57	52.80	65.66	**50.31**	67.48	**16.99**	**51.15**	68.01
FedSWA	66.26	79.14	30.86	66.55	77.73	**58.90**	**71.17**	**29.45**	**60.57**	70.08	42.94	61.35	12.60	43.46	61.60
FedOPT	**67.48**	75.46	**36.95**	**69.15**	**78.12**	54.60	63.80	24.82	56.00	66.25	47.85	65.03	14.63	46.93	64.06
pFedMe	58.90	72.39	21.85	56.43	70.97	53.99	63.80	26.52	55.01	63.85	21.47	42.94	6.61	26.44	42.23
Scenario	WizardCoder-15B					Mistral-7B					CodeQWen-7B				
	Top@5	Top@10	Pass@1	Pass@5	Pass@10	Top@5	Top@10	Pass@1	Pass@5	Pass@10	Top@5	Top@10	Pass@1	Pass@5	Pass@10
FedAvg	54.60	66.87	22.70	56.35	67.78	52.15	65.03	22.27	52.39	64.37	87.73	92.64	58.90	86.65	91.80
FedProx	43.56	58.90	15.34	46.20	60.61	51.53	**66.26**	**22.89**	**53.18**	**65.37**	83.44	90.18	50.92	82.63	89.23
FedSWA	**61.35**	**71.17**	**25.29**	**58.70**	**69.70**	**52.76**	65.64	21.71	52.12	65.31	86.50	92.02	57.57	87.00	92.08
FedOPT	53.99	65.03	18.74	51.43	63.37	37.42	47.24	14.87	36.33	46.45	**88.34**	**94.48**	**63.10**	**88.14**	**93.50**
pFedMe	46.63	63.19	14.58	45.17	61.59	43.56	53.37	18.64	43.02	54.13	83.44	89.57	53.28	84.40	90.96

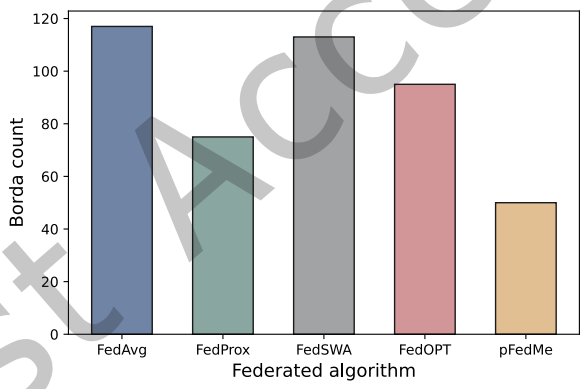


Fig. 11. The Borda count of different federated algorithms.

Finding 7: As the foundational and the most widely applied federated algorithm, FedAvg is rarely optimal in the bug fixing ability across all LLMs, but achieves the best overall performance instead, which demonstrates its stability and high practicality in terms of fine-tuning LLMs for program repair in federated learning.

At the same time, other algorithms do not exhibit consistency on each LLM and appear to exhibit varying performance across different LLMs. Among other types of federated algorithms we evaluated, FedSWA emerges as the top on CodeLlama-7B and WizardCoder-15B, achieving a *Top*10 of 71.17% on CodeLlama-7B and a *Pass*@10

of 69.70% on WizardCoder-15B. We also notice that it achieves the second-best performance on certain metrics across LLMs such as CodeLlama-13B, Mistral-7B, and CodeQWen-7B, highlighting the efficacy and potential of server-side optimization in improving the bug fixing capability of the LLMs in federated fine-tuning. In addition, FedSWA achieved an overall Borda count of 113 points, which is only 4 points lower than FedAvg, demonstrating a promising overall repairing ability.

On the other hand, despite FedOPT and FedProx not achieving comparable overall performance to FedAvg and FedSWA as presented in Figure 11, they still demonstrate promising results in specific scenarios. FedOPT showcases its superiority on CodeLlama-13B and CodeQWen-7B, outperforming the other algorithms in these cases. In particular, FedOPT achieves the best *Top*@10 and *Pass*@10 of up to 94.48% and 93.50% on CodeQWen-7B, which underscores the significance of employing optimization techniques on both the client and server sides for achieving optimal results with these models. Moreover, the results in Table 9 also reveal that FedProx demonstrates comparable performance on DeepSeekCoder-7B and Mistral-7B, achieving optimal results on some of the metrics. For example, FedProx achieves the highest *Top*@10 and *Pass*@10 of 66.26% and 65.37% on Mistral-7B, which emphasizes the effectiveness of client-side optimization in fine-tuning for certain LLMs. These results suggest that while FedOPT and FedProx may not be the best choice for all scenarios, they can still be valuable alternatives for specific LLMs and use cases.

Finding 8: Different types of federated algorithms excel in optimizing the performance of specific LLMs, highlighting the importance of considering the specific characteristics and requirements of each LLM when selecting the appropriate optimization approach. By tailoring the optimization process to the unique properties of each LLM, the fine-tuning performance can be effectively enhanced for program repair.

However, while different types of federated algorithms demonstrate specific strengths in different scenarios, the results show that the personalization algorithm pFedMe consistently underperforms compared to others in our evaluation. According to the overall ranking results in Figure 11, pFedMe only achieves 50 points of Borda count. As presented in Table 9, we notice that pFedMe causes devastating performance degradation on DeepSeekCoder-7B where the *Top*@10 and *Pass*@10 decrease by as much as 28.84% and 26.31% compared to FedAvg. It also leads to a reduction of 12.89% and 11.24% in terms of *Top*@10 and *Pass*@10 compared to FedProx on Mistral-7B. In addition, we also find that CodeQWen-7B still remains the least affected model, with a decrease of 4.91% and 2.54% on *Top*@10 and *Pass*@10 compared to FedOPT, aligning with the previously noted consistency of CodeQWen-7B. These observations on the personalization technique indicate that personalized learning for LLM fine-tuning remains a challenge and still requires further investigation.

Finding 9: Despite the potential benefits of personalized learning demonstrated in most traditional federated learning scenarios, the personalized federated algorithm exhibits the worst performance for the fine-tuning of LLMs, indicating that it remains a major challenge to adapt to LLM fine-tuning for program repair in the context of federated learning.

6 DISCUSSION

Impact of Dataset Size. According to the results of RQ1 in Section 5.1, we found that the local fine-tuning approach, which fine-tunes the LLMs on each client separately without cooperation, showed the worst performance throughout the evaluation, and the performance of centralized fine-tuning that leverages the complete data was surpassed by federated fine-tuning in some cases. The local fine-tuning approach updates the model

with only the local dataset within each client, whereas the full dataset is utilized for the centralized fine-tuning approach, suggesting that the size of the dataset plays a crucial role in the fine-tuning process. According to Jiang et al. [35], the size of a fine-tuning dataset appears to affect the performance of the LLMs. The results of Jiang's study demonstrate that fine-tuning with either too little or too much data can cause degradation in the model performance. While centralized fine-tuning has access to the full dataset and local fine-tuning only involves a fraction of the overall dataset, both fine-tuning methods may suffer from suboptimal performance due to the dataset size. The boundary of such size of the dataset that affects the fine-tuning of LLMs is unclear and remains to be further explored in future works.

On the other hand, the key point reflected in this issue is that while both local fine-tuning and centralized fine-tuning exhibit instability in fine-tuning the LLMs, federated fine-tuning combines the privacy-preserving benefits of local fine-tuning with the advantages of centralized fine-tuning that fully utilize valuable data, to achieve stable and consistent performance across most scenarios.

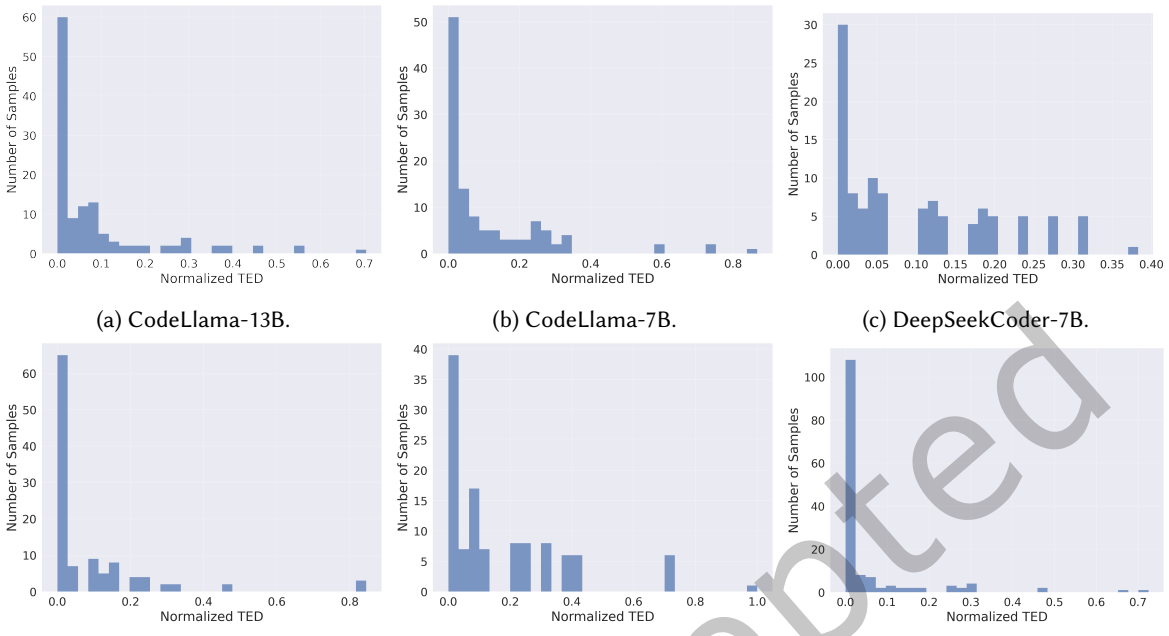
Impact of Code Features. Since we found that different degrees of Non-IID data on each type of code feature could barely affect the performance of federated fine-tuning, it is evident in our study that the performance of the fine-tuned LLMs was not affected across different code features either from an overall perspective. This suggests that companies having code repositories with either different coding styles, code complexities, or code embeddings are able to improve their models by federated fine-tuning under different data distributions, indicating that the benefits of federated fine-tuning have the potential to extend to industries across various domains.

Federated Algorithms. Our results also suggest that the FedSWA algorithm seems to be particularly well-suited for CodeLlama-based models, as it consistently performed well on CodeLLama-13B, CodeLLama-7B, and WizardCoder-15B, which are all built upon the CodeLlama base model. Future studies could explore a larger variety of models to understand the impact of the fundamental architectures of different LLMs.

Personalized Learning. We also found that personalization did not perform well for LLM fine-tuning in the context of program repair despite their success in other traditional federated learning tasks. This indicates that the unique characteristics of LLMs and the nature of the program repair task may require different approaches. Another difference is that we fine-tune the adapters rather than the original model, whereas in conventional federated learning tasks, the original model with full parameters is processed by personalization techniques. Further research is needed to investigate and develop personalization strategies specifically tailored for LLM fine-tuning on program repair or other code-related tasks to promote client-specific scenarios.

Patch Overfitting. Despite that the benchmark we use was enhanced by significantly improving the average number of test cases for each problem, prior research [119] has indicated that there remains a potential risk of overfitting by merely increasing the number of test cases. In contrast, the quality of the test cases is paramount in terms of mitigating patch overfitting. Therefore, we also conducted manual verification on the fixed code generated by LLMs to ensure the quality of the code. We follow the work of Yang et al. [116] to transform the code into abstract syntax trees (ASTs) and calculate the normalized Tree Edit Distance (TED) [25, 30, 45] between the fixed code and the ground-truth code to filter out the code that cannot align with the correct code (distance > 0). Figure 12 presents the distribution of normalized TED for each LLM. The results demonstrate that a large amount of code exactly matches the ground-truth where the distances are 0 and we only have to verify the remaining code samples.

Our manual verification reveals that there is only one overfitted fixed code for CodeLlama-13B, CodeLlama-7B and Mistral-7B, which is illustrated in Figure 13. This code aims to solve the problem of cumulative summation up to n. The fixed code on the left of Figure 13 implements the problem through an iterative process with time complexity $O(n)$ while the ground-truth code on the right solves it through the accumulation formula with time complexity $O(1)$. Since we checked the test cases for this problem and found that the maximal n in the test cases



(a) CodeLlama-13B. (b) CodeLlama-7B. (c) DeepSeekCoder-7B.

(d) WizardCoder-15B. (e) Mistral-7B. (f) CodeQWen-7B.

Fig. 12. Distribution of AST Distance between the fixed code and the ground-truth code.

is not large enough, the fixed code is only feasible for relatively small values of n, whereas the time cost for extremely larger values can be unacceptable.

```
public class SUM_TO_N {
      public static int sum_to_n(int n) {
            int sum = 0;
            for (int i=1; i<=n; i++) {
                  sum += i;
            }
            return sum;
      }
}
```

```
public class SUM_TO_N {
      public static int sum_to_n(int n) {
            return (n + 1) * n / 2;
      }
}
```

Fig. 13. Overfitted Fixed Code.

We also found that the rest fixed code was equivalent to the correct code in various ways. Figure 14 exhibits two equivalent pairs of code. In the first pair of code, the fixed code on the left assigns the value of *list.length* to the variable n, while the ground-truth code on the right uses *list.length* directly in the calculation. The second pair demonstrates different implementations of conditional statements that evaluate the value of n. Both cases contribute to large differences in the AST distances. The fixed code presented in Figure 15 is an example of the largest AST distances for different LLMs. We found that the LLMs would sometimes generate the unnecessary

code for the *main* function, which is redundant for the problem since the test cases only execute the solution function rather than the *main* function. However, the redundant code segments constitute a significant proportion of the large AST distances.

```java
public class MEDIAN {
    public static double median(int[] list) {
        Arrays.sort(list);
        int n = list.length;

        if (n % 2 == 1) {
            return list[n/2];
        } else {
            return (list[n/2-1] + list[n/2]) / 2.0;
        }
    }
}
```

```java
public class MEDIAN {
    public static double median(int[] list) {
        Arrays.sort(list);

        if (list.length % 2 == 1) {
            return list[(int)(list.length / 2)];
        } else {
            return (list[(int)(list.length / 2)] +
                    list[(int)(list.length / 2) - 1]) / 2.0;
        }
    }
}
```

```java
public class FIB {
    public static int fib(int n) {
        if (n == 0) {
            return 0;
        } else if (n == 1) {
            return 1;
        } else {
            return fib(n - 1) + fib(n - 2);
        }
    }
}
```

```java
public class FIB {
    public static int fib(int n) {
        if (n == 0 || n == 1)
            return n;

        return fib(n - 1) + fib(n - 2);
    }
}
```

Fig. 14. Equivalent Fixed Code.

Overall, the results demonstrate that the benchmark we use can effectively reduce the risk of patch overfitting, with a small subset of repair candidates being semantically equivalent to the ground-truth code.

Limitations and Generalizability. Our evaluation focused on models ranging from 7B to 15B parameters due to resource constraints. The findings may not fully extend to much larger models (such as those exceeding 70B parameters), which could exhibit different behaviors in federated settings. Additionally, while our cross-language setup (fine-tuning on C++ and evaluating on Java) demonstrates promising generalization, this might not extend equally well to all programming language pairs.

The approach could naturally extend to other code-related tasks such as code summarization, code translation, test generation, and defect prediction, which similarly benefit from privacy-preserving collaborative learning. The insights regarding resilience to code heterogeneity could inform federated learning applications in other domains with similar privacy concerns, such as healthcare or financial services. Beyond language models, our findings could inform the development of federated approaches for other model architectures used in software engineering, such as specialized graph neural networks for code representation or transformer-based models focusing on specific software artifacts such as documentation, UML diagrams, and commit messages.

7 THREATS TO VALIDITY

Threats to Internal Validity. One potential threat to internal validity arises from the code features. The code features used to construct the Non-IID data distributions may not fully capture the heterogeneity that exists in real-world code repositories across different industries. We utilize a private dataset sourced from a company providing online programming services. The variety of code features can be limited since it may not fully represent the diverse code repositories in other industries, such as game development, artificial intelligence, or scientific research, which can produce code repositories that mainly serve different purposes and use totally different fundamental frameworks and libraries. We construct data heterogeneity based on three different code features to mitigate this issue. In future studies, researchers should consider incorporating code repositories from diverse domains to further investigate the impact of heterogeneous collaboration.

```java
public class MAX_FILL {
    public static int max_fill(int[][] grid, int bucket_capacity) {
        int result = 0;
        for(int i = 0; i < grid.length; i++) {
            int row_capacity = 0;
            for(int j = 0; j < grid[i].length; j++) {
                row_capacity += grid[i][j];
            }
            result += (row_capacity + bucket_capacity - 1) / bucket_capacity;
        }
        return result;
    }

    public static void main(String args[]) {
        int grid1[][] = {
            {0, 0, 1, 0},
            {0, 1, 0, 0},
            {1, 1, 1, 1}
        };
        int grid2[][] = {
            {0, 0, 1, 1},
            {0, 0, 0, 0},
            {1, 1, 1, 1},
            {0, 1, 1, 1}
        };
        int gridea[][] = {
            {0, 0, 0},
            {0, 0, 0}
        };
        System.out.println(max_fill(grid1, 1));
        System.out.println(max_fill(grid2, 2));
        System.out.println(max_fill(gridea, 5));
    }
}
```

Fig. 15. Fixed with Redundant Code.

Table 10. Performance Variability of Federated Fine-tuning for CodeLlama-13B and DeepSeekCoder-7B over 5 runs.

Metric	CodeLlama-13B					DeepSeekCoder-7B				
	Top@5	Top@10	Pass@1	Pass@5	Pass@10	Top@5	Top@10	Pass@1	Pass@5	Pass@10
μ	64.54	77.91	36.59	68.52	77.54	48.47	69.94	15.98	50.89	69.56
σ	0.90	1.50	3.46	0.57	1.17	0.00	1.84	0.70	0.84	1.60

Another key internal validity threat arises from non-determinism inherent in our methodology, which could introduce variability in results. Sources of randomness in our study mainly include (1) LLM fine-tuning (e.g., random weight initialization), (2) code inference (e.g., temperature and top-k/p sampling during decoding). (3) hardware/software variability (e.g., randomness in GPU floating-point operation). In terms of the fine-tuning randomness, we set a fixed random seed during the fine-tuning phase for all LLMs. To evaluate the non-determinism the rest two sources may introduce, we conducted a small-scale study on CodeLlama-13B and DeepSeekCoder-7B based on the experimental configuration of RQ1 to analyze the variances of the results by repeating federated fine-tuning across 5 independent runs.

Table 10 presents the performance variability (i.e., the mean and standard deviation metrics) of the two studied LLMs. The results show that CodeLlama-13B demonstrates the highest standard deviation of 3.46% in *Pass*@1, while the *Pass*@5 performance shows minimal variation with a standard deviation of only 0.57%, indicating that the performance of CodeLlama-13B in generating correct code in only a single attempt is more sensitive to stochastic factors. In contrast, DeepSeekCoder-7B exhibits greater stability across the runs, with its maximum standard deviation reaching only 1.84% for *Top*@10. Notably, DeepSeekCoder-7B's *Top*@5 performance remains remarkably consistent across all five experimental runs, achieving zero standard deviation.

Overall, the observed variations across all metrics fall within acceptable bounds, indicating that the randomness inherent in our study does not significantly compromise the reliability of the results.

Threats to Construct Validity. While *Pass@k* provides an unbiased estimation of the model's performance, the choice of k values may introduce a threat to construct validity. We select the values of 5 and 10 for k from the perspective of developers [43, 73]. Since *Pass@k* calculates the expected value for the functional correctness, the evaluated results are able to more accurately reflect the true performance of the model as the value of k increases. However, generating more solutions for larger k values on each problem and each model during the inference phase of LLMs is computationally more expensive. To mitigate the bias of the results, we evaluate the fine-tuning methods on six LLMs across more than 1000 problems. Future work can further explore the potential of LLMs on program repair, given sufficient computational resources.

Threats to External Validity. A significant threat to external validity in APR research, including both traditional and learning-based approaches, is the potential for patch overfitting due to weak test suites. Patches that pass a limited set of test cases may still be incorrect when subjected to more comprehensive testing. Manual verification of generated patches is resource-intensive, particularly for large-scale problem sets. To mitigate this issue, we employ the EvalRepair [118] benchmark, which substantially enhances and expands the original HumanEval [17] test suites by incorporating test cases from EvalPlus [60]. This approach provides a more rigorous evaluation framework, enabling us to identify and filter out potentially faulty patches, thereby enhancing the generalizability and robustness of our fixes.

8 CONCLUSION

In this paper, we perform an empirical study on LLM-based federated fine-tuning for program repair, aiming to provide a comprehensive understanding of how federated learning can enhance LLM fine-tuning while preserving data privacy, demonstrating its effectiveness and practicality for both academia and industry. We evaluate federated fine-tuning across six code-related LLMs based on different architectures. In order to align with real-world scenarios, we leverage a private industrial dataset comprising 1239 programming problems to fine-tune the models. The fine-tuned models are further evaluated on an augmented program repair benchmark, which encompasses 163 programming problems and 583 test cases for each problem.

Our study investigates three key research questions to thoroughly examine the effectiveness and implications of LLM-based federated fine-tuning for program repair: (1) To validate the efficacy of federated fine-tuning for program repair, we compared the performance of federated fine-tuning with standard fine-tuning approaches and the original LLMs. The results demonstrate the competitive bug-fixing capabilities of federated fine-tuning, with federated fine-tuning even outperforming centralized fine-tuning in some cases, highlighting its ability to learn effectively from distributed codes. (2) We explore the impact of heterogeneous code on federated fine-tuning for program repair. We construct different degrees of feature-skewed Non-IID data, from mild scenarios to extreme scenarios, based on various code features, i.e., coding style, code complexity, and code embedding. We compare different distribution scenarios to explore the impact of data heterogeneity on LLM-based fine-tuning. Our findings revealed that data heterogeneity had minimal impact on the performance of LLM fine-tuning in the context of federated learning, contrary to traditional federated learning tasks. This insight suggests that enterprises

with diverse code repositories can still benefit from collaborative model training through federated learning without compromising performance. (3) We examine how different phases of optimization in federated learning influence the LLM fine-tuning for program repair. The optimization process in federated learning comprises several important stages, either on the client or server, which is critical to fine-tuning performance. We evaluate federated algorithms that specifically focus on client-side or server-side optimization. In addition, as another effective learning paradigm in federated learning, we also evaluate the effectiveness of personalized learning to further explore the impacts of different types of federated algorithms. Our results show that the widely used FedAvg algorithm demonstrated stability and practicality for LLM fine-tuning, whereas personalized learning algorithms like pFedMe faced challenges in adapting to the unique characteristics of LLMs and the program repair task, indicating the need for further research in developing personalization strategies specifically tailored for LLM fine-tuning.

This study sheds light on the potential of federated learning in facilitating collaborative software development and maintenance while preserving data privacy.

9 ACKNOWLEDGEMENT

This research was funded (partially) by the Australian Government through the Australian Research Council's Discovery Early Career Researcher Award, project number DE220101057, and by the NATURAL project, which has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation program (grant No. 949014).

REFERENCES

[1] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C Briand, and Thomas Stifter. 2020. Automated repair of feature interaction failures in automated driving systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 88–100.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[3] Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering*. 1443–1455.

[4] Vard Antinyan, Miroslaw Staron, and Anna Sandberg. 2017. Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. *Empirical Software Engineering* 22 (2017), 3057–3087.

[5] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[6] David Azcona, Piyush Arora, I-Han Hsiao, and Alan Smeaton. 2019. user2code2vec: Embeddings for profiling students based on distributional representations of source code. In *Proceedings of the 9th International Conference on Learning Analytics & Knowledge*. 86–95.

[7] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609* (2023).

[8] Niclas Boehmer, Robert Bredereck, and Dominik Peters. 2023. Rank aggregation using scoring rules. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5515–5523.

[9] CO Boulder. 2019. University of cambridge study: Failure to adopt reverse debugging costs global economy $41 billion annually. *Google Scholar Google Scholar Reference* (2019).

[10] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. 2013. Reversible debugging software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep* 229 (2013).

[11] Debora Caldarola, Barbara Caputo, and Marco Ciccone. 2022. Improving generalization in federated learning by seeking flat minima. In *European Conference on Computer Vision*. Springer, 654–672.

[12] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. 2015. De-anonymizing programmers via code stylometry. In *24th USENIX security symposium (USENIX Security 15)*. 255–270.

[13] Jialun Cao, Meiziniu Li, Ming Wen, and Shing-chi Cheung. 2023. A study on prompt design, advantages and limitations of chatgpt for deep learning program repair. *arXiv preprint arXiv:2304.08191* (2023).

[14] Tianshi Che, Ji Liu, Yang Zhou, Jiaxiang Ren, Jiwen Zhou, Victor S Sheng, Huaiyu Dai, and Dejing Dou. 2023. Federated learning of large language models with parameter-efficient prompt tuning and adaptive optimization. *arXiv preprint arXiv:2310.15080* (2023).
[15] Chaochao Chen, Xiaohua Feng, Jun Zhou, Jianwei Yin, and Xiaolin Zheng. 2023. Federated large language model: A position paper. *arXiv preprint arXiv:2307.08925* (2023).
[16] Haokun Chen, Ahmed Frikha, Denis Krompass, Jindong Gu, and Volker Tresp. 2023. FRAug: Tackling federated learning with Non-IID features via representation augmentation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 4849–4859.
[17] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[18] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin* 114, 3 (1993), 494.
[19] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2024. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems* 36 (2024).
[20] Zishuo Ding, Heng Li, Weiyi Shang, and Tse-Hsun Peter Chen. 2022. Can pre-trained code embeddings improve model performance? Revisiting the use of code embeddings in software engineering tasks. *Empirical Software Engineering* 27, 3 (2022), 63.
[21] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
[22] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*. 935–947.
[23] Tianyu Gao, Adam Fisch, and Danqi Chen. 2021. Making Pre-trained Language Models Better Few-shot Learners. 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)*, Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (Eds.). Association for Computational Linguistics, Online, 3816–3830. https://doi.org/10.18653/v1/2021.acl-long.295
[24] Waris Gill, Ali Anwar, and Muhammad Ali Gulzar. 2023. Feddebug: Systematic debugging for federated learning applications. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 512–523.
[25] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated clustering and program repair for introductory programming assignments. *ACM SIGPLAN Notices* 53, 4 (2018), 465–480.
[26] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).
[27] Sichong Hao, Xianjun Shi, Hongwei Liu, and Yanjun Shu. 2023. Enhancing Code Language Models for Program Repair by Curricular Fine-tuning Framework. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 136–146.
[28] Kevin Hsieh, Amar Phanishayee, Onur Mutlu, and Phillip Gibbons. 2020. The non-iid data quagmire of decentralized machine learning. In *International Conference on Machine Learning*. PMLR, 4387–4398.
[29] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685* (2021).
[30] Yang Hu, Umair Z Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. 2019. Re-factoring based program repair applied to programming assignments. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 388–398.
[31] Jonathan Huang. 2005. Maximum likelihood estimation of Dirichlet distribution parameters. *CMU Technique report* 76 (2005).
[32] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An empirical study on fine-tuning large language models of code for automated program repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1162–1174.
[33] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* (2023).
[34] Jingang Jiang, Xiangyang Liu, and Chenyou Fan. 2023. Low-parameter federated learning with large language models. *arXiv preprint arXiv:2307.13896* (2023).
[35] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1430–1442.
[36] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.
[37] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1646–1656.
[38] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*. 437–440.

[39] Peter Kairouz, Ziyu Liu, and Thomas Steinke. 2021. The distributed discrete gaussian mechanism for federated learning with secure aggregation. In *International Conference on Machine Learning*. PMLR, 5201–5212.
[40] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. 2021. Advances and open problems in federated learning. *Foundations and trends® in machine learning* 14, 1–2 (2021), 1–210.
[41] Katikapalli Subramanyam Kalyan. 2023. A survey of GPT-3 family large language models including ChatGPT and GPT-4. *Natural Language Processing Journal* (2023), 100048.
[42] Jinkyu Kim, Geeho Kim, and Bohyung Han. 2022. Multi-level branched regularization for federated learning. In *International Conference on Machine Learning*. PMLR, 11058–11073.
[43] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th international symposium on software testing and analysis*. 165–176.
[44] Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. 2020. Building implicit vector representations of individual coding style. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. 117–124.
[45] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering* 25 (2020), 1980–2024.
[46] Patrick Kreutzer, Georg Dotzler, Matthias Ring, Bjoern M Eskofier, and Michael Philippsen. 2016. Automatic clustering of code changes. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 61–72.
[47] Weirui Kuang, Bingchen Qian, Zitao Li, Daoyuan Chen, Dawei Gao, Xuchen Pan, Yuexiang Xie, Yaliang Li, Bolin Ding, and Jingren Zhou. 2024. Federatedscope-llm: A comprehensive package for fine-tuning large language models in federated learning. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 5260–5271.
[48] Jahnavi Kumar and Sridhar Chimalakonda. 2024. Code Summarization without Direct Access to Code-Towards Exploring Federated LLMs for Software Engineering. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 100–109.
[49] Xuan-Bach D Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. 2019. On reliability of patch correctness assessment. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 524–535.
[50] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256.
[51] Qinbin Li, Yiqun Diao, Quan Chen, and Bingsheng He. 2022. Federated learning on non-iid data silos: An experimental study. In *2022 IEEE 38th international conference on data engineering (ICDE)*. IEEE, 965–978.
[52] Tian Li, Shengyuan Hu, Ahmad Beirami, and Virginia Smith. 2021. Ditto: Fair and robust federated learning through personalization. In *International conference on machine learning*. PMLR, 6357–6368.
[53] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. 2020. Federated optimization in heterogeneous networks. *Proceedings of Machine learning and systems* 2 (2020), 429–450.
[54] Xiaoxiao Li, Meirui Jiang, Xiaofei Zhang, Michael Kamp, and Qi Dou. 2021. Fedbn: Federated learning on non-iid features via local batch normalization. *arXiv preprint arXiv:2102.07623* (2021).
[55] Zhen Li, Guenevere Chen, Chen Chen, Yayi Zou, and Shouhuai Xu. 2022. Ropgen: Towards robust code authorship attribution via automatic coding style transformation. In *Proceedings of the 44th International Conference on Software Engineering*. 1906–1918.
[56] Zijian Li, Zehong Lin, Jiawei Shao, Yuyi Mao, and Jun Zhang. 2024. Fedcir: Client-invariant representation learning for federated non-iid features. *IEEE Transactions on Mobile Computing* (2024).
[57] Wentao Liang, Xiang Ling, Jingzheng Wu, Tianyue Luo, and Yanjun Wu. 2023. A Needle is an Outlier in a Haystack: Hunting Malicious PyPI Packages with Code Clustering. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 307–318.
[58] Bo Lin, Shangwen Wang, Ming Wen, and Xiaoguang Mao. 2022. Context-aware code change embedding for better patch correctness assessment. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–29.
[59] Bill Yuchen Lin, Chaoyang He, Zihang Zeng, Hulin Wang, Yufen Huang, Christophe Dupuy, Rahul Gupta, Mahdi Soltanolkotabi, Xiang Ren, and Salman Avestimehr. 2021. Fednlp: Benchmarking federated learning methods for natural language processing tasks. *arXiv preprint arXiv:2104.08815* (2021).
[60] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).
[61] Xiao-Yang Liu, Rongyi Zhu, Daochen Zha, Jiechao Gao, Shan Zhong, Matt White, and Meikang Qiu. 2023. Differentially private low-rank adaptation of large language model using federated learning. *ACM Transactions on Management Information Systems* (2023).
[62] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. arXiv:1711.05101 [cs.LG] https://arxiv.org/abs/1711.05101

[63] Mi Luo, Fei Chen, Dapeng Hu, Yifan Zhang, Jian Liang, and Jiashi Feng. 2021. No fear of heterogeneity: Classifier calibration for federated learning with non-iid data. *Advances in Neural Information Processing Systems* 34 (2021), 5972–5984.
[64] Wenqiang Luo, Jacky Wai Keung, Boyang Yang, Tegawende F Bissyande, Haoye Tian, and Bach Le. 2025. Unlocking LLM Repair Capabilities in Low-Resource Programming Languages Through Cross-Language Translation and Multi-Agent Refinement. *arXiv preprint arXiv:2503.22512* (2025).
[65] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568* (2023).
[66] Lucy Ellen Lwakatare, Aiswarya Raj, Ivica Crnkovic, Jan Bosch, and Helena Holmström Olsson. 2020. Large-scale machine learning systems in real-world industrial settings: A review of challenges and solutions. *Information and software technology* 127 (2020), 106368.
[67] Xinge Ma, Jiangming Liu, Jin Wang, and Xuejie Zhang. 2023. FedID: Federated Interactive Distillation for Large-Scale Pretraining Language Models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 8566–8577.
[68] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. Sapfix: Automated end-to-end repair at scale. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 269–278.
[69] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying codebert for automated program repair of java simple bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 505–509.
[70] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*. PMLR, 1273–1282.
[71] Steinbach Michael. 2000. A comparison of document clustering techniques. In *KDD Workshop on Text Mining, 2000*.
[72] Do-Van Nguyen, Anh-Khoa Tran, and Koji Zettsu. 2022. FedProb: An aggregation method based on feature probability distribution for federated learning on non-IID data. In *2022 IEEE International Conference on Big Data (Big Data)*. IEEE, 2875–2881.
[73] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust enhancement issues in program repair. In *Proceedings of the 44th International Conference on Software Engineering*. 2228–2240.
[74] Yicheng Ouyang, Jun Yang, and Lingming Zhang. 2024. Benchmarking Automated Program Repair: An Extensive Study on Both Real-World and Artificial Bugs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 440–452.
[75] Rishov Paul, Md Mohib Hossain, Mohammed Latif Siddiq, Masum Hasan, Anindya Iqbal, and Joanna Santos. 2023. Enhancing automated program repair through fine-tuning and prompt engineering. *arXiv preprint arXiv:2304.07840* (2023).
[76] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. 2021. Program comprehension and code complexity metrics: An fmri study. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 524–536.
[77] Weiguo Pian, Yinghua Li, Haoye Tian, Tiezhu Sun, Yewei Song, Xunzhu Tang, Andrew Habib, Jacques Klein, and Tegawendé F Bissyandé. 2025. You Don't Have to Say Where to Edit! jLED–Joint Learning to Localize and Edit Source Code. *ACM Transactions on Software Engineering and Methodology* (2025).
[78] Weiguo Pian, Hanyu Peng, Xunzhu Tang, Tiezhu Sun, Haoye Tian, Andrew Habib, Jacques Klein, and Tegawendé F Bissyandé. 2023. MetaTPTrans: A meta learning approach for multilingual code representation learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5239–5247.
[79] Krishna Pillutla, Kshitiz Malik, Abdel-Rahman Mohamed, Mike Rabbat, Maziar Sanjabi, and Lin Xiao. 2022. Federated learning with partial model personalization. In *International Conference on Machine Learning*. PMLR, 17716–17758.
[80] Sashank J. Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečný, Sanjiv Kumar, and Hugh Brendan McMahan. 2021. Adaptive Federated Optimization. In *International Conference on Learning Representations*. https://openreview.net/forum?id=LkFG3lB13U5
[81] Jörg Rothe. 2019. Borda count in collective decision making: a summary of recent results. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 9830–9836.
[82] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
[83] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. 2015. Tricorder: Building a program analysis ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 598–608.
[84] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* (2023).
[85] Shriram Shanbhag and Sridhar Chimalakonda. 2022. Exploring the under-explored terrain of non-open source data for software engineering through the lens of federated learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1610–1614.
[86] André Silva, Sen Fang, and Martin Monperrus. 2023. Repairllama: Efficient representations and fine-tuned adapters for program repair. *arXiv preprint arXiv:2312.15698* (2023).

[87] Jingwei Sun, Ziyue Xu, Hongxu Yin, Dong Yang, Daguang Xu, Yiran Chen, and Holger R Roth. 2023. Fedbpt: Efficient federated black-box prompt tuning for large language models. *arXiv preprint arXiv:2310.01467* (2023).

[88] Canh T Dinh, Nguyen Tran, and Josh Nguyen. 2020. Personalized federated learning with moreau envelopes. *Advances in neural information processing systems* 33 (2020), 21394–21405.

[89] Min Tan, Yinfu Feng, Lingqiang Chu, Jingcheng Shi, Rong Xiao, Haihong Tang, and Jun Yu. 2023. FedSea: Federated Learning via Selective Feature Alignment for Non-IID Multimodal Data. *IEEE Transactions on Multimedia* (2023).

[90] Xunzhu Tang, Zhenghan Chen, Kisub Kim, Haoye Tian, Saad Ezzini, and Jacques Klein. 2023. Just-in-Time Security Patch Detection–LLM At the Rescue for Data Augmentation. *arXiv preprint arXiv:2312.01241* (2023).

[91] Xueyang Tang, Song Guo, and Jingcai Guo. 2021. Personalized federated learning with contextualized generalization. *arXiv preprint arXiv:2106.13044* (2021).

[92] Xunzhu Tang, Kisub Kim, Yewei Song, Cedric Lothritz, Bei Li, Saad Ezzini, Haoye Tian, Jacques Klein, and Tegawendé F. Bissyandé. 2024. CodeAgent: Autonomous Communicative Agents for Code Review. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 11279–11313. https://doi.org/10.18653/v1/2024.emnlp-main.632

[93] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is ChatGPT the ultimate programming assistant–how far is it? *arXiv preprint arXiv:2304.11938* (2023).

[94] Haoye Tian, Xunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and Tegawendé F Bissyandé. 2022. Is this change the answer to that problem? correlating descriptions of bug and code changes for evaluating patch correctness. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[95] Chih-Kai Ting, Karl Munson, Serenity Wade, Anish Savla, Kiran Kate, and Kavitha Srinivas. 2023. CodeStylist: a system for performing code style transfer using neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 16485–16487.

[96] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[97] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 1–29.

[98] Pablo Villalobos, Anson Ho, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, and Marius Hobbhahn. 2024. Position: Will we run out of data? Limits of LLM scaling based on human-generated data. In *Forty-first International Conference on Machine Learning*. https://openreview.net/forum?id=ViZcgDQjyG

[99] Boxin Wang, Yibo Jacky Zhang, Yuan Cao, Bo Li, H Brendan McMahan, Sewoong Oh, Zheng Xu, and Manzil Zaheer. 2023. Can Public Large Language Models Help Private Cross-device Federated Learning? *arXiv preprint arXiv:2305.12132* (2023).

[100] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated patch correctness assessment: How far are we?. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 968–980.

[101] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 1069–1088.

[102] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 172–184.

[103] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. 2007. How long will it take to fix this bug?. In *fourth international workshop on mining software repositories (MSR'07: ICSE Workshops 2007)*. IEEE, 1–1.

[104] Eliane S Wiese, Anna N Rafferty, Daniel M Kopta, and Jacqulyn M Anderson. 2019. Replicating novices' struggles with coding style. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 13–18.

[105] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics: Methodology and distribution*. Springer, 196–202.

[106] Herbert Woisetschläger, Alexander Erben, Shiqiang Wang, Ruben Mayer, and Hans-Arno Jacobsen. 2024. Federated fine-tuning of llms on the very edge: The good, the bad, the ugly. In *Proceedings of the Eighth Workshop on Data Management for End-to-End Machine Learning*. 39–50.

[107] Fangzhou Wu, Qingzhao Zhang, Ati Priya Bajaj, Tiffany Bao, Ning Zhang, Ruoyu Wang, Chaowei Xiao, et al. 2023. Exploring the Limits of ChatGPT in Software Security Applications. *arXiv preprint arXiv:2312.05275* (2023).

[108] Yonghao Wu, Zheng Li, Jie M Zhang, and Yong Liu. 2024. ConDefects: A Complementary Dataset to Address the Data Leakage Concern for LLM-Based Fault Localization and Program Repair. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 642–646.

[109] Yue Wu, Shuaicheng Zhang, Wenchao Yu, Yanchi Liu, Quanquan Gu, Dawei Zhou, Haifeng Chen, and Wei Cheng. 2023. Personalized federated learning under mixture of distributions. In *International Conference on Machine Learning*. PMLR, 37860–37879.
[110] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2022. Practical program repair in the era of large pre-trained language models. *arXiv preprint arXiv:2210.14179* (2022).
[111] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.
[112] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 959–971.
[113] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. 2023. Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244* (2023).
[114] Runhua Xu, Nathalie Baracaldo, Yi Zhou, Ali Anwar, and Heiko Ludwig. 2019. Hybridalpha: An efficient approach for privacy-preserving federated learning. In *Proceedings of the 12th ACM workshop on artificial intelligence and security*. 13–23.
[115] Aidan ZH Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. 2024. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.
[116] Boyang Yang, Haoye Tian, Weiguo Pian, Haoran Yu, Haitao Wang, Jacques Klein, Tegawendé F Bissyandé, and Shunfu Jin. 2024. CREF: an LLM-based conversational software repair framework for programming tutors. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 882–894.
[117] Boyang Yang, Haoye Tian, Jiadong Ren, Shunfu Jin, Yang Liu, Feng Liu, and Bach Le. 2025. Enhancing Repository-Level Software Repair via Repository-Aware Knowledge Graphs. *arXiv preprint arXiv:2503.21710* (2025).
[118] Boyang Yang, Haoye Tian, Jiadong Ren, Hongyu Zhang, Jacques Klein, Tegawendé F Bissyandé, Claire Le Goues, and Shunfu Jin. 2024. Multi-Objective Fine-Tuning for Enhanced Program Repair with LLMs. *arXiv preprint arXiv:2404.12636* (2024).
[119] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 831–841.
[120] Xiyuan Yang, Wenke Huang, and Mang Ye. 2023. Dynamic personalized federated learning with adaptive differential privacy. *Advances in Neural Information Processing Systems* 36 (2023), 72181–72192.
[121] Yanming Yang, Xing Hu, Zhipeng Gao, Jinfu Chen, Chao Ni, Xin Xia, and David Lo. 2024. Federated Learning for Software Engineering: A Case Study of Code Clone Detection and Defect Prediction. *IEEE Transactions on Software Engineering* (2024).
[122] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. Automated classification of overfitting patches with statically extracted code features. *IEEE Transactions on Software Engineering* 48, 8 (2021), 2920–2938.
[123] Wei Yuan, Quanjun Zhang, Tieke He, Chunrong Fang, Nguyen Quoc Viet Hung, Xiaodong Hao, and Hongzhi Yin. 2022. CIRCLE: continual repair across programming languages. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*. 678–690.
[124] Linan Yue, Qi Liu, Yichao Du, Weibo Gao, Ye Liu, and Fangzhou Yao. 2023. Fedjudge: Federated legal large language model. *arXiv preprint arXiv:2309.08173* (2023).
[125] Biao Zhang, Zhongtao Liu, Colin Cherry, and Orhan Firat. 2024. When Scaling Meets LLM Finetuning: The Effect of Data, Model and Finetuning Method. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=5HCnKDeTws
[126] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A survey of learning-based automated program repair. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–69.
[127] Quanjun Zhang, Chunrong Fang, Weisong Sun, Yan Liu, Tieke He, Xiaodong Hao, and Zhenyu Chen. 2024. Appt: Boosting automated patch correctness prediction via fine-tuning pre-trained models. *IEEE Transactions on Software Engineering* (2024).
[128] Quanjun Zhang, Chunrong Fang, Yang Xie, YuXiang Ma, Weisong Sun, and Yun Yang Zhenyu Chen. 2024. A Systematic Literature Review on Large Language Models for Automated Program Repair. *arXiv preprint arXiv:2405.01466* (2024).
[129] Ziqi Zhang, Yuanchun Li, Bingyan Liu, Yifeng Cai, Ding Li, Yao Guo, and Xiangqun Chen. 2023. FedSlice: Protecting Federated Learning Models from Malicious Participants with Model Slicing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 460–472.
[130] Jujia Zhao, Wenjie Wang, Chen Xu, Zhaochun Ren, See-Kiong Ng, and Tat-Seng Chua. 2024. Llm-based federated recommendation. *arXiv preprint arXiv:2402.09959* (2024).
[131] Fei Zheng. 2023. Input reconstruction attack against vertical federated large language models. *arXiv preprint arXiv:2311.07585* (2023).
[132] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 5673–5684.
[133] Armin Zirak and Hadi Hemmati. 2024. Improving automated program repair with domain adaptation. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–43.