Transfer Learning Based Automated Debugging Across Multiple Programming Languages

Aabha Pingle
apingle@usc.edu

Dattateja Reddy Anakala
anakala@usc.edu

Parth Sanjay Shelar
pshelar@usc.edu

Prerana Shenoy Sedimane Prakasha
sedimane@usc.edu

Sharan Murli
murli@usc.edu

Abstract

C/C++ is widely used in developing embedded systems and Internet of Things (IoT) devices due to its ability to work closely with hardware components. Large Language Models (LLMs) have shown significant improvement in learning and understanding code and have the potential to debug or correct buggy code. Recent advancements in code debugging have been achieved in languages like Java and Python. These models do not perform well on C/C++ through prompting alone and will need more powerful techniques to understand the intricacies of C/C++. This research presents RepairCLlama, a novel model fine-tuned on RepairLLaMA for debugging and correcting C/C++ code. Our evaluation across multiple datasets demonstrates that RepairCLlama achieves higher accuracy compared to existing baseline models. We provide an End-to-End Pipeline that includes data pre-processing, model fine-tuning, inference testing and evaluation metrics specifically designed to address the challenges of C/C++ code debugging. To facilitate further research in the C/C++ code debugging domain, we have made the datasets, models and code publicly available at https://github.com/aabhapingle/ANLP_Project. .

1 Introduction

We identify generating correct code as a crucial task in natural language processing (Dinh et al., 2023). A significant amount of research on code generation is concentrated on common languages such as Java (Just et al., 2014) and Python, given their widespread popularity and syntactic simplicity (Rozière et al., 2024; Jimenez et al., 2024). Furthermore, multilingual transfer learning for programming languages has emerged as an important area of study enabling models to generalize across multiple languages (Athiwaratkun et al., 2023; Li et al., 2024).

In the automatic program repair task, buggy code is input into a large language model (LLM) to identify errors and generate debugged and corrected code as output. This task requires addressing both semantic and syntactic bugs. For this purpose, we utilize RepairLLaMA (Silva et al., 2024), a model fine-tuned on CodeLlama, specifically for debugging Java code bugs. We focus on evaluating datasets composed of code snippets along with test cases, each containing a single function code. By exposing the model to examples of C/C++ code, RepairLLaMA can leverage the patterns it has already learned from Java to better adapt to the syntactic and semantic complexities of C/C++.

2 Related Work

The repetitive and complex nature of debugging within a professional environment demands significant developer time and effort, as highlighted by (Alaboudi and Latoza, 2021). Recent advancements in Large Language Models(LLMs) have shown tremendous potential in overcoming these challenges by improving code generation quality and automating code debugging. Rozière et al. (2024) present CodeLlama, an LLM trained on large code datasets, designed to support infilling with surrounding content along with code generation. CodeLlama shows strong cross-lingual abilities, performing well on various benchmarks like HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). These findings show that LLMs are flexible for zero-shot and multilingual code debugging across various programming languages. Li (2021) introduce PLATO, a framework that applies transfer learning to extend program analysis by using a series of downstream tasks. It allows BERT-based models to leverage prior knowledge learned from the labelled dataset of one programming language and transfer it to the others. By leveraging syntax enhancement, it preserves language struc-

ture and rules, allowing it to generalize learned features across programming languages. This framework shows that cross-lingual transfer works well for debugging, allowing LLM-based debugging across multiple programming languages. Similarly, we have utilized a pretrained LLM (trained for Java code correction) for code debugging and correction of another language like C/C++ through transfer learning.

3 Hypothesis

We hypothesize that RepairLLaMA, pretrained for debugging Java, can be fine-tuned for debugging tasks in C/C++, demonstrating cross-linguistic transferability in error detection and correction. We further expect that applying transfer learning will enable the model to outperform baseline models specifically for C/C++ code. We hypothesize that using more advanced metrics, such as CodeBLEU, will provide a holistic understanding of our model's performance compared to baseline models in code debugging tasks.

4 Methodology

We propose an approach to fine-tune RepairLLaMA for debugging C/C++ code to adapt and learn its syntactic and semantic structure. The buggy C/C++ code is initially preprocessed by converting the code into a single-function snippet and removing comments to prevent potential confusion. The model then processes this preprocessed code as input, learning C/C++ specific syntax and semantics through an end-to-end pipeline, as illustrated in Figure 1. Our approach leverages the benefits of transfer learning for Automated Program Repair (APR) of C/C++ programs, which significantly reduces computational cost and time compared to training a large language model from scratch. We also conducted additional experiments with various prompts provided along with the code. By utilizing prompt engineering techniques, we identified the most effective ways to structure prompts along with code, thereby enhancing the model's performance. Furthermore, we evaluated our approach using several metrics, including CodeBLEU (Ren et al., 2020), Plausible Match, and Abstract Syntax Tree (AST)-based evaluations. To thoroughly assess the adaptability of our approach, we performed two separate fine-tuning setups: one using the Deepfix dataset, and another with the TEGCER dataset.

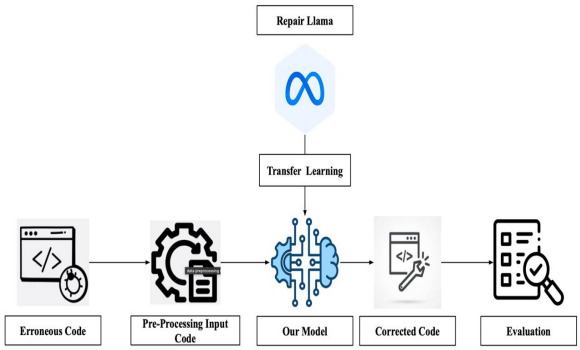


Figure 1: End-to-End Pipeline

4.1 Baseline Models

To establish a strong foundation for our research, we have utilized two baseline models: CodeLlama and RepairLLaMA. These models are designed for code generation and automated program repair(APR) respectively. CodeLlama, based on Meta's LLaMA 2 architecture, is trained on a massive dataset of code comprising of different programming languages namely Python, C++, Java, PHP, TypeScript, Csharp and Bash. This extensive training enables the model to perform a range of code-related tasks effectively, including generating syntactically correct code and understanding complex code structures. RepairLLaMA, an extension of CodeLlama, is optimized for automated program repair in Java by integrating Low-Rank Adaptation (LoRA) (Hu et al., 2021) techniques to enhance its debugging performance across multiple lines of code. With the help of these powerful models, we have a robust baseline that sets a strong benchmark for code debugging across different programming languages. This approach aims to provide a comprehensive analysis of our model's efficacy in the context of code debugging tasks. Both CodeLlama and RepairLLaMA are publicly accessible as open-source models, ensuring reproducibility and enabling other researchers to build upon our work. By employing these readily accessible baselines, we can effectively measure how Java-based APR models transfers it's syntactic and semantic patterns to C/C++ debugging scenarios.

4.2 Datasets

Our baseline model, RepairLLaMA, is fine-tuned on the Megadiff (Monperrus et al., 2021) dataset, a large collection of Java source code changes specifically focusing on single-function bug fixes. This fine-tuning allows the model to target error patterns

within a single function, improving its capability in localized bug repair. We utilize these datasets for our initial inference setup and further fine-tuning to evaluate our project's transfer learning capabilities: Deepfix (Gupta et al., 2017) is a dataset consisting of 6,147 erroneous C programs, primarily drawn from student submissions across 93 programming tasks. TEGCER (Ahmed et al., 2019) is an automated feedback tool for novice programmers. The dataset is curated from the student code submissions for an introductory C programming course. It has submissions from 40+ programming assignments and contains more than 20000 buggy and correct program pairs for APR tasks. ErrorCLR (Han et al., 2023) is a new dataset for buggy programs from student submissions utilized for semantic error classification, localization, and repair. HumanEvalPack (Muennighoff et al., 2023) is an extension of OpenAI's HumanEval and it covers 6 programming languages across 3 software engineering tasks namely code generation, translation, and fault localization. The dataset includes C++ data along with test cases. We selected these datasets due to the availability of test cases and wide range of syntactic and semantic errors. ErrorCLR and HumanEvalPack, in particular, provide test cases that allow us to validate and evaluate plausible matches. Additionally, all four datasets focus on single-function bugs, ensuring a localized transfer of debugging knowledge from Java to C/C++. We fine-tuned RepairLLaMA on the TEGCER and Deepfix datasets, which contain C buggy code and corresponding correct code. The details of these datasets are shown in Table 1a and 1b.

Dataset	Language	Number of examples	Avg number of lines of code
Deepfix	C	6147	34.42
TEGCER	C	21995	28.91
ErrorCLR	C/C++	8511	15.43
HumanEvalPack	C++	164	10.61

(a) Exploratory Data Analysis on the Dataset

Dataset	Test Cases	Single/multi function	Error Classification
Deepfix	✗	Single:4092 , Multi:2055	Syntactic
TEGCER	✗	Single: 14290, Multi: 7705	Semantic/Syntactic
ErrorCLR	✓	Single: 8498, Multi: 13	Semantic
HumanEvalPack	✓	Single: 138, Multi: 26	Semantic

(b) Trends Identified in the Dataset

Table 1: Dataset Characteristics and Identified Trends

4.2.1 Data Preprocessing

The Deepfix dataset, originally stored in a SQLite database, was converted into a CSV format by extracting program pairs. Entries with an error count greater than 0 were classified as buggy code, while those with an error count equal to 0 were treated as corrected code. For cases with multiple corrected codes, a similarity-based comparison was performed using the `SequenceMatcher` function to measure text similarity between the buggy code and each corrected candidate code. The corrected code with the highest similarity score to the buggy code was selected.

The TEGCER dataset, originally in CSV format, was curated by removing entries with identical buggy and corrected code through line-by-line comparison to retain only meaningful program corrections. For both datasets, the final step involved filtering programs to retain only single-function code to ensure a focused analysis of isolated function-level errors using regular expression to detect function definitions with common C return types (e.g., int, void, char).

For our fine-tuning setup with TEGCER, we use only the first 10,000 samples from the dataset for training and validation. For the setup with Deepfix, we are using all the 4092 samples. In both cases, the data is split such that 80% of the samples are randomly selected for training, while the remaining 20% are used for validation. From the dataset, only the buggyCode and correctedCode columns are retained, serving as input-output pairs for the training process. For tokenization, we employed the Hugging Face tokenizer. Input sequences exceeding the maximum length of 1024 tokens were truncated, while shorter sequences were padded to the maximum length to ensure uniformity in sequence length during training.

4.3 Algorithm

Low-Rank Adaptation (LoRA) is a parameter-efficient fine-tuning technique that reduces the number of trainable parameters by injecting low-rank matrices into pre-trained models. In our project, LoRA is applied to fine-tune the RepairLLaMA model for automated program repair (APR) in C/C++.

4.3.1 Mathematical Framework

In Transformer-based architectures, a weight matrix W in a dense layer maps inputs X to outputs Y:

$$Y = XW \tag{1}$$

LoRA reparametrizes W as:

$$W = W_0 + \Delta W \tag{2}$$

where:

- W_0 represents the pre-trained weights (frozen during fine-tuning).
- ΔW is the weight update, approximated by a low-rank decomposition:

$$\Delta W = BA \tag{3}$$

- $B \in R^{d \times r}$ and $A \in R^{r \times k}$ are trainable matrices.
- r is the rank of the adaptation matrices ($r \ll d, k$).

The forward computation becomes:

$$Y = XW_0 + XBA \tag{4}$$

During fine-tuning, only A and B are optimized, while W_0 remains unchanged.

4.3.2 Target Modules in the Transformer

LoRA is applied to specific attention projection matrices in the Transformer architecture:

- **Query Projection** (W_q): Projects inputs into query vectors for attention computation.
- **Key Projection** (W_k): Projects inputs into key vectors for attention computation.
- **Value Projection** (W_v): Projects inputs into value vectors for attention computation.
- **Output Projection** (W_o): Combines attention outputs.

This selective adaptation ensures task-specific learning while retaining the generalization capability of the frozen pre-trained weights.

5 Experiments

In this section, we describe the experiments conducted to evaluate the performance of RepairLLaMA and our fine-tuned model RepairCLlama. We perform a series of inference and fine-tuning experiments to analyze the model's debugging capabilities across C, C++ and Java code. The experiments are structured into three parts: inference experiments to assess the model's zero-shot and one-shot prompting abilities, error analysis to identify strengths and weaknesses of the model among different type of errors, and fine-tuning experiments to improve model performance using TEGCER and Deepfix datasets. The results are evaluated using a range of metrics, including train loss, validation loss, CodeBLEU, AST and semantic match scores.

5.1 Inference Experiments

For inference testing, we use the RepairLLaMA setup with AutoTokenizer (Wolf et al., 2020) and AutoModelForCausalLM from CodeLlama (trained on 7 billion parameters), fine-tuned with LoRA (Low-Rank Adaptation) for efficient parameter tuning. This configuration includes PeftModelForCausalLM, which is based on LoraModel with LlamaForCausalLM as its core architecture. The model is structured with 32 LlamaDecoderLayers, each containing a self-attention mechanism (LlamaSdpaAttention) and a multilayer perceptron (LlamaMLP).

For this experiment, we utilized the CodeLlama-7b-Instruct-hf model, the instruction-tuned version of CodeLlama, which is optimized for better comprehension and execution of natural language instructions. We noted that the CodeLlama model was performing well with instructions in the input prompt since it is the instruct version. However, since RepairLLaMA was fine-tuned over CodeLlama-7b-hf (the non-instruct version), the model wasn't working well with any English prompts in the input.

We conducted initial experiments using zero-shot and one-shot prompting techniques on the automatic code repair task. CodeLlama was evaluated in both prompting modes; however, it primarily returned unmodified input, as it lacks fine-tuning for debugging tasks. Subsequently, we attempted one-shot prompting with RepairLLaMA, but it yielded incomplete code due to its rigidity with prompts, requiring the input to be exclusively buggy code snippets. We observed that in codes containing multiple syntactic errors, only one error was localised and fixed by the model. Our evaluations, based on inference generation and metric analysis, were conducted on 15 examples across Java, C and C++ languages, as shown in Table 4. These observations underscore the need for fine-tuning RepairLLaMA on our C/C++ debugging dataset for more effective code repair outcomes.

5.2 Error Analysis

To better understand RepairLLaMA's strengths and weaknesses, we evaluated its performance across several error categories (see Table 2). We analyzed how effectively the model fixed different types of coding errors. The model excelled at correcting straightforward issues, such as operator misuse, which involves replacing incorrect operators (e.g.,

= instead of == or a misplaced arithmetic operator like + or -). Operator misuse is a syntactic error because it violates the grammar rules of the language but is relatively easy for the model to detect and fix. The model also showed reasonable capability in addressing basic logical errors, such as incorrect conditions in if-else statements. However, it struggled with more complex errors, especially in C/C++, where the code involved multiple functions or interdependent logic. These errors often included deeper semantic inconsistencies where the code compiles but does not behave as intended due to logical flaws. A few examples include incorrect loop conditions leading to off-by-one errors, misuse of variables (e.g., assigning or using the wrong variable), or passing incorrect arguments to functions.

Although RepairLLaMA can manage simpler issues, it requires further refinement to effectively address more intricate, multi-function debugging challenges, particularly in C/C++.

Error Type	Java	C++	C
Operator Misuse	✓	✓	✓
Missing Logic	✓	✗	✗
Variable Misuse	✓	✗	✗
Multi-Function Error	✗	✗	✗

Table 2: Error Analysis - Prompting RepairLLaMA

5.3 Fine-tuning Experiments

This section represents a comprehensive evaluation of our proposed model on the TEGCER and Deepfix datasets. The following three fine-tuning experiments were conducted: First, we fine-tuned the model using only the TEGCER dataset. Second, we fine-tuned the model using only the Deepfix dataset. Finally, we fine-tuned the model first on the TEGCER dataset and then on the Deepfix dataset. We experimented with various combinations of learning rates, max steps, epochs, and optimizers to identify the optimal fine-tuning configuration.

Specifically, we tested the following settings:

- **Max Steps**: 1000, 2000, and 4000
- **Epochs**: 5 and 10
- **Learning Rates**: 2e-5 and 5e-4
- **Optimizers**: Adam and AdamW Fused

For all experiments, we used a batch size of 4. The results, as shown in Table 3, indicate that the best performance was achieved with 4000 steps, 5 epochs, a learning rate of 2e-5, and the Adam optimizer.

5.3.1 Fine-tuning using LoRA

In our approach, we apply LoRA (Hu et al., 2021) to a 7B parameter RepairLLaMA model. We specify target modules (`q_proj, v_proj, k_proj, o_proj`) to adapt attention projection matrices according to RepairLLaMA. We set the LoRA rank (`r`) to 32 and a LoRA dropout of 0.1 to balance training stability and model capacity. These configurations were chosen based on insights from prior literature and initial experiments to optimize performance on the code debugging task. After training, we merge the LoRA-adapted weights back into the base model, resulting in a single set of parameters for inference, while preserving the benefits of the parameter-efficient training process.

5.3.2 Fine-tuning with prompts

Inspired by a version of CodeLlama, we experimented with using prompts alongside code to guide the model in code debugging, mimicking human debugging practices. For example, we concatenated prompts like "correct the above buggy code" in the model's input, where the buggy code was provided. This approach yielded significantly better performance compared to the prompt-less setup, particularly in terms of semantic match metrics.

5.3.3 Training Procedure and Hyperparameters

Training was conducted using the Hugging Face Trainer API, which streamlined the process of gradient accumulation, mixed-precision computation (`fp16`), and evaluation scheduling. We adopted a cosine learning rate scheduler with a warmup ratio of 0.02, allowing the optimizer to gradually adjust from a low learning rate to the target value.

To monitor the model's progress, we set the `save_steps` and `eval_steps` intervals to 100 steps for both dataset settings. We also enabled the `load_best_model_at_end` option to ensure that the final parameters (Table 3) corresponded to the best validation performance observed during training. These frequent evaluations and model checkpoints, combined with a small save total limit, helped prevent overfitting and reduced excessive checkpoint storage.

Dataset	Max Steps	Learning Rate	Train Loss	Validation Loss
TEGCER	4000	2e-5	0.2715	0.2744
TEGCER	2000	5e-4	0.4923	0.5383
Deepfix	2000	2e-5	0.7101	0.7046
Deepfix	2000	5e-4	0.6297	0.8294

Table 3: Fine-tuning Experiments - Parameters

5.4 Fine-tuned Model Inference Setup

In this section, we describe the inference setup for generating patches using the fine-tuned RepairLLaMA model. The model, fine-tuned with LoRA adapters on C/C++ code, was loaded alongside the tokenizer for inference. We employed the CodeLlama-7b-hf base model, quantized using 8-bit precision with a specified BitsAndBytesConfig to optimize memory efficiency.

The inference process begins with providing the model a buggy code snippet structured in a prompt-based format. The prompt guides the model to generate corrected patches, simulating the step-by-step process of human debugging.

To improve the quality and diversity of the generated patches, we utilized the beam search algorithm with a beam size of 10, which allows the model to explore multiple candidate solutions in parallel and select the most likely outputs. Additionally, early stopping was enabled to terminate generation when an optimal patch was produced, ensuring computational efficiency. We also configured the model to generate 10 output sequences for each buggy code input, which increases the chances of producing correct or plausible fixes.

The generated patches are post-processed by decoding the outputs, filtering out unnecessary tokens, and preparing them for evaluation. These patches are then analyzed to assess their correctness to the given buggy code.

This setup ensures that the fine-tuned RepairLLaMA model produces multiple candidate solutions efficiently, leveraging its fine-tuned capabilities for C/C++ code correction.

6 Evaluation metrics

We selected the following evaluation metrics, which are well-suited for code evaluation tasks such as code debugging. Each metric assesses a distinct aspect of the code generated by the model, enabling a comprehensive and accurate evaluation of its performance.

6.1 CodeBLEU

CodeBLEU (Ren et al., 2020) is a weighted combination of n-gram match (BLEU), Abstract Syntax Tree match, weighted n-gram match (BLEU-weighted), and data-flow match scores. This metric has shown a higher correlation with human evaluation than BLEU and accuracy metrics for code-related comparison. The results shown in Table 4 are evaluated with equal weights for each of the above metrics.

6.2 Abstract Syntax Tree (AST)

Abstract Syntax Trees match involves converting both the corrected code and the gold standard code provided by the dataset into abstract syntax trees and applying AST differencing (Falleri et al., 2014) to compare the two. It highlights the syntactic similarity between two code snippets.

6.3 Semantic Match

To assess plausible matching, we utilized test cases provided by the dataset creators, running each generated code sample through these test cases to verify successful execution. Our analysis revealed that many generated samples contained trivial syntactical errors, such as misplaced semicolons, which hindered compilation and execution. We calculated the plausible match by dividing the number of examples that passed all test cases by the total number of examples. Additionally, we conducted a manual comparison to evaluate whether the generated code logically corrected the errors. Notably, for code snippets with multiple bugs, the model often localized and corrected only one of the bugs, leaving others unresolved.

Experiments	CodeBLEU	Plausible Match	AST
CodeLlama inferred on C Code	0.63	0	0.71
RepairLLaMA inferred on Java Code	0.94	0.8	0.95
RepairLLaMA inferred on C Code	0.81	0.2	0.86
RepairLLaMA inferred on C++ Code	0.85	0.5	0.81
RepairCLlama (ours)	0.92	0.6	0.94

Table 4: Evaluation scores on various models

Experiments	CodeBLEU	Plausible Match	AST
CodeLlama inferred on C Code	0.61	0	0.70
RepairLLaMA inferred on C Code	0.78	0.16	0.83
RepairCLlama (ours)	0.91	0.54	0.93

Table 5: Evaluation scores on various models for 100 examples from TEGCER and Deepfix datasets

7 Results

In this section, we present and analyze the fine-tuning results obtained from the various experiments conducted in this project.

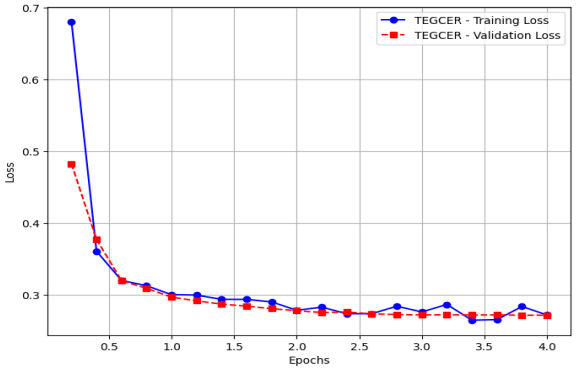


Figure 2: Training and Validation Loss vs epochs for TEGCER Dataset

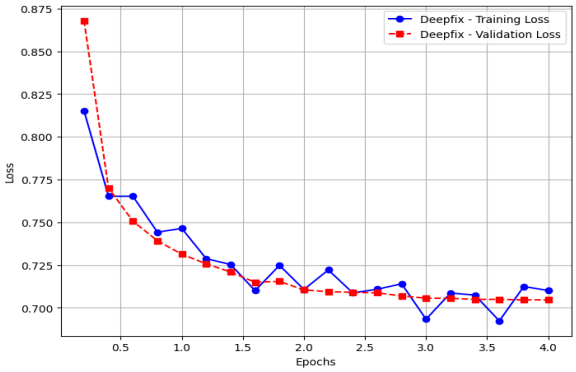


Figure 3: Training and Validation Loss vs epochs for Deepfix Dataset

Figure 2 illustrates how our model effectively learns during Experiment 1(TEGCER only fine-tuned), as seen by the monotonic decrease in validation loss. Notably, using <unk> as a padding token ensured proper handling of padding, which contributed to stabilizing training and further reducing the loss. Similarly, the results for Experiment 2 (Deepfix only fine-tuned) have been summarized in Figure 3 and the results for Experiment 3 (TEGCER and Deepfix combined fine-tuned) have been summarized in Figure 4.

Table 4 highlights the performance of our model against baseline models, demonstrating that it outperforms all others in the C/C++ code debugging task. All of these results have been evaluated on the same test set. Results in the last row of the table are evaluated on RepairLLaMA fine-tuned over the TEGCER training dataset. This is our best

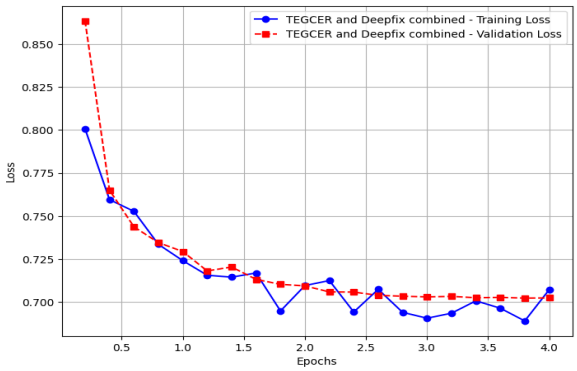


Figure 4: Training and Validation Loss vs epochs for model fine-tuned on TEGCER then fine-tuned on Deepfix Dataset

performing model. We observe that the models generated from Experiment 2 and Experiment 3 are not performing as well as the model generated from Experiment 1. This can be confirmed from the Figure 3 and Figure 4 where we see that the loss values are significantly higher. We observed better results on the TEGCER dataset compared to the Deepfix dataset. This can be attributed to the increased complexity of code in DeepFix, such as the presence of multiple loops and longer code snippets (Table 1a), whereas the TEGCER dataset consists of simpler code with fewer lines. Another reason is that TEGCER has a significantly higher number of training examples than DeepFix.

In Table 5, we evaluate the RepairCLlama on 100 examples from TEGCER as well as 100 examples from the DeepFix dataset. We observe a significant improvement in the evaluation scores for CodeBLEU, Plausible Match, and AST-based metrics with the RepairCLlama model. This validates our hypothesis that fine-tuning the model on a C/C++ debugging dataset enhances its performance on these programming languages.

8 Conclusion

In this work, we present RepairCLlama, a model that leverages transfer learning to enhance its debugging performance on C/C++ programming languages. Our results demonstrate that RepairCLlama outperforms all baseline models, achieving superior performance in automated program repair (APR) for C/C++ code. We have established a comprehensive experimental framework, which includes the curation of datasets and the development of a robust data pipeline comprising preprocessing and post-processing stages. To ensure rigorous

evaluation, we defined a comprehensive set of evaluation metrics, including CodeBLEU, AST similarity, and Semantic Match, and integrated these metrics into the model's inference pipeline. Our findings confirm the transferability of RepairLLaMA, originally trained on Java code, to effectively resolve bugs in C/C++ through fine-tuning. This highlights the model's adaptability and effectiveness in performing debugging tasks across programming languages.

9 Future Work

In future work, we will focus on several key areas. Firstly, we aim to further enhance RepairCLlama's performance on semantic match by exploring more diverse datasets. Secondly, we plan to extend the model's capabilities by fine-tuning it specifically for C++ code debugging, building upon the success achieved with C code debugging. Currently, the model only resolves single error in a given code snippet. To debug multiple errors, we aim to carry out an iterative inference within our pipeline where the model's own output is fed back in as input, allowing it to progressively correct additional bugs over multiple iterations. This iterative approach could bring the model closer to the way human developers tackle debugging: fixing errors step-by-step.

References

Umair Z. Ahmed, Renuka Sindhgatta, Nisheeth Srivastava, and Amey Karkare. 2019. Targeted example generation for compilation errors. In *The 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*. IEEE/ACM.

Abdulaziz Alaboudi and Thomas D. Latoza. 2021. An exploratory study of debugging episodes. *ArXiv*, abs/2105.02162.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Ramesh Nallapati, Baishakhi Ray, Sudipta Sengupta, Dan Roth, and Bing Xiang. 2023. Multi-lingual evaluation of code generation models. *Preprint*, arXiv:2210.14868.

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*.

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

Tuan Dinh, Jinman Zhao, Samson Tan, Renato Negrinho, Leonard Lausen, Sheng Zha, and George Karypis. 2023. Large language models of code fail at completing code with potential bugs. *Preprint*, arXiv:2306.03438.

Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 313–324, New York, NY, USA. Association for Computing Machinery.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31(1).

Siqi Han, Yu Wang, and Xuesong Lu. 2023. Errorclr: Semantic error classification, localization and repair for introductory programming assignments. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '23, page 1345–1354, New York, NY, USA. Association for Computing Machinery.

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. *Preprint*, arXiv:2106.09685.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? *Preprint*, arXiv:2310.06770.

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*, ISSTA 2014, page 437–440, New York, NY, USA. Association for Computing Machinery.

Mingda Li, Abhijit Mishra, and Utkarsh Mujumdar. 2024. Bridging the language gap: Enhancing multilingual prompt-based code generation in llms via zero-shot cross-lingual transfer. *Preprint*, arXiv:2408.09701.

Zhiming Li. 2021. Cross-lingual transfer learning framework for program analysis. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1074–1078.

Martin Monperrus, Matias Martinez, He Ye, Fernanda Madeiral, Thomas Durieux, and Zhongxing Yu. 2021. Megadiff: A Dataset of 600k Java Source Code Changes Categorized by Diff Size. *arXiv e-prints*, arXiv:2108.04631.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*.

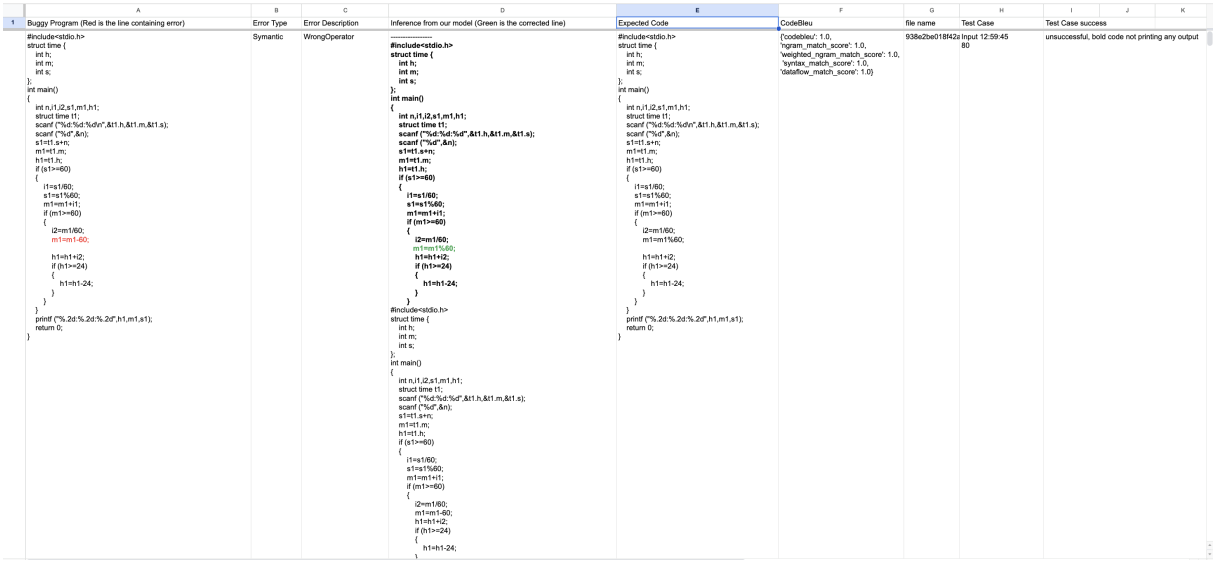
Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, M. Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *ArXiv*, abs/2009.10297.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

André Silva, Sen Fang, and Martin Monperrus. 2024. Repairllama: Efficient representations and fine-tuned adapters for program repair. *Preprint*, arXiv:2312.15698.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Huggingface's transformers: State-of-the-art natural language processing. *Preprint*, arXiv:1910.03771.

A Appendix



Buggy Program (Red is the line containing error)	Error Type	Error Description	Inference from our model (Green is the corrected line)	Expected Code	CodeBleu	file name	Test Case	Test Case success

#include<stdio.h>
struct time {
int h;
int m;
int s;
};
int main()
{
int n,i1,i2,s1,m1,h1;
struct time t1;
scanf ("%d:%d:%dn",&t1.h,&t1.m,&t1.s);
scanf ("%d",&n);
s1=t1.s+n;
m1=t1.m;
h1=t1.h;
if (s1>=60)
{
i1=s1/60;
s1=s1%60;
m1=m1+i1;
if (m1>=60)
{
i2=m1/60;
m1=m1-60;
h1=h1+i2;
if (h1>=24)
{
h1=h1-24;
}
}
}
printf ("%.2d:%.2d:%.2d",h1,m1,s1);
return 0;
}
	Symantic	WrongOperator	----------
			#include<stdio.h>
			struct time {
			int h;
			int m;
			int s;
			};
			int main()
			{
			int n,i1,i2,s1,m1,h1;
			struct time t1;
			scanf ("%d:%d:%dn",&t1.h,&t1.m,&t1.s);
			scanf ("%d",&n);
			s1=t1.s+n;
			m1=t1.m;
			h1=t1.h;
			if (s1>=60)
			{
			i1=s1/60;
			s1=s1%60;
			m1=m1+i1;
			if (m1>=60)
			{
			i2=m1/60;
			m1=m1%60;
			h1=h1+i2;
			if (h1>=24)
			{
			h1=h1-24;
			}
			}
			}
			#include<stdio.h>
			struct time {
			int h;
			int m;
			int s;
			};
			int main()
			{
			int n,i1,i2,s1,m1,h1;
			struct time t1;
			scanf ("%d:%d:%d",&t1.h,&t1.m,&t1.s);
			scanf ("%d",&n);
			s1=t1.s+n;
			m1=t1.m;
			h1=t1.h;
			if (s1>=60)
			{
			i1=s1/60;
			s1=s1%60;
			m1=m1+i1;
			if (m1>=60)
			{
			i2=m1/60;
			m1=m1-60;
			h1=h1+i2;
			if (h1>=24)
			{
			h1=h1-24;
			}
				#include<stdio.h>
				struct time {
				int h;
				int m;
				int s;
				};
				int main()
				{
				int n,i1,i2,s1,m1,h1;
				struct time t1;
				scanf ("%d:%d:%dn",&t1.h,&t1.m,&t1.s);
				scanf ("%d",&n);
				s1=t1.s+n;
				m1=t1.m;
				h1=t1.h;
				if (s1>=60)
				{
				i1=s1/60;
				s1=s1%60;
				m1=m1+i1;
				if (m1>=60)
				{
				i2=m1/60;
				m1=m1%60;
				h1=h1+i2;
				if (h1>=24)
				{
				h1=h1-24;
				}
				}
				}
				printf ("%.2d:%.2d:%.2d",h1,m1,s1);
				return 0;
				}
					{'codebleu': 1.0,
					'ngram_match_score': 1.0,
					'weighted_ngram_match_score': 1.0,
					'syntax_match_score': 1.0,
					'dataflow_match_score': 1.0}
						938e2be018f42a	Input 12:59:45
							80
								unsuccessful, bold code not printing any output

Figure 5: Inference Example on RepairLLaMA

INPUT	OUTPUT

#include <stdio.h>
#include <stdlib.h>
int main() {
int ti;
float tax;
scanf("%d", &ti);
if (ti == 200000) {
printf("tax=0");
} else if (200001 < ti < 500000) {
tax = ((ti - 200000) * 10) / 100;
printf("tax is %.2f", tax);
} else if (500001 < ti < 1000000) {
tax = 30000 + ((ti - 500000) * 20) / 100;
printf("tax is %.2f", tax);
} else if (1000000 < ti) {
tax = 130000 + ((ti - 1000000) * 30) / 100;
printf("tax is %.2f", tax);
}
return 0;
}
	#include <stdio.h>
	#include <stdlib.h>
	
	int main() {
	int ti;
	float tax;
	scanf("%d", &ti);
	if (ti == 200000) {
	printf("tax=0");
	}
	else if (200001 < ti && ti < 500000) {
	tax = ((ti - 200000) * 10) / 100;
	printf("tax is %.2f", tax);
	}
	else if (500001 < ti && ti < 1000000) {
	tax = 30000 + ((ti - 500000) * 20) / 100;
	printf("tax is %.2f", tax);
	}
	else if (1000000 < ti) {
	tax = 130000 + ((ti - 1000000) * 30) / 100;
	printf("tax is %.2f", tax);
	}
	
	return 0;
	}

TESTCASE	
Input : 200000 Output tax=0	

Figure 6: Inference Experiments Input and Output with test case

```
Prediction:  0
{'codebleu': 0.6976452476590235, 'ngram_match_score': 0.7108667889750 34, 'weighted_ngram_match_score': 0.710666582613441, 'syntax_
match_score': 0.6190476190476191, 'dataflow_match_score': 0.75}
Prediction:  1
{'codebleu': 0.6421334608217198, 'ngram_match_score': 0.5954165059120786, 'weighted_ngram_match_score': 0.6040697183271818, 'synta
x_match_score': 0.6190476190476191, 'dataflow_match_score': 0.75}
Prediction:  2
{'codebleu': 0.6421334608217198, 'ngram_match_score': 0.5954165059120786, 'weighted_ngram_match_score': 0.6040697183271818, 'synta
x_match_score': 0.6190476190476191, 'dataflow_match_score': 0.75}
Max CodeBLEU Prediction:  0.6976452476590235
```

Figure 7: Evaluation Metrics : CodeBLEU Predictions

```
========================================
SLURM_JOB_ID = 27817365
SLURM_JOB_NODELIST = b01-01
TMPDIR = /tmp/SLURM_27817365
========================================
Requirement already satisfied: accelerate==1.1.1 in ./repairllamaenv/lib/python3.12/site-packages (from -r requirements.txt (line 1)) (1
Requirement already satisfied: aiohappyeyeballs==2.4.3 in ./repairllamaenv/lib/python3.12/site-packages (from -r requirements.txt (line
Requirement already satisfied: aiohttp==3.11.2 in ./repairllamaenv/lib/python3.12/site-packages (from -r requirements.txt (line 3)) (3.1
Requirement already satisfied: aiosignal==1.3.1 in ./repairllamaenv/lib/python3.12/site-packages (from -r requirements.txt (line 4)) (1.
Requirement already satisfied: attrs==24.2.0 in ./repairllamaenv/lib/python3.12/site-packages (from -r requirements.txt (line 5)) (24.2.
Requirement already satisfied: bitsandbytes==0.42.0 in ./repairllamaenv/lib/python3.12/site-packages (from -r requirements.txt (line 6))
Requirement already satisfied: certifi==2024.8.30 in ./repairllamaenv/lib/python3.12/site-packages (from -r requirements.txt (line 7)) (
```

(a) CARC Job Scheduling: Installation of Requirements

```
Total samples in dataset: 13738
Samples used for training: 10000
Training samples: 8000
Validation samples: 2000

Sample buggy and corrected code:
                                          buggyCode                                          correctedCode
0  #include<stdio.h>\nint main(){\n    int a=10;\...  #include<stdio.h>\nint main(){\n    int a=10;\...
1  #include<stdio.h>\nint main(){\n    printf("Le...  #include<stdio.h>\nint main(){\n    printf("Le...
2  #include <stdio.h>\nint main() {\n\tint n,m,i,...  #include <stdio.h>\nint main() {\n\tint n,m,i,...
3  #include <stdio.h>\n#include <stdlib.h>\n\nint...  #include <stdio.h>\n#include <stdlib.h>\n\nint...
4  #include <stdio.h>\n#include <stdlib.h>\n\nint...  #include <stdio.h>\n#include <stdlib.h>\n\nint...

Loading checkpoint shards:   0%|          | 0/2 [00:00<?, ?it/s]
Loading checkpoint shards:  50%|          | 1/2 [01:26<01:26, 86.39s/it]
Loading checkpoint shards: 100%|          | 2/2 [01:56<00:00, 53.25s/it]
Loading checkpoint shards: 100%|          | 2/2 [01:56<00:00, 58.22s/it]
```

(b) fine-tunnig Data Analysis: Train and Validation Split

```
 99%|          | 496/500 [04:59<00:02,  1.65it/s]␛[A

 99%|          | 497/500 [05:00<00:01,  1.65it/s]␛[A

100%|          | 498/500 [05:00<00:01,  1.65it/s]␛[A

100%|          | 499/500 [05:01<00:00,  1.65it/s]␛[A

100%|          | 500/500 [05:02<00:00,  1.65it/s]␛[A


␛[A
100%|          | 4000/4000 [9:29:19<00:00,  5.48s/it]

100%|          | 500/500 [05:02<00:00,  1.65it/s]␛[A

                                                 ␛[A


100%|          | 4000/4000 [9:29:20<00:00,  5.48s/it]
100%|          | 4000/4000 [9:29:20<00:00,  8.54s/it]
/project/swabhas_1457/Group_18/Sharan/repairllamaenv/lib/python3.12/site-packages/peft/tuners/lora/bnb.py:83: UserWarning: Merge lora module to 8-bit
  warnings.warn(
{'eval_loss': 0.26610857248306274, 'eval_runtime': 302.6443, 'eval_samples_per_second': 6.608, 'eval_steps_per_second': 1.652, 'epoch': 8.0}
{'train_runtime': 34160.8006, 'train_samples_per_second': 1.873, 'train_steps_per_second': 0.117, 'train_loss': 0.2864993209838867, 'epoch': 8.0}
Finetuning completed and model saved.
```

(c) Final Training and Evaluation Loss

Figure 8: fine-tunnig Results

```
# Buggy C code prompt
buggy_code = """

int main() {
        printf("Hello World!")
        return 0;
}


Solve the error above line by line.
"""
```

(a) Input Buggy Code

```
Loading checkpoint shards:   0%|          | 0/2 [00:00<?, ?it/s]
Loading checkpoint shards:  50%|          | 1/2 [00:02<00:02,  2.26s/it]
Loading checkpoint shards: 100%|          | 2/2 [00:03<00:00,  1.41s/it]
Loading checkpoint shards: 100%|          | 2/2 [00:03<00:00,  1.54s/it]
Patch 1:

#include <stdio.h>

int main() {
        printf("Hello World!")
        return 0;
}


Solve the error above line by line.

#include <stdio.h>

int main() {
        printf("Hello World!")
        return 0;
}
```

(b) Loading our fine-tuned model

```
-----------------
Patch 6:

#include <stdio.h>

int main() {
        printf("Hello World!")
        return 0;
}

#include <stdio.h>

int main() {
        printf("Hello World!");
        return 0;
}
```

(c) Correct Patch Identified

Figure 9: Fine-tuned Model Inferences

Collaborative Contributions

The entire team worked together during the initial ideation phase, where we first considered creating an AI chatbot for specific tasks. After further discussions, we shifted our focus to software engineering tasks and eventually finalized code debugging using transfer learning as our project goal. While brainstorming ideas, we simultaneously searched for datasets that could be used and identified several options suitable for our task. We performed extensive research to explore prior work in the domain of code debugging using transfer learning and fine-tuned models. This helped us identify baseline models and relevant techniques to guide our approach. Initial inference testing was also carried out to analyze the model's performance on the selected datasets. The team actively participated in running fine-tuning jobs on CARC, where we scheduled jobs and optimized training processes. Each team member contributed equally to writing the report by taking responsibility for different sections. We also ensured the results were presented clearly by compiling them into structured tables and including supporting screenshots in the report and appendix for better readability. Finally, we collaboratively reviewed the report to identify and resolve any issues, ensuring clarity and consistency.

Team Member Contributions

- **Aabha**: Initially setup the pretrained LLM models(CodeLlama and RepairLLaMA) for inferencing. This formed the basis for the inference experimentation with prompting. Modified the fine-tuning script to adapat the tokenization / preprocessing changes for RepairLLaMA which led to an increase in the overall accuracy. Handpicked examples and generated inferences for the models and noted down the evaluation scores on them. Evaluated the best performing RepairCLlama model on the individual test sets, wrote Python script for that. Scheduled a few jobs on CARC for fine-tuning and evaluation on the test set for Deepfix and TEGCER Dataset. Contributed to the abstract, introduction, results, conclusions and future work sections of the report.

- **Dattateja**: Initially Worked on inferences using prompt engineering and investigated the impact of structured prompts on guiding the model's internal reasoning. Performed Exploratory data analysis on Deepfix dataset to understand the trends in the dataset. Performed error analysis on RepairLLaMA to understand the different errors solved by it. Experimented with multiple prompt templates, including those highlighting known bug types or offering explicit debugging hints, and systematically evaluated their effectiveness. Scheduled inference jobs on CARC to evaluate the different models. Contributed to the inference experimentation and Error Analysis sections of the report.

- **Parth**: Primarily responsible for data-related tasks, executed the preprocessing pipeline for the C/C++ datasets, including converting datasets into desired format as required for the finetuning step. Set up and optimized the GPU-based training environment on CARC (Centralized Advanced Research Computing) which involved configuring job scripts, managing computational resources, and ensuring smooth execution of Python-based training scripts in job format to handle large language models experiments. Contributed to the Algorithm and fine-tuning Experiments section (fine-tuning using LORA, fine-tuning with prompts) of the report.

- **Prerana**: Identified the evaluation metrics used by RepairLLaMA and explored additional suitable metrics for code generation and correction tasks. Implemented and set up the CodeBLEU, Plausible Match, and AST-based comparisons metrics, integrating them into the pipeline. Performed evaluations using these metrics and conducted in-depth result analyses by comparing model outputs against reference solutions and categorizing performance across different error types. Scheduled a few jobs on CARC for fine-tuning using LoRA and experimented with different hyperparameters. Plotted the fine-tuning loss graph to monitor the model's training progress and convergence across epochs. Contributed to the Evaluation Metrics and fine-tuning experiments (Training Procedure and hyperparameters, Fine-tuned Model Inference Setup) of the report.

- **Sharan**:

  Researched fine-tuning techniques for LLM models and applied Low-Rank Adaptation

(LoRA) for parameter-efficient fine-tuning of the RepairLLaMA architecture. Implemented fine-tuning scripts by utilizing baseline models from Hugging Face. Explored various hyperparameter configurations, including learning rates, batch sizes, max steps and training schedules to achieve stable convergence. Found the best training setup through repeated experiments, balancing model performance with efficient use of computational resources. Scheduled and managed CARC jobs for fine-tuning with different experimental setups. Created an inference script to evaluate the fine-tuned model by generating output patches. Contributed to the datasets, baseline models and a few portions of the algorithm section of the report.