Rust-doctor: Enhanced Feature for Rust Ownership and Lifetime Repair with Balanced Training Data Generation

Wenzhang Yang[1,2*], **Xiaoning Ren**[2*], **Cuifeng Gao**[2], **Yinxing Xue**[1,2],
[1] Institute of AI for Industries,
[2] Suzhou Institute for Advanced Research, University of Science and Technology of China,
Correspondence: yxxue@iaii.ac.cn

Abstract

As a relatively new programming language, Rust has gained significant popularity in recent years due to its safety features during compilation. However, Rust developers often face challenges stemming from its strict compilation checks due to the steep learning curve of safety rules. To make matters worse, the lack of training data and the unique semantics of Rust lead to poor performance in learning-based automated program repair techniques. To address these challenges, we propose a novel error injection approach to generate a balanced training dataset and leverage the Mid-level Intermediate Representation (MIR) as enhanced features for Rust's unique compilation error repair. Using these innovations, we fine-tuned a new code model, LLaRRA: **L**arge **L**anguage **a**nd **R**ust **R**epair **A**ssistant. Experimental results demonstrate that LLaRRA significantly outperforms state-of-the-art models in terms of Pass@K and Acc@K.

1 Introduction

Rust is a newborn system programming language with high performance and memory safety (Rus, 2024). To achieve these goals, Rust provides strict static rule checkers in compile-time to prevent plausible illegal memory access. Ownership and lifetime errors (OLEs) may occur when the code violates the corresponding rules. Benefits from both performance and safety advantages, Rust has been used in many safety-critical fundamental software (ann, 2023; Redox, 2023; Levy et al., 2017; Servo, 2023; Asay, 2020). However, Rust's checkers are a double-edge sword. Rigorous compile-time checking indeed serves fancy safety features while actually sacrifices the programming capability. Therefore, Rust developers struggle to fight with the official Rust compiler (*rustc*), and keep the loop of fixing OLEs and wait to check whether compilation success. To make things worse, *rustc* fails to provide enough information for developers (Zhu et al., 2022). Thus, to help the Rust developers, there is an emergency to automatically fix the Rust OLEs.

Recent years, many automatically program repair (APR) techniques are proposed to fix the program fault, including learning-based and unlearning-based approaches (Chen et al., 2021; Li et al., 2020, 2022b; Chi et al., 2023; Li et al., 2022a; Huang et al., 2023; Tufano et al., 2019; Wang et al., 2024; Zhang et al., 2023). Specifically, Yang et al. (Yang et al., 2024) proposed a rule-based approach, Rust-lancet, to fix ownership errors in Rust. However, this method is limited to ownership errors and is difficult to extend to lifetime errors. To maximize the effectiveness in fixing OLEs, learning-based techniques are preferable—assuming the availability of high-quality training data. Unfortunately, due to Rust as a relatively fresh programming language compared to the mainstream languages (C/C++ and Java) and few developers tend to upload uncompilable Rust program, lacking of enough compilation errors as training data, even the most powerful large language models (LLMs) cannot repair the OLE ideally. Hence, the challenge ❶ to enhance learning-based techniques is how to obtain pairs of high-quality uncompilable and compilable programs as training data. Existing work (Ahmed et al., 2018) usually collect data from students' coding assignments for mainstream programming languages, while Rust is not broadly introduced to school yet. The challenge ❷ is how to learn the underlying Rust program representation where OLE checkers are performed. These checkers analyze program in Intermediate Representation (IR) instead of source code. Thus, using pure Rust source code while lack of this IR feature can limit LLMs performance of OLE repair.

In this paper, we propose *Rust-doctor* to gen-

[*] These authors contributed equally to this work.

Findings of the Association for Computational Linguistics: EMNLP 2025, pages 3366–3376
November 4-9, 2025

erate OLE from real-world Rust projects to improve the OLE fixing performance of LLMs with enhanced features. To address challenge ❶, we propose an OLE injection framework to generate arbitrarily scale error dataset. In our framework, we use existing popular third-party libraries (crates) as a corpus and leverage LLMs to mutate compilable Rust functions using error-specific prompts. To keep the diversity of dataset, these functions are selected based on code similarity. To tackle challenge ❷, we introduce *rustc*'s MIR (MIR, 2024) as enhanced features for OLE repair. In Rust, MIR is designed to support more precise static checking, and the OLE is raised from MIR after the corresponding checking algorithm performed. Given the importance of MIR, we utilize an auto-encoder technique to capture its representation automatically and leverage the feature to improve repair capability of LLMs.

Overall, *Rust-doctor* takes multiple crates and uses k-means (Ahmed et al., 2020) to obtain function categories. After that, *Rust-doctor* selects a Rust function from each category evenly, consequently automatically mutates it by LLMs to trigger OLE and save the mutated function if the error is we want. Subsequently, a control flow graph is constructed based on MIR and its structure information is captured by a graph encoder. Finally, we fine-tune LLMs with Rust source code and features produced to fix Rust's OLE. The fine-tuned models not only enhanced by large scale high quality training data, but also learn the program representation where the OLE checkers are performed to fix OLE better.

The main contributions are as follows:

- *A high-quality Rust OLE dataset and benchmark.* We are the first to propose an automatic Rust OLE injection framework based on real-world Rust projects. Furthermore, we construct an open source high-quality Rust OLE dataset and benchmark for the OLE repair community.
- *Feature exploration for OLE fixing.* We explore the program representation features OLE checkers performed on. The features provide direct control flow information, enhancing the LLMs capabilities for fixing OLE.
- *Large language models for OLE fixing.* We implement and evaluate our approach as *Rust-doctor* and fine-tune an LLM *LLaRRA* based on the generated dataset with enhanced features. The experimental results show that *LLaRRA* significantly outperforms the state-of-the-art model, with Pass@K increasing by 12.61% and Acc@K increasing by 18.30%.
- *Open-source.* Upon publication, we will release the following assets to the public: the generated OLE repair data, the codebase, and the model checkpoints. Some pre-released assets are already available at https://sites.google.com/view/rustdoctor/index.

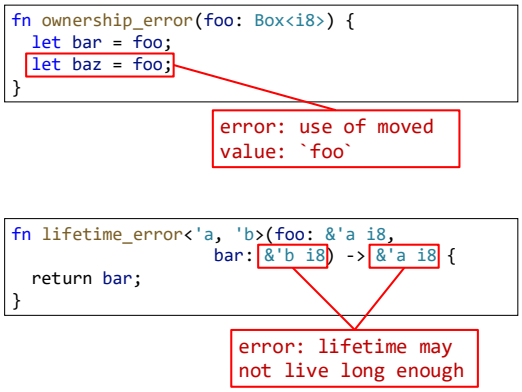


Figure 1: Ownership and Lifetime Errors

2 Preliminaries

2.1 Ownership and Lifetime Errors

There are two Rust-specific categories of compilation errors: ownership errors and lifetime errors (collectively referred to as OLE).

According to ownership rules, each value has a unique owner, typically a variable. When ownership is moved from one variable to another, the original variable becomes invalid. Attempting to use a moved value results in a compile-time ownership error. In Figure 1, `foo` is a `Box<i8>`, a heap-allocated value. After the move to bar, the ownership of `foo` is moved, and `foo` becomes invalid. The second assignment to `baz` attempts to use `foo` again, leading to a compilation error because the ownership has already been moved.

While ownership governs who owns a value, lifetimes track how long references are valid. Rust's compiler uses lifetimes to ensure that all references are valid during their use and that no reference outlives the data it points to. A lifetime error occurs when the compiler cannot guarantee that a returned reference will remain valid. As shown in Figure 1, the return type (`&'a`) declares that the returned reference must live at least as long as lifetime `'a`. However, the function attempts to return `bar`, which has

lifetime `'b`. If `'b` is shorter than `'a`, the returned reference could point to invalid memory, so the compiler raises a lifetime mismatch error.

2.2 Problem Formulation

Rust-doctor aims to address Rust OLE compilation errors by learning program semantics through fine-tuning techniques. Formally, let p_{err} represent a Rust program with OLEs from a dataset D. Its associated compilation error message and emitted MIR, provided by *rustc*, are denoted as m_{err} and ir_m, respectively. The objective of *Rust-doctor* is to learn a function f from the dataset D that takes (p_{err}, m_{err}, ir_m) as input and produces a corrected version $p_{correct}$, which can be successfully compiled by *rustc*. This process can be formally expressed as:

$$p_{correct} = f(p_{err}, m_{err}, ir_m)$$

To this end, we aim to generate a tuning dataset consisting of paired data $\langle p_{correct}, (p_{err}, m_{err}, ir_m)\rangle$. Given the proportion of the i-th OLE type, denoted as p_i, the primary goal of the generation process, aimed at balancing the dataset, can be expressed as:

$$\text{Maximize} \quad Entropy(D) = -\sum_{i=1}^{K} p_i \log_2 p_i$$

3 Related Work

In recent years, numerous works (Huang et al., 2023) have been proposed to repair buggy code. Tufano et al.(Tufano et al., 2019) introduced a code abstraction technique to repair buggy code using neural machine translation. Following this, several end-to-end approaches(Chen et al., 2021; Li et al., 2020, 2022b; Chi et al., 2023; Li et al., 2022a) were developed to enhance model performance in code fixing tasks. Additionally, some non-end-to-end and non-learning-based methods have been proposed. RATCHET (Wang et al., 2024) introduced a dual deep learning-based automated program repair tool, with one model for localization and another for repair. OrdinalFix (Zhang et al., 2023) focuses on ensuring output correctness with minimal modifications by leveraging shortest-path context-free language reachability with attribute checking. Notably, Yang et al. (Yang et al., 2024) were the first to address Rust's ownership-rule violations using rule-based fixing strategies. In the era of large language models (LLMs), Ma et al. (Ma et al., 2024) conducted an empirical study using four probing tasks to evaluate the models' capabilities in learning code syntax and semantics. Their results demonstrate that while code models excel at learning code syntax, even state-of-the-art LLMs require further improvement in understanding and learning code semantics.

On the other hand, while learning-based techniques have achieved a significant milestone in automated program repair, they still suffer from limitations when applied to tasks with insufficient training data. Over the past years, many researchers have concentrated on advancing automated data generation methods by LLMs or Pre-trained Language Models (PLMs). Meng et al.(Meng et al., 2022) proposed a straightforward approach that leverages a unidirectional PLM to generate class-conditioned texts guided by prompts. These generated texts are then used as training data to fine-tune a bidirectional PLM. To generate diverse and attribute-rich data, Yu et al.(Yu et al., 2023) explored the use of diversely attributed prompts to create training data with LLMs. Moreover, numerous code models (Rozière et al., 2023; Li et al., 2023; Muennighoff et al., 2024; Lozhkov et al., 2024a) have been fine-tuned using techniques such as code generation, data distillation, and online data collection, but they are not considering Rust's OLE repair.

In summary, although existing studies have made progress in improving the performance of APR, limitations remain in OLE repair for Rust. Specifically, the APR community lacks effective approaches for both OLE data generation and Rust-specific semantics learning. To address these challenges, we propose incorporating data generation techniques and leveraging MIR features to enhance the OLE repair performance of LLMs.

4 Proposed Method

An overview of the proposed approach is illustrated in Figure 2. The approach comprises three key components: a data generation module, a MIR encoding module, and a tuning module.

4.1 Data Generation

A balanced training dataset is benefit to the performance of LLMs. To this end, we leverage a balanced function selection approach and inject the compilation errors.

Initially, all functions and their corresponding

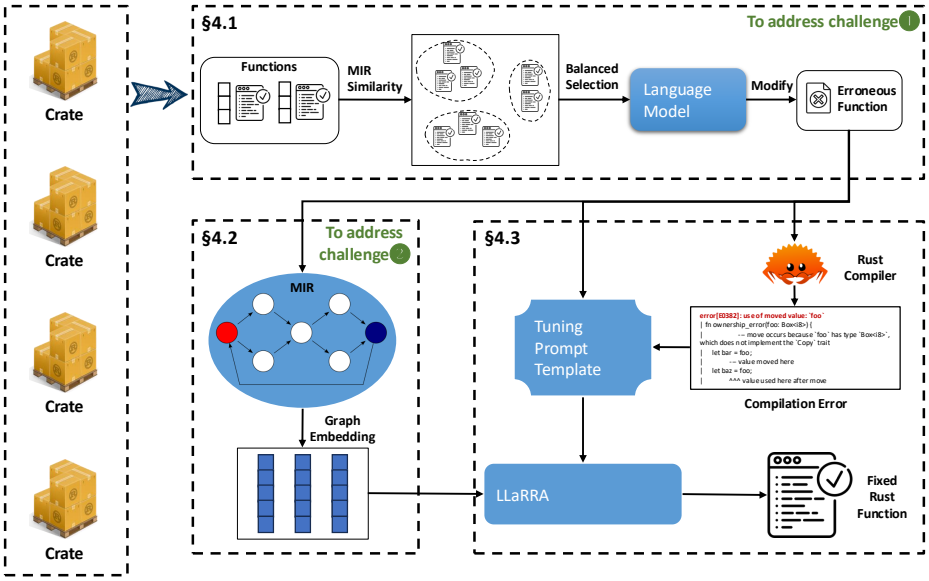


Figure 2: Overview of *Rust-doctor*

MIRs are extracted from the corpus crates. Since OLE checkers operate on MIR, MIR similarity serves as a metric for assessing the balance of the OLE fixing training dataset. To achieve this, *Rust-doctor* utilizes CodeBERT (Feng et al., 2020) to encode the MIR code and k-means for clustering. Subsequently, *Rust-doctor* selects functions evenly from each cluster and mutates them using carefully designed Rust-specific mutation prompts including *Statements Swapping*, *Statements Deleting*, *Statements Inserting*, *Expression Changing*, *Lifetime Changing*, *Lifetime Adding* and *Reference Adding*.

In this module, *Rust-doctor* is tasked with collecting not only erroneous Rust functions but also their corresponding erroneous MIRs. However, *rustc* cannot emit MIRs for programs with compilation errors. To address this, we modified the order of *rustc*'s compilation passes to emit MIR before performing OLE checking.

We successfully collected a total of 359K unique Rust OLE fixing samples, including 136K in MIR, 183K in compilation error messages, and 40K in erroneous Rust functions. To evaluate the dataset, we conducted a sanitizer to measure code edge coverage of the OLE checkers. The results show that our generated dataset achieves 71.58% edge coverage of the OLE checkers, indicating that our dataset provides relatively comprehensive coverage of the core OLE checking algorithms.

4.2 MIR Feature Extraction

MIR is the most fundamental and critical representation for analyzing OLEs, as *rustc* applies the borrow checker directly to it. From the programmer's perspective, OLEs typically involve data flow, aliasing, and function calls, where functions are annotated with lifetime tags—a hybrid form of data-flow and aliasing. Internally, *rustc* can derive all of this information directly from MIR. Therefore, we leverage MIR as the sole IR-level feature, as it captures nearly all of the nuances of OLEs.

To extract semantic information from the MIR of Rust functions, we construct a control flow graph where each Basic Block (BB) serves as a node, and the flow transfer between blocks forms an edge. Based on this graph structure, we perform graph embedding extraction. First, CodeBERT is used to encode the code of each node into vector embeddings. Subsequently, a final feature vector representing the entire MIR is obtained by pooling all the node features.

Formally, each basic block BB in the MIR control flow graph $X_{mir} = \{BB\}$, represents specific statements or operations within a Rust program. Let the textual representation of a node BB_i be t_i, and its corresponding embedding $\mathbf{v}_i$ is computed as follows:

$$\mathbf{v}_i = \text{CodeBERT}(t_i) \quad (1)$$

Here, $\text{CodeBERT}(\cdot)$ denotes the function mapping input text to a fixed-dimensional vector. To

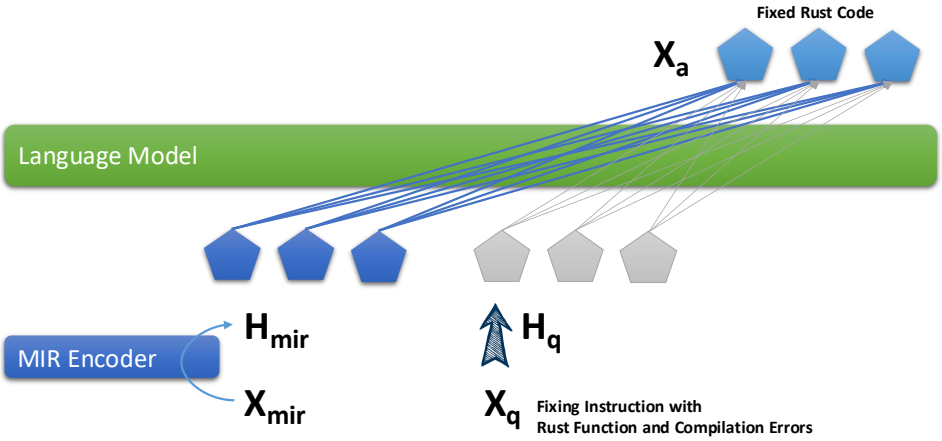


Figure 3: The overall architecture of *LLaRRA*

encode the entire control flow graph X_{mir}, we first compute embeddings for all nodes $BB_i \in X_{mir}$. The set of embeddings is denoted as:

$$\mathcal{V} = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_{|X_{mir}|}\} \tag{2}$$

Next, we apply attention-based pooling to aggregate these node embeddings into a single graph-level representation $\mathbf{g}$. Since the root cause of OLEs are often localized to a specific area of code, effectively capturing these local semantics is critical. Attention-based pooling allows the model to assign higher importance to nodes that are more relevant for identifying such root causes. The attention weights α_i for each node are computed as:

$$\alpha_i = \frac{\exp(\mathbf{w}^T \tanh(\mathbf{W}_a \mathbf{v}_i + \mathbf{b}_a))}{\sum_{j=1}^{|V|} \exp(\mathbf{w}^T \tanh(\mathbf{W}_a \mathbf{v}_j + \mathbf{b}_a))} \tag{3}$$

where $\mathbf{W}_a$, $\mathbf{b}_a$, and $\mathbf{w}$ are learnable parameters of the attention mechanism. The MIR graph-level embedding $\mathbf{H_{mir}}$ is then obtained as a weighted sum of the node embeddings:

$$\mathbf{H_{mir}} = \sum_{i=1}^{|\mathcal{V}|} \alpha_i \mathbf{v}_i \tag{4}$$

This components helps extract semantic information from critical nodes and incorporates it as part of the input. It enables the model to focus on the most relevant nodes in the control flow graph, which are likely to explain the root causes of OLEs.

4.3 Tuning

The network architecture is illustrated in Figure 3. We select a base LLM and perform task-specific modifications to its structure. Specifically, we employ *Llama3.1-8b-instruct* as our base LLM, as it is one of the most widely used instruction-tuned models among publicly available options. It offers competitive performance across diverse tasks and maintains an acceptable deployment cost, making it accessible to most research groups in the community.

To enhance the model's capabilities, we introduce an additional semantic feature input module positioned before the first decoder. The output of this module is concatenated with the token embedding sequence as a prefix and then fed into the first decoder. This design enables the model to leverage supplementary features, thereby improving its task performance. Since Rust's OLE checkers perform on MIR, we incorporate MIR semantic information as auxiliary input. This addition enhances the model's ability to understand specific errors, ultimately aiming to improve its error repair performance.

The first step involves designing the model's input. The primary input is a prompt, where the instruction specifies the task for the model, and the input provides the erroneous code and corresponding compilation error message. The prompt structure is as follows:

```
You are a highly skilled Rust developer Your task is to fix Rust
code that contains compilation errors and return only the
corrected complete Rust code. No explanations, comments, or extra
text output only the fixed code. Please fix the following code:
### Erroneous source code:
    {function}
### Compilation error message:
    {error message}
```

In addition to the prompt text, *LLaRRA* incorporates the MIR of the erroneous code as input to an auxiliary semantic module. This combined input

design ensures comprehensive training by leveraging both the textual information from the prompt and the structural semantic features extracted from the MIR encoder. Let X_q and H_{mir} represent the textual input and MIR representation, respectively. The combined input can be expressed as:

$$x_{\text{combined}} = f_{\text{concat}}(H_{mir}, f_{\text{enc}}(X_q))$$

where f_{enc} denotes the textual encoder module and f_{concat} represents the concatenation function that merges the encoded textual representation and the MIR representation.

Once the input design is finalized, the training objective is defined. To achieve efficient fine-tuning, we employ LoRA (Xu et al., 2024), which optimizes all network layers by introducing low-rank adaptations. The fine-tuning process minimizes the cross-entropy loss $\mathcal{L}_{\text{CE}}$ for supervised fine-tuning (SFT). The objective is to ensure the generated output y closely matches the target y_{true}. This can be expressed as:

$$\mathcal{L}_{\text{CE}} = -\sum_{t=1}^{T} y_{\text{true},t} \log P(y_t \mid x_{\text{combined}}, \theta)$$

Here, T is the sequence length, θ represents the model parameters, and $P(y_t \mid x_{\text{combined}}, \theta)$ denotes the predicted probability of the token at position t. This optimization enhances the model's ability to generate accurate and context-aware responses, improving its robustness and precision in handling erroneous code.

5 Experiments

In this section, we evaluate *Rust-doctor* by fine-tuning a new model based on the LLama3.1-8b-instr from Meta (Meta, 2024). Our goal is to address the following research questions:

- **RQ1:** How does *LLaRRA* perform compared with other state-of-the-art LLMs?
- **RQ2:** How does each component of *Rust-doctor* contribute to *LLaRRA*'s fixing ability?
- **RQ3:** What are the fixing capabilities overlap between *LLaRRA* and other LLMs?

5.1 Experimental Settings

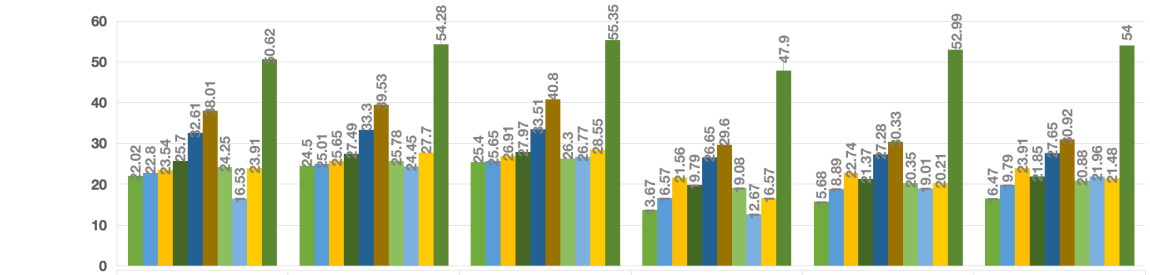
Datasets. The dataset utilized in our experiments was generated using our proposed data generation module, based on the top 30 most-downloaded crates from *crates.io* (cra, 2017), the Rust community's official package registry. Each data sample includes the pre-mutated Rust function as the ground truth and the mutated version as the model input, along with the corresponding MIR, erroneous function's file name, and compilation error message. After deduplication, the dataset comprises a total of 9,467 Rust OLE samples. The dataset is split into training and testing subsets in an 8:2 ratio, with the training set designated for model training and the testing set reserved for performance evaluation. After that, we apply function-level deduplication using a Rust-based code formatting tool, ensuring no overlap between the training and testing sets.

Compared Techniques. To evaluate the improvement of the model's performance compared to that of existing techniques, we employ several popular code models as comparative experimental techniques: *GPT-4-0613*, *GPT-4-turbo-2024-04-09*, *GPT-4o-2024-08-06*, *GPT-4o-mini-2024-07-18*, *DeepSeek-V3-2024-12-26* (Liu et al., 2024), *DeepSeek-R1-2025-01-20* (DeepSeek-AI et al., 2025), *Gemini-2.0-flash-ex-2024-12* (Team et al., 2023), *Llama3.1-8b-instruct* and *StarCoder* (Lozhkov et al., 2024b). A temperature of 0.5 was consistently applied across all models to balance output determinism and randomness.

Implementation Details. In the data generation module, we implemented an error injector using Rust and Python. Specifically, the global controller and core logic were developed in Rust, while LLama3.1-8b-instr was employed as the model to modify Rust functions.

The training process is configured with a sequence length cutoff of 2048 and a per-device batch size of 4, with gradient accumulation over 8 steps. The learning rate is set to 1.0e-4 and follows a cosine annealing schedule, with a warm-up ratio of 0.1 to gradually increase the learning rate during the initial steps. *LLaRRA* is trained for 3 epochs using mixed-precision (bf16) to optimize memory usage and computational efficiency. These hyperparameters are carefully selected to balance training performance and resource constraints, ensuring effective model optimization.

Metrics. In this study, we employ two metrics to evaluate error repair performance. The first metric, Pass@K, measures the percentage of generated programs that successfully compile among the top-K



	Pass@1	Pass@3	Pass@5	Acc@1	Acc@3	Acc@5
GPT-4	22.02	24.5	25.4	13.67	15.68	16.47
GPT-4-turbo	22.8	25.01	25.65	16.57	18.89	19.79
GPT-4o	23.54	25.65	26.91	21.56	22.74	23.91
GPT-4o-mini	25.7	27.49	27.97	19.79	21.37	21.85
DeepSeek-V3	32.61	33.3	33.51	26.65	27.28	27.65
DeepSeek-R1	38.01	39.53	40.8	29.6	30.33	30.92
Gemini	24.25	25.78	26.3	19.08	20.35	20.88
StarCoder	16.53	24.45	26.77	12.67	19.01	21.96
Base	23.91	27.7	28.55	16.57	20.21	21.48
LLaRRA	50.62	54.28	55.35	47.9	52.99	54

Figure 4: Comparing with other LLMs.

returned results. However, while Pass@K is useful, it has limitations in scenarios where erroneous lines are simply removed rather than correctly edited (Li et al., 2022a). To address this, we introduce a second metric, Acc@K, which calculates the percentage of correct results present in the top-K returned outputs. For all baseline techniques, K is set to 1, 3, and 5.

5.2 Comparing with Other LLMs (RQ1)

***LLaRRA* achieves the highest scores.** As shown in Figure 4, across all baseline models in the Rust OLE repair task, *LLaRRA* stands out by achieving the highest average Pass@K and Acc@K scores. Despite having only 8 billion parameters—the smallest among the benchmarked models—*LLaRRA* outperforms all existing popular LLMs, including the code-focused model *StarCoder*. Notably, it demonstrates a significant performance margin over the state-of-the-art reasoning model, *DeepSeek-R1*. These experimental results underscore the effectiveness of our proposed approach in enhancing LLM performance for the Rust OLE repair task.

The patches generated by *LLaRRA* are practical. To repair a compilation error, the easiest approach might be to delete the line containing the erroneous code. However, this solution is generally unacceptable in practice. As a result, a model with a high Pass@K score might produce more unexpected patches if it has a low Acc@K score. As shown in Figure 4, the average difference between Pass@5 and Acc@5 across other methods is 5.5%. In contrast, *LLaRRA* achieves a significantly smaller difference of 0.46%. This result demonstrates that patches generated by *LLaRRA* are highly likely to be both expected and correct.

Table 1: Ablation experiment results.

Models	Pass@K (%)			Acc@K (%)		
	Pass@1	Pass@3	Pass@5	Acc@1	Acc@3	Acc@5
Base	23.91	27.70	28.55	16.57	20.21	21.48
LLaRRA$_M$	44.33	46.70	47.44	43.87	46.19	46.98
LLaRRA$_B$	31.65	37.05	39.21	28.42	33.09	35.61
LLaRRA	50.62	54.28	55.35	47.90	52.99	54.00

Models struggle to improve performance by increasing sample number K. From Pass@1 to Pass@5, the performance of all models does not increase significantly, as demonstrated in Figure 4. In most cases, issues are resolved within the Pass@1 setting. Future work could explore methods to enable models to learn from iterative queries, thereby improving overall performance as K increases.

5.3 Ablation Study (RQ2)

To distinguish the contributions of balanced data generation and enhanced MIR feature, we conduct an ablation analysis by altering each module as follows:

- *LLaRRA*$_M$: Without employing the MIR encoder for enhanced feature extraction. Instead, this variant uses only pure prompt tuning instructions along with Rust functions and compilation error messages.
- *LLaRRA*$_B$: To assess the impact of the balanced data generation component, we remove the k-means clustering and balanced selection strategies. As a result, *LLaRRA*$_B$ is fine-tuned with the

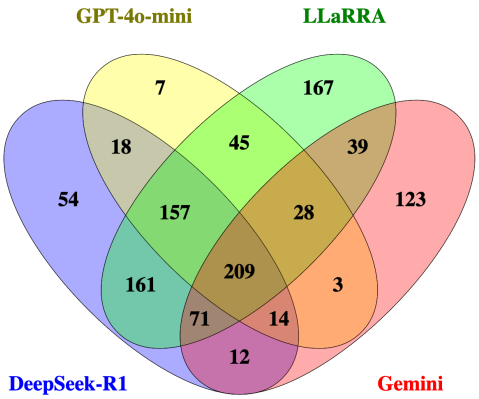


Figure 5: OLE coverage.

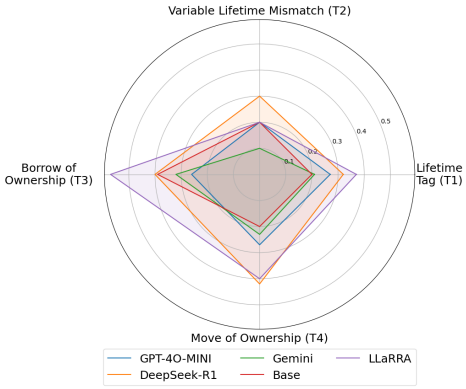


Figure 6: The OLE fixing radar of models

base model using a randomly generated tuning dataset.

In Table 1, the results highlight the effectiveness of our proposed method. Both modified models outperform the base model but perform worse than *LLaRRA*. Specifically, $LLaRRA_M$ resolves 18.89% more OLEs than the base model in terms of Pass@5, demonstrating that the enhanced MIR feature significantly boosts repair performance. The removal of balance data generation also leads to decrease in all the metrics, emphasizing the importance of balance data for OLE repair. Overall, the complete *LLaRRA* is better than all other variants, further justifying the effectiveness of the combination of generated dataset and enhanced MIR feature.

5.4 Fixing Capabilities Analysis (RQ3)

To reveal the overlap in fixing capabilities between *LLaRRA* and other LLMs, we analyze the fixed results using two distinct distributions. First, we calculate the OLE coverage to quantify the degree of complementarity offered by other models. Second, we classify all compilation errors into four categories and assess the fixing capabilities of selected LLMs for each error type.

***LLaRRA* fixes the most unique OLEs** We evaluate four models from different organizations. Among them, *GPT-4o-mini*, developed by OpenAI, delivers the best performance within the GPT series in RQ1. Additionally, we select *DeepSeek-R1* from the DeepSeek series, as its incorporation of CoT enhances reasoning capabilities. The other two models are *Gemini* and *LLaRRA*.

As shown in Figure 5, *LLaRRA* fixes the most unique OLEs (167), followed by *Gemini*. The experimental results demonstrate that our approach enables LLMs to better understand the deep semantics of Rust OLEs, significantly outperforming reasoning model and close-source LLMs. Furthermore, these results reveal the potential for improving OLE repair performance through output fusion across distinct models.

Error types and its degree of diffculty. We categorize all compilation error codes into the following four types:

- *Lifetime Tag (T1)*: An error code is categorized as a Lifetime Tag if the standard patch in the official error specification involves a lifetime tag.
- *Variable Lifetime Mismatch (T2)*: An error code belongs to Variable Lifetime Mismatch if there is a mismatch between the lifetimes of two variables, and the error does not fall under T1.
- *Borrow of Ownership (T3)*: An error code is categorized as Borrow of Ownership if the error involves the borrowing of ownership between two variables within a function body.
- *Move of Ownership (T4)*: An error code belongs to Move of Ownership if it involves a move within a function and does not fall under T3.

As shown in Figure 6, *LLaRRA* outperforms most other models, addressing a significant portion of the errors fixed by OLE. The only exception is error type T2, where *DeepSeek-R1* surpasses *LLaRRA* due to its superior reasoning capabilities, enabling it to better infer relationships and constraints between variable lifetimes.

6 Conclusion

In this study, we propose *Rust-doctor*, a novel data generation and enhanced feature approach designed to automatically generate Rust's OLE for

fine-tuning large language models (LLMs). Our approach aims to deliver high-quality OLE data to benefit the Rust program automatic repair community. Within the dataset, we successfully collected a total of 359K unique Rust OLE fixing samples, comprising 136K in MIR, 183K in compilation error messages, and 40K in erroneous Rust functions. Additionally, using the generated training dataset, we fine-tune and release a specialized LLM, *LLaRRA*. Experimental results demonstrate that *LLaRRA* outperforms all popular baseline LLMs, highlighting its potential as an effective OLE repair tool for the Rust programming language community.

7 Limitations

Regular syntax errors. We focus exclusively on addressing the challenges of repairing OLEs, without considering regular syntax errors. Prior work (Deligiannis et al., 2024) has demonstrated that regular syntax errors can be effectively resolved. Nonetheless, this limitation of our work remains.

Generalization. We evaluate *Rust-doctor* on the top 30 most downloaded crates. As these crates do not cover the full spectrum of coding scenarios, the generalization capability of the trained model may be limited. Nevertheless, we believe our methodology is applicable to a broader range of crates and has the potential to achieve better generalization performance.

Data Preparatioin. Our experiments rely on a synthesized dataset generated by a large language model (LLM), in which compilation errors were artificially injected. While modern LLMs can mimic human programming behavior, it remains uncertain whether the injected errors accurately reflect real-world Rust OLEs. Moreover, relying on LLMs for data generation may introduce noises, such as the use of the `compile_error!` macro. These factors introduce uncertainty that may limit the credibility of our findings and the practical applicability of the proposed approach.

References

2017. crates.io. https://crates.io/. Accessed: 2025-01-10.

2023. announcing-windows-11-insider-preview-build-25905. Accessed: 2023-07-21.

2024. Rust: A language empowering everyone to build reliable and efficient software.

Mohiuddin Ahmed, Raihan Seraj, and Syed Mohammed Shamsul Islam. 2020. The k-means algorithm: A comprehensive survey and performance evaluation. *Electronics*, 9(8):1295.

Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. 2018. Compilation error repair: for the student programs, from the student programs. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training, ICSE (SEET) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 78–87. ACM.

Matt Asay. 2020. Why AWS loves Rust, and how we'd like to help.

Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2021. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Trans. Software Eng.*, 47(9):1943–1959.

Jianlei Chi, Yu Qu, Ting Liu, Qinghua Zheng, and Heng Yin. 2023. Seqtrans: Automatic vulnerability fix via sequence to sequence learning. *IEEE Trans. Software Eng.*, 49(2):564–585.

DeepSeek-AI, Daya Guo, Dejian Yang, and etc. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Pantazis Deligiannis, Akash Lal, Nikita Mehrotra, Rishi Poddar, and Aseem Rastogi. 2024. Rustassistant: Using llms to fix compilation errors in rust code. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 267–279. IEEE Computer Society.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.

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 *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 1162–1174. IEEE.

Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, Shanghai, China.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, and 48 others. 2023. Starcoder: may the source be with you! *Trans. Mach. Learn. Res.*, 2023.

Xueyang Li, Shangqing Liu, Ruitao Feng, Guozhu Meng, Xiaofei Xie, Kai Chen, and Yang Liu. 2022a. Transrepair: Context-aware program repair for compilation errors. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 108:1–108:13. ACM.

Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. Dlfix: context-based code transformation learning for automated program repair. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 602–614. ACM.

Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022b. DEAR: A novel deep learning-based approach for automated program repair. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 511–523. ACM.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, and 38 others. 2024a. Starcoder 2 and the stack v2: The next generation. *CoRR*, abs/2402.19173.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024b. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Wei Ma, Shangqing Liu, Mengjie Zhao, Xiaofei Xie, Wenhan Wang, Qiang Hu, Jie Zhang, and Yang Liu. 2024. Unveiling code pre-trained models: Investigating syntax and semantics capacities. *ACM Trans. Softw. Eng. Methodol.*, 33(7):169:1–169:29.

Yu Meng, Jiaxin Huang, Yu Zhang, and Jiawei Han. 2022. Generating training data with language models: Towards zero-shot language understanding. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Meta. 2024. Llama 3.1.

MIR. 2024. MIR.

Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2024. Octopack: Instruction tuning code large language models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Redox. 2023. Redox - Your Next(Gen) OS.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, and 6 others. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Servo. 2023. Servo.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, and 1 others. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

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 Trans. Softw. Eng. Methodol.*, 28(4):19:1–19:29.

Jian Wang, Shangqing Liu, Xiaofei Xie, Jingkai Siow, Kui Liu, and Yi Li. 2024. Ratchet: Retrieval augmented transformer for program repair. In *35th IEEE International Symposium on Software Reliability Engineering, ISSRE 2024, Tsukuba, Japan, October 28-31, 2024*, pages 427–438. IEEE.

Yuhui Xu, Lingxi Xie, Xiaotao Gu, Xin Chen, Heng Chang, Hengheng Zhang, Zhengsu Chen, Xiaopeng Zhang, and Qi Tian. 2024. Qa-lora: Quantization-aware low-rank adaptation of large language models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Wenzhang Yang, Linhai Song, and Yinxing Xue. 2024. Rust-lancet: Automated ownership-rule-violation fixing with behavior preservation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 85:1–85:13. ACM.

Yue Yu, Yuchen Zhuang, Jieyu Zhang, Yu Meng, Alexander J. Ratner, Ranjay Krishna, Jiaming Shen, and Chao Zhang. 2023. Large language model as attributed training data generator: A tale of diversity

and bias. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Wenjie Zhang, Guancheng Wang, Junjie Chen, Yingfei Xiong, Yong Liu, and Lu Zhang. 2023. Ordinalfix: Fixing compilation errors via shortest-path CFL reachability. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 1200–1211. IEEE.

Shuofei Zhu, Ziyi Zhang, Boqin Qin, Aiping Xiong, and Linhai Song. 2022. Learning and programming challenges of rust: A mixed-methods study. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1269–1281.