DistiLRR: Transferring Code Repair for Low-Resource Programming Languages

Anonymous ACL submission

Abstract

Large language models (LLMs) have shown remarkable performance on code generation tasks. A recent application of LLMs for code generation is iterative code repair, where a model fixes an incorrect program by rationalizing about errors and generating a new program. However, code repair is primarily studied on high-resource languages like Python, and the framework's efficacy is under-explored on low-resource languages. To apply code repair for low-resource languages, we propose Distilling Low-Resource Repairs (DistiLRR), an approach that transfers the reasoning and code generation ability from a teacher model to a student model. Our results show that DistiLRR consistently outperforms baselines on low-resource languages, but has similar performance on high-resource languages. To investigate this behavior, we perform a further analysis and find that the correlation between rationale quality and code correctness is weaker than previously perceived. We hypothesize this weakness is magnified in low-resource settings where base models lack deep knowledge of a programming language, leading to wavering benefits of code repair between high-resource and low-resource languages.

1 Introduction

Recent advancements in large language models (LLMs) have displayed remarkable capacity in generating human-aligned code (Wang and Chen, 2023). While many models like GPT-4 (OpenAI, 2024) and CodeLlama (Rozière et al., 2024) have high performance on benchmarks like HumanEval (Chen et al., 2021), LLMs are primarily evaluated on high-resource programming languages (HRPLs), such as Python. Meanwhile, their performance lags behind for low-resource programming languages (LRPLs), such as Perl (Athiwaratkun et al., 2023). One reason for this gap is that LRPLs lack representation in pretraining data because they are rarer to find in a natural setting. For example, a modern code LLM DeepSeek-Coder (Guo et al., 2024) uses a training dataset scraped from public Github repositories, containing high-resource languages like Python and Java at rates of 15.12% an 18.63%, while low-resource languages like Perl and Golang are at rates of 0.1% and 0.32%. Thus, creating an efficient framework that improves LRPL code generation without the need of more human-written code is essential.

To address this problem, we begin by adopting the framework of code repair. Code repair appears especially useful in a low-resource setting because it augments inference with automatic feedback and reasoning, without needing extra human written code. The framework draws inspiration from the editing process of human programmers: erroneous feedback is automatically provided through executing tests, while programmers rationalize about those errors to modify the code. Our work models this, demonstrated in Figure 1.

Although seemingly effective, recent works conclude that self-repair is bottlenecked by the repair model's ability to rationalize about errors (Olausson et al., 2024), leading to lower improvements on weaker models. To further improve repairs for smaller LLMs, we propose Distilling Low-Resource Repairs (DistiLRR), where the ability to repair code is taught by a larger model. At the same time, distillation addresses the lack of human written LRPL code by creating synthetic data.

Our primary goal is to investigate the efficacy of distilling code repair for LRPLs. Along with evaluating the performance of DistiLRR, we also conduct a novel analysis on the wavering benefits of code repair between high-resource and low-resource languages. We hypothesize there exists another bottleneck beyond rationale quality: even if repair models are given high quality rationales, they often fail to fix incorrect code because they lack knowledge on how to convert a suggested plan into

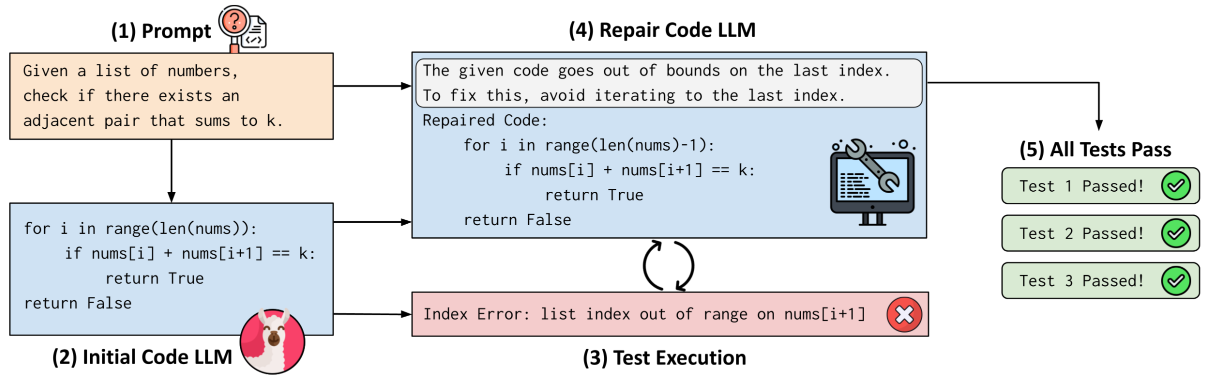


Figure 1: Our code repair framework. In (1) and (2), a code LLM is given a question and generates a solution. In (3), test cases are executed and an error message is extracted. In (4), a repair LLM is given the question, incorrect solution, and error message, and generates a repair. A repair contains a rationale explaining why the old code was incorrect and how to fix it, followed by new code. If the new code is still incorrect, we iteratively generate new repairs using the code from previous repairs. In (5), we stop when all tests pass or after a fixed number of iterations.

specific code modifications. This effect is magnified in a low-resource setting because base models are less knowledgeable on the syntax and semantics of a LRPL.

To demonstrate this, we conduct a comprehensive suite of experiments spanning three HRPLs, three LRPLs, three models, and two benchmarks. Since popular benchmarks like MBPP (Austin et al., 2021) and HumanEval (Chen et al., 2021) are originally in Python, we use variations that have been transcompiled to other programming languages (Athiwaratkun et al., 2023), namely MBXP and MultiLingual HumanEval (HumanEval for brevity). We present our main research questions and findings below.

- **How effective is DistiLRR?** Using DistiLRR models for repair leads to higher pass rates. We see a relative increase in the average pass@1 of CodeLlama-7b-Instruct by 99.5% for Perl, 112.8% for Golang, and 144.5% for Swift after four rounds of repair on HumanEval. We also see a relative increase in the average pass@1 by 69.0% for Python, 44.7% for Javascript, and 49.3% for Java.

- **How effective is transferring code repair for LRPLs compared to HRPLs?** DistiLRR outperforms other distilled code repair baselines on LRPLs, but has similar performance on HRPLs. Compared to supplementing GPT rationales in-context, we see a relative increase in the average pass@1 of CodeLlama-7b-Instruct by 21.9% for Perl, 11.0% for Golang, and 16.3% for Swift on HumanEval.

- **Why are there wavering benefits of code repair between high and low resource languages?** The correlation between rationale quality and code correctness is weaker than previously perceived. The rate at which a repair model provides a good rationale but still produces incorrect code is notably higher than all other outcomes. This occurs in HRPLs with an average rate of 69.9% and LRPLs with an average rate of 76.4%. DistiLRR mitigates this effect, increasing the rate of converting a good rationale into correct code by 31% relative to baselines.

2 Related Work

2.1 Repairing Code with LLMs

Using LLMs to iteratively repair their own responses with the aid of feedback has been a widely applicable area of research, as surveyed in (Pan et al., 2023) and (Fernandes et al., 2023). The efficacy of self correction approaches is also surveyed in (Kamoi et al., 2024). For repairing code generation in specific, frameworks like Self-Edit (Zhang et al., 2023), Self-Debugging (Chen et al., 2023), Self-Repair (Olausson et al., 2024), and Reflexion (Shinn et al., 2023) have shown promising increases in pass rates. Improving upon using an LLM out of the box for repair, ILF (Chen et al., 2024a) upgrades their repair model by fine-tuning on human annotated feedback.

2.2 Distillation for Code Repair

Distillation is the process of transferring knowledge from high capacity models, such as GPT-4

(OpenAI, 2024), to lower capacity models, such as open-source LLMs with 7B parameters or less. Previous works have shown distillation can effectively transfer the ability to generate code and independent reasoning (Sun et al., 2024; Wei et al., 2023b; Xu et al., 2023; Luo et al., 2023; Li et al., 2022a), but transferring the ability to iteratively repair code remains less explored. Recent methods like PERsD (Chen et al., 2024b) distills repaired code to construct a personalized fine-tuning dataset. The aforementioned Self-Repair (Olausson et al., 2024) also conducts an experiment where they transfer rationales from GPT-4 to CodeLlama-13b-Instruct in-context, but still use the base model for code generation. However, neither of these approaches investigates the efficacy of distilling code repair for low-resource languages.

2.3 Low-Resource Programming Languages

Code repair experiments are usually evaluated on high-resource languages like Python, but our work investigates the efficacy of code repair for different languages. For evaluation, many works (Athiwaratkun et al., 2023; Orlanski et al., 2023; Zheng et al., 2023) have created datasets to benchmark code generation in a multilingual setting. Since finding human written low-resource code is difficult, other approaches use capable LLMs to synthetically create low-resource code. Works like MultiPL-T (Cassano et al., 2024) and MultiPL-E (Cassano et al., 2022) translate popular pre-training datasets and monolingual benchmarks into a wide variety of different programming languages. Other works also study the relationship and transferability of coding ability between different languages (Baltaji et al., 2024; Gong et al., 2022).

3 Methodology

DistiLRR augments the normal code repair workflow by replacing the base model with a distilled repair model (DistiLRR model). We first provide an overview of a standard code repair framework, and follow with our process of transferring knowledge between teacher and student.

3.1 Code Repair Framework

We adopt code repair as the base of our framework to improve LRPL code generation. The main components in Figure 1 are the initial code generation, test execution, and iterative repair. We provide a formal explanation for each component.

First, we define M_{init} as the model generating initial answers. For a question q, we obtain $n \geq 10$ initial samples, because it allows us to compute pass@10, along with lower variance pass@1 and pass@5 estimates. We define $c_{t,i}$ as the i-th code sample generated on repair round t, where $t = 0$ denotes the initial generation. Obtaining the initial code generations is formalized in expression 1.

$$M_{init}(q) \rightarrow \{c_{0,i}\}_{i=1}^{n} \quad (1)$$

Next, we define E as the code executor. Given a set of code samples, we execute the test cases associated with q on each sample. This produces a set of error messages, where $e_{t,i}$ is the error message resulting from $c_{t,i}$. If $c_{t,i}$ passes all test cases, $e_{t,i}$ is null. Obtaining the error messages is formalized in expression 2.

$$E(q, \{c_{t,i}\}_{i=1}^{n}) \rightarrow \{e_{t,i}\}_{i=1}^{n} \quad (2)$$

Finally, we define M_{repair} as the model generating repairs. M_{repair} has the same underlying model architecture as M_{init}. A repair is composed of a chain-of-thought (Wei et al., 2023a) rationale $r_{t,i}$, and the associated code $c_{t,i}$. Obtaining a repair on an incorrect code sample is formalized in expression 3.

$$M_{repair}(q, c_{t,i}, e_{t,i}) \rightarrow (r_{t+1,i}, c_{t+1,i}) \quad (3)$$

For one of our baselines, we transfer knowledge in-context by replacing the rationale $r_{t,i}$ from M_{repair} with one from a larger model $M_{teacher}$. In this case, obtaining the teacher's rationale is formalized in expression 4, and obtaining the code from the base model is formalized in expression 5.

$$M_{teacher}(q, c_{t,i}, e_{t,i}) \rightarrow r_{t+1,i} \quad (4)$$

$$M_{repair}(q, c_{t,i}, e_{t,i}, r_{t+1,i}) \rightarrow c_{t+1,i} \quad (5)$$

3.2 Dataset Construction

To strengthen code repair, we transfer the ability to repair from a teacher model to a student model, resulting in a fine-tuned DistiLRR model. Our teacher model is GPT-3.5-Turbo (Ouyang et al., 2022; OpenAI, 2022), while our student models are CodeLlama-7b-Instruct (Rozière et al., 2024), CodeLlama-7b (Rozière et al., 2024), and Mistral-7b (Jiang et al., 2023). The fine-tuning datasets are constructed from MBXP (Athiwaratkun et al., 2023), which consists of multiple language specific

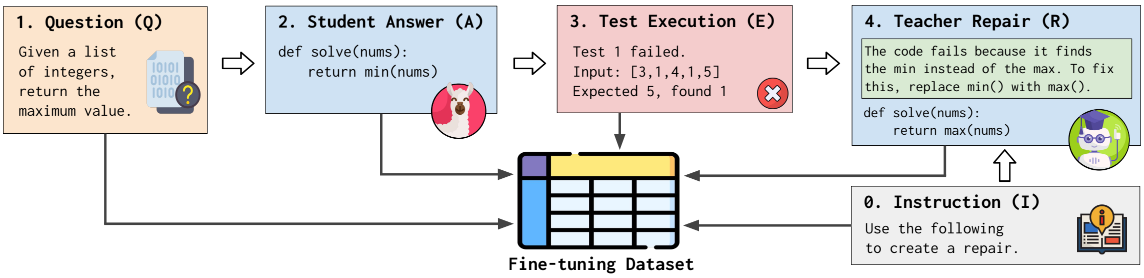


Figure 2: Our dataset construction pipeline. Examples in the fine-tuning dataset contain an instruction, the original question, the student's incorrect answer, the execution feedback, and the teacher's correct repair.

benchmarks, each containing around 960 questions with corresponding test cases. An artificial train-test split is created by taking 800 random examples as potential training data and reserving the rest for testing. We process potential training examples into a finalized dataset, visualized in Figure 2. Our dataset is formally composed of five-tuples in the form (I, Q, A, E, R), which we further explain.

Instruction and Question. Each five-tuple begins with a constant instruction I, informing the model to perform code repair. Next is a question Q, containing a problem description and function declaration. We collect Q by directly using the prompts provided in MBXP.

Answer and Error. The student's incorrect answer is represented with A, which is collected by prompting a student model with Q. To ensure A is incorrect, we allow the student to continually generate i.i.d samples, which are then immediately tested. Once a sample fails the given test cases, we select that sample as A. Then, we collect the associated error message E from the execution feedback.

Repair. Lastly, we finish with R, the teacher model's repair. We collect R by prompting the teacher model to generate a repair given (I, Q, A, E). Following our definition of a repair from Section 3.1, R carries two main components. First, it contains a rationale explaining why the error occurred and a plan to fix it. Second, it contains repaired code based on A, denoted with A'. To ensure A' is correct, we allow the teacher to continually generate i.i.d repairs, which are then immediately tested. Once A' passes the given test cases, we select the associated repair as R.

Quantity of Examples. Although the original train split starts with 800 examples, our construction pipeline results in fine-tuning datasets with around 400 examples. Referencing Figure 2, this is because we may fail to obtain a usable A in step (2) or a usable R in step (4). In step (2), student models may consistently generate correct code. We allow a maximum of 10 samples before discarding the current example. Conversely, in step (4), teacher models may consistently generate incorrect code. We allow a maximum of 20 samples before discarding the current example. When prompting the teacher model, we use few-shot prompting (Brown et al., 2020) with three examples as an attempt to generate better repairs. The exact dataset sizes are listed in Appendix A, and an example of our prompt format can be examined in Appendix D.

4 Experiment

Our goal is to understand the transferability of LLM code repair for HRPLs and LRPLs, so we conduct a comprehensive experiment with three high-resource languages and three low-resource languages. We identify Python, Javascript, and Java as high-resource, and identify Perl, Golang, and Swift as low-resource. These languages are picked based on having the highest three and lowest three pass rates observed in the original MBXP evaluations (Athiwaratkun et al., 2023), as well as cross referencing DeepSeek-Coder's pretraining dataset (Guo et al., 2024), since it loosely reflects the distribution of programming languages found on Github. For each language, we perform our dataset construction and fine-tune a DistiLRR model. Then, we generate an initial round of output and perform four rounds of code repair.

4.1 Experimental Setup

Models. To show DistiLRR generalizes to non-instruction-tuned, non-code-specific, and different model families, we run our experiments on CodeLlama-7b-Instruct (Rozière et al., 2024), CodeLlama-7b (Rozière et al., 2024), and Mistral-7b (Jiang et al., 2023). These models are used for the initial generation, and then a fine-tuned version of the same architecture is used as the DistiLRR model.

Benchmarks. Since we already have a train-test split on MBXP (Athiwaratkun et al., 2023) from Section 3.2, we evaluate on the test split, which contains around 160 programming problems. Additionally, we evaluate on MultiLingual HumanEval (Athiwaratkun et al., 2023), a variation of HumanEval (Chen et al., 2021) transcompiled to different languages, which also contains around 160 programming problems. Our evaluation on MultiLingual HumanEval (HumanEval for brevity) shows that DistiLRR models generalize to other datasets.

Metrics. We evaluate all generations using pass@k (Chen et al., 2021), a standard performance metric for code generation tasks. Since pass@k is prone to high variance, we use the unbiased estimator for pass@k, which estimates the probability that at least one out of k samples is correct. Given $n \geq k$ code samples where c are correct, we compute pass@k using Equation 6.

$$\text{pass@}k := \underset{\text{Problems}}{\mathbb{E}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \tag{6}$$

Training and Inference Details. During training, we perform a 90/10 train-dev split on the dataset resulting from Section 3.2, and train via LoRA fine-tuning (Hu et al., 2022). During the initial generation, we sample 10 answers for each question and compute pass@k using n=10, allowing us to measure certain baselines. However, we only perform code repair on the first 5 samples for later repair rounds and compute pass@1 using n=5, because we only care about the pass@1 for repairs. To encourage diversity between samples, we use nucleus sampling with a threshold of 0.95 and sampling temperature of 0.2. Further training and inference hyperparameters can also be found in Appendix B. For baselines that use a non-fine-tuned model for repair, we use one-shot prompting, whose format is shown in Appendix D.

4.2 Baselines

We compare the pass@1 of the DistiLRR model to five different baselines. These baselines help us investigate how other iterative repair approaches perform on HRPLs vs LRPLs, allowing us to analyze trade-offs and scenarios where DistiLRR works best.

Non-repair i.i.d. Sampling. We compare the efficiency of code repair with i.i.d sampling to see if DistiLRR achieves higher pass rates with fewer inference calls. Our experiment conducts 1 initial generation and 4 repair rounds for a total of 5 inference calls, so we compare the final pass@1 with the pass@5 and pass@10 of the initial generations.

Basic Iterative Repair. DistiLRR augments the preexisting idea of iterative repair with distillation, so we measure how impactful distillation is, along with how well iterative repair with a base model performs on LRPLs. We use the same code repair framework, but replace the DistiLRR model with its non-fine-tuned counterpart.

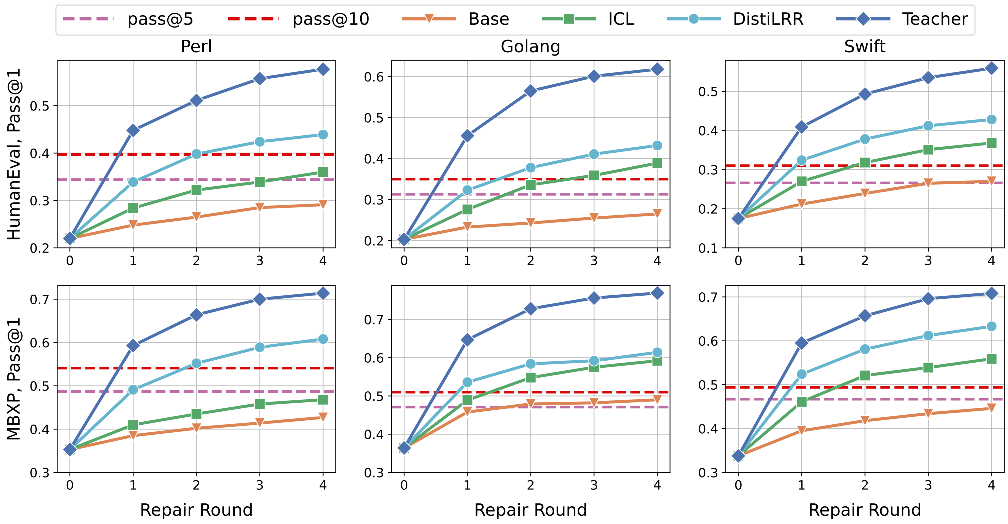
In-Context Teacher Rationales. We compare DistiLRR to an adjacent idea from Self-Repair (Olausson et al., 2024). First, a teacher model is prompted to generate the rationale portion of a repair. Then, a non-fine-tuned student model is prompted to generate the code portion of a repair, with the teacher's rationale appended in-context. For brevity, we refer to this approach as ICL (in-context learning). We use the same teacher and student models as Section 4.1, and our prompt to extract the teacher's rationale is in Appendix C.

Teacher Repair. For demonstrating the limitations of our method, we use the same code repair framework, but replace the DistiLRR model with the teacher model used during dataset construction. This acts as a rough upper bound for the student model, and illustrates potential room for improvement.

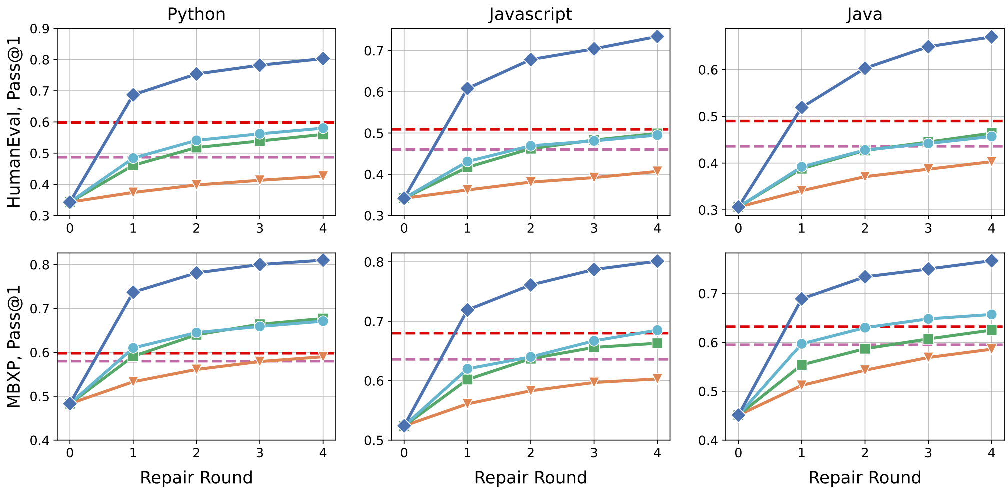
4.3 Results

Our experiments provide empirical results demonstrating the pass@1 improvements of our DistiLRR model, along with a wavering benefits of code repair between HRPLs and LRPLs. We report our results on CodeLlama-7b-Instruct in Figure 3, and similar results on CodeLlama-7b and Mistral-7b can be found in Appendix E and F.

DistiLRR vs i.i.d Sampling. We find that across all languages and both benchmarks, four rounds of code repair with DistiLRR outperforms the initial pass@5. Furthermore, DistiLRR outperforms the initial pass@10, with the exception of HRPLs on HumanEval. Thus, when limited to a small amount of inference calls, DistiLRR can be a more efficient alternative than i.i.d sampling for increasing pass rates.



(a) Low-resource languages



(b) High-resource languages

Figure 3: Average pass@1 versus repair round for CodeLlama-7b-Instruct. Round 0 denotes the initial generation. DistiLRR outperforms ICL on low-resource languages, but performs around the same on high-resource languages.

Impact of Distillation. Both DistiLRR and ICL consistently outperform repair using the base model. One possible reason for this is that both methods produce higher quality rationales, and the correctness of generated code is strongly influenced by the repair model's reasoning. Thus, weaker base models may not benefit as much from frameworks like code repair which rely on diagnosing and rationalizing about mistakes.

Beyond Rationale Quality. Although it is likely intuitive that better rationales result in better code repairs, we show there is more to boosting repair beyond increasing feedback quality. We observe the teacher pass@1 greatly outperforms ICL pass@1, even though the rationales are both generated by GPT-3.5-Turbo. Furthermore, the DistiLRR pass@1 surpasses ICL on LRPLs, despite presumably producing worse rationales than GPT-3.5-Turbo. In other words, higher quality rationales may still lead to incorrect code more often than lower quality rationales. This wavering benefit of rationale quality spurs us to investigate a model's ability to connect rationale suggestions with code modifications.

5 Analysis

After observing varying efficacy of code repair between LRPLs and HRPLs, we want an explanation on why DistiLRR consistently outperforms ICL on low-resource languages, but struggles to outperform on high-resource languages. Table 1 shows

Language	Initial	ICL Repair	DistiLRR
HumanEval Pass@1			
Perl	0.220	0.360 ↑63.6%	**0.439** ↑99.5%
Golang	0.203	0.389 ↑91.6%	**0.432** ↑112.8%
Swift	0.175	0.368 ↑110.2%	**0.428** ↑144.5%
Python	0.343	0.560 ↑63.2%	**0.580** ↑69.0%
Javascript	0.342	**0.499** ↑45.9%	0.495 ↑44.7%
Java	0.306	**0.464** ↑51.6%	0.457 ↑49.3%
MBXP Pass@1			
Perl	0.353	0.468 ↑32.5%	**0.608** ↑77.2%
Golang	0.364	0.592 ↑62.6%	**0.614** ↑68.6%
Swift	0.338	0.559 ↑65.3%	**0.633** ↑87.2%
Python	0.483	**0.677** ↑40.1%	0.671 ↑38.9%
Javascript	0.524	0.663 ↑26.5%	**0.685** ↑30.7%
Java	0.451	0.625 ↑38.5%	**0.657** ↑45.6%

Table 1: Pass@1 of initial generations vs pass@1 after code repair using DistiLRR and ICL. DistiLRR consistently outperforms ICL on LRPLs, but performs around the same or slightly worse on HRPLs. Weak gains are in orange, moderate gains are in light green, and strong gains are in dark green.

quantitative results of this. In the following, we perform two analyses, measuring both the quality of rationales and a model's knowledge of a language.

Previous works like Self-Repair (Olausson et al., 2024) hypothesize that code repair is bottlenecked by the model's underlying ability to create a high quality rationale, which our results support. However, there remains a lacking explanation of why repair models still generate incorrect code, even when given a sufficient rationale.

We hypothesize there exists a second bottleneck: even if repair models are given high quality rationales, they fail to fix incorrect code because they lack the knowledge to convert suggestions into specific code modifications. This effect is magnified in a low-resource setting because base models are less knowledgeable about the nuances of a LRPL, explaining why fine-tuned DistiLRR models outperform ICL.

5.1 Correlation between Rationale and Code

To support our hypothesis that a bottleneck exists in a model's ability to convert suggestions to code, we analyze the relationship between rationale quality and code correctness in Table 2. We quantitatively show that repair models are often exposed to sufficient rationales, yet still generate incorrect code, exposing a weaker correlation between the two than what was previously perceived.

To judge whether a rationale is sufficient or insufficient, we query GPT-4. Although human evaluation would be preferred, finding participants well-versed in languages like Perl and Swift and capable of solving programming problems found in HumanEval is challenging. We selected GPT-4 for our evaluations because the ICL rationales were generated using GPT-3.5-Turbo, and we aimed to use a more advanced model for better assessments.

To obtain judgements, we present a programming question, incorrect code, error message, and rationale to GPT-4, and instruct it to produce a verdict. A rationale is considered good if it contains accurate information and mentions sufficient detail to repair the given code, and bad otherwise. Our judgement prompt can be found in Appendix I. We obtain a verdict for all HumanEval rationales extracted between the initial generation and the first repair round.

From Table 2, we find that the rate of a good rationale leading to incorrect code is notably higher than all other outcomes. We also observe the rate of a good rationale leading to correct code is higher in DistiLRR than in ICL. This suggests that fine-tuning on both rationales and code teaches DistiLRR models to connect feedback with specific code modifications, improving their responsiveness to suggestions. Meanwhile, ICL is performed on frozen LLMs, so although the rationale is augmented, the underlying ability to connect that feedback to specific code modifications is not. We display various examples of this in Appendix L. Thus, this provides a possible explanation for why DistiLRR can outperform ICL, despite having lower quality rationales.

5.2 Knowledge of LRPLs

Finally, we analyze why DistiLRR sees the best improvements on LRPLs. To support the idea that a model's inability to convert suggestions into code modifications is magnified in a low-resource setting, we show that DistiLRR models have deeper understanding of LRPLs, while other baselines do not. We use the frequency of syntax errors as a proxy for knowledge, since generating code with syntax errors is a blatant sign that a model lacks comprehension of a language.

To measure this, we first extract the set of syntax errors from a particular code repair run. Syntax errors are those occurring before execution and caught during compilation or interpretation time. We can conveniently filter out syntax errors by parsing the execution feedback. Next, we compute the average amount of errors within the final repair

	DistiLRR			ICL		
	Code Fails	**Code Passes**	**Total**	**Code Fails**	**Code Passes**	**Total**
LRPLS						
Bad Rationale	12.4%	1.0%	13.4%	8.4%	0.5%	8.9%
Good Rationale	71.2%	15.4%	86.6%	81.6%	9.5%	91.1%
Total	83.6%	16.4%		90.0%	10.0%	
HRPLS						
Bad Rationale	19.7%	2.3%	22.0%	9.3%	0.7%	10.0%
Good Rationale	63.9%	14.1%	78.0%	75.9%	14.1%	90.0%
Total	83.6%	16.4%		85.2%	14.8%	

Table 2: Empirical relationship between rationale quality and code correctness from repair round 1. The rate of a good rationale leading to failing code is notably higher than all other outcomes. Furthermore, the total percent of good rationales produced by DistiLRR is lower than ICL, yet the total percent of passing code is higher.

Language	Initial Errors	Base Repair	ICL Repair	DistiLRR Repair
HumanEval Average Syntax Errors				
Perl	14.5	15.4 ↑0.9	17.8 ↑3.3	**9.20** ↓5.3
Golang	44.7	70.4 ↑25.7	48.7 ↑4.0	**26.6** ↓18.1
Swift	81.0	58.0 ↓23.0	50.4 ↓30.6	**37.4** ↓43.6
Python	12.1	15.6 ↑3.5	18.2 ↑6.1	**14.2** ↑2.1
Javascript	9.10	9.80 ↑0.7	27.6 ↑18.5	**9.00** ↓0.1
Java	39.6	41.2 ↑1.6	**37.0** ↓2.6	41.2 ↑1.6
MBXP Average Syntax Errors				
Perl	12.1	9.50 ↓2.6	13.7 ↑1.6	**2.70** ↓9.4
Golang	33.2	29.2 ↓4.0	26.8 ↓6.4	**14.6** ↓18.6
Swift	60.4	36.0 ↓24.0	27.8 ↓32.6	**11.0** ↓49.4
Python	1.80	5.20 ↑3.4	5.10 ↑3.3	**3.60** ↑1.8
Javascript	4.60	4.20 ↓0.4	11.8 ↑7.6	**3.60** ↓1.0
Java	29.2	26.4 ↓2.8	21.4 ↓5.0	**20.4** ↓8.8

Table 3: Average number of syntax errors after code repair for each baseline. We also provide the deltas between the initial and final amount of errors. On LRPLs, DistiLRR has a higher decline in syntax errors. On HRPLs, DistiLRR performs closer to baselines.

round, along with their absolute differences from the initial generation. Note that non-syntax errors can transform into syntax errors when repair models attempt to update code, leading to occasional increases. The average number of syntax errors for CodeLlama-7b-Instruct can be seen in Table 3, and similar results on CodeLlama-7b and Mistral-7b can be seen in Appendix J and K.

For LRPLs, the decrease in syntax errors with DistiLRR is higher than the other baselines. Averaging over the 3 LRPLs, DistiLRR has a delta of -24.0, ICL has -10.1, and base repair has -4.5. Since DistiLRR models are generating syntactically correct code at a notably higher rate, this suggests that fine-tuned models have better knowledge than base models. Thus, boosting rationale quality alone is not enough for encouraging a base model to generate a working repair, and applying DistiLRR can help transfer knowledge of a programming language.

For HRPLs, the decrease in syntax errors are much smaller or even non-existent, and the amount of errors between DistiLRR, ICL, and base repair are relatively close. Averaged over the 3 HRPLs, DistiLRR has a delta of -0.73, ICL has +4.5, and base repair has +1.0. Since DistiLRR is generating syntactically correct code at a marginally higher rate, this suggests that base models already have sufficient knowledge on HRPLs. Thus, this provides a potential explanation for why DistiLRR outperforms ICL on LRPLs, but performs similarly on HRPLs.

6 Conclusion

We transferred the ability to repair code and demonstrated that DistiLRR achieves better pass rates and knowledge on low-resource languages. We also exposed that the correlation between rationale quality and code correctness is lower than previously perceived. DistiLRR mitigates this weakness by improving a model's understanding of a programming language, resulting in better responsiveness to feedback. Further research in distillation is important because it allows smaller models to gain fluency without costly human labeling, creating efficient and high-performing LLMs suitable for consumer-grade devices. Such advancements would democratize the benefits of closed source research, making better code generation accessible for a wider range of languages, applications, and users.

Limitations

A natural limitation is the lack of instruction tuning datasets for LRPLs. Our constructed fine-tuning datasets only contain around 400 examples, which may be limiting the generalizability of fine-tuned models. Hypothetically, training models with larger datasets could lead to new observations on the efficacy of DistiLRR and derive a better understanding on the scalability of our approach. Nonetheless, we already show noteworthy improvements even with just 400 examples.

One limitation within our evaluation is the lack of more challenging multilingual datasets. Other popular benchmarks like APPS (Hendrycks et al., 2021) and CodeContests (Li et al., 2022b) provide harder problems, which may demand stronger reasoning, but are only available in high-resource languages. Studying the efficacy of DistiLRR on more reasoning heavy questions in low-resource languages would be a good future evaluation.

Another limitation in our evaluation are the stochastic processes within training and inference. To the best of our ability, we mitigate variance in our evaluation by seeding our training and inference, and by using the unbiased estimator of pass@k. However, since we use nucleus sampling for decoding, we observe there can be slight variations in our results.

Lastly, an underlying limitation is our hardware for training and inference. We use Nvidia Titan RTX GPUs with 24GB memory, so the size of student models that we can fine-tune is limited, which is why we choose 7b models for our experiments. Furthermore, since our evaluation has many dimensions (6 languages, 3 models, 5 baselines, 2 benchmarks, 160 questions each benchmark), we are limited in the amount of sampling we can do for each question. Although it may be interesting to obtain higher pass@k rates like k=10 or k=100, these are not time efficient to measure and do not contribute that much to our arguments. Thus, we choose to only show pass@1 for repair rounds.

Ethics Statement

Since computing resources and research funding is extremely valuable, querying costly models like GPT-4 should be conducted responsibly. Estimating costs before running experiments and making necessary adjustments is a responsible and resource-conscious approach to using such APIs.

Furthermore, there exists the possibility that users apply code repair for harmful applications. People with malicious intentions could use our research to improve code generation in certain domains that produce dangerous code, such as attacks on privacy and security. We encourage that code repair and DistiLRR be used for socially responsible technology.

References

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, 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, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Razan Baltaji, Saurabh Pujar, Louis Mandel, Martin Hirzel, Luca Buratti, and Lav Varshney. 2024. Learning transfers over several programming languages. *Preprint*, arXiv:2310.16937.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2024. Knowledge transfer from high-resource to low-resource programming languages for code llms. *Preprint*, arXiv:2308.09895.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *Preprint*, arXiv:2208.08227.

Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R. Bowman, Kyunghyun Cho, and Ethan Perez. 2024a. Improving code generation by training with natural language feedback. *Preprint*, arXiv:2303.16749.

Hailin Chen, Amrita Saha, Steven Hoi, and Shafiq Joty. 2024b. Personalised distillation: Empowering open-sourced llms with adaptive learning for code generation. *Preprint*, arXiv:2310.18628.

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.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *Preprint*, arXiv:2304.05128.

Patrick Fernandes, Aman Madaan, Emmy Liu, António Farinhas, Pedro Henrique Martins, Amanda Bertsch, José G. C. de Souza, Shuyan Zhou, Tongshuang Wu, Graham Neubig, and André F. T. Martins. 2023. Bridging the gap: A survey on integrating (human) feedback for natural language generation. *Preprint*, arXiv:2305.00955.

Zi Gong, Yinpeng Guo, Pingyi Zhou, Cuiyun Gao, Yasheng Wang, and Zenglin Xu. 2022. Multicoder: Multi-programming-lingual pre-training for low-resource code completion. *Preprint*, arXiv:2212.09666.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *Preprint*, arXiv:2401.14196.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *Preprint*, arXiv:2105.09938.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b. *Preprint*, arXiv:2310.06825.

Ryo Kamoi, Yusen Zhang, Nan Zhang, Jiawei Han, and Rui Zhang. 2024. When can llms actually correct their own mistakes? a critical survey of self-correction of llms. *Preprint*, arXiv:2406.01297.

Shiyang Li, Jianshu Chen, Yelong Shen, Zhiyu Chen, Xinlu Zhang, Zekun Li, Hong Wang, Jing Qian, Baolin Peng, Yi Mao, Wenhu Chen, and Xifeng Yan. 2022a. Explanations from large language models make small reasoners better. *Preprint*, arXiv:2210.06726.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022b. Competition-level code generation with alphacode. *Science*, 378(6624).

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. *Preprint*, arXiv:2306.08568.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2024. Is self-repair a silver bullet for code generation? *Preprint*, arXiv:2306.09896.

OpenAI. 2022. Introducing chatgpt. Accessed: 2024-06-14.

OpenAI. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Gabriel Orlanski, Kefan Xiao, Xavier Garcia, Jeffrey Hui, Joshua Howland, Jonathan Malmaud, Jacob Austin, Rishabh Singh, and Michele Catasta. 2023. Measuring the impact of programming language distribution. *Preprint*, arXiv:2302.01973.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. *Preprint*, arXiv:2203.02155.

Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. 2023. Automatically correcting large language models: Surveying the landscape of diverse self-correction strategies. *Preprint*, arXiv:2308.03188.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, 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.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Preprint*, arXiv:2303.11366.

Zhihong Sun, Chen Lyu, Bolun Li, Yao Wan, Hongyu Zhang, Ge Li, and Zhi Jin. 2024. Enhancing code generation performance of smaller models by distilling the reasoning ability of llms. *Preprint*, arXiv:2403.13271.

Jianxun Wang and Yixiang Chen. 2023. A review on code generation with llms: Application and evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pages 284–289.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023a. Chain-of-thought prompting elicits reasoning in large language models. *Preprint*, arXiv:2201.11903.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023b. Magicoder: Source code is all you need. *Preprint*, arXiv:2312.02120.

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. *Preprint*, arXiv:2304.12244.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. *Preprint*, arXiv:2305.04087.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *Preprint*, arXiv:2303.17568.

A Fine-tuning Dataset Sizes

Fine-tuning Dataset Sizes					
Language	Initial	Post-Student	Post-Teacher	Train	Dev
CodeLlama-7b-Instruct					
Perl	800	649	489	440	49
Golang	800	601	455	409	46
Swift	800	635	470	423	47
Python	800	559	446	401	45
Javascript	800	509	394	354	40
Java	800	667	510	459	51
CodeLlama-7b					
Perl	800	680	489	440	49
Golang	800	614	456	410	46
Swift	800	651	465	418	47
Python	800	596	470	423	47
Javascript	800	586	470	423	47
Java	800	642	499	449	50
Mistral-7b					
Perl	800	689	533	479	54
Golang	800	745	539	459	54
Swift	800	625	468	421	47
Python	800	602	487	438	49
Javascript	800	535	413	371	42
Java	800	573	439	395	44

Table 4: The final fine-tuning dataset sizes for each model, starting from the original MBXP train split of 800 questions. Intermediate sizes at each step of our dataset construction are also provided.

B Training and Inference Hyperparameters

We provide our training and inference hyperparameters used throughout experiments. All training and inference are conducted on Nvidia Titan RTX (24GB) GPUs.

For training, we use LoRA fine-tuning with a rank of 128, lora alpha of 128, lora dropout of 0.1, maximum sequence length of 2048, batch size of 4, gradient accumulation steps of 2, weight decay of 0.01, cosine learning rate scheduler with warm up steps of 10, and checkpoint every 50 steps. For models in the CodeLlama family, we train for 8 epochs with a learning rate of 2e-5, and for Mistral-7b, we train for 5 epochs with a learning rate of 5e-6. To obtain our final distilled repair model, we pick the checkpoint with the lowest validation loss.

For inference, we use nucleus sampling with a threshold of 0.95, sampling temperature of 0.2, and limit the maximum new tokens to 800. When generating the initial 10 samples, we use random seeds from 1 to 10. For later repair rounds, all generations use a random seed of 17.

C In-Context Rationale Prompt

```
### Instruction: You are given an incorrect python function and an error message.
Explain how to modify the code to fix the error, but do not write any actual code.
### Incorrect Code:
<Q,A>
### Error: <E>
### Response:
```

Figure 4: The prompt given to GPT-3.5-Turbo to generate the rationale portion of a repair. This is only used for the in-context learning baseline. <Q,A> is replaced with the question and previous answer, while <E> is replaced with the corresponding error.

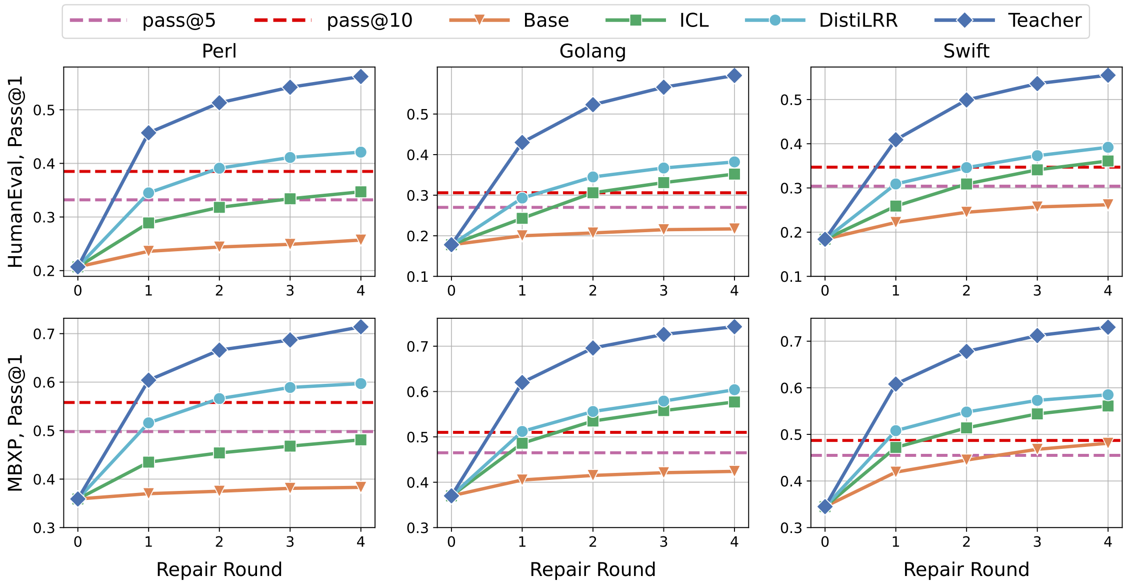
D Repair Prompt

We provide the general format of our repair prompt. When conducting code repair with a DistiLRR model, we use zero-shot prompting. When conducting code repair with our baselines, we use one-shot prompting. When creating a fine-tuning dataset with the teacher model, we use three-shot prompting. The few-shot examples change with each programming language.

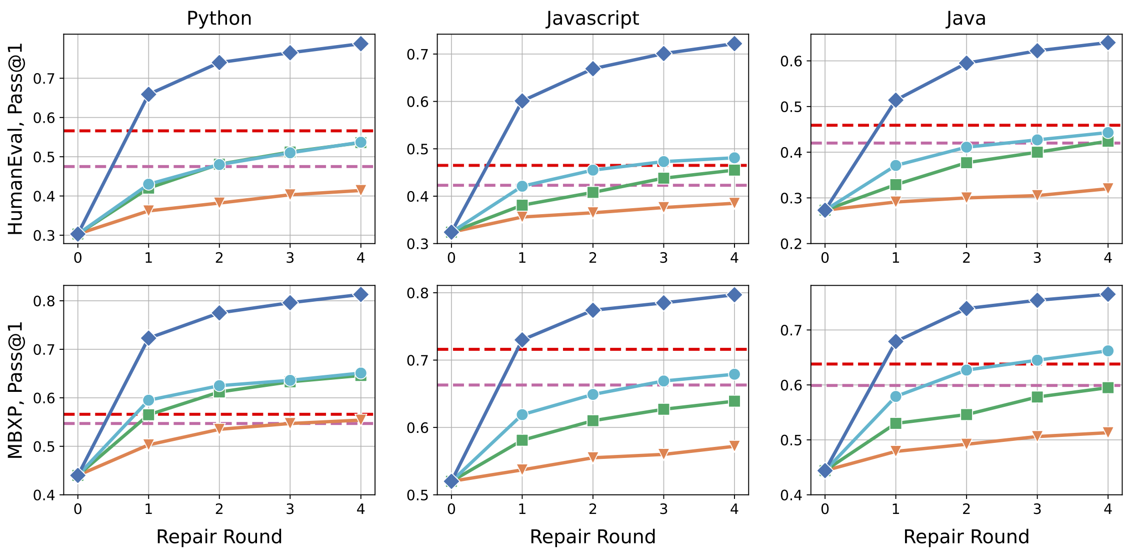
````
### Instruction: You are given an incorrect python function and an error message. Explain how to fix
the error, and then write an updated python function with the correct code.

### Incorrect Code:
def check_String(str):
        """
        Write a python function to check whether a string has at least one letter and one number.
        >>> check_String('thishasboth29')
        True
        >>> check_String('python')
        False
        >>> check_String ('string')
        False
        """
        if (str.isalnum()):
                return True
        else:
                return False

### Error:
Traceback (most recent call last):
  File "temp.py", line 27, in <module>
    check(check_String)
  File "temp.py", line 23, in check
    assert candidate('python') == False
AssertionError

### Response: The incorrect function is returning True for an input of "python", but it should return
False, since there is no number. This is because the incorrect code does not check for both a letter
and number. To fix this, the correct function can use two booleans to find both a letter and a number.
```python
def check_String(str):
        """
        Write a python function to check whether a string has at least one letter and one number.
        >>> check_String('thishasboth29')
        True
        >>> check_String('python')
        False
        >>> check_String ('string')
        False
        """
        has_letter = False
        has_number = False
        for char in str:
                if char.isalpha():
                        has_letter = True
                elif char.isdigit():
                        has_number = True
                if has_letter and has_number:
                        return True
        return False
```

### Instruction: You are given an incorrect python function and an error message. Explain how to fix
the error, and then write an updated python function with the correct code.

### Incorrect Code: <Q,A>

### Error: <E>

### Response:
````

Figure 5: The prompt for generating a repair. For brevity, we only show a one-shot example. <Q,A> is replaced with the question and previous answer, while <E> is replaced with the corresponding error.

E Evaluation on CodeLlama-7b



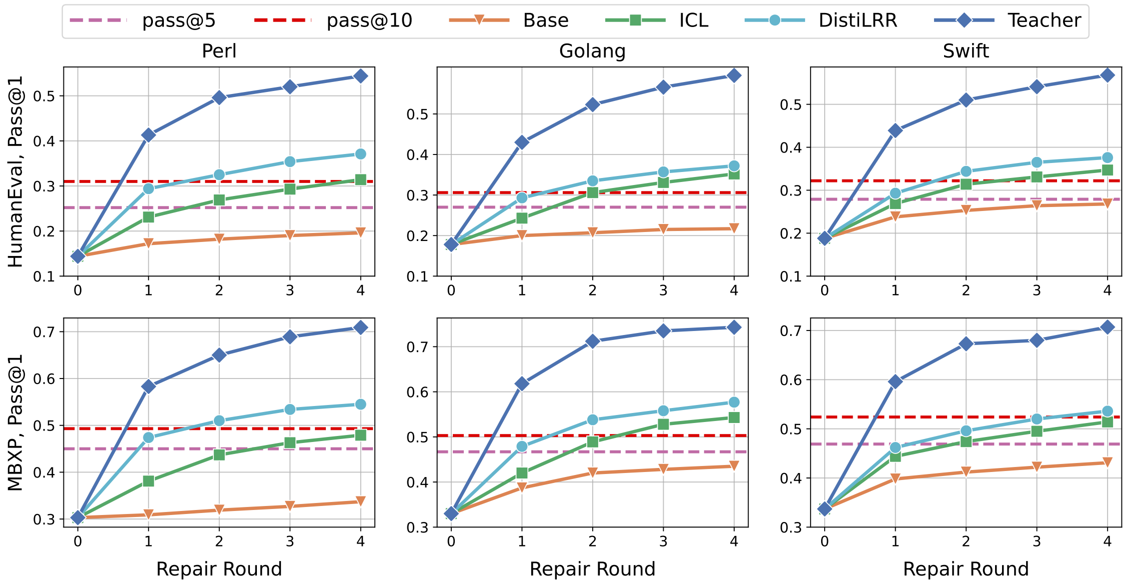
(a) Low-resource languages



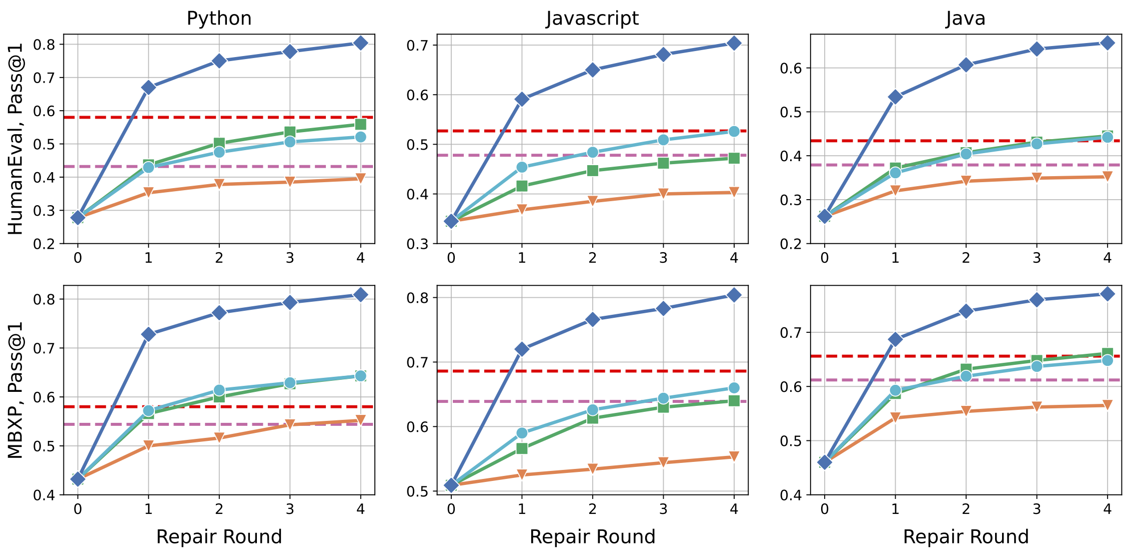
(b) High-resource languages

Figure 6: Average pass@1 versus repair round for CodeLlama-7b. Round 0 denotes the initial generation.

F Evaluation on Mistral-7b



(a) Low-resource languages



(b) High-resource languages

Figure 7: Average pass@1 versus repair round for Mistral-7b. Round 0 denotes the initial generation.

G Pass@1 Comparison on CodeLlama-7b

HumanEval Pass@1			
Language	Initial	ICL Repair	DistiLRR
Perl	0.207	0.347 ↑67.6%	**0.421** ↑103.3%
Golang	0.178	0.352 ↑97.7%	**0.372** ↑108.9%
Swift	0.184	0.361 ↑96.1%	**0.392** ↑113.0%
Python	0.303	0.536 ↑76.8%	**0.537** ↑77.2%
Javascript	0.324	0.455 ↑40.4%	**0.481** ↑48.4%
Java	0.273	0.424 ↑55.3%	**0.443** ↑62.2%
MBXP Pass@1			
Perl	0.359	0.481 ↑33.9%	**0.597** ↑66.2%
Golang	0.370	0.597 ↑61.3%	**0.604** ↑63.2%
Swift	0.345	0.561 ↑62.6%	**0.585** ↑69.5%
Python	0.440	0.646 ↑46.8%	**0.651** ↑47.9%
Javascript	0.520	0.639 ↑22.8%	**0.679** ↑30.5%
Java	0.444	0.595 ↑34.0%	**0.662** ↑49.0%

Table 5: Pass@1 of initial generations vs pass@1 after code repair using ICL and DistiLRR for CodeLlama-7b. Weak gains are in orange, moderate gains are in light green, and strong gains are in dark green.

H Pass@1 Comparison on Mistral-7b

HumanEval Pass@1			
Language	Initial	ICL Repair	DistiLRR
Perl	0.144	0.314 ↑118.0%	**0.371** ↑157.6%
Golang	0.140	0.310 ↑121.4%	**0.321** ↑129.2%
Swift	0.188	0.357 ↑89.8%	**0.366** ↑94.6%
Python	0.278	**0.559** ↑101.0%	0.520 ↑87.0%
Javascript	0.345	0.472 ↑36.8%	**0.526** ↑52.4%
Java	0.262	**0.445** ↑69.8%	0.442 ↑68.7%
MBXP Pass@1			
Perl	0.303	0.479 ↑58.0%	**0.545** ↑79.8%
Golang	0.330	0.543 ↑64.5%	**0.576** ↑74.5%
Swift	0.337	0.514 ↑52.5%	**0.536** ↑59.0%
Python	0.432	**0.643** ↑48.8%	**0.643** ↑48.8%
Javascript	0.509	0.640 ↑25.7%	**0.660** ↑29.6%
Java	0.460	**0.661** ↑43.6%	0.648 ↑40.8%

Table 6: Pass@1 of initial generations vs pass@1 after code repair using ICL and DistiLRR for Mistral-7b. Weak gains are in orange, moderate gains are in light green, and strong gains are in dark green.

I GPT-4 Judgement Prompt

```
### Instruction: You are given an incorrect python function, an error message, and a
rationale to fix the error. Classify if the rationale is 'Good' or 'Bad'. If the rationale
provides enough detail to fix the code, output 'Good'. Otherwise, output 'Bad'.
### Incorrect Code: <Q,A>
### Error: <E>
### Rationale: <R>
### Response:
```

Figure 8: The prompt given to GPT-4 to judge rationale sufficiency. <Q,A> is replaced with the question and previous answer, <E> is replaced with the corresponding error, and <R> is replaced with the repair model's rationale.

J Syntax Errors for CodeLlama-7b

HumanEval Average Syntax Errors					
Language	Initial Errors	Base Repair	ICL Repair	DistiLRR Repair	Teacher Repair
Perl	21.2	21.0 ↓0.2	20.6 ↓0.6	12.8 ↓8.4	**7.2** ↓14.0
Golang	39.1	72.8 ↑33.7	36.6 ↓2.5	30.8 ↓8.3	**22.2** ↓16.9
Swift	78.1	57.2 ↓20.9	47.0 ↓31.1	48.4 ↓29.7	**40.4** ↓37.7
Python	17.1	22.4 ↑5.3	23.5 ↑6.4	12.5 ↓4.6	**7.8** ↓9.3
Javascript	10.6	10.0 ↓0.6	28.6 ↑18.0	13.2 ↑2.6	**5.4** ↓5.2
Java	44.7	55.0 ↑10.3	44.6 ↓0.1	42.4 ↓2.3	**20.4** ↓24.3
MBXP Average Syntax Errors					
Perl	16.4	15.0 ↓1.4	15.7 ↓0.7	6.2 ↓10.2	**3.5** ↓12.9
Golang	30.7	48.2 ↑17.5	18.2 ↓12.5	15.2 ↓15.2	**12.6** ↓18.1
Swift	62.4	37.6 ↓24.8	22.6 ↓39.8	**19.0** ↓43.4	19.4 ↓43.0
Python	2.3	7.4 ↑5.1	5.7 ↑3.4	2.5 ↑0.2	**1.3** ↓1.0
Javascript	7.1	5.0 ↓2.1	14.6 ↑7.5	7.0 ↓0.1	**2.0** ↓5.1
Java	31.4	33.2 ↑1.8	24.0 ↓7.4	20.6 ↓10.8	**9.6** ↓21.8

Table 7: Average number of syntax errors after code repair for CodeLlama-7b. We also include a column containing results from the GPT-3.5-Turbo teacher. We continue to see the trend where DistiLRR sees larger declines than ICL for LRPLs.

K Syntax Errors for Mistral-7b

HumanEval Average Syntax Errors					
Language	Initial Errors	Base Repair	ICL Repair	DistiLRR Repair	Teacher Repair
Perl	26.4	30.4 ↑4.0	24.0 ↓2.4	11.0 ↓15.4	**9.4** ↓17.0
Golang	55.9	72.4 ↑16.5	48.2 ↓7.7	31.0 ↓24.9	**25.2** ↓30.7
Swift	62.0	60.0 ↓2.0	54.0 ↓8.0	55.4 ↓6.6	**39.8** ↓22.2
Python	14.5	17.2 ↑2.7	12.4 ↓2.1	13.5 ↓1.0	**8.0** ↓6.5
Javascript	6.7	**7.4** ↑0.7	16.6 ↑9.9	7.8 ↑1.1	7.8 ↑1.1
Java	41.4	42.2 ↑0.8	36.2 ↓5.2	31.4 ↓10.0	**19.2** ↓22.2
MBXP Average Syntax Errors					
Perl	26.3	25.2 ↓1.1	24.0 ↓2.3	6.0 ↓20.3	**4.2** ↓22.1
Golang	43.6	40.0 ↓3.6	27.2 ↓16.4	13.8 ↓29.8	**13.2** ↓30.4
Swift	49.3	36.6 ↓12.7	32.6 ↓16.7	25.6 ↓23.7	**21.6** ↓27.7
Python	0.9	3.6 ↑2.7	4.2 ↑3.3	3.2 ↑2.3	**2.2** ↑1.3
Javascript	7.8	7.6 ↓0.2	10.4 ↑2.6	7.4 ↓0.4	**3.6** ↓4.2
Java	29.9	15.0 ↓14.9	11.2 ↓18.7	14.0 ↓15.9	**7.2** ↓22.7

Table 8: Average number of syntax errors after code repair for Mistral-7b. We also include a column containing results from the GPT-3.5-Turbo teacher. We continue to see the trend where DistiLRR sees larger declines than ICL for LRPLs.

L DistiLRR vs Baselines Examples

We hypothesize base models struggle to convert suggestions from the rationale into specific code modifications. We provide examples of this for each language. In each example, ICL is provided a stellar rationale but generates incorrect code, while DistiLRR produces a decent rationale but generates correct code. These examples support that the correlation between rationale quality and code correctness is weaker than previously perceived, and that DistiLRR teaches models to respond better to feedback.

```
### Instruction: You are given an incorrect perl
function and an error message. Explain how to fix
the error, and then write an updated perl
function with the correct code.


### Incorrect Code:
# You are an expert Perl programmer, and here is
your task.
# # remove_vowels is a function that takes string
and returns string without vowels.
# >>> remove_vowels('')
# ''
# >>> remove_vowels("abcdef\nghijklm")
# 'bcdf\nghjklm'
# >>> remove_vowels('abcdef')
# 'bcdf'
# >>> remove_vowels('aaaaa')
# ''
# >>> remove_vowels('aaBAA')
# 'B'
# >>> remove_vowels('zbcd')
# 'zbcd'
#
sub remove_vowels
{
  my ($text) = @_;
  $text =~ s/[aeiou]//g;
  return $text;
}


### Error: Exception -- test case 4 did not pass.
at temp.pl line 57.


### Response:
```

Repair Prompt

```
The incorrect function is using the wrong regex pattern to
remove vowels. The correct function should use the pattern
"[aeiou]".

sub remove_vowels
{
  my ($text) = @_;
  $text =~ s/[aeiou]//g;
  return $text;
}
```

Base Repair

```
To fix the error, we need to modify the regular expression
pattern used in the substitution to be case-insensitive.
Currently, it only matches lowercase vowels, so we need to
make it match both lowercase and uppercase vowels. This
can be achieved by adding the 'i' flag at the end of the
pattern.

sub remove_vowels
{
  my ($text) = @_;
  $text =~ s/[aeiou]/i//g;
  return $text;
}
```

ICL Repair

```
The issue with the incorrect code is that it is not
handling the case where the input string contains uppercase
vowels. To fix this, we need to modify the regular
expression to include uppercase vowels as well.

sub remove_vowels
{
    my ($text) = @_;
    $text =~ s/[aeiouAEIOU]//g;
    return $text;
}
```

DistiLRR Repair

Figure 9: Perl example from HumanEval question 51. The initial code is wrong because it does not remove uppercase vowels. In base repair, we see a weak rationale that fails to diagnose the uppercase issue. In ICL, we see a stellar rationale that proposes using the "i" regex flag for case insensitivity. However, the generated code incorrectly modifies the regex to "/[aeiou]/i//g" instead of "/[aeiou]//gi", displaying a lack of knowledge. In DistiLRR, we see a good rationale that suggests adding uppercase letters to the regex, followed by correct code modifications.

```
### Instruction: You are given an incorrect
golang function and an error message. Explain
how to fix the error, and then write an updated
golang function with the correct code.


### Incorrect Code:
// You're an expert Golang programmer
// Return the largest prime factor of n.
Assume n > 1 and is not a prime.
// >>> largest_prime_factor(13195)
// 29
// >>> largest_prime_factor(2048)
// 2
//
func largest_prime_factor (n int) int {

    for n % 2 == 0 {
        n = n / 2
    }
    for n % 3 == 0 {
        n = n / 3
    }
    for n % 5 == 0 {
        n = n / 5
    }
    for n % 7 == 0 {
        n = n / 7
    }
    return n
}


### Error: panic: Exception --- test case 0
failed to pass


### Response:
```

Repair Prompt

```
The incorrect function is wrong because it does not check for the prime factors of
n. To fix this, the correct function should check for the prime factors of n, and
return the largest one.
func largest_prime_factor (n int) int {
    for i := 2; i < n; i++ {
        if n % i == 0 {
            return i
        }
    }
    return n
}
```

Base Repair

```
The incorrect code is not correctly finding the prime factors of "n". Instead of
hardcoding prime factors, you need to generalize the loop to check for all possible
prime factors.
One way to achieve this is to iterate through all numbers starting from 2 up to the
square root of "n" and check if each number is a factor of "n" and a prime number.
This way, we can ensure that we are finding the largest prime factor of the given
number "n" accurately.
func largest_prime_factor (n int) int {
    if n == 1 {
        return 1
    }
    if n % 2 == 0 {
        return 2
    }
    for i := 3; i <= int(n ** 0.5); i += 2 {
        for n % i == 0 {
            n /= i
        }
    }
    return n
}
```

ICL Repair

```
Test case 0 failed because the function is not correctly finding the largest prime
factor of n. The function is currently returning n instead of the actual largest
prime factor. To fix this, we need to update the function to iterate over all
prime factors of n and return the largest one. Here is the corrected code:
func largest_prime_factor(n int) int {
    if n <= 1 {
        return n
    }
    for i := 2; i*i <= n; i++ {
        if n%i == 0 {
            n = n / i
            i--
        }
    }
    return n
}
```

DistiLRR Repair

Figure 10: Golang example from HumanEval question 59. The initial code is wrong because it hardcodes prime factors. In base repair, we see a weak rationale that is unable to diagnose the hardcoding issue. In ICL, we see a stellar rationale that suggests iterating over all numbers from 2 to $\sqrt{n}$. However, the generated code results in a compilation error due to using "n $**$ 0.5" to obtain the square root (which does not work in golang), displaying a lack of knowledge. In DistiLRR, we see a good rationale that provides a plan of iterating over prime factors, followed by correct code modifications.

```
### Instruction: You are given an incorrect swift function
and an error message. Explain how to fix the error, and then
write an updated swift function with the correct code.



### Incorrect Code:
/**
 * You are an expert Swift programmer, and here is your task.
 * Return list of prime factors of given integer in the order
from smallest to largest.
 * Each of the factors should be listed number of times
corresponding to how many times it appears in factorization.
 * Input number should be equal to the product of all factors
 * >>> factorize(8)
 * [2, 2, 2]
 * >>> factorize(25)
 * [5, 5]
 * >>> factorize(70)
 * [2, 5, 7]
 *
 */
func factorize(n : Int) -> [Int] {

    var factors = [Int]()
    var i = 2
    while i * i <= n {
        while n % i == 0 {
            factors.append(i)
            n /= i
        }
        i += 1
    }
    if n > 1 {
        factors.append(n)
    }
    return factors
}



### Error: temp.swift:21:15: error: left side of mutating
operator isn't mutable: 'n' is a 'let' constant
            n /= i
            ~ ^



### Response:
```

Repair Prompt

```
The incorrect function is trying to modify the input number, which is a
constant. To fix this, the correct function should return a new array of
prime factors instead of modifying the input number.
func factorize(n : Int) -> [Int] {
    var factors = [Int]()
    var i = 2
    while i * i <= n {
        while n % i == 0 {
            factors.append(i)
            n /= i
        }
        i += 1
    }
    if n > 1 {
        factors.append(n)
    }
    return factors
}
```

Base Repair

```
To fix the error in the given Swift function, you need to change the
parameter declaration of `n` from `let` to `var`. This change will make `n`
mutable so that you can reassign it when performing the division operation
inside the while loop. By making this change, you will be able to update the
value of `n` during the factorization process.
func factorize(n : Int) -> [Int] {
    var factors = [Int]()
    var i = 2
    while i * i <= n {
        while n % i == 0 {
            factors.append(i)
            n /= i
        }
        i += 1
    }
    if n > 1 {
        factors.append(n)
    }
    return factors
}
```

ICL Repair

```
The error is due to trying to modify a constant 'n' in the function. To fix
this, we need to create a mutable copy of 'n' before modifying it. Here is
the corrected function:
func factorize(n: Int) -> [Int] {
    var n = n
    var factors = [Int]()
    var i = 2
    while i * i <= n {
        while n % i == 0 {
            factors.append(i)
            n /= i
        }
        i += 1
    }
    if n > 1 {
        factors.append(n)
    }
    return factors
}
```

DistiLRR Repair

Figure 11: Swift example from HumanEval question 25. The initial code is wrong because it tries to modify the immutable input variable n. For swift, input parameters are immutable by default. In base repair, we see a weak rationale that does not specify how to fix the error. In ICL, we see a stellar rationale that explicitly mentions using the "var" keyword. However, the generated code remains the same because the base model is unable to make the necessary code modifications, displaying a lack of responsiveness. In DistiLRR, we see a good rationale that proposes making a mutable copy, followed by correct code modifications through the additional line "var n = n".

```
### Instruction: You are given an incorrect
python function and an error message. Explain
how to fix the error, and then write an updated
python function with the correct code.


### Incorrect Code:
def digits(n):
    """Given a positive integer n, return the
product of the odd digits.
    Return 0 if all digits are even.
    For example:
    digits(1)  == 1
    digits(4)  == 0
    digits(235) == 15
    """
    s = str(n)
    product = 1
    for digit in s:
        if int(digit) % 2 == 1:
            product *= int(digit)
    return product



### Error: Traceback (most recent call last):
  File "temp.py", line 41, in <module>
    check(digits)
  File "temp.py", line 38, in check
    assert candidate(2468) == 0
           ^^^^^^^^^^^^^^^^^^^^
AssertionError



### Response:
```

Repair Prompt

```
The incorrect function is returning 0 for an input of
2468, but it should return 15, since there are 15 odd
digits. This is because the incorrect code does not check
for odd digits. To fix this, the correct function can use
the modulo operator to check for odd digits.
def digits(n):
    s = str(n)
    product = 1
    for digit in s:
        if int(digit) % 2 == 1:
            product *= int(digit)
    return product
```

Base Repair

```
The error occurs when the function is called with an even
number and it fails the test case for an even number. To
fix this error, you need to modify the code to handle the
case where all digits are even. Specifically, you should
return 0 when there are no odd digits in the given number,
rather than returning 1.
def digits(n):
    s = str(n)
    product = 1
    for digit in s:
        if int(digit) % 2 == 1:
            product *= int(digit)
    if product == 1:
        return 0
    return product
```

ICL Repair

```
The error in the code is that the function is not correctly
handling the case where all digits are even. The function
should return 0 if all digits are even. To fix this, we
need to update the logic to check if all digits are even
before returning the product. Here is the corrected code:
def digits(n):
    s = str(n)
    product = 1
    odd_count = 0
    for digit in s:
        if int(digit) % 2 == 1:
            odd_count += 1
            product *= int(digit)
    if odd_count == 0:
        return 0
    return product
```

DistiLRR Repair

Figure 12: Python example from HumanEval question 131. The initial code is wrong because it returns 1 if there are no odd digits, instead of 0. In base repair, we see a weak rationale that states wrong information. In ICL, we see a stellar rationale that addresses the edge case and suggests returning 0 if there are no odd digits. However, the generated code implements this incorrectly, displaying weak responsiveness. In DistiLRR, we see a good rationale that suggests checking if all digits are even, followed by correct code modifications via the additional variable "odd_count".

```
### Instruction: You are given an incorrect javascript function and an error message. Explain how to fix the error, and then write an updated javascript function with the correct code.


### Incorrect Code:
/**
 * Evaluate whether the given number n can be written as the sum of exactly 4 positive even numbers
 * Example
 * is_equal_to_sum_even(4) == False
 * is_equal_to_sum_even(6) == False
 * is_equal_to_sum_even(8) == True
 *
 */
function isEqualToSumEven(n) {
 for (let i = 2; i <= n; i += 2) {
    if (n - i === 2) {
      return true;
    }
  }
  return false;
}


### Error:
temp.js:29
    throw 'Error at 1th assert statement. Value = ' + JSON.stringify(x0)
    ^
Error at 1th assert statement. Value = true
(Use `node --trace-uncaught ...` to show where the exception was thrown)


### Response:
```

Repair Prompt

```
The incorrect code is using a for loop to iterate through the numbers from 2 to n, and then checking if the number is equal to 2. However, the function description wants to check if the number can be written as the sum of exactly 4 positive even numbers. To fix this, the correct code should use a different approach to check if the number can be written as the sum of 4 positive even numbers.
function isEqualToSumEven(n) {
    let arr = [2, 4, 6, 8];
    let sum = 0;
    for (let i = 0; i < arr.length; i++) {
        if (n - arr[i] === 0) {
            return true;
        }
        sum += arr[i];
    }
    return false;
}
```

Base Repair

```
The error message is indicating that the code provided is not returning the correct output for the test case where `n` is 8. Currently, the code checks if `n - i` is equal to 2, which doesn't cover all possible combinations. You should update the logic in the function to iterate over possible combinations of 4 positive even numbers and check if their sum equals `n`. You can do this by considering all possible even numbers less than or equal to `n/4` and checking if there exists a combination of 4 even numbers that sum up to `n`. By making these adjustments, you can ensure that the function correctly checks if the given number `n` can be represented as the sum of exactly 4 positive even numbers.
function isEqualToSumEven(n) {
  let count = 0;
  for (let i = 2; i <= n; i += 2) {
    if (count === 4) return true;
    for (let j = 2; j <= n; j += 2) {
      if (count === 4) return true;
      for (let k = 2; k <= n; k += 2) {
        if (count === 4) return true;
        for (let l = 2; l <= n; l += 2) {
          if (count === 4) return true;
          if (i + j + k + l === n) {
            count++;
          }
        }
      }
    }
  }
  return false;
}
```

ICL Repair

```
The error indicates that the function is failing the first test case. The issue lies in the logic of the function, which is not correctly checking if the given number can be written as the sum of exactly 4 positive even numbers. To fix this, we need to adjust the condition in the loop to ensure that the sum of 4 even numbers is equal to the given number. Here is the corrected code for the function:
function isEqualToSumEven(n) {
    for (let i = 2; i <= n; i += 2) {
        for (let j = i + 2; j <= n; j += 2) {
            for (let k = j + 2; k <= n; k += 2) {
                if (k + 2 === n) {
                    return true;
                }
            }
        }
    }
    return false;
}
```

DistiLRR Repair

Figure 13: Javascript example from HumanEval question 138. The initial code is wrong because it checks if n is a sum of 2 even numbers, instead of 4. In base repair, we see a decent rationale that lacks a highly detailed plan. In ICL, we see a stellar rationale suggesting a highly detailed plan. However, the generated code implements this incorrectly by misusing the "count" variable, displaying weak responsiveness. In DistiLRR, we see a decent rationale that lacks a highly detailed plan, but correct code modifications anyways, displaying strong responsiveness.

```
### Instruction: You are given an incorrect java function and an error message. Explain how to fix the error, and then write an updated java function with the correct code.


### Incorrect Code:
/**
 * This function takes a list l and returns a list l' such that
 * l' is identical to l in the odd indicies, while its values at the even indicies are equal
 * to the values of the even indicies of l, but sorted.
 * >>> sort_even([1, 2, 3])
 * [1, 2, 3]
 * >>> sort_even([5, 6, 3, 4])
 * [3, 6, 5, 4]
 *
 */
public static List<Integer> sortEven(List<Integer> l) {
    List<Integer> l2 = new ArrayList<Integer>();
    for (int i = 0; i < l.size(); i++) {
        if (i % 2 == 0) {
            l2.add(l.get(i));
        }
    }
    Collections.sort(l2);
    for (int i = 0; i < l.size(); i++) {
        if (i % 2 != 0) {
            l2.add(l.get(i));
        }
    }
    return l2;
}


### Error:
Exception in thread "main" java.lang.Exception: Exception -- test case 0 did not pass. x0 = [1, 3, 2]
        at Main.main(Main.java:24)


### Response:
```

Repair Prompt

```
The incorrect function is trying to sort the even indices of the list, but it is not correctly identifying the even indices. To fix this, the correct function should use the modulo operator to check if the index is even.
public static List<Integer> sortEven(List<Integer> l) {
    List<Integer> l2 = new ArrayList<Integer>();
    for (int i = 0; i < l.size(); i++) {
        if (i % 2 == 0) {
            l2.add(l.get(i));
        }
    }
    Collections.sort(l2);
    for (int i = 0; i < l.size(); i++) {
        if (i % 2 != 0) {
            l2.add(l.get(i));
        }
    }
    return l2;
}
```

Base Repair

```
The error message indicates that the test case [1, 3, 2] did not pass. To fix this issue in the code, the function sortEven should correctly handle the even and odd indices when constructing the new list l2. Make sure that the values at even indices of l are added to l2 in sorted order, while the values at odd indices are retained from the original list. This adjustment will ensure that the function behaves as intended for sorting even indices.
public static List<Integer> sortEven(List<Integer> l) {
    List<Integer> l2 = new ArrayList<Integer>();
    for (int i = 0; i < l.size(); i++) {
        if (i % 2 == 0) {
            l2.add(l.get(i));
        } else {
            l2.add(l.get(i));
        }
    }
    Collections.sort(l2);
    return l2;
}
```

ICL Repair

```
The error indicates that the test case is failing because the output is not correct. The issue in the incorrect code is that the sorting is happening incorrectly. To fix this, we need to sort the even indices of the list separately and then add them back to the original list in the correct order.
public static List<Integer> sortEven(List<Integer> l) {
    List<Integer> l2 = new ArrayList<Integer>();
    for (int i = 0; i < l.size(); i++) {
        if (i % 2 == 0) {
            l2.add(l.get(i));
        }
    }
    Collections.sort(l2);
    for (int i = 0; i < l.size(); i++) {
        if (i % 2 == 0) {
            l.set(i, l2.get(i / 2));
        }
    }
    return l;
}
```

DistiLRR Repair

Figure 14: Java example from HumanEval question 37. The initial code is wrong because it first appends even indices and then appends odd indices, instead of interleaving them. In base repair, we see a weak rationale and no code modifications. In ICL, we see a stellar rationale suggesting to interleave odd/even indices. However, the generated code incorrectly implements the sorting of even indices by sorting the entire list at the end, displaying weak responsiveness. In DistiLRR, we see a good rationale suggesting to interleave odd/even indices, followed by correct code modifications.