Bridge-Coder: Transferring Model Capabilities from High-Resource to Low-Resource Programming Language

Anonymous ACL submission

Abstract

Most LLMs universally excel at generating code for high-resource programming languages (HRPLs) like `Python`, a capability that has become standard due to the abundance of training data. However, they struggle significantly with low-resource programming languages (LRPLs) such as `D`, exacerbating the digital divide. This gap limits developers using LRPLs from equally benefiting and hinders innovation within underrepresented programming communities. To make matters worse, manually generating data for LRPLs is highly labor intensive and requires expensive expert effort. In this work, we begin by analyzing the NL-PL Gap, where LLMs' direct-generated LRPL data often suffers from subpar quality due to the misalignment between natural language (NL) instructions and programming language (PL) outputs. To address this issue, we introduce *Bridge-Assist Generation*, a method to generate LRPL data utilizing LLM's general knowledge, HRPL proficiency, and in-context learning capabilities. To further maximize the utility of the generated data, we propose *Bridged Alignment* to obtain **Bridge-Coder**. To thoroughly evaluate our approach, we select four relatively LRPLs: `R`, `D`, `Racket`, and `Bash`. Experimental results reveal that Bridge-Coder achieves significant improvements over the original model, with average gains of 18.71 and 10.81 on two comprehensive benchmarks, M-HumanEval and M-MBPP.

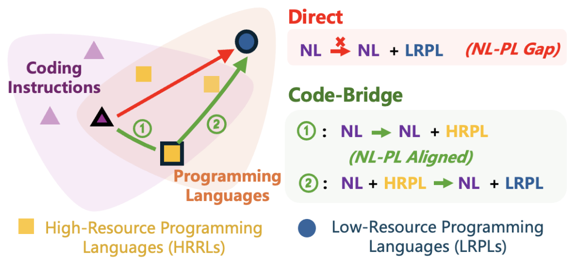

Figure 1: A comparison between `Direct` and the `Code-Bridge` approach in solving coding-related tasks. When responding to such tasks, models' response involves both Natural Language (NL) and Programming Language (PL). However, due to the lack of training data in LRPLs, the *NL-PL Gap* makes it challenging for models to generate accurate responses directly. In contrast, `Code-Bridge` mitigates this gap by leveraging the model's proficiency in HRPLs to generate an intermediate NL-PL aligned process. This intermediate step serves as a bridge, guiding the model to produce more accurate and coherent responses in LRPLs.

1 Introduction

Large Language Models (LLMs) have shown remarkable capabilities (Chen et al., 2021b) in assisting with coding tasks such as code generation (Austin et al., 2021a), debugging (Zhong et al., 2024; Xia et al., 2024), code explanation (Nam et al., 2023). With the introduction of tools like GitHub Copilot (Microsoft, 2023; Services, 2023), models like OpenAI Codex (Chen et al., 2021b) have greatly enhanced the efficiency of developers by automating repetitive tasks, providing real-time code suggestions, and offering in-depth explanations of code functionality.

However, when it comes to low-resource programming languages (LRPLs), the instruction-following abilities of LLMs are significantly diminished (Cassano et al., 2022). This limits LLMs' ability to effectively support developers working with LRPLs (Zheng et al., 2023; Orlanski et al., 2023), preventing these developers from fully benefiting from the advanced capabilities that LLMs provide for High-Resource Programming Languages (HRPLs). This uneven distribution of benefits exacerbates digital inequality, further widening the gap between developers in different programming ecosystems.

While research has been conducted on low-resource natural languages (LRNLs) (Xue, 2020; Lample, 2019; Huang et al., 2019; Conneau et al., 2019; Hu et al., 2020), LRPLs remain relatively underexplored and present distinct challenges. First, unlike NLs, where one language can address a broad range of tasks, specific PLs are often designed for specialized domains, limiting their versatility. Second, programming demands greater attention to the logical structure and coherence of PL. Finally, programming tasks require generating responses that seamlessly combine both NL (e.g., instructions or comments) and PL (the actual code), adding a significant layer of complexity to LRPLs.

As illustrated in Figure 1, this interplay between NL and PL introduces what we refer to as the NL-PL Gap—a disconnect arising from the need to align NL instructions with outputs in both NL and PL. While LLMs exhibit some proficiency in LRPLs derived from indirect sources, this gap limits their fully ability and often results in suboptimal outputs. However, by first leveraging LLMs' capabilities in HRPLs to generate an intermediate Code-Bridge, the model can produce more accurate responses. The Code-Bridge, which contains both the HRPL solution and NL-formatted comments explaining it, replaces purely NL-based instructions. This approach offers two key benefits: (1) enhancing task comprehension by incorporating both NL and PL information, and (2) leveraging PL's logical structure to guide the generation of more accurate and coherent outputs in LRPLs.

In light of this, we propose a two-stage approach to develop an enhanced model, **Bridge-Coder**, for improving performance in LRPLs. The first stage, *Bridge-Assisted Generation*, begins by leveraging LLMs' general knowledge for task screening to ensure that the selected tasks are answerable within the LRPL context. We then synthesize the Code-Bridge and let LLMs use in-context learning abilities to reference this bridge when responding to instructions in LRPLs, enabling it to generate more accurate and coherent results. Given the existence of the NL-PL Gap, it is critical to effectively utilize this data to help LLMs better address LRPL tasks. To this end, the second stage, *Bridged Alignment*, begins with assisted alignment, where intermediate guidance is provided to help the LLM gradually bridge the NL-PL Gap, avoiding an abrupt learning leap. This is followed by direct alignment, which focuses on enhancing the model's ability to independently respond to NL instructions in LRPLs.

To thoroughly evaluate our approach, we select four functionally diverse yet relatively low-resource programming languages (LRPLs): `R`, `D`, `Racket`, and `Bash`. We conduct experiments on two comprehensive benchmarks, M-HumanEval and N-MBPP, each featuring hundreds of tasks per language that require passing numerous test cases to ensure correctness. The results demonstrate that our method significantly enhances the model's performance in LRPLs. Additionally, we perform extensive experiments to validate the technical choices within our approach, which may offer valuable insights into the underexplored domain of low-resource programming languages.

In summary, our contributions are:

- We identify the NL-PL Gap as the primary factor behind LLMs' poor performance in LRPLs. This gap emerges due to programming language datasets containing both PL and NL components (e.g., instructions, comments), complicating the alignment between NL instructions and PL outputs.

- We introduce a two-stage method to obtain `Bridge-Coder` that has enhanced performance in LRPLs by first leveraging LLMs' potential to generate high-quality data, then following assist and direct alignment steps.

- Through extensive experiments across various LRPLs, we demonstrate the effectiveness and generalization of `Bridge-Coder`. Moreover, we provide valuable insights that can guide future research in the underexplored field of low-resource programming languages.

2 Related Work

2.1 Code LLMs

Foundation Models. Training on code samples with billions of tokens using hundreds of high-performance GPUs, decoder-only code foundation LLMs have been proven to have strong code generation ability across various tasks. Specifically, Codex (Chen et al., 2021a) is OpenAI's earliest domain-specific LLM for coding and is believed to support the Copilot service, which helps with automatic code completion across different IDEs (Microsoft, 2023). Additionally, the open-source community has developed a series of code LLMs, such as InCoder (Fried et al., 2022) and CodeT5 (Wang et al., 2021), to further support the development of stronger or domain-specific code assistants. More precisely, Deepseek-coder (Guo et al., 2024) family models and StarCoder (Li et al., 2023) family models trained their model parameters from scratch with trillions of tokens scraped from web pages related to code. Code-Llama (Roziere et al., 2023) and Code-Qwen (Hui et al., 2024) family models perform post-training from general-purpose models with code-related datasets to achieve high-performance code foundation models.

Downstream Models. Besides developing code foundation models, researchers often finetune these code foundation models for specific applications. Maigcoder (Wei et al., 2023) utilizes open-source code snippets to create instruction datasets for further improving code LLMs' instruction-following abilities. Wizard-Coder (Luo et al., 2024) and Wav-Coder (Yu et al., 2023) use evol-instruct (Xu et al., 2023) to extract effective instruction-code pairs from proprietary LLMs through few-shot prompting and self-improvement. OctoCoder (Muennighoff et al., 2023) uses Git commits and code changes to generate instruction-following data and enhance the model's coding ability. Besides these works, there exist several works (Paul et al., 2024; Sun et al., 2024) focusing on using intermediate representation like from LLVM to improve Code LLMs. Cassano et al. (2024) proposed to translate high resource PLs to low resource PLs with the help of compiler.

2.2 LLMs' Inherent Capabilities

Large Language Models (LLMs) possess several intrinsic capabilities that are a result of the extensive training. One of their core strengths is general knowledge reasoning (Liang et al., 2022), which arises from the vast amount of diverse data they are trained on (Touvron et al., 2023a,b; Guo et al., 2024). This general reasoning ability enables LLMs to provide accurate responses to a wide range of tasks across different domains. Another most powerful capability of LLMs is In-Context Learning (ICL) (Brown et al., 2020). ICL enables models to generate more accurate responses by learning from the context provided in the input, without the need for further training. As a training-free approach, ICL is highly flexible and can be applied in various ways, including data generation (Wang et al., 2022), personalized conversations (Pi et al., 2024), where the model adapts to user preferences; and task-specific guidance, where context helps refine and improve response accuracy. ICL's versatility makes it a valuable tool for enhancing performance across different applications.

Leveraging these capabilities, our approach uses general knowledge reasoning for task screening to ensure solvable tasks are selected and applies ICL to utilize the Code-Bridge for more accurate LRPL outputs.

2.3 Low-Resource Programming Languages

LRPL Benchmarks. An ideal multilingual code generation benchmark requires diverse text queries, verified test cases, and execution environments. MultiPL-E(Cassano et al., 2022) fulfills these criteria by extending HumanEval and MBPP to multiple programming languages through human expert translation and modification, while also providing verified test cases and execution sandboxes. In contrast, other benchmarks like FIM(Face, 2023), CrossCodeEval (Ding et al., 2024), and CodeXGLUE (Lu et al., 2021) either lack a focus on text-to-code generation or do not specifically address LRPLs, which is the primary focus of our work.

LRPL Transcompilers. Although transcompilers (source-to-source compilers) can theoretically translate code between programming languages (PL-to-PL), they fail to address the NL-PL Gap (NL+PL-to-NL+PL), which is central to our research. Transcompilers also require significant engineering effort and become impractical for many language pairs, such as Python, Java, D, Racket, R, and Bash, which result in 36 combinations (Emre et al., 2021). Existing works like IRCoder (Paul et al., 2024) focus on PL-only semantics using intermediate representations. In contrast, our approach targets NL-PL pairs, offering a holistic solution that integrates natural language understanding with programming language consistency.

3 NL-PL Gap

The NL-PL Gap refers to the disparity that arises when LLMs are tasked with following natural language (NL) instructions in programming language (PL). This gap is particularly pronounced in low-resource programming languages (LRPLs). The NL-PL gap stems from the following key factors:

Data Imbalance. The statistics in Table 9 highlights the stark data imbalance between high-resource and low-resource programming languages. Languages like JavaScript, Python, and Java have millions of files in the StarCoder dataset, providing LLMs with extensive NL-PL aligned data. In contrast, low-resource languages such as R, Racket,

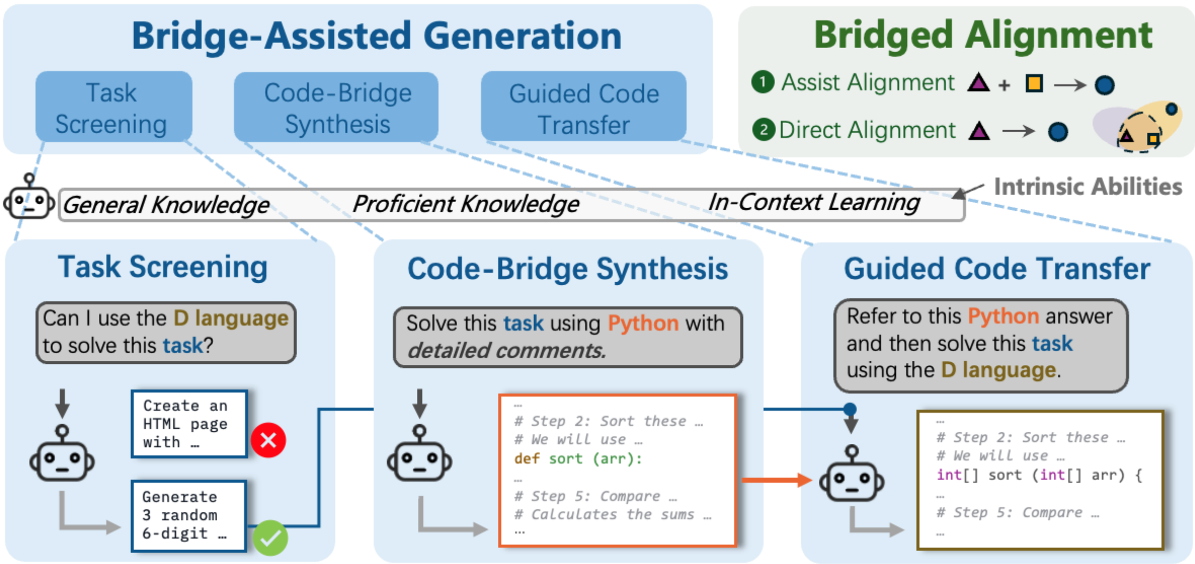

Figure 2: An illustration of **Bridge-Coder**. In *Bridge-Assisted Generation*, the LLM first identifies tasks suitable for the target low-resource programming language (LRPL). Then, it generates a code-bridge in a high-resource programming language (HRPL), combining both code and comments to explain the solution. This code-bridge is then used to help bridge the NL-PL gap in LRPLs. In *Bridged Alignment*, the model is first guided by the code-bridge to assist in aligning the NL-PL gap, and later progresses to generating responses directly from natural language instructions without the code-bridge.

and D are vastly underrepresented, with only a fraction of the data available. This disparity limits the model's ability to learn effective mappings from NL to PL in LRPLs, significantly contributing to the NL-PL Gap. The GitHub and TIOBE indices further reflect this imbalance, reinforcing the challenges faced by LLMs when generating code for underrepresented languages.

Complexity of Mapping NL to PL. Unlike purely natural language tasks, coding tasks require models to first understand NL instructions and then generate executable PL code, often together with NL comments to explain the code. In HRPLs, models excel due to the abundance of NL-PL aligned data. However, for LRPLs, this mapping is more difficult due to limited data. As shown in our experiments, directly generating code for LRPLs leads to lower-quality outputs, whereas using a code-bridge as a transitional step improves code quality by mitigating the NL-PL gap.

4 Methodology

In this section, we present the details of our approach to obtain **Bridge-Coder**, including two phases: Bridge-Assisted Generation (Section 4.1) and Bridged Alignment (Section 4.2). The key idea of Bridge-Assisted Generation is fully leveraging LLMs' intrinsic capabilities to generate instruction following data for low-resouce programming languages (LRPLs). Afterward, Bridged Alignment gradually helps the model overcome the NL-PL Gap, improving its performance on LRPLs. An illustration of Bridge-Coder is shown in Figure 2.

4.1 Bridge-Assisted Generation

This section introduces a novel approach for generating training data for LRPLs. We first leverage the LLM's general knowledge reasoning abilities to identify the task set $\mathcal{T}$ that can be effectively solved using the target LRPL, denoted as PL_{tar}. Next, we utilize the LLM's strong understanding and generation capabilities in a HRPL to generate the code-bridge, denoted as PL_{bdg}. Finally, by using the LLM's in-context learning (ICL) abilities, we rephrase the task $\mathcal{T}$ with the help of PL_{bdg}, which enables the LLM to generate the desired response in the target LRPL PL_{tar}.

4.1.1 Task Screening

Existing code instruction datasets often include general-purpose tasks, while others can only be solved with specific programming languages. If unsuitable tasks are not filtered out, the model may fail to respond when the task is unanswerable in the target language. Even more concerning, several studies (Spiess et al., 2024; Shum et al., 2024) have highlighted that, due to mis-calibration, LLMs tend to confidently generate incorrect answers in such cases. This issue further emphasizes the importance of task screening to prevent such errors and improve response quality.

We observe that while current LLMs struggle with LRPL code generation, they perform much better in classification tasks that simply judge whether a task can be solved using a specific LRPL. This is because classification tasks, unlike code generation, do not require the LLM to bridge the NL-PL Gap. Instead, the model can rely on its general reasoning abilities to provide a straightforward 'Yes' or 'No'. answer. Additionally, as the model's accuracy improves, we enhance this process by requiring the LLM to provide logical explanations for its judgments, further validating its decision-making process. We validate the importance of this screening step with experimental evidence in Section 6.3.4, showing its critical role in improving output quality.

4.1.2 Code-Bridge Synthesis

When LLMs answer task $\mathcal{T}$, they first need to comprehend the natural language (NL) instructions and then generate a response in the programming language (PL), which often includes adding NL comments to the code. This makes NL-PL alignment in the training data crucial. In high-resource programming languages (HRPLs), the abundance of NL-PL aligned data in the training sets allows LLMs to perform effectively.

Here, we leverage the existing capabilities of LLMs in HRPLs to follow the NL instruction. Furthermore, we also ask LLMs to include comments explaining the key steps and the thought process behind the solution. In this way, we create the code-bridge PL_{bdg}, which combines both NL (i.e., comments) and PL (i.e., code). This serves as a reinterpretation of the NL instruction from the perspective of PL logic, transforming what might seem like simple instructions in NL into a more explicit and detailed process in PL. Programming languages often require a step-by-step breakdown and precise logic that natural language tends to abstract away, making PL_{bdg} an essential way for bridging this gap. This approach ensures that even with limited NL-PL aligned data in LRPLs, LLMs can still effectively generate correct and coherent solutions by leveraging the detailed structure and reasoning provided by the code-bridge.

4.1.3 Guided Code Transfer

For LRPLs, which are underrepresented in training data, the NL-PL Gap presents a major challenge. Although LLMs possess some code generation capabilities, the lack of well-aligned data between NL instructions and PL reasoning leads to suboptimal solutions, making it difficult to generate accurate responses to NL instructions in LRPLs.

To overcome this, we utilize the previously generated code-bridge to mitigate the NL-PL gap. During this step, when generating responses in LRPLs, the code-bridge is appended to the instruction as additional context. By harnessing the in-context learning (ICL) capabilities of LLMs, this approach allows the model to reference the PL logic in the code-bridge, guiding it when responding to NL instructions. This significantly improves the quality of the model's output in LRPLs.

This process is analogous to a non-native English speaker first drafting their thoughts in their native language and then translating them into English. The code-bridge acts as a "draft" in PL, enabling the LLM to better interpret NL instructions and produce more accurate answers in LRPLs.

4.2 Bridged Training

We draw inspiration from the concept of curriculum learning (Bengio et al., 2009) and apply it to the learning of LRPLs. To effectively bridge the NL-PL gap and improve LLM performance in low-resource programming languages, we divide the training process into two stages.

Assist Alignment. In the first stage, the primary goal is to assist the LLM in bridging the NL-PL gap by providing additional support through the code-bridge. The input includes the instruction of task $\mathcal{T}$, along with the code-bridge in the high-resource programming language PL_{bdg}, which serves as a guide. The LLM uses this reference to assist in generating the target response in the low-resource programming language PL_{tar}. The loss function can be formalized as:

$$\mathcal{L}_{\text{assist}} = -\sum_{t=1}^{T} \log P(y_t^{PL_{\text{tar}}} \mid y_{<t}^{PL_{\text{tar}}}, \mathcal{T}, PL_{\text{bdg}})$$

Direct Alignment. In the second stage, the focus shifts to helping the LLM adapt to real-world scenarios by asking it to directly follow NL instructions without any assistance from the code-bridge. This approach ensures the model becomes more

Models	**M-HumanEval** pass@1					**M-MBPP** pass@1				
	R	D	Bash	Racket	Avg	R	D	Bash	Racket	Avg
CodeLlama	18.42	11.76	10.09	12.34	13.15	24.75	20.75	19.77	21.50	21.69
CodeGemma	20.92	10.47	8.10	9.04	12.13	24.73	15.94	11.57	20.57	18.20
DeepSeek-Coder-Base	29.28	20.73	24.51	19.84	23.59	38.23	30.83	28.67	32.48	32.55
Magicoder-DS	38.31+9.03	19.47-1.26	29.17+4.66	29.17+9.33	29.03+5.44	41.13+2.90	32.49+1.66	28.52-0.15	37.75+5.27	34.97+2.42
Magicoder-S-DS	40.63+11.35	24.60+3.87	33.06+8.55	30.50+10.66	32.20+8.61	44.03+5.80	31.76+0.93	24.43-4.24	37.83+5.35	34.51+1.96
DeepSeek-Coder-DG	37.56+8.28	27.76+7.03	37.26+12.75	30.95+11.11	33.38+9.79	46.74+8.51	36.47+5.64	34.26+5.59	33.96+1.48	37.86+5.31
DeepSeek-Coder-IC	42.93+13.65	31.39+10.66	37.26+12.75	33.81+13.97	36.35+12.76	47.12+8.89	38.52+7.69	34.16+9.65	37.30+4.82	39.28+6.73
DeepSeek-Coder-BC	**49.11**+19.83	**35.51**+14.78	**42.99**+18.48	**41.57**+21.73	**42.30**+18.71	**50.53**+12.30	**43.51**+12.68	**35.83**+7.16	**43.57**+11.09	**43.36**+10.81

Table 1: Comparison of different models on two mainstream benchmarks for multilingual code generation, each featuring multiple test inputs per case. To ensure a comprehensive and challenging evaluation, we adopt pass@1, allowing models only a single attempt to produce the correct solution. Here, -DG denotes DeepSeek-Coder-Base fine-tuned on data from direct generation, -IC on data generated via in-context learning, and -BC using our proposed framework. We also compare Magicoder-DS (Wei et al., 2023) which also finetuned based on the our base model. **Bold** values indicate the best performance. + and − represent the difference compared to DeepSeek-Coder-Base.

capable of solving tasks independently, as it would in practical applications where such assistance is not available. The training loss for this phase is calculated as:

$$\mathcal{L}_{\text{direct}} = -\sum_{t=1}^{T} \log P(y_t^{PL_{\text{tar}}} \mid y_{<t}^{PL_{\text{tar}}}, \mathcal{T})$$

This two-step process facilitates a smooth and effective learning progression, moving from guided learning with assistance to independent problem-solving in LRPLs, as validated in the subsequent experiments in Section 6.2, highlighting the benefits of this approach.

5 Evaluation Settings

5.1 LRPLs and Benchmarks

LRPLs. To fully validate the generalization ability of our method, we selected four low-resource programming languages: R, D, Racket, and Bash. These languages cover a broad range of functionalities, including statistical computing, systems programming, language creation, and automation. Detailed descriptions of these languages can be found in the Appendix A.2.

Benchmarks. We utilize the adapted versions of two widely recognized benchmarks that also contain low-resource programming languages, M-HumanEval(Chen et al., 2021b) and M-MBPP(Austin et al., 2021b).[1] Both benchmarks are highly challenging due to their rigorous requirements: each programming task consists of hundreds of problems, where a solution must pass numerous test cases to be considered correct. For example, M-HumanEval evaluates over 150 tasks per language, with each requiring an average of 9.6 test cases to validate correctness, ensuring a stringent evaluation process. Similarly, M-MBPP evaluates nearly 400 tasks per language, further testing the robustness of models under diverse scenarios. To better reflect real-world demands, we adopt the pass@1 metric, which requires the model to generate a correct solution on the first attempt.

5.2 Models

Baseline Models. We select five baseline models for comparison: CodeLlama (Roziere et al., 2023); CodeGemma (Team et al., 2024); DeepSeek-Coder-Base (Guo et al., 2024), which serves as the base model for our subsequent fine-tuning experiments; Magicoder-DS (Wei et al., 2023), a widely used benchmark model for code generation, trained on the OSS-Instruct dataset and built upon DeepSeek-Coder-Base; and MagicoderS-DS (Wei et al., 2023), an enhanced version of Magicoder-DS, further trained on both the OSS-Instruct and Evol-Instruct datasets (Luo et al., 2024), offering superior performance across various coding benchmarks while also being based on DeepSeek-Coder-Base.

Models for Generation. The models used during the *Bridge-Assisted Generation* process include: Llama3-70B (Dubey et al., 2024), our primary model, which combines strong general-purpose reasoning with high performance in code generation tasks, making it suitable for a wide range of applications; Llama3-8B, a smaller variant of Llama3 that leans more towards general-purpose tasks with moderate code generation capabilities; and StarCoder2-Instruct-15B (Li et al., 2023), a specialized code LLM with strong capabilities for code-related tasks but limited general-purpose

[1]We use the MultiPL-E (Cassano et al., 2022) framework for evaluation to evaluate models on this two benchmarks.

reasoning abilities.

5.3 Comparison

Data Generation. In our experiments, we employ two different approaches for generating data: `DG` (Direct Generation): In this approach, the model generates code directly from the natural language task without any intermediate steps. `IC` (In-Context Examples): This approach utilizes some human generated examples to help the model better understand the task during generation. `BC` (Ours): This approach introduces a code-bridge, which acts as an intermediary between the task and the final code generation.

Training Methods. We compare several training techniques, where our training data is represented as {input; output}. $\mathcal{T}$ denotes the NL (natural language) task instruction, PL_{bdg} represents the HRPLs output that acts as a bridge, and PL_{tar} is the answer in the target low-resource programming language (LRPL). The *Separate Alignment* method is represented as $\{\mathcal{T}; PL_{\text{bdg}}\} \cup \{\mathcal{T}; PL_{\text{tar}}\}$. *Direct Alignment* involves $\{\mathcal{T}; PL_{\text{tar}}\}$. *Assist Alignment* combines $\{\mathcal{T}, PL_{\text{bdg}}; PL_{\text{tar}}\}$, while *Bridged Alignment* begins with assist alignment and transitions to direct alignment.

6 Experimental Results

6.1 Main Results

As shown in Table 1, the performance of various models underscores the challenges of adapting to Low-Resource Programming Languages (LRPLs). Models like `CodeLlama`(Roziere et al., 2023) and `CodeGemma`(Team et al., 2024) exhibit strong capabilities in HRPLs but struggle significantly on LRPLs. For instance, `CodeGemma` achieves lower average scores on both M-HumanEval and M-MBPP compared to `DeepSeek-Coder-Base`, highlighting its limited ability to generalize beyond HRPLs. Similarly, `CodeLlama`, while competitive in HRPL scenarios, shows minimal performance in languages like `Bash` and `Racket`, further emphasizing the difficulty of LRPL adaptation without targeted strategies.

Models such as `DeepSeek-Coder-DG` and `DeepSeek-Coder-IC`, which incorporate directly generated data or in-context learning, demonstrate modest improvements over `DeepSeek-Coder-Base`. However, their gains are inconsistent and limited, particularly in scenarios requiring deeper NL-PL alignment. For example, while `DeepSeek-Coder-IC` performs well in `R` and `Racket`, it still lags in `Bash`, showcasing the limitations of direct data generation strategies for LRPLs.

In contrast, our proposed Bridge-Coder framework (`DeepSeek-Coder-BC`) achieves substantial improvements across all LRPLs and benchmarks. By introducing the code-bridge as an intermediate step, our method effectively enhances NL-PL alignment, producing higher-quality LRPL training data that leverages the model's HRPL strengths. For example, `DeepSeek-Coder-BC` achieves significant improvements of +19.83 in `R` and +21.73 in `Racket` on M-HumanEval, with consistent gains across all benchmarks. It also outperforms all baselines, with average improvements of +18.71 on M-HumanEval and +10.81 on M-MBPP.

Training Methods	R	D	Avg
Separate Alignment	46.89	23.87	35.38
Direct Alignment	42.63	32.45	37.54
Assist Alignment	42.93	33.87	38.40
Bridged Alignment	**49.11**	**35.51**	**42.31**

Table 2: Comparison of different aligning methods.

6.2 Comparison of Training Methods

We compared various training methods to assess their effectiveness in aligning NL instructions with LRPL outputs. As shown in Table 2, *Assist Alignment* alone performs worse because the model becomes overly reliant on the code-bridge and struggles to generalize to NL-only instructions. *Direct Alignment* also underperforms, as the model is forced to bridge the NL-PL gap without any support, highlighting the importance of gradual learning. Our *Bridged Training* approach, which begins with *Assist Alignment* and transitions to *Direct Alignment*, consistently achieves the best results. To ensure the improvements weren't solely due to the HRPL component of the code-bridge, we tested *Separate Alignment*, which showed instability in D, confirming that combining the two phases of *Bridged Training* leads to more robust and effective performance..

6.3 Further Analysis

6.3.1 Code LLM Performs Better?

One might expect that code-specific models would perform best for generating code-related data, but

Synthesis	Transfer	R	D	Avg
Code	Code	29.56	26.61	28.08
Code	General	32.22	<u>27.50</u>	29.86
General	General	**37.53**	**28.16**	**32.85**
General	Code	<u>34.23</u>	25.90	<u>30.07</u>

Table 3: Comparison of code-specific models (Code) and general-purpose models (General) in different combinations of Code-Bridge Synthesis and Guided Code Transfer. **Bold** indicates the best result, and <u>underline</u> indicates the second-best result.

Assist Format	R	D	Avg
PL	42.64	32.57	37.61
NL	40.89	30.72	35.81
NL + PL	**44.71**	**35.51**	**40.11**

Table 4: Comparison of different assist formats in the Assist Alignment during the second phase.

Code-Bridge HRPL	R	D
Python	**47.61**	**33.87**
C++	44.29	30.62
JAVA	46.47	31.93

Table 5: Further Analysis of different Programming Languages used as Code-Bridge in the first stage.

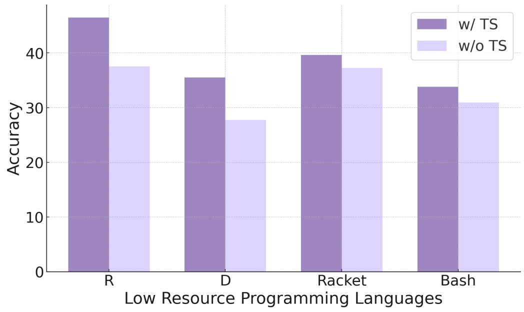

Figure 3: Ablation of Task Screening.

as shown in Table 3, the combination of code models for both Synthesis and Transfer stages actually performs the worst. In contrast, general-purpose models consistently improve performance, with the best results coming from using general models for both stages. This can be attributed to the fact that Code-Bridge Synthesis primarily leverages the model's HRPL capability, which reduces the performance gap between code-specific and general models. However, in the Guided Code Transfer stage, in-context learning (ICL) becomes more critical, where general models seem to outperform code-specific ones.

6.3.2 NL vs. PL: Which Matters More?

In the first phase of our Bridged Training, we explored whether using NL-formatted comments or PL-formatted code as part of the Assist Alignment yields better alignment. As shown in Table 4, training with code (PL) alone outperforms comments (NL) alone. However, relying solely on code is still not the optimal approach. The combination of both NL and PL (code & comments) leads to the best results, highlighting the complementary nature of NL and PL in bridging the NL-PL gap and improving overall performance. This also explains why, in our generation of the code-bridge, we emphasize the need for explanations in the form of NL comments to assist and enhance the code output.

6.3.3 Different HRPLs as Code-Bridge

The results in Table 5 demonstrate that Python outperforms C++ and Java as the code-bridge programming language. This is likely due to Python's prevalence in the training data, which enables the model to generate more accurate and effective code-bridges. Python's extensive library ecosystem for tasks like data science and automation also provides more tools for generating robust code. Additionally, Python's simplicity and readability contribute to better alignment with natural language instructions, facilitating a smoother NL-PL transition. In contrast, C++ and Java's more complex syntax and explicit logic make them less effective for generating efficient code-bridges in this context.

6.3.4 Ablation of Task Screening

Figure 3 highlights the importance of task screening. While the dataset without screening includes more tasks, the performance on unanswerable tasks is poor. With Task Screening (w/ TS), accuracy improves significantly across all LRPLs (R, D, Racket, Bash). This demonstrates that filtering out tasks beyond the model's capability leads to better results and validates the effectiveness of using LLMs' general reasoning for task screening.

7 Conclusion

This paper tackles the challenge of generating high-quality programs in low-resource languages. By leveraging LLMs' intrinsic abilities and expertise in high-resource programming languages, we create a new, high-quality dataset for low-resource languages. We also propose a progressive alignment to mitigate the gap. Experimental results show our methods significantly outperform the baseline.

Limitations

Despite strong instruction-following capabilities, our work remains confined to repository-level text-to-code generation, which involves long-context modeling and resolving lost-in-the-middle issues. Additionaly, future studies should address multi-round text-code challenges, requiring repeated interactions and more detailed instructions.

References

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

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021b. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48.

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. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):677–708.

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison 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, Joshua 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. 2021a. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021b. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Unsupervised cross-lingual representation learning at scale. In *Annual Meeting of the Association for Computational Linguistics*.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. 2024. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2021. Translating c to safer rust. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–29.

Hugging Face. 2023. Open llm leaderboard-a hugging face space by huggingfaceh4.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *CoRR*, abs/2204.05999.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Junjie Hu, Sebastian Ruder, Aditya Siddhant, Graham Neubig, Orhan Firat, and Melvin Johnson. 2020. Xtreme: A massively multilingual multi-task benchmark for evaluating cross-lingual generalization. *ArXiv*, abs/2003.11080.

Haoyang Huang, Yaobo Liang, Nan Duan, Ming Gong, Linjun Shou, Daxin Jiang, and M. Zhou. 2019. Unicoder: A universal language encoder by pre-training with multiple cross-lingual tasks. In *Conference on Empirical Methods in Natural Language Processing*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

G Lample. 2019. Cross-lingual language model pretraining. *arXiv preprint arXiv:1901.07291*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, et al. 2022. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. Wizardcoder: Empowering code large language models with evol-instruct. *International Conference on Learning Representations (ICLR)*.

Microsoft. 2023. Github copilot – your ai pair programmer. GitHub repository.

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

Daye Nam, Andrew Peter Macvean, Vincent J. Hellendoorn, Bogdan Vasilescu, and Brad A. Myers. 2023. Using an llm to help with code understanding. *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 1184–1196.

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. In *International Conference on Machine Learning*, pages 26619–26645. PMLR.

Indraneil Paul, Jun Luo, Goran Glavaš, and Iryna Gurevych. 2024. Ircoder: Intermediate representations make language models robust multilingual code generators. *arXiv preprint arXiv:2403.03894*.

Renjie Pi, Jianshu Zhang, Tianyang Han, Jipeng Zhang, Rui Pan, and Tong Zhang. 2024. Personalized visual instruction tuning. *arXiv preprint arXiv:2410.07113*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Services. 2023. A. w. ai code generator - amazon codewhisperer - aws. Amazon Page.

Noam Shazeer and Mitchell Stern. 2018. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, pages 4596–4604. PMLR.

KaShun Shum, Minrui Xu, Jianshu Zhang, Zixin Chen, Shizhe Diao, Hanze Dong, Jipeng Zhang, and Muhammad Omer Raza. 2024. First: Teach a reliable large language model through efficient trustworthy distillation. *arXiv preprint arXiv:2408.12168*.

Claudio Spiess, David Gros, Kunal Suresh Pai, Michael Pradel, Md Rafiqul Islam Rabin, Susmit Jha, Prem Devanbu, and Toufique Ahmed. 2024. Quality and trust in llm-generated code. *arXiv preprint arXiv:2402.02047*.

Tao Sun, Linzheng Chai, Jian Yang, Yuwei Yin, Hongcheng Guo, Jiaheng Liu, Bing Wang, Liqun Yang, and Zhoujun Li. 2024. Unicoder: Scaling code large language model via universal code. *arXiv preprint arXiv:2406.16441*.

CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A Choquette-Choo, Jingyue Shen, Joe Kelley, et al. 2024. Codegemma: Open code models based on gemma. *arXiv preprint arXiv:2406.11409*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023a. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023b. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language model with self generated instructions. *arXiv preprint arXiv:2212.10560*.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708. Association for Computational Linguistics.

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

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. 2023. Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244*.

L Xue. 2020. mt5: A massively multilingual pre-trained text-to-text transformer. *arXiv preprint arXiv:2010.11934*.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2023. Wavecoder: Widespread and versatile enhanced instruction tuning with refined data generation. *arXiv preprint arXiv:2312.14187*.

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. *CoRR*, abs/2303.17568.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Ldb: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*.

A Appendix

A.1 Detailed Prompts

This is a section in the appendix.

A.2 Low-Resource Programming Languages

- **R**: A programming language widely used for statistical computing, data analysis, and visualization. It is highly popular in academia, research, and data science due to its extensive libraries and tools for handling complex data.

- **D**: A systems programming language designed for high performance and productivity. It combines the power of C and C++ with more modern features, making it ideal for applications that require efficiency and low-level system access, while maintaining a developer-friendly syntax.

- **Racket**: A functional programming language that excels in language creation and experimentation. It is commonly used in academic settings and research for developing new programming languages, as well as for teaching concepts in computer science and functional programming.

- **Bash**: A Unix shell and command language widely used for scripting and automation tasks in system administration, software development, and task automation. Bash scripts are frequently used for managing servers, executing repetitive tasks, and automating workflows in Linux environments.

Prompt for Task Screening

You are a highly knowledgeable assistant that specializes in problem-solving across various programming languages.

You should judge whether <Programming Language> can be used to solve the problem below.

You should always respond with either “Yes" or “No", followed by a concise explanation. Be concise and direct in your responses.

Here is the task: <Task>

Table 6: The prompt for screening tasks that are unanswerable in Low-Resource Programming Language.

Prompt for Bridge-Assisted Generation

You are a highly knowledgeable assistant that specializes in problem-solving across various programming languages.

Help me use <Programming Language> to solve the problem below. In your response, you need to provide **detailed comments** to explain the key steps and the reasoning process, rather than only responding the solution.

Here is the task: <Task>

Table 7: The prompt for synthesizing code-bridge in High-Resource Programming Language.

Prompt for Guided Code Transfer

You are a highly knowledgeable assistant that specializes in problem-solving across various programming languages.

Help me use <Programming Language> to solve the problem below.

Here is the task: <Task>

To help you better solve this task, you can refer to this solution in <Programming Language>: <Code-Bridge>

Table 8: The prompt for generating answers in Low-Resource Programming Language. <Code-Bridge> is the answer in a High-Resource Programming Language.

B Implementation Details

For optimization, we used the Adafactor (Shazeer and Stern, 2018) optimizer with a learning rate of 5×10^{-5}. The model was trained for 2 epochs with a warm-up of 15 steps. The batch size was set to 512. To improve efficiency, we employed Flash Attention (Dao et al., 2022) and used the bf16 precision for faster and more memory-efficient training.

C Statistics

Language	StarCoder		GitHub (%)	TIOBE (%)
	Num. files	Percentage		
Bash	-	-	-	43
C++	6,377,914	6.379	7.0	4
C#	10,839,399	5.823	3.1	5
D	-	-	-	35
Go	4,730,461	3.101	7.9	12
Java	20,151,565	11.336	13.1	3
JavaScript	21,108,587	8.437	14.3	7
Python	12,962,249	7.875	-	1
R	39,194	0.039	0.05	19
Racket	4,201	0.004	-	-
Rust	1,386,585	1.188	1.1	22
Ruby	3,405,374	0.888	6.2	15

Table 9: Programming Language Statistics: The StarCoder parts are based on data from their report (Li et al., 2023). The last two columns are derived from GitHut 2.0 and the 2022 TIOBE Programming Community Index, as referenced in the MultiPLE benchmark paper (Cassano et al., 2022).