CodeRise: Bootstrapping LLMs for Ultra Low-Resource Programming Languages via Progressive Self-Refinement Curriculum

Anonymous ACL submission

Abstract

Large Language Models (LLMs) struggle with code generation for Ultra Low-Resource Programming Languages (ULRPLs) due to the scarcity of training data. Existing synthetic data generation methods fail in this context, suffering from a severe cold-start problem where base models cannot even produce syntactically valid code, and resulting in samples that lack diversity. To overcome these challenges, we propose CodeRise, a novel two-stage framework that autonomously generates a high-quality, diverse, and progressively complex curriculum for ULRPLs. The framework first tackles cold-start and distribution issues by leveraging the full formal syntax of the target language as structural guidance and applying a biased sampling strategy over library modules. Building on this foundation, the model is then fine-tuned to generate more complex code without explicit syntax input. An adaptive curriculum, informed by Pass@k scores, gradually increases task difficulty, while a multi-turn self-debugging mechanism improves code correctness and overall generation quality. We evaluated our framework on Tengo, a representative ULRPL, using adapted versions of HumanEval and MBPP, and our newly proposed benchmark, Tengo-Eval. Experiments show that CodeRise significantly outperforms both training-free and training-based baselines, setting a new standard for LLM-driven code generation in Ultra Low-Resource environments.

1 Introduction

Large Language Models (LLMs) (OpenAI et al., 2024b; Hui et al., 2024; Guo et al., 2024) have demonstrated outstanding performance across a wide range of code-related tasks, including code generation (Chen et al., 2021; Lai et al., 2023), code repair (Chen et al., 2024; Zhong et al., 2024; Tian et al., 2024), code completion (Bavarian et al., 2022; Ding et al., 2024) and program analysis (First et al., 2023). This remarkable success has been predominantly observed in the context of high-resource languages such as Python, Java, and JavaScript, where vast and high-quality codebases are readily available for training. The success is even beginning to extend to many Low-Resource Programming Languages (LRPLs) like R or Perl (Cassano et al., 2022; Giagnorio et al., 2025; Joel et al., 2024; Cassano et al., 2024). However, the powerful capabilities of LLMs do not universally extend to the Ultra Low-Resource Programming Languages (ULRPLs) (Mora et al., 2024), despite their importance in various specialized domains. These languages suffer from a multi-dimensional data scarcity: not only is their volume of publicly available code inherently low, but systematic collection is also challenging. For instance, many ULRPLs lack unique file extensions, making them invisible to automatic identification tools like GitHub's Linguist[1]. This severe lack of data directly limits the ability of models to learn the syntax, libraries, and coding patterns specific to ULRPLs.

A promising direction to overcome this data scarcity is to leverage the LLM itself to generate synthetic training data, a principle exemplified by methods like *Self-Instruct* (Wang et al., 2023) and *SelfCodeAlign* (Wei et al., 2024a). The Self-Instruct paradigm typically operates by iteratively bootstrapping instructions from a seed corpus, prompting a powerful teacher model to generate corresponding responses, and then validating these generations. However, existing relevant techniques, which are primarily designed for high-resource languages, exhibit limitations when directly applied to ULRPLs. A primary obstacle for ULRPLs is the cold-start problem. Even with state-of-the-art LLMs, their initial proficiency in these languages is often so low that they fail to produce even a single syntactically valid code sample.

[1] https://github.com/github-linguist/linguist

This fundamental inability prevents common self-generation loops, a cornerstone of data augmentation in high-resource settings from successfully bootstrapping.

A straightforward workaround to launch this process is to provide the model with the complete language syntax in-context. However, this approach is fraught with issues. The extremely long context leads to slow inference speeds and a low pass rate, as shown in Figure 1, making it impractical for large-scale data generation. Furthermore, even with this syntax guidance, the generated data still exhibits a severe long-tail distribution, clustering around basic language features as shown in Table 1. We hypothesize that this phenomenon arises because LLMs tend to default to generating instructions for "universal programming concepts", which are highly transferable concepts such as string manipulation.

To overcome these challenges, we introduce **CodeRise**, a two-stage framework that systematically guides an LLM from near-zero knowledge to proficient code generation in ULRPLs. In the *Syntax-Guided Module-Balanced Generation* stage, we tackle the cold-start problem and the long-tail distribution of generated samples. By explicitly providing language syntax as in-context guidance and adopting a biased sampling strategy over language-specific libraries, this stage enables the LLM to bootstrap the generation process and produce an initial dataset that is both syntactically correct and semantically diverse. Building on this foundation, the *Progressive Self-Refined Generation* stage focuses on scaling complexity and improving efficiency. It removes the need for lengthy syntax prompts and introduces an adaptive curriculum to gradually increase task difficulty, along with a multi-turn self-debugging mechanism to iteratively correct model errors. Together, these two stages allow CodeRise to progressively enhance the model's generation ability and produce high-quality code in languages with extremely limited data resources.

We conduct extensive experiments on the ULRPL **Tengo**, evaluating on our migrated versions of the HumanEval and MBPP benchmarks, alongside our newly constructed **TengoEval** benchmark. Using our two-stage framework, CodeRise, we successfully generate a high-quality dataset of 30,000 instruction-code pairs. The results demonstrate that models fine-tuned on this data consistently and significantly outperform a comprehensive suite of baselines, including both training-free and training-based approaches, underscoring the efficacy of our progressive data generation strategy.

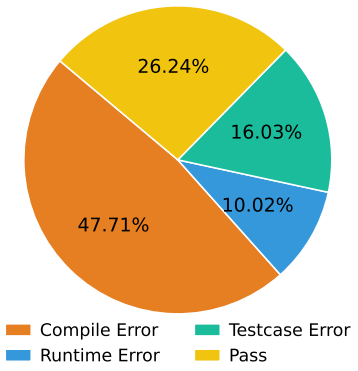

Figure 1: Failure analysis for the Qwen-2.5-Coder-32B-Instruct using in-context learning with the language's full documentation, the majority of failures are execution errors.

Our contributions are threefolds:

- We propose CodeRise, a self-improvement framework specifically designed to overcome the critical cold-start and long-tail challenges in ULRPLs.
- We release the first open benchmark suite for the Tengo, combining migrated HumanEval and MBPP tasks with a newly developed dataset, TengoEval.
- We design a novel progressive self-refinement process that circumvents the need for a powerful teacher model by instead equipping the model to learn from reliable external feedback.

2 Related Work

LLMs for Coding In recent years, Large Language Models (Guo et al., 2024; Rozière et al., 2024) have achieved remarkable progress in code-related tasks, such as code generation (Zheng et al., 2024; Han et al., 2024), code completion (Yu et al., 2024a; Hayes et al., 2025) and code translation (Szafraniec et al., 2023; Xue et al., 2024; Pan et al., 2024), and the impact is expanding across the entire software engineering (Jimenez et al., 2023; Xie et al., 2025; Yang et al., 2024). Besides, reasoning models like OpenAI-o1 (OpenAI et al., 2024a) and DeepSeek-R1 (DeepSeek-AI et al., 2025) have also demonstrated impressive coding performance

Module	Frequency	Proportion(%)
text	6202	47.8
math	1438	11.1
rand	1291	10.0
fmt	1354	10.4
json	935	7.2
enum	791	6.1
times	650	5.0
base64	284	2.2
os	17	0.1
hex	9	0.1

Table 1: Distribution of standard library usage from an 30k dataset generated with the SelfCodeAlign baseline. The statistics are based on the 12971 samples that involved API calls. The table clearly illustrates the long-tail problem, with usage heavily concentrated on a few modules.

due to their long reasoning ability. However, despite these advancements, current LLMs still struggle to perform well on ULRPLs due to the scarcity of training data, limited exposure to syntax patterns, and the lack of task-specific supervision. While some prior work has addressed challenges in low-resource programming languages (Giagnorio et al., 2025; Cassano et al., 2024, 2022), such as R and Perl, their methods rely on the model's existing initial capability in those languages. This foundation is absent for the ULRPLs setting.

Data-Driven Code Tuning Instruction tuning refers to the process of further training a pre-trained large language model on a dataset consisting of instruction-response pairs (Ouyang et al., 2022; Wei et al.; Li et al., 2024) . This paradigm has proven highly effective for adapting LLMs to specialized domains, particularly for code-related tasks (Muennighoff et al., 2024; Yu et al., 2024b; Liu et al., 2024). However, a primary challenge for effective instruction tuning is the need for high-quality data, which is often expensive and labor-intensive to collect. To address this, *Self-Instruct* (Wang et al., 2023; Xu et al., 2024) paradigm was proposed, leveraging a powerful LLM to generate its own instruction-following data from a small set of seeds. This paradigm has been successfully adapted to the code domain, leading to the development of powerful models like WizardCoder (Luo et al., 2024), Magicoder (Wei et al., 2024b), and frameworks such as SelfCodeAlign (Wei et al., 2024a). However, the success of these methods is predicated on a critical assumption, the model must possess a strong initial capability in the target language. This assumption holds for languages like Python, but breaks down for ULRPLs.

3 Methodology

To address the challenges of data scarcity and the pervasive long-tail distribution problem inherent in ultra-low-resource programming languages (ULRPLs), we propose an iterative self-improvement framework. This framework adheres to a curriculum learning paradigm, enabling an LLM to autonomously generate diverse, correct, and progressively complex fine-tuning data through two distinct stages. Stage 1, termed Syntax-Guided Data Generation, primarily aims to create an initial high-quality dataset for the preliminary fine-tuning of the LLM's foundational understanding. Stage 2, Advanced Data Generation via Self-Debugging, is meticulously engineered to achieve two key objectives: generating more complex and challenging tasks efficiently, and substantially improving data quality through robust self-correction mechanisms.

3.1 Syntax-Guided Module-Balanced Generation

The data generation process in Stage 1 is executed as a multi-step pipeline designed to create a foundational and diverse dataset. The pipeline begins by generating a composite "seed concept set" derived from both code snippet analysis and a diversity-focused module sampling strategy. This seed set is then used to formulate a specific programming instruction. Following this, the model leverages the complete language syntax to autonomously generate both a code implementation and its corresponding test suite. Finally, the entire script is rigorously validated in a sandbox environment to ensure its correctness.

Concept Inspiration from Code Snippets Similar to previous works like OSS-Instruct (Wei et al., 2024b) and SelfCodeAlign (Wei et al., 2024a), we select a code snippet from a meticulously curated open-source code corpus, typically ranging from 1 to 15 lines. This chosen snippet serves as an inspiration for the LLM. Prompted with this snippet, the LLM is tasked with extracting and formulating a set of related "programming concept" that represent the core functionality, algorithms, or common patterns found in the snippet. This approach lever-

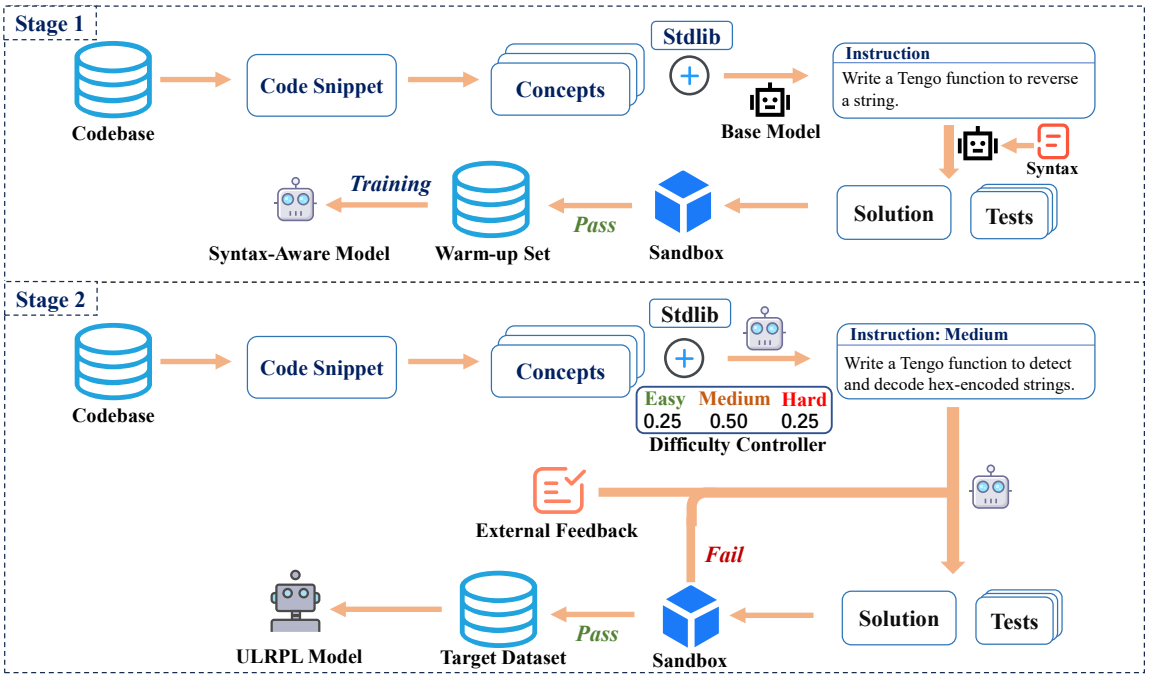

Figure 2: The overview of our two-stage CodeRise framework. **Stage 1: Syntax-Guided Module-Balanced Generation** (Top) addresses the cold-start problem by using explicit syntax guidance and diversity-focused module sampling to create a high-quality Warm-up Dataset. **Stage 2: Progressive Self-Refined Generation** (Bottom) then builds upon the fine-tuned model, employing a curriculum-based difficulty controller and a multi-turn self-debugging loop guided by external feedback to efficiently generate complex, high-quality instruction data for the final ULRPL Model.

ages the LLM's understanding of diverse coding patterns to derive high-level, actionable concepts, which form the building blocks for subsequent task instruction generation.

Module Sampling Relying solely on concepts derived from code snippets leads to a long-tail distribution in the generated dataset, with tasks clustering around common language features like string manipulation. To counteract this bias, we employ a weighted sampling strategy to select a module from the language's standard library, then this module is injected as an additional concept to guide task generation. The sampling weight for each module is determined by our logarithmic inverse frequency formula:

$$w_i = \frac{1}{log(c_i + 2)^p} \quad (1)$$

where c_i is the frequency that module i appears in correct solution. p is a hyperparameter to control the intensity of this weighting.

Code Generation and Validation The seed concept set from the previous step serves as the input to a sequential pipeline that generates and validates each data point. First, this set is embedded into a prompt that instructs the LLM to formulate a specific programming task in natural language. Next, this newly created instruction, along with the complete language syntax as a guiding scaffold, is provided to the LLM in a second prompt. In this step, the model's task is to generate both the code solution and a corresponding set of unit tests. Finally, each resulting (instruction, code, test) triplet is subjected to rigorous automated validation in a secure sandbox. An instruction-code pair is only accepted into our cold-start dataset if the code executes flawlessly and passes all associated tests, all other attempts are discarded.

3.2 Progressive Self-Refined Generation

Building upon the model fine-tuned in Stage 1, this advanced stage elevates the data generation process along two key dimensions: generation efficiency and data quality. A significant leap in efficiency is achieved by no longer providing the complete language syntax in the prompt. This is viable because the model has already internalized foundational syntactic knowledge, which drastically reduces context length and improves throughput. To enhance quality and generation yield, this

stage replaces Stage 1's "discard-on-failure" strategy with the multi-turn self-debugging mechanism. This strategy enables the model to methodically correct its errors by leveraging reliable external feedback, facilitating the successful generation of more complex instruction-code pairs and elevating the dataset's overall quality.

Curriculum-Based Difficulty Control To systematically increase the complexity of generated tasks, Stage 2 introduces a structured curriculum learning approach. While the initial seed generation process remains consistent with Stage 1 combining concepts from code snippets with diversity-focused module sampling, we now introduce a crucial new dimension: explicit difficulty levels.

Crucially, we categorize code tasks into three difficulty tiers: `Easy, Medium, and Hard`, each with a tailored prompt template. The curriculum's progression is adaptive, driven by the model's mastery as quantified by its Pass@10 score p $(0 \leq p \leq 1)$ on recent tasks. This score dynamically adjusts the sampling probability for each tier. The unnormalized weights are calculated as follows:

$$\begin{aligned} \text{Weight}_{\text{hard}} &= \alpha \cdot p \\ \text{Weight}_{\text{medium}} &= C \\ \text{Weight}_{\text{easy}} &= \alpha \cdot (1 - p) \end{aligned} \tag{2}$$

where α is a hyperparameter balancing the influence of Pass@10 on hard/easy tasks, and p represents the current Pass@10. The Medium tier's weight is a fixed constant C that acts as a stable baseline, ensuring the model always has a default difficulty to fall back on.

These unnormalized weights are then converted into probabilities using a softmax function with temperature T.

$$P_i = \frac{\exp(\text{Weight}_i/\text{T})}{\sum_j \exp(\text{Weight}_j/\text{T})} \tag{3}$$

where P_i is the probability of selecting difficulty tier i, temperature T controls the randomness of the tier selection. This adaptive strategy ensures a progressive increase in task complexity, maintaining appropriate control over the curriculum.

Multi-Turn Self-Debugging A cornerstone of Stage 2 is the Multi-Turn Self-Debugging Mechanism, designed to significantly increase both the success rate and quality of data generation. In a crucial departure from the "discard-on-failure" policy in Stage 1, the process begins after an initial solution fails validation in the sandbox. Instead of being discarded, the system first analyzes the specific type of error to trigger one of two tailored debugging strategies, which are detailed as follows:

Debugging Compilation or Runtime Errors. When a failure is caused by a compilation or runtime error, we employ a feedback-driven correction strategy. A new debugging prompt is constructed for the LLM, containing three key sources of information: 1) the verbatim error message from the compiler or runtime; 2) a pre-authored Debugging Guideline with expert strategies for common language-specific issues; 3) relevant chunks from the documentation using BM25 retrieval. By synthesizing these combined sources of feedback, the LLM is tasked with diagnosing the root cause of the error and generating a revised code solution.

Debugging Unit Test Failures via Pseudo Ground Truth. For logical flaws, we employ a distinct cross-lingual debugging strategy with the core objective of transferring the model's strong reasoning capabilities from a high-resource language (Python) to our target language. To achieve this, the model is prompted to generate a Python solution that serves as a "Pseudo Ground Truth". This Python solution is generated only upon the first occurrence of a logical error for a given task. The LLM then performs a comparative analysis to identify and correct logical discrepancies in its low-resource code, with the process concluding once a revised version successfully passes all unit tests.

Iterative Refinement. The two debugging strategies described above both operate within an iterative refinement loop. After each correction attempt, the revised code is resubmitted to the sandbox for re-validation, creating a "debug-correct-revalidate" cycle. This cycle continues until either the code successfully passes all tests or a predefined maximum number of attempts is reached. Stage 2 enables the successful creation of more complex instruction-code pairs that would otherwise be filtered out, thereby elevating the overall quality and challenge level of the final dataset.

4 Experiments

4.1 Experimental Setup

This section details the models, parameters, and evaluation benchmarks used to validate our proposed framework. Our research primarily focuses on enhancing code generation capabilities for Ultra-Low-Resource Programming Languages (ULR-

Model	Method	HumanEval Pass@1	MBPP Pass@1	TengoEval Pass@1
Qwen-Coder-2.5-32B-Instruct	Zero-Shot	0.61	2.34	1.00
	Full-Syntax Prompting	33.5	36.8	23.0
	Few-Shot Examples	22.6	41.8	15.0
	Translation Examples	24.4	42.6	18.0
	Translation Rules	15.9	32.0	11.0
	SelfCodeAlign	61.6	65.1	49.0
	CodeRise (Ours)	**71.3**	**75.5**	**60.0**
Qwen-Coder-2.5-7B-Instruct	Zero-Shot	0.61	0.00	1.00
	Full-Syntax Prompting	12.2	19.0	14.0
	Few-Shot Examples	9.75	25.7	9.00
	Translation Examples	4.27	6.80	6.00
	Translation Rules	2.44	10.4	7.00
	SelfCodeAlign	39.6	41.7	31.0
	CodeRise (Ours)	**50.3**	**57.9**	**39.0**
Llama-3.1-8B-Instruct	Zero-Shot	2.44	3.13	2.00
	Full-Syntax Prompting	9.76	10.4	10.0
	Few-Shot Examples	15.2	29.0	11.0
	Translation Examples	14.6	30.4	7.00
	Translation Rules	3.67	2.86	6.00
	SelfCodeAlign	32.9	33.6	29.0
	CodeRise (Ours)	**35.4**	**36.7**	**31.0**
DeepSeek-R1-0528	Zero-Shot	18.3	27.8	16.0
	Full-Syntax Prompting	67.1	75.5	56.0

Table 2: Pass@1 Performance Comparison of CodeRise against training-free and training-based baselines on the migrated HumanEval and MBPP, and the domain-specific TengoEval benchmarks for the ULRPL Tengo.

PLs). For this purpose, we selected **Tengo** as the target ULRPL due to its inherent characteristics that exemplify the challenges associated with such low-resource environments.

Model We use the powerful Qwen-2.5-Coder-32B (Hui et al., 2024) as the data generator to create the training datasets in both Stage 1 and Stage 2. The models selected for fine-tuning are Qwen-2.5-Coder-32B, Qwen-2.5-Coder-7B-Instruct (Hui et al., 2024) and Llama-3.1-8B-Instruct (Grattafiori et al., 2024).

Baseline To evaluate our framework, we compare it against a comprehensive set of training-free and training-based baselines. The training-free methods include standard Zero-Shot and Few-Shot prompting, along with richer context strategies: Full-Syntax Prompting (providing the entire language manual in-context), Translation Rules and Translation Example(Giagnorio et al., 2025). Our training-based comparisons consist of a model fine-tuned via the SelfCodeAlign(Wei et al., 2024a).

Datasets and Metric To evaluate our framework, we employ three benchmarks. Since standard benchmarks for Tengo do not exist, we manually migrated two widely recognized code generation benchmarks: HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). Furthermore, to better assess real-world performance, we constructed a new benchmark, TengoEval, which contains 100 problems tailored to the practical applications of the language. We use Pass@1 (Chen et al., 2021) as our primary metric, which measures the percentage of problems solved in a single attempt.

4.2 Results

To evaluate the effectiveness of CodeRise, we conducted a comprehensive set of experiments, with the main results summarized in Table 2. This table compares the Pass@1 performance of our

Module	Frequency	Proportion(%)
text	6746	43.7 / -4.1
math	1562	10.1 / -1.1
rand	1373	8.9 / -1.1
fmt	1676	10.9 / +0.5
json	1057	6.8 / -0.4
enum	820	5.3 / -0.8
times	692	4.5 / -0.5
base64	504	3.3 / +1.1
os	652	4.2 / +4.1
hex	352	2.3 / +2.2

Table 3: Distribution of standard module usage from our 30k dataset, the statistics are based on the 15434 samples that involved API calls. The last column showing the change compared to the baseline in Table 1.

Method	HumanEval
CodeRise	**71.3**
w/o Diversity Strategy	70.7
w/o Curriculum Learning	64.0
w/o Self-Debugging	67.1
Stage 1 Only (30k data)	59.8
Stage 1 Only (2k data)	51.8

Table 4: Ablation study results (Pass@1) on HumanEval datasets using the Qwen-2.5-Coder-32B-Instruct model.

Method	Syntax	Pass Rate (%)
SelfCodeAlign	✓	26.24
CodeRise (Stage 1)	✓	23.64
CodeRise (Stage 2)	×	**62.86**

Table 5: Comparison of data generation efficiency and yield.

CodeRise framework against a diverse set of baselines on the migrated HumanEval and MBPP benchmarks. From these results, we can derive the following conclusions:

CodeRise significantly outperforms both training-free and strong training-based baselines across all tested open-source models. As shown in Table 2, when applied to the powerful Qwen-Coder-2.5-32B-Instruct model, our method outperforms all training-free and strong training-based baselines. Furthermore, its performance is highly competitive with the powerful DeepSeek-R1-0528 model, especially considering the latter relies on a Full-Syntax Prompting strategy that is exceptionally time-consuming due to its long context requirements. This trend of superior performance extends to the smaller models as well, on both Qwen-Coder-2.5-7B-Instruct and Llama-3.1-8B-Instruct, CodeRise consistently outperforms all baselines, demonstrating the stability.

Our framework effectively mitigates the long-tail distribution while preserving natural usage patterns. As demonstrated in Table 3, our method significantly rebalances the module usage distribution compared to the baseline (e.g., *os*, *hex*). Crucially, this is achieved without artificially distorting the overall distribution. This demonstrates that our framework addresses the most extreme aspects of the long-tail problem, resulting in a more diverse yet practical training dataset.

CodeRise significantly improves the efficiency and overall yield of the data generation process. As detailed in Table 5, our CodeRise (Stage 2) process achieves a final effective pass rate of 62.86%, a 2.4x increase in yield over the SelfCodeAlign baseline. It is worth noting that the initial pass rate of our CodeRise (Stage 1) generation is slightly lower than the baseline. This is an expected trade-off, as our diversity sampling introduces more challenging tasks. Ultimately, our framework's ability to robustly solve these harder problems through self-debugging leads to a superior outcome in both data quality and overall efficiency.

4.3 Analysis

4.3.1 Ablation

To better understand the contribution of each component in our CodeRise framework, we conduct a series of ablation studies on the HumanEval dataset. The results are presented in Table 4. We define the following ablation settings: a) Stage 1 Only: The model is fine-tuned on the dataset generated through the Stage 1 method; b) w/o Diversity Strategy: remove the module sampling component in CodeRise; c) w/o Curriculum Learning: The full CodeRise framework is used, but the adaptive curriculum in Stage 2 is disabled. Tasks are generated without any explicit difficulty guidance or tier-specific prompting; d) w/o Self-Debugging: The full CodeRise framework is used, but the multi-turn self-debugging mechanism in Stage 2 is disabled, and failed attempts are discarded; For a fair comparison, all models were fine-tuned on 30k samples. Based on the above ablation study, we can derive the following conclusions:

The Stage 2 self-refinement process is the primary driver of performance. The results in Table 4 clearly show a massive performance gap between our full framework and the Stage 1 Only model. We attribute this significant difference to the quality of the training data. The Stage 1 Only dataset is generated using a simple "discard-on-failure" policy without any curriculum or difficulty control. This inherently biases the dataset towards simpler problems that the model can solve in a single pass. In contrast, our Stage 2 process can generate and successfully solve much more complex problems, and training on this higher-quality data is what ultimately leads to the superior performance of our final model.

Adaptive curriculum and multi-turn self-debugging are all effective components. The adaptive curriculum proves to be the most impactful component, its removal (w/o Curriculum Learning) causes a substantial performance degradation. This highlights that systematically generating more challenging problems is essential for pushing the model beyond its initial capabilities. The self-debugging mechanism is the crucial counterpart to this process. Its removal (w/o Self-Debugging) leads to a 4.2% decrease in performance on HumanEval, demonstrating that providing the model with multiple attempts and reliable external feedback is what enables it to successfully solve the difficult tasks proposed by the curriculum. This aligns with our core philosophy: for ULRPLs where no powerful "teacher" model exists, it is more effective to introduce external, verifiable knowledge to guide a model's own refinement process. Fo details on self-debugging, please refer to the case study in Appendix B.

Finally, the w/o Diversity Strategy setting only leads to a small drop in performance. We attribute this phenomenon to the nature of the benchmark itself, HumanEval focuses more on self-contained algorithm problems, which rarely need the various library modules that our strategy promotes. While its impact is not fully captured by this specific test, we argue that the diversity strategy is critical for building a genuinely useful model for real-world applications, where proficiency across the full standard library is essential.

4.3.2 Sweet Spot of Warm-up

An important consideration for CodeRise is determining the optimal amount of Stage 1 data for the initial fine-tuning. The objective is to achieve two goals: (1) effectively reducing execution errors in the target language; (2) preserving the model's general ability, it supports our aim of using a single model to save significant computational resources. To find this balance, we tracked the model's performance at 100-step intervals (batch size = 8). At each checkpoint, we evaluated its Execution Error Rate on a unified set of 5k instructions, and the performance on HumanEval-Python. As shown in Figure 3, the execution error rate on Tengo drops from nearly 100% to under 30% in the first 200 training steps. The intersection of these trends reveals a "sweet spot" around 200-300 steps. Therefore, we identify this range as the optimal warm-up point, as it achieves a substantial reduction in execution errors at a minimal cost to the model's general capabilities.

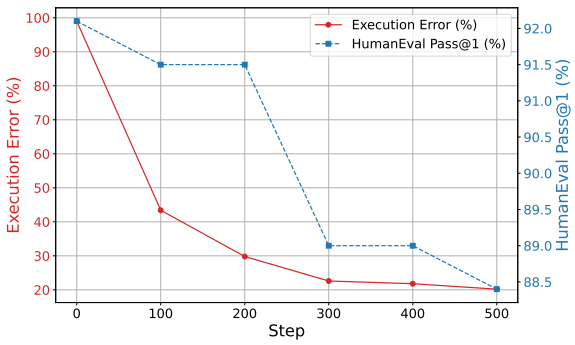

Figure 3: Identifying the sweet spot for the Stage 1 syntax warm-up. We plot the Execution Error Rate (%) on Tengo (left) and the Pass@1 on HumanEval-Python (right) against the number of fine-tuning steps.

5 Conclusion

In this paper, we addressed the critical challenge of applying Large Language Models to ULRPLs, where progress has been hindered by a lack of high-quality training data. We identified two primary obstacles for existing self-generation methods: a severe cold-start problem and the tendency to produce datasets with a long-tail distribution. We introduced CodeRise, a novel two-stage framework that overcomes these issues. By first using syntax-guided generation and a diversity-focused sampling strategy, and then refining the model with an adaptive curriculum and a multi-turn self-debugging mechanism, CodeRise generates a diverse, complex, and high-quality dataset. Our experiments on the ULRPL Tengo demonstrate that this approach significantly outperforms a wide range of baselines, proving the effectiveness of our methodology.

Limitations

Our study has two primary limitations that open avenues for future research. First, our empirical validation is confined to Tengo, a scripting language that shares syntactic similarities with high-resource languages. The effectiveness of our Syntax-Guided Generation might be partially attributed to this similarity, as transferring latent knowledge from the base model is more straightforward. Generalizing CodeRise to syntactically heterogeneous ULRPLs (e.g., stack-based or declarative) remains a critical next step and may require fundamental adjustments to the bootstrapping process. Second, our current instruction generation process does not account for the domain-specific nature of many ULRPLs. While our generated data is syntactically valid, it may not reflect the idiomatic or practical use cases of the target language. Future work should focus on developing mechanisms to align instruction generation with the intended application scope, thereby improving the real-world relevance of the synthetic dataset.

References

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. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *Preprint*, arXiv:2207.14255.

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. *Proc. ACM Program. Lang.*, 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, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *Preprint*, arXiv:2208.08227.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, and 1 others. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Yifeng Ding, Hantian Ding, Shiqi Wang, Qing Sun, Varun Kumar, and Zijian Wang. 2024. Horizon-length prediction: Advancing fill-in-the-middle capabilities for code generation with lookahead planning. *CoRR*, abs/2410.03103.

Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 1229–1241. ACM.

Alessandro Giagnorio, Alberto Martin-Lopez, and Gabriele Bavota. 2025. Enhancing code generation for low-resource languages: No silver bullet. *Preprint*, arXiv:2501.19085.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, and 1 others. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, and 1 others. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *Preprint*, arXiv:2401.14196.

Hojae Han, Jaejin Kim, Jaeseok Yoo, Youngwon Lee, and Seung-won Hwang. 2024. Archcode: Incorporating software requirements in code generation with large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 13520–13552. Association for Computational Linguistics.

Jamie Hayes, Ilia Shumailov, William P. Porter, and Aneesh Pappu. 2025. Measuring memorization in RLHF for code completion. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, and Others. 2024. Qwen2.5-coder technical report. *CoRR*, abs/2409.12186.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *CoRR*, abs/2310.06770.

Sathvik Joel, Jie JW Wu, and Fatemeh H. Fard. 2024. A survey on llm-based code generation for low-resource and domain-specific programming languages. *Preprint*, arXiv:2410.03981.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2023. The stack: 3 TB of permissively licensed source code. *Trans. Mach. Learn. Res.*, 2023.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 18319–18345. PMLR.

Kaixin Li, Qisheng Hu, James Xu Zhao, Hui Chen, Yuxi Xie, Tiedong Liu, Michael Shieh, and Junxian He. 2024. Instructcoder: Instruction tuning large language models for code editing. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics, ACL 2024 - Student Research Workshop, Bangkok, Thailand, August 11-16, 2024*, pages 50–70. Association for Computational Linguistics.

Jimmy Lin, Xueguang Ma, Sheng-Chieh Lin, Jheng-Hong Yang, Ronak Pradeep, and Rodrigo Nogueira. 2021. Pyserini: A Python toolkit for reproducible information retrieval research with sparse and dense representations. In *Proceedings of the 44th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2021)*, pages 2356–2362.

Bingchang Liu, Chaoyu Chen, Zi Gong, Cong Liao, Huan Wang, Zhichao Lei, Ming Liang, Dajun Chen, Min Shen, Hailian Zhou, Wei Jiang, Hang Yu, and Jianguo Li. 2024. Mftcoder: Boosting code llms with multitask fine-tuning. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2024, Barcelona, Spain, August 25-29, 2024*, pages 5430–5441. ACM.

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. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Federico Mora, Justin Wong, Haley Lepe, Sahil Bhatia, Karim Elmaaroufi, George Varghese, Joseph E. Gonzalez, Elizabeth Polgreen, and Sanjit A. Seshia. 2024. Synthetic programming elicitation for text-to-code in very low-resource programming and formal languages. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

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

OpenAI, :, Aaron Jaech, Adam Kalai, Adam Lerer, and 1 others. 2024a. Openai o1 system card. *Preprint*, arXiv:2412.16720.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, and 1 others. 2024b. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Long Ouyang, Jeffrey 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 F. Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 82:1–82:13. ACM.

Stephen Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, and 1 others. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Marc Szafraniec, Baptiste Rozière, Hugh Leather, Patrick Labatut, François Charton, and Gabriel Synnaeve. 2023. Code translation with compiler representations. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Hui Haotian, Liu Weichuan, Zhiyuan Liu, and Maosong Sun. 2024. DebugBench: Evaluating debugging capability of large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 4173–4198, Bangkok, Thailand. Association for Computational Linguistics.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 13484–13508. Association for Computational Linguistics.

Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. In *International Conference on Learning Representations*.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro von Werra, Arjun Guha, and Lingming Zhang. 2024a. Selfcodealign: Self-alignment for code generation. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024b. Magicoder: Empowering code generation with oss-instruct. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. 2025. Swe-fixer: Training open-source llms for effective and efficient github issue resolution. *CoRR*, abs/2501.05040.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2024. Wizardlm: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Min Xue, Artur Andrzejak, and Marla Leuther. 2024. An interpretable error correction method for enhancing code-to-code translation. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Preprint*, arXiv:2405.15793.

Xinran Yu, Chun Li, Minxue Pan, and Xuandong Li. 2024a. Droidcoder: Enhanced android code completion with context-enriched retrieval-augmented generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, pages 681–693. ACM.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2024b. Wavecoder: Widespread and versatile enhancement for code large language models by instruction tuning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 5140–5153. Association for Computational Linguistics.

Lin Zheng, Jianbo Yuan, Zhi Zhang, Hongxia Yang, and Lingpeng Kong. 2024. Self-infilling code generation. *Preprint*, arXiv:2311.17972.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 851–870, Bangkok, Thailand. Association for Computational Linguistics.

A Implementation Details

This section provides the implementation details of CodeRise to ensure full reproducibility. We report the key hyperparameters for our data generation pipeline. For all inference processes conducted during testing and evaluation, the sampling temperature was consistently set to $T = 0$.

A.1 Syntax Warm-up

For the initial syntax warm-up, we fine-tuned the Qwen-2.5-Coder-32B-Instruct model on a dataset of 2k samples generated from Stage 1. The training was conducted for 1 epoch, with a learning rate of 2×10^{-5} , and the batch size is set to 8. We utilized a cosine scheduler and AdamW optimizer for the training process.

A.2 Main Model Training

We fine-tuned three models on the 30k samples dataset generated by CodeRise. All training runs were conducted for 3 epochs using the AdamW optimizer with a cosine learning rate scheduler and a maximum sequence length of 2048 tokens. No warm-up steps were used. For Qwen-2.5-Coder-32B-Instruct, we used a learning rate of 2×10^{-5} with a batch size of 8. For Qwen-2.5-Coder-7B-Instruct and Llama-3.1-8B-Instruct, we used a learning rate of 4×10^{-5} with a batch size of 32.

A.3 Seed CodeBase

To ensure a fair comparison with prior work, we use the same seed codebase[2](Wei et al., 2024a) as the SelfCodeAlign framework, which consists of 574k functions. This dataset is a curated collection of functions originally sourced from The Stack V1[3](Kocetkov et al., 2023) corpus, a 3.1 TB dataset consisting of permissively licensed source code in 30 programming languages.

B Case Study

Figure 4 presents a case study where the model was tasked with calculating Euclidean distance. The initial code contained a compilation error, it used tuple assignment to unpack coordinates, which is a common feature in Go but unsupported in Tengo. Upon failure, it receives three complementary pieces of information: the direct compiler error identifies what is wrong; the pre-authored Debugging Guideline explains why it is wrong in Tengo and provides the correct pattern; and the retrieved documentation confirms that tuple assignment is a feature explicitly absent from the language. This multi-faceted feedback enables the model to effectively diagnose and resolve the language-specific syntactic issue.

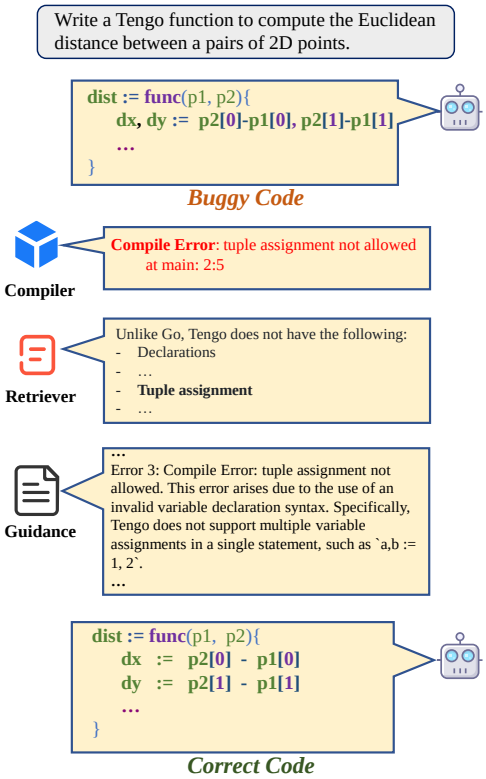

Figure 4: A case study of the CodeRise mechanism. An Compile Error caused by tuple assignment is resolved after the model receives synergistic feedback from the compiler, a documentation retriever, and our pre-authored guidance.

C Document Retrieval Strategy

The document was divided into three structurally parts: **(1) Basic Syntax**, **(2) Built-in Functions**, and **(3) Standard Libraries**. The chunking strategy was tailored to the characteristics of each section. For **Basic Syntax** and **Built-in Functions**, we leveraged the native Markdown hierarchy, treating each top-level heading as an individual chunk. In contrast, the **Standard Libraries** section, which consists of numerous concise function descriptions, was uniformly partitioned by grouping every three consecutive function definitions into one chunk. This design ensures semantic coherence within chunks while maintaining consistent retrieval gran-

[2] https://huggingface.co/datasets/bigcode/python-stack-v1-functions-filtered

[3] https://huggingface.co/datasets/bigcode/the-stack

ularity across heterogeneous documentation types.

Since retrieval is only triggered for *compilation* or *runtime* errors, BM25(Robertson and Zaragoza, 2009) proves more effective than dense retrieval. We utilized the BM25 algorithm from the Pyserini(Lin et al., 2021) with its default settings. The feedback messages from compilers often contain explicit lexical cues that directly correspond to documentation text, such as "Runtime Error: wrong number of arguments in call to user-function:has_prefix". In such cases, keyword-based retrieval better exploits these literal overlaps, whereas dense embeddings tend to smooth away the precise token-level distinctions that are crucial for error localization and resolution. Consequently, BM25 provides more reliable and interpretable matches for this type of debugging-oriented retrieval.