Language Model Adaptation for Low-Resource Programming Languages: A Comparative Study

Ananya Singha
Microsoft, India
ananyasingha@microsoft.com

Mukul Singh
Microsoft, USA
singhmukul@microsoft.com

Hosein Hasanbeig
Microsoft, USA
hosein.hasanbeig@microsoft.com

Arjun Radhakrishna
Microsoft, USA
arradha@microsoft.com

Sumit Gulwani
Microsoft, USA
sumitg@microsoft.com

Abstract

Large Language Models (LLMs) have achieved remarkable success in code generation, yet their capabilities remain predominantly concentrated in well-resourced programming languages such as Python and Java. In contrast, low-resource programming languages present a significant challenge due to limited available data and unique syntax features. In this paper, we systematically implement and evaluate four core adaptation techniques (retrieval-augmented generation, agentic architectures, tool calling and feedback guided generation) to understand how these models can be better improved for underrepresented programming languages. Our findings reveal that tool calling is particularly effective for low-resource languages, outperforming its performance on high-resource counterparts. Conversely, high-resource languages show a stronger preference for agentic workflows and RAG, likely due to the models' deeper familiarity and pretraining exposure to these languages.

1 Introduction

Recent years have seen a surge of significant advancement in code-oriented LLMs across a variety of languages. These efforts include Jetbrain Mellum JetBrains (2024), OpenCoder Huang et al. (2024), Meta LLM Compiler Cummins et al. (2024), StarCoder Lozhkov et al. (2024), CodeGeeX Zheng et al. (2023), CodeT5 Wang et al. (2021, 2023), CodeBERT, PLBART and UniXcoder Guo et al. (2022), AlphaCode Li et al. (2022), InCoder Fried et al. (2022), PolyCoder Xu et al. (2022), CodeGen Nijkamp et al. (2022), industry systems such as GitHub Copilot, Meta Code Llama Roziere et al. (2023), Google Codey, and BigCode StarCoder Li et al. (2023).

However, these successes have been skewed towards well-represented programming languages, such as Python and Java, where abundant training data is available. Low-resource programming languages, i.e., those with relatively little public code or documentation, remain a challenge.

Just as LLMs for natural language struggle with low-resource human languages, e.g., languages with limited text corpora, code-oriented LLMs find it difficult to achieve proficiency in less common programming languages. Challenges for low-resource natural languages include data scarcity, vocabulary issues, tokenization issues, wrong function usage and domain mismatch. Solutions include multilingual pretraining such as XLM-R Conneau et al. (2019)), transfer learning, data augmentation

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: .

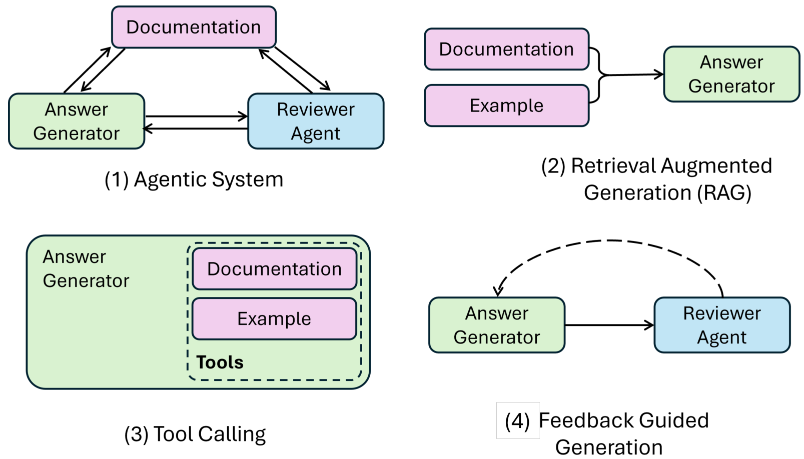

Figure 1: Overview of adaptation methods evaluated for code generation in low-resource programming languages.

and back-translation Sennrich et al. (2015); Khatry et al. (2023), unsupervised or weakly supervised learning, and tokenizer adaptation.

Challenges for low-resource programming languages mirror those found in low-resource natural languages, including limited training data, diverse syntax and library support, and increased evaluation difficulty. Addressing these issues in code-focused LLMs has prompted several strategies. Approaches include multilingual data sampling and balanced training Li et al. (2023), the use of shared sub-word vocabularies to facilitate cross-language generalization Roziere et al. (2020)), cross-language transfer learning, leveraging online documentation Puri et al. (2021), and code translation methods Lu et al. (2021); Singh et al. (2023).

Through these advances, noteworthy trends have emerged, such as the use of Retrieval-Augmented Generation (RAG) Yu et al. (2024), agentic architectures Plaat et al. (2025), tool-calling and feedback guided generation methods for more efficient code generation.

In this paper, we provide a systematic analysis of core techniques driving the state-of-the-art in code generation for low-resource programming languages: RAG, agentic architectures, tool calling and execution guided generation. To assess their practical utility, we implemented each technique and conducted experiments to evaluate their effectiveness and limitations in extending LLM capabilities to underrepresented programming languages. Our work situates these advancements within the broader landscape, providing empirical insights into their impact and areas for improvement.

2 Low-Resource Languages

We operationally define *low-resource languages* as those exhibiting two main criteria:

1. **Quantitative Low-Resource Status:** Each language is classified as low-resource according to widely recognized industry benchmarks, specifically the TIOBE Index, GitHub Language Rankings, and IEEE Spectrum. These metrics reflect both community adoption and representation in public code repositories.
2. **Availability of Authoritative Documentation:** Despite their relative scarcity in training corpora and global usage, we require every selected language to be supported by accessible, official documentation. This criterion is essential for tool-calling and retrieval-based methods, enabling standardized and fair evaluation.

This definition is motivated by and extends prior work Roziere et al. (2020). This dual-pronged approach ensures that our focus languages (Ada, Swift, Prolog, Clojure, Dart, Elixir) are representative of real-world low-resource settings while remaining evaluable under controlled, reproducible conditions.

Table 1 summarizes the low prevalence of these languages across major benchmarks, underscoring their "low-resource" status within the broader programming landscape GitHub (2024); IEEE Spectrum (2024); TIOBE Software BV (2024).

Table 1: Prevalence of target languages in major industry benchmarks, demonstrating their low-resource status.

Language	TIOBE (%)	GitHub Rank	IEEE Rank
Ada	0.79	23	25
Swift	0.73	21	20
Prolog	0.80	26	23
Clojure	0.10	32	37
Dart	0.50	32	27
Elixir	0.16	24	22

3 Adaptation Methods

We investigate core adaptation techniques for enabling effective code generation in low-resource programming languages. Each technique offers a different balance of scalability, cost, and language specificity. Below, we describe each method and how it applies to the context of low-resource programming languages.

3.1 Agentic Architecture

Agentic architectures structure systems around autonomous or semi-autonomous agents that coordinate decision-making through sequences of modular, interpretable steps. This paradigm is especially advantageous in low-resource programming language settings, where the limited availability of training data or task-specific expertise can be offset by dynamic planning and tool integration. Agentic systems decompose complex problems into smaller, solvable sub-tasks such as documentation retrieval, code synthesis, or output validation and allow the system to adaptively choose appropriate strategies at runtime Mondal et al. (2023).

In our implementation, we adopt a minimal agentic loop built on a reactive control flow, consisting of three cooperating agents:

Answering Agent Responsible for generating candidate answers to programming queries using a language model. It operates over the complete interaction history (turn-level memory) and incorporates relevant contextual cues from prior steps.

Documentation Lookup Agent Implements an embedding-based retrieval system over the programming language's official documentation corpus. Given a natural language query, it retrieves semantically similar documentation passages using a vector store (e.g., FAISS[1]) and passes these to the answering agent or reviewer.

Review and Feedback Agent Evaluates the output of the answering agent, optionally suggesting corrections or improvements. If the answer is unsatisfactory, it prompts the answering agent with refined instructions or additional retrieved context. This architecture allows for iterative refinement, grounded code generation, and dynamic fallback behavior—all critical for handling sparse or ambiguous queries in under-documented language environments. These concepts map closely to the taxonomy described by Sapkota et al. (2025), distinguishing agentic systems' multi-agent collaboration and orchestration capabilities.

3.2 Tool Calling

Tool calling enables a language model to extend its capabilities by invoking external programs or APIs, allowing it to offload computation, verification, or knowledge retrieval to specialized tools

[1]FAISS: Facebook AI Similarity Search, an open-source library for efficient similarity search and clustering of dense vectors Facebook AI Research (2017)

(e.g. Toolformer enables self-supervised learning of tool-calling behavior) Schick et al. (2023). In low-resource language contexts, where pretrained models lack deep syntactic or semantic fluency, tool calling bridges the capability gap by enabling real-time interaction with reliable resources.

We implement a tool calling framework with access to two key utilities:

> **Documentation Tool**: Accepts a natural language query and returns the most relevant documentation segment using an embedding-based retrieval mechanism. This tool interfaces with a preprocessed documentation corpus indexed using sentence-transformer embeddings.
>
> **Example Tool**: Retrieves the closest matching code example from an example bank, also using dense vector similarity. These examples are manually curated or programmatically extracted from source repositories, and they support analogical reasoning during code synthesis.

The system issues calls to these tools based on confidence heuristics and query complexity. Retrieved results are integrated into the generation pipeline either as grounding input to the model or as structured prompts. Frameworks such as ToolLLM likewise learn when and how to call APIs at scale, using datasets like ToolBench across thousands of endpoints Qin et al. (2024). Similar inspiration comes from ToolGen, which encodes API invocation as part of token sequences for large-scale tool retrieval and structured calling Wang et al. (2024).

3.3 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) is a hybrid approach that combines generative language modeling with non-parametric retrieval Lewis et al. (2020). It is particularly effective for low-resource languages, where direct model supervision is limited. RAG leverages an external corpus to augment the model's generation capabilities with grounded, factual context retrieved on demand.

In our setting, we use the state-of-the-art RAG system for low resource languages Dutta et al. (2024) which is based on a dual-retrieval RAG pipeline optimized for code-related tasks:
(A) We maintain two separate corpora: (1) the official documentation and (2) a curated set of real-world code examples. Both are encoded using transformer-based embedding models and stored in efficient vector indices. (B) Given a user query, we first perform an initial round of embedding-based retrieval to identify top-k relevant entries from each corpus independently. (C) In the second stage, these retrieved items are re-ranked based on their contextual relevance to the input prompt (using cross-encoder scoring), and a final set of passages/examples is concatenated with the user query to form the model input. Recent work such as cAST explores structure-aware chunking via AST to better segment code documentation for RAG pipelines Zhang et al. (2025).

This approach allows the model to remain lightweight while being augmented with semantically relevant, high-precision content. By externalizing domain knowledge, RAG improves interpretability and generalizability across previously unseen programming scenarios Zhou et al. (2024).

3.4 Execution-Guided Generation

Learning from failures is a well-established paradigm in language model research, where models iteratively refine their outputs based on feedback from execution results. Several works in the literature Shinn et al. (2023); Gupta et al. (2024) employ reviewer agents to analyze model outputs against ground truth, providing feedback that guides subsequent generations toward better problem-solving.

In our setup, we implement a similar reviewer agent that consumes the model-generated code along with execution feedback and the output produced by the code executor. The reviewer agent then generates actionable feedback-highlighting execution or syntax errors and suggesting improvements. This feedback is appended to the original prompt and passed back to the model. This iterative loop helps the model learn from its mistakes and progressively refine its output, ultimately generating syntactically correct and executable code that solves the target task.

4 Experimental Setup

For embeddings we use the `all-MiniLM-L6-v2` model from sentence transformer library Reimers and Gurevych (2019). For the cross-encoder model for RAG setup, we use the `ms-marco-MiniLM-L6-v2` model. We use cosine-similarity to measure embedding distance. $k = 10$ for all retrieval and reranking unless otherwise specified. We maintain a prompt input and output token limit of 4096 each. For RAG, examples are dynamically removed in increasing order of relevance in case of limit overflow.

We use six models for evaluation, listed next with the abbreviation in brackets that will be used for the rest of the paper for that model: (1) `microsoft/phi-4` (Phi-4); (2) `microsoft/phi-3-instruct-128k` (Phi-3); (3) OpenAI GPT O1 (O1); (4) OpenAI GPT-4o (4o); (5) `deepseek-r1-distill-qwen-32b` (DeepSeek); (6) `mistral-instruct-7b` (Mistral).

All adaptation method parameters and model checkpoints are fully enumerated and can be found at `redacted for anonymity`.

4.1 Documentation and Example Extraction

Low-resource programming languages often lack high-quality online documentation, making it difficult for off-the-shelf models to learn their syntax and semantics. To mitigate this, we collect and parse official documentation for six such languages: Ada, Clojure, Dart, Elixir, Prolog, and Swift. Using custom scripts and the Python `BeautifulSoup` library, we recursively crawl and extract structured information–including classes, functions, methods, APIs, and associated metadata such as descriptions, signatures, and usage details and code examples. Summary statistics of retrieved documents are presented in Table 2 .

Table 2: Statistics on retrieved documentations.

Language	**Pages**	**Sections**	**Items**	**Samples with Examples**
Ada	235	170	220	148
Clojure	3	98	114	109
Dart	9	27	1189	590
Elixir	182	79	119	64
Prolog	64	137	241	682
Swift	12	89	125	94

4.2 Tasks

To cover a wide variety of code tasks covering (1) generation, (2) understanding, and (3) repair, we use a Multiple-Choice Question Answering (MCQA) dataset over low resource languages that constructs MCQ tasks over these. These tasks are deterministically generated over the CodeNet dataset. [2] Other than MCQA, we also consider the code generation task using the MultiPL-E Cassano et al. (2023) benchmark.

4.3 Metrics

To evaluate the effectiveness of various SOTA adaption techniques for code-generation in low resource programming language, we employ two primary evaluation metrics. First, we measure accuracy over the MCQA tasks. Second, we report the pass@k (i.e., functional correctness) for the MultiPL-E Cassano et al. (2023) benchmark.

4.4 Termination Criteria

For iterative methods such as execution-guided or feedback-driven generation, we use a fixed maximum number of refinement steps ($N = 5$ by default). If a solution is not found within these steps, the process terminates and the most recent output is returned.

[2]The MCQA dataset can be accessed at `redacted for anonymity`

Table 3: Performance comparison of various adaption techniques across six low-resource programming languages using GPT-4o

(a) Multiple-Choice Question Answering(MCQA) (%)

Domain	*Base*	*Agentic*	*Tool*	*RAG*	*Feedback*
Ada	55.6	77.2	**86.7**	73.8	78.9
Swift	34.8	57.0	**69.1**	65.3	60.4
Prolog	31.2	44.5	55.4	51.7	**56.2**
Clojure	20.4	38.9	**49.8**	34.6	49.0
Dart	18.9	36.1	**50.0**	35.9	40.1
Elixir	21.1	38.0	**48.5**	33.4	42.6
Python	70.6	78.2	81.4	79.0	**85.6**
C++	72.3	78.8	75.9	83.1	**85.4**

(b) MultiPL-E Pass@1 (%) //

Domain	*Base*	*Agentic*	*Tool*	*RAG*	*Feedback*
Ada	45.2	69.8	**77.2**	62.3	66.1
Swift	32.8	42.5	**49.0**	45.1	48.5
Prolog	21.6	38.0	**45.6**	40.2	34.0
Clojure	36.7	**48.1**	**48.1**	43.5	37.2
Dart	29.1	43.2	42.3	**45.7**	39.3
Elixir	27.5	40.4	**46.9**	41.8	35.9
Python	88.1	**88.5**	87.3	83.2	86.7
C++	78.3	**80.4**	79.2	75.6	80.1

5 Results

5.1 Performance across various Techniques

Table 3 shows the performance of non-adapted base models as well as various adaptation techniques on six low resource programming languages for both (a) MCQA accuracy and (b) MultiPL-E Pass@1. We find that using tool-calling performs significantly better than any other approach for code-generation. Upon investigation, we see a few emergent patterns, (1) the model prefers requesting information (documentation tool) over proactively being given information (RAG) and is able to use the information better. We find that the model is more likely to use information provided when it request it rather than when it is included in the original prompt; (2) the model is able to reason better over new information when all the processing happens in the same turn memory (single model message history) as opposed to this process being split over multiple models (agentic architecture).

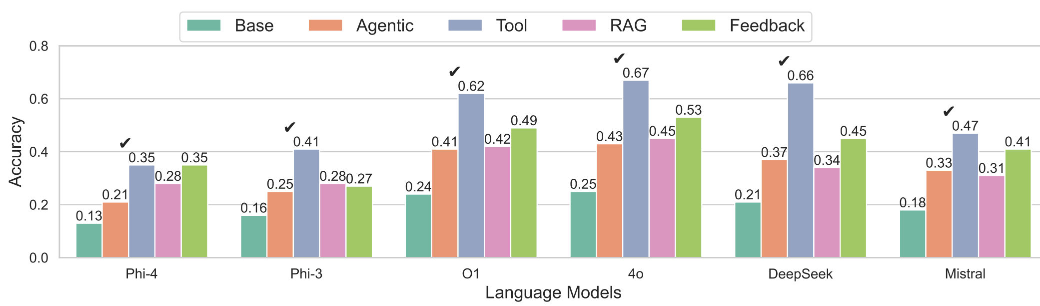

Figure 2: Performance variation of different language models across various adaptation techniques. The scores are averaged over six low-resource languages.

5.2 Do models have a preference?

To investigate whether certain models exhibit a preference for specific adaptation techniques, we analyze six models: Phi-4 Abdin et al. (2024b), Phi-3 Abdin et al. (2024a), Mistral-7B Jiang et al. (2023), DeepSeek-distill-Qwen-32B Guo et al. (2025), GPT-4o Hurst et al. (2024), and O1 Jaech et al. (2024). Figure 2 presents their performance across various setups, aggregated over all six low-resource programming languages. Our analysis reveals that tool calling generally yields the best results. However, smaller models (fewer than 20B parameters) tend to generate completions without invoking external tools, potentially due to lower tool-use confidence. This behavior likely contributes to RAG's relative effectiveness in such cases. We have also included a plot of model sizes versus tool-calling frequency (Figure 3).

Additionally, we conduct a detailed manual trajectory analysis across more than 100 samples for O1, GPT-4o, Phi-3, and Phi-4. The results are summarized in Table 4. Phi-4 and O1 exhibit shorter tool-call trajectories and reduced iteration, suggesting a Dunning-Kruger effect in low-resource scenarios. Conversely, Phi-3 and 4o engage in less explicit planning and answer verification, which appears beneficial on specific datasets (e.g., MCQA). This finding is consistent with recent studies on reasoning confusion and test-time scaling Ghosal et al. (2025).

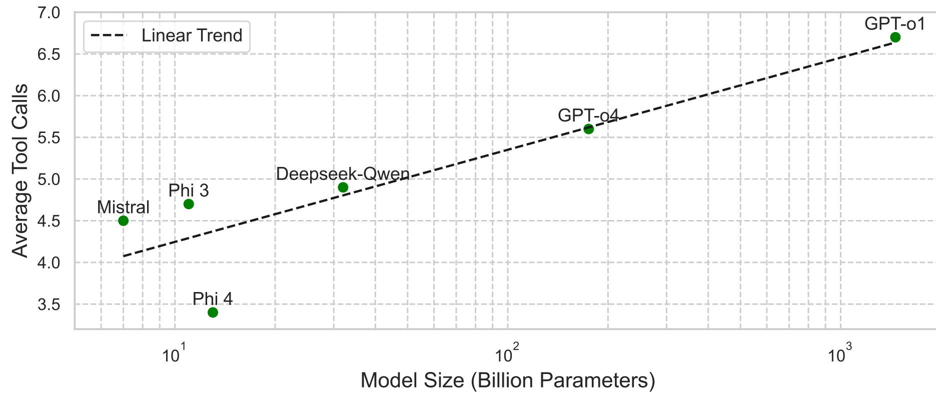

Figure 3: Model size versus tool-calling frequency. Smaller models tend to generate completions without invoking external tools, whereas larger models are more likely to utilize such tools.

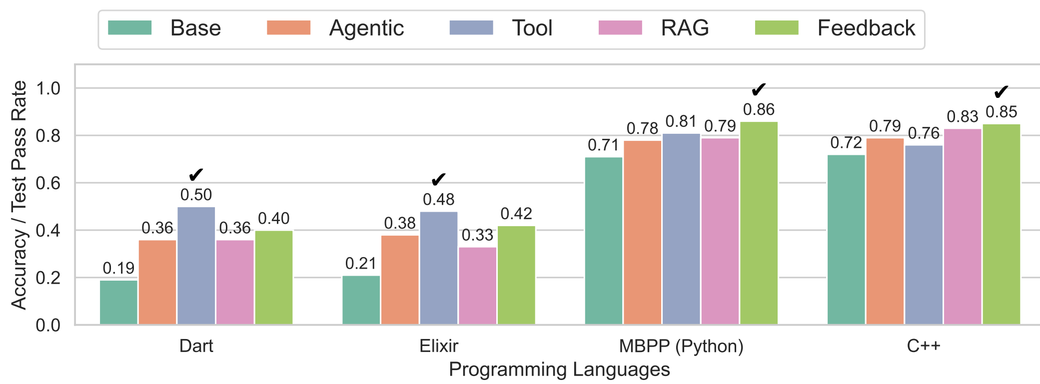

Figure 4: Performance across various adaptation methods for high and low resource programming languages using GPT-4o. For high-resource languages such as Python and C++, tool-based approaches underperform compared to retrieval-augmented generation (RAG) and agentic methods.

5.3 Do low resource programming languages behave differently?

We also investigate whether low-resource programming languages exhibit different behavior compared to high-resource ones. As shown in Figure 4, for high-resource languages such as Python and C++, tool-based approaches underperform compared to retrieval-augmented generation (RAG) and agentic methods. This trend is in stark contrast to the patterns observed for low-resource languages.

One possible explanation is that language models, having been extensively trained on high-resource languages, are more capable of handling tasks in a standalone manner. Consequently, they perform better in settings like the agentic workflow, and RAG where the history of information may not be shared. We also present results on ablation and computation overhead in the appendix section A.1

Table 4: Manual trajectory analysis across models. Phi-4 and O1 exhibit shorter tool-call trajectories. Conversely, Phi-3 and 4o engage in less explicit planning and answer verification.

Metric	Phi-3	Phi-4	GPT-4o	GPT-O1
Number of Tool Calls	4.5	3.4	5.6	2.7
Planning (% of tasks)	26.4	45.5	34.2	68.8
Answer Verification (% of tasks)	9.3	23.3	5.5	39.2

6 Discussion

6.1 The Nature of Low-Resource Programming Languages

Our results highlight the multifaceted challenges posed by low-resource programming languages (LRPLs). Unlike low-resource human languages, LRPLs often lack not only training data but also infrastructure such as compilers, test suites, and community support. These deficiencies amplify the brittleness of code generation systems, particularly in zero- or few-shot settings. In addition, semantic drift due to ambiguous or undocumented language behavior may limit the utility of transfer learning from high-resource languages.

6.2 Adaptation Strategies: When and Why They Work

We observed distinct trade-offs across adaptation strategies: **Retrieval-Augmented Generation (RAG)** performed well when suitable exemplars were available, particularly for syntactically similar languages. **Agentic methods** shined in multi-step or tool-heavy reasoning tasks but often suffered from execution overhead or hallucinated workflows. **Tool-calling** was especially effective when deterministic APIs or interpreters existed, highlighting the importance of grounding model output in structured systems. **Execution-guided decoding** proved valuable for testable tasks but was brittle in languages lacking formal testing infrastructure.

These findings suggest that no single paradigm dominates universally; rather, the choice should depend on task properties, tooling availability, and the degree of syntactic or semantic similarity to high-resource languages.

Language Transfer and Structural Priors Cross-lingual transfer benefits were strongest when the LRPL shared syntax or semantics with a high-resource counterpart (e.g., OCaml and ReasonML, or R and S). This supports the idea that *structural priors* encoded in pretraining data influence model adaptability. However, languages with unique paradigms (e.g., logic, constraint-based, or stack-based languages) showed diminished transfer, reinforcing the need for inductive biases that better match these domains.

Limitations of Current Evaluation Benchmarks Our experiments relied on existing benchmarks such as MultiPL-E and LitBench. While useful, these datasets do not always reflect real-world development conditions -especially for LRPLs where codebases are smaller, community conventions are looser, and documentation is sparse. Moreover, many benchmarks are biased toward short, testable functions rather than full programs, limiting our understanding of how models handle more complex or compositional tasks in LRPLs.

Practical Implications for Language and Tool Designers Our findings suggest that even modest tooling support (e.g., test runners, interpreters, or linters) can significantly enhance model usability in LRPLs. Designers of new programming languages might consider how language design choices and ecosystem tooling impact LLM-based coding assistance - particularly if adoption by human developers is to be augmented by AI agents.

6.3 Future Directions

Several promising research directions emerge: **Unsupervised or synthetic pretraining** for LRPLs to simulate data-rich conditions; **Meta-learning or few-shot tuning** that adapts better to structural novelty; **Hybrid neuro-symbolic methods** that integrate learned policies with static analyzers or compilers; **Curriculum-based training** to scaffold knowledge from high-resource to low-resource languages progressively; **Dynamic prompt synthesis or retrieval** that generalizes across unseen language grammars. Such directions could address the brittleness and low generalizability currently observed in LRPL adaptation and open up broader language coverage in code intelligence systems.

7 Conclusion

Adapting LLMs to excel in low-resource programming languages is a multi-faceted challenge that has seen substantial progress. At a high level, recent successes are built on: leveraging transfer from high-resource languages, careful balanced training, data synthesis strategies, parameter-efficient adaptation, and robust benchmarking. Open contributions and the synergy between academia and industry have accelerated advances. Trends include continued data synthesis, integrating tool use, stronger parameter-adaptive fine-tuning, evolving benchmarks, and striving for not just syntactic but idiomatic, maintainable code. With these foundations, we may soon reach parity in LLM code generation across the broad diversity of programming languages.

8 Limitations

As the field continues to evolve rapidly, several efforts have addressed the challenges of low-resource programming languages. Our work offers timely insights into the effectiveness of key adaptation techniques; retrieval-augmented generation (RAG), agentic architectures, tool calling, and feedback-guided generation—in this context. However, certain limitations remain. First, our evaluation spans only six low-resource and two high-resource languages. While diverse, this selection may not reflect the full range of language complexity or generalize to niche or domain-specific languages. Second, we use SOTA adaptation techniques with tuned configurations to ensure consistency. This may underestimate the potential performance achievable with task-specific finetuning, particularly for agentic workflows and tool calling. Finally, we evaluate only a few open or accessible foundation models and adaptation techniques. This exclusion limits the completeness of our benchmarking relative to real-world usage scenarios. Future work could expand in these directions, as well as explore how user interaction patterns impact model performance in low-resource settings.

References

Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, Alon Benhaim aand Misha Bilenko, Johan Bjorck, Sébastien Bubeck, Martin Cai, Qin Cai, Vishrav Chaudhary, Dong Chen, Dongdong Chen, Weizhu Chen, Yen-Chun Chen, Yi-Ling Chen, Hao Cheng, Parul Chopra, Xiyang Dai, Matthew Dixon, Ronen Eldan, Victor Fragoso, Jianfeng Gao, Mei Gao, Min Gao, Amit Garg, Allie Del Giorno, Abhishek Goswami, Suriya Gunasekar, Emman Haider, Junheng Hao, Russell J. Hewett, Wenxiang Hu, Jamie Huynh, Dan Iter, Sam Ade Jacobs, Mojan Javaheripi, Xin Jin, Nikos Karampatziakis, Piero Kauffmann, Mahoud Khademi, Dongwoo Kim, Young Jin Kim, Lev Kurilenko, James R. Lee, Yin Tat Lee, Yuanzhi Li, Yunsheng Li, Chen Liang, Lars Liden, Xihui Lin, Zeqi Lin, Ce Liu, Liyuan Liu, Mengchen Liu, Weishung Liu, Xiaodong Liu, Chong Luo, Piyush Madan, Ali Mahmoudzadeh, David Majercak, Matt Mazzola, Caio César Teodoro Mendes, Arindam Mitra, Hardik Modi, Anh Nguyen, Brandon Norick, Barun Patra, Daniel Perez-Becker, Thomas Portet, Reid Pryzant, Heyang Qin, Marko Radmilac, Liliang Ren, Gustavo de Rosa, Corby Rosset, Sambudha Roy, Olatunji Ruwase, Olli Saarikivi, Amin Saied, Adil Salim, Michael Santacroce, Shital Shah, Ning Shang, Hiteshi Sharma, Yelong Shen, Swadheen Shukla, Xia Song, Masahiro Tanaka, Andrea Tupini, Praneetha Vaddamanu, Chunyu Wang, Guanhua Wang, Lijuan Wang, Shuohang Wang, Xin Wang, Yu Wang, Rachel Ward, Wen Wen, Philipp Witte, Haiping Wu, Xiaoxia Wu, Michael Wyatt, Bin Xiao, Can Xu, Jiahang Xu, Weijian Xu, Jilong Xue, Sonali Yadav, Fan Yang, Jianwei Yang, Yifan Yang, Ziyi Yang, Donghan Yu, Lu Yuan, Chenruidong Zhang, Cyril Zhang, Jianwen Zhang, Li Lyna Zhang, Yi Zhang, Yue Zhang, Yunan Zhang, and Xiren Zhou. 2024a. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv*.

Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J Hewett, Mojan Javaheripi, Piero Kauffmann, et al. 2024b. Phi-4 technical report. *arXiv preprint arXiv:2412.08905*.

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. 2023. Multipl-e: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691.

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. *arXiv preprint arXiv:1911.02116*.

Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2024. Meta Large Language Model Compiler: Foundation Models of Compiler Optimization. *arXiv preprint arXiv:2407.02524*.

Avik Dutta, Mukul Singh, Gust Verbruggen, Sumit Gulwani, and Vu Le. 2024. RAR: Retrieval-augmented retrieval for code generation in low resource languages. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 21506–21515, Miami, Florida, USA. Association for Computational Linguistics.

Facebook AI Research. 2017. Faiss: A library for efficient similarity search and clustering of dense vectors. `https://github.com/facebookresearch/faiss`.

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. *arXiv preprint arXiv:2204.05999*.

Soumya Suvra Ghosal, Souradip Chakraborty, Avinash Reddy, Yifu Lu, Mengdi Wang, Dinesh Manocha, Furong Huang, Mohammad Ghavamzadeh, and Amrit Singh Bedi. 2025. Does thinking more always help? understanding test-time scaling in reasoning models. *arXiv preprint arXiv:2506.04210*.

GitHub. 2024. Github language rankings. Accessed: 2025-06-30. Available at: https://octoverse.github.com/.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Priyanshu Gupta, Shashank Kirtania, Ananya Singha, Sumit Gulwani, Arjun Radhakrishna, Sherry Shi, and Gustavo Soares. 2024. Metareflection: Learning instructions for language agents using past reflections. *arXiv preprint arXiv:2405.13009*.

Siming Huang, Tianhao Cheng, J.K. Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J.H. Liu, Chenchen Zhang, et al. 2024. OpenCoder: The open cookbook for top-tier code large language models. *arXiv preprint arXiv:2411.04905*.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

IEEE Spectrum. 2024. The top programming languages. Accessed: 2025-06-30. Available at: https://spectrum.ieee.org/top-programming-languages.

Aaron Jaech et al. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720*.

JetBrains. 2024. Introducing Mellum: A Large Language Model Built for Developers. JetBrains Company Blog (Nov 2024). Proprietary code LLM announcement.

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

Anirudh Khatry, Joyce Cahoon, Jordan Henkel, Shaleen Deep, Venkatesh Emani, Avrilia Floratou, Sumit Gulwani, Vu Le, Mohammad Raza, Sherry Shi, Mukul Singh, and Ashish Tiwari. 2023. From words to code: Harnessing data for program synthesis from natural language.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*. Introduces the RAG framework combining seq2seq and dense-retrieval.

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*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. *arXiv preprint arXiv:2402.19173*.

Shuai Lu, Daya Li, Shuming Fang, Chaojie Xu, Shao Guo, Ming Zhang, Nan Zou, and Zhoujun Huang. 2021. Codexglue: a benchmark dataset and open challenge for code intelligence. *arXiv preprint arXiv:2102.04664*.

Shanka Subhra Mondal, Taylor W. Webb, and Ida Momennejad. 2023. Improving planning with large language models: A modular agentic architecture. In *arXiv preprint arXiv:2310.00194*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474*, 30.

Aske Plaat, Max van Duijn, Niki van Stein, Mike Preuss, Peter van der Putten, and Kees Joost Batenburg. 2025. Agentic large language models, a survey. *arXiv preprint arXiv:2503.23037*.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. Toolllm: Facilitating large language models to master 16,000+ real-world apis. In *International Conference on Learning Representations (ICLR)*. Introduced ToolBench and ToolLLaMA framework.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

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.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in neural information processing systems*, 33:20601–20611.

Ranjan Sapkota, Konstantinos I. Roumeliotis, and Manoj Karkee. 2025. Ai agents vs. agentic ai: A conceptual taxonomy, applications and challenges. *arXiv*, abs/2505.10468.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36. Self-supervised tool use via simple APIs.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Improving neural machine translation models with monolingual data. *arXiv preprint arXiv:1511.06709*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652.

Mukul Singh, José Cambronero, Sumit Gulwani, Vu Le, Carina Negreanu, Elnaz Nouri, Mohammad Raza, and Gust Verbruggen. 2023. Format5: Abstention and examples for conditional table formatting with natural language. *Proc. VLDB Endow.*, 17(3):497–510.

TIOBE Software BV. 2024. Tiobe index for june 2024. Accessed: 2025-06-30. Available at: https://www.tiobe.com/tiobe-index/.

Jonathan Wang et al. 2024. Toolgen: Tool retrieval and invocation via transformer tokens. *arXiv preprint arXiv:2410.03439*. Encodes API calls as transformer tokens for retrieval.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Yue Wang et al. 2023. Codet5+: Open code large language models for code understanding and generation. arXiv preprint arXiv:2306.04560.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN international symposium on machine programming*, pages 1–10.

Yue Yu, Wei Ping, Zihan Liu, et al. 2024. Retrieval-augmented generation for large language models. *arXiv preprint arXiv:2312.10997*.

Yi Zhang et al. 2025. cAST: Enhancing code rag with structure aware chunking via AST. *arXiv preprint arXiv:2506.15655*. Structure-aware chunking improves retrieval for code contexts.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684.

Yushu Zhou et al. 2024. A survey of retrieval-augmented language models (rallms). *ACM Computing Surveys*. Comprehensive survey of RAG and RA-LLM architectures and applications.

A Appendix

A.1 Extended Results

A.1.1 Ablation Tests

While model-based components such as review agent, feedback agents and retrievers might face challenges on low-resource languages, our experiments suggest that adaptation methods can still provide significant gains. Specifically:

- The retrieval models leverage embeddings trained on large code corpora, enabling some transfer even to low-resource languages.
- Review/feedback agents are often used to guide solution refinement, and their benefit is observed even when the underlying model has imperfect low-resource coverage.

There is a theoretical limitation in using one model for both review and generation as the gaps in information and biases will carry over. Past work Plaat et al. (2025) has shown that models are better at reviewing than generating. Furthermore, providing the reviewer with extra context improves its review ability. To evaluate this hypothesis, we have performed an ablation test over reviewer agent configurations and reported the results in Table 5. In one variant, the reviewer agent is simply the same model while in the context enriched variant, the reviewer agent is provided API documentation and examples retrieved from the corpus. The results indicate that providing context to the model performs significantly ($\approx 4.7\%$) better over the base model.

Table 5: Performance comparison: same model versus same model with context

Domain	Same Model	Same Model + Context
Ada	62.3	66.1
Swift	42.3	48.5
Prolog	30.8	34.0
Clojure	33.6	37.2
Dart	32.7	39.3
Elixir	29.6	35.9

A.1.2 Computational Overhead

While our primary focus was on accuracy and qualitative utility, we like to emphasize that computational and latency costs are important for real-world deployment.

Table 6: Computation cost by adaptation technique.

Technique	# Tokens	# Calls
Agentic	545.3	4.5
Tool	722.5	5.6
RAG	671.9	2.0
Feedback	486.5	2.6

Table 6 and 7 reveal a clear trade-off between improved task performance and the computational overhead introduced by various adaptation methods. Tool-based and agentic techniques tend to incur a higher number of API calls and token consumption per task compared to lighter-weight approaches such as Feedback (2.6 calls, 486.5 tokens) or RAG (2.0 calls, 671.9 tokens). This increase in computational cost is often accompanied by measurable performance gains, especially for more complex or ambiguous queries where access to external resources or multi-step reasoning is beneficial.

However, for large-scale deployments or latency-sensitive applications, this overhead may become a limiting factor. Frequent tool calls and increased token usage can translate directly to higher latency

Table 7: Computation cost by model.

Model	# Tokens	# Calls
Phi-4	385.1	3.4
Phi-3	380.5	4.5
GPT-O1 (medium reasoning)	1081.3	2.7
GPT-4o	320.4	5.6
Mistral-7B	416.4	1.8
DeepSeek-distill-Qwen-32B	820.7	4.2

and operational cost, particularly when scaling to thousands or millions of queries. In scenarios such as real-time or interactive coding assistants, some degree of overhead (e.g., a moderate increase in calls or tokens) may be acceptable if it substantially improves the relevance or correctness of responses. For these use cases, the user-perceived benefit of higher-quality assistance justifies the added latency or resource consumption.

Conversely, in high-throughput settings—such as batch processing, code review pipelines, or low-power/on-device environments—lighter-weight techniques (e.g., Feedback or RAG alone) may be preferred. These methods can offer reasonable accuracy and coverage with significantly reduced computational and latency costs. Our experimental results summarize these trade-offs, supporting the need to balance adaptation strategy selection with application requirements and system constraints.

NeurIPS Paper Checklist

The checklist is designed to encourage best practices for responsible machine learning research, addressing issues of reproducibility, transparency, research ethics, and societal impact. Do not remove the checklist: **The papers not including the checklist will be desk rejected.** The checklist should follow the references and follow the (optional) supplemental material. The checklist does NOT count towards the page limit.

Please read the checklist guidelines carefully for information on how to answer these questions. For each question in the checklist:

- You should answer [Yes] , [No] , or [NA] .
- [NA] means either that the question is Not Applicable for that particular paper or the relevant information is Not Available.
- Please provide a short (1–2 sentence) justification right after your answer (even for NA).

The checklist answers are an integral part of your paper submission. They are visible to the reviewers, area chairs, senior area chairs, and ethics reviewers. You will be asked to also include it (after eventual revisions) with the final version of your paper, and its final version will be published with the paper.

The reviewers of your paper will be asked to use the checklist as one of the factors in their evaluation. While "[Yes] " is generally preferable to "[No] ", it is perfectly acceptable to answer "[No] " provided a proper justification is given (e.g., "error bars are not reported because it would be too computationally expensive" or "we were unable to find the license for the dataset we used"). In general, answering "[No] " or "[NA] " is not grounds for rejection. While the questions are phrased in a binary way, we acknowledge that the true answer is often more nuanced, so please just use your best judgment and write a justification to elaborate. All supporting evidence can appear either in the main paper or the supplemental material, provided in appendix. If you answer [Yes] to a question, in the justification please point to the section(s) where related material for the question can be found.

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: We have clearly listed our contribution in the Introduction Section 1. We also briefly summarize it in abstract.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We have a Limitation section (Section 7) describing the limitations.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.

- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

   Justification: The paper studies and evaluate the four core adaptation techniques to understand how they work for low-resource programming languages.

   Guidelines:

   - The answer NA means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We have an experimental setup Section 4 to educate reader about the experimentation setup and reproducibility. We also aim to release our train/test setup for the evaluation.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.

- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: We will release code and database.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyper-parameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

Answer: [Yes]

Justification: Yes we provide all information related to train/test setup in Section 4.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Yes, all the results and the plots that are shows in Section 5 Result section shows all the experimentation results using the metrics that are defined in Section 4.2.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
   - If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We have describe all about the compute resource in Section 4.3 along with costs.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.

- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We have reviewed and follow NeurIPS Code of Ethics.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [NA]

    Justification: Our work studies and evaluate the four core adaptation techniques to understand how they work for low-resource programming languages. This this work does have any positive and negative direct societal. Though we have a limitation section that talks how our evaluation setup is limited, in sense of scope of evaluation tasks.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: We don't release any trained models or dataset.

    Guidelines:

- The answer NA means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We have cited all assets (datasets) used along citations for all the related work.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [NA]

    Justification: We are not releasing any new assets as a part of this work.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

Justification: NA

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: No human subjects were used for these research.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: The paper studies and evaluate the four core adaptation techniques on LLMs to understand how they work for low-resource programming languages. We have declared this explicitly in abstract and introduction.

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.