Adapting Language Models for Low-Resource Programming Languages

Anonymous EMNLP submission

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 and back-translation (Sennrich et al., 2015), 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 available documentation (Puri et al., 2021), and code translation methods (Lu et al., 2021).

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

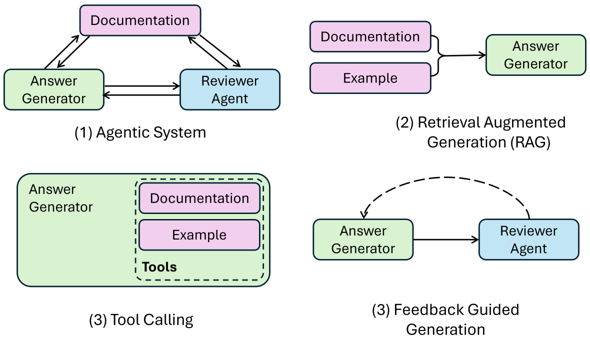

Figure 1: Overview of adaptation methods evaluated for code generation in low-resource programming languages.

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 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.

2.1 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.

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) 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.

2.2 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. 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.

2.3 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) is a hybrid approach that combines generative language modeling with non-parametric retrieval. 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 implement 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.

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.

2.4 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.

3 Experimental Setup

3.1 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.

3.2 Tasks

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

3.3 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 Results

4.1 Performance across various Techniques

Table 1 shows the performance of 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

Table 1: Performance comparison of various adaption techniques across six low-resource programming languages using GPT-4o

(a) MCQA Accuracy (%)

Domain	*Agentic*	*Tool*	*RAG*	*Feedback*
Ada	77.2	**86.7**	73.8	78.9
Swift	57.0	**69.1**	65.3	60.4
Prolog	44.5	55.4	51.7	**56.2**
Clojure	38.9	**49.8**	34.6	49.0
Dart	36.1	**50.0**	35.9	40.1
Elixir	38.0	**48.5**	33.4	42.6

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

Domain	*Agentic*	*Tool*	*RAG*	*Feedback*
Ada	69.8	**77.2**	62.3	66.1
Swift	42.5	**49.0**	45.1	48.5
Prolog	38.0	**45.6**	40.2	34.0
Clojure	**48.1**	**48.1**	43.5	37.2
Dart	**43.2**	42.3	45.7	39.3
Elixir	40.4	**46.9**	41.8	35.9

message history) as opposed to this process being split over multiple models (agentic architecture).

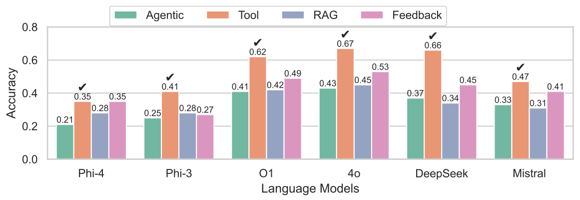

Figure 2: Performance variation of different language models across various adaptation techniques. The scores are averaged over six low-resource languages.

4.2 Do models have a preference?

To investigate whether certain models exhibit a preference for specific adaptation techniques, we analyze six models: Phi-4, Mixtral-7B, DeepSeek-distill-Qwen-7B, GPT-4o, GPT-4.1, and GPT-o1. 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) achieve performance comparable to tool calling when using retrieval-augmented generation (RAG). A closer examination of model traces indicates that these smaller models are less inclined to invoke tools, often opting to generate answers directly from the prompt. This behavior likely contributes to RAG's relative effectiveness in such cases.

4.3 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 3, for high-resource languages such as Python and C++, tool-based approaches underperform compared to retrieval-augmented generation (RAG) and

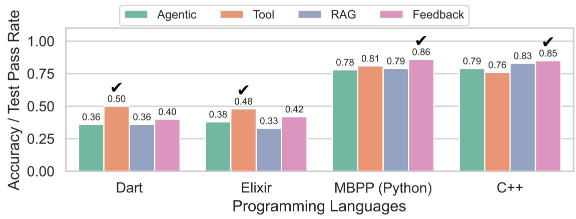

Figure 3: Performance across various adaptation methods for high and low resource programming languages using GPT-4o.

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.

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.

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

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*.

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*.

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*.

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*.

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

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*.

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*.

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.

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.

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*.

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.