Evaluating Tokenizer Adaptation Methods for Large Language Models on Low-Resource Programming Languages

Georgii Andriushchenko
Innopolis University
georgyandryuschenko@gmail.com

Vladimir Ivanov
Innopolis University
v.ivanov@innopolis.ru

Abstract

Large language models (LLMs), which are primarily trained on high-resource programming languages (HRPLs), tend to perform suboptimally for low-resource programming languages (LRPLs). This study investigates the impact of tokenizer adaptation methods on improving code generation for LRPLs. StarCoder 2 and DeepSeek-Coder models adapted to Elixir and Racket using methods such as Fast Vocabulary Transfer (FVT), FOCUS, and Zero-shot Tokenizer Transfer (ZeTT) are evaluated and compared with the original and fine-tuned models. Our experiments reveal that ZeTT outperforms other methods, achieving significant improvements in handling syntax, program logic, and data types for LRPLs. However, we also highlight performance declines in non-target languages like Python after tokenizer adaptation. The study approves the positive impact of tokenizer adaptation in enhancing LRPL code generation and suggests directions for future research, including token embeddings improvement. The code for experiments reproduction is available in the GitHub repository[1].

1 Introduction

Previous studies showed that large language models trained on source code (Code LLMs) excel at generating code (Zheng et al., 2023) in high-resource programming languages (HRPLs) (Lozhkov et al., 2024; Cassano et al., 2024; Chen et al., 2022) from docstrings. However, Code LLMs demonstrate suboptimal code generation performance on low-resource programming languages (LRPLs) (Cassano et al., 2024, 2022; Chai et al., 2024; Yan et al., 2024). This disparity in performance puts LRPLs at a potential risk of becoming extinct without adequate support from LLMs because programmers often use LLMs to accelerate their work. Previous work attempted to address this issue via continued training (Cassano et al., 2024, 2022), but the performance gap of Code LLMs on LRPLs could also be caused by underfit Code LLM tokenizers doing ineffective tokenization (Dagan et al., 2024). This study provides a comprehensive evaluation of the code generation capabilities of the Code LLMs adapted to LRPLs using various tokenizer adaptation methods. We highlight the challenges of the LRPL code generation improvement with tokenizer adaptation methods. Based on the experimental results, we also demonstrate that better performance on LRPL code generation could be achieved with the Zero-shot Tokenizer Transfer (ZeTT) (Minixhofer et al., 2024) method.

Thus, the study makes the following contributions:

1. Evaluates code generation performance of popular open-source Code LLMs on LRPLs and an HRPL.
2. Adapts Code LLMs to LRPLs using various tokenizer adaptation methods.
3. Compares code generation performance of original Code LLMs and their adaptations on LRPLs.

2 Related Works

2.1 Continued Training

In their work, (Cassano et al., 2024) rightly observed that Code LLMs demonstrate sub-optimal performance on LRPLs such as Julia, Lua, OCaml, R, and Racket due to the lack of training source code written in these languages. To address this problem, they composed semi-synthetic training data by using an LLM to translate Python code to LRPL code. The authors also proposed another approach to obtain LRPL code in their previous study

[1] https://github.com/datapaf/LRPLTokenizerAdaptations

Tokenizer Name	Vocab. Size	New Tokens	Keywords	
			Racket	*Elixir*
StarCoder 2	49 152	-	26%	70%
StarCoder 2 Racket	53 340	4 188 +9%	31% +5%	74% +4%
StarCoder 2 Elixir	52 202	3 050 +6%	27% +1%	82% +12%
DeepSeek-Coder	32 022	-	22%	64%
DeepSeek-Coder Racket	39 883	7 861 +25%	31% +9%	74% +10%
DeepSeek-Coder Elixir	38 981	6 959 +21%	25% +3%	82% +18%

Table 1: Statistics of the original and adapted tokenizers. The original tokenizers are highlighted in bold. The vocabulary expansion percentage and the keywords increase percentage are highlighted in green.

(Cassano et al., 2022), which involves translation using a set of compilers. However, this approach was used only to create a code generation benchmark comprising 18 LRPLs.

2.2 Tokenizer Adaptation

Tokenizer adaptation involves changing the tokenizer of the model to a new tokenizer that contains more tokens from the target language to create a better representation of the language (Csaki et al., 2023). (Mosin et al., 2023) proposed a simple tokenizer adaptation approach that reuses the embeddings of the original model. The implementation of this approach was optimized by (Gee et al., 2022) in their Fast Vocabulary Transfer (FVT) approach. FOCUS (Dobler and De Melo, 2023) has recently overcome the performance of WECHSEL (Minixhofer et al., 2022) and RAMEN (Tran, 2020) on multilingual XNLI (Conneau et al., 2018) and QuAD (Möller et al., 2021) tasks, making an advancement in tokenizer adaptation. The authors of Zero-shot Tokenizer Transfer (ZeTT) (Minixhofer et al., 2024) proposed to train a Transformer (Vaswani et al., 2017) encoder as a hypernetwork to produce embeddings for the tokens of the new tokenizer. Currently, this is a state-of-the-art tokenizer adaptation method that overcomes the previous cutting-edge methods FOCUS and OFA (Liu et al., 2024) on natural language and code tasks.

3 Experimental Setup

3.1 Motivation for Tokenizer Adaptation

It was previously demonstrated that a model with a tokenizer containing more target language tokens has improved text understanding and produces a text with higher quality (Mosin et al., 2023; Gee et al., 2022; Dobler and De Melo, 2023; Minixhofer et al., 2024). This may be a premise that tokenizer adaptation could boost the quality of LRPL code generation for Code LLMs since the structures of code and natural language are similar (Allamanis et al., 2018). The similarity is also approved by the fact that models originally developed for natural language were effective for source code (Hindle et al., 2016).

3.2 Programming Languages

To assess the effect of tokenizer adaptation on the quality of generated LRPL code, we consider *Elixir* and *Racket* LRPLs. The motivation for the choice is provided in Appendix A. It also makes sense to check whether the adapted models retain their capabilities of generating code in HRPLs. Thus, we considered *Python* programming language as an HRPL since it is a popular and widely used PL according to the Stack Overflow survey[2]. This is approved by the Stack v2 (Lozhkov et al., 2024) statistics: Python is in the top 10 of PLs by the number of bytes in the dataset.

3.3 Code LLMs (Baselines)

Tokenizer adaptation experiments are performed on *StarCoder 2* (Lozhkov et al., 2024) with 3 billion parameters and *DeepSeek-Coder* (Guo et al., 2024) with 1.3 billion parameters. Appendix B contains the discussion of the model choice.

3.4 Training Data

There is an obvious lack of publicly available and high-quality datasets with the code written in LRPLs. Due to this natural reason, the training of tokenizers and models is performed on the data from the Stack v2 (Lozhkov et al., 2024) dataset[3].

[2] https://survey.stackoverflow.co/2024/technology

[3] The dataset contains the code whose licenses are considered permissive by the authors. List of license identifiers: https://huggingface.co/datasets/bigcode/the-stack-v2/blob/main/license_stats.csv

Model Name	Adaptation to Racket			Adaptation to Elixir		
	Racket	*Elixir*	*Python*	*Racket*	*Elixir*	*Python*
starcoder2-3b	8	**20**	**24**	**8**	20	**24**
+ FT	**30**	4	12	0	28	8
+ FVT	28	2	10	0	**30**	0
+ FOCUS	24	0	6	0	28	0
deepseek-coder-1.3b-base	12	**38**	**30**	12	38	**30**
+ FT	26	24	**30**	8	28	28
+ FVT	18	16	22	10	26	22
+ FOCUS	24	0	6	0	28	0
+ ZeTT Adapted Tokenizer	**28**	16	18	**18**	**32**	28
+ ZeTT Original Tokenizer	26	20	22	10	30	22

Table 2: Pass@1 (%) values on McEval benchmark for the original models and the adapted models using various tokenizer adaptation methods. The names of the adaptation methods are provided after the "+" sign. "FT" abbreviation stands for the fine-tuned model. Note that the StarCoder 2 model does not have a ZeTT-adapted version since HF Transformers does not support conversion of this model to a Flax model.

It contains the subsets containing code for the selected LRPLs with 227 thousand Racket source code files and 1.8 million Elixir source code files.

3.5 Adaptation to LRPLs

3.5.1 Fine-tuning

To check that tokenizer adaptation provides an improvement, we fine-tuned the models on the LRPLs to check whether tokenizer adaptation indeed provides an improvement. StarCoder 2 and DeepSeek-Coder were both fine-tuned on the LRPL source code taken from the Stack v2 dataset. Even though Racket and Elixir are subsets of the Stack v2 differ in size, we trained the models on the same amount of source code files. Appendix E provides the fine-tuning details.

3.5.2 Tokenizer Adaptation

In this study, we adapted the models using several tokenizer adaptation methods:

1. Fast Vocabulary Transfer (FVT) (Gee et al., 2022)
2. FOCUS (Dobler and De Melo, 2023)
3. Zero-shot Tokenizer Transfer (ZeTT) (Minixhofer et al., 2024)

The details of the methods are provided in Appendix F, Appendix G, and Appendix H. Note that the embeddings initialization, involved in tokenizer adaptation, was performed for both the input and output embeddings. After the initialization, the model with the adapted tokenizer is fine-tuned on the LRPL source code according to Appendix E. The details of the entire pipeline are provided in Appendix C.

3.6 Adapted Tokenizers

We adapted tokenizers to LRPLs using vocabulary expansion: tokens of an auxiliary tokenizer trained on LRPL code are added to the model tokenizer. In our experiments, we trained auxiliary tokenizers with a vocabulary size of 1/3 of the model tokenizer's vocabulary size. However, the actual amount of added tokens will be lower since model and auxiliary tokenizers often have overlapping tokens. The adapted tokenizers are summarized in Table 1. More details are presented in Appendix D.

3.7 Code Generation Benchmarks

We assessed the quality of code generation on several benchmarks.:

1. MultiPL-E (Cassano et al., 2022)
2. McEval (Chai et al., 2024)

Detailed descriptions of the benchmarks are provided in Appendix I.

4 Evaluation Results and Discussion

4.1 Effect of Vocabulary Expansion on Tokenization

The results of the analysis of the adapted tokenizers in Appendix D demonstrate that tokenizers now use new, larger tokens when tokenizing code in the target LRPL. In the case of DeepSeek-Coder, there is a statistically significant ($\leq 5\%$) decrease in the mean tokens per text (MTPT) and the mean

bytes per token (MBPT). However, in the case of StarCoder, the situation is controversial since the decrease in MTPT happens to be not statistically significant. The reason for that could be the fact that the tokenizer vocabulary of StarCoder 2 was expanded by less than 10%, which could be insufficient. Despite that, the tokenizers consistently use 50-60% of the added tokens. These added tokens are indeed significant for the target LRPLs since they add up to 9% of Racket keywords and up to 18% of Elixir keywords.

4.2 Comparison of Tokenizer Adaptation Methods on Target LRPLs

The results of the evaluation of original and adapted models on the MultiPL-E benchmark are presented in Appendix J. Evaluation results on the McEval benchmark may be seen in Table 2. These evaluation results are used to compare tokenizer adaptation methods.

4.2.1 Racket

FVT and FOCUS improve the performance of the base models but do not achieve the performance of the fine-tuned model. ZeTT versions demonstrate promising results, often overcoming the fine-tuned model on HumanEval (15.99%) and McEval (28%) benchmarks.

4.2.2 Elixir

As in the Racket case, FVT and FOCUS often fail to achieve the code generation abilities of the fine-tuned model. At the same time, ZeTT-variants, especially with adapted tokenizer, are highly effective for Elixir. ZeTT with adapted tokenizer achieves 17.79% on HumanEval and 22.36% on MBPP, outperforming FT. ZeTT with the original tokenizer leads in MBPP (24.66%).

4.3 Performance of Adapted Models on Non-target PLs

Python performance consistently declines in almost all cases, except for a single case during McEval evaluation. Most Racket-adapted models show reduced Elixir performance on McEval. However, there are cases when fine-tuning DeepSeek-Coder on Racket improves the model performance on Elixir MultiPL-E tasks from 4.11% up to 17.68%, which could be a sign of cross-lingual transferability. We noticed that DeepSeek-Coder fine-tuned on Racket code used some of the classic idioms when generating Elixir code, which might positively affect pass@1 values. For example, DeepSeek-Coder trained on Racket code uses explicit pattern-matching recursion with an accumulator that is common in functional languages like Racket to solve a task of list processing. Unlike this, the original DeepSeek-Coder uses the built-in `Enum.map/2`.

Similar severe performance declines may be observed in the Racket performance of Elixir-adapted models. The declines could be the sign of catastrophic forgetting (French, 1999; Muennighoff et al., 2023; Vu et al., 2022). After fine-tuning a model on some target PL, the model fails to generate code in a non-target PL. For example, DeepSeek-Coder fine-tuned on Racket struggles with basic Python tasks it previously handled. The common mistakes of a fine-tuned DeepSeek-Coder are incomplete implementations, logical errors, and omitted imports.

4.4 Vocabulary Expansion Importance in ZeTT

To check the effect of vocabulary expansion in ZeTT adaptations, we performed experiments with both ZeTT-adapted models featuring original and adapted tokenizers. The experimental results demonstrate that even though the ZeTT-adapted model with the adapted tokenizer often shows better performance, the model with the original tokenizer has a comparable performance as well. This may indicate that the quality of token embeddings, and their semantic content, could be no less impactful than the token length. Cross-lingual knowledge, provided by CodeBERT, may enrich the token embeddings with valuable cross-lingual knowledge. Thus, the improvement of LRPL tokens' embeddings with cross-lingual knowledge could be a promising future work.

4.5 ZeTT Improvements in Target LRPLs

Compared to the fine-tuned models, ZeTT models obtain the following improvements. For Elixir, the ZeTT model works correctly with function argument passing, array manipulation, recursive logic, indices handling, operators, and data types. For Racket, the issues related to recursive functions, base cases, and built-in and helper functions are resolved.

We hypothesize that ZeTT might be effective largely due to its hypernetwork-based approach to embedding prediction and the transfer of knowledge from CodeBERT. ZeTT predicts embeddings for new tokens using the CodeBERT hypernet-

work rather than relying on heuristics. CodeBERT is a Transformer encoder pre-trained on several HRPLs. This allows ZeTT to analyze the constituents of new tokens, incorporate relevant prior knowledge, and generate semantically rich embeddings. An example could be the transfer of knowledge about recursion, lambda functions, closures, and functional programming patterns from HRPLs like JavaScript, Python, and Ruby. As a result, ZeTT improves LLM's abilities to handle better Racket and Elixir constructs such as function argument passing, helper functions, array manipulation, base cases, recursion, indices, operators, data types.

5 Conclusion

The study provides a comprehensive evaluation of code generation capabilities in low-resource programming languages (LRPLs), revealing the suboptimal performance of current popular Code LLMs without tokenizer adaptation. Among the tested tokenizer adaptation methods, ZeTT is the most effective approach that outperforms FVT and FOCUS in handling syntax, program logic, operators, and data types. The findings highlight the critical role of tokenizers and token embeddings in LRPL code generation. The obtained results could be helpful in further research of Code LLMs' performance in LRPL code generation.

Limitations

Despite that the study provides valuable insights into the improvement of code generation abilities of Code LLM in LRPLs, the study has several limitations that could potentially influence the conclusions:

- The study considers only 2 LRPLs and a single HRPL;
- We used relatively small Code LLMs of 1-3 billion parameters in the experiments;
- We noticed that tokenizer adaptation methods are sensitive to how the embeddings are trained after initialization;
- The fine-tuning strategy that we applied in all adaptation methods may not be optimal for each method;
- Peformance of a ZeTT-adapted model depends on the choice of hypernetwork.

Additionally, we evaluated only pass@1 metric to assess the functional correctness of generated code, but no other code quality aspects such as efficiency, readability, and idiomatic style were considered.

Finally, the following applicability limitations of the study may be observed. Due to the HRPL performance decline, the adapted models have limited applicability in multilingual settings. Also, tokenizer adaptation is relatively complex, which may limit usability in industrial tasks.

Ethical Considerations

For training and evaluation purposes, we used publicly available data and code. The Stack v2 used for models training contains the code whose licenses are considered permissive by the dataset authors. Particular license identifiers are provided on the official HuggingFace page[4] of the dataset. Evaluation data for MultiPL-E and McEval are composed by their authors and are publicly available.

Acknowledgements

This work was supported by the Ministry of Economic Development of the Russian Federation (agreement No. 139-10-2025-034 dd. 19.06.2025, IGK 000000C313925P4D0002).

We also appreciate the valuable comments provided by the anonymous reviewers.

References

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *Transactions of the association for computational linguistics*, 5:135–146.

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

[4] https://huggingface.co/datasets/bigcode/the-stack-v2/blob/main/license_stats.csv

programming languages for code llms. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):677–708.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, and 1 others. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*.

Linzheng Chai, Shukai Liu, Jian Yang, Yuwei Yin, Ke Jin, Jiaheng Liu, Tao Sun, Ge Zhang, Changyu Ren, Hongcheng Guo, and 1 others. 2024. Mceval: Massively multilingual code evaluation. *arXiv preprint arXiv:2406.07436*.

Fuxiang Chen, Fatemeh H Fard, David Lo, and Timofey Bryksin. 2022. On the transferability of pre-trained language models for low-resource programming languages. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 401–412.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Alexis Conneau, Ruty Rinott, Guillaume Lample, Adina Williams, Samuel Bowman, Holger Schwenk, and Veselin Stoyanov. 2018. XNLI: Evaluating cross-lingual sentence representations. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2475–2485, Brussels, Belgium. Association for Computational Linguistics.

Zoltan Csaki, Pian Pawakapan, Urmish Thakker, and Qiantong Xu. 2023. Efficiently adapting pretrained language models to new languages. *37th Conference on Neural Information Processing Systems (NeurIPS 2023)*.

Gautier Dagan, Gabriel Synnaeve, and Baptiste Rozière. 2024. Getting the most out of your tokenizer for pre-training and domain adaptation. In *Proceedings of the 41st International Conference on Machine Learning*, pages 9784–9805.

Konstantin Dobler and Gerard De Melo. 2023. Focus: Effective embedding initialization for monolingual specialization of multilingual models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13440–13454.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and 1 others. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547.

Robert M French. 1999. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135.

Leonidas Gee, Andrea Zugarini, Leonardo Rigutini, and Paolo Torroni. 2022. Fast vocabulary transfer for language model compression. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 409–416.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Communications of the ACM*, 59(5):122–131.

Daniel Jurafsky. 2000. Speech and language processing.

Yihong Liu, Peiqin Lin, Mingyang Wang, and Hinrich Schütze. 2024. Ofa: A framework of initializing unseen subword embeddings for efficient large-scale multilingual continued pretraining. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 1067–1097.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Andre Martins and Ramon Astudillo. 2016. From softmax to sparsemax: A sparse model of attention and multi-label classification. In *International conference on machine learning*, pages 1614–1623. PMLR.

Benjamin Minixhofer, Fabian Paischer, and Navid Rekabsaz. 2022. Wechsel: Effective initialization of subword embeddings for cross-lingual transfer of monolingual language models. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 3992–4006.

Benjamin Minixhofer, Edoardo Maria Ponti, and Ivan Vulić. 2024. Zero-shot tokenizer transfer. *arXiv preprint arXiv:2405.07883*.

Timo Möller, Julian Risch, and Malte Pietsch. 2021. GermanQuAD and GermanDPR: Improving non-English question answering and passage retrieval. In *Proceedings of the 3rd Workshop on Machine Reading for Question Answering*, pages 42–50, Punta Cana, Dominican Republic. Association for Computational Linguistics.

Vladislav Mosin, Igor Samenko, Borislav Kozlovskii, Alexey Tikhonov, and Ivan P Yamshchikov. 2023. Fine-tuning transformers: Vocabulary transfer. *Artificial Intelligence*, 317:103860.

Niklas Muennighoff, Thomas Wang, Lintang Sutawika, Adam Roberts, Stella Biderman, Teven Le Scao, M Saiful Bari, Sheng Shen, Zheng Xin Yong, Hailey Schoelkopf, and 1 others. 2023. Crosslingual generalization through multitask finetuning. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15991–16111.

Yuval Pinter, Robert Guthrie, and Jacob Eisenstein. 2017. Mimicking word embeddings using subword rnns. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 102–112.

Ke Tran. 2020. From english to foreign languages: Transferring pre-trained language models. *arXiv preprint arXiv:2002.07306*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Tu Vu, Aditya Barua, Brian Lester, Daniel Cer, Mohit Iyyer, and Noah Constant. 2022. Overcoming catastrophic forgetting in zero-shot cross-lingual generation. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 9279–9300.

Weixiang Yan, Haitian Liu, Yunkun Wang, Yunzhe Li, Qian Chen, Wen Wang, Tingyu Lin, Weishan Zhao, Li Zhu, Hari Sundaram, and 1 others. 2024. Codescope: An execution-based multilingual multitask multidimensional benchmark for evaluating llms on code understanding and generation. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5511–5558.

Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372*.

A Choice of LRPLs

The choice of LRPLs on the distribution of source code bytes over PLs in the deduplicated Stack v2 dataset[5]. We considered programming languages that overcome the 99% quantile to be low-resource. In total, according to our approach, 512 languages are considered low-resource, which is 82% of the languages presented in the dataset. *Elixir* and *Racket* PLs were chosen for experiments since they are presented in both code generation benchmarks, MultiPL-E (Cassano et al., 2022) and McEval (Chai et al., 2024).

[5] https://huggingface.co/datasets/bigcode/the-stack-v2-dedup

B Choice of Code LLMs

Tokenizer adaptation experiments are performed on *StarCoder 2* (Lozhkov et al., 2024) with 3 billion parameters and *DeepSeek-Coder* (Guo et al., 2024) with 1.3 billion parameters. These are the modern and popular open-source Code LLMs having the smallest amount of parameters to save computational resources and time when performing experiments. Even though these models have the smallest number of parameters, they are good enough to generate working code in various PLs. Adapting the tokenizer of the two different Code LLMs is useful to determine whether the approach is generalizable over model architectures. Additionally, these models are comparable since they have a relatively close number of parameters. The models do not differ much in their complexity and, therefore, in their abilities. One may correctly notice that Starcoder 2 has more than 2 times as many parameters as DeepSeek, so their abilities should differ significantly. However, those are the smallest models that are maximally close to each other in terms of a number of parameters.

C Tokenizer Adaptation Pipeline

The pipeline consists of the following three steps.

1. An LRPL-specific tokenizer is created using the vocabulary expansion approach;
2. The embeddings of the new tokens are initialized according to the tokenizer adaptation method;
3. The entire LLM is fine-tuned on the target LRPL.

Figure 1 provides a visualized summary of the pipeline.

D Adapted Tokenizers

The summary of the adapted tokenizers is provided in Table 1. We define **keywords** as the special words reserved by a programming language. The list of keywords was collected from the grammars of the Visual Studio Code[6] language servers for Racket[7] and Elixir[8]. In total, we collected 122 keywords for Racket and 50 keywords for Elixir. The keywords percentage for the tokenizers is the

[6] https://code.visualstudio.com/
[7] https://github.com/Eugleo/magic-racket/
[8] https://github.com/timmhirsens/vscode-elixir

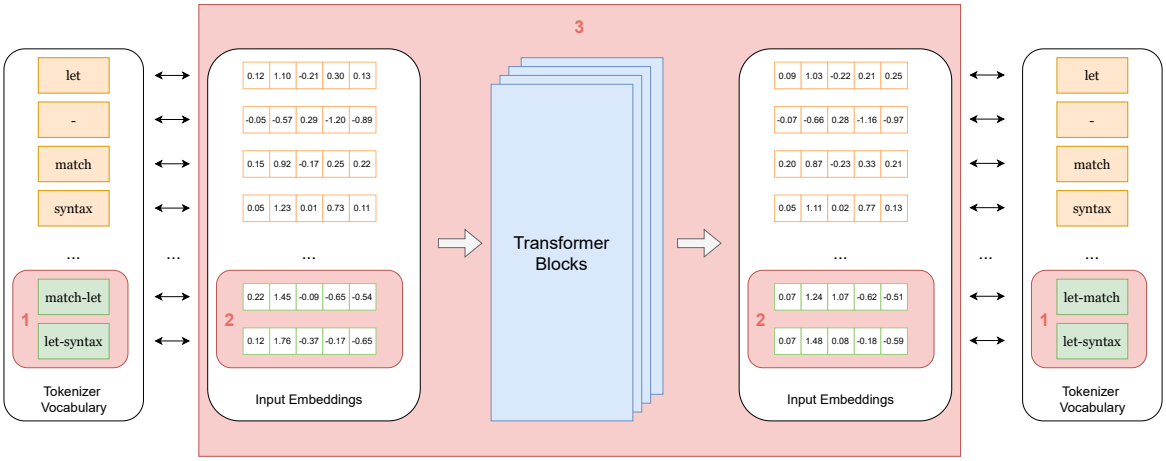

Figure 1: Schematic overview of the LLM tokenizer adaptation pipeline visualized on LLM components. The components affected by the steps of the adaptation are highlighted with numbered red rectangles. The number of a red rectangle indicates the number of the adaptation step. The step **(1)** involves adding new tokens (green rectangles on the diagram) to the original tokenizer. Note that the tokenizer vocabularies from the left and the right are the same. The step **(2)** requires initialization of the corresponding input and output embeddings (green cells on the diagram) to the newly added tokens. Input and output embeddings may be different for the same tokens. In the step **(3)**, the entire model is trained on LRPL code.

ratio of the keywords present in the tokenizers' vocabulary over the total number of keywords.

To check whether vocabulary expansion makes a difference in tokenization, we calculated the mean number of tokens per text (Table 7) and the mean number of bytes per token (Table 7). Vocabulary usage (Table 6) was calculated to check how many of the added tokens were used in total.

E Fine-tuning Parameters

Fine-tuning is the step that follows after the embeddings initialization in each tokenizer adaptation method. To provide a fair comparison, we performed fine-tuning with the same training parameters for each method. We optimize all model parameters during fine-tuning. The fine-tuning was performed using TRL[9] SFTTrainer on 224000 code samples with the following training parameters:

- Maximal Gradient Norm: 1
- Batch Size: 4
- Warmup Ratio: 0.25
- Training Epochs: 1
- Learning Rate: 5e-5
- Scheduler: cosine
- Weight Decay: 1

[9] https://huggingface.co/docs/trl/en/index

F FVT Adaptation Details

The approach proposes to initialize the embeddings for the new tokens using the embeddings of the original model. To do that, the new token is split into constituent tokens using the original tokenizer of the model. Next, the embeddings of the constituent tokens are averaged to obtain a single average embedding:

$$E_{\text{new}}(t_i) = \frac{1}{|\mathcal{T}_a(t_i)|} \sum_{t_j \in \mathcal{T}_a(t_i)} E_{\text{old}}(t_j) \quad (1)$$

where $E_{\text{new}}, E_{\text{old}}$ - embeddings of the adapted and original model correspondingly; t_i, t_j - added token and constituent token respectively; $\mathcal{T}_a$ - original tokenizer. Note that with this approach, the embeddings of the old tokens are preserved.

G FOCUS Adaptation Details

The method firstly trains fastText (Bojanowski et al., 2017) embeddings for all the tokens of the new tokenizer. Then, each new token gets an embedding initialized with the weighted average of the model embeddings of all the old tokens.

$$E_{\text{new}}(t_i) = \frac{1}{|\mathcal{V}_{\mathcal{T}_a}|} \sum_{t_j \in \mathcal{V}_{\mathcal{T}_a}} w_{t_j} E_{\text{old}}(t_j) \quad (2)$$

where $\mathcal{V}_{\mathcal{T}_a}$ - vocabulary of the original tokenizer; w_{t_j} - weight of a token. The weights are deter-

mined by the cosine similarity between the fastText embedding of the target token and the fastText embedding of an old token. Irrelevant embeddings are excluded from the averaging using sparsemax (Martins and Astudillo, 2016)

In our experiments, we used the implementation[10] of the method provided by the method's authors. The fastText embeddings were trained with the default training parameters, provided in the FOCUS implementation.

H ZeTT Adaptation Details

The method approaches embedding initialization in a conceptually new way: it uses a Transformer Encoder (Vaswani et al., 2017) hypernetwork $H_\theta : \mathcal{T}_b \to \phi_b$, to predict the embeddings ϕ_b of the tokens in the vocabulary of the adapted tokenizer $\mathcal{T}_b$. During the training, the hypernetwork should first pass the MIMIC-style (Pinter et al., 2017) warmup stage. After that, the hypernetwork parameters θ are trained on the following loss:

$$\mathcal{L}_\theta^{\text{final}} = \mathcal{L}_\theta(\mathcal{T}_b, H_\theta(\mathcal{T}_b), \psi) + \alpha \cdot \mathcal{L}_\theta^{\text{aux}} \quad (3)$$

where $\mathcal{L}_\theta$ is a CLM (Jurafsky, 2000) objective, ψ are the language model (non-embedding) parameters, and α is a weight of the auxiliary loss that is defined as

$$\mathcal{L}_\theta^{\text{aux}} = \frac{\sum_t \|H_\theta[\mathcal{V}_{\mathcal{T}_b}[t]] - \phi_a[\mathcal{V}_{\mathcal{T}_a}[t]]\|_2}{|\mathcal{V}_{\mathcal{T}_a} \cap \mathcal{V}_{\mathcal{T}_b}|} \quad (4)$$

where $t \in |\mathcal{V}_a \cap \mathcal{V}_b|$. Meanwhile, the language model parameters ψ are not trained during the hypernetwork training.

In our experiments, we used the implementation[11] of the method authors to train a CodeBERT (Feng et al., 2020) hypernetwork with the following training parameters:

- `loss`: clm
- `n_embd`: 2048
- `n_token_subsample`: 8192
- `identity_n_subsample`: 8192
- `identity_steps`: 14000
- `warmup_steps`: [14000, 15000]
- `steps`: 56000
- `learning_rate`: [3e-4, 6e-5]
- `max_grad_norm`: 0.1
- `hn_surface_maxlen`: 7
- `weight_decay`: 0.01
- `train_batch_size`: 2
- `hn_hidden_size`: 2048
- `hn_intermediate_size`: 4096
- `lexical_loss_weight`: 32

For interpretation of the training parameters, please refer to the documentation of the original ZeTT implementation.

I Code Generation Benchmarks

MultiPL-E (Cassano et al., 2022). The benchmark includes the tasks from HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) datasets translated to other PLs. Due to the large amount of experiments, we only evaluated pass@1 metric for 50 samples per task with 0.2 temperature on both datasets.

McEval (Chai et al., 2024). The benchmark provides a set of custom-curated tasks. It contains 50 tasks and tests for each PL from the vast set. The benchmarks only evaluate pass@1 over a set of tasks since it requires the models to greedily generate the code.

J MultiPL-E Evaluation Results

The original and adapted models are evaluated on both datasets of the MultiPL-E benchmark: HumanEval and MBPP. Table 3 presents pass@1 metrics for models adapted to Racket, while Table 4 shows the metrics for Elixir-adapted models.

[10] https://github.com/konstantinjdobler/focus
[11] https://github.com/bminixhofer/zett

Model Name	HumanEval			MBPP		
	Racket	*Elixir*	*Python*	*Racket*	*Elixir*	*Python*
starcoder2-3b	8.21	**9.28**	**30.43**	14.72	**6.87**	**41.98**
+ FT	**15.25**	0.00	16.43	22.88	0.00	12.85
+ FVT	13.42	0.00	15.71	23.89	0.04	11.64
+ FOCUS	13.66	0.30	11.88	**24.28**	0.48	6.84
deepseek-coder-1.3b-base	9.75	15.01	**31.77**	17.69	4.11	**43.36**
+ FT	14.15	**16.07**	29.20	**23.45**	**17.68**	41.86
+ FVT	10.14	12.15	25.32	10.34	12.32	36.41
+ FOCUS	9.98	0.00	0.00	10.50	0.85	3.47
+ ZeTT Adapted Tokenizer	14.73	8.26	28.33	22.18	8.09	36.75
+ ZeTT Original Tokenizer	**15.99**	9.06	26.84	21.98	12.30	40.01

Table 3: Pass@1 (%) values on MultiPL-E benchmark for the original models and the models adapted to Racket using various tokenizer adaptation methods. The names of the adaptation methods are provided after the "+" sign. "FT" abbreviation stands for the fine-tuned model. Note that the StarCoder 2 model does not have a ZeTT-adapted version since HF Transformers does not support converting this model to a Flax model.

Model Name	HumanEval			MBPP		
	Racket	*Elixir*	*Python*	*Racket*	*Elixir*	*Python*
starcoder2-3b	**8.21**	9.28	**30.43**	**14.72**	6.87	**41.98**
+ FT	0.00	**16.10**	4.26	1.25	**10.47**	0.19
+ FVT	0.60	15.22	2.77	0.47	8.85	0.02
+ FOCUS	0.05	15.84	2.44	0.13	8.27	0.00
deepseek-coder-1.3b-base	**9.75**	15.01	**31.77**	**17.69**	4.11	**43.36**
+ FT	8.56	16.68	25.73	15.98	6.70	25.73
+ FVT	5.03	12.93	18.70	9.77	16.59	27.64
+ FOCUS	0.73	12.76	0.00	1.00	10.33	0.58
+ ZeTT Adapted Tokenizer	5.96	**17.79**	24.74	8.39	22.36	4.94
+ ZeTT Original Tokenizer	6.32	16.58	24.00	10.17	**24.66**	16.98

Table 4: Pass@1 (%) values on MultiPL-E benchmark for the original models and the models adapted to Elixir using various tokenizer adaptation methods. The names of the adaptation methods are provided after the "+" sign. "FT" abbreviation stands for the fine-tuned model. Note that the StarCoder 2 model does not have a ZeTT-adapted version since HF Transformers does not support conversion of this model to a Flax model

Tokenizer Name	Racket			Elixir		
	Mean	*Std*	*p-value*	*Mean*	*Std*	*p-value*
StarCoder 2	918	1350	-	557	903	-
StarCoder 2 Racket	900	1320	0.3349	557	902	1.0000
StarCoder 2 Elixir	918	1349	1.0000	545	885	0.3426
DeepSeek-Coder	1044	1497	-	655	1031	-
DeepSeek-Coder Racket	987	1412	0.0056	647	1020	0.5812
DeepSeek-Coder Elixir	1027	1473	0.4183	617	970	0.0073

Table 5: Mean tokens per text (MTPT) for the original and adapted tokenizers calculated for 10 000 samples. Mean and standard deviation values are rounded to the nearest integer. The original tokenizers are highlighted in bold. P-values of the two-tailed t-test between MTPTs of the original and adapted tokenizers are indicated in the dedicated column. Statistically significant differences (p-value < 5%) are highlighted in green, while the others are highlighted in red.

Tokenizer Name	Racket				Elixir			
	Used		**Unused**		**Used**		**Unused**	
	Total	*Added*	*Total*	*Added*	*Total*	*Added*	*Total*	*Added*
StarCoder 2	91	-	9	-	95	-	5	-
StarCoder 2 Racket	89	64	11	36	92	64	8	36
StarCoder 2 Elixir	88	41	12	59	93	41	7	59
DeepSeek-Coder	93	-	7	-	93	-	7	-
DeepSeek Racket	86	59	14	41	86	59	14	41
DeepSeek Elixir	86	53	14	47	88	53	12	47

Table 6: Vocabulary usage (%) by the original and adapted tokenizers. The original tokenizers are highlighted in bold. The "Used" group of columns indicates the percentage of all added tokens used in the tokenization of a training dataset. The "Unused" group of columns is similar to the "Used" group but indicates tokens that were not used in tokenization.

Tokenizer Name	Racket			Elixir		
	Mean	*Std*	*p-value*	*Mean*	*Std*	*p-value*
StarCoder 2	2.8861	5.2765	-	3.9213	3.6107	-
StarCoder 2 Racket	2.9331	5.6742	0.0140	3.9251	3.6258	0.3525
StarCoder 2 Elixir	2.8876	5.2781	1.0000	4.0061	3.6760	0.0001
DeepSeek-Coder	2.6686	4.4462	-	3.3679	3.2266	-
DeepSeek-Coder Racket	2.7986	4.8579	0.0001	3.4116	3.2680	0.0001
DeepSeek-Coder Elixir	2.7082	4.4792	0.0026	3.5727	3.3596	0.0001

Table 7: Mean bytes per token (MBPT) for the original and adapted tokenized calculated over training datasets. The original tokenizers are highlighted in bold. P-values of the two-tailed t-test between MBPTs of the original and adapted tokenizers are indicated in the dedicated column. Statistically significant differences (p-value < 5%) are highlighted in green, while the others are highlighted in red.