PERC: Plan-As-Query Example Retrieval for Underrepresented Code Generation

Jaeseok Yoo[1] **Hojae Han**[1] **Youngwon Lee**[1] **Jaejin Kim**[12] **Seung-won Hwang**[12]*

[1]Seoul National University

[2]Interdisciplinary Program in Artificial Intelligence, Seoul National University

{jaeseok2.yoo,stovecat,ludaya,jaejin.kim,seungwonh}@snu.ac.kr

Abstract

Code generation with large language models has shown significant promise, especially when employing retrieval-augmented generation (RAG) with few-shot examples. However, selecting effective examples that enhance generation quality remains a challenging task, particularly when the target programming language (PL) is underrepresented. In this study, we present two key findings: (1) retrieving examples whose presented algorithmic plans can be referenced for generating the desired behavior significantly improves generation accuracy, and (2) converting code into pseudocode effectively captures such algorithmic plans, enhancing retrieval quality even when the source and the target PLs are different. Based on these findings, we propose Plan-as-query Example Retrieval for few-shot prompting in Code generation (PERC), a novel framework that utilizes algorithmic plans to identify and retrieve effective examples. We validate the effectiveness of PERC through extensive experiments on the CodeContests, HumanEval and MultiPL-E benchmarks: PERC consistently outperforms the state-of-the-art RAG methods in code generation, both when the source and target programming languages match or differ, highlighting its adaptability and robustness in diverse coding environments.

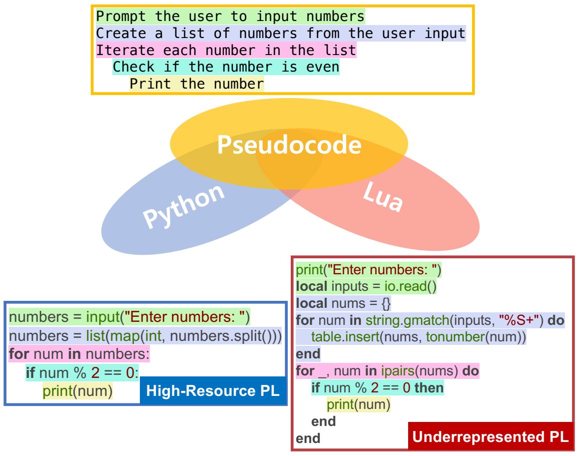

Figure 1: Two Python and Lua code snippets have different modalities but implement the same algorithmic plans. PERC uses pseudocode describing the algorithmic plans to minimize noise from modality differences. Colors in code represent equivalent steps.

1 Introduction

Code generation using large language models (LLMs) has shown significant potential, particularly when retrieval-augmented generation (RAG) with few-shot examples is employed (Parvez et al., 2021; Nashid et al., 2023; Zhang et al., 2023). However, selecting effective examples to improve code generation quality remains a challenging task. This is even more difficult when the target programming language (PL) is underrepresented, as the construction of the few-shot example pool for retrieval is non-trivial.

To construct the retrieval pool for an underrepresented PL, we can transfer the retrieval pool from a high-resource PL. However, this in turn interferes with the state-of-the-art few-shot prompting approaches in code generation (Nashid et al., 2023; Zhang et al., 2023), which employ code to retrieve examples. Figure 1 illustrates such an example; although the Python code on the lower left and the Lua code on the lower right follow the same algorithmic steps, both lexical-based (Robertson and Zaragoza, 2009) and embedding-based (Song et al., 2020) retrieval fall short in capturing their algorithmic similarity due to syntactic and structural difference (An et al., 2023).

To overcome this, we propose Plan-as-query Example Retrieval for few-shot prompting in Code generation (PERC). PERC leverages algorithmic plans such as pseudocode to retrieve examples.

*Corresponding author.

Proceedings of the 31st International Conference on Computational Linguistics, pages 7982–7997
January 19–24, 2025.

Converting code to pseudocode reduces syntactic noise and captures algorithmic similarity, enhancing retrieval quality across programming languages. Also, such plans can aid generation by participating in reasoning chains, further improving the generation accuracy.

We demonstrate PERC's effectiveness in two key scenarios: First, plan-based example selection improves code generation accuracy for same-language tasks on CodeContests (Li et al., 2022) and HumanEval (Chen et al., 2021). Second, on MultiPL-E (Cassano et al., 2023), PERC enhances code generation for underrepresented languages by leveraging data from high-resource languages.

Our contribution is three-fold:

- We propose PERC, a novel framework of leveraging algorithmic plans for few-shot example retrieval in code generation.
- We demonstrate that plan-based retrieval improves same-language code generation on competitive programming and general-purpose coding tasks.
- We confirm that PERC can leverage high-resource PLs to improve code generation accuracy in underrepresented PLs.

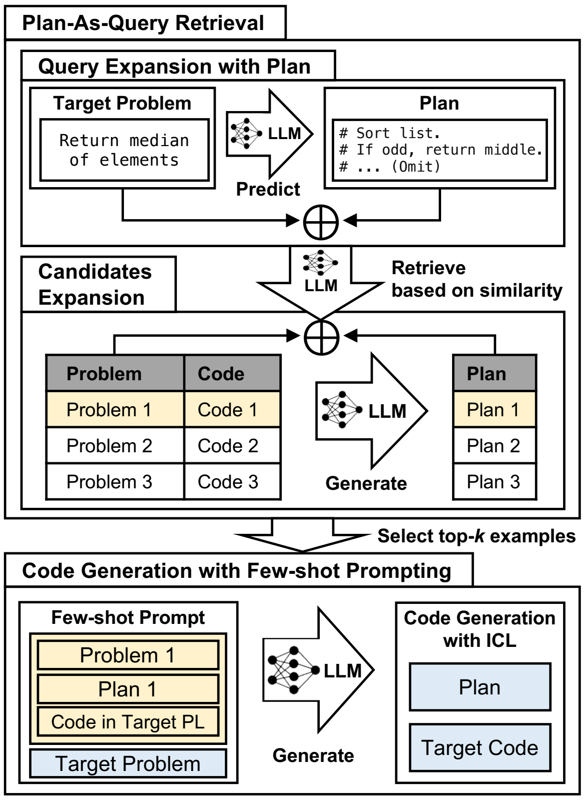

Figure 2: Overview of PERC: (1) plan-based retrieval and (2) code generation with few-shot prompting. PERC retrieves examples with the most similar algorithmic plans. Yellow and blue respectively signifies the triplet of the selected few-shot example and the target problem.

2 Related Work

Retrieval-Augmented Code Generation Previous works adopting RAG in code generation tasks have primarily focused on enhancing the accuracy of generated code by retrieving from the target PL pool (Parvez et al., 2021; Lu et al., 2022). More recently, CEDAR (Nashid et al., 2023) retrieved few-shot examples based on code-code similarity, and RepoCoder (Zhang et al., 2023) leveraged LLM-generated code snippets in target PL to expand queries, allowing for improved retrieval.

Utilizing Algorithmic Plans in Code Retrieval and Generation Han et al. (2021) viewed pseudocode as algorithmic plan of code, to reduce the modality gap between text and code in code search task. Jiang et al. (2024) used pseudocode-based algorithmic plans for code generation through few-shot prompting, and Sun et al. (2024) used pseudocode to bridge different programming languages.

Our distinction. We are the first to leverage algorithmic plans in retrieval-augmented code generation, which allows to retrieve effective few-shot examples by reducing lexical bias. As a result, our proposed PERC naturally adapts to source-target PLs mismatches.

3 Preliminaries

Given a natural language query t_q describing a desired program, code generation aims to return the corresponding implementation. In few-shot example retrieval, we draw relevant text-code pairs (t, c) from an example pool $\mathcal{P}$ to supplement LLM's knowledge and guide generation.

Problem-As-Query A baseline approach maps query t_q and text descriptions t into a shared latent space using encoder ψ. The top-k examples are selected based on similarity $\mathrm{sim}(\cdot)$:

$$E = \mathrm{topk}_{(t,c)\in\mathcal{P}}\ \mathrm{sim}(\psi(t_q), \psi(t)), \qquad (1)$$

where E is the set of indices of chosen examples.

CEDAR Nashid et al. (2023) selects examples based on code-code similarity. As the prompt lacks code for querying, we use the LLM's predicted code $\hat{c}_q$ from t_q to retrieve examples:

$$E_{\text{CD}} = \text{topk}_{(t,c)\in\mathcal{P}} \, \text{sim}(\psi(\hat{c}_q), \psi(c)). \quad (2)$$

RepoCoder Zhang et al. (2023) generates code $\hat{c}_q$ from t_q and expands the query, following LLM-based query expansion (Wang et al., 2023). The retrieval combines both problem description and code:

$$E_{\text{RC}} = \text{topk}_{(t,c)\in\mathcal{P}} \, \text{sim}(\psi(t_q; \hat{c}_q), \psi(t; c)), \quad (3)$$

where semicolon denotes concatenation.

4 PERC

PERC retrieves relevant examples using algorithmic plans in pseudocode. These plans capture high-level logic while minimizing cross-lingual lexical differences, thereby supporting accurate code generation.

As depicted in Figure 2, the workflow of PERC consists of two key steps. First, it drafts a plan for the given problem to form an expanded query, which is used to retrieve examples that were projected to plan space in indexing time. Then, the retrieved examples and their plans are integrated into a reasoning chain to generate a revised plan and the final code.

4.1 Plan-As-Query Example Retrieval

A key contribution of PERC is the use of algorithmic plans written in pseudocode, for effective retrieval. Specifically, an LLM generates pseudocode $\hat{p}$ for the retrieval pool $\mathcal{P}$ offline, and $\hat{p}_q$ for t_q at inference time. Then, the query is expanded with $\hat{p}_q$ as follows:

$$E_{\text{PERC}} = \text{topk}_{(t,\hat{p},c)} \, \text{sim}(\psi(t_q; \hat{p}_q), \psi(t; \hat{p})), \quad (4)$$

where the in-context example for $\hat{p}_q$ is provided in Appendix D. As illustrated in Figure 1, Eq (4) avoids surface-level details by projecting code c into plans. This is in contrast to Eq (3), which exposes the retriever to such distractions, especially when $\hat{c}$ and c use different PLs.

4.2 Code Generation with Examples

We use generated pseudocode as intermediate reasoning steps for code generation (Jiang et al., 2024). Each few-shot example in our prompt consists of a triplet $(t, \hat{p}, c)$, where text description t, generated pseudocode $\hat{p}$, and code c guide the LLM to utilize pseudocode in its reasoning chain:

$$\texttt{prompt} = [[t; \hat{p}; c]_{(t,\hat{p},c)\in E_{\text{PERC}}}; t_q]. \quad (5)$$

When the target programming language differs from that of example code c, we replace c with generated code $\hat{c}$ in the target language, where the LLM generates $\hat{c}$ using the in-context example shown in Appendix D:

$$\texttt{prompt} = [[t; \hat{p}; \hat{c}]_{(t,\hat{p},c)\in E_{\text{PERC}}}; t_q]. \quad (6)$$

5 Experiments

5.1 Experimental Setup

Experiments were conducted using GPT-3.5-Turbo-16k and GPT-4o-mini as the backbone LLMs. Other implementation details regarding the embedding-based retrieval and hyperparameter configuration for code generation can be found in Appendix A.

Metrics We evaluated the performance of PERC using the widely used Pass@1 metric (Chen et al., 2021), which is an unbiased estimator of the model's chance of producing a correct code sample in a single attempt.

Baselines We compared our method against several established baselines to highlight the effectiveness of PERC: 1) **w/o Examples** generates code directly without using few-shot examples, 2) **Random Selection** uses a randomly chosen, then fixed set of examples, 3) **Problem-As-Query Retrieval** retrieves examples based on problem-problem similarity, 4) **CEDAR** (Nashid et al., 2023) uses code-code similarity, and 5) **RepoCoder** (Zhang et al., 2023) expands the query with predicted code.

Datasets For evaluation, we used CodeContests (Li et al., 2022), HumanEval (Chen et al., 2021), and MultiPL-E (HumanEval; Cassano et al., 2023) benchmarks. For CodeContests, we used the train split as the example pool, while MBPP (Austin et al., 2021) benchmark was used for the other two. Throughout the benchmarks, we used Python—a high-resource PL—as the source. We used Python as the primary target PL since it is the only officially supported language in both CodeContests and HumanEval benchmarks. For additional target PLs, we selected languages based on the frequency classes (NICHE, LOW, MEDIUM) established in MultiPL-E. We randomly chose two PLs from each class: Ruby and Go (MEDIUM), Rust and R (LOW), and Lua and Julia (NICHE).

Benchmark	CodeContests	HumanEval	MultiPL-E					
Target PL	Python	Python	Rust	Julia	Lua	Ruby	Go	R
Data Availability	High-resource	High-resource	High-resource		⟵	⟶	Underrepresented	
Few-shot Prompting Method	Pass@1 (%)							
w/o Examples	2.72	73.17	62.82	51.45	49.94	47.76	30.91	14.78
Random Selection	5.82	72.56	61.86	52.77	60.68	<u>68.01</u>	45.58	33.42
Problem-As-Query Retrieval	4.97	69.51	<u>63.14</u>	53.65	60.93	65.22	<u>75.06</u>	31.06
CEDAR	5.82	72.44	62.63	<u>53.71</u>	<u>61.61</u>	61.68	<u>75.06</u>	32.67
RepoCoder	<u>6.48</u>	<u>73.78</u>	62.37	52.83	60.81	67.27	71.75	<u>34.16</u>
PERC	**6.61**	**76.04**	**63.78**	**54.21**	**64.10**	**69.81**	**76.49**	**34.35**

Table 1: Pass@1 scores of GPT-3.5-Turbo-16k augmented with different strategies for retrieving few-shot examples from Python source pools across CodeContests, HumanEval, and MultiPL-E benchmarks. Boldface indicates the best values while underline indicates the second-highest accuracy.

Benchmark	CodeContests	HumanEval	MultiPL-E					
Target PL	Python	Python	Rust	Ruby	Lua	Julia	Go	R
Data Availability	High-resource	High-resource	High-resource		⟵	⟶	Underrepresented	
Few-shot Prompting Method	Pass@1 (%)							
w/o Examples	4.97	87.87	81.22	78.88	66.96	61.51	52.73	49.07
Random Selection	5.58	86.59	81.35	79.50	73.98	68.36	47.01	53.42
Problem-As-Query Retrieval	7.64	87.07	80.58	80.12	72.92	65.97	68.83	52.80
CEDAR	<u>8.18</u>	**88.17**	81.15	80.75	74.04	<u>69.37</u>	<u>71.43</u>	53.42
RepoCoder	7.33	86.46	<u>81.54</u>	<u>81.99</u>	**75.65**	67.42	70.71	<u>55.28</u>
PERC	**8.48**	**88.17**	**82.95**	**83.85**	<u>75.22</u>	**70.69**	**71.69**	**57.14**

Table 2: Pass@1 scores of GPT-4o-mini augmented with different strategies for retrieving few-shot examples from Python source pools across CodeContests, HumanEval, and MultiPL-E benchmarks.

5.2 RAG from Same PL Pool: CodeContests and HumanEval

Table 1 shows that PERC outperforms all baselines, achieving Pass@1 scores of 6.61% and 76.04% on CodeContests and HumanEval, respectively, using the GPT-3.5-Turbo-16k model. Similarly, the results for GPT-4o-mini in Table 2 show Pass@1 scores of 8.48% and 88.17%, demonstrating consistently high performance across benchmarks. This supports that retrieval based on algorithmic plans better captures the logic of the code and allows to surface more effective demonstrations in top-k.

5.3 RAG from Cross-PL Pool: MultiPL-E

The results for each PL in MultiPL-E, presented in Tables 1 and 2, are sorted in descending order of code generation accuracy without examples. PLs with higher accuracy are considered to have higher data availability, while those with lower accuracy are regarded as underrepresented.

Using the GPT-3.5-Turbo-16k model, PERC achieves the best Pass@1 scores across all PLs, with notable results such as 69.81% for Ruby, 63.78% for Rust, and 64.10% for Lua, as shown in Table 1. The results for GPT-4o-mini, presented in Table 2, also emphasizes PERC's effectiveness, showing consistent Pass@1 score improvements, including 83.85% for Ruby, 70.69% for Julila, and 57.14% for R.

By effectively transferring knowledge from high-resource PLs, PERC demonstrated improved code generation accuracy for different PLs, showing its ability to bridge knowledge gaps across PLs with different data availability. In contrast, state-of-the-art approaches RepoCoder and CEDAR struggled with code generation. This limitation stemmed from their reliance on code-based retrieval, where modality differences introduced noise and hindered the identification of algorithmically relevant code.[1]

6 Analysis and Discussion

Open-Source LLM as a Backbone Table 3 shows consistent accuracy improvements with the open-source model Llama-3.1-8B-Instruct. PERC outperformed the baselines and demonstrated effective performance across PLs like Ruby, Lua, and R in the MultiPL-E benchmark. This highlights the improvements with PERC generalizes well to

[1]One may consider a cost-exhaustive approach of translating all the code in the pool to target (underrepresented) PLs, which incurs $\mathcal{O}(|\mathcal{T}||\mathcal{P}|)$ cost where $\mathcal{T}$ is the set of target PLs to handle, whereas PERC only requires $\mathcal{O}(|\mathcal{P}|)$.

Benchmark	MultiPL-E		
Target PL	Ruby	Lua	R
Method	Pass@1 (%)		
w/o Examples	46.09	39.63	18.82
Random Selection	45.34	38.70	22.36
RepoCoder	46.21	41.06	18.94
PERC	**47.33**	**44.22**	**23.66**

Table 3: Pass@1 scores of Llama-3.1-8B-Instruct augmented with different strategies for retrieving few-shot examples on Python source pool, on MultiPL-E benchmarks.

Benchmark	CodeContests		MultiPL-E	
Cand. PL	C++	Java	C++	Java
Target PL	Python	Python	Lua	Lua
RepoCoder	5.45	5.94	58.88	58.01
PERC	**6.61**	**6.06**	**64.60**	**64.60**

Table 4: Pass@1 scores of PERC and RepoCoder when using C++ and Java candidates in the CodeContests and MultiPL-E Lua benchmarks.

Cand. PL	Python	Python/C++	Python/C++/Java
RepoCoder	6.48	5.88	4.85
PERC	**6.61**	**6.48**	**6.12**

Table 5: Pass@1 scores on CodeContests as more examples from different PLs are added to the retrieval pool.

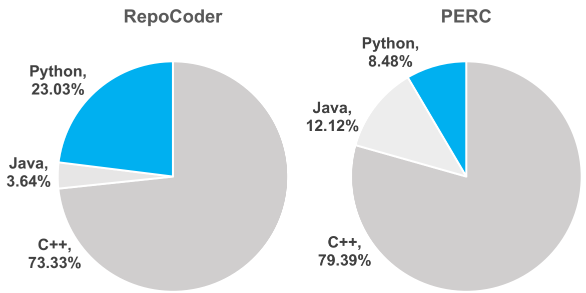

Figure 3: Source PL distribution of retrieved examples on CodeContests, under the Mixed PL Pool setting of Table 5. The target PL, Python, is highlighted in blue.

smaller, public backbone models.

Using C++ and Java as Source PLs Table 4 shows that PERC is also effective when using source PLs other than Python, namely C++ and Java. These results demonstrate that selecting examples based on algorithmic plans, regardless of the source PLs, can enhance code generation accuracy. The accuracy improvements using C++ and Java code pool for all MultiPL-E benchmarks are detailed in Appendix B.

Mixed PL Pool As shown in Table 5, when C++ and Java code were incrementally added to the retrieval pool, PERC maintained higher accuracy while RepoCoder suffered from severe performance degradation. Again, this showcases the adaptability and robustness of retrieving examples with plans, than with code.

As illustrated in Figure 3, PERC outperforms RepoCoder while retrieving less examples of the target PL, Python; this empirically supports that PERC retrieves useful examples in non-target PLs.

Method	Pass@1 (%)
PERC	**64.10**
- Converting to Target PL	55.28

Table 6: Pass@1 of PERC on MultiPL-E-Lua benchmark with and without code conversion to target PL.

Method	Pass@1 (%)
PERC w/ Converted Code	5.52
PERC w/ Gold Code	5.70

Table 7: Pass@1 difference when replacing generated target PL code with gold target PL code on subsets of example pools containing both C++ and Python code in CodeContets

Converting Code to Target PL As illustrated in Section 4.2, PERC uses the converted code $\hat{c}$ rather than the original code of the retrieved example c, if the source and target PLs do not match. Table 6 shows such conversion is crucial, as syntax, APIs, and other elements specific to the source PL may inadvertently influence the generation and lead to performance degradation. Additionally, as shown in Table 7, replacing the generated target PL code with the gold target PL code in CodeContests,[2] resulted in a minimal Pass@1 difference. This indicates potential errors that can be introduced in conversion to target PL are negligible.

7 Conclusion

We presented PERC, a novel framework for code generation that utilizes algorithmic plans both at indexing and generation time to select more effective few-shot examples to guide LLM. PERC demonstrates notable improvements in Pass@1 on CodeContests and HumanEval benchmark which represent scenario where the target PL is of high-resource, and also on MultiPL-E benchmarks for targeting underrepresented PLs.

[2]The subset of examples with both available correct C++ and Python code was used for evaluation.

Limitations

While PERC brings notable performance improvements in code generation with few-shot prompting, even if the source and target PLs do not match, our observation in Section 6 is that PERC shows slight performance drop as more and more programming languages are added to the retrieval pool. With an ideal retrieval system suited to selecting the most effective examples, one should observe monotonically increasing performance as more candidates are added to the pool; we leave more investigation and improvements as future work.

Acknowledgments

The first author was supported by Samsung Electronics. This work was partly supported by Electronics and Telecommunications Research Institute (ETRI) grant funded by ICT R&D program of MSIT/IITP (2022-0-00995, Automated reliable source code generation from natural language descriptions), and the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2024-2020-0-01789) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation).

References

Shengnan An, Bo Zhou, Zeqi Lin, Qiang Fu, Bei Chen, Nanning Zheng, Weizhu Chen, and Jian-Guang Lou. 2023. Skill-based few-shot selection for in-context learning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13472–13492, Singapore. Association for Computational Linguistics.

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

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Hojae Han, Youngwon Lee, Minsoo Kim, and Seungwon Hwang. 2021. Bridging code-text representation gap using explanation. In *Proceedings of The 13th Asian Conference on Machine Learning*, volume 157 of *Proceedings of Machine Learning Research*, pages 1033–1048. PMLR.

Xue Jiang, Yihong Dong, Lecheng Wang, Fang Zheng, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-planning code generation with large language models. *ACM Trans. Softw. Eng. Methodol.*

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A retrieval-augmented code completion framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6227–6240, Dublin, Ireland. Association for Computational Linguistics.

Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-based prompt selection for code-related few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2450–2462.

Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2719–2734, Punta Cana, Dominican Republic. Association for Computational Linguistics.

Stephen E. Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3:333–389.

Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. 2020. Mpnet: Masked and permuted pre-training for language understanding. *Advances in neural information processing systems*, 33:16857–16867.

Tao Sun, Linzheng Chai, Jian Yang, Yuwei Yin, Hongcheng Guo, Jiaheng Liu, Bing Wang, Liqun Yang, and Zhoujun Li. 2024. UniCoder: Scaling code large language model via universal code. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1812–1824, Bangkok, Thailand. Association for Computational Linguistics.

Liang Wang, Nan Yang, and Furu Wei. 2023. Query2doc: Query expansion with large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 9414–9423, Singapore. Association for Computational Linguistics.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, Singapore. Association for Computational Linguistics.

A Implementation Details

Throughout the experiments, we used GPT-3.5-Turbo-16k and GPT-4o-mini-2024-07-18 as the base LLM; for decoding, we applied nucleus sampling with $p = 0.95$ and sharpening with temperature $T = 0.8$. In all retrieval experiments, we employed semantic search using embedding models for retrieval, namely MPNet (Song et al., 2020)[3] as the encoder ψ. For a fair comparison, we also used this retriever for the RepoCoder baseline as well, which originally used a sparse bag-of-words model. The top-k candidates were then selected based on MPNet-based embeddings using cosine similarity, ensuring consistency across all methods.

In all few-shot prompting-based code generation experiments, ICL was performed using an inference chain consisting of problem description, pseudocode, and code. To ensure fair comparison of retrieval methods, the code was replaced with code converted to the target PL, as done in PERC's approach. Regarding the number of shots, we employed 3-shot prompting for all benchmarks, except CodeContests, where 1-shot prompting was necessary due to the model's 16k token limit.

Our code generation implementation primarily relied on the LangChain library.[4] To execute and evaluate the generated code, we used code from the CodeEval repository[5] hosted on Huggingface for running evaluating on CodeContests and HumanEval benchmarks. Additionally, we utilized code from the bigcode-evaluation-harness repository[6] to evaluate on the MultiPL-E benchmark suite.

B Using C++ and Java Candidates for MultiPL-E Benchmarks

Benchmark	MultiPL-E					
Candidate PL	C++	Java	C++	Java	C++	Java
Target PL	Lua	Lua	Ruby	Ruby	R	R
RepoCoder	58.88	58.01	68.51	65.28	30.43	33.91
PERC	**64.60**	**64.60**	**71.18**	**66.15**	**31.68**	**34.66**

Table 8: Experimental results comparing the Pass@1 of PERC and RepoCoder when using C++ and Java candidates in the MultiPL-E HumanEval-Lua, Ruby, and R benchmarks.

As shown in Table 8, even when the candidate programming languages are C++ and Java rather than Python, PERC outperforms RepoCoder in the MultiPL-E HumanEval-Lua, Ruby, and R benchmarks. These results demonstrate that selecting examples based on pseudocode, regardless of the candidate PLs, can improve code generation accuracy for underrepresented PLs by leveraging the knowledge from examples written in high-resource PLs.

[3]https://huggingface.co/sentence-transformers/all-mpnet-base-v2
[4]https://github.com/langchain-ai/langchain
[5]https://huggingface.co/spaces/evaluate-metric/code_eval
[6]https://github.com/bigcode-project/bigcode-evaluation-harness

C Ablation Studies

C.1 Pseudocode as Reasoning Chain

Benchmark	MultiPL-E Lua	
Reasoning Chain	-	Pseudocode
w/o Examples	49.94	-
Random Selection	59.38	**60.68**
Problem-As-Query Retrieval	60.81	**60.93**
CEDAR	59.69	**61.61**
RepoCoder	58.94	**60.81**
PERC	61.99	**64.10**

Table 9: Pass@1 scores of GPT-3.5-Turbo-16k with and without pseudocode as a reasoning chain.

C.2 Different Retrieval Methods

Benchmark	MultiPL-E Lua		
Retrieval Method	MPNet	CodeBERT	BM25
w/o Examples	49.94	49.94	49.94
Random Selection	60.68	60.68	<u>60.68</u>
Problem-As-Query Retrieval	60.93	58.76	60.37
CEDAR	<u>61.61</u>	<u>60.99</u>	60.25
RepoCoder	60.81	60.50	**63.23**
PERC	**64.10**	**61.80**	59.01

Table 10: Pass@1 scores of GPT-3.5-Turbo-16k with different retrievers.

D In-Context Learning Examples

We provide concrete examples of prompts used for in-context learning in Tables 11, 12, and 13. The final prompt is created by appending the target problem to 3-shot examples in the same format as shown in the tables.

Problem Description

```
def count_bidirectional(test_list):
    """Write a function to count bidirectional tuple pairs.
    """
```

Instruction for Plan Generation

Write a plan for the problem.

Generated Plan

```
# Define a function to count bidirectional tuple pairs.
# Create a map to store pairs and their reverse occurrences.
# Initialize a counter for bidirectional pairs.
# Iterate over the input list of tuples.
    # Retrieve the two elements of the tuple.
    # Create both the original pair and the reverse pair.
    # Check if the reverse pair exists in the map.
        # If it does, increment the bidirectional pair count and decrement the occurrence of the reverse pair in the map.
    # If the reverse pair does not exist in the map, increment the occurrence of the original pair in the map.
# Return the count of bidirectional pairs.
```

Table 11: The in-context example used for predicting the algorithmic plan from the test problem for query expansion by PERC. The problem description example is from Austin et al. (2021).

Problem Description

```
def count_bidirectional(test_list):
    """Write a function to count bidirectional tuple pairs.
    """
```

Code

```
def count_bidirectional(test_list):
    pair_count = {}
    bidirectional_count = 0
    for tup in test_list:
        a, b = tup[0], tup[1]
        original = (a, b)
        reverse = (b, a)
        if reverse in pair_count and pair_count[reverse] > 0:
            bidirectional_count += 1
            pair_count[reverse] -= 1
        else:
            pair_count[original] = pair_count.get(original, 0) + 1
    return bidirectional_count
```

Instruction for Plan Generation

Write a plan for the problem.

Generated Plan

```
# Define a function to count bidirectional tuple pairs.
# Create a map to store pairs and their reverse occurrences.
# Initialize a counter for bidirectional pairs.
# Iterate over the input list of tuples.
    # Retrieve the two elements of the tuple.
    # Create both the original pair and the reverse pair.
    # Check if the reverse pair exists in the map.
        # If it does, increment the bidirectional pair count and decrement the occurrence of the reverse pair in the map.
    # If the reverse pair does not exist in the map, increment the occurrence of the original pair in the map.
# Return the count of bidirectional pairs.
```

Table 12: The in-context example used for converting code in source PL to algorithmic plan by PERC. The problem description example is from Austin et al. (2021).

Problem Description

```
def count_bidirectional(test_list):
    """Write a function to count bidirectional tuple pairs.
    """
```

Code

```
def count_bidirectional(test_list):
    pair_count = { }
    bidirectional_count = 0
    for tup in test_list:
        a, b = tup[0], tup[1]
        original = (a, b)
        reverse = (b, a)
        if reverse in pair_count and pair_count[reverse] > 0:
            bidirectional_count += 1
            pair_count[reverse] -= 1
        else:
            pair_count[original] = pair_count.get(original, 0) + 1
    return bidirectional_count
```

Plan

```
# Define a function to count bidirectional tuple pairs.
# Create a map to store pairs and their reverse occurrences.
# Initialize a counter for bidirectional pairs.
# Iterate over the input list of tuples.
    # Retrieve the two elements of the tuple.
    # Create both the original pair and the reverse pair.
    # Check if the reverse pair exists in the map.
        # If it does, increment the bidirectional pair count and decrement the occurrence of the reverse pair in the map.
    # If the reverse pair does not exist in the map, increment the occurrence of the original pair in the map.
# Return the count of bidirectional pairs.
```

Instruction for Plan Generation

Convert to Lua code for the problem following the plan.

Converted Code to Target PL

```
function count_bidirectional(test_list)
    local pair_count = { }
    local bidirectional_count = 0

    for _, tup in ipairs(test_list) do
        local a, b = tup[1], tup[2]
        local original = {a, b}
        local reverse = {b, a}
        if pair_count[reverse] and pair_count[reverse] > 0 then
            bidirectional_count = bidirectional_count + 1
            pair_count[reverse] = pair_count[reverse] - 1
        else
            pair_count[original] = (pair_count[original] or 0) + 1
        end
    end

    return bidirectional_count
end
```

Table 13: The in-context example used for converting code in source PL to target PL by PERC. The problem description example is from Austin et al. (2021).

E Retrieved Example and Generated Code with PERC

We provide concrete example of the few-shot examples retrieved by PERC and the resultant pseudocode and generated code on HumanEval, in which the target PL is high-resource (python), in Tables 14 and 15. We also provide examples from MultiPL-E-Lua, underrepresented target PL setting, in Tables 16 and 17.

Problem Description t_q

```
from typing import List

def below_zero(operations: List[int]) -> bool:
    """ You're given a list of deposit and withdrawal operations on a bank account that starts with
    zero balance. Your task is to detect if at any point the balance of account falls below zero, and
    at that point function should return True. Otherwise it should return False.
    > below_zero([1, 2, 3])
    False
    > below_zero([1, 2, -4, 5])
    True
    """
```

Predicted Pseudocode $\hat{p}_q$

```
# Set the initial balance of the bank account to zero
# Iterate through each operation in the list
    # Update the balance of the bank account by adding the current operation
    # Check if the balance is less than zero
        # If it is, return True

# If the balance never falls below zero, return False
```

Retrieved Problem with Pseudocode $t; p$

Write a function to calculate the sum of the negative numbers of a given list of numbers.

```
# Initialize the sum of negative numbers to 0

# Iterate through each number in the list
    # Check if the number is negative
        # If it is negative, add it to the sum

# Return the sum of negative numbers
```

Retrieved Code c

```
def sum_negativenum(nums):
    sum_negativenum = list(filter(lambda nums:nums<0,nums))
    return sum(sum_negativenum)
```

Table 14: The example of pseudocode to extend the query for a target HumanEval problem #3, along with the problem, pseudocode, code, and converted code in the target PL of the retrieved candidate through PERC.

Problem Description

from typing import List

```
def below_zero(operations: List[int]) -> bool:
    """ You're given a list of deposit and withdrawal operations on a bank account that starts with
    zero balance. Your task is to detect if at any point the balance of account fallls below zero, and
    at that point function should return True. Otherwise it should return False.
    > below_zero([1, 2, 3])
    False
    > below_zero([1, 2, -4, 5])
    True
    """
```

Generated Pseudocode

```
# Initialize a variable "balance" to 0

# Iterate through each operation in the given list
    # Add the operation to the balance

    # Check if the balance is less than 0
        # If it is, return True
```

Generated Code

```
def below_zero(operations: List[int]) -> bool:
    balance = 0
    for operation in operations:
        balance += operation
        if balance < 0:
            return True
    return False
```

Table 15: An example of pseudocode and code generated using few-shot prompting with PERC for HumanEval problem #3, based on examples from Table 14.

Problem Description t_q

– You're given a table of deposit and withdrawal operations on a bank account that starts with
– zero balance. Your task is to detect if at any point the balance of account falls below zero, and
– at that point function should return true. Otherwise it should return false.
– > below_zero(1, 2, 3)
– false
– > below_zero(1, 2, -4, 5)
– true
local function below_zero(operations)

Predicted Pseudocode $\hat{p}_q$

```
# Set the initial balance of the bank account to zero
# Iterate through each operation in the list
    # Update the balance of the bank account by adding the current operation
    # Check if the balance is less than zero
        # If it is, return True

# If the balance never falls below zero, return False
```

Retrieved Problem with Pseudocode $t; p$

Write a python function to return the negative numbers in a list.

```
# Initialize an empty list to store the negative numbers
# Iterate through each element in the input list
    # Check if the current element is less than 0
        # If the element is negative, append it to the list of negative numbers

# Return the list of negative numbers
```

Retrieved Code c

```
def neg_nos(list1):
    out = []
    for num in list1:
        if num < 0:
            out.append(num)
    return out
```

Converted Code in Target PL $\hat{c}$

```
function getNegativeNumbers(numbers)
    local negativeNumbers = {}

    for _, num in ipairs(numbers) do
        if num < 0 then
            table.insert(negativeNumbers, num)
        end
    end

    return negativeNumbers
end
```

Table 16: The example of pseudocode to extend the query for a target HumanEval-Lua problem #3, along with the problem, pseudocode, code, and converted code in the target PL of the retrieved candidate through PERC.

Problem Description

– You're given a table of deposit and withdrawal operations on a bank account that starts with
– zero balance. Your task is to detect if at any point the balance of account fallls below zero, and
– at that point function should return true. Otherwise it should return false.
– > below_zero(1, 2, 3)
– false
– > below_zero(1, 2, -4, 5)
– true
local function below_zero(operations)

Generated Pseudocode

```
# Return the count of bidirectional pairs.
# Initialize the balance of the account to 0
# Iterate through each operation in the list
    # Add the operation to the balance
    # Check if the balance is less than 0
        # If it is, return true
# If the loop completes without returning true, return false
```

Generated Code

```
function below_zero(operations)
    local balance = 0

    for _, operation in ipairs(operations) do
        balance = balance + operation

        if balance < 0 then
            return true
        end
    end

    return false
end
```

Table 17: An example of pseudocode and code generated using few-shot prompting for the target HumanEval-Lua problem #3, based on PERC-retrieved examples from Table 16.