CUET_Expelliarmus at BLP2025 Task 2: Leveraging Instruction Translation and Refinement for Bangla-to-Python Code Generation with Open-Source LLMs

Md Kaf Shahrier, Suhana Binta Rashid,
Hasan Mesbaul Ali Taher, Mohammed Moshiul Hoque
Department of Computer Science and Engineering,
Chittagong University of Engineering & Technology, Chittagong 4349, Bangladesh
{u1804035,u1804037,u1804038}@student.cuet.ac.bd
moshiul_240@cuet.ac.bd

Abstract

Large language models (LLMs) have recently demonstrated strong performance in generating code from natural-language prompts. However, current benchmarks are primarily focused on English, overlooking low-resource languages like Bangla. This creates a critical research gap, as there are no well-established resources or systematic evaluations of code generation from Bangla instructions. To address the gap, we present a system that generates executable Python code from Bangla instructions. We design a two-stage pipeline: the Bangla instructions are first translated and refined into a clear English version to reduce ambiguity, and then the Python code is generated from the refined instructions through iterative error correction. For both instruction refinement and code generation, we used the open-source GPT-20B OSS model. On the official test set, our system achieves competitive results. We also analyze common errors such as unclear instructions, logical mistakes, runtime issues, and the need for external knowledge beyond the model's training data. Overall, our findings show that a simple translation–refinement pipeline can be an effective, low-cost approach to code generation in low-resource languages.

1 Introduction

Large Language Models (LLMs) have recently advanced automatic code generation, enabling systems to produce executable programs directly from natural-language instructions. Early progress was achieved with general-purpose models such as GPT-3 (Brown et al., 2020), followed by code-specific extensions such as Codex (Chen et al., 2021), CodeT5 (Wang et al., 2021), and PolyCoder (Xu et al., 2022), which significantly improved performance. Benchmarks such as HumanEval (Chen et al., 2021), APPS (Hendrycks et al., 2021), and CodeXGLUE (Lu et al., 2021) have since become standard for evaluating English-to-code generation.

However, progress in low-resource languages remains limited. Multilingual models such as XLM-RoBERTa (Conneau et al., 2020) and mBERT (Devlin et al., 2019) demonstrate cross-lingual transfer in NLP, but their application to code generation remained underexplored. In particular, Bangla, one of the most widely spoken languages globally, lacks systematic benchmarks and studies for code generation. While resources like BanglaBERT (Bhattacharjee et al., 2022) and other NLP benchmarks have supported Bangla text understanding they do not extend to executable code generation. Broader studies show that English-trained models often underperform on low-resource languages (Blasi et al., 2022). While proprietary systems (e.g., GPT-4, Claude 3) (OpenAI et al., 2023; Anthropic, 2024) and open-source initiatives (e.g., Aya, LLaMA-3) (Üstün et al., 2024; Grattafiori et al., 2024) expand multilingual coverage, systematic evaluation of Bangla instruction-to-code-generation remains absent.

To address this gap, benchmarks such as mHumanEval (Raihan et al., 2025a) and MBPP-Bangla (Raihan et al., 2025b) provide initial resources for Bangla-to-Python evaluation. Building on these, we develop a two-stage pipeline that translates and refines Bangla instructions into structured English, and then generates Python code from the refined instructions using iterative test-driven error correction. The critical contributions of the work are summarized as follows:

- We present the first systematic study on Bangla-to-Python code generation using open-source LLMs.
- We introduce a two-stage pipeline that leverages instruction translation, one-shot prompting for refinement, and zero-shot prompting for code generation.
- We incorporate a test-driven automatic cor-

Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025), pages 608–614

rection loop to ensure the reliability of the generated code.

- We conduct comparative experiments on the official test datasets.

  All resources and code used in this study are available at: https://github.com/Hasan-Mesbaul-Ali-Taher/BLP_25_Task_2 to support reproducibility and transparency.

2 Related Work

Although several studies have been conducted on code generation in high-resource languages, this issue is at a rudimentary stage in Bengali. Chen et al. (2022) introduced CodeT, a method that uses the same pretrained LMs to auto-generate test cases and then select the best code by running samples and checking dual execution agreement and improved pass@1 across benchmarks without manual test creation. Wang and Chen (2023) reviewed LLM-based code generation and highlights both applications and evaluation methods. This also emphasizes that while LLMs significantly improved developer productivity, the assessment of generated code remains underexplored, with limited quality measures considered. Nahin et al. (2025) introduced TituLLMs, the first Bangla LLMs (1B, 3B), which are trained on 37B tokens with an extended tokenizer and evaluated on five new benchmarks. TituLLMs outperformed the initial multilingual models. Besides, they have made the TituLLMs models and benchmarking datasets publicly available. A recent study (Bhowmik et al., 2025) identified NLP challenges, evaluated 10 LLMs across eight translated datasets, found performance gaps between Bengali and English, highlighted weaknesses in smaller models, and highlighted the need for better benchmarks and datasets.

Recent works have begun to address the lack of benchmarks for Bangla-to-code generation. The *mHumanEval* benchmark (Raihan et al., 2025a) extends the HumanEval dataset to over 200 natural languages, including Bangla, and uses machine translation and expert validation for 15 languages. This resource highlights LLMs' multilingual capabilities and enables cross-lingual evaluation for code generation. Complementing this, the *TigerCoder* suite (Raihan et al., 2025b) introduced the first dedicated family of Bangla code LLMs (1B and 9B parameters) with a curated Bangla code instruction dataset and the MBPP-Bangla benchmark. Their models achieved 11- 18% higher Pass@1 accuracy than existing multilingual baselines and demonstrated that carefully curated data can substantially improve performance even for smaller-scale LLMs.

3 Task Description

The primary objective of this task is to automatically generate executable Python code from natural language instructions written in Bangla as described in the paper (Raihan et al., 2025c). Unlike traditional classification tasks, this problem requires models to bridge the gap between informal Bangla text and structured programming logic. A successful system must be able to (i) interpret the intent of the Bangla instruction, (ii) transform the intent into a well-defined problem specification, (iii) generate Python code that adheres to this specification, and (iv) ensure functional correctness by passing the provided test cases. This task is particularly challenging due to the scarcity of Bangla resources, the ambiguity of instructions, and the need to produce both syntactically valid and semantically correct code.

3.1 Dataset Description

The shared task organizers provided four datasets for development and evaluation at different stages. During the competition, we received the *Trial* and *Development (Dev)* datasets. After the task concluded, the *Test v1* and *Test Full* datasets were released for final benchmarking. Table 1 summarizes the key characteristics of these datasets.

Dataset	Rows	Instr.	Resp.	Tests
Trial	74	Yes	Yes	2–3
Dev	400	Yes	No	3
Test v1	500	Yes	No	1
Test Full	500	Yes	Yes	3

Table 1: Overview of datasets.

4 Methodology

The proposed approach follows a two-stage pipeline that reliably generates executable Python code from Bangla instructions. In the first stage, Bangla problem statements are translated into English and refined into well-structured specifications using a text-to-text generation model with one-shot prompting. In the second stage, these refined instructions are passed to a text-to-code generation model, which produces candidate Python solutions.

For both of the stages, we have used the open-source **GPT-20B OSS** model. To ensure functional correctness, the generated code is validated against provided test cases, and a retry mechanism is employed when failures occur. The codes are then stored in the final solution set. The overview of the entire methodology have presented in Figure 1.

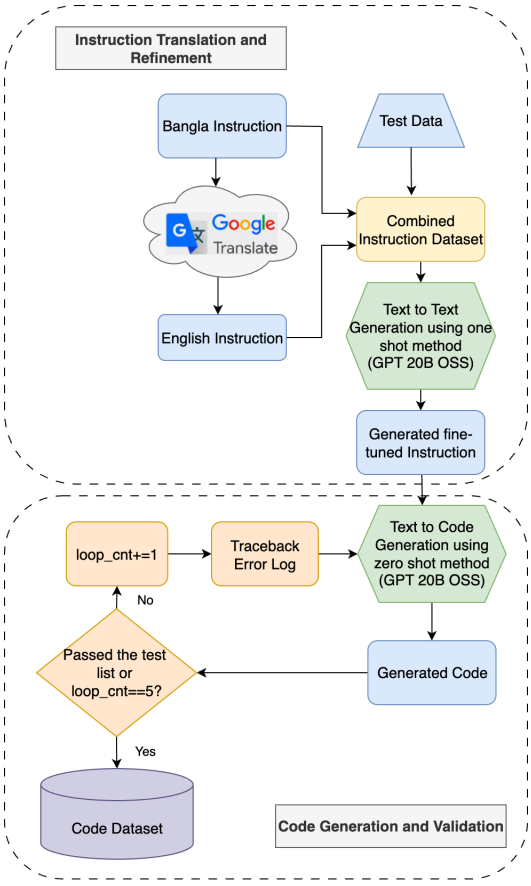

Figure 1: Overview of the proposed two-stage pipeline for Bangla instruction to Python code generation.

4.1 Instruction Translation and Refinement

Bangla instructions b are often informal, ambiguous, and stylistically diverse, which makes direct code generation difficult. To address this, we employ a translation–refinement strategy. Each instruction b is first translated into English instruction e using Google Translate, which provides a literal but sometimes noisy version. The Bangla instruction b, its English translation e, and the test cases T are then concatenated into a single input. This input is passed to a text-to-text generation model with one-shot prompting, where an example guides the model to convert slightly noisy instructions into a refined instruction r that forms a well-structured specification. The refined output r explicitly defines the function name, parameters, return type, and task requirements. As illustrated in Algorithm 1, this refinement step ensures that the instructions fed into the code generation stage follow a consistent structure, reduce ambiguity, and support reliable code generation.

Algorithm 1 Pseudo-code of the Proposed Methodology

Input: Bangla instruction b, test list T
Output: Verified Python code C

Translate b into English instruction e using Google Translate.
Concatenate $\{b, e, T\}$ into a single input.
Generate refined instruction r using a text-to-text model (one-shot).
Initialize loop counter $cnt \leftarrow 0$.
repeat
    Generate candidate code c from r using a text-to-code model.
    Execute c against the test list T.
    if all tests pass **then**
        $C \leftarrow c$
        break
    else
        Update the prompt with error feedback.
    end if
    $cnt \leftarrow cnt + 1$
until C is valid or $cnt = 5$
return C

4.2 Code Generation and Validation

The refined instruction r obtained from the previous stage is passed to a text-to-code generation model, which is prompted in a zero-shot setting to produce Python code c. The prompt is carefully structured to guide the model to generate only the function definition and the required logic, and to avoid unnecessary explanations or comments. This ensures that the output c remains concise and directly executable. Once a candidate solution c is generated, it is executed against the provided test list T. If the solution passes all test cases, it is accepted and stored as solution code C. If any test fails, the system captures the corresponding traceback and uses it to prompt the code generation model again. This retry mechanism is repeated up to 5 iterations, as illustrated in Algorithm 1 and Figure 1. If all test cases pass at any step, the candidate solution c is stored as the solution code C immediately. Otherwise, after five iterations, the

final candidate solution c is stored regardless of outcome. This error-driven re-generation increases the likelihood of producing functionally correct solutions.

4.3 Experimental Setup

All experiments were conducted on Kaggle using a Tesla T4 GPU (16 GB VRAM), 30 GB RAM, and a dual-core CPU, running Ubuntu and Python 3.10 with preinstalled PyTorch, CUDA, and Hugging Face Transformers. This environment ensured reproducibility with minimal setup. Table 2 summarizes the hyperparameters. The prompt limit (512) sets the maximum input size, and the generation limit (1024) controls reasoning and code output length. Reasoning capacity was kept low to avoid long chains that exceeded token limits. The batch size was restricted to 2 due to GPU constraints. A low temperature (0.1) ensured deterministic outputs, and a top-p (0.9) temperature balanced diversity, avoiding unlikely tokens.

Hyperparameter	Value
Prompt Token Limit	512
Generated Token Limit	1024
Reasoning Capacity	Low
Batch Size	2
Temperature	0.1
Top-p	0.9

Table 2: Hyperparameters used in the Kaggle T4 GPU experiments.

5 Results and Analysis

To evaluate our system, we conducted experiments across five folds of the test set, each consisting of 500 instances. At each fold, the model generated a candidate code per instruction, and the results were normalized by dividing the number of correct codes by 500. For evaluation, codes were first validated against a single test case; if at least one candidate passed, it was then checked against all test cases. When multiple candidates passed all tests, the shortest solution (measured in character length) was selected for final validation. We adopt this criterion because shorter programs generally use fewer tokens. This choice reduces the likelihood of unnecessary complexity and of including redundant or unintended operations that do not affect the program's functional correctness. Moreover, selecting the shortest candidate discourages the model from relying on specific patterns that may satisfy the test cases by accident, without truly generalizing. This makes the evaluation more consistent and keeps the comparison focused on the essential logic rather than stylistic variations.

During the shared task evaluation phase, the proposed system initially achieved an accuracy of *0.370*. This was primarily due to indentation errors in the generated code, which caused only single-line return statements to be accepted. After addressing this issue, we observed a significant improvement in performance as shown in Table 3. The results are reported under three experimental settings.

1. **Bangla Instruction + Test Cases:** Raw Bangla instructions with provided test cases passed directly to the code generation model.
2. **Refined Instruction:** Translated and refined English instructions containing explicit function definitions, parameter types, return values, and test cases.
3. **Refined Instruction + Error Log:** The refined instruction is further augmented with error feedback from failed generations, allowing iterative re-generation.

Model	F1	F2	F3	F4	F5
Bangla Instr. + Test Cases	0.104	0.108	0.108	0.112	0.128
Refined Instruction	0.860	0.870	0.876	0.876	0.884
Refined Instr. + Error Log	0.860	0.904	0.910	0.920	0.924

Table 3: Accuracy across folds (F1–F5). Each fold corresponds to 500 test instances.

Table 3 reports fold-wise accuracies for the three evaluation settings. Performance is measured across five folds of 500 instances each, using the same candidate-generation and test-case–based validation procedure described earlier. It also highlights the steady improvement from one setting to the next and reports a peak accuracy of 0.924.

5.1 Ablation Study

To quantify the individual contributions of each component in our pipeline, we conduct an ablation

across three variants: (1) **Raw Bangla Instruction + Test Cases**, (2) **Refined Instruction**, and (3) **Refined Instruction + Error Log**. The results are presented in Table 3. Each setting introduces exactly one additional step and allows us to isolate the impact of instruction refinement and error-driven regeneration. As shown in Figure 2, the refinement step yields the largest performance gain, taking a massive leap in the accuracy from an average of $\sim$0.112 to $\sim$0.873. This demonstrates that translating and restructuring the instruction into a precise English specification drastically reduces the ambiguity of the Bangla prompts. Adding error logs provides a further boost in the accuracy, achieving a peak accuracy of 0.924 in the fifth fold. The improvement is consistent across all folds. This demonstrates that the error feedback helps correct runtime and logical mistakes that persist even after the refinement step.

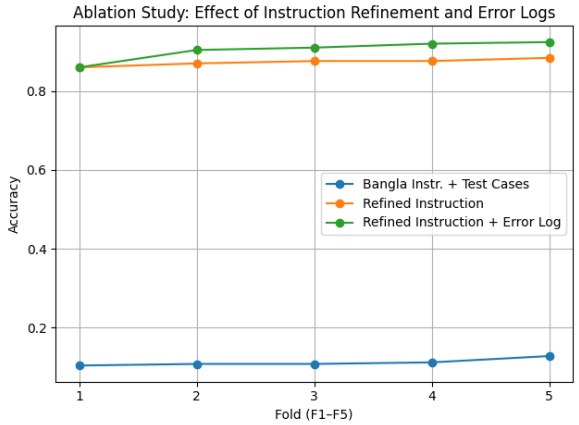

Figure 2: Ablation study comparing accuracy across folds for different pipeline variants.

Overall, the ablation indicates that *instruction refinement* is the most influential component of the pipeline, whereas the *error-driven regeneration* provides complementary gains. Together, these components make our two-stage pipeline highly effective for Bangla-to-Python code generation in low-resource settings.

6 Conclusion

This study introduced a two-stage pipeline for Bangla-to-Python code generation. The pipeline first translates and refines Bangla instructions, then generates code with test-driven correction. Experiments on shared-task datasets showed that direct Bangla-to-code generation gave suboptimal results. In contrast, the proposed refinement and error feedback loop led to substantial improvements, reaching an accuracy of up to 0.924. This work is the first systematic approach to Bangla code generation and shows that translation and refinement are effective for low-resource programming tasks. Future work may extend this approach to larger multilingual large language models and more comprehensive Bangla code benchmarks. Task-specific models could be evaluated to enhance translation, refinement, and code generation. For example, the current system uses the same GPT 20B OSS model for both instruction refinement and code generation. Future research could investigate using a specialized text-oriented model to refine instructions and another model optimized for code generation. Such specialization may yield significant performance gains. The pipeline could also be adapted to support more programming languages and complex tasks. Model selection may be broadened to include smaller yet more capable models, such as the Qwen family, which could achieve high accuracy with reduced computational resources. Advanced methodologies, such as AI agents operating sequentially to refine instructions and generate code, may also be explored to improve automation and overall system performance.

Limitations

This study comes with several shortcomings. First, the translation step relies on an external tool, i.e., Google Translate, which may introduce noise or semantic shifts in Bangla instructions and may potentially affect downstream code generation. Second, the experiments were constrained by limited computational resources and conducted in small batch sizes and restrictive hyperparameters. Third, our evaluation was conducted exclusively on the available shared-task datasets, which remain limited in scale and diversity. Moreover, the framework has only been tested with a single open-source model, GPT-20B OSS, and its generalizability to other large language models or programming languages has not yet been explored.

References

Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku — model card. https://www.anthropic.com/claude-3-model-card.

Abhik Bhattacharjee, Tahmid Hasan, Kazi Samin Mubasshir, Md Saiful Islam, Wasi Uddin Ahmad,

Anindya Iqbal, M. Sohel Rahman, and Rifat Shahriyar. 2022. Banglabert: Language model pretraining and benchmarks for low-resource language understanding evaluation in bangla. In *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 1318–1327, Seattle, United States. Association for Computational Linguistics.

Shimanto Bhowmik, Tawsif Tashwar Dipto, Md Sazzad Islam, Sheryl Hsu, and Tahsin Reasat. 2025. Evaluating llms' multilingual capabilities for bengali: Benchmark creation and performance analysis. *Preprint*, arXiv:2507.23248.

Damián Blasi, Antonios Anastasopoulos, and Graham Neubig. 2022. Systematic inequalities in language technology performance across the world's languages. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5487–5510. Association for Computational Linguistics.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and 1 others. 2020. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *Preprint*, arXiv:2207.10397.

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, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. 2020. Unsupervised cross-lingual representation learning at scale. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8440–8451.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4171–4186.

Andrea Grattafiori and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Ethan Arora, Collin Guo, Dawn He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Dongdong Ma, Shujie Zhou, Sining Liu, Duyu Jiang, Jiahai Lin, Daxin Tang, and 1 others. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Proceedings of the 35th Conference on Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*.

Shahriar Kabir Nahin, Rabindra Nath Nandi, Sagor Sarker, Quazi Sarwar Muhtaseem, Md Kowsher, Apu Chandraw Shill, Md Ibrahim, Mehadi Hasan Menon, Tareq Al Muntasir, and Firoj Alam. 2025. Titullms: A family of bangla llms with comprehensive benchmarking. *Preprint*, arXiv:2502.11187.

OpenAI, Josh Achiam, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Nishat Raihan, Antonios Anastasopoulos, and Marcos Zampieri. 2025a. mHumanEval - a multilingual benchmark to evaluate large language models for code generation. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 11432–11461, Albuquerque, New Mexico. Association for Computational Linguistics.

Nishat Raihan, Antonios Anastasopoulos, and Marcos Zampieri. 2025b. Tigercoder: A novel suite of llms for code generation in bangla. *arXiv preprint arXiv:2509.09101*.

Nishat Raihan, Mohammad Anas Jawad, Md Mezbaur Rahman, Noshin Ulfat, Pranav Gupta, Mehrab Mustafy Rahman, Shubhra Kanti Karmakar, and Marcos Zampieri. 2025c. Overview of BLP-2025 task 2: Code generation in bangla. In *Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025)*. Association for Computational Linguistics (ACL).

Ahmet Üstün, Viraat Aryabumi, Zheng-Xin Yong, Wei-Yin Ko, Daniel D'souza, Gbemileke Onilude, Neel Bhandari, Shivalika Singh, Hui-Lee Ooi, Amr Kayid, Freddie Vargus, Phil Blunsom, Shayne Longpre, Niklas Muennighoff, Marzieh Fadaee, Julia Kreutzer, and Sara Hooker. 2024. Aya model: An instruction finetuned open-access multilingual language model. *arXiv preprint arXiv:2402.07827*.

Jianxun Wang and Yixiang Chen. 2023. A review on code generation with llms: Application and evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pages 284–289.

Yue Wang, Weishi Wang, Shafiq Joty, Steven C.H. Lin, and Hwee Tou Ng. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 8696–8708.

Frank F Xu, Suraj Nair, Shuyan Lin, Pengcheng Yin, and Graham Neubig. 2022. Polycoder: A model

for programming by example in multiple languages.
arXiv preprint arXiv:2202.13169.