BRACU_CL at BLP-2025 Task 2: CodeMist: A Transformer-Based Framework for Bangla Instruction-to-Code Generation

Md. Fahmid-Ul-Alam Juboraj[1], **Soumik Deb Niloy**[1], **Mahbub E Sobhani**[1,2]
Farig Yousuf Sadeque[1,*]
[1]BRAC University [2]United International University
{md.fahmid.ul.alam.juboraj, soumik.deb.niloy}@g.bracu.ac.bd
msobhani2410011@mscse.uiu.ac.bd, farig.sadeque@bracu.ac.bd
∗ denotes corresponding author

Abstract

We propose CodeMist, a hybrid framework for Bangla-to-Python code generation, focusing on enhancing code accuracy through a two-stage pipeline of generation and debugging. In the development phase, standalone models such as `TigerLLM` and `StarCoder` achieved low accuracies of 27% and 24%, respectively, while advanced models like `Gemini-1.5-flash` and `Gemma` reached 60% and 64%. Pairing `Gemma` with the `GPT-OSS` debugger resulted in a substantial improvement to 99.75%, emphasizing the importance of a dedicated debugging stage. In the test phase on unseen data, `GPT-OSS` alone achieved 67%, which increased to 71% with self-debugging. The highest performance of 84% was achieved by combining `Gemini-2.5-flash` as the generator with `GPT-OSS` for debugging. These results demonstrate that integrating a strong generative model with an effective debugging component produces superior and robust code generation outcomes, outperforming existing approaches such as `TigerLLM`. The full implementation of the framework is publicly available at https://github.com/fahmid-juboraj/Code_generation.

1 Introduction

Automated code generation has witnessed rapid advancement with the emergence of Large Language Models (LLMs) such as *CodeT5* (Wang et al., 2021) and *CodeGen* (Nijkamp et al., 2022), which achieve over 90% accuracy on benchmarks like *HumanEval* (Raihan et al., 2025a). These models translate natural language instructions into executable code, substantially enhancing developer productivity and software development efficiency.

Despite these successes, most LLMs are trained primarily on English-based instructions, creating a linguistic bias in code generation research (Raihan et al., 2025c). This English-centric training restricts accessibility for non-English-speaking developers and undermines the inclusivity of AI-assisted programming tools. While languages such as Chinese and Japanese have begun to receive research attention, Bangla, the fifth most spoken language globally with over 300 million speakers, remains largely unexplored in this domain. Existing multilingual models struggle to generalize Bangla semantics to programming constructs due to the lack of high-quality Bangla–Python parallel datasets and the absence of specialized benchmarks for evaluation.

To address these limitations, this study introduces CodeMist, a hybrid Bangla-to-Python code generation framework that integrates generative modeling with automated debugging which employs the Gemini API to generate Python code from Bangla instructions and subsequently refines it through GPT-OSS, a locally fine-tuned GPT-based model capable of detecting and correcting syntax and logical errors. This dual-stage framework not only enhances the accuracy and correctness of generated code but also demonstrates the effectiveness of coupling code generation with an adaptive debugging mechanism for non-English programming tasks. The contribution lies in expanding code generation research beyond English, offering a foundational step toward linguistically inclusive AI programming systems.

2 Related Work

BanglaBERT (Bhattacharjee et al., 2022) pioneered Bangla-specific language modeling and introduced the BLEU benchmark for NLU evaluation. Subsequent models such as *TigerLLM* (Raihan et al., 2025d) and *TituLLMs* (Ahmed et al., 2025) improved reproducibility, coverage, and transliteration handling. Community-driven efforts like the *BanglaLLM* project (BanglaLLM Organization, 2024) released open-source models such as *Bangla-LLaMA-13B* (Zehady and the

Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025), pages 656–662

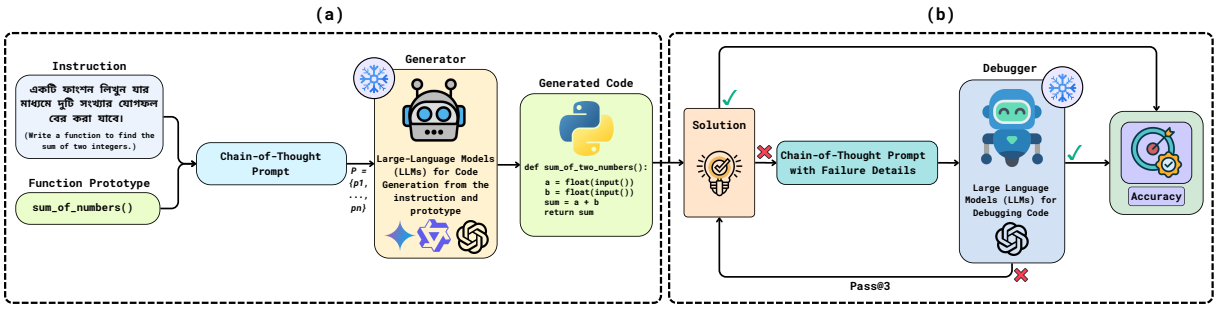

Figure 1: **(a)** A Chain-of-Thought prompt combines a natural-language instruction in Bangla with a function prototype, which is provided to the Generator to produce candidate implementations. **(b)** A Debugger constructs a Chain-of-Thought prompt with failure details, and debugging LLMs are engaged to iteratively repair and re-evaluate the generated code, where ✓ denotes correct code and ✗ denotes faulty code.

BanglaLLM Team, 2024), enabling wider experimentation. Parameter-efficient fine-tuning approaches such as *LoRA* (Hu et al., 2021) have been applied to resource-constrained models like *Gemma 2B* (Dasgupta, 2024) and *Bangla-LLaMA* (Saiful, 2023), demonstrating scalable adaptation methods.

The BLEU benchmark (Bhattacharjee et al., 2022) evaluates core Bangla NLU tasks, including sentence classification and sequence labeling. Later datasets introduced task-specific challenges such as tense classification in *BanglaTense* (Rahman et al., 2025) and sentiment analysis in *BnSentMix* (Data Analytics Research Group, 2024). Despite these advances, standardized benchmarks remain limited (Khan et al., 2025), restricting cross-model comparison. Community repositories like (Bangla NLP Community, 2024) consolidate datasets, yet coverage of generative tasks, especially code generation, is insufficient. *mHumanEval* addresses this gap with a massively multilingual benchmark, featuring prompts in 204 languages and solutions in 25 programming languages to evaluate cross-lingual coding performance.

While LLMs achieve strong performance in English code generation (Chen et al., 2021; Nijkamp et al., 2022), Bangla instruction-based code generation remains largely unexplored. Most Bangla LLMs target natural language understanding rather than code synthesis. Only a few, such as *TigerCoder* (Raihan et al., 2025b), focus on code generation, but they are limited by small datasets, lack of standardized evaluation, and minimal error-correction mechanisms. The absence of large-scale Bangla–Python parallel corpora and the challenge of mapping Bangla semantics to programming logic make reliable code generation non-trivial. Our work addresses these limitations by curating a comprehensive Bangla-to-Python dataset and developing a two-phase hybrid framework combining code generation with iterative debugging using models like *GPT-OSS*, achieving higher accuracy and robust synthesis on both development and unseen test sets.

3 Methodology

In this section, we outline the proposed pipeline for generating Python code from Bangla instructions, as shown in Figure 1.

3.1 Problem Formulation

We are studying the task of generating code from instructions provided in the Bangla language. Each problem consists of a Bangla instruction set, denoted as $INS = \{ins_1, ins_2, \ldots, ins_n\}$, paired with a corresponding function prototype set, $FP = \{fp_1, fp_2, \ldots, fp_n\}$ and explicit CoT prompt $CP = \{cp_1, cp_2, \ldots, cp_n\}$. The objective is to generate the correct Python code, represented as $\hat{Y}$, for each pair (ins_i, fp_i). This framework enables us to compare different modeling approaches: code generator models that directly map from (ins_i, fp_i, cp_i) to $\hat{Y}$ and code debugger models that iteratively refine the generated code. Throughout the execution of our CodeMist pipeline, the parameters for all models were kept frozen. The entire procedure can mathematically be abbreviated as follows:

$$\hat{Y} = LM_{\text{dbg}}([cp,\ LM_{\text{gen}}(ins, fp, cp)]) \quad (1)$$

3.2 Motivation

Automated code generation from Bangla instructions often faces errors due to linguistic ambigu-

Generation	Debugging	Accuracy(%)
StarCoder	None	24.00
TigerLLM	None	27.00
Qwen-1.5B	None	27.00
TigerCoder	None	33.00
TigerCoder	GPT-OSS	53.00
Gemini-1.5-flash	None	60.00
Gemma	None	64.00
Gemma	**GPT-OSS**	**99.75**

Table 1: Performance of Code Generation and Debugging (Development Phase)

ity and model limitations. A pipeline that integrates code generation with recurring debugging enhances reliability while lowering manual effort. This approach enhances the accuracy and usability of generated programs, facilitating effective Bangla-to-code translation and evaluation.

3.3 CodeMist

In this section, we provide the details of our code generator-debugger pipeline.

3.3.1 Code Generator

The process begins with a CSV file containing multiple columns such as `instruction`, `id`, and other contextual features. Each instruction is first processed by a Prompt Design module, which transforms the input text into a structured prompt optimized for the Gemini API. This module may incorporate additional contextual information from the CSV, such as example inputs/outputs or constraints, to reduce ambiguity and improve model comprehension. The quality and clarity of these prompts are crucial, as they directly influence the correctness and functionality of the generated code. The prepared prompt is then sent to the Gemini API, which returns the generated code through its response interface. All outputs are stored in a JSON file containing each `id` and its corresponding response. The correctness of the generated code is subsequently evaluated using a Python script, `Scoring.py`. This evaluation informs both performance metrics and areas where debugging may be necessary.

3.3.2 Code Debugger

For cases where the generated code is incorrect or suboptimal, **CodeMist** activates the Code Debugger pipeline. This phase begins by logging errors and linking them with the corresponding instructions and failure reasons in an updated

Generation	Debugging	Accuracy(%)
GPT-OSS	None	67.00
GPT-OSS	GPT-OSS	71.00
Gemini-2.5-flash	**GPT-OSS**	**84.00**

Table 2: Performance of Code Generation and Debugging (Test Phase)

CSV file. A local model interface is initialized by downloading and configuring an appropriate model (e.g., GPT-OSS:20B) along with a system prompt that combines the original instruction, contextual information, and the identified failure reason. This enriched prompt guides the local model to produce corrected versions of the code. By explicitly including both the original instruction and failure context, the debugger can resolve syntax, logical, and runtime errors more effectively. Each erroneous case is reprocessed to generate improved outputs, which replace the previous responses and are saved in a new submission JSON file following the same schema (`id`, `response`). The corrected codes are subsequently evaluated using `Scoring_V2.py` to enable comparative performance analysis between the initial Gemini-generated results and the refined open-source LLM-based solutions.

4 Experimental Analysis

4.1 Dataset

We utilize two datasets provided by the BLP Workshop (Raihan et al., 2025c) for training, validation, and evaluation of Bangla-to-Python code generation models. Both datasets contain Bengali problem descriptions paired with corresponding Python test cases but do not include ground-truth solutions. The `dev_v2.csv` dataset comprises 400 tasks and is used for model development and hyperparameter tuning, while the `test_v1.csv` dataset includes 500 tasks and serves as the held-out evaluation set. Each instance contains three fields—`id`, `instruction`, and `test_list`—where `instruction` provides the problem statement in Bengali and `test_list` defines the functional requirements in Python. Together, these datasets enable systematic assessment of code generation models' ability to generalize from natural language to executable programs.

4.2 Performance Evaluation

During the development phase, predictions are evaluated using a static checker that executes all

assertion-based test cases. The metric is defined as:

$$\text{Pass@1}_{\text{dev}} = \frac{N_{\text{PASS}}}{N_{\text{TOTAL}}} \times 100\% \quad (2)$$

where N_{PASS} denotes the number of tasks whose generated code passes all test cases, and N_{TOTAL} is the total number of evaluated tasks.

For the final evaluation phase, a runtime-based evaluator with timeout and exception handling computes the functional correctness as:

$$\text{Pass@1}_{\text{final}} = \frac{N_{\text{correct}}}{N_{\text{total}}} \quad (3)$$

where N_{correct} represents the number of tasks whose generated programs pass all functional tests under execution, and N_{total} is the total number of tasks evaluated.

4.3 Experimental Results

4.3.1 Quantitative Results

During the development phase, several code generation models were evaluated independently and in combination with debugging support. As shown in Table 1, *TigerCoder* (Raihan et al., 2025b) achieved an accuracy of 33% without debugging, which improved to 53% when paired with the *GPT-OSS* (Community and Contributors, 2024) debugger. This illustrates the benefit of an explicit debugging phase in correcting syntax and logical errors in the generated code. Similarly, other standalone generation models such as `StarCoder` (Li et al., 2023) and *Qwen-1.5B* (Team, 2024c) achieved relatively low accuracies of 24% and 27%, respectively, indicating limited capability in producing fully functional code from Bangla instructions without assistance. In contrast, more advanced models like *Gemini-1.5-flash* (Team, 2024a) and *Gemma* (Team, 2024b) achieved higher accuracies of 60% and 64%, demonstrating improved contextual understanding and code synthesis. The most notable improvement was observed when Gemma was combined with the GPT-OSS debugger, achieving a remarkable accuracy of 99.75%. This underscores the significant role of the debugging component in refining model outputs. Overall, these findings confirm that integrating a powerful generator with a specialized debugger can dramatically enhance performance.

Subsequently, in the test phase, models were evaluated on unseen prompts to measure generalization. As shown in Table 2, *GPT-OSS* alone achieved a baseline accuracy of 67%, which increased to 71% with self-debugging (GPT-OSS + GPT-OSS). The combination of *Gemini-2.5-flash* (Team, 2024a) as the generator and *GPT-OSS* as the debugger achieved the best performance of 84%, highlighting the robustness of cross-model debugging.

4.3.2 Qualitative Analysis of Failures

To categorize failure types, we clustered failure descriptions based on their top keywords and manually interpreted the resulting groups.

Logical Failures. Keywords such as `testlist`, `comma`, and `syntax` indicate *structural or parsing errors* caused by formatting mistakes or missing elements.

Analytical Failures. Terms like `character`, `string`, and `execution` reflect *runtime or reasoning errors*, requiring understanding of expected program behavior.

Mathematical Failures. Keywords including `test`, `assertion`, and `exception` correspond to *validation or computation errors* such as failed assertions or unmet conditions.

5 Conclusion

This study demonstrates the effectiveness of a hybrid LLM-based pipeline for Bangla programming instruction understanding and Python code generation. By leveraging the Gemini API for initial synthesis and a local Ollama model for targeted correction, our approach overcomes language-specific limitations and achieves substantial performance gains. The iterative design ensures both scalability and adaptability, enabling error tracking, prompt refinement, and systematic evaluation. The improved accuracy underscores the value of integrating cloud-based and local models to enhance multilingual code generation. Future work will explore fine-tuning domain-specific Bangla LLMs, expanding datasets with richer semantic coverage, and applying reinforcement-based evaluation to further optimize generation quality.

6 Limitations

While our study provides valuable insights into Bangla to Python generation and the benefits of a generator–debugger pipeline, it has several limitations. First, all models were evaluated in a Chain-

of-Thought (Wei et al., 2022) setting without fine-tuning, limiting our understanding of how training or domain adaptation might affect results. Second, we relied on fixed prompts and debugging strategies, without exploring adaptive or iterative prompting that could refine reasoning. Third, our evaluation used a limited dataset, restricting conclusions about generalization and robustness. Finally, our assessment focused solely on automated correctness metrics, without human or qualitative analyses to capture broader usability and failure patterns.

References

Khandaker Tahmid Ahmed, Mahir Rahman, Taufiq Islam, and Sabbir Khan. 2025. Titullms: A family of bangla llms with comprehensive benchmarking. *arXiv preprint arXiv:2502.11187*.

Bangla NLP Community. 2024. Awesome datasets for bangla language computing. GitHub Repository.

BanglaLLM Organization. 2024. Banglallm: Bangla large language model. Hugging Face Model Hub.

Abhik Bhattacharjee, Tahmid Hasan, Wasi Uddin Ahmad, Yuan Li, Yong-Bin Kang, 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.

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*.

OpenAI Community and Contributors. 2024. Gpt-oss: Open-source gpt-like models for code understanding. `https://github.com/openai`. Used as Debugging Model.

Abhishek Dasgupta. 2024. Building a bangla llm by finetuning gemma 2b llm using low-rank-adaptation. *Medium Blog Post*.

Data Analytics Research Group. 2024. Bnsentmix: A large-scale bengali-english code-mixed sentiment analysis dataset.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low□rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Sabbir Ahmed Khan, Md Arafatur Rahman, Md Saiful Islam, and Kazi Sakib Ahmed. 2025. Evaluating llms' multilingual capabilities for bengali. *arXiv preprint arXiv:2507.23248*.

Raymond Li, Loubna Ben Allal, Peter Izsak, and et al. 2023. Starcoder: A large language model for code generation. `https://huggingface.co/bigcode/starcoder`. BigCode Project.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. In *International Conference on Learning Representations*.

Md Mahfuzur Rahman, Md Saiful Islam, Kazi Sakib Ahmed, and Sabbir Khan. 2025. Banglatense: A large-scale dataset of bangla sentences categorized by tense. *Data in Brief*, 53:110132.

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).

Nishat Raihan, Joanna CS Santos, and Marcos Zampieri. 2025d. Tigerllm - a family of bangla large language models. *arXiv preprint arXiv:2503.10995*.

Mohammed Saiful. 2023. Bangla llama: Finetune bangla llama model using lora approach. GitHub Repository.

Google DeepMind Team. 2024a. Gemini 1.5: Unlocking multimodal capabilities in long contexts. `https://deepmind.google/technologies/gemini/`. Accessed: 2025-09-27.

Google DeepMind Team. 2024b. Gemma: Lightweight open models from google deepmind. `https://ai.google.dev/gemma`. Accessed: 2025-09-27.

Qwen Team. 2024c. Qwen1.5: A comprehensive multilingual large language model. `https://huggingface.co/Qwen`. Accessed: 2025-09-27.

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

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Abdullah Khan Zehady and the BanglaLLM Team. 2024. Bangla llama 13b instruct v0.1. Hugging Face model card, https://huggingface.co/BanglaLLM/bangla-llama-13b-instruct-v0.1. Open-source Bangla instruction-tuned large language model (13B parameters).

A Appendix

A.1 Dataset Sample

This table provides a few representative samples from our dataset. Each row shows the sample ID, the corresponding task instruction in Bangla, and the Python assertions used to verify the expected output.

Table 3: Sample instances from the Dataset

ID	Instruction	Test List (Sample)
1	একটি ফাংশন লিখুন যা দিয়ে দেওয়া জোড়া থেকে তৈরী করা যায় সর্বাধিক শৃঙ্খল।	[’assert max_chain_length([Pair(5, 24), Pair(15, 25)]) == 3’]
2	একটি প্রদত্ত স্ট্রিং-এ প্রথম পুনরাবৃত্ত অক্ষর নির্ণয় করুন।	[’assert first_repeated_char(”abcabc”) == ”a”’]
3	একটি ফাংশন লিখুন যা n এর চেয়ে ছোট বা সমান একটি লুডিক সংখ্যা বের করবে।	[’assert get_ludic(10) == [1, 2, 3, 5, 7]’]
4	একটি প্রদত্ত স্ট্রিং এর শব্দগুলোকে বিপরীত করার প্রোগ্রাম লিখুন।	[’assert reverse_words(”python program”) == ”program python”’]
5	প্রদত্ত পূর্ণসংখ্যাটি একটি মৌলিক সংখ্যা কিনা তা যাচাই করুন।	[’assert prime_num(13) == True’]

A.2 Sample of Generated Failed With Details

This table shows the first five samples from `failed_with_details_test.csv`. Each row lists the sample ID, the type of failure encountered during testing, the task instruction in Bangla, and the corresponding Python test list that caused the failure.

Table 4: Samples from `failed_with_details_test.csv`

ID	Failures	Instruction	Test List
3	[X][X][X][X] Failed to parse test_list: invalid syntax near “,”.	একটি ফাংশন লিখুন যা একটি অভিধানে সবচেয়ে সাধারণ মানটি খুঁজে বের করবে।	[’assert count_common ([’red’,’green’,’black’], [’green’,’white’,’red’]) == 2’]
5	[X][X][X][X] Failed to parse test_list: invalid syntax near “]”.	একটি স্ট্রিংকে ছোট অক্ষরে বিভক্ত করার জন্য একটি ফাংশন লিখুন।	[’assert split_lowerstring (”AbCd”) == [’bC’,’d’]’]
6	[X][X][X][X] Failed to parse test_list: invalid syntax near “_”.	একটি ফাংশন লিখুন যাতে ছোট অক্ষরের ক্রম খুঁজে পাওয়া যায়।	[’assert text_lowercase_underscore (”aab_cbbbc”) == True’]
8	[X][X][X][X] Failed to parse test_list: invalid syntax near “(”.	প্রথম স্ট্রিং থেকে দ্বিতীয় স্ট্রিংয়ে উপস্থিত অক্ষরগুলি মুছে দিন।	[’assert str_to_list (”probass-curve”, ”pros”) == [”b”,”a”,”c”,”u”,”v”,”e”]’]
15	[X] Error in function definition: unexpected character.	একটি প্রদত্ত অ্যারেতে পুনরাবৃত্তি না হওয়া উপাদানগুলির যোগফল নির্ণয় করুন।	[’assert find_Product ([1,1,2,3],4) == 6’]