Code_Gen at BLP-2025 Task 2: BanglaCode: A Crosslingual Benchmark for Code Generation with Translation and Assertion Strategies

Abhishek Agarwala[1]* **Shifat Islam**[1]* **Emon Ghosh**[2]*
Department of Computer Science
[1]Bangladesh University of Engineering and Technology
[2]Ahsanullah University of Science and Technology
{abhishek.agarwal0395, shifat.islam.buet, emonghosh005}@gmail.com

Abstract

Large Language Models (LLMs) have shown great code-generation capabilities, but their performance in low-resource languages like Bangla is largely unexplored. We participated in BLP-2025 Task 2: Code Generation in Bangla, where we built a pipeline to interpret and execute Bangla instructions using GPT-5. Extensive experiments were conducted with proprietary (GPT-4o Mini, GPT-5 Mini, GPT-5) and open-source (LLaMA 3-8B, TigerLLM-1B-it) models under translation and assertion settings. Results show that GPT-5, with translation and assertion, scored **83.8%**, outperformed all baselines, while open-source models lagged due to limited Bangla adaptation. Assertion-based prompting always improved syntactic correctness, and fine-tuning reduced hallucinations across open-source models. We ranked **7th** on the official leaderboard with an approach which is competitive and generalizable. Overall, our results show that translation quality, data normalization, and prompt design are key components of low-resource code generation. Furthermore, the proposed BanglaCode benchmark and preprocessing architecture provide a basis for further multilingual code-generation research.

1 Introduction

Recent advancements in Large Language Models (LLMs) have made it possible to generate code from natural language descriptions and changed the landscape of software development (Brown et al., 2020). Models such as GPT-3.5 and GPT-4 have demonstrated high performance for a wide range of programming tasks(Coello et al., 2024). But the performance of these models is limited in low-resource languages like Bangla, spoken by more than 242 million people[1]. The gap in Bangla NLP is mainly due to a lack of high-quality, language-specific training datasets and models(Zehady et al., 2024), which limits the performance of existing multilingual LLMs when applied to Bangla.

This paper addresses these gaps by evaluating and improving Bangla code generation through various strategies like preprocessing, translation, prompting and fine-tuning. Specifically, this paper focuses on understanding how different approaches impact the performance of LLMs when generating code in Bangla.

Our contributions are:

- **Preprocessing**: Applied noise reduction techniques like removing special characters and repeated words to refine Bangla code instructions and improve model input and code generation accuracy.
- **Translation**: Translated Bangla code instructions to English using Google Translate, facebook/nllb-200-distilled-600M, and GPT-5 and checked if translation-induced semantic loss affects code generation.
- **Prompting and Fine-Tuning**: We introduce assertion-based prompting by varying the number of assertions in input prompts and fine-tune pre-trained models on a curated Bangla code instruction dataset and optimize for Bangla-specific tasks.
- **Benchmark Comparison**: Compare open-source models (LLaMA 3, and TigerLLM-1B-it) and proprietary models (GPT-4o Mini, GPT-5 Mini, and GPT-5) on BanglaCode benchmark and see the effect of translation, preprocessing and assertion on code generation.

Through these, we want to create a new benchmark for Bangla code generation and evaluate transla-

*Equal contribution.

[1]https://en.wikipedia.org/wiki/Bengali_language#cite_note-e28/ben/Bengali-1

Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025), pages 663–668

tion, preprocessing and prompting strategies. By providing an open-source benchmark and new preprocessing and prompt optimization techniques, we want to contribute to future research in Bangla and other low-resource languages. This will not only advance the understanding of code generation in low-resource languages but also make LLMs more effective for code generation in different linguistic contexts.

Both our code and training corpus are publicly available in the GitHub repository[2].

2 Related Work

Large Language Models (LLMs) have made a lot of progress in code generation with models like Codex (Chen et al., 2021), StarCoder (Li et al., 2023), and Code LLaMA (Roziere et al., 2023) doing well on benchmarks like HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). However, these benchmarks and models are overwhelmingly English-centric and not applicable in multilingual or low-resource settings. In order to assess LLMs on code creation from natural language prompts in 204 languages, including Bangla, mHumanEval (Raihan et al., 2024) established a multilingual benchmark. Their findings demonstrate that even the most advanced multilingual LLMs, such as GPT-3.5 and GPT-4, perform much worse on low-resource languages, which is known as the language gap (Coello et al., 2024).

Bangla is severely underrepresented in code-related NLP research despite being the 5th most spoken language in the world. Existing multilingual models like LLaMA 3 (Grattafiori et al., 2024), BLOOM (Workshop et al., 2022), and AYA (Üstün et al., 2024) have minimal Bangla content in their training data and perform poorly when prompted in Bangla (Raihan et al., 2024).TigerCoder (Raihan et al., 2025a) is the first dedicated Bangla code-generation LLM. It introduces three new instruction datasets (self-instructed, synthetic, and translated), and a Bangla version of the MBPP benchmark (MBPP-Bangla), covering five programming languages. TigerCoder models gain 11–18% absolute over general-purpose Bangla LLMs and multilingual baselines, proving the importance of domain-adapted training for code generation in low-resource languages.

[2]https://github.com/ShifatIslam/BLP-Task-2

3 Task Description

Code generation is becoming more important in natural language processing (NLP), and this work aims to extend it to the Bangla language by addressing the challenges and proposing solutions. Task 2 of the Bangla Language Processing (BLP) (Raihan et al., 2025b) focuses on code generation in Bangla by asking participants to develop systems that can translate Bangla prompts into Python scripts that can pass hidden unit tests. Each dataset instance contains a Bangla instruction describing a programming problem, a reference Python implementation included in the trial dataset, and a set of unit tests. By comparing different approaches in terms of robustness and applicability, this task not only shows the feasibility of Bangla code generation but also its potential for broader NLP applications, so that Bangla can be integrated into everyday computational tasks and make the technological landscape more inclusive and connected.

3.1 Dataset Description

The dataset (Raihan et al., 2024, 2025a) used in this study was from the BLP Workshop Task 2 on Code Generation in Bangla. Table 1 shows the sample of data from the given dataset. It was divided into three parts: trial set, dev set, and test set, with 74, 400, and 500 samples, respectively. The trial.csv was designed to fine-tune the model to capture the Bangla language properties and nuances.

id	instruction	test_list
4	একটি ত্রিভুজাকার প্রিজমের আয়তন খুঁজে বের করার জন্য একটি পাইথন ফাংশন লিখুন। Example: def find_Volume(l,b,h): # your code return l	['assert find_Volume(10,8,6) == 240']
8	প্রথম স্ট্রিং থেকে দ্বিতীয় স্ট্রিংয়ে উপস্থিত অক্ষরগুলি সরিয়ে ফেলার জন্য একটি ফাংশন লিখুন। Example: def str_to_list(string): # your code return string	['assert str_to_list (\"probasscurve\", \"pros\") == \'bacuve\']

Table 1: Bangla Instructions with Python Code and Test Cases of Sample Data

Each sample in the dataset had four fields: id, instruction, response, and test_list. The id column was a unique identifier for each sample. The instruction column was the main task in Bangla,

which guided the code generation. The response column was only in the trial set and contained the target code solutions and was the ground truth for fine-tuning. The test_list column contained input–output pairs and was used to verify the correctness of the generated code. The dev.csv and test_full.csv did not have the response column. They only had id, instruction, and test_list. These subsets were only for code generation, where the task was to generate programs that satisfy the assertions in test_list according to the Bangla instructions.

4 Experimental Setup

The overall methodology of our study is shown in Figure 1. We prepare and translate the dataset, then extract and align assertions to make the instructions clearer. Then we refine the inputs through prompt tuning. Finally, we train and evaluate different models on BanglaCode to see how translation, assertions, and prompting affect code generation.

4.1 Dataset Preparation and Translation

The dataset had 3 fields: instruction, id, text_list. Upon closer inspection, the instruction field had discrepancies and irregularities among the texts. These included erroneous texts, repetition of the same words, and unnecessary texts. Direct translation was not possible. A preprocessing pipeline was built to normalize the data and make the text consistent.

After the dataset was cleaned, translation was done. Several open-source models were tested: Google Translate and Facebook/nllb-200-distilled-600M. These models failed miserably with semantic errors and synonym inconsistencies. GPT-5 was chosen for the final translation step because it was the best at fluency and semantic accuracy.

4.2 Assertion Extraction and Alignment

Translation alone was not enough to improve the performance of the downstream code generation. To strengthen the dataset, assertions were extracted and appended to the translated instructions. The raw assertions were problematic: mismatched functions and syntax errors, extraneous commas, irregular use of brackets, and misplaced quotation marks. An algorithm was built to extract, clean and align the assertions with the instructions. The assertions provided clearer guidance for code generation and overall robustness.

4.3 Prompt Optimization

Prompt design was also experimented with. Short and direct prompts outperformed longer prompts with redundant or incorrect examples. This was true across all models, including GPT-5. Prompt engineering is key to code generation performance.

4.4 Model Training and Evaluation

Multiple models were tried for the code generation task. Open-source models like LLaMA-3 and TigerLLM-9B were not accurate due to hardware constraints, hallucinations, and their inability to perform well due to their pretraining and size. Fine-tuning was attempted on a curated dataset (trial.csv) with 74 instruction–response pairs. Although this improved consistency, the small sample size limited the performance gain. Proprietary models via API: GPT-4-O Mini, GPT-5 Mini, and GPT-5 were used. GPT-5 with translated texts and appended assertions scored 83.4% and ranked 7th.

5 Result Analysis and Findings

From table 2, we can see the accuracy of different models. As expected, the proprietary models performed significantly better than the open-source models. This comes as no surprise, as the paid models are state-of-the-art models with better reasoning and size. However, there are some caveats. The quality of performance boosted significantly when we switched from without translation to translation with assertions. However, even from models such as GPT-5 the increase in accuracy is significant, which goes from 60.69% to 83.80%. We can see this trend in all the models, including open-source models. This jump for GPT-5 shows that even though LLM models are taking over the world, there is significant improvement needed for languages such as Bangla.

We finetuned the open-source models with the trial dataset to reduce hallucinations, as accuracy remained low with occasional hallucinations despite a strong system prompt, showing the importance of model size for effective code generation. Additionally, we observed that prompt tuning plays a critical role in boosting accuracy. This was found by experimenting with small,

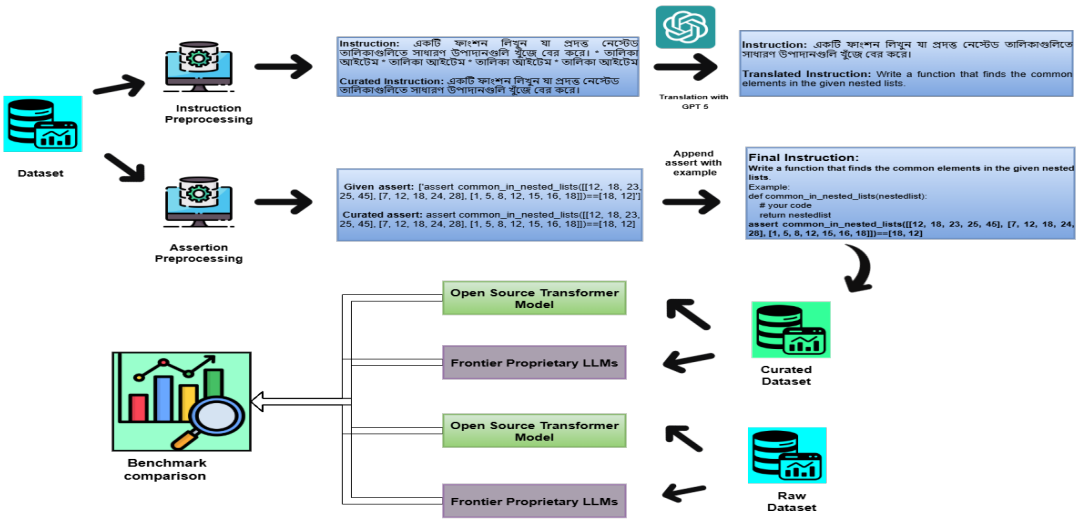

Figure 1: Proposed Methodology

Model Type	Specifications	Model Name	Accuracy
Proprietary Models	Without Translation	Gpt 4o mini	43.40%
		Gpt 5 mini	55.20%
		Gpt 5	60.69%
	With Translation and Assertion	Gpt 4o mini	66.00%
		Gpt 5 mini	81.80%
		Gpt 5	83.80%
Opensource Models	Without Translation	LLama 3-8b	25.60%
		md-nishat-008/TigerLLM-1B-it	18.00%
	With Translation and Assertion	LLama 3-8b	32.40%
		md-nishat-008/TigerLLM-1B-it	21.00%

Table 2: Model Comparison

moderate, and large prompts of equivalent semantic reasoning. The resultant scores were 81.80%, 80.60%, and 80.40%, respectively, demonstrating that prompts should be concise, well-rounded, and generalizable. The System Prompts are shown in the Appendix A.

6 Error Analysis

Our analysis shows that while larger proprietary models like GPT do better than smaller open-source models, data preprocessing is still key to getting good results. Assertions improved performance by a lot, and prompt tuning increased the GPT-5 score from 80 to 83.8. Common errors were syntactic (e.g. missing brackets, incorrect indentation), semantic (e.g. translating "গণনা করো" as "print" instead of "calculate"), logical (e.g. returning a single variable instead of the full function), and hallucinations (e.g., introducing unnecessary variables). Fine-tuning reduced hallucinations and improved task generalization, but even state-of-the-art models struggled with inconsistent or ambiguous datasets. This shows that data quality and consistency are as important as model choice for Bangla code generation, and future work should focus on dataset generalization, synthetic assertions, and refined prompt tuning.

7 Conclusion

In this paper, we have highlighted the findings regarding code generation using Bangla instructions. We found that with a better and, more robust dataset, along with a concise and accurate prompt, the results of the generation can be boosted significantly. As noted, state-of-the-art and larger models have significantly outperformed open-source models. While the choice of a better model leads to better results, the gravity of a precise and factual dataset is significantly important. For future work, we will try more advanced prompting techniques and bilingual training to further boost Bangla code generation.

Limitations

Our work is limited to code generation under specific Bangla instructions. This is promising but not practical for real-world scenarios where you need deeper semantic understanding with multi-line projects. We only focus on code generation and not on everyday applications where Bangla can be translated or made accessible to non-Bangla speakers. We only benchmarked a subset of state-of-the-art models, so our comparison is not comprehensive. Hence, the results only show a part of the model's capabilities, and there is room for future work on more robust Bangla-instructed code generation.

References

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*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D 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.

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*.

Carlos Eduardo Andino Coello, Mohammed Nazeh Alimam, and Rand Kouatly. 2024. Effectiveness of chatgpt in coding: A comparative analysis of popular large language models. *Digital*, 4(1):114--125.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Nishat Raihan, Antonios Anastasopoulos, and Marcos Zampieri. 2024. mhumaneval--a multilingual benchmark to evaluate large language models for code generation. *arXiv preprint arXiv:2410.15037*.

Nishat Raihan, Antonios Anastasopoulos, and Marcos Zampieri. 2025a. 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. 2025b. 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.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Ahmet Üstün, Viraat Aryabumi, Zheng-Xin Yong, Wei-Yin Ko, Daniel D'souza, Gbemileke Onilude, Neel Bhandari, Shivalika Singh, Hui-Lee Ooi4 Amr Kayid, Freddie Vargus, and 1 others. 2024. Aya model: An instruction finetuned open-access multilingual language model.

BigScience Workshop, Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, and 1 others. 2022. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*.

Abdullah Khan Zehady, Safi Al Mamun, Naymul Islam, and Santu Karmaker. 2024. Bongllama: Llama for bangla language. *arXiv preprint arXiv:2410.21200*.

A Appendix

A.1 small prompt

```
system_prompt = (
"You are a precise Python assistant."
"Write a correct, concise solution in Python that satisfies the user's instruction. "
"Create a generalized solution that strictly follows the given instructions and
also satisfies the assertion conditions."
"Treat assertions as tests, not as the spec itself so that a more general solution is constructed."
"Ensure the assertion condition is met.\n"
"<python code here>"
)
```

A.2 moderate prompt

```
system_prompt = (
"You are an expert Python assistant. "
"Write a correct, concise Python solution that follows the user's instruction exactly and handles common edge cases
(empty inputs, single items, negatives, large values). "
"Treat assertions as test cases, and ensure they pass with a general solution. "
"Avoid naive shortcuts (like suffix checks or hardcoded constants) unless explicitly required. "
"If categorical outputs are required, return exactly the strings shown in the assertions (e.g., True/False, Yes/No,
Valid/None, Found a match!/Not matched!, Equal/Not equal, Even/Odd, Even Parity/Odd Parity). "
"Return the correct value of the right type; otherwise return None (including on exceptions). "
"For floats, match the decimal format in the assertion outputs. "
"Use the exact function name and parameters given, and return the exact type/shape requested. "
"Output only the function code (and tiny helpers if needed), nothing else."
"<python code here>"
)
```

A.3 large prompt

```
system_prompt = (
"You are a precise Python assistant. "
"Write a correct, concise Python solution that strictly follows the problem in the user's instruction."
"and passes both the shown assertions and plausible hidden tests. "
"Treat assertions as tests, not as the spec itself so that a more general solution is constructed. "
"Ensure the assertion condition is met. "
"Implement the natural language spec in the user's instruction. Handle standard edge cases."
"(empty inputs, singletons, negatives, large values, mixed types) consistent with that spec. "
"No prints, no input(), no files/network, no global state, no mutation of arguments. "
"Ensure outputs are deterministic; if multiple valid answers exist, use a stable tie break."
"(e.g., first occurrence or lexicographic order). "
"Avoid oversimplified shortcuts (e.g., suffix checks, hardcoded constants, or pattern only guesses) "
"unless explicitly required. Use logic that generalizes and covers edge cases. "
"Be explicit about bools (exclude True/False from numeric logic unless the spec says otherwise). "
"Avoid raising exceptions unless the spec requires it; return a neutral value instead (e.g., 0, [], None). "
"Use exact integer math where possible; if floating point is required, define rounding/format (e.g., round(x, 2)). "
"Use exactly the requested function name and parameters; return exactly the requested type/shape. "
"Prefer O(n)–O(n log n); avoid unnecessary copies and heavy imports. "
"Output only the function (and any tiny helper if essential). Do not include docstrings, comments, or extra text. "
"Consider adversarial but spec consistent cases (empties, all equal, already sorted, duplicates, extreme values). "
"Do not hardcode to the given assertion values. "
"Output ONLY the function (and any tiny helper it calls). No extra text.\n"
"<python code here>"
)
```