TeamB2B at BLP-2025 Task 2: BanglaForge: LLM Collaboration with Self-Refinement for Bangla Code Generation

Mahir Labib Dihan, Sadif Ahmed, Md Nafiu Rahman
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology (BUET)
Dhaka, Bangladesh
{mahirlabibdihan, ahmedsadif67, nafiu.rahman}@gmail.com

Abstract

Bangla is a low-resource language for code generation, lacking large-scale annotated datasets and tools to transform natural language specifications into executable programs. This makes Bangla-to-code generation a challenging task requiring innovative solutions. To address this, we introduce **BanglaForge**, a novel framework for generating code from Bangla function descriptions. BanglaForge leverages a retrieval-augmented dual-model collaboration paradigm with self-refinement, combining in-context learning, llm-based translation, systematic prompt engineering, and iterative self-refinement based on execution feedback, where a coder generates initial solutions and a reviewer enhances them for robustness. On the **BLP-2025 Bangla Code Generation** benchmark, BanglaForge achieves a competitive Pass@1 accuracy of **84.00%**, demonstrating the effectiveness of retrieval, model collaboration, and self-refinement for low-resource Bangla code generation.

1 Introduction

Large language models (LLMs) have shown strong capabilities in code generation, where natural language descriptions are automatically transformed into executable programs. Models such as Codex, CodeT5, and StarCoder, trained on large-scale codetext corpora, can produce syntactically valid and semantically correct solutions, performing well on benchmarks like HumanEval (Chen et al., 2021). These advances reduce the gap between human intent and code, making programming more accessible. However, most existing systems are designed for English inputs, leaving low-resource languages underserved. Models often struggle with informal structures, domain-specific terms, and semantic nuances, resulting in incorrect or brittle outputs.

We introduce **BanglaForge**, a framework for generating executable code from Bangla task descriptions. Each input is represented as a triple: the Bangla description, its English translation, and unit test assertions. This structure leverages the models stronger English understanding while retaining Bangla context. BanglaForge combines retrieval-augmented prompting, iterative self-refinement with execution feedback, and a dual-model coderreviewer pipeline. Our system achieves a **Pass@1 accuracy of 84%** on **BLP-2025 Bangla Code Generation Benchmark** (Raihan et al., 2025c), demonstrating the potential of practical low-resource code generation.

Our contributions can be summarized as follows:

- A retrieval-augmented few-shot prompting approach using TF-IDF to select relevant Bangla–Python pairs, improving in-context learning despite limited labeled data.
- A LLM-based translation component that translates Bangla instructions into English with the help of a glossary to enable accurate cross-lingual code generation.
- An iterative self-refinement protocol that leverages execution feedback to detect and correct errors across refinement cycles.
- A dual-model architecture where a generator model focuses on functional correctness and a reviewer model enhances robustness, style, and coverage of edge cases.

We release our implementation of BanglaForge at https://github.com/mahirlabibdihan/BanglaForge to facilitate reproducibility and further research.

2 Related Works

Research in Bangla NLP has evolved from early word embeddings to specialized LLMs. Initial efforts such as BnVec introduced embeddings

Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025), pages 642–655

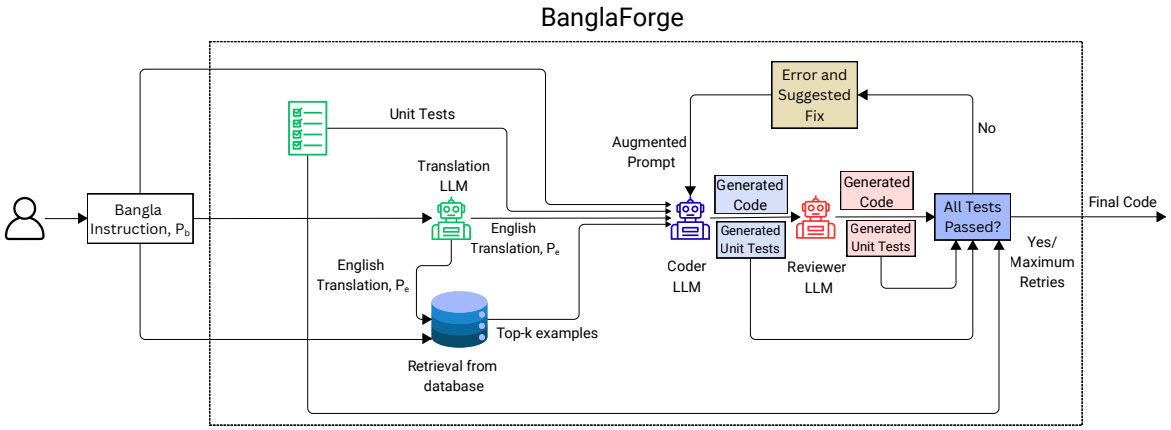

Figure 1: Workflow of the proposed **BanglaForge** framework. A Bangla instruction (P_b) is translated into English (P_e) and, together with unit tests, used to retrieve top-k bilingual examples. The **Coder LLM** then generates Python code and additional test cases. The **Reviewer LLM** validates, refines, and re-prompts upon errors until all tests (original and generated) are passed, yielding the final code.

like fastText, Word2Vec, and GloVe trained on diverse corpora, with customized fastText outperforming multilingual baselines in classification tasks (Kowsher et al., 2021, 2022; Mojumder et al., 2020). Recent advances include Bangla LLMs and benchmarks such as TigerCoder (Raihan et al., 2025b) and BanglaByT5 (Bhattacharyya and Bhattacharya, 2025), which advanced code generation and tokenization strategies. However, existing work largely focuses on pretraining and benchmarking without complete generation pipelines. Our work addresses these gaps by introducing retrieval-augmented prompting, iterative self-refinement, and a dual generator-reviewer design. A detailed discussion is provided in Appendix A.

3 Dataset

We build on the resources introduced for Bangla code generation across recent shared tasks and benchmarks. Our dataset comes from the Bangla Code Generation shared task (Task 2) at BLP-2025 (Raihan et al., 2025c), where the objective is to translate Bangla natural language programming prompts into Python functions that satisfy hidden unit tests. The dataset is distributed through an official starter kit[1], which also provides baseline code and evaluation scripts.

Each entry is a JSON object containing four fields: an `id`, a Bangla instruction describing the task, a `response` field with the reference Python implementation (training only), and a `test_list` field of assert-based unit tests.

Field	Value
id	1
instruction	প্রদত্ত অ্যারে থেকে সমান উপাদান জোড়া গণনা করার জন্য একটি পাইথন ফাংশন লিখুন।
response	`def count_equal_pairs(arr): # Implementation of the function`
test_list	`assert count_equal_pairs([1,2,2,3]) == 1`

Figure 2: Example data point

Split	Purpose	Size
Trial	Initial experiments	74
Development	Validation	400
Test	Final evaluation	500

Table 1: Dataset Split Statistics for Bangla Code Generation

For development and testing, we adopt two external Bangla code generation benchmarks. The **mHumanEval-Bangla** dataset (Raihan et al., 2025a), a Bangla extension of HumanEval, is used during the development phase, enabling programmatically testable evaluation on held-out prompts. The **MBPP-Bangla** dataset (Raihan et al., 2025b), adapted from MBPP as part of the TigerCoder framework, is used during both development and test phases, providing diverse programming problems in Bangla with associated unit tests.

4 Methodology

We propose **BanglaForge**, a retrieval-augmented dual-LLM framework for generating Python code from Bangla natural language specifications.

[1] https://noshinulfat.github.io/blp25_code_generation_task/#/get-started

The system tackles low-resource code generation through structured prompt design, bilingual translation, example retrieval, and a two-stage generation-review process involving a *Coder LLM* and a *Reviewer LLM*. Together, these components ensure both functional correctness and stylistic reliability, even in underrepresented languages like Bangla. An overview of the complete workflow is shown in Figure 1 and in Algorithm 1 (Appendix). Each stage is described in detail below.

4.1 Problem Formulation and Input Representation

Each task in the dataset consists of a Bangla instruction P_b and its corresponding public unit tests $T = \{t_1, \ldots, t_n\}$. To enable code synthesis, the instruction is translated into English using a translation model equipped with a controlled glossary for mathematical and algorithmic terms (e.g., GCD, LCM, sum). The glossary is curated by the authors which was motivated from the provided dataset and commonly seen technical terms in code related works. The translated instruction P_e retains the semantic fidelity of the Bangla instruction while ensuring syntactic clarity for code generation. The systems objective is to synthesize a Python function f such that all $t_i \in T$ are satisfied given the constraints in P_b. Function prototypes are normalized to valid Python syntax, aligning argument and return types with unit test definitions.

4.2 Retrieval-Augmented Example Selection.

To enhance contextual understanding, both Bangla and English task descriptions are used to retrieve semantically similar solved examples from a bilingual database $\mathcal{D} = \{(p_i^b, p_i^e, c_i, T_i)\}_{i=1}^{N}$. Each entry contains the Bangla and English prompts, the reference code (c_i), and associated test cases (T_i). Both P_b and P_e are embedded using TF-IDF unigram bigram representations. We chose TF-IDF due to its high computational efficiency and strong performance on smaller datasets, as dense retrievers typically require a large training corpus to be effective (Arabzadeh et al., 2021). For our task, TF-IDF's strength in matching exact, high-signal technical keywords (e.g., GCD," factorial") is paramount. This lexical precision provides a fast and more reliable baseline for retrieving analogous code problems than a dense model's generalized semantic understanding (Karpukhin et al., 2020). The top-k examples (typically $k = 5$) are selected and inserted into the prompt as few-shot exemplars. For experiments on the Development set, the database $\mathcal{D}$ consists of the Trial set. For experiments on the Test set, we use the combined Trial+Development sets as the database. This bilingual, retrieval-augmented setup enables contextual grounding and helps the model capture problem-solving patterns from similar tasks. The retrieved example format is provided in Appendix C.

4.3 Stage 1: Code Generation by Coder LLM.

The *Coder* LLM receives a composite input consisting of the Bangla instruction P_b, English translation P_e, the retrieved top-k example pairs (p_i^b, p_i^e, c_i, T_i), and the provided unit tests T. Based on this augmented prompt, the Coder LLM generates a Python code candidate c_0 intended to satisfy T, and additional synthetic test cases T_c designed to cover potential edge or missing cases. This stage focuses on functional code generation guided by contextual analogies from retrieved examples. The output (c_0, T_c) is then passed to the Reviewer LLM for refinement. The detailed prompt for Coder LLM is provided in Appendix C.

4.4 Stage 2: Code Review and Refinement by Reviewer LLM.

The *Reviewer* LLM acts as a validator and refiner. It takes as input the code and test cases generated by the Coder LLM along with the original task description and unit tests. Its responsibilities include running static and logical checks on c_0, correcting syntax or runtime issues, improving variable naming, structure, and input validation, generating an additional set of refined unit tests T_r to ensure covering edge cases. If any error or inconsistency is detected, the Reviewer LLM suggests an explicit fix and the process is repeated up to a maximum of M iterations ($M = 5$). The detailed prompt for Reviwer LLM is provided in Appendix C.

4.5 Iterative Self-Refinement Protocol.

The refinement loop is formally defined as: $c_{i+1} = \mathcal{R}(c_i, e_i, \mathcal{P})$, where $\mathcal{R}$ denotes the Reviewer LLM, e_i is the detected error, and $\mathcal{P}$ represents the augmented prompt containing feedback. The cycle continues until all test casesoriginal (T), coder-generated (T_c), and reviewer-generated (T_r)are successfully passed, or until the retry limit M is

Model	Few Shot	# Examples	Translation	# Unit Tests	Pass@1
Dev Set					
Gemma-1B	N/A	0	No	0	27.25%
GPT-OSS-20B	Manual	3	No	0	60.25%
GPT-OSS-20B	Manual	5	No	0	61.25%
DeepSeek-R1-Llama-70B	Manual	5	Yes	0	57.75%
Gemini-2.0-Flash	Manual	3	Yes	0	60.00%
Gemini-2.0-Flash	Manual	5	Yes	0	62.50%
Lg Exaone Deep 32B	Manual	5	Yes	1	85.25%
Lg Exaone Deep 32B	Manual	5	Yes	3	94.25%
Lg Exaone Deep 32B	RAG (Trial)	5	Yes	3	95.50%
Test Set					
Lg Exaone Deep 32B	RAG (Trial+Dev)	5	Yes	1	80.60%
Gemini-2.5-Pro	**RAG (Trial+Dev)**	**5**	**Yes**	**1**	**84.00%**

Table 2: Pass@1 accuracy of models on the BLP-2025 Development and Test sets.

reached. This multi-level testing ensures that the final solution generalizes beyond the given test cases. The errors and suggested fixes are provided in Appendix C.

5 Experiment

5.1 Evaluation Metrics

We evaluate performance using the Pass@1 accuracy metric, which measures the proportion of problems solved correctly in the first iteration. This metric provides a clear and direct assessment of the systems accuracy in solving problems without requiring further refinements.

5.2 Models

We evaluate several large language models (LLMs) for Bangla code generation. The models tested on the Development set include **Gemma-1B** (Gemma, 2024), **GPT-OSS-20B** (Initiative, 2024), **DeepSeek-R1-Llama-70B** (AI, 2025), **Gemini-2.0-Flash** (DeepMind, 2024), and **Lg Exaone Deep 32B** (Research, 2024), with different prompting strategies and unit-test settings. For the final evaluation on the Test set, we select **Lg Exaone Deep 32B** (Research, 2024) and **Gemini-2.5-Pro** (DeepMind, 2025) under their best-performing configurations within a retrieval-augmented dual-stage pipeline.

5.3 Results

We evaluate our system on the BLP-2025 Bangla code generation benchmark. Our experiments are conducted in two stages: first on the Development set to explore different models and prompting strategies, and then on the Test set to report final results. Table 2 presents the Pass@1 accuracy for various models and configurations across both sets.

The development set results reveals that small-scale models such as **Gemma-1B** achieve only 27.25% Pass@1, underscoring the challenge of Bangla-to-code translation without contextual guidance. Larger open-source models like **GPT-OSS-20B** shows improvements (60.25-61.25%) under few-shot prompting, though performance gains taper off with additional in-context examples. Introducing translation-based prompting further improves comprehension of Bangla instructions, as seen with **DeepSeek-R1-Llama-70B** (57.75%) and **Gemini-2.0-Flash** (60-62.5%).

A major performance leap is observed with the **Lg Exaone Deep 32B** model, which combines translation and lightweight unit-test feedback. Accuracy rises from 85.25% with one visible test to 94.25% with three tests, highlighting the benefit of guided reasoning through intermediate validation. When enhanced with our RAG pipeline on the trial set, the model achieves 95.5% Pass@1 on the development benchmarkdemonstrating consistent improvements through contextual retrieval and refinement.

On the held-out test set, the RAG-augmented **Lg Exaone Deep 32B** achieves 80.6% Pass@1, while the more recent **Gemini-2.5-Pro** model further pushes performance to **84.0%**. These results confirm that retrieval augmentation combined with multilingual comprehension yields robust generalization across unseen Bangla programming tasks.

6 Conclusion

In this paper, we presented a retrieval-augmented dual-model framework for generating Python code

from Bangla instructions. Combining structured prompting, iterative self-refinement, and a generator-reviewer design, our system achieved Pass@1 accuracy of 84% on the BLP-2025 benchmark. The approach consistently outperforms baselines, showing the effectiveness of retrieval augmentation and feedback-driven refinement for low-resource code generation. Future work will expand the framework to other languages and incorporate reinforcement-based refinement. Additionally, improvements in RAG corpus and Bangla-to-English translation quality are expected to further enhance the overall performance of the pipeline.

7 Limitations

While BanglaForge demonstrates strong performance on the BLP-2025 Bangla code generation benchmark, several limitations remain. First, the system relies heavily on high-quality bilingual translation; inaccuracies in Bangla-to-English mapping or glossary coverage can propagate errors to the generation stage. Second, the retrieval component depends on TF-IDF, which captures lexical overlap but may miss deeper semantic similarities, especially in complex algorithmic prompts. Third, the framework assumes well-structured Bangla input; informal phrasing or dialectal variations could reduce translation fidelity and retrieval relevance. Additionally, self-refinement cycles are limited to a fixed number of iterations and do not incorporate adaptive stopping or learning from prior refinements. Finally, since the dataset itself originates from machine-translated English sources, true Bangla-native problem framing and linguistic diversity remain under-represented. Future work should explore human-curated datasets, semantic retrieval models, and reinforcement-based refinement to address these limitations.

References

DeepSeek AI. 2025. Deepseek-r1: Reasoning models built on llama-70b. https://github.com/deepseek-ai/DeepSeek-R1. Accessed: 2025-10-05.

Negar Arabzadeh, Xinyi Yan, and Charles L. A. Clarke. 2021. Predicting efficiency/effectiveness trade-offs for dense vs. sparse retrieval strategy selection. *Preprint*, arXiv:2109.10739.

Abhik Bhattacharjee, Tahmid Hasan, Wasi Uddin Ahmad, Kazi Samin Mubasshir, Md Saiful Islam, 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. Association for Computational Linguistics.

Pramit Bhattacharyya and Arnab Bhattacharya. 2025. Banglabyt5: Byte-level modelling for bangla. *arXiv preprint arXiv:2505.17102*. Cs.CL.

Pramit Bhattacharyya, Joydeep Mondal, Subhadip Maji, and Arnab Bhattacharya. 2023. Vacaspati: A diverse corpus of bangla literature. *arXiv preprint arXiv:2307.05083*.

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*.

Google DeepMind. 2024. Gemini 2.0: A family of multimodal language models. https://deepmind.google/technologies/gemini/. Accessed: 2025-10-05.

Google DeepMind. 2025. Gemini 2.5 pro: Next-generation multimodal reasoning model. https://deepmind.google/technologies/gemini-2-5/. Accessed: 2025-10-05.

Team Gemma. 2024. Gemma: Open models by google deepmind. https://deepmind.google/technologies/gemma/. Accessed: 2025-10-05.

Mohammad Mehadi Hasan, Fatema Binte Hassan, Md Al Jubair, Zobayer Ahmed, Sazzatul Yeakin, and Md Masum Billah. 2025. Bangla bert for hyperpartisan news detection: A semi-supervised and explainable ai approach. *arXiv preprint arXiv:2507.21242*.

Open Source Science Initiative. 2024. Gpt-oss: An open-source family of general-purpose large language models. https://huggingface.co/gpt-oss. Accessed: 2025-10-05.

Mohsinul Kabir, Mohammed Saidul Islam, Md Tahmid Rahman Laskar, Mir Tafseer Nayeem, M Saiful Bari, and Enamul Hoque. 2023. Benllmeval: A comprehensive evaluation into the potentials and pitfalls of large language models on bengali nlp. *arXiv preprint arXiv:2309.13173*.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781. Association for Computational Linguistics.

Md. Kowsher, Md. Shohanur Islam Sobuj, Md. Fahim Shahriar, Nusrat Jahan Prottasha, and Mohammad Shamsul Arefin. 2022. An enhanced neural word embedding model for transfer learning. *Applied Sciences*, 12(6):2848.

Md. Kowsher, Md. Jashim Uddin, Anik Tahabilder, Nusrat Jahan Prottasha, Mahid Ahmed, K. M. Rashedul Alam, and Tamanna Sultana. 2021. Bnvec: Towards the development of word embedding for bangla language processing. *International Journal of Engineering and Technology*, 10(2):95–102.

Hemal Mahmud and Hasan Mahmud. 2024. Enhancing sentiment analysis in bengali texts: A hybrid approach using lexicon-based algorithm and pretrained language model bangla-bert. *arXiv preprint arXiv:2411.19584*.

Pritom Mojumder, Md. Faruque Hossain, Mahmudul Hasan, and K. M. Azharul Hasan. 2020. A study of fasttext word embedding effects in document classification in bangla language. In *ICONCS 2020*, pages 441–453. Springer, Cham.

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 and Marcos Zampieri. 2025. Tigerllm: A family of bangla large language models. *arXiv preprint arXiv:2503.10995*.

LG AI Research. 2024. Exaone deep 32b: Large-scale multilingual foundation model by lg ai research. https://www.lgresearch.ai/model/exaone-deep-32b. Accessed: 2025-10-05.

Sagor Sarker. 2020. Bangla-bert: A pretrained bert model for bengali. https://github.com/sagorbrur/bangla-bert.

Sheikh Shafayat, Quamran Hasan H. M. Minhajur Rahman Chowdhury Mahim, Rifki Afina Putri, James Thorne, and Alice Oh. 2024. Benqa: A question answering and reasoning benchmark for bengali and english. *arXiv preprint arXiv:2403.10900*.

A Related Works

The trajectory of research in Bangla NLP has shifted from foundational embeddings and lightweight classification models to full-fledged Bangla LLMs and, more recently, toward modular architectures that integrate retrieval and feedback. In the early days, emphasis was placed on crafting vector representations tailored to Banglas morphological richness and vocabulary distribution. The BnVec project, for instance, introduced Bangla-specific fastText, Word2Vec, and GloVe embeddings that placed importance on vocabulary coverage and representation quality (Kowsher et al., 2021). Later work showed that embeddings trained on Bangla corpora outperform multilingual embedding baselines in text classification and related tasks (Kowsher et al., 2022; Mojumder et al., 2020). Meanwhile, the Vacaspati corpus and derived models such as Vac-FT and Vac-BERT demonstrated that diversifying corpus domains and scaling data can boost embedding and language model utility beyond standard fastText baselines (Bhattacharyya et al., 2023).

As the field progressed, researchers began developing Bangla-centric pretrained language models for both understanding and generation. A notable early example is BanglaBERT, introduced by Bhattacharjee et al., which is a BERT (ELECTRA-discriminator)style model pretrained on a 27.5 GB Bangla corpus (Bangla2B+) and evaluated on a suite of Bangla NLU benchmarks that include classification, NLI, NER, and QA tasks under the BLUB benchmark (Bhattacharjee et al., 2022). BanglaBERT outperforms multilingual baselines on those tasks, showing that language-specific pretraining brings tangible gains in low-resource settings. Building on that, more recent works such as enhanced sentiment analysis pipelines fine-tune and hybridize BanglaBERT with lexicon/rule components (Mahmud and Mahmud, 2024), or apply it for domain tasks like hyperpartisan news detection with semi-supervised learning and explainability (Hasan et al., 2025). Alongside, general-purpose monolingual models for Bangla (e.g. Bangla-Bert-Base by Sagor Sarker et al.) have also been proposed and used across classification and NER tasks (Sarker, 2020).

Complementing these, newer model lines push

toward generative and evaluation capacities in Bangla. TigerLLM, is a suite of Bangla LLMs trained on large Bangla corpora and shows gains over prior open and proprietary models across Bangla benchmarks (Raihan and Zampieri, 2025). In the programming domain, TigerCoder introduces dedicated Bangla code LLMs (1B and 9B) and the MBPP-Bangla benchmark, reporting 11 to 18 % Pass@1 improvement over multilingual baselines (**?**). In evaluation, BenLLMEval provides a wide evaluation of off-the-shelf LLMs (GPT-3.5, LLaMA-2, Claude, etc.) on Bangla tasks (summarization, QA, paraphrase, classification), revealing substantial performance gaps in zero-shot settings (Kabir et al., 2023). The BEnQA benchmark offers parallel BengaliEnglish QA and reasoning tasks derived from exam questions; it shows that chain-of-thought prompting helps reasoning tasks and that including English context can improve performance in Bengali (Shafayat et al., 2024).

Despite advances in modeling, most existing works treat the language model as a single-step generator without built-in mechanisms for grounding, correction, or iteration. In broader NLP and code domains, however, robust generation systems increasingly incorporate retrieval-augmented architectures (e.g. RAG), cross-lingual retrieval for low-resource grounding, retrievalaugmented data augmentation (RADA), multi-stage or hierarchical retrieval (e.g. for code), and iterative refinement via coderreviewer loops or test-driven feedback. These techniques have been shown to reduce hallucination, improve factual grounding, and correct logical or syntactic errors in generated outputs.

These retrieval, review, and iteration techniques remain underexplored in Bangla and especially in Banglacode generation. In this work, we explicitly address that gap by combining Bangla-focused models (e.g. TigerLLM, TigerCoder) with retrieval-based prompt augmentation, a separate reviewer module, and iterative self-refinement. This hybrid design aims to boost reliability and real-world usability in Bangla code generation systems.

B Experimental Setup

All models were configured with the following default generation parameters: `temperature` = 0.7, `top_p` = 0.9, and `max_new_tokens` = 1024. Each query generated `n` = 1 output sample per decoding pass.

C Model Prompts

This section details the prompts used in our **Bangla2Py** framework. The prompts are designed to guide the Large Language Models (LLMs) through the code generation, refinement, and review stages. Placeholders like `{instruction}` are dynamically populated by the pipeline.

C.1 Coder Model Prompts

The **Coder LLM** is the first stage of our system and is responsible for writing the initial Python solution. It receives both the Bangla task description and its English translation, along with a set of retrieved examples and the provided unit tests. The coder's system prompt clearly defines its role as a Python code generator and instructs it to produce only executable code no explanations or comments (Figure 3). The main task prompt includes several few-shot examples followed by the current problem. Each example shows the task instruction (in both languages), the correct solution, and unit tests (Figure 4). If the generated code fails any test, the coder receives a short feedback message describing the error type (e.g., syntax error, timeout, or assertion failure) along with a fix hint (Figure 5). It then regenerates an improved version in the next iteration. This feedback-guided prompting helps the coder LLM progressively refine its output and produce cleaner, test-ready code with a built-in `main()` function for validation.

C.2 Reviewer Model Prompts

The **Reviewer LLM** acts as the second stage and takes the code produced by the coder, along with the original BanglaEnglish instructions and all test cases (both given and generated). Its prompt defines the role of a code reviewer focusing on improving correctness, readability, and coverage of edge cases without changing the function signature. The reviewer checks for logical mistakes, inefficient loops, missing validations, or weak test coverage. It then returns a refined version of the code, adds extra corner-case tests, and ensures the final version passes both visible and hidden cases. If errors are still detected, the reviewer can repeat this process with updated feedback until all tests are passed or a retry limit is reached.

C.3 Few-Shot Example Template

To help both LLMs generalize better, we use retrieval-augmented few-shot examples in the

System Prompt for Coder Model

You are a Python programming assistant.
The user will provide a function stub where the original docstring is written in Bangla with a translated version and a unit test case.
Your task is to read the Bangla + English (Translated) docstring and the unit test case, understand the requirement, function parameters, return type, and complete the function implementation in Python.
Your response must be in English, not Bangla, and must only contain valid Python code. Do not add explanations, comments, or extra text. Just return the code solution.
Your main task is to carefully read the Bangla + English (Translated) docstring and the unit test case and infer:
1. The expected number of parameter and their types
2. The expected return type
3. The correct implementation logic
Important guidelines:
1. The function signature is already provided in the instruction. Implement the function as specified.
2. Include a **main function** (using `def main():`) in your code that contains necessary unit tests or example calls to validate your function.
3. Do **not** call `main()` anywhere in your code. This will be executed externally.
4. Try to keep the code as simple as possible.
5. Your response should contain only one python block enclosed in a code block like:

```
“‘python
# your code here
“‘
```

Figure 3: System prompt for the coder model.

prompts. The system retrieves the top-k most similar problems from the bilingual database using both Bangla and English task texts. Each example includes:

- The Bangla and English instructions,
- The reference Python solution, and
- The corresponding unit tests.

These examples are formatted in a consistent template and placed before the current task in the prompt (see Figure 8). This structure lets the models recognize patterns in how Bangla instructions map to Python logic, guiding them to produce correct and well-structured code even for unseen problems.

D Error Refinement

The iterative feedback follows an augmentation protocol as outlined in Table 3.

Error Type	Feedback Hint / Guidance
Syntax Error	Check indentation, missing colons, or parentheses; ensure valid Python syntax.
Runtime Error	Ensure variables are initialized and referenced correctly; verify data types and control flow.
Assertion Failure	Compare expected vs. actual outputs; review logical steps and boundary conditions.
Timeout Error	Optimize loops or recursion; include clear termination conditions.
System Exit	Avoid abrupt exits; allow the program to complete execution normally.

Table 3: Error categories and corresponding feedback hints used in prompt augmentation.

E Ablation Study

To analyze the contribution of each component in **BanglaForge** , we perform ablation experiments using the **Lg Exaone Deep 32B** model on the BLP-2025 development set. The best full configuration achieves a Pass@1 accuracy of **95.5%**, and all reported variations are measured relative to this setting. Each ablation disables or modifies a single module while keeping the rest of the pipeline fixed.

E.1 Effect of English Translation

We first evaluate the role of bilingual translation. When the system relies solely on Bangla instructions without their English counterparts, comprehension drops significantly. The LLM often fails to parse algorithmic phrases and control keywords written in Bangla. As shown in Table 4, removing the translation stage reduces Pass@1 accuracy by

Main Prompt Template for Coder

```
{examples}
» Your Task
> Instruction
```python
def {function_call}:
"""{instruction}"""
"""Translated: {instruction_en}"""
"""{docstring}"""
```
Now complete the python code for the function '{function_name}' and add a
'main' function with unit tests. You should use the 'check' function for unit tests,
which is helpful for debugging. For example:
```python
def {function_call}:
# Your code

def check(test_id, test_val, expected):
assert test_val == expected, f"Test {test_id}: Expected {expected}, got
{test_val}"

def main():
{check_example}
# Add more unit tests
```
```

Figure 4: Main prompt template for the coder, which includes few-shot examples and the current task.

Failed Attempt Feedback Template

```
» Last failed code
> Response:
{last_response}
> Error:
{last_error}
> Suggested Fix:
{fix_instructions}
```

Figure 5: Template for providing feedback to the coder model after a failed execution attempt. This is appended to the main prompt during the self-refinement loop.

nearly 22 percent, confirming that current models still struggle to reason directly over Bangla-only text.

Setting	Pass@1 (%)
Full Model (Bangla + English)	95.5
Bangla Only	73.6

Table 4: Effect of English translation on Pass@1 accuracy (Lg Exaone Deep 32B, Dev Set).

E.2 Effect of Glossary-based Translation

We also analyze the impact of the controlled translation glossary used for mathematical and algorithmic terms. Without this glossary, the translation model often produces inconsistent or incorrect terminology, confusing the Coder during reasoning. As shown in Table 5, removing the glossary results in a notable performance drop of over 7 points, confirming that LLMs struggle to translate some Bangla words properly, leading to incorrect function generation.

Setting	Pass@1 (%)
With Glossary (Full Model)	95.5
Without Glossary	88.2

Table 5: Effect of using the controlled translation glossary.

E.3 Effect of Feedback Loop

Next, we disable the iterative self-refinement mechanism. Without execution feedback or re-prompting, the model cannot correct runtime or

System Prompt for Reviewer Model

You are a Python code reviewer and programming assistant.
The user will provide a function stub or implementation where the original docstring is written in Bangla with a translated English version, along with unit test cases.
Your task is to:
1. Do not alter the given function signature.
2. Review the implementation for correctness, clarity, efficiency, and robustness.
3. Refactor or improve the implementation if needed, but the function signature must remain identical.
4. Ensure the function works correctly not only for the provided tests but also for **hidden test cases** and **corner cases** (e.g., empty inputs, boundary values, invalid values, very large inputs).
5. Add a `main` function with unit tests that use the provided `check` function.
6. Include the given test cases and add additional edge/corner case tests that a hidden evaluator might check.
7. Do not add explanations, comments, or extra text. Just return the code solution.

Important guidelines:
1. The function signature is already provided. Implement or refactor the function as specified.
2. Include a **main function** (using `def main():`) that contains both the given unit tests and extra corner/hidden-case tests you find necessary.
3. Do **not** call `main()` anywhere in your code. It will be executed externally.
4. Keep the code clean, correct, and as simple as possible while ensuring it passes all tests, including edge and hidden cases.
5. Your response must be only one valid Python code block enclosed in triple backticks:

```python
# your code here
```

Figure 6: System prompt for the reviewer model.

Main Prompt Template for Reviewer

» Your Task
The following function is already implemented:

```
"""{instruction}"""
"""Translated: {instruction_en}"""
```python
{existing_code}
```

Figure 7: Main prompt template for the reviewer model.

logic errors, leading to a steep performance drop. Table 6 shows that accuracy declines by more than 25 percent, emphasizing that feedback-driven correction is vital for reliable synthesis.

Setting	Pass@1 (%)
Full Model	95.5
Without Feedback Loop	69.8

Table 6: Impact of feedback-driven refinement.

E.4 Effect of Reviewer LLM

To measure the Reviewers contribution, we bypass the second-stage review and directly execute the Coder output. Although the generated code remains mostly functional, it lacks stylistic polish and robustness on edge cases. Table 7 shows a moderate decline of about 5 percent, verifying that the Reviewer mainly improves coverage and reliability.

Setting	Pass@1 (%)
Full Model	95.5
Without Reviewer	90.4

Table 7: Effect of disabling the Reviewer LLM.
```

Few-shot Example Template

» Example {idx}:
> Instruction
```python
def {function_call}:
"""{instruction}"""
"""Translated: {instruction_en}"""
"""{docstring}"""
```
> Solution
```python
{solution}

def check(test_id, test_val, expected):
assert test_val == expected, f"Test {test_id}: Expected {expected}, got {test_val}"

def main():
{test_main}
```

Figure 8: Template for formatting each of the k-nearest examples for retrieval-augmented generation.

System Prompt for Translator Model

Translate the following Bangla Python Code Instruction to English and only return the English translation. Do not change the example function and parameter names and only update the function parameter types and return variable types of Example function prototype to actual python syntax based on the provided unit test. Do not give the full code implementation. Just give the updated prototype.

Use the following glossary for translation: {glossary}

Unit Test: {test}

Figure 9: System prompt for the translator model.

E.5 Number of Feedback Iterations

We vary the maximum feedback iterations (M) to observe convergence behavior. As shown in Table 8, fewer iterations significantly reduce success rate since many tasks require multiple refinement cycles. Beyond five iterations, improvements saturate.

Max Iterations (M)	**Pass@1 (%)**
1	84.1
3	92.4
5	95.5
7	95.5

Table 8: Effect of limiting feedback iterations (M).

E.6 Effect of Retrieval Augmentation (RAG)

We compare our retrieval-augmented setup against a manually few-shot configuration. In the manual setup, the examples are fixed and not selected dynamically based on similarity, while the RAG variant retrieves the top-k relevant bilingual examples for each new task. As Table 9 shows, retrieval augmentation provides a small but consistent improvement of about 1.3 points, indicating that example relevance matters more than sheer quantity.

Setting	**Pass@1 (%)**
With RAG (Full Model)	95.5
Manual Few-shot (Fixed Examples)	94.2

Table 9: Comparison between manual few-shot and RAG-based prompting.

E.7 Number of Retrieved Examples (k)

Finally, we study the impact of the retrieval context size. As Table 10 shows, removing examples ($k = 0$) severely hampers the models grounding ability, dropping performance below 70%. Accuracy improves steadily up to $k = 5$, after which

Bangla	English
গ.সা.গু / গরিষ্ঠ সাধারণ গুণনীয়ক	GCD
ল.সা.গু / লঘিষ্ঠ সাধারণ গুণিতক	LCM
যোগফল	sum
বিয়োগফল	difference
গুণফল	product
ভাগফল	quotient
মডুলাস / মডুলো	modulus / modulo
শক্তি / ঘাত	power
মূল	root
বর্গমূল	square root
ঘনমূল	cube root
অবশিষ্ট	remainder
মৌলিক সংখ্যা / প্রাইম নাম্বার	prime number
যৌগিক সংখ্যা / কম্পোজিট নাম্বার	composite number
ফাইল / নথি	file
সময় জটিলতা	time complexity

Bangla	English
সমান	equal
সমান নয়	not equal
ছোট	less than
বড়	greater than
সমান বা ছোট	less than or equal to
সমান বা বড়	greater than or equal to
বিজোড়	odd
যুগ্ম	even
গুণনীয়ক	factor
গুণিতক	multiple
ভাজক / ডিভাইসর	divisor
গুণ	multiply
ভাগ	divide
যোগ	add
বিয়োগ	subtract
স্থান জটিলতা	space complexity
অ্যালগরিদম / পদ্ধতি	algorithm

Bangla	English
যদি / ইফ	if
অন্যথা / এলস	else
যদি না / এলস ইফ	else if
লুপ / লুপ করো	loop
যতক্ষণ / হোয়াইল	while
জন্য / ফর	for
থামাও / ব্রেক	break
অগ্রসর হও / কন্টিনিউ	continue
ফেরত / রিটার্ন	return
তালিকা / লিস্ট	list
অ্যারে	array
স্ট্যাক / পাইল	stack
কিউ / পংক্তি	queue
ডেক / ডিক	deque
সেট / সমষ্টি	set
টেস্টকেস / পরীক্ষা	test case
ইনডেক্স / সূচক	index
ইটারেট / পুনরাবৃত্তি করো	iterate
পুনরাবৃত্তি / রিকারশন	recursion

Bangla	English
ম্যাপ	map
ডিকশনারি / অভিধান	dictionary
হ্যাশম্যাপ	hashmap
গ্রাফ / ছক	graph
ট্রি / গাছ	tree
বাইনারি ট্রি / বাইনারি গাছ	binary tree
হীপ	heap
প্রায়োরিটি কিউ / অগ্রাধিকার কিউ	priority queue
ডিএফএস / গভীরতা প্রথম অনুসন্ধান	DFS
বিএফএস / প্রস্থ প্রথম অনুসন্ধান	BFS
সাজাও / বাছাই	sort
ক্রমবর্ধমান	ascending
ক্রমহ্রাসমান	descending
অনুসন্ধান / সার্চ	search
বাইনারি সার্চ / বাইনারি অনুসন্ধান	binary search

Bangla	English
লিনিয়ার সার্চ / রৈখিক অনুসন্ধান	linear search
স্ট্রিং / পাঠ্য	string
উল্টা / বিপরীত	reverse
সাবস্ট্রিং / উপস্ট্রিং	substring
সংযুক্ত / সংযোজন	concatenate
দৈর্ঘ্য / আয়তন	length
অক্ষর / চরিত্র	character
বড় হাতের	uppercase
ছোট হাতের	lowercase
প্যালিনড্রোম / সমপাঠ্য	palindrome
ইনপুট / প্রবেশ	input
আউটপুট / বাহির	output
প্রিন্ট	print
পড়ো / পাঠ	read
লিখো	write

Figure 10: Glossary for the translation prompt

marginal gains diminish due to context saturation.

Number of Examples (k)	Pass@1 (%)
0 (No Examples)	69.3
3	88.9
5 (Full)	95.5
7	94.7

Table 10: Effect of retrieved example count (k).

E.8 Comprehensive Summary

Table 11 consolidates all variants. The results confirm that English translation and the feedback loop contribute the largest performance boosts, while the glossary, reviewer, and RAG components further improve consistency, code quality, and generalization.

F Algorithm

Algorithm 1 shows the pseudocode of our pipeline.

G Failure Cases and Dataset Limitations

The dataset for Bangla-to-Python code generation was created by translating existing English datasets MBPP (Mostly Basic Python Problems) and HumanEval into Bangla using machine translation. While this approach enables rapid dataset construction, it introduces several limitations that affect both dataset quality and model performance.

G.1 Semantic and Syntactic Translation Errors

Machine translation occasionally produces Bangla sentences that are grammatically incorrect or semantically ambiguous. Such translations may hinder a models ability to correctly interpret the input and generate the intended Python code. For example:

- The English adjective “even” was translated as এমনকি instead of the more contextually accurate জোড় in cases where *even* refers to parity in numbers. This leads to semantic confusion and misinterpretation of the question context.

G.2 Incorrect or Misleading Terminology for Programming Concepts

Programming terms often lack direct equivalents in Bangla. Machine translation systems attempt to

Configuration	Translation	Glossary	Feedback Loop	Reviewer	RAG	Pass@1 (%)
Full BanglaForge Pipeline	Yes	Yes	Yes	Yes	Yes	**95.5**
Without Translation	No	Yes	Yes	Yes	Yes	73.6
Without Glossary	Yes	No	Yes	Yes	Yes	88.2
Without Feedback Loop	Yes	Yes	No	Yes	Yes	69.8
Without Reviewer	Yes	Yes	Yes	No	Yes	90.4
Manual Few-shot (No RAG)	Yes	Yes	Yes	Yes	No	94.2
Fewer Iterations (M = 1)	Yes	Yes	Yes	Yes	Yes	84.1
Fewer Examples (k = 3)	Yes	Yes	Yes	Yes	Yes	88.9
No Examples (k = 0)	Yes	Yes	Yes	Yes	Yes	69.3

Table 11: Comprehensive ablation results on the BLP-2025 development set using Lg Exaone Deep 32B.

Algorithm 1 Algorithm of BanglaForge

Input: BanglaInstruction P_b, PublicUnitTests T
Output: ExecutableCode
$M \leftarrow$ maximum retry limit
$attempt \leftarrow 0$
EnglishInstruction, $P_e \leftarrow$ TranslatorLLM.translate(P_b)
Examples, $E \leftarrow$ Database.retrieveExamples(P_b, P_e)
PromptCoder $\leftarrow$ constructPrompt(P_b, P_e, T, E)
while $attempt < M$ **do**
    $attempt \leftarrow attempt + 1$
    (c, T_c) $\leftarrow$ CoderLLM.generate(PromptCoder)
    PromptReviewer $\leftarrow$ constructReviewPrompt(c, T_c)
    (c_r,T_r) $\leftarrow$ ReviewerLLM.refine(PromptReviewer)
    Result $\leftarrow$ executeCode(c_r, $T \cup T_c \cup T_r$)
    if Result.allTestsPassed **then**
        return c_r
    else
        Feedback $\leftarrow$ generateFeedback(Result.errors)
        PromptCoder $\leftarrow$ updatePromptWithFeedback(PromptCoder, Feedback)
    end if
end while

generate literal translations, but these often fail to capture technical meaning. For example:

- The English term *"Map"* was translated to মানচিত্র (meaning a geographic map in Bangla), instead of referring to *Map* as in a data structure such as HashMap or dictionary. This causes ambiguity, making it challenging for both humans and models to interpret correctly.
- Similarly, terms like *stack*, *queue*, *hashmap*, or *dictionary* may be incorrectly translated, or not translated at all, resulting in inconsistent terminology across the dataset.

G.3 Loss of Context or Intent

Machine translation may fail to preserve the precise context or intent of the original English instructions. Programming problems often rely on subtle nuances, and even small changes in wording can alter the meaning of a problem. This issue is exacerbated when the translated text uses uncommon or unnatural phrasing, reducing clarity for model training.

G.4 Lack of Standardized Technical Vocabulary

Bangla currently lacks standardized technical vocabulary for many programming concepts, leading to inconsistent translations. In some cases, the same English term is translated differently across dataset entries. This inconsistency makes it difficult for a model to reliably learn the intended mapping from Bangla instructions to Python code.

G.5 Impact on Model Performance

These translation-related issues contribute to notable failure cases in Bangla-to-Python code generation. Models trained on such data may misinterpret problem statements, produce incorrect code, or fail to generalize to unseen examples. Addressing these limitations would require:

- Careful human curation of translations for correctness and consistency.
- Development of a standardized Bangla programming lexicon.
- Use of bilingual glossaries to retain original technical terms where necessary.