Overview of BLP-2025 Task 2: Code Generation in Bangla

Nishat Raihan[1], Mohammad Anas Jawad[2], Md Mezbaur Rahman[2], Noshin Ulfat[3], Pranav Gupta[5], Mehrab Mustafy Rahman[2], Shubhra Kanti Karmakar[4], Marcos Zampieri[1]

[1]George Mason University, [2]University of Illinois Chicago, [3]IQVIA, [4]University of Central Florida, [5]Lowe's

mraihan2@gmu.edu

Abstract

This paper presents an overview of the BLP 2025 shared task **Code Generation in Bangla**[1], organized with the BLP workshop co-located with AACL. The task evaluates Generative AI systems capable of generating executable Python code from natural language prompts written in Bangla. This is the first shared task to address Bangla code generation. It attracted 152 participants across 63 teams, yielding 488 submissions, with 15 system-description papers. Participating teams employed both proprietary and open-source LLMs, with prevalent strategies including prompt engineering, fine-tuning, and machine translation. The top `Pass@1` reached 0.99 on the development phase and 0.95 on the test phase. In this report, we detail the task design, data, and evaluation protocol, and synthesize methodological trends observed across submissions. Notably, we observe that the high performance is not based on single models; rather, a pipeline of multiple AI tools and/or methods.

1 Introduction

Despite being the world's fifth most spoken language, Bangla remains underrepresented in Large Language Models (LLMs)—especially for code generation, even as recent advances markedly improve code synthesis (Touvron et al., 2023; Hui et al., 2024a; Team et al., 2025). State-of-the-art (SOTA) models now exceed 90% `Pass@1` on prominent benchmarks such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), spurring adoption in software engineering (Pasquale et al., 2025) and education (Raihan et al., 2025c). Yet these gains disproportionately accrue to a few high-resource languages (Joshi et al., 2020; Blasi et al., 2022; Ahuja et al., 2023; Wang et al., 2023; Raihan et al., 2024).

Bangla—spoken by over 242 million native speakers[2], still lacks dedicated code-generation resources: datasets are scarce, tooling is limited, and benchmarks are largely absent (Bhattacharjee et al., 2022; Zehady et al., 2024). Consequently, general-purpose Bangla models are outperformed by their English counterparts on code-related tasks (Bhattacharyya et al., 2023; Uddin et al., 2023), underscoring the need for targeted data, evaluation suites, and modeling efforts.

While Bangla Natural Language Understanding (NLU) and Generation (NLG) see considerable growth with resources like BanglaRQA (Ekram et al., 2022) and BEnQA (Shafayat et al., 2024), the domain of code generation remains relatively under-explored. Prior work in this area is limited to two main benchmarks: `mHumanEval-Bangla` (Raihan et al., 2025a), a subset of a multilingual evaluation benchmark containing 164 prompts adapted from the HumanEval dataset, and `MBPP-Bangla` (Raihan et al., 2025b), which provides 974 coding prompts adapted from the MBPP dataset. For this shared task, we utilize a combined dataset composed of both `mHumanEval-Bangla` and `MBPP-Bangla`.

Our motivation for this shared task is to improve the performance of Bangla NLP models on code generation. The primary objective is to introduce a more advanced task that evaluates the emerging code generation capabilities of LLMs. As the first task of its kind for

[1]Task website: https://noshinulfat.github.io/blp25_code_generation_task/#/home

[2]https://www.ethnologue.com/

Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025), pages 373–387
December 23, 2025 ©2025 Association for Computational Linguistics

Bangla, we provide extensive support to participants, including a starter kit[3], tutorials, and seminars. Participants are also granted the flexibility to use any proprietary or open-source models, alongside any NLP methods. This open approach is intended not only to provide a strong starting point but also to uncover diverse strategies for solving a new and complex task for LLMs in a mid-resource language, yielding key insights for future research.

We elaborate on the task and present our findings, the remainder of this paper is organized as follows: Section 2 discusses the datasets used during the task, Section 3 describes the task and the it's two phases (dev & test), Section 4 includes the participants' results, Section 6 summarizes the approaches taken by the system description papers and Section 6 investigates the key insights.

2 Data

We utilize the only two available benchmarks for Bangla Code Generation: `mHumanEval-Bangla` (Raihan et al., 2025a) and `MBPP-Bangla` (Raihan et al., 2025b). These are selected for their distinct and complementary qualities.

Specs	HumanEval-Bangla	MBPP-Bangla
# of Tasks	164	974
Prompt	Bangla	Bangla
Solution	Python	Python
Problem source	Hand-written	Crowd-sourced
Task focus	Function completion	Basic–intermediate
Problem format	Docstring	Short prompt
Tests per task	7.7 (avg.)	3
Metric	pass@1	pass@1

Table 1: Dataset details for `HumanEval-Bangla` and `MBPP-Bangla`.

`MBPP-Bangla` offers scale and breadth: its 974 short, crowd-sourced Bangla prompts yield more coverage, which estimates and stress a model's ability to handle a wide variety of basic–intermediate tasks. `HumanEval-Bangla` complements this with depth: 164 hand-written, docstring-based function-completion problems paired with denser test suites (7.7 test cases on avg.) vs. 3 tests per task) probe precise adherence to specification. Evaluating on both benchmarks provides a more detailed picture of Bangla-to-Python code generation—breadth and robustness from `MBPP-Bangla`, and precision and rigor from `HumanEval-Bangla`. We have made the combined version publicly available. [4]

3 Task Description

In this task, we evaluate LLMs on one of their emerging capabilities, code generation. The task becomes more challenging as the prompts used in our task are in Bangla. As mentioned before, this is the first shared task of its kind in the Bangla NLP domain.

In formal definition, the task entails:

> Given a set of coding prompts (task descriptions and/or docstrings) in Bangla, the participants will have to use (prompt, finetune, etc.) LLMs to generate corresponding Python code snippets that pass all the test cases for that particular task. The evaluation metric is `Pass@1`, meaning that the models will have only one attempt to pass all the test cases for a particular prompt.

Task examples from both benchmarks are shown in Figure 1 and 2. We launch the dev phase of the task on the Codabench [5] platform on August 10th, 2025.

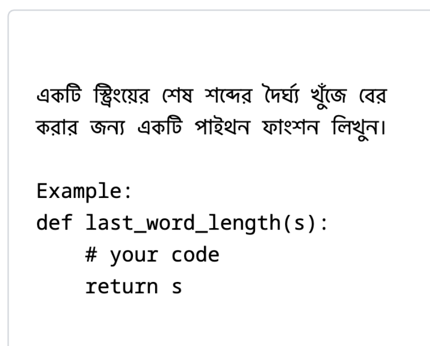

Figure 1: Sample prompt from MBPP-Bangla. Translation: *'Write a Python function to find the length of the last word in a given string'*.

[3] Starter Kit: https://noshinulfat.github.io/blp25_code_generation_task/#/starter-kit

[4] Datasets: https://noshinulfat.github.io/blp25_code_generation_task/#/starter-kit

[5] Competition website: https://www.codabench.org/competitions/10089/

Specification	DEV	TEST	Total
Start Date	August 10, 2025	September 7, 2025	—
End Date	September 8, 2025	September 14, 2025	—
Duration	28 Days	7 Days	35 Days
Participants	152	97	152
Teams	63	32	63
Submissions	301	187	488
Average (Submission)	4.78	5.84	7.75
Test Cases	Fully Released	One per Task	—
Highest Score (Pass@1)	0.99	0.95	—
Lowest Score	0.00	0.02	—
Average Score	0.52	0.59	—

Table 2: DEV/TEST timeline and participation summary. *Average (Submission)* denotes average submissions per team.

```
from typing import List

def
has_close_elements(numbers:
List[float], threshold: float)
-> bool:
    """ প্রদত্ত সংখ্যার তালিকায়, প্রদত্ত
প্রান্তিকের চেয়ে অন্য কোন দুটি সংখ্যা একে
অপরের কাছাকাছি আছে কিনা তা পরীক্ষা
করুন। উদাহরণঃ
    >>>
has_close_elements([1.0, 2.0,
3.0], 0.5)
    False
    >>>
has_close_elements([1.0, 2.8,
3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
```

Figure 2: Sample prompt from mHumanEval-Bangla. Translation: *'Write a Python function to find that if two numbers in a list are closer to each other than the given threshold'*.

3.1 DEV Phase

In the DEV phase, systems operate under high observability: fully released test cases and a longer horizon enable targeted debugging and steady pipeline stabilization. Teams submit at a disciplined rate (Avg. 4.78 submissions per team), and performance spans a wide range, from 0.00 to a near-ceiling 0.99 Pass@1 (Table 2). The broad spread, together with an average score of 0.52, indicates heterogeneous readiness—strong systems rapidly approach the ceiling, while weaker pipelines expose specification and edge-case errors that visible tests help uncover.

Methodologically, DEV functions as an internal-validity probe: with rich feedback, improvements reflect engineering rigor and prompt–test alignment rather than guesswork. The combination of high best score and moderate average suggests a bimodal landscape in which top teams consolidate gains early while others iterate to resolve stability issues. These dynamics make DEV well-suited for ablations and reproducibility checks, as changes map cleanly onto observable error reductions (Table 2).

3.2 TEST Phase

The TEST phase tightens observability—one visible test per task over a shorter window—shifting the emphasis from iterative debugging to generalization under uncertainty. Teams react by concentrating effort: average submissions per team increases to 5.84 despite a smaller field, and performance compresses upward, with the lowest score rising to 0.02 and the average improving to 0.59 (Table 2). This pattern is consistent with maturation effects (pipelines refined during DEV) and selection effects (fewer weak entries), producing stronger mid-pack outcomes.

At the top end, the best Pass@1 is slightly lower (0.95 vs. 0.99 in DEV), which is expected when feedback is constrained. The small top-line drop, paired with a higher mean, suggests that TEST emphasizes robustness over oppor-

Rank	Team Name	System Paper	Best Model	Pass@1
1	NALA_MAINZ	(Saadi et al., 2025)	GPT-5	0.95
2	Retriv	(Asib et al., 2025)	Qwen2.5-Coder-14B	0.93
3	Musafir	(Hasan et al., 2025)	Qwen2.5-14B-Instruct	0.92
4	AdversaryAI	(Riyad and Junaed, 2025)	Gemini 2.5 Pro	0.85
6	Code_Gen	(Agarwala et al., 2025)	GPT-5	0.84
7	TeamB2B	(Dihan et al., 2025)	Gemini-2.5-Pro	0.84
8	NSU_PiedPiper	(Fahmid et al., 2025)	Qwen2.5-Coder-14B	0.83
11	Barrier Breakers	(Jalil et al., 2025)	GPT OSS 120B	0.82
12	PyBhasha	(Dewan and Rifat, 2025)	Ensemble	0.80
13	JU_NLP	(Pal and Das, 2025)	GPT-4.1	0.77
16	AlphaBorno	(Rahman et al., 2025)	GPT-4o	0.72
17	PyBangla	(Islam et al., 2025)	Qwen3-8B	0.72
21	CUET_Expelliarmus	(Shahrier et al., 2025)	GPT-20B OSS	0.37
22	CodeAnubad	(Roy, 2025)	Gemma-2-9b-it	0.37
26	Troopers	(Farazi and Reza, 2025)	TigerLLM (RSFT)	0.32

Table 3: TEST phase results for the teams who submitted system description papers, ranked by Pass@1 scores (descending), scores rounded to two decimals. Complete results in Table 5 (Appendix B).

tunistic tuning: ceiling systems lose limited headroom, while the median gains from designs that encode safer defaults and broader guardrails. In effect, TEST acts as an external-validity probe, rewarding solutions that transfer beyond DEV's fully visible conditions and revealing residual brittleness in pipelines that depend on extensive test exposure (Table 2).

4 Results

DEV (63 teams). Table 4 (Appendix A) shows a clear separation between a small group of top systems and a wide middle. Under full test visibility, teams diagnose errors, adjust prompts and post-processing, and move up steadily. Top scores sit near perfect, but many teams still leave points on the table because of edge cases and inconsistent handling of problem specs. When scores are the same at the reported precision, we break ties by *shorter average solution length* (shorter wins). This favors solutions that meet the spec with minimal code rather than long, brittle fixes.

TEST (32 teams). Table 3 reflects how systems behave with less feedback. The middle of the leaderboard strengthens, while the very top tightens—high performers keep most of their lead, but not all of it. Designs that rely on stable defaults, careful I/O handling, and simple control flow hold up best; runs that depend on DEV-style trial-and-error drop back. We use the same tie-breaking rule here: *shorter average solution length* wins ties. In practice, this pushes teams toward concise, robust code that generalizes beyond the DEV environment.

5 Approaches

We briefly discuss the approaches of the 15 submitted *System* description papers in this section.

NALA_MAINZ (Saadi et al., 2025) (Rank 1)

The authors present the top-ranked system for the task. A lean multi-agent pipeline couples a code-generation agent with a selective debugger. The coder emits an initial solution and immediately runs unit tests; failures condense into error traces that guide the debugger to propose minimal, localized patches within a small step budget. The system augments supervision with matched external tests and lightweight auto-generated assertions, and it optionally translates Bangla prompts to English. Ablations indicate most gains come from error-trace–guided repair, with test augmentation adding complementary improvements.

Retriv (Asib et al., 2025) (Rank 2)

The authors propose a test-driven, feedback-guided framework. Their system uses a Qwen2.5-14B model (Hui et al., 2024b), fine-tuned with QLoRA (Dettmers et al., 2023), to generate an initial Python solution from translated English instructions. The code is immediately executed against unit tests. If a fail-

ure occurs, the error trace is fed back into the model prompt to guide a correction. This refinement loop is repeated up to three times with increasing temperature to encourage diverse solutions. The combination of parameter-efficient fine-tuning and iterative, execution-guided self-correction proved highly effective, securing the second-place rank.

Musafir (Hasan et al., 2025) (Rank 3)

This team employs a two-stage cascade pipeline. First, Bangla instructions are translated into English using a model optimized to preserve technical semantics. This step allows them to leverage powerful, English-centric code generation models. The translated prompt is then fed to a Qwen-based code generation model (Yang et al., 2024), which performs zero-shot code generation. The final output is validated using the provided unit tests. This direct translation-generation strategy effectively bridges the resource gap for Bangla, demonstrating a robust and high-performing approach that achieved third place in the competition.

AdversaryAI (Riyad and Junaed, 2025) (Rank 4)

The authors introduce TriGen (Think, Refine, and Generate), a system centered on a self-refinement loop. For open-source models, they use LoRA (Hu et al., 2022) to fine-tune on a dataset augmented with Chain-of-Thought (Wei et al., 2022) reasoning steps. The core of the system is an iterative process: an initial code solution is generated and tested. If it fails, the model receives the error feedback and is prompted to debug and correct its own output. This execution-guided refinement is applied to both their fine-tuned models and to a few-shot prompted Gemini 2.5 Pro, which yielded their top-performing submission.

Code_Gen (Agarwala et al., 2025) (Rank 6)

This work focuses on the impact of input quality, using a pipeline of preprocessing, translation, and assertion-based prompting with GPT-5. The authors first normalize the raw Bangla instructions to remove noise. Next, they translate the cleaned instructions to English to align with the model's strengths. Critically, they append the provided unit test assertions directly to the final prompt. This gives the model explicit examples of the required input-output behavior. Their experiments show that this assertion-augmented, translation-based approach significantly boosts performance, highlighting the importance of prompt clarity and context.

TeamB2B (Dihan et al., 2025) (Rank 7)

This team presents BanglaForge, a framework built on a retrieval-augmented (Lewis et al., 2020), dual-model collaborative pipeline. The system first uses TF-IDF to retrieve relevant solved examples, which are used for few-shot prompting. An initial "Coder" LLM generates a code solution. This solution is then passed to a "Reviewer" LLM, which validates the code, enhances its robustness, and refines it based on execution feedback from unit tests. This iterative cycle between the generator and reviewer agents, grounded by retrieved examples, effectively improves the final code's quality and correctness.

NSU_PiedPiper (Fahmid et al., 2025) (Rank 8)

The authors combine Chain-of-Thought (CoT) prompting (Zhou et al., 2024) with an iterative debugging loop (Liu et al., 2024). Using a Qwen-based model (Qwen Team et al., 2024), an initial solution is generated from a CoT prompt that encourages step-by-step reasoning. This code is then validated against unit tests. If any tests fail, the generated code and the resulting error messages are passed to a specialized debugger prompt. The model then attempts to fix the identified issues. This refinement process can be repeated up to three times, effectively using execution feedback to systematically correct errors from the initial reasoning phase.

Barrier Breakers (Jalil et al., 2025) (Rank 11)

This team introduces a novel approach that combines Test-Driven Development (TDD) and a Code Interpreter (CI) (Wang et al., 2024) without requiring model fine-tuning. First, in the TDD phase, the LLM generates additional test cases from the Bangla prompt. These new tests,

combined with the provided one, are injected into the final prompt for code generation. In the CI phase, the generated code is executed in a sandbox. If any compilation or assertion errors occur, the error message is fed back to the model for up to five retry attempts, enabling iterative self-correction.

PyBhasha (Dewan and Rifat, 2025) (Rank 12)

The authors investigate the impact of instruction quality and model ensembling. They compare three instruction variants: original Bangla, English translations via Facebook NLLB (Team et al., 2022), and semantic-aware English rewrites using GPT-4.1. Finding the GPT-4.1 rewrites most effective, they implement a two-stage ensemble for their final submission. The primary model (Qwen2.5-Coder-14B) generates the initial solution. If this solution fails unit tests, it is passed to a secondary model (Claude Sonnet 4) as a fallback, leveraging the complementary strengths of different architectures to improve the overall success rate.

JU_NLP (Pal and Das, 2025) (Rank 13)

This team employs a straightforward yet effective zero-shot prompting strategy (Brown et al., 2020). They construct a detailed prompt that instructs the model to act as a senior Python developer, providing it with the original Bangla problem statement, the required function signature, and the visible unit tests. The prompt explicitly tells the model it can translate the instruction internally before generating the code. They test this approach across several proprietary models, with their best result coming from GPT-4.1, demonstrating the strong out-of-the-box, cross-lingual reasoning capabilities of modern frontier models.

AlphaBorno (Rahman et al., 2025) (Rank 16)

This work systematically evaluates several prompting strategies. After translating Bangla instructions to English with GPT-4o, they compare zero-shot, few-shot, and Chain-of-Thought baselines. Their key finding is that providing explicit behavioral constraints is more effective than abstract reasoning. Their best-performing method augments a zero-shot prompt with synthetic unit tests to cover edge cases. This is combined with a self-repair loop where failed execution feedback is used to prompt the model for a correction, with GPT-4o achieving the highest score under this configuration.

PyBangla (Islam et al., 2025) (Rank 17)

The authors introduce BanglaCodeAct, an agent-based framework inspired by the ReAct paradigm. Their system uses a general-purpose multilingual LLM (Qwen3-8B) in an iterative Thought-Code-Observation loop without any task-specific fine-tuning. For each problem, the agent first generates a 'Thought' in Bangla outlining its plan. It then produces Python 'Code' to implement the plan. This code is executed, and the 'Observation' (output or error) is fed back to the agent. This cycle of self-correction continues until the code passes all unit tests, proving effective for low-resource code generation.

CUET_Expelliarmus (Shahrier et al., 2025) (Rank 21)

This team proposes a two-stage pipeline using the open-source GPT-20B OSS model. In the first stage, the Bangla instruction is translated to English and then refined using a one-shot prompt to create a well-structured specification. This refined English instruction is then passed to the second stage for code generation using a zero-shot prompt. The generated code is validated against unit tests. If a test fails, the traceback error is used as feedback to re-prompt the model, with this iterative correction loop running for up to five attempts.

CodeAnubad (Roy, 2025) (Rank 22)

This work tackles the extreme data scarcity of the task with an iterative self-improvement strategy. The authors first fine-tune a Gemma-2-9b model on the initial 74 training samples using QLoRA (Dettmers et al., 2023). This model is then used to generate solutions for the development set. All solutions that pass the unit tests are harvested and added to the training set. The model is then re-trained on this augmented dataset. This process creates a positive feedback loop, progressively improving performance by

curating a high-quality, in-domain dataset from the model's own verified outputs.

Troopers (Farazi and Reza, 2025) (Rank 26)

The authors implement a reward-selective fine-tuning (RSFT) pipeline (Dong and others, 2023). The process begins by sampling multiple candidate programs from a base model for each Bangla prompt. Each candidate is executed in a sandbox, and only those that pass all unit tests (the "winners") are retained. This curated set of high-quality, execution-verified instruction-code pairs forms the dataset for supervised fine-tuning (SFT). The base model is then efficiently updated on this dataset using LoRA adapters, selectively reinforcing correct program synthesis without complex reinforcement learning.

6 Analysis

Since the task focuses on generation, all the systems are built around one or more LLMs. Table 6 (Appendix C) lists all the models used by each system.

6.1 Preference on LLMs

Participants have used a total of 20 different LLMs, including 6 proprietary and 14 open-source models. As Figure 3 illustrates that TigerLLM is the most used model along with two other open-source ones (LLaMA 3 and Qwen2.5).

6.2 Best Performing LLMs

As Figure 4 shows, Qwen2.5 was the best-performing model by most systems, followed by the proprietary models and some other open-source models.

6.3 Methodologies

As shown in Figure 5, teams build upon a common foundation. Prompting is nearly universal, while a majority (8 of 15 teams) use Machine Translation to leverage powerful English-centric models, a key strategy for systems like Musafir (R3). Five teams employ Finetuning to specialize models; Retriv (R2) uses QLORA for efficiency, while Troopers (R26) implements a reward-selective pipeline (RSFT) to train on verified-correct code.

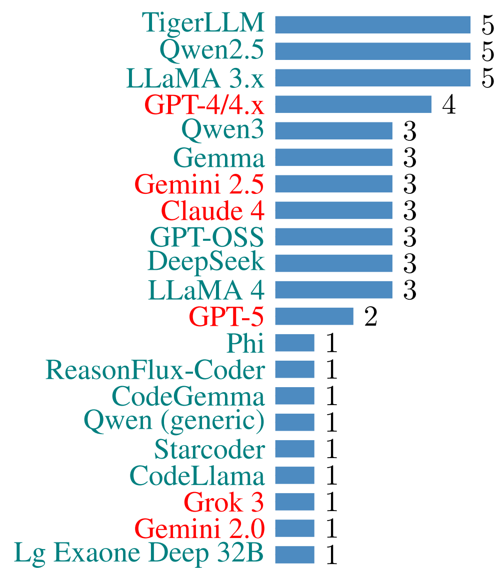

Figure 3: Most used LLMs by the submitted systems. Proprietary models are in red.

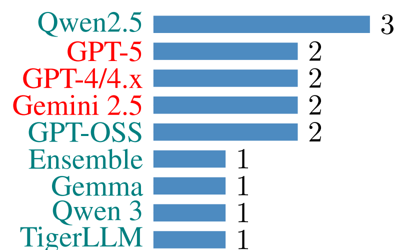

Figure 4: Best performing LLMs by the submitted systems. Proprietary models are in red.

The most critical differentiator for top-tier systems is the implementation of a self-correcting feedback loop. This is often initiated with Chain-of-Thought (CoT) prompting to improve the model's initial reasoning, as seen with NSU_PiedPiper (R8). The core of this approach is Iterative Self-Correction, where generated code is executed and any resulting errors are fed back to the model for debugging. This refinement process proves central to the success of the highest-performing teams, including NALA_MAINZ (R1), Retriv (R2), and AdversaryAI (R4).

A few teams explore more specialized strategies. TeamB2B (R7) utilizes Retrieval-Augmented Generation (RAG) to provide models with relevant examples, while Barrier Break-

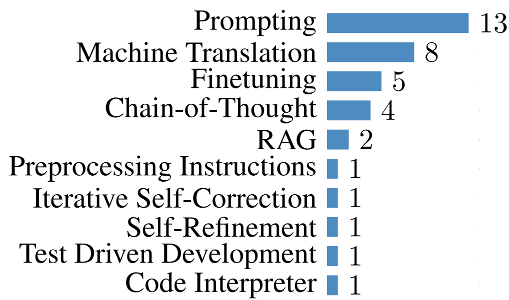

Figure 5: Most used methodologies by the submitted systems.

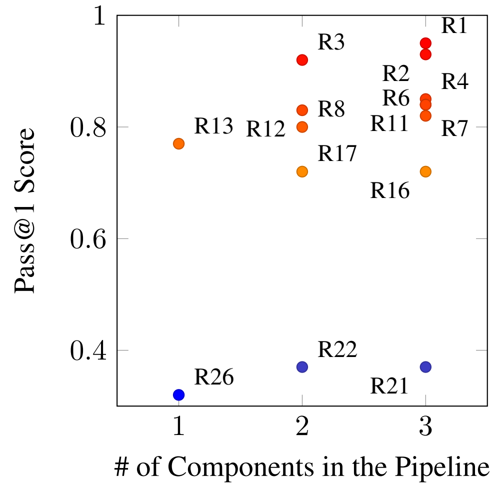

Figure 6: Correlation between pipeline complexity (number of distinct high-level components/methodologies employed) and the final Pass@1 score for the 15 teams with system description papers. Each point is labeled with the team's final rank.

ers (R11) uniquely combines Test-Driven Development (TDD) with a Code Interpreter (CI) for safe, iterative refinement. These advanced methods underscore a clear trend: top performance requires moving beyond foundational techniques to build robust, multi-step systems that emulate real-world development workflows.

6.4 Pipeline Components

Figure 6 visualizes the relationship between the architectural complexity of a system and its final score. A strong positive correlation is evident: systems employing a greater number of integrated methodologies consistently achieved higher performance. Notably, five of the top eight teams utilized complex pipelines integrating at least three distinct techniques, such as translation, Chain-of-Thought, and iterative self-correction. This trend highlights that success in this task was not merely dependent on model choice, but was significantly driven by the sophistication of the overall pipeline. Simpler approaches, while effective to a degree, generally did not reach the top performance tiers.

7 Conclusion

In this shared task, the first of its kind for Bangla code generation, we successfully benchmarked the capabilities of modern LLMs on a low-resource language. We observed a clear methodological trend from the diverse systems submitted: top performance was not driven by model choice alone, but by pipeline complexity. We found that the most effective systems implemented robust, multi-step workflows with self-correction loops that emulate a developer's iterative debugging process. We observe that, finetuning and machine translation were the most effective methods during the test phase. While open-source models only performed better after pairing them up with some test-driven coding tools.

Our results establish a strong baseline and highlight the effectiveness of agentic, self-refining architectures. For future work, we recommend focusing on developing capable Bangla-native code models to reduce the dependency on translation, expanding benchmark complexity, and exploring how these successful pipeline strategies can be transferred to other languages.

We plan to build on our findings, and our priorities include refining these agentic workflows, developing native Bangla code models to reduce the current dependency on translation, and increasing benchmark complexity to repository-level tasks like bug fixing. Advancing these areas will not only improve Bangla code generation but also provide a transferable blueprint for other under-resourced languages, making AI-driven software development more globally accessible.

Limitations

While our task is intentionally focused on generating self-contained, function-level Python code, we acknowledge this does not encompass the full complexity of real-world software engineering. This focused scope, however, was a deliberate design choice to establish a clear, controlled, and reproducible benchmark—a critical first step for a new task in a low-resource language. Similarly, our use of the stringent Pass@1 metric, which is standard in code generation benchmarks, provides an unambiguous signal of functional correctness. While many top systems relied on translating prompts to English, we view this not as a limitation of the task, but as a key finding that accurately reflects the current state-of-the-art strategies for bridging the resource gap, providing a realistic baseline for future work to improve upon.

Ethical Considerations

The datasets used in this task are derived from publicly available, open-source benchmarks, mitigating data privacy concerns. A primary goal of our work is to enhance the accessibility of programming tools for Bangla speakers, promoting linguistic inclusivity in technology. However, we acknowledge that any code generation system carries a potential risk of misuse for generating malicious code, although the function-level scope of our task makes this risk indirect. The prevalent use of proprietary models also means we rely on the safety and bias mitigations implemented by model providers. While the technical nature of the prompts limits the potential for social bias, the common strategy of translating prompts to English could amplify biases present in the target English-centric LLMs.

References

Abhishek Agarwala, Shifat Islam, and Emon Ghosh. 2025. Code_gen at blp-2025 task 2: Banglacode: A crosslingual benchmark for code generation with translation and assertion strategies. In *Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025)*. Asian Chapter of Association for Computational Linguistics (AACL).

Kabir Ahuja, Anirudh Das, Sandipan Das, Ashwini Deshpande, Sebastian Gehrmann, Anup Gopinath, Arya Guha, Pooja Kumar-Jois, Prem Mani, Ashwin Paranjape, et al. 2023. Mega: Multilingual evaluation of generative ai. *arXiv preprint arXiv:2310.10567*.

K M Nafi Asib, Sourav Saha, and Mohammed Moshiul Hoque. 2025. Retriv at blp-2025 task 2: Test-driven feedback-guided framework for bangla-to-python code generation. In *Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025)*. Asian Chapter of Association for Computational Linguistics (AACL).

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

Abhik Bhattacharjee, Tahmid Hasan, Wasi 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*.

Pramit Bhattacharyya, Joydeep Mondal, Subhadip Maji, and Arnab Bhattacharya. 2023. Vacaspati: A diverse corpus of bangla literature. In *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*.

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

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in Neural Information Processing Systems*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. In *arXiv preprint arXiv:2107.03374*.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems*, 36:10088–10115.

Foyez Ahmed Dewan and Nahid Montasir Rifat. 2025. Pybhasha at blp-2025 task 2: Effectiveness of semantic-aware translation and ensembling in bangla code generation. In *Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025)*. Asian Chapter of Association for Computational Linguistics (AACL).

Mahir Labib Dihan, Sadif Ahmed, and Md Nafiu Rahman. 2025. Teamb2b at blp-2025 task 2: Banglaforge: Llm collaboration with self-refinement for bangla code generation. In *Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025)*. Asian Chapter of Association for Computational Linguistics (AACL).

Yao Dong et al. 2023. Raft: Reward ranked finetuning for generative foundation models. In *NeurIPS 2023*.

Syed Mohammed Sartaj Ekram, Adham Arik Rahman, Md Sajid Altaf, Mohammed Saidul Islam, Tareq Mahmood Jamil, Shadman Sakib Alam, Irfan Kabir, Mohammad Nasim, Enamul Hossain, and Nawshad Akhter. 2022. Banglarqa: A benchmark dataset for under-resourced bangla language reading comprehension-based question answering with diverse question-answer types. In *Findings of the Association for Computational Linguistics: EMNLP 2022*.

Ahmad Fahmid, Fahim Foysal, Wasif Haider, Shafin Rahman, and Md Adnan Arefeen. 2025. Nsu_piedpiper at blp-2025 task 2: A chain-of-thought with iterative debugging approach for code generation with bangla instruction. In *Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025)*. Asian Chapter of Association for Computational Linguistics (AACL).

Musa Tur Farazi and Nufayer Jahan Reza. 2025. Troopers at blp-2025 task 2: Reward-selective fine-tuning based code generation approach for bangla prompts. In *Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025)*. Asian Chapter of Association for Computational Linguistics (AACL).

Sakibul Hasan, Md Tasin Abdullah, Abdullah Al Mahmud, and Ayesha Banu. 2025. Musafir at blp-2025 task 2: Generating python code from bangla prompts using a multi-model cascade and unit test validation. In *Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025)*. Asian Chapter of Association for Computational Linguistics (AACL).

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024a. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024b. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Jahidul Islam, Md Ataullha, and Saiful Azad. 2025. Pybangla at blp-2025 task 2: Enhancing bangla-to-python code generation with iterative self-correction and multilingual agents. In *Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025)*. Asian Chapter of Association for Computational Linguistics (AACL).

Sajed Jalil, Shuvo Saha, and Hossain Mohammad Seym. 2025. Barrier breakers at blp-2025 task 2: Enhancing bengali llm code generation capabilities through test driven development and code interpreter. In *Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025)*. Asian Chapter of Association for Computational Linguistics (AACL).

Pratik Joshi, Sebastin Santy, Amar Budhiraja, Kalika Bali, and Monojit Choudhury. 2020. The state and fate of linguistic diversity and inclusion in the nlp world. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Myle Ott, Wen-tau Chen, Alexis Conneau, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474.

M Liu et al. 2024. Iterative debugging for neural code generation. *ACM Transactions on Programming Languages and Systems*, 46(3):1–33.

Pritam Pal and Dipankar Das. 2025. Ju_nlp at blp-2025 task 2: Leveraging zero-shot prompting for bangla natural language to python code generation. In *Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025)*. Asian Chapter of Association for Computational Linguistics (AACL).

Liliana Pasquale, Antonino Sabetta, Marcelo d'Amorim, Péter Hegedűs, Mehdi Tarrit Mirakhorli, Hamed Okhravi, Mathias Payer, Awais Rashid, Joanna CS Santos, Jonathan M Spring, et al. 2025. Challenges to using large language models in code generation and repair. *IEEE Security & Privacy*, 23(2):81–88.

Qwen Team, Jinze Bai, Shuai Xu, Yankai Zhang, Zhenru Wang, Songyang Wang, Ziyan Li, Wang Wang, Li Wang, Siyuan Liu, et al. 2024. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*.

Mohammad Ashfaq Ur Rahman, Muhtasim Ibteda Shochcho, and Md Fahim. 2025. Alphaborno at blp-2025 task 2: Code generation with structured prompts and execution feedback. In *Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025)*. Asian Chapter of Association for Computational Linguistics (AACL).

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)*.

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, Christian Newman, and Marcos Zampieri. 2024. Code llms: A taxonomy-based survey. In *Proceedings of IEEE BigData*.

Nishat Raihan, Mohammed Latif Siddiq, Joanna CS Santos, and Marcos Zampieri. 2025c. Large language models in computer science education: A systematic literature review. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1*, pages 938–944.

Omar Faruqe Riyad and Jahedul Alam Junaed. 2025. Adversaryai at blp-2025 task 2: A think, refine, and generate (trigen) system with lora and self-refinement for code generation. In *Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025)*. Asian Chapter of Association for Computational Linguistics (AACL).

Soumyajit Roy. 2025. Codeanubad at blp-2025 task 2: Efficient bangla-to-python code generation via iterative lora fine-tuning of gemma-2. In *Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025)*. Asian Chapter of Association for Computational Linguistics (AACL).

Hossain Shaikh Saadi, Faria Alam, Mario Sanz-Guerrero, Minh Duc Bui, Manuel Mager, and Katharina von der Wense. 2025. Nala_mainz at blp-2025 task 2: A multi-agent approach for bengali instruction to python code generation. In *Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025)*. Asian Chapter of Association for Computational Linguistics (AACL).

Sheikh Shafayat, H Hasan, Minhajur Mahim, Rifki Putri, James Thorne, and Alice Oh. 2024. BEnQA: A question answering benchmark for Bengali and English. In *Findings of the Association for Computational Linguistics: ACL 2024*.

Md Kaf Shahrier, Suhana Binta Rashid, Hasan Mesbaul Ali Taher, and Mohammed Moshiul Hoque. 2025. Cuet_expelliarmus at blp2025 task 2: Leveraging instruction translation and refinement for bangla-to-python code generation with open-source llms. In *Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025)*. Asian Chapter of Association for Computational Linguistics (AACL).

Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, et al. 2025. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*.

NLLB Team, Marta R Costa-jussà, James Cross, Onur Çelebi, Maha Elbayad, Kenneth Heafield, Guillaume Wenzek, Kevin Lin, Tatiana Tran, Shruti Le, et al. 2022. No Language Left Behind: scaling human-centered machine translation. *arXiv preprint arXiv:2207.04672*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Md Nafis Uddin, Masum Khan, Nabila Hasan, and Mahmudul Hossain. 2023. Exploring code-mixed bangla text in large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*.

Tianyi Wang, Yang Ye, Panupong Pasupat, Aohan Wan, Grant Friedman, Jiacheng Tu, Maya Schaar, Jason Wei, Suriya Gunasekar, Matthew Richardson, et al. 2023. Babelcode: Llm as a polyglot programmer. *arXiv preprint arXiv:2303.03845*.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better llm agents. *arXiv preprint arXiv:2402.04391*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. 2024. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*.

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

Denny Zhou et al. 2024. Advancing chain-of-thought reasoning in large language models. *Nature Machine Intelligence*, 6(1):45–58.

A DEV Phase Results

Rank	Team_Name	Score
1	BRACU_CL	1.00
2	NALA_MANIZ_2	1.00
3	Team_Trinity	0.99
4	not_Decided	0.99
5	Code_Gen	0.97
6	Team B2B	0.97
7	Musafir	0.94
8	Oleksandr Usyk	0.94
9	PyBangla	0.94
10	alubhorta	0.94
11	Metaphor	0.93
12	NO Name	0.91
13	Alpha Borno	0.90
14	Gradient Masters	0.88
15	NLPirates	0.87
16	PyBhasha	0.86
17	Nsu_PiedPiper	0.85
18	CUET_SIURS	0.84
19	JU_NLP	0.84
20	Retriv	0.84
21	SamNLP	0.84
22	fallen_dark-358115	0.84
23	Ecstasy	0.83
24	NeuralCoders	0.83
25	CUET_DuoBingo	0.82
26	3_idiots	0.78
27	BanglaBytes	0.73
28	Py_Chunker	0.68
29	delayed	0.64
30	BarrierBreakers	0.62
31	AdversaryAI	0.60
32	Team_AA	0.56
33	BLPCG	0.51
34	theDarkKnights	0.48
35	soumyajit	0.47
36	wspr	0.46
37	rms92	0.44
38	NeuralCoders	0.41
39	KodomAli Coders	0.38
40	PrompterXPrompter	0.38
41	UIU_NLP	0.32
42	unknown	0.31
43	Md_Abdur_Rahman	0.29
44	CUET_Zahra_Duo	0.17
45	Wahid	0.11
46	Organizers	0.10
47	Quasar	0.10
48	Team_Ban	0.10
49	turtur	0.10
50	SoloGuy	0.09
51	huday	0.09
52	troublemaker	0.09
53	Arekta Team	0.08
54	Kaf	0.08
55	None	0.08
56	Sweet Dreams	0.08
57	disco	0.08
58	cuet_1376	0.03
59	tryNLP	0.03
60	nafiurahman-353732	0.01
61	CUET_NLP_Zahra_Duo	0.00
62	Troopers	0.00
63	programophile	0.00

Table 4: DEV phase results. 63 Teams - ranked by Pass@1 scores (descending) — scores rounded to two decimals.

B TEST Phase Results

Rank	Team Name	Pass@1
1	NALA_MAINZ	0.95
2	Retriv	0.93
3	Musafir	0.92
4	AdversaryAI	0.85
5	BRACU_CL	0.84
6	Code_Gen	0.84
7	TeamB2B	0.84
8	NSU_PiedPiper	0.83
9	One Braincell	0.83
10	fallen_dark-370156	0.83
11	Barrier Breakers	0.82
12	PyBhasha	0.80
13	JU_NLP	0.77
14	This Team has no name	0.77
15	NLPirates	0.74
16	AlphaBorno	0.72
17	PyBangla	0.72
18	CUET_DuoBingo	0.70
19	CUET_SIURS	0.67
20	Ecstasy	0.66
21	CUET_Expelliarmus	0.37
22	CodeAnubad	0.37
23	Gradient Masters	0.36
24	team_trinity	0.36
25	nidala	0.33
26	Troopers	0.32
27	Team Random	0.28
28	delayed	0.18
29	Organizers	0.17
30	huday	0.09
31	SyntaxMind	0.08
32	Team Random	0.02

Table 5: TEST phase results for the teams who submitted system description papers. 32 Teams — ranked by Pass@1 scores (descending), scores rounded to two decimals.

C LLMs Used by Teams

The following table details the various Large Language Models (LLMs) employed by the teams who submitted system papers, sorted by their final rank in the competition.

Rank	Team Name	Models Used
1	NALA_MAINZ	GPT 5, Claude 4, Gemini 2.5
2	Retriv	Phi, Qwen3, Qwen2.5-Coder, Llama-3.1, ReasonFlux-Coder, codegemma
3	Musafir	Qwen
4	AdversaryAI	TigerLLM, Gemma 3, Gemini 2.5, Llama3, Qwen3, Qwen2.5
6	Code_Gen	GPT 4, GPT 5, LLaMA 4, TigerLLM, Deepseek
7	TeamB2B	Gemini 2.5, Gemma-1B, GPT-OSS, DeepSeek-R1, Gemini2.0, Lg Exaone Deep
8	NSU_PiedPiper	Qwen2.5-Coder-14B
11	Barrier Breakers	LLaMA 4, Llama 3.2, GPT-OSS
12	PyBhasha	GPT 4, Claude 4, TigerLLM-9B, Qwen2.5-Coder, LLaMA-3.1
13	JU_NLP	GPT 4, LLaMA 4
16	AlphaBorno	GPT 4, Claude 3.7 Sonnet, Qwen Coder 2.5, Grok 3
17	PyBangla	TigerLLM, Qwen3, Llama-3.1, DeepSeek-Coder-V2
21	CUET_Expelliarmus	GPT-20B-OSS
22	CodeAnubad	Gemma, Starcoder, CodeLlama
26	Troopers	TigerLLM

Table 6: All the LLMs used by the teams (only includes the submitted system description papers).