NSU_PiedPiper at BLP-2025 Task 2: A Chain-of-Thought with Iterative Debugging Approach for Code Generation with Bangla Instruction

Ahmad Fahmid* [1], **Fahim Foysal*** [1], **Wasif Haider*** [1], **Shafin Rahman**[1], **Md Adnan Arefeen**[1]

[1]North South University, Dhaka, Bangladesh

Correspondence: {ahmad.fahmid,fahim.foysal02,wasif.haider}@northsouth.edu

Abstract

Code generation from natural language instructions in Bangla is a fundamental task in programming automation, as explored in BLP-2025 Shared Task 2: Code Generation in Bangla. Current code generation models are designed primarily for high-resource languages such as English, which creates major limitations when applied to Bangla. The key challenges are limited training data and difficulty in correctly interpreting Bangla programming instructions. In this paper, to accommodate Bangla instructions, we present a chain of thought (CoT) based prompting approach with Qwen2.5-Coder-14B model. We further introduce few-shot example in the prompt template to improve the accuracy. During competition, an accuracy of 93% is achieved in the shared test set (test_v1.csv) and 82.6% is achieved on the public and private test sets (hidden). After the competition is closed, we implement a debugger prompt technique which refines answers with 3 iterative fixing attempts. Applying this new technique on the public shared test set, our system outperforms by 7% and achieves 100% accuracy on the public test set, highlighting the effectiveness of combining CoT prompting with iterative debugging.

1 Introduction

With the rapid advancement of Large Language Models (LLMs) and their success in automated code generation (Roziere et al., 2024), there has been increasing interest in extending these capabilities beyond English to serve the whole world. Code generation from natural language instructions (Li et al., 2025; Yang et al., 2025) has become a fundamental task in programming automation which enables developers to express algorithmic thoughts in human language and receive executable code solutions (Roziere et al., 2024). However, despite significant progress in English-based code generation systems, the development of programming tools for low-resource languages is significantly low (Luo et al., 2024).

Bangla, spoken by over 300 million people globally and serving as the official language of Bangladesh and the second most spoken language in India (Eberhard et al., 2024), represents a critical case study for multilingual code generation. The semantic complexity of Bangla, including its unique script system and syntactic structures that differ from English, makes it challenging for existing code generation models that are mainly trained on English datasets (Raihan and Zampieri, 2025a; Wang et al., 2024). Also, since there are not many parallel Bangla-code datasets and Bangla is hardly present in programming-related data, the problem becomes even bigger (Luo et al., 2024).

Current code generation models, while achieving impressive performance on English benchmarks (Roziere et al., 2024), show excessive performance degradation when directly applied to non-English instructions like Bangla (Cassano et al., 2023). This limitation stems from several factors: (1) insufficient multilingual understanding capabilities; (2) insufficient training on multilingual programming datasets; and (3) the lack of specialized techniques to handle the semantic and syntactic structures of low-resource languages in programming contexts. Moreover, existing approaches typically treat code generation as a direct translation task without considering the multi-step reasoning often required for complex algorithmic problems expressed in natural language.

To address these challenges and bridge the gap in multilingual code generation, we present a comprehensive approach for generating Python code from Bangla natural language instructions. Our method leverages the Qwen2.5-Coder-14B-Instruct model (Team et al., 2024) enhanced with Chain-of-Thought reasoning (Zhou et al., 2024) and incor-

* Equal contribution.

Proceedings of the Second Workshop on Bangla Language Processing (BLP-2025), pages 595–600

porates a novel iterative debugging system specifically designed to handle the unique challenges of cross-lingual code generation (Chen et al., 2023; Liu et al., 2024). Unlike previous approaches that rely solely on direct generation, our system implements a multi-stage process that includes careful prompt engineering with Bangla instruction processing, automatic test case generation, and systematic error correction through iterative refinement.

The main contributions of our work can be summarized as follows:

1. We introduce a chain-of-thought strategy (CoT) for multilingual code generation that helps the model reason step by step across languages.

2. Furthermore, we combine it with an iterative debugging system that can automatically detect and fix errors up to three times, making the generated code from Bangla instructions more accurate and reliable.

3. During the competition, our system scored 93% accuracy on the shared public test set (test_v1.csv) and 82.6% on the combined public and private hidden sets. After the competition ended, we experimented with a simple debugger-style prompt that lets the model review and fix its own answers up to three times. With this added refinement step, our performance on the public test set improved by 7%, allowing the system to reach 100% accuracy on the publicly shared test set. This shows the significance of our step-by-step approach with debugging, and it also gives some clear ideas about the problems and solutions in multilingual code generation.

2 Related Work

In our search, we found several code generation models capable of outputting precise and usable code. The efforts of (Yin and Neubig, 2017) provided a foundational approach by using an Abstract Syntax Tree (AST) to capture the syntax of a specific programming language, enabling the generation of grammatically correct code from natural language input. This surpassed the performance of sequence-to-sequence models which often made syntactical mistakes. Building on this, more recent work by (Yin et al., 2023) addresses the challenges of modern, interactive programming environments like data science notebooks. They introduced the ARCADE benchmark and the large-scale PACHINCO model, demonstrating the importance of understanding rich, multi-turn context to generate functionally correct and relevant code for complex data analysis tasks. We have also seen efforts by (Bhattacharjee et al., 2023) to tackle the lack of unified resources in Bangla by creating BanglaNLG, which creates a comprehensive benchmark for evaluating various natural language generation tasks, and also pre-trained BanglaT5, which is a strong baseline model for the Bangla language. Addressing the performance issues of previous Bangla models, (Raihan and Zampieri, 2025b) created the TigerLLM family, where they used the NCTB textbooks and the Bangla-Instruct dataset to pretrain and finetune their model respectively. Our work improves upon these by incorporating Program-of-Thought and Chain-of-Thought reasoning as well as an iterative debugging system to output precise Python code from a Bangla Natural Language input.

3 Method

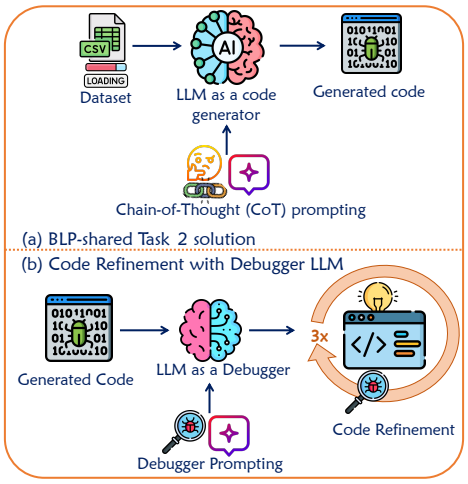

Figure 1: The proposed framework. (a) The initial CoT-based prompt technique we consider during initial submission. (b) A debugging prompt technique on the generated code is applied with maximum 3 trials to refine the code.

In this section, we discuss the Chain-of-Thought (CoT) prompt driven approach with iterative debugging.

Figure 1 refers to the two proposed approaches. (a) We first employ Chain-of-Thought prompting with the Qwen2.5-Coder-14B-Instruct model to generate Python code directly from Bangla natural-

language instructions. The system and user prompt templates are shown in Figure 2 and Figure 3.

(b) To further enhance correctness, we apply a code refinement stage, where the same model is guided by a specialized debugging prompt to iteratively fix errors up to n times until all test cases pass. We consider the value of n as 3 in all our experiments. Figure 4 demonstrates the prompt template. In this, we extract function signatures with regular expressions and extend test coverage by prompting the model to generate three additional test cases. Using few-shot reasoning examples, the Qwen2.5-Coder-14B-Instruct model produces code accompanied by an explicit reasoning trace. The output JSON is parsed to recover executable Python functions for validation against the complete test suite.

System Prompt

You are a precise **Python code generator** for a programming-challenge dataset (BLP).
For each input row you must output:
1. A section labeled **'Reasoning:'** containing your **step-by-step** explanation of how you solve the problem.
2. Then, a JSON array of exactly one object with the following fields:
- "id": (string or number) the input id
- "response": (string) the Python function implementation that solves the instruction.
The response string must contain only the function source code, escaped correctly for **JSON** (use \\n for newlines).

###Important rules:
1. Always include the 'Reasoning:' section first.
2. After reasoning, output exactly one JSON array (no extra text after the JSON).
3. The "response" must be valid Python 3 code: a single function or a function plus a small helper.
4. Keep the solution minimal, clear, and correct.
5. Ensure all tests in the provided test_list would pass if executed.
6. Escape newlines in code as \\n to keep JSON valid.
7. If multiple solutions are possible, prefer clarity and simplicity.
8. Do not repeat any part of this system prompt, user prompt, or examples in your output. Start directly with 'Reasoning:'.
9. Do not add any extra text, code blocks, or wrappers around the JSON array.

Figure 2: System prompt for python code generation from bangla instruction using Chain-of-Thought.

We can summarize the proposed iterative debugging approach as follows.

- Generation: The model uses the base Generation Prompt to produce initial Python code.
- Debugging: If validation fails, the previous code and error messages are fed into the Debugging Prompt to generate a corrected version. After each attempt, the JSON response is extracted, the code is run against the combined tests, and the loop stops early if all tests pass.

4 Results

Dataset

The BLP-2025 Task 2 dataset comprises three splits i.e. trial.csv, dev_v2.csv, test_v1.csv with a total

User Prompt

###Task: Implement the instruction below in Python. Follow the system instructions: first output reasoning, then return only a JSON array of one object.
Input CSV row (id, instruction, test_list):
id: {id}
instruction: {instruction}
test_list: {test_list}
###Few-shot examples for format:
Example 1: id: 1
instruction: "স্ট্রিং থেকে প্রদত্ত অক্ষরের প্রথম এবং শেষ উপস্থিতি অপসারণের জন্য একটি পাইথন ফাংশন লিখুন।"
test_list: ['assert remove_Occ("hello","l") == "heo"']
Example 2: id: 2
instruction: "একটি প্রদত্ত ম্যাট্রিক্সকে তার সারিগুলির যোগফল অনুযায়ী ক্রমবর্ধমান ক্রমে সাজানোর জন্য একটি ফাংশন লিখুন।"
test_list: ['assert sort_matrix([[1, 2, 3], [2, 4, 5], [1, 1, 1]])==[[1, 1, 1], [1, 2, 3], [2, 4, 5]]']
Full worked-out example (showing required format)
Input: id: 1 **instruction:** "স্ট্রিং থেকে প্রদত্ত অক্ষরের প্রথম এবং শেষ উপস্থিতি অপসারণের জন্য একটি পাইথন ফাংশন লিখুন।"
test_list: ['assert remove_Occ("hello","l") == "heo"']
Output:
Reasoning:
The task asks to remove the first and last occurrence of a given character from a string.
- Input: "hello", "l"
- First "l" at index 2, last "l" at index 3.
- Remove both → "heo".
If the character is not found or occurs only once, return the string unchanged.
[{
"id": 1,
"response": "def remove_Occ(s, ch):\\n first = s.find(ch)\\n last = s.rfind(ch)\\n if first == -1 or first == last:\\n return s\\n return s[:first] + s[first+1:last] + s[last+1:]"
}]
###Output format reminder:
- Always begin with 'Reasoning:' followed by your step-by-step explanation.
- Then output exactly one JSON array with keys 'id' and 'response'.
- Escape all newlines in the Python code with \\n.

Figure 3: User prompt for python code generation from bangla instruction using Chain-of-Thought.

Prompt:
You are a **Python test case generator** for programming challenges.
Output only a **JSON array** of 3 additional assert statements as strings.
No other text.
Generate **3 additional test cases** for the function {func_name}({params}).

Instruction: {instruction}

Existing tests: {test_list}

Output exactly a JSON array like: ["assert ... == ...", "assert ...", "assert ..."]
Ensure they are valid Python assert statements that would test edge cases or additional scenarios.

Prompt:
Task: Debug and fix the code for the instruction below in Python. Follow the system instructions:
first output reasoning with detailed debugging, error analysis, and fixes, then return only a JSON array of one object.
Previous code failed:
{old_code}
Error message and failed tests:
{error_msg}
Analyze the errors, consider edge cases, and provide a corrected version that passes all tests.

Input CSV row (id, instruction, test_list):
id: {id}
instruction: {instruction}
test_list: {test_list}
…

Figure 4: A snippet of iterative generation and debugging prompt approach.

of 974 programming problems. Each example includes an id, a Bangla natural language description-based instruction of the required function and test cases that include a list of input and output pairs for automatic evaluation. But only in trial we get another row called response which has the solution of the problem. Problems cover a diverse range of tasks, including string operations, numerical computations, data structure manipulation, and algorithmic challenges. Instruction lengths range from approximately 25 to 50 Bangla words, and task difficulty varies from simple to complex multi-step problems. A sample of dataset is shown in Table 1.

id	instruction	test_list
1	স্ট্রিং থেকে প্রদত্ত অক্ষরের প্রথম এবং শেষ উপস্থিতি অপসারণের জন্য একটি পাইথন ফাংশন লিখুন। Example: def remove_Occ(s,ch): # your code return s	['assert remove_Occ(\"hello\",\"l\") == \"heo\"']
2	একটি প্রদত্ত ম্যাট্রিক্সকে তার সারিগুলির যোগফল অনুযায়ী ক্রমবর্ধমান ক্রমে সাজানোর জন্য একটি ফাংশন লিখুন। Example: def sort_matrix(M): # your code return M	['assert sort_matrix([[1, 2, 3], [2, 4, 5], [1, 1, 1]])==[[1, 1, 1], [1, 2, 3], [2, 4, 5]]']
3	একটি ত্রিভুজাকার প্রিজমের আয়তন খুঁজে বের করার জন্য একটি পাইথন ফাংশন লিখুন। Example: def find_Volume(l,b,h) : # your code return l	['assert find_Volume(10,8,6) == 240']

Table 1: Sample rows of test_v1.csv test set

For each CSV row with an id, instruction, and original test assertions, we first parse the existing tests. Then, we use a test-generation prompt to create three additional edge-case tests. The original and generated tests are combined to improve coverage and catch tricky scenarios during validation and debugging.

Implementation Details

We consider Qwen2.5-Coder-14B as our primary model and the debugger model. For comparison, we also consider Qwen2.5-7B model. The maximum new token generation is set to 512, temperature 0.15, top-p is set to 1. We set the maximum number of tries with the debugger to 3. We use NVIDIA RTX 5070 GPU to run the experiments locally.

Experimental Results and Performance Metrics

Our method improved code generation a lot by using chain-of-thought (CoT) prompting combined with iterative debugging. We tested this on both test_v1.csv public test set and test_v1.csv with additional private test set. During the challenge, we used test_v1.csv with additional private test set to create CoT-only solutions. As shown in Table 2, for the Qwen2.5-Coder-14B-Instruct model, we got 82.6% accuracy on the test_v1.csv with additional private test set and 93% on the test_v1.csv public test set. The smaller Qwen2.5-7B model reached 81% accuracy on the test_v1.csv public test set.

After the challenge, we improved the system with CoT + Debugger. Since we no longer had access to test_v1.csv with additional private test set, we evaluated only on test_v1.csv shared test set. With this enhanced method, Qwen2.5-7B improved to 88% accuracy (7% higher) and Qwen2.5-Coder-14B-Instruct reached a perfect score of 100% (also 7% improvement).

Model	Accuracy (CoT) (%)		Accuracy (CoT + Debug) (%)
	(test_v1.csv) + private test set	test_v1.csv	test_v1.csv
Qwen2.5-7B	—	81	**88**
Qwen2.5-Coder-14B-Instruct	82.6	93	**100**

Table 2: Accuracy comparison table using CoT and CoT+Debugger approach.

5 Ablation Study

The biggest improvement was on test_v1.csv shared test set. Debugging fixed runtime errors and handled tricky edge cases, which made the larger model perfect on test_v1.csv shared test set evaluations. This shows that our multi-stage approach is very effective at improving code quality and making sure the code passes tests.

Evaluation and Error Analysis with GPT-5: We save the raw outputs from every attempt to disk, and partial results are stored along the way so we can resume if something interrupts the process. When the pipeline finishes, we create two JSON files: one with full explanations and one with only the ID and response fields for submission. This logging helps us track each iteration, see what types of errors occurred, which errors were fixed, and which ones persisted. It allows us to understand how failures were resolved over time in the debugging workflow.

To measure the impact of debugging, error analysis is administered using GPT-5 model as a judge

Category	Question	Response (CoT)	Response (CoT + Debugger)

Correctly generated w/ only CoT	**instruction:** একটি প্রদত্ত ম্যাট্রিক্সকে তার সারিগুলির যোগফল অনুযায়ী ক্রমবর্ধমান ক্রমে সাজানোর জন্য একটি ফাংশন লিখুন।
	Example:
	def sort_matrix(M):
	# your code
	return M
	test_list: ["assert sort_matrix([[1, 2, 3], [2, 4, 5], [1, 1, 1]])==[[1, 1, 1], [1, 2, 3], [2, 4, 5]]"]
		{
		"id": 2,
		"response": "def sort_matrix(M):\n # Calculate the sum of each row\n row_sums = [sum(row) for row in M]\n\n # Sort the rows based on their sums\n sorted_rows = sorted(M, key=lambda row: sum(row))\n\n # Return the sorted matrix\n return sorted_rows"
		}
			N/A

Correctly generated w/ CoT + Debugger	**instruction:** একটি প্রদত্ত তালিকা থেকে অদ্ভুত সংখ্যা অপসারণের জন্য একটি পাইথন ফাংশন লিখুন।
	Example:
	def remove_odd(l):
	# your code
	return l
	test_list: ["assert remove_odd([1,2,3]) == [2]"]
		def remove_odd(l):
		result = []
		for num in l:
		if num % 2 == 0:
		result.append(num)
		return result
			{
			"id": 402,
			"response": "def remove_odd(l):\n result = []\n for num in l:\n if num % 2 == 0:\n result.append(num)\n return result"}

Unable to generate correctly	**instruction:** একটি জটিল সংখ্যার কোণ পেতে একটি ফাংশন লিখুন। Example: def angle_complex(a,b): # your code return a test_list: ["assert angle_complex(0,1j)==1.5707963267948966 ", "assert angle_complex(0,1j)==1.5707963267948966 "]	{ "id": 114, "response": "import cmath\n\ndef angle_complex(a, b):\n return cmath.phase(complex(a, b))\n\nassert angle_complex(0, 1j) == 1.57j7963267948966\nassert angle_complex(we, 1j) == 1.57j7163267948966"}	{ "id": 114, "response": "import cmath\n\ndef angle_complex(a, b):\n return cmath.phase(complex(a, b))\n\nassert angle_complex(0, 1j) == 1.57j7963267948966\nassert angle_complex(we, 1j) == 1.57j7163267948966"}

Figure 5: Examples of solved problems by CoT and CoT+Debugger. An example of unsolved problems are also shown.

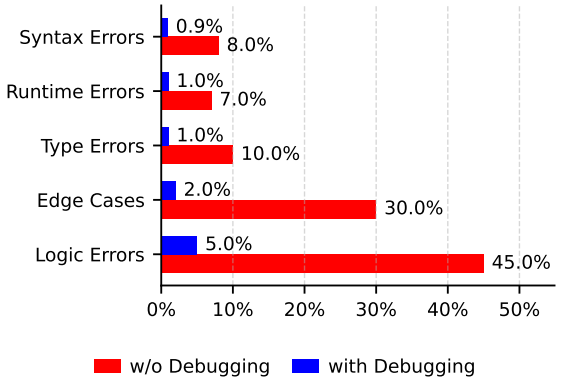

Figure 6: Using GPT-5 as a judge to analyze and track error reduction

that reveals a sharp drop in failure rates after debugging as shown in Figure 6. It shows the logic errors fall from 45% to 5%, edge-case issues from 30% to 2%, type errors from 10% to 1%, runtime errors from 7% to 1%, and syntax errors from 8% to 0.9%. These results clearly show that our debugging approach significantly improved the overall quality of the output.

6 Qualitative Analysis

We demonstrate a qualitative analysis of our approach. Examples of problems successfully solved using CoT and CoT + Debugger are shown in Figure 5. The first example is solved by CoT. In the second one, our approach correctly indents the code, hence return the correct python code. We further show a case that remains unsolved in the thrid example.

7 Conclusion

This paper addresses Bangla code generation from natural language instructions. By employing Chain-of-Thought prompting and iterative debugging, we achieved 100% on the test_v1.csv test set through systematic error correction. Future research directions include fine-tuning models on Bangla-specific programming datasets and developing larger instruction datasets tailored to Bangla programming.

References

Abhik Bhattacharjee, Tahmid Hasan, Wasi Uddin Ahmad, and Rifat Shahriyar. 2023. BanglaNLG and BanglaT5: Benchmarks and resources for evaluating low-resource natural language generation in Bangla. In *Findings of the Association for Computational Linguistics: EACL 2023*, pages 726–735, Dubrovnik, Croatia. Association for Computational Linguistics.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Arjun Guha, and 1 others. 2023. Multi-lingual code

generation and evaluation with mbxp. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*, pages 1064–1075.

Xinyun Chen, Lin Chan, Shiyu Yu, Pengcheng Bai, Shih-wei Chen, Ed Choi, Mingsheng Chen, and Denny Zhou. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46554.

David M. Eberhard, Gary F. Simons, and Charles D. Fennig. 2024. *Ethnologue: Languages of the World*, 27 edition. SIL International, Dallas, Texas.

Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E Gonzalez, and Ion Stoica. 2025. S*: Test time scaling for code generation. *arXiv preprint arXiv:2502.14382*.

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

Ziyin Luo and 1 others. 2024. Multilingual code generation: Challenges and opportunities. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*.

Nishat Raihan and Marcos Zampieri. 2025a. TigerLLM - a family of Bangla large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics.

Nishat Raihan and Marcos Zampieri. 2025b. TigerLLM - a family of Bangla large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 887–896, Vienna, Austria. Association for Computational Linguistics.

Baptiste Roziere and 1 others. 2024. Code llama: Open foundation models for code. *arXiv preprint arXiv:2408.09187*.

Qwen Team, Jinze Bai, Shuai Xu, Yankai Zhang, Zhenru Wang, Songyang Wang, Ziyan Li, Wang Wang, Li Wang, Siyuan Liu, Siyu Wang, and 1 others. 2024. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*.

Y. Wang and 1 others. 2024. Recent advances in bangla natural language processing. In *Proceedings of the 2024 Asia-Pacific Language Processing Conference*.

Jian Yang, Wei Zhang, Jiaxi Yang, Yibo Miao, Shanghaoran Quan, Zhenhe Wu, Qiyao Peng, Liqun Yang, Tianyu Liu, Zeyu Cui, and 1 others. 2025. Multi-agent collaboration for multilingual code instruction tuning. *arXiv preprint arXiv:2502.07487*.

Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Oleksandr Polozov, and Charles Sutton. 2023. Natural language to code generation in interactive data science notebooks. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 126–173, Toronto, Canada. Association for Computational Linguistics.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, Vancouver, Canada. Association for Computational Linguistics.

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