AdversaryAI at BLP-2025 Task 2: A Think, Refine, and Generate (TriGen) System with LoRA and Self-Refinement for Code Generation

Omar Faruqe Riyad
Shahjalal University of
Science and Technology
`riyad.omf@gmail.com`

Jahedul Alam Junaed
Shahjalal University of
Science and Technology
`nowshadjunaed@gmail.com`

Abstract

In this paper, we propose a system for generating Python code from Bangla prompts. Our approach fine-tunes open-source models with parameter-efficient techniques and leverages proprietary models via prompting. To enhance the reasoning of smaller models, we adopt a Chain-of-Thought (CoT) augmented fine-tuning, enabling them to learn intermediate reasoning steps before generating code. A self-refinement loop further improves performance by iteratively critiquing and correcting code based on execution feedback. We also employ few-shot prompting to guide inference more effectively. Applied to both open-source and proprietary models, this pipeline achieved its best results with Gemini 2.5 Pro, where our system ranked 4th on the competition leaderboard with a Pass@1 score of 0.85. We conclude with a detailed analysis of these findings.

1 Introduction

LLMs have rapidly advanced natural language processing and reasoning, achieving strong results in machine translation (Zhu et al., 2023; Feng et al., 2024), summarization (Zhang et al., 2024), dialogue (Wang et al., 2024), and complex reasoning (Lai et al., 2024). Their ability to interpret natural language and produce contextually appropriate outputs has opened new possibilities for both research and applications.

Within this broader progress, code generation has emerged as a promising direction. By translating natural language instructions into executable programs, LLMs can accelerate development, support education, and lower the barrier to programming. Benchmarks such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) show that state-of-the-art models can generate correct code from English prompts, underscoring their potential as programming assistants.

However, this progress is largely confined to high-resource languages, leaving languages like Bangla remain overlooked (Joel et al., 2024). The lack of datasets and tools limits training and evaluation, and direct translations often introduce errors or miss linguistic nuances. Closing this gap is vital for fairness, inclusivity, and broader access to programming. In this paper, we present a system for Bangla prompt to Python code generation. Our key contributions are:

- Adapting a pretrained LLM with LoRA for lightweight specialization.
- Expanding limited training data through a silver-to-gold augmentation strategy that generates and verifies high-quality examples.
- Enriching LLM training with two styles of chain-of-thought: concise hint-style and detailed step-by-step reasoning.
- Designing an iterative execution-feedback loop that allows the model to debug and improve its own solutions across multiple refinement steps.

2 Related Work

Large language models (LLMs) have transformed code generation, moving from rule-based systems (Gulwani, 2011) to Transformer architectures (Vaswani et al., 2017). Pretrained models like Codex (Chen et al., 2021), CodeT5 (Wang et al., 2021), and CodeGen (Nijkamp et al., 2022) set the paradigm of mapping natural language directly to executable code, establishing new program synthesis benchmarks.

Beyond full fine-tuning, parameter-efficient approaches such as LoRA (Hu et al., 2022) allow task adaptation with minimal overhead. Instruction tuning with synthetic data, including Evol-Instruct and CodeAlpaca (Luo et al., 2023; Chaudhary, 2023), further improves generalization. Reinforcement learning with human or execution feedback also aligns outputs with functional correctness (Ouyang et al., 2022; Le et al., 2022).

In parallel, code-focused LLMs such as Code Llama (Roziere et al., 2023), StarCoder (Li et al., 2023), DeepSeek-Coder (Guo et al., 2024), and WizardCoder (Luo et al., 2023) show that domain adaptation significantly boosts performance over general-purpose LLMs. Yet, most advances target high-resource languages like English. Models perform far worse with low-resource languages such as Bangla (Joel et al., 2024). Translation-based methods, multilingual pretraining, and cross-lingual prompting offer partial solutions (Zhu et al., 2023; Feng et al., 2024). Recently, Raihan et al. (2025b) introduced TigerCoder, a Bangla code LLM that surpassed multilingual baselines and showed translation alone cannot close the gap, emphasizing the need for native-language resources.

3 Task and Dataset

3.1 Task Overview

The Code Generation shared task (Raihan et al., 2025c) required generating Python programs from Bangla problem prompts. A solution was correct if it passed all hidden unit tests, with evaluation performed in a sandbox environment under resource constraints. Systems were ranked by Pass@1, the percentage of prompts solved correctly.

3.2 Dataset Overview

Participants were provided with trial, dev (Raihan et al., 2025a), and test (Raihan et al., 2025b) datasets. Each entry included a Bangla instruction, a reference Python solution (trial set only), and unit tests in the form of assert statements. During training, reference solutions and public tests were available, while at evaluation, only prompts were given. Appendix B shows an example of data and datasets distribution.

4 System Description

Our submission is built upon TriGen (Think, Refine, Generate), a multi-strategy approach that leverages both parameter-efficient fine-tuning of open-source models and advanced prompt engineering of large-scale, proprietary models. We describe our two primary systems below.

4.1 System A: Fine-tuning of Open Source models

4.1.1 Base Model Selection

We began by evaluating several open-source, instruction-tuned models, including Llama3 (3B) and Qwen3 (4B), to establish a performance baseline. As shown in Table 2, the Qwen3-4B-Instruct model demonstrated superior initial performance and was selected as the base model for our later experiments. We utilized LoRA (Hu et al., 2022), to train only a small set of adapter layers, keeping the base model's weights frozen. This approach reduces computational requirements while maintaining high performance.

4.1.2 Data Augmentation and Translation

To expand our limited training data, we adopted a "silver-to-gold" data augmentation strategy (Riyad et al., 2023). We used a powerful proprietary model, Gemini 2.5 Pro, to generate high-quality solutions for the entire dev set. We then executed these generated solutions against the provided test cases and filtered for only those that passed, creating a verified dev set. This high-quality dataset was then merged with the original trial set to create an augmented training corpus (trial + verified_dev).
To investigate the impact of language alignment with our model's English pre-training, we created a pure-English version of our augmented training set using Gemini 2.5 Pro for translation.

4.1.3 Chain-of-Thought (CoT) as Hint and Step

To further improve the model's logical reasoning, we integrated CoT into our fine-tuning process. The core idea behind CoT is that prompting a model to generate intermediate reasoning steps improves its ability to solve complex problems (Wei et al., 2022; Gonzalez et al., 2024). We extend this concept to "reasoning-augmented fine-tuning" (Chung et al., 2024). The hypothesis is that by training the model on examples that explicitly include a reasoning plan (Instruction + Plan -> Code), the model internalizes the process of algorithmic decomposition. We used LLMs to generate two distinct styles of reasoning:

- **Hint-style CoT**: Strategy-level cues (e.g., "divide the number", "compute gcd", "apply multiplication"), without revealing intermediate solutions or the final output.
- **Step-by-Step (SbS) CoT**: Detailed, sequential reasoning steps that explicitly describe the path toward the solution.

We then integrated the training set with this reasoning plan. Our results showed that while the hint-style CoT, which provides only high-level cues, led to a modest improvement, fine-tuning with explicit

SbS-style CoT produced a substantial gain in accuracy.

4.1.4 Few-Shot Prompting

We designed a system prompt to enforce the competition's strict output requirements and guide the model's logic. The prompt explicitly instructs the model to follow the function name and signature, handle edge cases, produce self-contained code, and generate necessary helper classes. We utilized a few-shot approach within the system prompt by providing a concrete example of the desired response.

4.1.5 Self-Refinement Loop

We implemented a self-refinement loop (Figure 1) inspired by (Madaan et al., 2023).

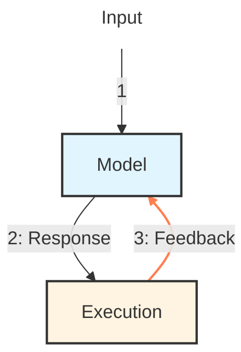

Figure 1: Execution and self-refinement loop

Think (Initial Generation): The model produces an initial code solution, which is executed against the provided public test case.

Refine (Self-Correction): For failed solutions, we construct a refinement prompt containing the original instruction, the failed code, and the corresponding error message or failed assertion from the execution environment. The model is then prompted to act as a debugger and generate a corrected solution.

Generate (Final Output): The refined code produced in the previous step becomes the new candidate solution. Iterating this loop up to three times yielded the best results.

4.2 System B: Gemini 2.5 Pro with few-shot and self-refinement

While our System-A performed well, we achieved our top-performing result by leveraging a large-scale model, Gemini 2.5 Pro. We combined few-shot prompting with self-refinement loop (Section-4.1.4 and Section-4.1.5). This approach achieved a final Pass@1 score of 0.85 on the hidden test set, outperforming all of our locally fine-tuned models.

5 Results

We began by fine-tuning several models on the trial set to establish a baseline. Among them, Qwen3-4B-Instruct achieved the best initial performance with a Pass@1 of 0.58 on the dev set (Table 2), so we used it for the later experiments.

Our first key finding was that translating the dataset to English, contrary to our initial hypothesis, resulted in a slight performance degradation to 0.50, suggesting that potential semantic shifts during translation outweighed the benefits of aligning with the model's primary pre-training language. We therefore proceeded with the Bangla-centric dataset for all subsequent fine-tuning experiments.

As shown in Table 1, each subsequent strategy yielded incremental gains. Augmenting the training data with a verified dev set improved the Pass@1 score to 0.54. Integrating Chain-of-Thought (CoT) as high-level hints provided a further boost to 0.56, while the more detailed Step-by-Step (SbS) CoT was significantly more effective, raising the score to 0.59. Our best fine-tuned system, which applied a self-refinement loop to the CoT-enhanced model's outputs, achieved a final Pass@1 of 0.62.

In parallel, we evaluated Gemini 2.5 Pro, a state-of-the-art proprietary model. A single-pass generation using a robust few-shot prompt achieved a score of 0.84. Applying our self-refinement loop to this model yielded our overall best result of **0.85**, which placed our team 4th on the official competition leaderboard.

6 Discussion

6.1 Error Analysis

A qualitative analysis of the failures reveals systematic challenges in both model reasoning and dataset construction. We categorize these errors into four primary themes, with detailed examples for each presented in Appendix C.

6.1.1 Ambiguous Instruction

A significant portion of errors originated from ambiguous or misleading problem statements. **Semantic ambiguity**, where the instruction was underspecified (Appendix C.1); **misleading instructions**, where the prompt suggested a simple algorithm but the test case required a more complex

System / Method	Training Data	Pass@1
Baseline (Qwen3)	trial (English)	0.50
Baseline (Qwen3)	trial (Bangla)	0.51
+ Data Augmentation	trial + verified_dev	0.54
+ CoT Fine-Tuning	trial + verified_dev + CoT (Hint)	0.56
+ CoT Fine-Tuning	trial + verified_dev + CoT (SbS)	0.59
+ CoT Fine-Tuning (Self-Refinement)	**trial + verified_dev + CoT (SbS)**	**0.62**
Gemini 2.5 Pro (Single-Pass)	N/A (Few-Shot)	0.84
Gemini 2.5 Pro (Self-Refinement)	**N/A (Few-Shot)**	**0.85**

Table 1: Pass@1 Performance of different systems on hidden **test set**.

one (Appendix C.2); and poor **translation quality**, which introduced ambiguity (Appendix C.3).

6.1.2 Failures of Constraint Adherence

Another major failure mode occurred when a model understood the general problem but failed to adhere to crucial, explicit constraints. This manifested as **test case ignorance**, where models, particularly smaller fine-tuned ones, prioritized a standard algorithm over the specific logic demanded by the test case (Appendix C.4). It also appeared as **memorization bias**, where models defaulted to common pre-trained patterns (e.g., a harmonic sum to n) instead of following a specific constraint (a sum to $n-1$) in the prompt (Appendix C.5).

6.1.3 Algorithmic Deficiencies

These errors represent Test case overfitting and reasoning failures. Models often produced hard-coded solutions that passed the visible test case but lacked generalizability (Appendix C.6); employed **sub-optimal or greedy algorithms** that failed in hidden test cases (Appendix C.7); and exhibited **semantic inconsistency**, resulting in runtime errors (Appendix C.8).

6.1.4 Inherent Dataset Challenges

Finally, a notable number of failures originated from structural flaws within the dataset itself: **Incorrect ground truth** in the test cases (Appendix C.9); **Incorrect test syntax** that would not be evaluated as intended by a standard Python interpreter (Appendix C.10); **Floating-Point Precision** issues (Appendix C.11); and **conflicting function signatures** between the instruction's example and the test case (Appendix C.12).

6.2 Findings

Our experiments yield several key insights. First, while the "silver-to-gold" augmentation consistently improved performance, translating the dataset back to English slightly degraded the results (Table 3), probably due to loss of semantic fidelity during the round-trip translation. This suggests that for our base model, which was already pre-trained on a large Bangla corpus, in-domain language consistency was more critical than alignment with its primary English pre-training. Second, our CoT experiments revealed a clear hierarchy of reasoning. The superior performance of Step-by-Step (SbS) plans over abstract hints indicates that models benefit more from explicit, structured problem decomposition. We conclude that SbS-style CoT provides a more effective learning signal, forcing the model to internalize a repeatable algorithmic workflow. Third, self-refinement enabled the model to reflect on execution errors and perform targeted repair, helping it resolve difficult edge cases that single-pass generation missed. Finally, as detailed in Section 6.1, many failures originated from the model's misinterpretation of ambiguous or misleading instructions, suggesting that clearer problem specifications could yield significant performance gains.

7 Limitations

Although TriGen achieves strong results, several limitations remain. The training data is relatively small, and part of it depends on silver-to-gold augmentation using a proprietary model, which may introduce bias. While the step-by-step reasoning and self-refinement loop improve performance, they rely heavily on the quality of the unit tests. When tests are incomplete, ambiguous, or contain errors, the refinement process can still converge on incorrect but test-passing solutions. Finally, the pipeline is optimized for single-function tasks, so its generalizability to more complex and diverse programming tasks has not yet been evaluated.

Conclusion

In this paper, we presented our TriGen system for Bangla prompt to Python code generation. Our experiments showed that combining data augmentation, chain-of-thought reasoning, and self-refinement led to our best fine-tuned system, while Gemini with few-shot prompting and self-refinement achieved the strongest overall result. Our analysis also reveals that limited training data and instruction ambiguity still constrain system reliability. In addition, our CoT supervision was generated automatically with Gemini and not manually validated. Even partial human inspection could provide higher-quality signals. In future work, we aim to expand high-quality Bangla code datasets, explore richer test case contexts during training and refinement to improve generalization, and extend TriGen to other low-resource languages.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca.

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*.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tai, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Alex Castro-Ros, Marie Pellat, Kevin Robinson, and 16 others. 2024. Scaling instruction-finetuned language models. *J. Mach. Learn. Res.*, 25(1).

Zhaopeng Feng, Yan Zhang, Hao Li, Bei Wu, Jiayu Liao, Wenqiang Liu, Jun Lang, Yang Feng, Jian Wu, and Zuozhu Liu. 2024. Tear: Improving llm-based machine translation with systematic self-refinement. *arXiv preprint arXiv:2402.16379*.

Andres Gonzalez, Md Zobaer Hossain, and Jahedul Alam Junaed. 2024. Numdecoders at semeval-2024 task 7: Flant5 and gpt enhanced with cot for numerical reasoning. In *Proceedings of the 18th International Workshop on Semantic Evaluation (SemEval-2024)*, pages 1260–1268.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

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*.

Sathvik Joel, Jie JW Wu, and Fatemeh H Fard. 2024. A survey on llm-based code generation for low-resource and domain-specific programming languages. *arXiv preprint arXiv:2410.03981*.

Xin Lai, Zhuotao Tian, Yukang Chen, Yanwei Li, Yuhui Yuan, Shu Liu, and Jiaya Jia. 2024. Lisa: Reasoning segmentation via large language model. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9579–9589.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328.

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

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: iterative refinement with self-feedback. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, and 1 others. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.

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).

Omar Faruqe Riyad, Trina Chakraborty, and Abhishek Dey. 2023. Team_Syrax at BLP-2023 task 1: Data augmentation and ensemble based approach for violence inciting text detection in Bangla. In *Proceedings of the First Workshop on Bangla Language Processing (BLP-2023)*, pages 247–254, Singapore. Association for Computational Linguistics.

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

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Peng Wang, Songshuo Lu, Yaohua Tang, Sijie Yan, Wei Xia, and Yuanjun Xiong. 2024. A full-duplex speech dialogue scheme based on large language model. *Advances in Neural Information Processing Systems*, 37:13372–13403.

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

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, 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 *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA. Curran Associates Inc.

Tianyi Zhang, Faisal Ladhak, Esin Durmus, Percy Liang, Kathleen McKeown, and Tatsunori B Hashimoto. 2024. Benchmarking large language models for news summarization. *Transactions of the Association for Computational Linguistics*, 12:39–57.

Wenhao Zhu, Hongyi Liu, Qingxiu Dong, Jingjing Xu, Shujian Huang, Lingpeng Kong, Jiajun Chen, and Lei Li. 2023. Multilingual machine translation with large language models: Empirical results and analysis. *arXiv preprint arXiv:2304.04675*.

A Performance on Dev-set and Language Impact

Model	Pass@1
TigerLLM-1B-it	0.15
Llama-3.2-3B-Instruct	0.25
Qwen2.5-Coder-3B-Instruct	0.51
Qwen3-4B-Instruct	0.58

Table 2: Pass@1 performance of different instruction-tuned models on the **dev-set**.

Model	Dataset	English	Bangla
Llama3	Dev-set	0.18	0.25
Qwen3	Test-set	0.50	0.51
Gemini 2.5 Pro	Test-set	0.82	0.85

Table 3: Impact of language (English vs. Bangla) on Pass@1 performance across models trained on trial set.

Table 2 shows Pass@1 performance of different models on the dev-set. Table 3 shows impact of language (English vs. Bangla) on Pass@1 performance.

B Dataset

The dataset was provided in JSON format, where each object contained a unique identifier, a Bangla instruction, an optional reference solution, and a list of test cases. Table 4 presents a sample entry, while Table 5 summarizes the number of instances in each set.

C Error Analysis - Examples

This appendix provides detailed examples for each category of error discussed in Section 6.1.

C.1 Semantic Ambiguity

Example 1

- **ID:** 15 (dev-set)
- **Instruction:** "একটি প্রদত্ত টুপল টুপল এর সংখ্যার গড় মান খুঁজে পেতে একটি ফাংশন লিখুন।"

Field	Example
`id`	231
`instruction`	প্রদত্ত অ্যারে থেকে সমান উপাদান জোড়া গণনা করার জন্য একটি পাইথন ফাংশন লিখুন।

`response`	`def count_Pairs(arr,n):`
	`  cnt = 0;`
	`  for i in range(n):`
	`    for j in range(i + 1,n):`
	`      if (arr[i] == arr[j]):`
	`        cnt += 1;`
	`  return cnt;`

`test_list`	`['assert count_Pairs([1,1,1,1],4) == 6', 'assert count_Pairs([1,5,1],3) == 1', 'assert count_Pairs([3,2,1,7,8,9],6) == 0']`

Table 4: An example entry from the Bangla code generation dataset.

Dataset	Number of Data
trial-set	74
dev-set	400
verified_dev	392
trial + verified_dev	466
test-set	500

Table 5: Distribution of datasets used in our experiments.

- **Analysis:** The instruction is ambiguous. It could refer to an overall average, row-wise averages, or column-wise averages. Only the test case clarified that column-wise averages were required.

Example 2

- **ID:** 66 (test-set)
- **Instruction:** "একটি আয়তক্ষেত্রে বর্গক্ষেত্রের সংখ্যা গণনা করার জন্য একটি পাইথন ফাংশন লিখুন।"
- **Model Response:**

~~~
```python
def count_Squares(m,n):
    return (m*n)
```
~~~

- **Analysis:** The problem is naturally interpreted as "how many 1×1 squares fit". That leads to m * n. But the test case clarified that it's about counting all possible squares (of all sizes).

C.2 Misleading Instruction

- **ID:** 5 (test-set)
- **Instruction:** "একটি স্ট্রিংকে ছোট অক্ষরে বিভক্ত করার জন্য একটি ফাংশন লিখুন।"
- **Analysis:** The instruction suggests we should just separate lowercase letters. However, the test case `split_lowerstring("AbCd") == ['bC', 'd']` reveals that a more complex logic is needed, where each new substring **starts** with a lowercase letter.

C.3 Translation Quality

Example 1

- **ID:** 15 (test-set)
- **Instruction:** "একটি প্রদত্ত অ্যারেতে পুনরাবৃত্তি না হওয়া উপাদানগুলির পণ্যটি খুঁজে পেতে একটি পাইথন ফাংশন লিখুন।"
- **Analysis:** The data point contains incorrect translation, where "product" was translated to 'পণ্য'. It has no contextual meaning.

Example 2

- **ID:** 67 (test-set)
- **Instruction:** "একটি পাইথন ফাংশন লিখুন যাতে সম এবং অদ্ভুত অঙ্কগুলির যোগফলের মধ্যে পার্থক্য খুঁজে পাওয়া যায়।"
- **Analysis:** The literal translation is incorrect in the mathematical context, where 'জোড়' and 'বেজোড়' should have been used instead of 'সম' and 'অদ্ভুত'.

C.4 Test Case Ignorance

- **ID:** 91 (test-set)
- **Instruction:** "প্রদত্ত অ্যারেতে k-তম উপাদান খুঁজে পেতে একটি ফাংশন লিখুন।"
- **Test Case:**

```
assert kth_element
([12,3,5,7,19], 5, 2) == 3
```

- **Analysis:** Models often defaulted to the standard "k-th smallest" algorithm (which would yield 5). They failed to adhere to the test case, which specified a positional lookup (the element at the second position of the unsorted array is 3).

C.5 Memorization Bias

- **ID:** 238 (test-set)
- **Instruction:** “n-1 এর হারমোনিক সমষ্টি গণনা করার জন্য একটি ফাংশন লিখুন।”
- **Model Response:** The model generated code to calculate the harmonic sum of ‘n’.
- **Analysis:** The model defaulted to the most common version of the harmonic sum problem, ignoring the specific ‘n-1’ constraint in the prompt. It defaulted to the patterns seen during pretraining.

C.6 Overfit Logic

- **ID:** 338 (test-set)
- **Instruction:** “একটি ফাংশন লিখুন যা প্রদত্ত দৈর্ঘ্যের ক্রমগুলি গণনা করে যার অ-নেতিবাচক উপসর্গ যোগফল রয়েছে যা প্রদত্ত মান দ্বারা উত্পন্ন হতে পারে।”
- **Test Case:** ‘assert bin_coff(4) == 2’
- **Model Response (Qwen3):**

~~~
```python
def bin_coff(n):
    if n <= 0: return 0
    if n == 1: return 1
    return 2
```
~~~

- **Analysis:** The smaller fine-tuned model generated a hardcoded solution that passes the single public test case but contains no generalizable algorithm.

C.7 Sub-optimal Algorithm Choice (Greedy Approach)

- **ID:** 29 (test-set)
- **Instruction:** “একটি প্রদত্ত স্ট্রিং এর অক্ষরগুলোকে পুনরায় সাজানো যায় কিনা তা পরীক্ষা করার জন্য একটি ফাংশন লিখুন যাতে একে অপরের সাথে সংলগ্ন দুটি অক্ষর ভিন্ন হয়।”
- **Analysis:** This problem requires careful handling of character frequencies to avoid getting ‘stuck.’ The optimal solution often involves a max-heap to prioritize placing the most frequent characters first. Our fine-tuned models often defaulted to a simpler but incorrect greedy approach that failed on more complex hidden test cases (e.g., “aaabc”).

C.8 Example: Semantic Error

- **ID:** 116 (test-set)
- **Instruction:** “দুটি প্রদত্ত সংখ্যার সাধারণ বিভাজকগুলির যোগফল খুঁজে বের করার জন্য একটি পাইথন ফাংশন লিখুন।”

```
def sum(a, b):
    # your code
    return a
```

- **Analysis:** The model generated a function named ‘sum’, which shadows the built-in Python ‘sum()’ function. The subsequent call to ‘sum()’ inside the function now refers to the function itself, not the Python built-in, causing an infinite recursion and a ‘RecursionError’ during execution. This represents a failure to consider the broader context of the programming language’s standard library.

C.9 Incorrect Ground Truth

- **ID:** 451 (test-set)
- **Test Case:**

```
assert upper_ctr('PYthon')
    == 1
```

- **Analysis:** The expected count of uppercase letters is incorrect (should be 2).

C.10 Incorrect Test Syntax

- **IDs:** 303, 426 (test-set)
- **Test Cases:**

```
assert pos_nos([-1,-2,1,2])
    == 1,2

assert neg_nos([-1,4,5,-6])
    == -1,-6
```

- **Analysis:** The right-hand side of these assertions is not a valid tuple. Python’s ‘assert’ syntax is ‘assert expression, [message]’. Consequently, the test ‘assert pos_nos(...) == 1,2’ is interpreted as ‘assert (pos_nos(...) == 1), 2’, where ‘2’ is the optional error message. This means the test would incorrectly pass if the function returned ‘1’. The correct syntax should have enclosed the expected output in parentheses, e.g., ‘== (1, 2)’.

C.11 Floating-Point Precision Issues

- **ID:** 129 (test-set)
- **Test Case:**

```
    assert circle_circumference(10)
        == 62.830000000000005
```

- **Analysis:** The test case expects a very specific floating-point number. A solution using Python's more accurate 'math.pi' would fail this test. This forces the model to reverse engineer a less accurate hardcoded value for π (e.g., 3.1415) to pass the test, penalizing standard and correct programming practices in favor of a specific floating-point representation.

C.12 Conflicting Function Signatures

- **ID:** 338 (test-set)
- **Instruction's Example Signature:**

```
def bin_coff(n, r):
    # your code
    return n
```

- **Test Case:**

```
    ['assert bin_coff(4) == 2']
```

- **Analysis:** The instruction provides an example function signature with two parameters ('n', 'r'), while the test case invokes the function with only a single argument ('n=4'). This creates a direct conflict that the model must resolve. A model that incorrectly prioritizes the instruction's example would generate a two-parameter function, leading to an immediate 'TypeError' at runtime when the single-argument test case is executed.

D Data Pre-processing and Augmentation Details

This section describes the data pre-processing, augmentation, and formatting pipelines used in our experiments.

D.1 Initial Data Cleaning and Normalization

Before any training, we applied several cleaning steps to the raw 'trial', 'dev', and 'test' datasets.

- **Newline Normalization:** We observed that some entries contained Windows-style newline characters (`\r\n`). All newlines were standardized to the Unix-style (`\n`) to prevent the model from learning and reproducing inconsistent line breaks.
- **Code Fence Enforcement:** To ensure the model learned the precise output format, we programmatically verified that every `response` in our training data was correctly enclosed in a ` ```python ... ``` ` code fence. Any responses missing these fences were automatically wrapped.

D.2 "Silver-to-Gold" Data Augmentation

To augment our training data, we generated a high-quality, verified version of the 'dev' set.

1. **Silver Data Generation:** We used the Gemini 2.5 Pro API to generate solutions for all problems in the 'dev' set, creating our initial "silver" dataset.
2. **Execution-Based Verification:** We then executed each generated solution against its corresponding unit tests.
3. **Gold Data Filtering:** Only the solutions that passed the test case were retained, resulting in a high-fidelity 'gold' or 'verified_dev' set. This set was then merged with the original 'trial' set to form our final augmented training corpus.

D.3 Translation

To investigate the impact of language, we translated all Bangla instructions into English. For translations, we compared the output of the Googletrans library with the Gemini 2.5 Pro API. Upon manual inspection, we found the Gemini-generated translations to have higher semantic fidelity and contextual accuracy. All final translation experiments were therefore conducted using the Gemini-translated datasets.

D.4 Dynamic Prompt Construction for Fine-Tuning

we constructed the prompts during the data loading phase to serve as a form of data augmentation and to provide richer context to the model. Our prompt construction function performed the following steps for each training example:

1. **Function Signature Extraction:** The public 'test_list' was parsed to programmatically extract the correct function name and signature. This information was explicitly added to the prompt to mitigate errors from conflicting or ambiguous signatures in the original instruction.

2. **Final Formatting:** The dynamically constructed user prompt was then formatted using the specific chat template of the model being trained (e.g., Qwen3 or Llama 3) to ensure proper tokenization and handling of special roles like 'system', 'user', and 'assistant'.

Finally, during the training process itself, we employed a loss mask to ensure the model only learned from the assistant's response tokens, ignoring the prompt tokens. This focuses the model's learning entirely on the target output.

E Experimental Setup

This section provides a detailed overview of the models, hyperparameters, and infrastructure used for our fine-tuning and inference experiments.

E.1 Fine-Tuning Infrastructure and Model Configuration

All fine-tuning experiments were conducted on a single T4 GPU with 16GB of VRAM, provided via Google Colab. To facilitate training on this hardware, we leveraged the `unsloth` library for memory-efficient model loading and optimization.

The base model for our fine-tuning experiments was `Qwen/Qwen3-4B-Instruct`. It was loaded in 8-bit precision with a maximum sequence length of 1024 tokens. We then applied Parameter-Efficient Fine-Tuning (PEFT) using the LoRA (Low-Rank Adaptation) methodology with the following configuration, as shown in Table 6.

LoRA Parameter	**Value**
Rank (`r`)	16
Alpha (`lora_alpha`)	32
Dropout (`lora_dropout`)	0.0
Bias	`none`

Table 6: LoRA configuration used for fine-tuning experiments.

E.2 Training Hyperparameters

We used the `SFTTrainer` from the TRL library for supervised fine-tuning. To prevent overfitting and select the best model checkpoint, we split our training data into a 90% training set and a 10% validation set. We enabled early stopping with a patience of 3 epochs, monitoring the `eval_loss` on the validation set. All models were trained for a maximum of 10 epochs. The key training hyperparameters are detailed in Table 7.

Hyperparameter	**Value**
Learning Rate	5×10^{-5}
Batch Size (per device)	4
Effective Batch Size	8
Optimizer	AdamW (8-bit)
LR Scheduler	Cosine
Warmup Ratio	0.1
Weight Decay	0.01
Precision	FP16

Table 7: Key hyperparameters used for the SFTTrainer.

E.3 Inference Strategy

E.3.1 Fine-Tuned Model Inference

For generating solutions from our fine-tuned models, we employed a batched generation strategy to maximize throughput. The tokenizer's padding side was set to 'left' to ensure correct output in a batch context. We used deterministic decoding by setting 'do_sample=False'. A batch size of 8 was used, with a maximum generation length of 1024 tokens.

E.3.2 API-Based Model Inference

For our experiments with the Gemini 2.5 Pro API, we developed a separate inference script. This script processed prompts sequentially but included a retry mechanism with exponential backoff to handle API rate limits and errors. To ensure persistence and prevent data loss, results were saved to a JSONL file after each successful API call, making the process fully resumable. The generation was performed with a low temperature of 0.1 to favor deterministic and correct code.

E.4 Evaluation

The official evaluation metric for this shared task is the **pass rate**, also referred to as **Pass@1**. This metric is defined as the percentage of problems for which a system's generated code passes all hidden unit tests when executed in a sandboxed environment. A higher pass rate corresponds to a higher rank on the competition leaderboard. All scores reported in this paper uses **Pass@1**.

F Prompts Used for Generation and Refinement

This section details the system and user prompts employed for our experiments.

F.1 System Prompt for Initial Code Generation

Figure 2 shows the system prompt that was used for the “Think” (initial generation) stage. It is designed to enforce strict output formatting and guide the model’s logic.

F.2 User Prompt Template for Initial Generation

The user prompt was dynamically constructed for each problem, providing the instruction and the public test case as structured input (Figure 3).

F.3 System Prompt for Self-Refinement

For the “Refine” (self-correction) stage, a different system prompt was used to frame the task as a debugging and code review exercise (Figure 4).

F.4 User Prompt Template for Self-Refinement

The user prompt for the refinement loop was dynamically constructed to include the execution feedback (Figure 5).

~~~
You are an expert, competition-focused Python programmer.
Your task is to generate lean, correct, and executable Python code to solve the given problem.

**Core Requirements:**
- Your entire response MUST be a single, fenced code block starting with ```python and ending with
    ```.
- Only generate the function or class requested. Do not include example usage or print statements.
- The code must be self-contained and runnable.
- The function must use the exact name and signature from the provided instruction or test cases.
- If helper classes are required by the test cases, you must define them.

**Code Style Requirements:**
- **No Explanations:** Do not add comments that explain your reasoning, translate the instruction, or
     describe basic Python functionality.
- **Minimal Docstrings:** If you include a docstring, it MUST be a single line explaining the
    function's high-level purpose. Do not use multi-line docstrings or describe arguments (Args/
    Returns).
- **Example of desired style:**
  ```python
  def add_numbers(a, b):
      \"\"\"Returns the sum of two numbers.\"\"\"
      return a + b
  ```

**Logic Requirements:**
- If the instruction is ambiguous, the provided test cases are the source of truth.
- Prioritize correctness and efficiency. Do not generate pseudo-code.
~~~

Figure 2: System Prompt

```
{instruction}

You must implement the solution strictly following the function signature in the instruction. If any
    helper classes or methods are needed to run the following test cases, you must define them as
    well.

Test cases:
{test_list}
```

Figure 3: User prompt

~~~
You are an expert Python code reviewer and debugger.
Your task is to fix a flawed Python function that failed to solve a programming problem.

You will be presented with:
1.  The original problem description.
2.  Your previous, incorrect code submission.
3.  The specific error message or failed test case that caused the failure.

**--- Your Goal: Analyze and Correct ---**
- **Analyze the Error:** Carefully examine the failed test case or error message. This is the most
    important clue to understanding the mistake.
- **Identify the Flaw:** Compare the failed test case with your incorrect code to pinpoint the
    logical flaw.
- **Implement the Fix:** Write a new, corrected version of the Python code that directly addresses
    the identified flaw and will pass the test case.
- **Generalize:** Ensure the corrected solution is robust and handles other potential edge cases, not
     just the single failed test.

**--- Output Requirements ---**
- Your entire response MUST be a single fenced code block beginning with ```python and ending with
    ```.
- Do not include any text, explanations, or apologies before or after the code block.
- Provide only the corrected, complete, and self-contained Python code.

**Example of the ONLY acceptable output format:**
  ```python
  def add_numbers(a, b):
      \"\"\"Returns the sum of two numbers.\"\"\"
      return a + b
  ~~~
```

Figure 4: Self-refinement System prompt

```
**Problem:**
{instruction}

**Your Incorrect Code:**
{failed_code}

**Reason for Failure:**
{error_info}

Based on the error, provide a corrected and complete Python code
that solves the problem and passes the failed test case.
```

Figure 5: Self-refinement user prompt
```