Educational Code Generation using LLMs with Self-Refinement

Ziyue Gong
Dept. ECE
University of Toronto
joy.gong@mail.utoronto.ca

Zihan Wan
Dept. ECE
University of Toronto
zihanzane.wan@mail.utoronto.ca

Yuan Wang
Dept. ECE
University of Toronto
ywang.wang@mail.utoronto.ca

Mahta Miandashti
Dept. ECE
University of Toronto
mahta.miandashti@mail.utoronto.ca

Abstract

We propose a code-generation agent that performs initial code synthesis, self-refinement, and explanation generation for LeetCode-style problems. Leveraging the pre-trained StarCoder2-7B model via prompt engineering, our pipeline accepts natural language problem statements, produces executable solutions, iteratively improves failing code, and generates human-readable explanations. We evaluate on a curated dataset of 12 LeetCode problems (balanced across easy, medium, and hard difficulties) and measure performance in terms of test pass rates and explanation quality.

Assentation of Teamwork

Ziyue Gong — Initial code generation, test execution, part of explanation generation.
Zihan Wan — Refine loop.
Yuan Wang — Evaluation.
Mahta Miandashti — Part of explanation generation.

1 Introduction

LLM code assistants like GitHub Copilot, Claude, and Cursor are transforming software development, offering both productivity gains and challenges to traditional workflows. To examine their construction and capabilities, we design a code-generation agent using a pre-trained model for initial generation, iterative refinement, and explanation, evaluated on LeetCode tasks. This project applies prompt engineering and measures performance to better understand the strengths and limitations of current code-generation LLMs.

2 Preliminaries and Problem Formulation

Given a NL description of a coding question and the starter code, the objective is to generate Python code that passes the task test suite and produce a clear, human-readable text explaining the solution. To achieve this, we select a pre-trained model balancing performance and size, prepare a 15-problem LeetCode dataset for prompt engineering and demonstration, and a three-stage pipeline—initial code generation with testing, self-refinement, and explanation generation—that outputs correct solutions with explanations.

3 Design

Data
Construct easy_ds (5 easy), medium_ds (5 medium), hard_ds (5 hard), example_ds (first entry from each set for few-shot prompts) from newfacade/LeetCodeDataset [1], with each entry containing fields 'task_id', 'prompt', 'completion', 'entry_point', 'test', 'query', 'response', 'input_output', 'meta' (detailed usage mentioned below).

Model
Uses pretrained bigcode/starcoder2-7b [2]

Agent Workflow
We propose a pipeline as shown in the diagram.

Step1: Initial Code Generation + test
Create a few-shot FIM prompt from example_ds (3 examples + target). For the target, provide only <prefix> (problem) and let the model fill <middle> (solution). Extract clean code, build a test script, and run it with pistonpy, capturing any errors.

[Optional] Step2: Self-refine stage:
After an initial attempt fails, we iteratively "repair" the code by feeding the model (a) the task again, (b) the last failing implementation, and (c) optionally the previous test error (truncated).
Each iteration applies the same refine template to produce new code, runs the code, tests it, and updates the refine log.
If the error from pistonpy testing is empty (test passes), skip this step and proceed.
Finally, return the first passing version; if none pass, return the best runnable version along with brief logs from each round.

Step3: Explanation Generation
Using few-shot FIM prompting, generate a clean, coherent explanation for the final code.

Agent performance evaluation
For each explanation, compute readability (Flesch Reading Ease, Flesch-Kincaid Grade, SMOG, Gunning Fog) to approximate pedagogical clarity. Compute discourse coherence with spaCy + TextDescriptives (first-/second-order coherence) to assess logical flow. Measure code–explanation alignment by embedding both texts with a code-aware encoder (CodeBERT) and taking cosine similarity.

4 Methodology

Step1: Initial Code Generation + test
Code generation prompt template:

```
ONE_PROMPT_BLOCK_TEMPLATE = """<start>
<fim_prefix>
{prefix}     # data['query'], text description of the problem with starter code
<fim_middle>
{middle}     # data['completion'], python code solution
<end>"""

TARGET_BLOCK_TEMPLATE = """<start>
<fim_prefix>
```

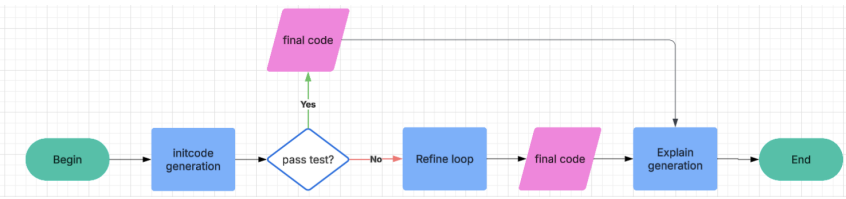

Figure 1: Pipeline Diagram

```
{prefix}     # data['query'], text description of the problem with starter code
<fim_middle>"""
```

Code Execution:

```
# Assemble an executable test script
merged_code = (
        q["prompt"].rstrip()        # necessary python import
        + "\n\n"
        sol.rstrip()                    # candidate implementation
        + "\n\n"
        + q["test"].rstrip()            # assert-based tests
        + "\n\n"
        + f"check({q['entry_point']})\n"  # driver that triggers the tests
    )
'''
# Run tests with
piston.run(
        language="python",
        version=py_version,
        code=merged_code,
    )

# Sample return by piston.run():
>>>  {..., 'stderr': '', ...}
```

[Optional] Step2: Self-refine stage:

```
REFINE_PROMPT_TEMPLATE = """
<issue_start>username_0: {instruction}  # "Fix code: ..." (+ optional last error 500 chars)
{function_start}    # starter stub: class/method signature
{buggy_function}    # last failing code (full text)
Upvotes: 100<issue_comment>
username_1: Sure, here is the fixed code.
'''python
{function_start}    # model is directed output a full solution under this header"""
```

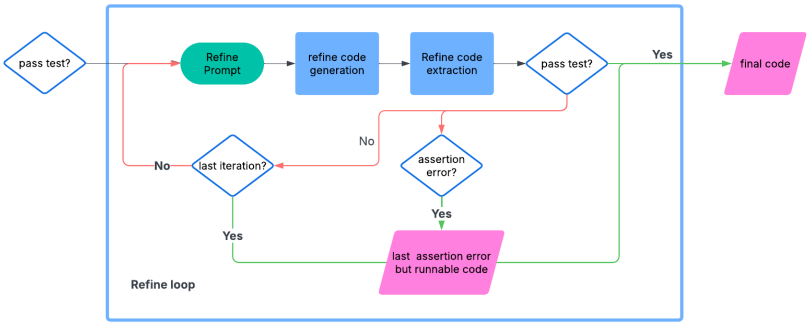

Figure 2: Refine loop process.

The self-refine stage is built using methodology flow described as pseudo-code below:

Listing 1: Self-refine stage pseudo-code

```
current_buggy_code = init_code
current_error = ""  # initial test error is NOT injected in round 1

for t in range(MAX_REFINES):
```

```
    instruction = make_instruction(problem_desc, current_error)  # append error iff
    non-empty
    prompt = REFINE_PROMPT_TEMPLATE.format(
        instruction=instruction,
        buggy_function=current_buggy_code,
        function_start=starter_code
    )

    llm_out = model.generate(prompt)
    candidate = extract_refine_out(llm_out)  # parse fenced code

    # Guardrails before testing
    if is_empty(candidate) or missing_target_def(candidate) or too_long(candidate,
    2000):
        # auto-complete the signature and skip updating current_error
        candidate = prepend_header_and_indent(starter_code, candidate)  # 4-space
    indent
        # (no test result update if extraction failed last round)

    # Build test script and run
    merged = assemble_test_script(imports=q["prompt"], impl=candidate,
                                  tests=q["test"], entry=q["entry_point"])
    result = piston.run(language="python", version=py_version, code=merged)

    if result.run.stderr == "" and result.run.code == 0:
        return candidate  # early stop on pass

    # Prepare next round
    current_buggy_code = candidate
    current_error = truncate(result.run.stderr, 500)
```

Step3: Explanation Generation

```
FEW_SHOT_EXPLAIN_TEMP = """<start>
## Task
{query}  # data['query']: text description + starter code
## Solution
{completion} # data['completion']: code solution
## Explanation
{explain}  # gpt generated explanation for completion
<end>"""

TARGET_EXPLAIN_TEMP = """<start>
<fim_prefix>
## Task
{query}  # data['query']: text description + starter code
## Solution
{completion} # data['completion']: code solution
<fim_suffix>
## Explanation
<fim_middle>"""
```

Agent Performance Evaluation

Readability scoring
Rule-based indices (Flesch, FK, SMOG, Gunning Fog) via `textstat`.
Coherence estimation
Sentence-level coherence features using `spaCy` + `textdescriptives`.
Semantic alignment
Embeddings for both `final_code` and `final_explain` using a code-aware model (e.g., `CodeBERT` from `transformers`); similarity via cosine (`scikit-learn`).
Aggregation & I/O
`pandas` for loading Easy/Medium/Hard result tables, grouping, and export.

5 Numerical Experiment

Goal We evaluate the agent's final natural-language explanations and their relationship to task outcomes. Per-item metrics are computed and aggregated by difficulty (Easy/Medium/Hard). We report the test values in tables (learning curves omitted).

Experiments For each item, we compute: (i) readability (Flesch, FK Grade, SMOG, Gunning Fog), (ii) discourse coherence (first/second order), and (iii) code–explanation semantic alignment (cosine similarity between embeddings). Aggregates (means) are reported per difficulty.

Settings and Parameters

Component	Setting / Value
Readability indices	`textstat`: Flesch, FK Grade, SMOG, Gunning Fog (defaults)
Coherence	`spaCy` (`en_core_web_sm`) + `textdescriptives` (1st/2nd order)
Embeddings	`microsoft/codebert-base` (HF Transformers)
Similarity	cosine similarity (scikit-learn)
Aggregation	mean over items, grouped by difficulty; values rounded to 2 decimals

Table 1: Evaluation parameters used in `final_explanation_evaluation.ipynb`.

Difficulty	Pass rate	Pass iter	Flesch ↑	FK Grade ↓	SMOG ↓	Gunning Fog ↓	1st-order coh. ↑	2nd-order coh. ↑	Code–expl. sim ↑
Easy	0.25	1.00	65.85	8.26	10.42	10.50	0.62	0.60	0.90
Medium	0.25	1.00	45.25	13.19	13.33	15.69	0.23	0.23	0.92
Hard	0.25	4.00	60.61	8.71	11.45	11.13	0.44	0.36	0.91

Table 2: Agent performance metrics by difficulty. ↑/↓ indicate desirable direction.

6 Discussion

With a 4-iteration cap, the agent solves 3/12 tasks (25%); Easy/Medium typically succeed in $\sim$1 iteration, while Hard often hits the cap. Explanations are most readable/coherent on Easy and least on Medium. Code–explanation alignment is consistently high (cosine ≥ 0.90), indicating faithfulness. Readability/coherence show weak coupling with success—accuracy appears budget-limited rather than style-limited. Limits: $n = 12$, automatic proxies.

7 Conclusions

We built a three-stage code-generation agent that solves LeetCode problems using a pre-trained LLM, combining few-shot FIM prompting, automated testing, iterative refinement, and explanation generation.

We explored different model versions, refined the pipeline design, and experimented with prompt templates, decoding parameters (e.g., temperature, top-n), and the use of special tokens and tags to simplify parsing and evaluation.

Future improvements include testing on a larger dataset and enhancing the refinement loop by providing more targeted error feedback—pinpointing the specific buggy logic rather than just assertion errors.

References

[1] newfacade, "LeetCodeDataset," GitHub, 2025.
`https://github.com/newfacade/LeetCodeDataset/tree/main/data`

[2] "bigcode/starcoder2-7b," Hugging Face, Jun. 06, 2024.
`https://huggingface.co/bigcode/starcoder2-7b`

[3] A. Lozhkov *et al.*, "StarCoder 2 and The Stack v2: The Next Generation," *arXiv*, Feb. 29, 2024.
`https://arxiv.org/abs/2402.19173`

[4] "pistonpy," *PyPI*, Dec. 07, 2021.
`https://pypi.org/project/pistonpy/`