Using Lower-Scale LLMs to Efficiently Generate Code

EE 782 - Group Project Report

Dattaraj Salunkhe, Saurabh Srivastava, Swayam Patel
22B1296, 22B1294, 22B1816

***Abstract*—This group project investigates the ability of lower-scale large language models (LLMs) — smaller, more resource-efficient transformer-based models — to generate efficient and robust code. We describe our approach to use smaller models in order to increase efficiency, our experiments and our thought process behind our approach. Finally, we discuss conclusions and directions for future work.**

***Index Terms*—LLM, code generation, efficiency, fine-tuning, evaluation**

I. Introduction

Recent progress in transformer-based language models has dramatically improved automatic code generation. However, large models are expensive to train and serve. This project explores whether lower-scale LLMs (models with fewer parameters and reduced compute requirements) can produce code that is both efficient and robust for common programming tasks. The goals of this project are: (1) design a lightweight pipeline for fine-tuning and prompting smaller models for code generation; (2) explore whether the pipeline performs competitively on the benchmark datasets.

II. Methodology

Our methodology is organized into two primary approaches. The first is a two-stage pipeline that focuses on syntactic robustness, while the second extends this framework with an additional stage aimed at improving logical correctness and reasoning.

A. Two-Stage Pipeline

The two-stage pipeline consists of a code generation model followed by a syntax-focused critique model. In this setup, the generator first produces the candidate code solution. This code is then passed through a compiler, which provides syntactic validation. If syntax errors are detected, the compiler's error messages, along with the generated code itself, are forwarded to the critic model. The critic is responsible for analyzing these signals and producing actionable feedback for the generator, prompting it to refine the code in subsequent iterations. This creates a feedback loop that incrementally improves syntactic accuracy. Algorithm 1 shows the pseudocode for the 2-stage pipeline.

B. Three-Stage Pipeline

The three-stage pipeline augments the earlier approach by incorporating an additional model dedicated to logical verification. The first two stages operate in the same manner as described above. Once syntactic validity is achieved, the code is executed against the provided test cases. If any test fails, the code and corresponding error information are supplied to the third model, which focuses on diagnosing logical issues. Its feedback is then passed back to the generator. After regeneration, the code must again pass through the original two-stage process to ensure that improved logic has not reintroduced syntax errors. This cyclical process allows both syntactic and logical refinement. ALgorithm 2 shows the pseudocode for the 3-stage pipeline.

C. Models Considered

Our exploration began with the DistilGPT2 [2] model as the code generator. However, its performance was inadequate: because it is derived from DistilBERT and primarily optimized for natural language generation, it frequently produced non-functional outputs such as comments rather than executable code. Even after pre-training the model on the CodeParrot [6] Python dataset, it failed to produce meaningful programs, often outputting only sequences of import statements. From this, we concluded that code generation requires models with significantly larger capacity, as iterative feedback alone cannot compensate for a model that fundamentally lacks the capability to reason about code structure.

Following discussions with our professor, we also evaluated the possibility of using DeBERTa [3]. However, being an encoder-only architecture, it would necessitate constructing a custom decoder for generation. Moreover, its architecture is not inherently aligned with generative tasks involving code, leading us to discard this option.

We subsequently adopted the StarCoder (1B) base model [4]. While it performed satisfactorily on simpler tasks—often succeeding after a few iterations of the two-stage pipeline—it struggled on the MBPP [10] dataset. Its shortcomings stemmed from being a base rather than an instruct model: it tended to ignore prompt instructions, frequently generating test cases despite explicit prompts not to. This motivated our final choice of the Qwen2.5 Coder Instruct model [5] (1.5B) for code generation. This model already demonstrated strong performance

Algorithm 1 Two-Stage Pipeline

```
Require:
    taskPrompt: Natural language description of the coding problem
    testCases: List of test inputs and expected outputs
    testImports: Import statements required for the test environment
    maxIterations: Maximum number of refinement steps
Ensure:
    finalImplementation: Python code that passes all tests
    reflectionLog: Chronological list of generated reflections
currentReflection ← ””
reflectionLog ← [ ]
for $iteration = 1$ to maxIterations do
    // Code generation
    candidateCode ← GENERATECODE(taskPrompt, currentReflection)
    // Execution and testing
    executionResult ← EXECUTETESTS(candidateCode, testImports, testCases)
                        ▷ Returns: passed, stdout, stderr, returnCode
    // Evaluation
    if executionResult.passed then
        finalImplementation ← candidateCode
        return finalImplementation, reflectionLog
    end if
    // Reflection generation
    rawReflection ← CRITICANALYZEFAILURE(
                        candidateCode,
                         testImports + testCases,
                         executionResult.stderr,
                         executionResult.stdout)
    // Reflection cleaning
    currentReflection ← CLEANREFLECTION(rawReflection)
    // Logging
    Append currentReflection to reflectionLog
end for
finalImplementation ← candidateCode
return finalImplementation, reflectionLog
```

on MBPP, and we aimed to assess whether our pipeline could further improve its results.

For the syntax critic in the two-stage pipeline, we initially selected Google's FlanT5-Small model [7]. In the three-stage pipeline, however, we replaced it with the DeepSeek Coder (1.3B) instruct [8] model. This decision was based on empirical observations: FlanT5-Small occasionally produced malformed feedback or random character sequences, whereas DeepSeek Coder consistently generated more reliable and code-aware critiques. Importantly, this change affected only the critic; the overall methodology remained the same.

For the logic critic in the three-stage pipeline, we employed Microsoft's Phi-3 Mini 128k Instruct model [9]. We selected

Algorithm 2 Three-Stage Pipeline for Iterative Code Refinement

```
Require:
    taskPrompt: Natural-language description of the coding problem
    testList: List of tests (inputs + expected outputs)
    maxIters: Maximum number of refinement steps
Ensure:
    success: Whether the code passed all tests
    finalCode: Final generated implementation
    iterations: Number of iterations used
code ← null
feedback ← null
for $iteration = 1$ to maxIters do
    // Code generation
    candidateCode ← GENERATECANDIDATECODE(taskPrompt, feedback)
    // Function name harmonization
    code ← HARMONIZEFUNCTIONNAME(candidateCode, JOIN(testList, “\n”))
    // Syntax checking
    syntaxError ← CHECKSYNTAX(code)
    if syntaxError $\neq$ null then
        feedback ← SYNTAXFEEDBACK(code, syntaxError)
        continue
    end if
    // Logical execution + testing
    returnCode, stdout, stderr ← RUNTESTS(code, testList)
    if returnCode = 0 and stderr = ”” then
        return (true, code, iteration)
    end if
    // Logic-level feedback
    feedback ← LOGICFEEDBACK(code, stderr, testList)
end for
return (false, code, maxIters)
```

this model because its design emphasizes strong reasoning and instruction-following capabilities, which suggested that it would be well-suited for identifying logical flaws in generated code. We expected Phi-3 Mini to provide more coherent and actionable feedback on test-case failures compared to smaller or less reasoning-oriented models. This made it an appropriate choice for guiding the generator toward improved logical correctness in subsequent iterations. Table 1 shows the parameter size of each model mentioned above.

III. RESULTS

We evaluated both the two-stage and three-stage pipelines on the MBPP dataset. Owing to computational constraints, the experiments were conducted on a randomly selected subset of 25 tasks from the dataset. For the two-stage pipeline, we obtained an accuracy of 84%. As expected from the

TABLE I
MODELS USED AND THEIR PARAMETER SIZES

Model	**Parameter Size**
DistilGPT2	82M
DeBERTa	86M
StarCoder Base	1.0B
Qwen2.5 Coder Instruct	1.5B
FlanT5-Small	80M
DeepSeek Coder	1.3B
Phi-3 Mini 128K Instruct	3.8B

capabilities of the Qwen model, several test cases were solved correctly on the first iteration. However, a number of additional tasks were successfully completed in subsequent iterations, demonstrating that the feedback loop contributed meaningfully to improved performance. The maximum number of iterations allowed was 3.

For the three-stage pipeline, we observed an accuracy of 100%, which is a bit erroneous. Upon examining the execution logs, we suspect that a small number of tasks may have been incorrectly marked as passing. While the majority of tasks appear to be evaluated correctly, a few instances indicated unexpected behavior that we were unable to attribute to a specific issue in the implementation. Nevertheless, aside from these isolated cases, the three-stage pipeline behaved as intended.

IV. Discussion and Conclusions

We also referred to the Reflexion [1] framework, which proposes a structured approach for leveraging large-scale LLMs to generate code and reflect upon their own outputs. The effectiveness of this architecture relies heavily on the model's reasoning ability, which in turn is strongly correlated with model size. Since our objective was to work with lower-parameter models, and given our observations of the reasoning capabilities exhibited by the models used in our pipelines, we concluded that directly implementing the Reflexion structure would not be beneficial in this setting. Lower-scale LLMs tend to produce noisy or uninformative reflections, thereby failing to meet the core requirement of the Reflexion approach—high-quality, self-generated reasoning.

Our experiments showed that extremely small LLMs such as DistilGPT2 and DeBERTa are unable to generate usable code, even after additional pre-training. Mid-scale models like StarCoder-Base can produce syntactically valid solutions for simple problems when combined with our two-stage pipeline, but they perform poorly on benchmark datasets such as MBPP. In contrast, models such as Qwen, which already demonstrate reasonable performance on MBPP, exhibit moderate improvement when integrated into our pipeline. While the pipeline improves reliability, further gains are dependent on enhancing the quality of critic feedback, as the critic models occasionally fail to provide sufficiently actionable or coherent guidance.

References

[1] A. Datta, A. Aljohani, and H. Do, *Secure Code Generation at Scale with Reflexion*, arXiv preprint arXiv:2511.03898, 2025. Available at: https://arxiv.org/abs/2511.03898.

[2] Hugging Face, *DistilGPT2*, Available at: https://huggingface.co/distilgpt2.

[3] Microsoft, *DeBERTa Base*, Available at: https://huggingface.co/microsoft/deberta-base.

[4] BigCode, *StarCoderBase-1B*, Available at: https://huggingface.co/bigcode/starcoderbase-1b.

[5] Qwen Team and Alibaba Group, *Qwen2.5-Coder-1.5B-Instruct*, Available at: https://huggingface.co/Qwen/Qwen2.5-Coder-1.5B-Instruct.

[6] Hugging Face, *CodeParrot – Python Code Dataset*, Available at: https://huggingface.co/codeparrot/codeparrot.

[7] Google, *FLAN-T5-Small*, Available at: https://huggingface.co/google/flan-t5-small.

[8] DeepSeek AI, *DeepSeek-Coder-1.3B-Instruct*, Available at: https://huggingface.co/deepseek-ai/deepseek-coder-1.3b-instruct.

[9] Microsoft, *Phi-3-Mini-128K-Instruct*, Available at: https://huggingface.co/microsoft/Phi-3-mini-128k-instruct.

[10] Google Research, *MBPP – Mostly Basic Python Problems Dataset*, Available at: https://huggingface.co/datasets/google-research-datasets/mbpp.