Runtime Analysis for Iterative Code Refinement

Nathan J. Whitehead
University of Texas at Dallas
nathan.whitehead@utdallas.edu

***Abstract*—Large Language Models (LLMs) have demonstrated remarkable capabilities in code generation, yet their outputs often require manual verification and correction by users. This paper explores an automated approach where an LLM not only generates code but also executes it, evaluates the output against expected results, and iteratively refines the solution until it meets the given specifications. By integrating execution-based validation and feedback-driven refinement, this system aims to reduce human intervention in debugging and improve code correctness, as well as enable lower-powered models to provide correct results while reducing computation overhead. We discuss potential applications, including interactive programming environments and automated software prototyping, and outline challenges such as execution safety, performance trade-offs, and constraints in generalizing across diverse coding tasks. Our work seeks to advance the automation of code generation by enabling LLMs to self-correct through empirical validation, bridging the gap between AI-generated code and reliable, executable solutions.**

I. Introduction

Large language models (LLMs) have demonstrated remarkable capabilities in natural language understanding and code generation. As their adoption increases, users seek ways to interact with these models in a more structured and automated manner. One emerging challenge is the need for LLMs to not only generate responses but also verify and refine their outputs based on execution results. This is particularly relevant in environments such as Jupyter Notebooks, where iterative code refinement can be useful for debugging, learning, and experimentation.

Currently, existing solutions are unable to autonomously validate generated code and refine outputs iteratively. Current AI coding assistants such as GitHub Copilot and OpenAI's ChatGPT provide valuable suggestions but lack built-in mechanisms for executing code, verifying results, and refining responses dynamically without tedious human intervention. Our approach integrates automated validation to iteratively refine generated code until it satisfies the user's specifications.

A. Example

To illustrate the problem, consider a simple task where a user requests a Python function to compute the Fibonacci sequency up to a given integer n. The user provides the following prompt to the LLM: "Write a Python function that returns a list containing the first n Fibonacci numbers." The LLM then responds with the following output:

```
def fibonacci(n):
  fib = [0, 1]
  for i in range(n-2):
    fib.append(fib[i] + fib[i+1])
  return fib
```

While this code is syntactically correct, it will fail for small values of n. (e.g., `fibonacci(1)` should return [0], but will instead raise an index error). Without execution-based validation, the user would need to manually inspect and debug the issue, increasing cognitive load on the user, and wasting time.

When we apply execution-based refinement, the system will run the generated code to test various inputs, e.g., `fibonacci(1)`, `fibonacci(2)`, `fibonacci(10)`, and detect the error on $n = 1$. The system will then refine the function based on the observed failures by passing the program output back to the LLM. A corrected version will then be generated, such as:

```
def fibonacci(n):
  if n<=0:
    return []
  elif n==1:
    return [0]
  fib = [0,1]
  for i in range(n-2):
    fib.append(fib[i] + fib[i+1])
```

This process will then repeat until the code passes test cases (either user-provided or LLM-generated), or until the process times out. In this example, the second function is valid, and so this will be provided to the user.

II. Background

A. Large Language Models for Code Generation

Recent advancements in large language models (LLMs) have significantly improved automated code generation. Models such as ChatGPT and Claude leverage vast amounts of training data to generate syntactically and semantically correct code given a natural language prompt. These models have demonstrated remarkable performance in completing programming tasks. However, LLMs are fundamentally probabilistic and do not inherently verify the correctness of the generated code. As a result, users must manually refine and validate the generated code, which can be time-consuming and error-prone.

B. Execution-Based Validation

To bridge the gap between code generation and feedback, execution-based validation techniques have emerged to help close that loop without necessitating human input. Unlike static analysis methods that assess code structure without running it, execution-based approaches involve running the generated code and evaluating its output against expected results. This process can be facilitated through unit tests, assert statements, or predefined correctness criteria. By leveraging execution results, models can be guided to refine and improve their outputs, reducing the reliance on manual verification by users.

C. Self-Correcting Code Generation

A promising direction in automated code generation is integrating self-correction mechanisms that iteratively refine generated code based on execution feedback. Self-correction involves identifying errors through execution validation, diagnosing potential causes, and generating improved versions of the code. While traditional LLMs generate code in a single pass, a self-correcting system introduces a feedback loop that can enhance reliability and efficiency.

D. Static vs. Dynamic Analysis in Code Validation

In the broader context of program correctness, two primary validation strategies exist: static and dynamic analysis. Static analysis techniques, such as type checking and linting can detect potential errors before execution but often struggle with runtime-specific issues. Dynamic analysis, on the other hand, involves executing the program in a controlled environment to observe its behavior. While dynamic approaches provide higher confidence in correctness, they require computational resources and may be constrained by test case coverage and security concerns. Integrating both strategies can lead to a more robust validation framework, particularly for LLM-generated code.

E. Challenges in Automated Code Refinement

Despite progress in LLM-driven code generation and validation, several challenges remain. Execution-based validation requires well-defined test cases, which are not always available. Additionally, generated code may pass validation but this doesn't guarantee there are no inefficiencies, security vulnerabilities, or edge-case failures. Addressing these challenges requires advancements in automated debugging, adaptive learning strategies, and hybrid validation techniques that combine formal verification with empirical testing.

III. Related Work

Several studies have explored the integration of both static code analysis methods and dynamic execution-based code validation. For instance, the work by [1] showcases an on-device model which adjusts prompts based on validation results. Similarly, [2] propose a preference-based learning approach that refines code generation outputs based on user feedback. [3] propose an interactive code generation system intended to assist users in refining generated code through a combination of dynamic testing and user feedback. These studies highlight the importance of integrating validation mechanisms into the code generation process to improve reliability and user satisfaction. [4] propose a formal system for providing behavioral specifications to code generation models, enabling the generation of code that satisfies specific user-defined constraints and tests.

There are also a number of studies which incorporate the related approach of static analysis. For example [5] integrates static analysis techniques with dynamic testing to improve the security of generated code.

IV. Problem Definition

Automated code generation using Large Language Models (LLMs) has significantly advanced in recent years, yet the generated code often lacks guaranteed correctness, requiring manual verification and refinement. Part of the issue is that LLMs operate as probabilistic models, generating code based on learned patterns rather than explicit execution-based validation. This introduces a gap between code that seems right, and follows the syntax rules of its language, but fails at runtime due to logical errors, missing dependencies, or incorrect assumptions about the codebase.

We define the problem as follows:

Givens:

- A natural language prompt P describing a programming task.
- An LLM M capable of generating code C from P.
- A specification S that defines expected behavior or correctness citeria, either explicitly through test cases or implicitly through execution outputs.

Objective:

- Develop an iterative refinement process where M generates an initial code candidate C_0, executes it, evaluates the output against S, and updates C accordingly until C_n satisfies S.

A. Constraints and Challenges

Execution safety. Running unverified, automatically generated code can involve security risks, including infinite loops, memory overflows, and malicious execution. This can be mitigated using techniques developed for preexisting platforms that run user-generated code such as onlinegdb or google colab.

Ambiguous Specifications. In many cases, the user will not provide enough context to cover all cases, and robust test cases may not be available, which makes validation challenging.

Computational Cost. Iterative refinement may significantly increase computational requirements, and the cost will only increase as the complexity of the programming task increases.

Generality Across Tasks. Ensuring that the approach works across a diverse set of programming tasks across multiple complexity levels and domain constraints adds additional challenge, and continuous development may be required to cover additional domains.

V. Approach

We propose a general framework for iterative code refinement that integrates language model generation, execution-based validation, and feedback-driven correction. This abstraction is not tied to a specific model, dataset, or implementation, and can be instantiated in multiple environments depending on task complexity, available infrastructure, and domain requirements.

A. Overview of the Refinement Framework

The core idea is to view the process of code generation as an optimization loop over a space of program candidates. The system seeks a program C_n such that C_n satisfies a specification S derived from a user prompt P. At each step, the framework evaluates the current candidate's behavior and uses the results to guide the next refinement.

1) **Code Generation:** A language model generates an initial program candidate C_0 given a natural language task description P.
2) **Execution:** The code C_i is executed on a predefined set of test cases or inputs derived from specification S.
3) **Validation:** The output of the execution is compared against the expected results in S. If C_i fails to meet the specification, error information E_i is extracted.
4) **Refinement:** A new prompt is constructed containing P, C_i, and E_i. The model is then asked to generate a refined version C_{i+1}.
5) **Iteration:** Steps 2–4 are repeated until C_n satisfies S, or a maximum iteration limit is reached.

B. Algorithmic Formulation

The refinement process can be formalized as follows:

Input: Prompt P, Specification S, LLM M, Max iterations k
$C_0 \leftarrow M(P)$
for $i = 0$ to k **do**
  $R_i \leftarrow \text{execute}(C_i)$
  if R_i satisfies S **then**
    return C_i
  else
    $E_i \leftarrow \text{extract\_errors}(R_i)$
    $C_{i+1} \leftarrow M(P, C_i, E_i)$
  end if
end for
return `FAILURE`

This loop abstracts away the specific mechanics of code execution and validation, allowing multiple strategies to be applied depending on context. For example, specifications could consist of unit tests, behavior traces, or even human feedback.

C. Feedback Construction

The effectiveness of the refinement depends heavily on the structure and content of the feedback passed to the model. Our framework supports modular feedback construction strategies, which may include:

- Raw exception messages (e.g., stack traces or assertion failures)
- Annotated diffs between expected and observed outputs
- Natural language summaries of validation failures

The structure of the feedback prompt can be adjusted to guide the model more explicitly depending on the nature of the task or the capability of the LLM.

D. Separation from Implementation

Although our implementation uses specific tools (e.g., Llama 3.2, Docker, Python), the framework does not assume any particular technology stack. Execution may be performed in a sandbox, a virtual machine, or a remote server. Similarly, refinement can be done by any generative model capable of producing code given feedback. We refer the reader to Section VI for details of our concrete instantiation.

E. Illustrative Example

Consider a task where the prompt asks for a function that returns the square of an integer. The LLM initially generates a function that returns $n * n$. During execution, a test case reveals the function raises an error on input type `None`. This triggers refinement, where the model is given both the original prompt and the observed exception. It then adds a guard clause checking for null input. This showcases how error-driven feedback can incrementally produce robust solutions.

F. Evaluation Considerations

The framework can be evaluated on several axes, including:

- **Convergence Rate:** How many iterations are required on average for code to satisfy S.
- **Correctness:** Final pass rate on a test suite.
- **Resource Efficiency:** Time and computation consumed across iterations.

- **Model Robustness:** Sensitivity of success rate to quality of error feedback.

VI. Implementation

The implementation of our iterative refinement system consists of four main components: prompt-based code generation, containerized execution, result validation, and feedback-guided refinement. This section describes the tools, libraries, and strategies used to implement each stage.

A. Libraries and Tools

We used the following libraries and frameworks in our system:

- **Llama 3.2 (local)** – A freely available LLM running locally using ollama, which enables fast inference on local computing resources.
- **Docker** – Used to create isolated execution environments for running unverified code.
- **Python 3.13** – Used within containers to execute generated programs.
- **Python modules** – Available python modules which help with running the LLM calls, and that the generated code could use as needed. Including re, requests, logging, os, json, numpy, collections, matplotlib, subprocess.
- **MBPP Dataset** – The Mostly Basic Python Problems dataset, containing nearly 1000 problems with ground-truth assert-based test cases.

B. Code Generation

The LLM is prompted using a template that includes the natural language description of the problem along with a request to implement a Python function. Each prompt is constructed dynamically and provided as input to the Llama 3.2 model using a local inference API. Initial generation does not include any test cases; these are introduced only in the validation stage. Verbatim prompts are provided in Appendix A.

C. Execution Environment

To prevent unsafe or infinite execution, all generated code is run inside a Docker container configured with:

- Resource limits (1GB RAM, 1 CPU core)
- Execution timeout of 15 seconds
- No access to the host filesystem or network

Each generated function is wrapped in a Python script that includes the candidate solution and one or more test cases drawn from the MBPP dataset. The script is copied into the container and run using the standard Python interpreter. Execution output is captured using `stdout/stderr` redirection and parsed for validation.

D. Validation

Validation is done using the `assert` statements provided in the MBPP dataset. These statements serve as ground truth for correctness. If the generated code raises an exception or fails an assertion, the error message and traceback are extracted from the captured output. This information forms the basis of feedback provided to the LLM.

E. Feedback Mechanism and Refinement

When validation fails, a new prompt is constructed for the LLM. This refinement prompt includes:

- The original problem description
- The previously generated code
- The error message or unexpected output

This process is repeated until either the function passes all validation tests or a maximum of 5 refinement iterations has been reached.

F. Implementation Challenges

A few notable complications arose during development:

- **LLM Output Quality**: Initial attempts to use a smaller local model with only 1 billion parameters resulted in nearly all generated code failing validation. These outputs also frequently failed to follow expected formatting, making it difficult to extract usable code automatically.
- **Code Extraction Robustness**: Even with a more capable LLM, reliably extracting code was challenging. Early attempts to format the model output as JSON were largely unsuccessful, as the model often violated JSON syntax. Switching to Markdown-style output improved consistency, but some completions still omitted closing code blocks or included explanatory text within the code, requiring additional post-processing logic. Still, occasional failures to extract code correctly were observed and handled by re-prompting the LLM.
- **Computing Resource Constraints**: Running the LLM locally meant that the system was limited by the available CPU and RAM on the host machine. This meant that running the tests took several hours, and the system was unable to run multiple tests in parallel. This could be improved by using a more powerful machine or running the LLM on a cloud service.

VII. Evaluation

To evaluate the effectiveness of our iterative code refinement framework, we conducted experiments focusing on the following key metrics:

- **Correctness:** The percentage of tasks for which the final generated code passed all test cases, especially compared to those which passed without refinement.
- **Convergence Rate:** The average number of iterations required for the framework to produce a correct solution.
- **Robustness:** The framework's ability to handle diverse prompts, including ambiguous or incomplete specifications.

A. Experimental Setup

We evaluated our framework using the MBPP (Mostly Basic Python Problems) dataset, which consists of 974 Python programming tasks with ground-truth test cases. Each task was provided as a natural language prompt to the LLM, and the framework iteratively refined the generated code until it satisfied the test cases or reached the maximum iteration limit of 5.

The experiments were conducted on a local machine with the following specifications:

- CPU: Apple M3
- RAM: 16GB
- LLM: Llama 3.2 running locally with 3.2 billion parameters

B. Results

Correctness: Out of the 974 tasks, the framework successfully generated correct solutions for 45.5% of the tasks. The remaining 54.5% failed mainly due to the weakness of the model being used. However, when comparing the results to the baseline of using the LLM without refinement, the framework improved correctness by 46%. The model was able to generate correct solutions for 27% of tasks without refinement, and 45.5% with refinement. It's also possible that more correct solutions could be generated with more rounds of refinement, but we limited the number of iterations to 5 to avoid excessive computation.

Convergence Rate: On average for problems where a solution was found, the framework required 1.97 iterations to produce a correct solution. Tasks with well-defined test cases and detailed prompts converged faster, while tasks with edge cases or ambiguous specifications required more iterations.

Robustness: The framework demonstrated robustness in handling diverse prompts, but performance degraded for tasks with incomplete specifications or highly complex logic. In many cases, the framework often failed to converge within the iteration limit. This would likely improve with a more powerful model, as the LLM used in this experiment was limited to 3.2 billion parameters.

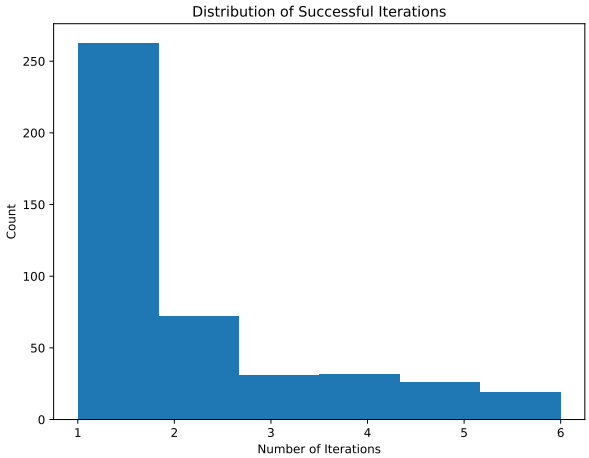

Fig. 1. Iteration results for the iterative code refinement framework.

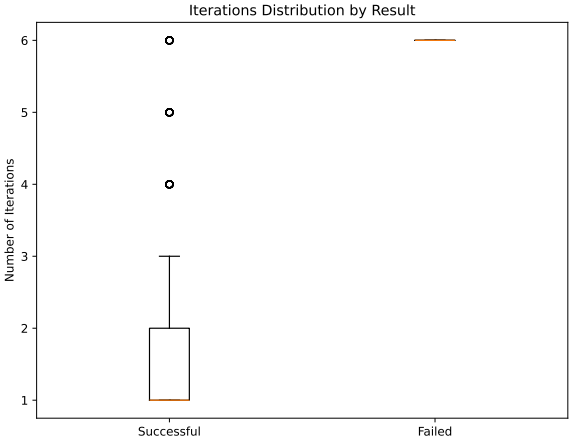

Fig. 2. Iteration results for the iterative code refinement framework.

C. Discussion

The results demonstrate that our framework effectively improves the correctness of LLM-generated code through iterative refinement. However, the computational cost and reliance on well-defined test cases highlight areas for future improvement. Optimizing the feedback mechanism and incorporating static analysis techniques could further enhance the framework's efficiency and robustness.

Furthermore, the low-budget model used in this experiment was unable to generate correct solutions for many tasks, and so the results may not be representative of the performance of larger models which can handle more complex problems. Future work will explore the use of larger models, as well as the integration of static analysis techniques to improve the framework's performance.

VIII. Conclusion

We have presented a framework for iterative code refinement that leverages execution-based validation and feedback-driven correction to improve the correctness of LLM-generated code. By integrating these components, we aim to reduce the reliance on manual verification and enhance the reliability of AI-generated code. Our evaluation on the MBPP dataset demonstrates the framework's potential to improve correctness and convergence rates, while also highlighting challenges related to execution safety, computational cost, and robustness across diverse tasks. Future work will focus on optimizing the framework's performance, exploring the integration of static analysis techniques, and evaluating the approach across a wider range of programming tasks and domains. By addressing these challenges, we aim to advance the automation of code generation and refinement, ultimately bridging the gap between AI-generated code and reliable, executable solutions.

Appendix A
LLM Prompts

The following are the prompts used to generate code and provide feedback to the LLM during the iterative refinement process.

First, the prompt used to generate the initial code is as follows:

```
# llm settings
PROGRAMMING_LANGUAGE = "python"
HOSTNAME = "localhost"
PORT = 11434
OLLAMA_API_URL = f"http://{HOSTNAME}:{PORT}/api/generate"
MODEL = "llama3.2:latest"
MAX_ITER = 5

DEFAULT_PROMPT = f"""
You are a {PROGRAMMING_LANGUAGE} expert. You will be given a prompt to create a {PROGRAMMING_LANGUAGE} funcion.
Your task is to create a {PROGRAMMING_LANGUAGE} function that solves the problem described in the prompt.
Create the simplest function that solves the problem.

**IMPORTANT**
- The response should be in markdown format, with the code block labeled as '{PROGRAMMING_LANGUAGE}'.
- The response should contain two parts: explanation and code, in that order.
- Example: Explanation sentences.\n```{PROGRAMMING_LANGUAGE}\n# code here\n```
- There should only be ONE explanation section, and ONE code block in the response.
- No other text should be included in the response.

**EXPLANATION RULES**
- The explanation should be no more than 3 sentences.
- The explanation should describe the code, the approach, and any important details.

**CODE RULES**
- The function should be a valid {PROGRAMMING_LANGUAGE} function.
- The function should be named `candidate`.
- The function should not contain any print statements.
- The function should contain explanatory comments.
- Only output a single function. If you are unsure, output the most universal version of the function.
- If subfunctions are needed, they should be defined inside the main function.
- No code should be outside the function.
- No test cases should be included in the code block.
"""
```

The prompt used for subsequent iterations is as follows:

```
CORRECTION_PROMPT = f"""
You are a {PROGRAMMING_LANGUAGE} expert. You help to correct user-generated code and fix errors or failed tests.
The code below is not valid. You will be given:
- the original prompt for the code
- the code that was generated
- the error message from the test

Your task is to correct the code and make it valid and pass the test.
The user code is not necessarily well-structured, so you should feel
free to rewrite, refactor, and improve the code as needed.

The following rules must apply to your response:

**IMPORTANT**
- The response should be in markdown format, with the code block labeled as '{PROGRAMMING_LANGUAGE}'.
- The response should contain two parts: explanation and code, in that order.
- Example: Explanation sentences.\n```{PROGRAMMING_LANGUAGE}\n# code here\n```
- There should only be ONE explanation section, and ONE code block in the response.
- No other text should be included in the response.
- The function produced MUST be named `candidate`, or the test cases will fail.

**EXPLANATION RULES**
- The explanation should be no more than 3 sentences.
- The explanation should describe the code, the approach, and any important details.

**CODE RULES**
- The function should be a valid {PROGRAMMING_LANGUAGE} function.
- The function should be named `candidate`.
- The function should not contain any print statements.
- The function should contain explanatory comments.
- Only output a single function. If you are unsure, output the most universal version of the function.
- If subfunctions are needed, they should be defined inside the main function.
- No code should be outside the function.
- No test cases should be included in the code block.
```

```
See the below for the original prompt, the code that was generated, and the error message.

The original prompt is:

%%%PROMPT%%%

The code that was generated is:

```{PROGRAMMING_LANGUAGE}
%%%CODE%%%
```

The error message is:
```
%%%ERROR%%%
```

"""
```

References

[1] H. Su, "Llm-based code development model with active prompt adjustment and on-device validation," 2025.

[2] L. T. et al, "Codelutra: Boosting llm code generation via preference-guided refinement," 2024.

[3] S. F. et al, "Llm-based test-driven interactive code generation: User study and empirical evaluation," IEEE Transactions on Software Engineering, vol. 50, no. 9, pp. 2254–2268, 2024.

[4] K. H. L. et al, "Effective llm-driven code generation with pythoness," 2025.

[5] A. N. et al, "Autosafecoder: A multi-agent framework for securing llm code generation through static analysis and fuzz testing," in 38th Conference on Neural Information Processing Systems. NeurIPS, 2024.