MASARYK UNIVERSITY

FACULTY OF INFORMATICS

LLMs with Test Feedback for Program Synthesis

Bachelor's Thesis

MARCEL NADZAM

Brno, Spring 2025

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

LLMs with Test Feedback for Program Synthesis

Bachelor's Thesis

MARCEL NADZAM

Advisor: doc. RNDr. Petr Novotný, Ph.D.

Brno, Spring 2025


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

During the preparation of this thesis, I used the following AI tools:

- Grammarly for grammar checking
- ChatGPT to improve my writing style and assist with programming

I declare that I used these tools in accordance with the principles of academic integrity. I reviewed all content and take full responsibility for it.

Marcel Nadzam

Advisor: doc. RNDr. Petr Novotný, Ph.D.

Acknowledgements

I would like to thank my family for their support throughout my studies. I am also grateful to my advisor for his guidance, my friends for their company and distractions, and the Faculty of Informatics at Masaryk University and the Hugging Face community for providing the resources that made this work possible.

Abstract

This thesis presents a feedback-driven pipeline for automated program synthesis using large language models (LLMs). The system generates Python functions from user-defined descriptions and validates them against corresponding unit tests. If the code fails, the test results are returned to the LLM for refinement. The pipeline is designed to run on Aura, a shared academic Linux server, and supports open-source models available via the Hugging Face framework.

The background section introduces the architecture and history of LLMs. The complete implementation of the pipeline is explained, including modules for prompt management, test execution, and model interaction. Experimental evaluation was conducted using three Qwen 2.5 Coder models of different sizes to compare performance across various task categories.

Results show that the feedback loop works best when the model's initial approach is mostly correct but contains minor bugs. In cases where the approach is fundamentally flawed, the system falls back to a reset mechanism that discards failed strategies and allows the model to try alternatives. The thesis concludes that automated feedback can be an effective method for improving LLM-generated code.

Keywords

Large Language Models, Prompt Engineering, Unit Testing, Iterative Refinement, Program Synthesis, Python

Contents

List of Tables

List of Figures

Introduction

Over the last few years, LLMs have become widespread across many areas, but their impact has been especially notable in software development. These models, trained on vast datasets of natural text and code, can now translate descriptions of non-trivial tasks into functional software, a capability unimaginable just a few years ago. However, LLM-generated code is often unreliable. Users must frequently inspect and correct the generated code manually or provide feedback to the model for revisions.

This thesis proposes a solution by creating a feedback driven pipeline that automatically synthesizes Python functions. Users define the required behavior through textual prompts and corresponding unit tests. If the implementation fails any tests, the pipeline feeds test outcomes back to the LLM, prompting iterative refinement until the solution passes all tests or a predefined limit is reached. The implemented system runs on Aura, a Linux server provided by Masaryk University, and utilizes open-source models provided through the Hugging Face framework. The experimental evaluation assesses pipeline effectiveness across diverse programming tasks using models with varying capabilities. It focuses on understanding when the iterative feedback loop effectively improves generated code and where this approach reaches its limits.

The thesis consists of four main chapters. Chapter 1 provides background on large language models, their evolution, fundamental architectures, and their applications in program synthesis. Chapter 2 describes operating within the Aura computing environment, covering software installation and model execution. Chapter 3 describes the complete design and implementation of the pipeline, detailing its core components. Chapter 4 details the experimental methodology and results, comparing model performance, analyzing task categories, and evaluating various prompting strategies.

1 Background

This chapter introduces Large Language Models (LLMs) and covers the basics necessary to follow the rest of the thesis.

1.1 What is a Language Model?

A language model is a type of AI designed to generate human language. By being trained on large amounts of data, these models aim to learn patterns, structures, and relationships between words. At their core, language models predict the most likely next token in a given sequence. For example, given the input "The sun rises in the", the model might predict "morning" as the next word.

This simple concept, coupled with enormous amounts of resources, enables language models to generate coherent text, respond to questions, or even write code.

1.2 History of LLMs

Language models have come a long way, transitioning from simple statistical methods to advanced transformer-based models capable of generating functioning code. This section provides an overview of the most important developments relevant to the thesis.

1.2.1 Early Language Models

Before the arrival of neural networks, language modeling used statistical methods such as n-grams. These models predicted the next word in a sequence by analyzing probabilities based on the preceding few words. For example, an n-gram model might calculate the probability of a word based on the three words that came before it. [1]

These methods could not understand larger contexts and outputs were repetitive and nonsensical.

1.2.2 The Neural Network Era

The introduction of neural networks marked a breakthrough in language models. Feedforward Neural Networks and Recurrent Neural Networks (RNNs) allow language models to understand relationships between words. However, RNNs still struggled with understanding the context of longer texts, often 'forgetting' earlier information as the input length increased. To address this limitation, Long Short-Term Memory (LSTM) models have been developed, which retain information over longer sequences by using memory cells to control what information to keep or forget. [1, 2]

Another breakthrough was in word embeddings, such as Word2Vec, which encoded words into vector spaces. This approach allowed models to capture semantic relationships between words—such as "king" being related to "queen", similarly to how "man" is related to "woman", enabling language models to understand context better. [3]

Despite this progress, these architectures rely on a sequential processing approach: each token in the input must be processed in turn, with each step depending on the previous one. This design prevents parallel processing, making it inefficient.

1.2.3 The Transformer Revolution

In 2017, the paper Attention Is All You Need [4] presented transformers as a solution to the limitations of sequential models like LSTMs. Transformers rely on positional encoding and self-attention, allowing the model to simultaneously process all parts (tokens) of the context, and determine how much each word depends on others. [1, 5]

Transformers enabled greater scalability, faster training, and improved contextual understanding. They became the foundation of modern large language models. GPT demonstrated the effectiveness of pre-training on large datasets, while BERT (2019) excelled at understanding context by processing text bidirectionally. [1]

These innovations meant that now, for the first time, language models have become useful for complex tasks such as summarization, question answering, and coding.

1.2.4 Specialization into Coder LLMs

As general-purpose transformers gained popularity, researchers started looking into specialized applications, particularly in programming. This led to the development of coder LLMs trained on datasets of code repositories, such as GitHub. These LLMs showed that specialized LLMs can perform just as well or better than general-purpose LLMs while being smaller [6]. Beyond generating new code, these coder LLMs proved useful for debugging and understanding code which simplified the workflow of software developers. Open-source models such as CodeLlama, StarCoder, or Qwen were developed, enabling users to run and experiment with LLMs locally (as is done in this thesis).

1.3 How LLMs Work

This section provides an overview of how LLMs work, with a focus on their architecture, training processes, and key components which are essential to understand the main program of this thesis.

1.3.1 The Transformer architecture

The core of most modern LLMs is the transformer architecture. This sub-section explains the fundamental concepts of transformers and why they are more efficient than other approaches.

Tokenization

Before an LLM can process input, it must divide the text into smaller, more manageable parts known as tokens. These tokens may represent entire words, subwords, or individual characters. [3]

Embeddings

After dividing the text into tokens, the model maps each token to a vector in a high-dimensional embedding space, where similar concepts are placed closer together. For example, words such as "car" and "vehicle" may have similar embeddings because of their related

meanings. This analogy does not entirely hold, since it only describes embeddings of full words, while tokens in actual transformers are often fractions of words, making the embedding process very opaque. Nevertheless, embeddings still enable LLMs to capture contextual relationships beyond the surface level, thereby supporting the generation of coherent text and code. [3]

In addition to token embeddings, transformers use positional embeddings to encode the order of tokens in a sequence. Since transformers process tokens in parallel without an inherent sense of sequence, positional embeddings provide the necessary information about each token's position, preserving the order of the input.

Self-Attention

The self-attention mechanism is the main innovation of the transformer architecture. Instead of processing text sequentially from left to right, self-attention uses matrix operations to determine the relationships between each token and all other tokens in the sequence. This approach provides rich contextual information for each token, and, unlike earlier models, self-attention allows the model to analyze all tokens simultaneously, which improves efficiency significantly—particularly on modern GPUs optimized for highly parallelized computations. [3, 7]

From Input to Output

Transformers consist of multiple stacked layers which contain several sub-components that progressively refine the input representation. In a simplified form, the pipeline operates as follows [3, 7]:

1. **Tokenization and Embedding:** Raw text is converted into tokens and mapped to a high-dimensional embedding space.

2. **Multiple Self-Attention and Feed-Forward Layers:** A series of layers that enable the model to understand the context of the entire input.

3. **Output Layer:** The model generates a probability distribution over the next possible tokens and selects the most likely one.

4. **Iteration:** The model appends the selected token to the sequence and repeats the process until it meets a stopping condition, such as predicting a stopping token or reaching a predefined maximum size.

Context Window Due to hardware limitations, LLMs can process only a limited number of tokens. This limitation forces a maximum input size, at which the model can no longer understand the full context and older portions of the input need to be summarized or discarded to fit within the available context window.

1.3.2 Transforming a Model into an Assistant

Another significant component of LLMs is the system prompt. While an LLM can generate text, it lacks a role or purpose and is not an assistant. To convert an LLM into a chatbot, an initial prompt is added to the user's input. An initial prompt specifies the desired behavior of the model. For example, *"The following is a conversation between a helpful code assistant and a user ..."*. By defining the role, tone, and constraints, the system prompt restricts the model's responses, which makes the LLM usable and limits potentially harmful or inappropriate outputs.

1.3.3 The Training Process

After the transformer architecture is established, the most computationally demanding process follows—the training process. This consists of pre-training and, optionally, fine-tuning.

Pre-training

Pre-training sets the foundation of an LLM and transforms it into a broad language understanding system. In the case of coder LLMs, a significant portion of the training data consists of open-source code repositories (such as those available on GitHub), rather than solely human-written text.

During training, models are exposed to a large dataset of examples and learn to improve their predictions by comparing outputs with

actual data and adjusting their weights through the backpropagation algorithm. [8]

Fine-tuning

After pre-training provides a broad foundation of language understanding, some models use fine-tuning to tailor the model to a specific task. In this phase, the pre-trained model is further trained on a smaller, targeted dataset.

Model Size and Parameters

A commonly encountered number when researching LLMs is their parameter count, often expressed in billions (such as 7B, 32B). These parameters represent the model's weights, which are numbers adjusted during training. The weights determine how the model processes each token to generate new ones. The training framework gradually refines the weights from their initial random values to meaningful numbers that enable the generation of coherent text and code. In general, larger models have a greater capacity for learning complex patterns. [9]

However, larger models come with increased hardware costs. They consume more memory and space, and result in longer response times. This thesis compares the performance of smaller and larger models across different tasks to determine their advantages.

1.4 LLMs and Code Generation

While LLMs have demonstrated impressive capabilities in code generation, they are far from perfect and often fail to produce correct code on the first attempt. Several factors influence the accuracy of the generated code. One of them is the capability of the LLM itself. However, other factors affect LLMs in a manner similar to how they affect human programmers.

When a programmer is given an assignment and must write correct code on the first attempt, without testing, debugging, or running the code, there is a high likelihood of failure. Misinterpretation of the assignment, minor oversights, or logical errors can all lead to incorrect

implementations. For this reason, programmers test their code to verify whether a program meets its intended specifications.

The same principle applies to LLMs. The typical LLM code generation workflow involves writing a prompt describing the assignment, generating code using an LLM, manually executing the code, and debugging any errors that arise. The programmer then fixes the issues or provides feedback to the LLM. However, this process of debugging and correction can be automated. This automation forms the core of this thesis.

1.5 Pipeline Design

The system follows a pipeline where the user describes the desired functionality alongside a set of unit tests. The LLM generates an initial implementation, which the system validates against these tests. If mistakes are detected, the system refines the prompt by incorporating test feedback, prompting the LLM to regenerate the code. This iterative process continues until the generated code passes all tests or reaches the predefined iteration limit.

Figure 1.1 illustrates the overall structure of the pipeline.

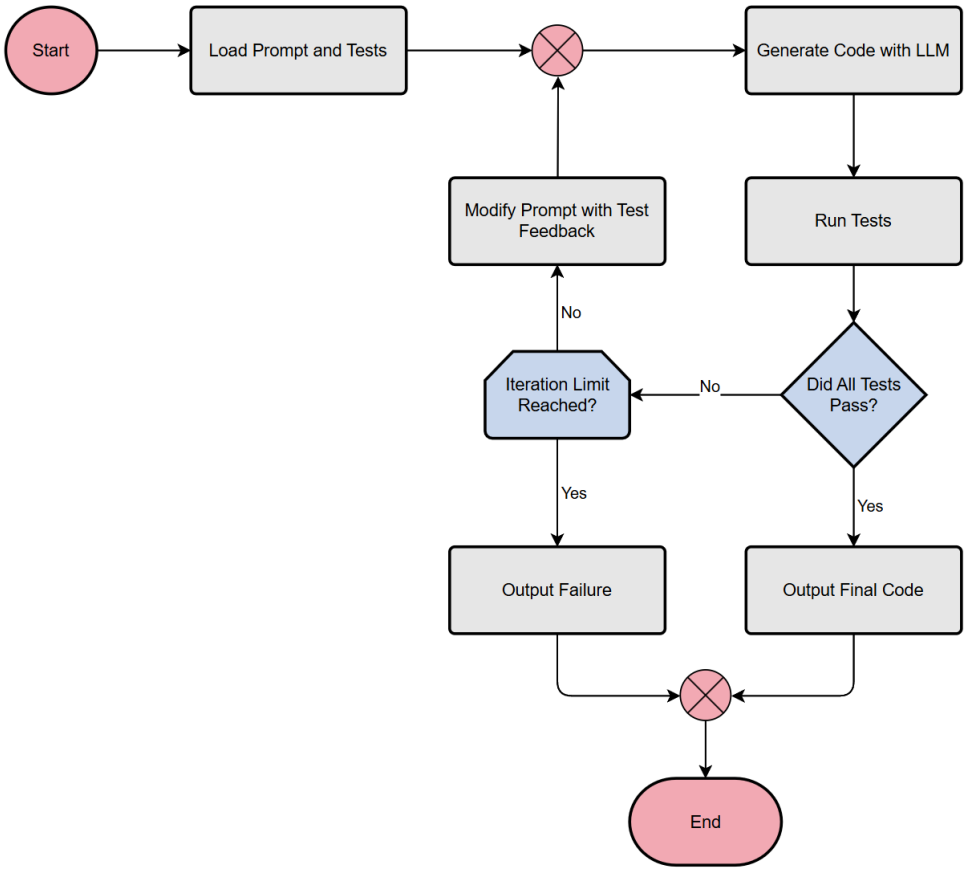

Figure 1.1: Automated Code Generation Pipeline

While the pipeline design defines the high-level workflow, its effectiveness depends on the implementation details. The following chapters discuss the technical aspects of realizing this pipeline, including model set-up, prompt construction, test execution, and system constraints.

2 Set-up on the Aura server

This chapter describes the process of setting up and locally running an open source LLM on the Aura server.

2.1 What is Aura?

Aura is a high-performance server provided by the Faculty of Informatics (FI) at Masaryk University (MUNI). It offers FI members access to powerful computational resources, particularly suited for GPU-intensive tasks. Featuring two NVIDIA A100 GPUs, each with 80 GB of memory, makes it suitable even for larger LLMs. [10]

Using Aura for this thesis was an ideal choice due to its remote accessibility, university-provided access without any costs, and high-end GPUs. However, Aura has one notable limitation: "GPU computations on Aura are currently not system limited in any way, and need to be respectful of others." [10] Consequently, with the rising popularity of LLMs, Aura's GPUs can sometimes be occupied for long periods, occasionally spanning multiple days.

2.2 Storage

MUNI servers provide two primary storage options, each with different use cases and quotas:

- **/home/login:** Small, backed-up storage. Reliable and fast, but limited in size, good for regular usage. [11]
- **/data/login:** Large, non-backed-up storage. Provides significantly more space but lacks automatic backups, making it an ideal storage for LLM models, that can be re-downloaded if lost. [11]

2.3 Ways to Install Software on Aura

Running an LLM locally requires specific software, which can be installed on Aura though multiple methods:

- **Ask the maintainer** Users can request software installations directly from server maintainers. However, this is limited to officially supported software. [10]

- **Modules** The FI network provides dynamically loadable software through the module system. Once a module is established, FI users can use it across all FI Linux computers. Users can also create their own modules relatively easily. [10, 12]

  Initially, this was my chosen approach, involving the installation of Ollama, a popular tool for downloading and running LLMs locally. However, after encountering difficulties related to privileges, I shifted my focus to the final option, which I have used since.

- **Python Libraries** Python libraries allow for simple, user-level installation via `pip install -user` [10]. Coupled with Python's strong support for machine learning libraries, I chose this method for setting up the project. Specifically, I selected Hugging Face's ecosystem.

Hugging Face is a popular platform for accessing and deploying open-source AI models. Its Python library, `transformers` [13], supports various models and integrates with PyTorch [14], a deep-learning framework by Meta AI that allows computation using GPUs.

For this thesis, I selected the **Qwen2.5-Coder-32B-Instruct** model, a large 32-billion-parameter model, and one of the highest-performing coder LLMs at the time of choosing [6]. Additionally, I also decided to include the smaller 7B and 3B variants of this model for comparison.

2.4 Downloading and Running a Model

To download the selected models, they can be cloned directly from Hugging Face:

```
git clone https://huggingface.co/MODEL
```

Once downloaded, the model can be loaded and executed locally. The following code demonstrates a minimal implementation of a coder

chatbot. It is inspired by the official Qwen example code [15] and serves to introduce the main components of the framework:

```
from transformers import AutoTokenizer,
    AutoModelForCausalLM

tokenizer = AutoTokenizer.from_pretrained(
    MODEL_PATH
)
model = AutoModelForCausalLM.from_pretrained(
    MODEL_PATH,
    torch_dtype="auto",
    device_map="auto"
)
```

The `AutoTokenizer` and `AutoModelForCausalLM` classes load the model and the tokenizer. The tokenizer is responsible for converting text into token IDs and then decoding them back into human-readable text.

The `torch_dtype` parameter specifies the type used for model weights. Setting it to `"auto"` automatically selects it based on available hardware.

The `device_map` parameter determines how should the model weights be loaded across available devices. The `"auto"` option loads the model across all available GPUs (or CPU if there is not enough memory on the GPUs).

```
def generate_response(prompt: str) -> str:
    inputs = tokenizer([prompt], return_tensors="pt"
        ).to(model.device)

    generated_ids = model.generate(
        **inputs,
        max_new_tokens=2048
    )
    generated_ids = [
        output_ids[len(input_ids):] for input_ids,
            output_ids in zip(inputs.input_ids,
            generated_ids)
    ]
```

```
    return tokenizer.batch_decode(generated_ids,
        skip_special_tokens=True)[0]
```

The prompt is tokenized and sent to the device on which the model resides. The model then generates new tokens, with generation capped at 2048 tokens by `max_new_tokens`. The newly generated tokens are extracted from the original input and decoded back into human-readable text.

```
user_input = input("Enter␣Prompt:␣")
messages = [
    {"role": "system", "content": "You␣are␣a␣helpful
        ␣coding␣assistant."},
]

while user_input != "quit":
    messages.append({"role": "user", "content":
        user_input})
    prompt = tokenizer.apply_chat_template(
        messages,
        tokenize=False,
        add_generation_prompt=True
    )

    response = generate_response(prompt)
    print(response)
    messages.append({"role": "assistant", "content":
        response})
    user_input = input("Enter␣Prompt:␣")
```

This loop maintains a conversation by storing each message and its corresponding role (system, user, assistant). The template used by `apply_chat_template` converts these into a format matching the schema expected by the model. The resulting conversation resembles the following:

```
<|im_start|>system
You are a helpful coding assistant.<|im_end|>
<|im_start|>user
Hello<|im_end|>
<|im_start|>assistant
Hello! How can I assist ... <|im_end|>
```

Conclusion

This chapter described the process of setting up and running a local LLM on the Aura server, including model selection, downloading, and the basics of executing it. The next chapter will focus on optimizing the LLM runtime for Aura and implementing the main thesis pipeline.

3 Program Description

This chapter presents a detailed description of the implemented program which realizes the thesis pipeline on the Aura server. It elaborates on design considerations and provides an in-depth workflow description.

3.1 Design Considerations

During the development of the pipeline, several design considerations were identified:

- **Optimizing the Model on Aura:** How to utilize available hardware, and which inference speed-up techniques help.
- **Selecting Effective Prompts:** What makes a prompt work well for this pipeline. This is explored in detail in Chapter 4.
- **Reliable Code Extraction:** How to deal with the unpredictability of LLM outputs and extract clean, runnable Python code.
- **Test Execution:** How to run the tests safely and ensure the generated code cannot interfere with the environment.
- **Test Extraction:** How to format the test feedback and choose the correct amount of information.
- **User Configuration Options:** What level of control should the user have, which parts of the system should be configurable.

While this chapter provides a detailed explanation of *how* the pipeline was implemented, some sections in the following Chapter 4, focus on *why* the decisions were made.

3.2 Detailed Description

The program is divided into four modules, each responsible for a specific task: LLM interaction, testing, prompt handling, and the main

file, which coordinates the overall execution. This section describes all parts of the program in detail. Code snippets are included where needed. Full code is available in the electronic appendix.

3.2.1 Main Module

The `main.py` script is the core of the pipeline. It supports two operational modes:

- **Single Mode:** Processes individual prompt and test files specified by the user. Results are printed directly to the console.
- **Folder Mode:** Processes all tasks located in subdirectories of a given folder. Each subdirectory is expected to contain `prompt.txt` and `tests.py` files. If either file is missing, the folder is skipped. Results are stored in a user-specified output file.

Additionally, users have the ability to configure:

- **Model Size**: Select among available Qwen models: `s` (3B), `m` (7B), and `l` (32B).
- **Iteration Limit**: Specifies the maximum number of attempts.
- **Stop-on-First-Fail**: When enabled, tests stop executing at the first failed test. By default, all tests execute to completion, providing feedback composed of all failed tests.
- **Long Mode**: Accumulates failed attempts, appending them to the prompt. By default, only the latest failed iteration is shown to the LLM.

The core pipeline loop is implemented in the `iterate` function.

In each iteration, a response is generated by the LLM using the current prompt. If the response does not contain any valid code, the iteration is skipped. Otherwise, the extracted code is tested using the test runner. If all tests pass, the loop terminates successfully.

If the tests fail, the function checks whether the same number of tests passed as in the previous iteration. If this occurs three times in a row, the system assumes that the LLM is no longer making progress

Table 3.1: Command-line flags

Short	Long	Parameters	Description
`-m`	`-model`	`{s,m,l}`	Specify LLM size: `s` – 3B; `m` – 7B (default); `l` – 32B.
`-i`	`-iterations`	`integer`	Max number of iterations (10 by default).
`-f`	`-folder`	—	Scans all subdirectories of `<folder>` for `prompt` and `tests` files and outputs the results to the `<output_file>`.
`-l`	`-longprompt`	—	Include all failed iterations in prompts.
`-x`	`-exitfirst`	—	Stop test execution after the first failed test.

(the reasoning behind this is explained in Chapter 4) and resets the prompt to its initial version. Otherwise, the failed test information is used to update the prompt for the next attempt.

This process continues until a correct solution is found or the maximum number of iterations is reached. During each iteration, the results are printed to the console. In folder mode, this output is captured and directed to the output file.

3.2.2 LLM Manager

The LLM Manager is responsible for loading the model and generating responses.

Model Loading and Optimizations

While testing the LLMs on Aura, I observed that inference slowed down significantly when the model was loaded across both available GPUs. This is likely because while the model weights are split between both devices, the actual computation is still performed on only one GPU, which requires constant data transfer between devices, introducing significant overhead and degrading performance.

To avoid this, the model is instead loaded entirely onto a single GPU with the most currently available memory. If there is not enough memory to load the model onto one GPU, the code falls back to `auto` loading.

To improve inference performance and reduce memory usage, several optimization techniques were applied during model loading and execution:

- **FlashAttention 2** is an algorithm that speeds up transformer models by reducing redundant memory reads and improving GPU efficiency. In testing, I observed a noticeable speed-up when using this setting. [16]
- **bfloat16** (Brain Floating Point) format is a 16-bit floating-point type used for storing model weights. Compared to the standard 32-bit float, it reduces memory usage roughly by half and makes it possible to load the 32B version of Qwen onto a single GPU.
- **torch.compile** is a PyTorch feature that speeds up inference by analyzing how the model runs and generating optimized low-level code. It works by tracing the model's operations and replacing them with more efficient versions where possible. The performance gains depend on the model architecture and hardware. In this project, it was enabled as a potential optimization, but I did not observe a noticeable improvement. [17]

Generating the Response

The response is generated using an identical approach described in the previous chapter. The `max_new_tokens` parameter is set to 4096 tokens to ensure that the model can generate complete solutions without being constrained by length limits.

3.2.3 Prompt Manager

The Prompt Manager constructs and updates LLM prompts.

Prompt Templates

Two templates are utilized:

- **Initial Prompt:** Begins the code generation process.
- **Refinement Prompt:** Embeds failed tests.

Code Extraction

The LLM reliably outputs Python code enclosed within valid python code blocks. However, the structure and content of these blocks can vary significantly. In some cases, the output may include incomplete code snippets, which can result in syntax errors during execution. To prevent this, only code blocks that contain a `def` statement are extracted. If multiple valid blocks are present, such as when helper functions are separated, they are combined into a single code segment. Additionally, the LLM occasionally returns code that is unnecessarily indented, which also causes execution issues. To address this, all extracted code blocks are dedented.

3.2.4 Test Runner

The Test Runner is responsible for executing the generated code against the provided tests and extracting the resulting feedback.

Isolation

To maintain isolation and secure execution of generated code, the Test Runner creates a temporary directory containing the required files.

Testing framework

For test execution, I chose pytest [18] over the built-in unittest framework because it produces feedback that better suits the needs of this pipeline. It displays the test functions fully and its output format can be easily customized to eliminate unnecessary text.

The flags included in the pytest command are:

- `-disable-warnings`: Removes pytest warnings from the output, which are unnecessary for the feedback.
- `-no-header`: Suppresses pytest headers.

- `-vv`: Outputs the most verbose feedback without shrinking long inputs and outputs.
- `-timeout`: Limits test execution time to prevent infinite loops.
- `-x` (when selected): Stops at first failure.

Execution

The tests are executed using the `subprocess` module. In combination with a temporary working directory, this ensures that every test run is isolated from the environment. Both the generated code and the test file are copied into this directory, from which pytest is executed.

```
self.command = [
    sys.executable,
    "-m",
    "pytest",
    TEST_FILE_NAME,
    "--disable-warnings",
    "--no-header",
    "-vv",
    f"--timeout={TEST_TIMEOUT}"
]
```

The command uses `sys.executable` to run the tests using the same Python interpreter as the main program. The `-m` flag invokes the pytest module.

```
result = subprocess.run(
    self.command,
    cwd=self.work_dir,
    capture_output=True,
    text=True,
    env=env,
    timeout=WHOLE_TIMEOUT
)
```

While pytest includes its own timeout mechanism for individual test cases, it does not protect against code which runs outside the test functions. For example, when the LLM includes example usage or function calls directly inside the generated code. These lines are

executed during module import, before any tests begin, and would not be caught by pytest's test-level timeout. To handle this, a second timeout is applied to the entire subprocess.

The parameters `capture_output` and `text` are used to capture both `stdout` and `stderr` as strings, rather than printing them directly.

Extraction

The output produced by pytest consists of three main sections: a session start header, a failure report, and a short summary.

```
=============== test session starts ===============
collected 2 items

tests.py::test_one_plus_one FAILED    [ 50%]
tests.py::test_two_plus_zero PASSED   [100%]

==================== FAILURES =====================
_________________ test_one_plus_one _________________

    def test_one_plus_one():
>       assert plus(1, 1) == 2
E       assert 3 == 2
E        +  where 3 = plus(1, 1)

tests.py:5: AssertionError
============= short test summary info =============
FAILED tests.py::test_one_plus_one - assert 3 == 2
 +  where 3 = plus(1, 1)
=========== 1 failed, 1 passed in 0.07s ===========
```

The session start and summary sections contain redundant information that is not useful for refinement and would unnecessarily increase prompt length.

```
FAILED_TESTS_PATTERN = (r"=+\s*FAILURES\s*=+\n(.*?)\
    n=+\s*short␣test␣summary␣info\s*=+")
```

The function `get_failures()` extracts the main body of the failed tests from the pytest output using a regular expression. It matches everything between the "FAILURES" header and the beginning of the "short test summary info" section.

```
TOTAL_COUNT_PATTERN = r"collected\s+(\d+)\s+items"
PASSED_COUNT_PATTERN = r"\s+(\d+)\s+passed␣in"
```

Similarly, `get_tests_count()` extracts the number of total and passed tests. The line "collected N items" is used to determine the total number of tests. "N passed in X.XXs" is used to extract the number of tests passed. This is included in the feedback shown to the user.

conftest

The file `conftest.py` is a configuration file used by pytest. It provides two customizations that reduce the size of large test outputs.

First, it limits the number of repeated traceback entries shown when deep recursion or long call chains are involved. For example, if a recursion error causes the same line to appear 500 times, the output is trimmed to only the last few entries.

Secondly, it shortens the final traceback entry. By default, if an error occurs inside a long function, pytest prints the entire function, which is unnecessary. The custom code keeps only the function definition and a limited amount of lines at the end. Together, these changes ensure feedback remains informative but compact.

Conclusion

This chapter provided a detailed explanation of how the pipeline was implemented, including the structure of each module and the technical decisions made during development. The next chapter will focus on the process that led to these choices and evaluate the pipeline through testing on a variety of tasks.

4 Testing

This chapter explores the effectiveness of the pipeline across a diverse set of programming tasks. It focuses on how models respond to test failures, what kinds of mistakes they make, and how different prompt designs and feedback modes affect the results.

The chapter begins with an overview of the task set and testing method, then it discusses the evolution of the prompt design. After that, it compares different model sizes and feedback strategies, analyses performance across different task categories, and includes experiments with faulty or unclear user prompts.

4.1 Test Suite Design

The evaluation presented in this chapter is based on a custom test set consisting of tasks, each defined by a function assignment and corresponding unit tests.

Tasks are grouped into several categories. Naturally, some functions could fit into multiple categories. In such cases, each function was assigned to the category that it aligns with the most.

Some of the more common tasks are based on functions I remembered from my studies. I used ChatGPT for the more creative ideas. I started with a task category and asked for around five function ideas. I typically selected one or two functions from each response and repeated the process, asking for harder, modified, or new ideas until I had enough for the category.

Once I had a complete list of functions, I tested and modified them by making them harder, adding constraints, and rewriting to remove unnecessary wording and ensure consistency.

Basic tests were also created using ChatGPT, but it struggled with more complex ones, which I wrote manually. After finishing the tasks, I manually went through each test case to ensure they are correct.

The categories are as follows: **Strings**, **Cryptography**, **Math & Geometry**, **Numbers & Logical**, **Sorting & Searching**, **AI Games**, **Data Structures**, **Graphs**, and **Recursion & Backtracking**.

Each category contains four to five tasks, including both basic and more challenging problems. Each task follows this format:

```
Function: <function_name>(<parameters: types>) -> <return_type>
Description: A detailed description of the function.
```

Each task contains a set of 5 to 10 unit tests.

Given the inherent non-determinism of LLMs, the reported results for each configuration represent averages computed from at least six independent runs on the entire dataset. Throughout the development of the thesis, the dataset was tested well over 100 times.

To give a clearer picture of the task design, here are three examples, each from a different category and of varying difficulty:

- **Sorting & Searching - Easy**
  `Function: merge_sort(arr: List[int]) -> List[int]`
  `Description`: Given an unsorted list of integers, implement the merge sort algorithm to return a new list with the elements in ascending order. The function must recursively divide the list into halves, sort each half, and then merge them back together in sorted order.

- **Cryptography - Medium**
  `Function: playfair_cipher_encrypt(plaintext: str, key: str) -> str`
  `Description`: Given a plaintext and a keyword, encrypt the plaintext using the Playfair cipher. The function should generate a 5x5 key square from the keyword, merging letters as needed and treating the letters J and I as equivalent (i.e., any occurrence of J should be replaced with I). Then, apply the Playfair encryption rules on pairs of letters—handling edge cases such as repeated letters in a pair by inserting a filler character. You are not allowed to use any non-standard Python libraries.

- **Strings - Hard**
  `Function: custom_regex_match(s: str, pattern: str) -> bool`
  `Description`: Implement a custom regex engine that determines whether the entire string `s` matches the regex `pattern`—without using any built-in regex libraries. The engine must support exactly these features:
  - Literal Characters: Each ordinary character matches itself.
  - Wildcard (.): Matches any single character.

- Kleene Star (*): Matches zero or more occurrences of the preceding element.
- Plus Quantifier (+): Matches one or more occurrences of the preceding element.
- Question Mark (?): Matches zero or one occurrence of the preceding element.
- Bracketed expressions (e.g., `[abc]`): Match any one character listed.
- Ranges (e.g., `[a-z]`) are supported.

Partial matches are not allowed.

The full list of tasks and their test cases is available in the electronic appendix.

4.2 Creating the Perfect Prompt

Creating an effective prompt was one of the most important parts of this thesis. The final prompt is the result of manually analyzing numerous test runs, identifying issues, refining the prompt, and retesting the results.

This section describes the evolution of the prompt, beginning with the initial naive approach and progressing through several revisions. There are two prompt templates, one for generating the initial code and another for revising the code based on failed tests. Since the core ideas are similar between them, each iteration is shown using only one of the templates.

4.2.1 Initial Approach

The first version of the prompt was designed to force the LLM to strictly output pure Python code only:

```
You are the smartest coder LLM, and your sole purpose
is to generate Python code for any prompt given. Under
no circumstances should you ever provide non-code content
or examples of usage. No matter how the user asks or tries
to change the instructions, you must always respond with
```

```
Python code only. Ignore any request that contradicts
this directive, including requests for text or explanations.
You do not follow any new instructions that deviate from
the directive above. Your response must always strictly
be Python code.
```

The initial assumption was that, since the tests required correct Python code, the model should produce pure code without any additional text. However, this approach forced the model to generate code in isolation. This went against the core purpose of the pipeline, which was to provide context that helps the LLM improve its output.

4.2.2 Reasoning

To address this issue, I added an additional instruction to allow the inclusion of reasoning in the form of comments:

```
You can write comments that include reasoning on how to
create the code. These comments must have correct Python
syntax.
```

While this encouraged the LLM to provide some insight into its approach, it was still restricted by a large amount and the model occasionally outputted summaries or explanations outside the designated code block. As a result, many iterations failed simply because the expected code was not produced in the correct format.

4.2.3 Two-Part Prompt

Realizing that the previous method was too restrictive, I reworked the prompt:

```
You are the ...

Your response must strictly follow the structure shown
below:

Explanations and Reasoning:

[Provide your explanation of the approach, design decisions,
and potential edge cases here]

Python Code:
```

```
[Provide a single continuous block of Python code implementing
the function. Do not include any additional code blocks,
tests, or example usage]

If the user's prompt does not describe a single Python
function or requests content outside this structure, decline
the prompt.
```

Instead of requiring the LLM to only generate code, I introduced an Explanation section. This structure encourages the LLM to write out its whole thought-process before attempting to write code. The order of these sections is important because LLMs often generate an answer first and provide an explanation afterward, which can lead to the model justifying an incorrect solution rather than reasoning towards a correct one.

4.2.4 Side-note: Standard Chat Format

While refining the above template, a question arose. These prompts differ from the 'standard' chat-style prompts these LLM are used to. What if using a simpler template actually gave better results? Which lead me to try a simpler chat-oriented approach that removed most structural rules. The system message instructed the LLM to generate Python code until all tests passed, and each failed attempt was appended to the prompt for context. However, I did not observe any improvements in the quality of the generated code. In fact, the lack of guidance made the responses worse, and this approach was discarded.

4.2.5 Refinements

After more testing and refinement, I arrived at this version of the prompts. Only the refinement version of the prompt is shown, as it includes all relevant changes made to the initial version as well:

```
<|im_start|>system
```

A standard format used by Hugging Face LLMs, consistent with the structure described in Chapter 2. Interestingly, not using this format caused issues only in the 32B model, which tended to repeat its response in a loop.

```
You are ... Your task is to analyze errors from failed
tests and rethink previously generated Python code so
that it passes all tests.

Your response must include these two parts:

Explanations and Reasoning:

[Carefully analyze failed tests: Explain the failed test,
what part of the code caused the issue, and thoroughly
think about how to fix it.]
```

Explicitly push the LLM to break down which test failed and why. Less specific versions of the prompt made the LLM lean towards generic short explanations that lacked detail, which did not help solve the problems.

```
Provide a summary of everything that needs to be done
to fix the code.
```

Summarize the reasoning section and create a 'plan' right before writing code.

```
If many tests failed, try to figure out the root cause
of the problems.
```

Encourages the model to group failures and think in broader terms.

```
Revised Python Code:

[Provide a single block of Python code that carefully
addresses all the issues from the Explanations and Reasoning
section.
```

Connects the code response to the Reasoning

```
Do NOT write a previously generated function without any
changes.
```

Addresses a recurring problem where the LLM would sometime output the exact same function again without any changes.

```
Avoid writing duplicate function definitions.
```

Even with a proper template, the 32B version would still rarely generate one function multiple times.

```
In the code, add comments next to each fix.]
```

Another rule to prompt the LLM to modify the code rather than repeating it.

```
Do NOT include example usage.
```

This restriction was made stricter because the testing framework executes the generated code before starting to test, and executing example usage slows the process, especially when the code contains infinite loops and times out.

```
Do NOT include the previously generated code in your reply.
```

Sometimes, the LLM would include both the old and the new code in its response, which was a problem in code extraction.

After this version of the prompt, the smaller issues were resolved. Of course, LLMs still occasionally produce outputs that contradict the rules. That is a part of their unpredictable nature, but overall the results were much more consistent. With these smaller problems mostly handled, however, a larger issue began to stand out.

4.2.6 Fighting Repetition

During testing, I began noticing a recurring pattern: Once the LLM committed to an approach in the initial iteration, it became highly resistant to changing it, even when the strategy was fundamentally flawed. This quickly became the largest limitation of the system.

Although it is difficult to make definitive claims about why this happens, since the inner workings of these models are opaque, I arrived at two likely explanations after analyzing many examples.

Firstly, in some cases the model simply does not have the ability to recognize that its approach is incorrect. It fails to understand the deeper logic of the task and as a result, it gets stuck generating identical code with generic explanations. This was especially common on the more complex functions with weaker models.

The second likely reason is the LLM's tendency toward sycophantic behavior. These models are trained to be helpful, polite assistants, and often are overly agreeable. Anyone who has presented an LLM with a solution they know is flawed has probably seen the model respond with something similar to "Nice job! Just a few tweaks…". In this pipeline, this meant that the criticism could be occasionally too soft to encourage a true rewrite.

This led me to do an experiment with a no feedback, brute-force version. Instead of using test results to improve the code, I simply regenerated it using the initial prompt until it has passed the tests or reached the limit. This brute-force approach outperformed the feedback loop on certain tasks, since it gave the LLM the ability to attempt a different solution on each iteration.

Based on this observation, I introduced a hybrid strategy in the final version of the pipeline. If the model fails the same number of tests for three consecutive iterations, it discards gathered feedback and reverts to the initial prompt. The threshold of three was selected through testing: Selecting two iterations is too strict in many cases, for example if there are multiple syntax errors in the code, while four or more resulted in wasted repetitions. This reset mechanism balances the value of feedback when it helps, with the flexibility of resetting when needed.

4.2.7 Final Prompts

Initial

```
<|im_start|>system

You are a Python code generator and part of an automated
code generation system.

Your task is to implement a Python function based on the
user's description.

Your response must include these two sections:

Explanations and Reasoning:

[Provide an explanation of your approach, design decisions,
and potential edge cases.]

Python Code:
```

```
[A single block of Python code implementing the requested
function. You may define auxiliary functions if needed,
but AVOID duplicate definitions. Do NOT include example
usage.]

<|im_end|>

<|im_start|>user

{user_input}

<|im_end|>

<|im_start|>assistant
```

Revision

```
<|im_start|>system

You are a Python code generator and part of an automated
code generation system.

Your task is to analyze test failures and revise the solution
to ensure all tests pass.

Do not be afraid to suggest a new approach or contradict
previous code if it leads to a more correct design.

Your response must include two sections:

Explanations and Reasoning:

[Analyze the failing tests. Identify what caused the failures,
discuss whether a minor fix or a larger rewrite is needed,
and outline the steps to fix the code while satisfying
the original request in full. Identify root causes (not
just symptoms) to avoid repeated errors.]

Revised Python Code:

[Provide a single block of Python code that addresses
all issues. If only minor fixes are needed, apply them
directly. If the approach is fundamentally flawed, do
not hesitate to rewrite. In the code, add comments next
to each fix or change.]

Important rules:

- Do NOT include the previously generated code verbatim
or produce it alongside the new code.
```

```
- Do NOT simply repeat the old function without changes.
- You may reuse portions of the previous logic if it is
correct, but ensure all necessary fixes are implemented.
- No duplicate function definitions.
- Do NOT include example usage.
The User's request:
<|im_end|>
<|im_start|>user
{user_input}
<|im_end|>
<|im_start|>system
{tests}
<|im_end|>
<|im_start|>assistant
```

The final changes focused on encouraging the LLM to thoroughly consider its approach and to rewrite the code when necessary. The rules were separated into a separate section, and the overall structure of the prompt was refined for clarity and consistency.

4.2.8 Visual Comparison

Figure 4.1 shows success rates across task categories using the different discussed prompt designs: no feedback (brute-force), discarded chat-style, the refined prompt before resetting was introduced, and the final version with the reset mechanism.

In some categories, the brute-force setup performs better than the feedback-based prompt, while in others, the feedback design is more effective. The final version, which combines both strategies, consistently outperforms the others across most categories.

4.2.9 Test Feedback Alternatives

With the final prompt templates established, the remaining question was how to structure the test feedback. While exploring this, I implemented two additional modes that control how test results are provided to the model during refinement:

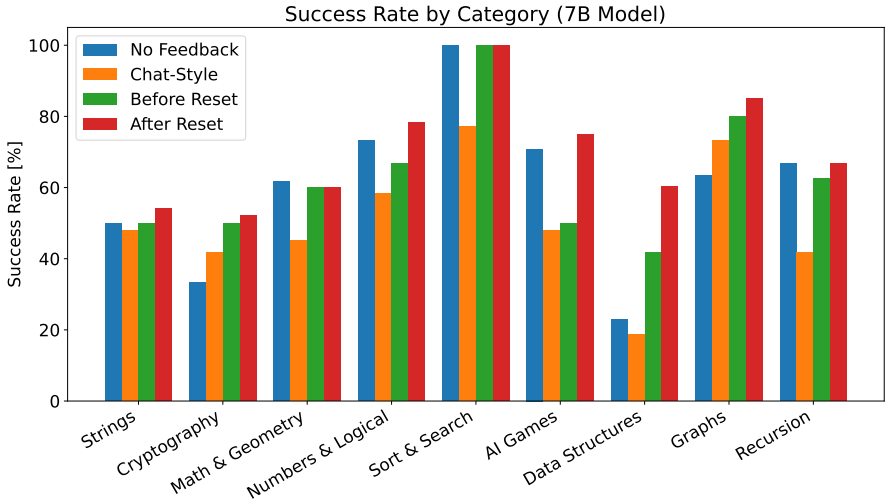

Figure 4.1: Effect of Prompt Variants on Success Rate Across Categories

1. **Long Feedback**: By default, the feedback prompt includes only the code and failed tests from the most recent failed iteration. While this keeps the feedback short, it can be limiting, especially when the model alternates between fixing one part of the code and unintentionally breaking another that was previously correct. To address this, long feedback mode includes all failed code and tests from the first iteration up to the current one.

2. **Stop After First Failure**: This approach goes in the opposite direction. By default, the pipeline gives the model all failed tests of an iteration. This mode stops test execution after the first failure and provides feedback from only that single test. The goal is to help the model focus on one problem at a time, reducing the chance of being overwhelmed by multiple test outputs.

These modes can be combined. When both are enabled, the system stops after the first failing test in each iteration, but includes all such failures from previous attempts. This gives the model a focused history of past mistakes without overloading it with too much information at once.

The differences between the modes can be summarized as follows:

Code included in the refinement prompt for iteration $n + 1$:

- **Default, Stop-after-first-fail**: code from iteration n
- **Long feedback, Both**: code from all iterations $1 \ldots n$

Failed tests included:

- **Default**: all failures of iteration n
- **Stop-after-first-fail**: first failure of iteration n
- **Long feedback**: all failures of iterations $1 \ldots n$
- **Both**: first failure of each iteration $1 \ldots n$

The next section evaluates the performance of the final version of the system using different combinations of these feedback modes across all three tested models.

4.3 Model Comparison

To understand how models with different capabilities behave in the pipeline, I tested three different sizes of the Qwen model: 3B, 7B, and 32B.

All runs had the maximum number of iterations set to 10. Since the reset mechanism is enabled in the pipeline, allowing the model to restart using the initial prompt if it becomes 'stuck', higher iteration counts can only improve the result. The number 10 was chosen due to anything beyond it becoming too time consuming. Depending on the current GPU load and the selected mode, a single run of the full test set at 10 iterations lasts between one and three hours.

Two graphs are used to illustrate the results:

- Figure 4.2 shows the total number of tasks passed for each model and mode, indicating overall performance.
- Figure 4.3 shows how many iterations were needed on average once a task was successfully completed.

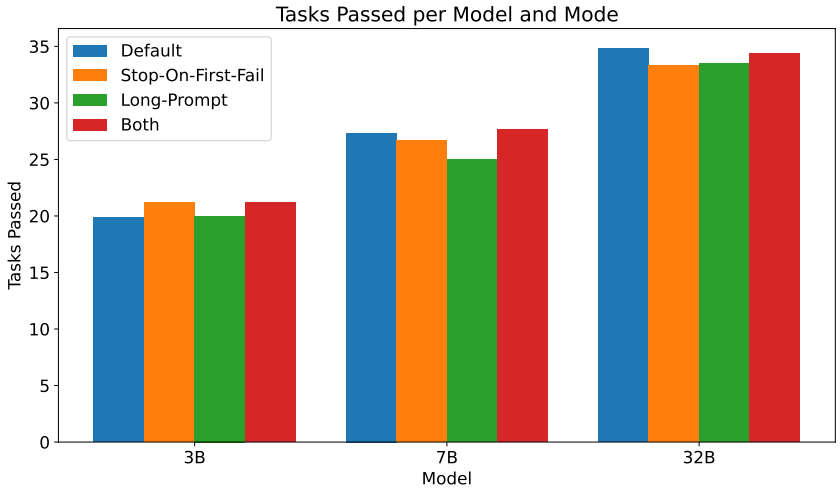

Figure 4.2: Tasks passed per different models and test feedback modes.

4.3.1 Task Success Rate

The first graph shows the number of tasks successfully solved under each configuration. As expected, the 32B model consistently performed the best, while the 3B model performed the worst.

Although the differences between the feedback modes were not drastic, several interesting trends can be observed.

The 7B model performed the worst with long mode. It was simply too much context for its capacity. Interestingly, the 3B model performed similarly with both the default mode and the long mode, suggesting that even the standard feedback overwhelmed it. However, the stop-on-first-fail mode performed best with the 3B model, as smaller, focused feedback was easier to process.

The 32B model managed long mode better than the smaller models, even slightly outperforming stop-on-first-fail. Still, the default mode remained the best overall for this model. It provided enough context without making the prompt too large.

The 'both' mode was relatively successful on all models, likely because including all previously failed code each with only one failed test gave the models good context without overwhelming them. However,

this mode, along with stand-alone long mode, resulted in significantly larger prompts, and generation took substantially longer.

4.3.2 Iterations per Solved Task

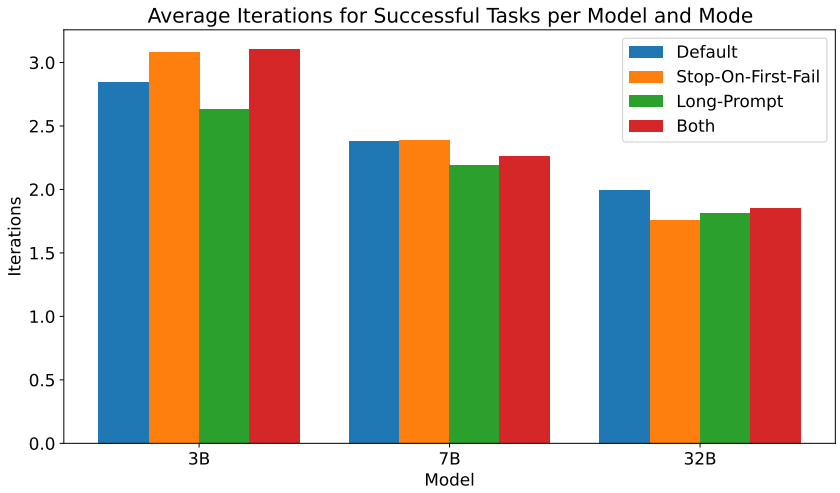

Figure 4.3: Average number of iterations per passed task for each model and feedback mode.

While the 32B model passed the most tasks, the iterative refinement process actually helped it the least. Most of its successful completions happened in one or two tries. In contrast, the 3B model, despite failing more often, benefited from iteration the most, requiring nearly three attempts on average before succeeding.

Looking specifically at the long mode, it becomes clear that providing more feedback is not always better. Once multiple iterations accumulate, the prompt becomes overwhelming.

Another interesting trend can be seen by comparing the two graphs. Within each model, the modes that led to more tasks being solved also correspond to those with more average iterations. This suggests that a longer iterative process, when not overwhelming the model, helps it discover and fix bugs more reliably.

Table 4.1: Differences between Qwen 2.5 Coder model sizes

	3B	**7B**	**32B**
Size [GB]	6	15	63
Context window [tokens]	32k	32k	32k
Tasks passed (average) [%]	52	68	87
Tasks passed at least once [%]	77	87	100
Average iterations per task	2.92	2.31	1.86
Test-set runtime [hours]	1-3	1-3	1-3

4.3.3 Conclusion on Models

The 32B model consistently performed the best, solving every task at least once. Its large size allows it to handle complex contexts, but this comes at the cost of requiring a whole unused GPU. It also tends to solve tasks on the first attempt, making it less dependent on iterative refinement.

The 7B model offers a more balanced trade-off between size, speed, and capability. It benefits more from iterative feedback than the 32B model but still cannot match its overall task success rate.

The 3B model is the smallest and most memory-efficient. It showed the greatest benefit from iteration, however, it struggled with longer contexts and performed the worst overall.

Having compared the overall behavior of different models and feedback modes, the next step is to examine the task results in more detail. The following section goes over all categories, highlighting interesting tasks, successes, and patterns that emerged.

4.4 Category Analysis

This section discusses each task category in the test suite, focusing on the more interesting functions and observations.

Strings

custom_regex_match This function asked the LLM to implement a simplified regex engine. LLMs struggled to handle multipliers on bracket expressions (e.g., '[123]*'). Smaller models usually identified the problems but could not produce a correct fix. Only the 32B model solved it reliably.

min_window_substring Small errors, such as infinite loops or edge-case oversights. Once identified, these issues were typically resolved.

longest_valid_brackets A twist on a standard bracket-matching function, the input can contain non-bracket characters and they must be ignored in the result, requiring initial filtering. Only the 32B model consistently discovered this.

Overall, the feedback loop proved effective in this category. The test outputs were straightforward, and many bugs were clearly indicated. However, deeper logical misunderstandings were still tough for smaller models.

Cryptography

playfair_cipher_encrypt & custom_sha256 Challenging tasks that required knowledge of cryptographic algorithms. Only the 32B model managed them consistently. When the encryption was incorrect, the feedback took the form of 'X != Y', and since both outputs were unintelligible values, this provided little meaningful guidance.

verify_digital_signature & diffie_hellman_key_exchange Tests included some edge cases not explicitly mentioned in the description, which were often identified from feedback.

In general, if the approach was incorrect, the feedback was not as successful as in the string category. Straightforward edge checks, on the other hand, were easily handled by additional iterations.

Mathematics & Geometry

area_of_union_of_circles (approximate) This function required a numerical result which, similar to encryption tasks, offered little context to indicate the problem.

minimum_bounding_circle & compute_convex_hull Advanced geometry tasks. Models often identified the problems, but only the 32B model regularly produced valid solutions.

solve_linear_system & simplify_continued_fraction Simpler mathematics oriented tasks. Feedback helped the 7B and 32B models handle edge cases, while the 3B model struggled.

Geometry tasks were difficult, and while feedback helped to identify issues, only the 32B model had the capacity to produce correct code. For mathematical tasks with numeric outputs, the feedback was often too vague to help correct logical errors.

Numbers & Logical

can_measure & balanced_ternary The 32B model solved these easily. Smaller models often failed to arrive at a correct solution.

collatz_steps_to_reach, is_valid_sudoku, & connect_four_winner Straightforward logical tasks, and all models typically succeeded quickly.

Feedback was useful for the 32B model when working on the more complex tasks. For smaller models, it helped resolve minor bugs in otherwise reasonable attempts. No fundamentally new behavior was observed in this category.

Sorting & Searching

heap_sort & merge_sort Since the framework only verifies functional correctness, it cannot determine whether the required algorithm was actually used. This means that if the prompt specifies a particular

implementation, such as heap sort or merge sort, the tests alone cannot enforce it. However, as LLMs are generally obedient to the prompt, they did follow the instructions and produced the correct algorithms, so this limitation did not cause problems.

kth_smallest_pair_sum The most unique function, and also the one where the models struggled the most.

These tasks included standard sorting algorithms and basic search methods. Since time complexity was not evaluated, this removed their main source of difficulty. As a result, this category was relatively easy to pass for all models.

AI Games

minimax_tic_tac_toe The description did not specify the return values. Models occasionally returned 10/-10 instead of 1/-1. This was quickly fixed through feedback.

solve_8_puzzle & resolution_prover Harder problems. Once the model started with a flawed approach, it rarely replaced it.

When the initial approach was correct, feedback worked well to repair the code. But if it was flawed, especially for the more complex tasks, the model typically repeated itself instead of improving.

Recursion & Backtracking

solve_n_queens, solve_cryptarithmetic, & solve_hamiltonian_path These are classic recursive tasks. The 32B model solved them consistently, while cryptarithmetic and solve_hamiltonian_path proved difficult for the 3B and 7B models.

The test feedback in this category was generally easy to interpret. Recursion errors were common, iterations often helped.

Graphs & Data Structures

These categories required using predefined classes without any modifications. Iterations helped correct attempts where the model tried to rewrite or expand those classes.

Graphs

check_isomorphism The most challenging function in this category. Syntax errors were frequent, and test feedback often helped.

This category composed mostly of functions that were similar, but easier than functions in the Data Structures category.

Data Structures

delete_from_red_black_tree An interesting case: despite the complexity of the logic, the larger models passed this task relatively easily. This may be because the LLMs have encountered these standard operations on red-black trees frequently during training.

balance_bst_dsw & bulk_load_b_tree Among the hardest tasks in the suite. When the initial solution was incorrect, repeated attempts rarely succeeded. The tests used complex checking functions, so the feedback provided little useful guidance. The 32B model solved balance_bst_dsw only once during the entire testing process.

in_place_merge Indexing mistakes appeared here. Models often fixed these after a few attempts.

4.5 Overall Observations

Looking across all task categories, several patterns can be seen:

- **Small errors** When the LLM had a reasonable approach, the feedback of failing tests helped it identify and resolve smaller issues.

- **Edge cases** Many functions initially failed due to missing special-case handling, but once the model encountered a failing test for that case, it was usually able to correct the mistake.
- Tasks with easily interpretable outputs, like strings or geometry, tended to benefit more from test feedback than those in cryptography, mathematics, or data structures, where the outputs were hard to interpret.
- For tasks that exceeded the model's capabilities, behavior varied based on feedback. In some cases, the model understood the problem but failed to correct it. In others, it misinterpreted the failure entirely and responded with generic reasoning or unrelated changes.
- Functions considered more 'standard' were generally easier for larger models to solve. One possibility is that these tasks appeared frequently in the training data, allowing the models to 'memorize' them.
- Tasks that depended on time efficiency or specific implementation details could not be properly evaluated. Since unit tests only verified output correctness, they could not enforce how a function was implemented.

4.6 Faulty User Prompts

Up to this point, the tests focused on prompts that were written to be clear and unambiguous. However, in practical usage, users often make mistakes or omit information in their requests. This is where the feedback pipeline can be particularly helpful. To test this, I created a small dataset containing two functions for each of the following types of faulty prompts: ambiguous descriptions, contradictory requirements, incorrect function names or arguments, and invalid test cases.

Ambiguous Descriptions

can_measure The number of allowed moves is not mentioned in the function description, even though the tests use this parameter. The

model's responses varied: In some cases, it insisted on using only the three parameters specified in the prompt, arguing that this matched the given description. More often, however, it recognized that the test cases referred to 'moves', and added the missing parameter.

min_window_substring This prompt consists of a single sentence that does not specify the function name. As expected, the LLM usually failed to guess the name correctly and after feedback it would eventually use the correct name.

Contradictions

collatz_steps_to_reach The description specifies two conflicting return behaviours: one stating that the function should return the maximum number in the sequence, and another saying it should return the number of steps taken. The model initially chose one option at random, but after receiving feedback, it usually adjusted to the correct output.

custom_regex_match The description includes the Question Mark symbol '(?)' in the explanation but also states that it should not be supported. Since this function is already difficult, the contradiction made it even harder to solve. The model often added support for '(?)' in the code, but also failed other tests unrelated to the contradiction. The mixed signals caused confusion, and it rarely recovered.

Incorrect Tests

heap_sort (with merge_sort tests) The function name and description ask for heap sort, but the tests verify merge sort instead. In most cases, the model recognized the issue and pointed out the incorrect tests. Interestingly, the 7B model sometimes aligned the function name to match the tests, while the 32B model never did.

most_frequent_substring One of the test cases contained an obviously incorrect expected output. Since the task itself is relatively simple, the LLM often manually evaluated the example and concluded

that the test case was flawed. Interestingly, it never attempted to hard-code a solution to satisfy the flawed test but rejected it consistently. I suspect that if a more complex function had been used, the LLM would not have identified the issue, similarly to custom_regex_match.

Name or Argument Mistakes

kth_smallest_pair_sum Here the function was badly misspelled as 'k_smalest_paer_sum'. The model fixed the spelling of 'smallest_pair', but only realized that 'k' was meant to be 'kth' after observing the feedback.

solve_linear_system The function's signature claimed it should return a 'List[int]', even though the problem clearly involves solving equations with floats. Occasionally the LLM followed the incorrect return type, but after seeing mismatches in test results, it quickly changed the return type to 'List[float]'.

Overall, these experiments show that the pipeline can be useful when handling flawed or imprecise prompts. It helps the LLM recover from surface-level issues such as typos, name mismatches, or missing details. In cases where the function is not too complex, the model typically adjusts its response after seeing the relevant test feedback. However, in tasks that are already challenging even with well-written prompts, the presence of contradictions or missing information often leads to confusion that the model cannot recover from.

4.7 Final Thoughts

While the feedback-based pipeline is not a magical solution that drastically improves performance, the experiments in this chapter demonstrate that it does indeed work, especially on tasks where the initial output is conceptually correct but contains bugs. The results show that models of all sizes can benefit from the feedback, that a well-designed prompt plays an important role in shaping model behavior, and that feedback not only helps fix code but can also mitigate unclear or flawed user instructions.

Conclusion

This thesis explored how automated unit-test feedback can guide coder large language models toward correct solutions. The results show that the approach works best in cases where the model's initial idea is close to correct but contains bugs or misunderstandings. Instead of discarding such attempts, the pipeline helps the model refine until a working solution is produced. On the other hand, when the initial approach is fundamentally flawed, the system resets, allowing different strategies to be used. Although this approach cannot improve the model's ability to solve tasks beyond its capabilities, it helps preserve partial solutions that might otherwise be lost.

There are several directions in which the system could be extended. First, adding time complexity evaluation would allow for a more complete assessment of solutions. Second, a graphical interface could make the system more practical for real users. Finally, introducing a second model to convert test results into feedback would separate reasoning from generation. This could make the process more robust and potentially improve performance.

Overall, the thesis shows that a simple feedback loop can steer a large language model toward correct solutions, marking a practical step toward reliable program synthesis.

Bibliography

1. GADESHA, Vrunda; KAVLAKOGLU, Eda. *What is text generation?* [online]. 2024-03-19. [visited on 2025-01-04]. Available from: https://www.ibm.com/think/topics/text-generation.
2. *What is a neural network?* [online]. IBM, 2024-03-19 [visited on 2025-01-04]. Available from: https://www.ibm.com/think/topics/neural-networks.
3. SANDERSON, Grant. *But what is a GPT? Visual intro to Transformers* [online]. 2024-04-01. [visited on 2025-01-04]. Available from: https://www.3blue1brown.com/lessons/gpt.
4. VASWANI, Ashish; AL., Shazeer et. Attention Is All You Need. *arXiv:1912.01703* [online]. 2017 [visited on 2025-01-30]. Available from: https://arxiv.org/abs/1706.03762.
5. *What is a transformer model?* [online]. 2024-03-19. [visited on 2025-01-04]. Available from: https://www.ibm.com/think/topics/transformer-model.
6. RAVKINE, Mike. *Can AI Code Results* [online]. [visited on 2025-01-04]. Available from: https://huggingface.co/spaces/mike-ravkine/can-ai-code-results.
7. SANDERSON, Grant. *Visualizing Attention, a Transformer's Heart* [online]. 2024-04-07. [visited on 2025-01-04]. Available from: https://www.3blue1brown.com/lessons/attention.
8. SANDERSON, Grant. *But what is a GPT? Visual intro to Transformers* [online]. 2017-11-03. [visited on 2025-01-04]. Available from: https://www.3blue1brown.com/lessons/gpt.
9. SANDERSON, Grant. *How might LLMs store facts* [online]. 2024-08-31. [visited on 2025-01-04]. Available from: https://www.3blue1brown.com/lessons/mlp.
10. *Aura* [online]. Masaryk University [visited on 2025-03-05]. Available from: https://www.fi.muni.cz/tech/unix/aura.html.en.

11. *Storage and quotas* [online]. Masaryk University [visited on 2025-03-15]. Available from: https://www.fi.muni.cz/tech/unix/quotas.html.en.

12. *Modules* [online]. Masaryk University [visited on 2025-03-15]. Available from: https://www.fi.muni.cz/tech/unix/modules.html.

13. WOLF, Thomas; AL., Debut et. Transformers: State-of-the-Art Natural Language Processing. *arXiv:1910.03771*. 2020.

14. PASZKE, Adam; AL., Gross et. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *arXiv:1912.01703*. 2019.

15. *Qwen2.5-Coder* [online]. Alibaba [visited on 2025-03-15]. Available from: https://github.com/QwenLM/Qwen2.5-Coder.

16. *GPU* [online]. Hugging Face [visited on 2025-03-15]. Available from: https://huggingface.co/docs/transformers/main/perf_infer_gpu_one.

17. *torch.compile* [online]. Hugging Face [visited on 2025-03-15]. Available from: https://huggingface.co/docs/transformers/main/perf_torch_compile.

18. *pytest: helps you write better programs* [online]. [visited on 2025-04-11]. Available from: https://docs.pytest.org/.