TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Karmo Peetermann 221657IAPM

EVALUATING TEST-DRIVEN DEVELOPMENT FOR LLM CODE GENERATION ON COMPLEX PROBLEMS

Master's Thesis

Supervisor: Ants Torim
PhD

Tallinn 2025

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Karmo Peetermann 221657IAPM

Testipõhise arenduse hindamine LLM-koodigenereerimisel keerukate probleemidega

Magistritöö

Juhendaja: Ants Torim
PhD

Tallinn 2025

Author’s Declaration of Originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Karmo Peetermann

04.05.2025

Abstract

Large Language Models (LLMs) have significantly impacted software development by automating code generation tasks, yet challenges remain in producing accurate solutions to complex programming problems. This thesis evaluates how Test-Driven Development (TDD) practices can enhance the effectiveness and reliability of LLM-generated code when addressing complex software tasks.

The research begins by analyzing different prompt creation methods, using textual descriptions, test cases, and their combination, to identify which approach yields the most accurate code generation from LLMs. Following this, the study introduces an iterative TDD-based approach to LLM-based code generation, comparing it to zero-shot methods in terms of correctness and efficiency.

Quantitative evaluations conducted on the coding benchmarks MBPP and HumanEval demonstrate that integrating TDD practices significantly improves code generation accuracy and reliability. Furthermore, a case study on custom-designed complex scenarios shows that the iterative TDD approach consistently outperforms zero-shot methods.

This thesis provides valuable insights into practical techniques for integrating TDD principles into LLM-based code generation, offering a clear strategy for software developers aiming to leverage LLM capabilities for complex software engineering challenges.

The thesis is written in English and is 67 pages long, including 6 chapters, 4 figures and 5 tables.

Annotatsioon

Testipõhise arenduse hindamine LLM-koodigenereerimisel keerukate probleemidega

Suured keelemudelid (LLMid) on oluliselt mõjutanud tarkvaraarendust, automatiseerides koodigeneratsiooni ülesandeid, kuid täpsete lahenduste loomine keerukatele programmeerimisprobleemidele on endiselt väljakutse. Käesolev lõputöö hindab, kuidas testimisel põhineva arenduse (TDD) meetodid võivad suurendada LLMide genereeritud koodi tõhusust ja usaldusväärsust keerukate tarkvaraülesannete lahendamisel.

Uurimistöö algab erinevate sisendite loomise meetodite analüüsiga, kasutades tekstilisi kirjeldusi, testjuhtumeid ning nende kahe kombinatsiooni, et tuvastada, milline lähenemine tagab kõige täpsema koodigeneratsiooni LLMide abil. Seejärel tutvustatakse iteratiivset, TDD-põhist lähenemist LLM-põhises koodigeneratsioonis, võrreldes seda nullõppe meetoditega nii täpsuse kui ka efektiivsuse osas.

Kvantitatiivsed hinnangud, mis viidi läbi koodikirjutamise võrdlusuuringutega MBPP ja HumanEval, näitavad, et TDD praktikate integreerimine parandab oluliselt koodigeneratsiooni täpsust ja usaldusväärsust. Lisaks näitab spetsiaalselt loodud keeruliste stsenaariumite juhtumiuuring, et iteratiivne TDD-lähenemine ületab järjepidevalt nullõppe meetodeid.

Antud lõputöö pakub praktilisi teadmisi TDD põhimõtete integreerimiseks LLM-põhisesse koodigeneratsiooni ning esitab selge strateegia tarkvaraarendajatele, kes soovivad rakendada LLMide võimalusi keerukate tarkvaraarendusprobleemide lahendamisel.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 67 leheküljel, 6 peatükki, 4 joonist, 5 tabelit.

List of Abbreviations and Terms

LLM	Large Language Model
TDD	Test-Driven Development
MBPP	Mostly Basic Python Problems
SFP	Simple Function Point
TF	Transactional Functions
DF	Data Functions
CoT	Chain-of-Thought

Table of Contents

1. Introduction

Large Language Models (LLMs) are powerful tools capable of performing a wide range of natural-language tasks. Tools such as Copilot [1], ChatGPT [2], Claude [3], and others have revolutionized the way software development is carried out. By providing quick suggestions and generating code for trivial problems, they have already been shown to increase the speed and efficiency of the development process [4]. However, these tools and their underlying models are still far from perfect. They often fail to produce correct output, so relying on them is not always advantageous [4].

Using these tools can be frustrating, particularly when the problem in question is highly complex. For tasks that involve extensive logic, the LLM often cannot understand the entire problem in one go. Researchers and practitioners have explored several strategies to mitigate this limitation [5, 6, 7, 8]. From an end-user perspective, the most promising approach so far is prompt engineering—supplying the LLM with additional, carefully crafted instructions [5, 6]. The instructions help to steer the generation in the correct direction, by disallowing certain actions or forcing certain patterns to the generation. That said, putting the instructions together is quite difficult as there are no clear rules how the rules affect the models. Therefor a lot of testing in this field is needed, to find possible solutions.

Various Test-Driven Development (TDD) techniques have shown good results in different coding benchmarks demonstrating their value for guiding LLMs [5, 6]. These approaches use unit tests as a directing mechanism for the code generation by providing a well structured instructions to the LLM. Furthermore the same tests can also be used as a verification tool, where the test run feedback can be automatically used to enhance the results even further. And while it has shown the potential in solving simple problems, little research has been done in validating the TDD practices with complex problems [9, 10].

This thesis evaluates TDD practices in LLM-based code generation and introduces an iterative, TDD-driven workflow. Its goal is to identify strategies that improve the success rate and reliability of generated code, enabling existing LLMs to tackle more complex tasks.

1.1 Motivation

LLMs are an amazing tool for enhancing the productivity of software development. LLMs can produce simple functionality and suggest valuable completions more quickly than engineers, reducing development time. For more complex tasks, LLMs can still offer insights, but achieving a successful generation remains a challenge. Incorrect generations create additional noise that the engineer has to work through, further reducing productivity.

As with any tool, knowing how to use the tool helps you use its full potential. For LLM-based tools, however, the technology is still fairly new, and practical know-how is limited. While theoretical benchmarks can show excellent results [5, 6], translating those gains to everyday work is not straightforward. Therefore, separate studies on real-world use cases should be conducted to reveal the models' true potential.

The thesis begins with an analysis of how the input prompt affects the generations produced by LLMs. Several methods are compared: using textual descriptions, test cases, and a combination of both. While textual descriptions are the most natural way to explain problems, the TDD framework attempts to emphasize the importance of test cases by entirely replacing the problem description [11]. However, this approach has been criticized for overly limiting the input. Therefore, a combination of textual description and tests has been suggested as the optimal method for clearly understanding the problem [12].

In the second part of the thesis, TDD practices are used to develop an iterative approach that is expected to yield better generation results compared to zero-shot methods. Incorporating the developer as a controller in the generation process is intended to enhance the capabilities of LLM-based tools, enabling developers to tackle more complex problems with the same tools.

1.2 Research objectives

The primary objective of this thesis is to evaluate the effectiveness of TDD practices in enhancing the performance of LLMs for code generation, particularly when dealing with complex problems.

To achieve this goal, the research addresses the following key questions:

1. Which approach is most efficient for prompt creation in LLM-based generation: textual descriptions, test cases, or a combination of both?

2. How does utilizing error messages as a feedback-loop mechanism affect the correctness of generated outputs?
3. How can an iterative generation approach be effectively implemented using TDD practices?
4. Do the results obtained from an iterative generation approach differ significantly from zero-shot approaches given a fixed generation cost?

To address these questions, a quantitative study will be conducted on publicly available coding benchmarks MBPP [13] and HumanEval [14] to evaluate best practices of TDD and their effectiveness. Finally, a case study will be carried out using a complex problem statement designed to reflect a real-world software-development scenario. The problem will be approached using multiple methods and compared to evaluate differences in code correctness and usability.

2. Background

In this chapter we first examine the history and development of LLMs, with a particular focus on tools created to assist software development. Next, we provide an overview of TDD and explain how these practices relate to LLM-based code generation. Finally, we review the existing work in this field, outline their processes, and identify the research gaps that this thesis tries to answer.

2.1 Large Language Models

LLMs are natural-language-processing models that have a human-like ability to understand and generate text. The core idea behind current state-of-the-art models is based on transformer architectures, which use self-attention mechanisms to process natural language [4]. Transformers read all the words from the input at once instead of one by one. Using self-attention, each word checks every other word and decides how much it should "pay attention" to them. By repeating this process in several layers, the model builds a detailed context of the whole sentence, which it then uses to understand or generate text [4].

The largest models now use up to trillions of parameters, enabling them to comprehend very complex ideas and concepts [4]. To reach this scale, they are trained on huge amounts of data. As a result, tools like ChatGPT [2] and Claude [3] can act as universal assistants for a wide range of jobs. Even when a model has not seen the exact data before, it can reason about the topic and often provide an accurate answer.

2.1.1 History and development

Although language models have existed for decades, the last breakthrough with wide commercial use came only after the introduction of transformer models [15]. The first transformer-based LLM offered commercially was OpenAI's GPT-3, released via the OpenAI API in June 2020 [4]. Since then, many companies have begun offering their own models, including Anthropic, Meta, Google, and others.

Besides the development of LLMs, there has also been a resurgence of LLM-based tools. These tools act as a mid-layer that applies LLMs in specific ways. For coding, the most famous example is likely GitHub Copilot [1], which integrates with various IDEs to give developers convenient access to LLM chat functionality.

2.1.2 Application in software development

LLM-based tools have become irreplaceable for software development, replacing and automating various parts of the job.

Code generation

LLMs can generate functional code snippets, complete partial implementations, and provide intelligent code suggestions across multiple programming languages. Tools such as GitHub Copilot [1] and OpenAI Codex [16] leverage these capabilities to help developers write code more efficiently and accurately.

Automated testing and debugging

LLMs can generate small helper functions or individual logic blocks. These intelligent suggestions provide an effective autocomplete capability, boosting developers' productivity. Instead of having to consult documentation, built-in tools such as Copilot explain the relevant code patterns in seconds.

Software design and architecture

LLMs can also suggest ways to design systems. Their strong pattern-recognition abilities often point in the right direction, helping developers produce better-designed solutions. These suggestions are useful both for generating initial ideas and for refining an existing architecture.

2.1.3 LLM-based tools

The first commercially available applications were general-purpose conversational search engines such as ChatGPT [2]. These systems quickly proved more helpful for many information-seeking tasks than traditional search engines like Google, which in turn motivated competitors such as Claude, Gemini, and others.

Beyond these general-purpose chatbots, an explosion of domain-specific tools has emerged, all powered by LLMs. For researchers, Elicit [17] and ResearchRabbit [18] can summarize papers, generate citations, and surface relevant studies. In software development, Copilot [1] and Tabnine [19] integrate directly into popular IDEs, putting LLM assistance at developers' fingertips. Meanwhile, text-processing utilities such as Notion AI [20] and Grammarly [21] provide spelling checks, style improvements, and proofreading.

2.2 Test-Driven Development

Test-Driven Development (TDD) is a software-development methodology that provides a clear framework for producing robust, well-tested code. The key idea is to write tests before implementing the code, thereby defining and constraining the required functionality in advance. The implementation is then grown iteratively until it satisfies the initial requirements. This approach helps ensure high code quality and supports a modular design [4].

Although TDD is not directly discussed in LLM-based code-generation workflows, several of its principles are adopted to improve results. Two common practices are using unit tests as part of the prompt and feeding error messages from failing test runs back into the model similar to the Red-Green-Refactor cycle in TDD [5, 6]. To understand these parallels more clearly, we first discuss the core TDD practices.

2.2.1 Red-Green-Refactor Cycle

In the iterative approach, the Red-Green-Refactor cycle is used. As the name suggests, this cycle has three phases, and the goal is to complete one piece of logic by the end of each iteration. The cycle should therefore be applied only to problems small enough to understand and solve in a single iteration. If the problem is too large, the cycle should be restarted on a smaller sub-problem.

Red phase

The red phase begins by writing tests for functionality that does not yet exist. Because these tests fail, most development tools highlight them in red. Since no implementation is present, simple mocked values are often used. By starting with a failing test, developers establish a natural flow in which the newly written logic is always validated immediately, promoting a quick feedback loop [11].

For more complex logic, the triangulation method can be employed. Triangulation involves writing several test cases for the same feature to clarify its general behavior and to cover important edge cases [11].

The red phase can be compared to the prompt creation in LLM-based code generation. The developer begins by defining the problem in a particular way to gain a clearer understanding of the task at hand.

Green phase

The green phase's goal is to write the minimal amount of code needed to pass the previously written tests. This locks down the required functionality and clarifies the core of the issue. If the expected behavior is still unclear, it is acceptable to begin with mocked responses and generalize the implementation later as new test cases are added. When the task is trivial, the full solution can be implemented in one step.

Refactor

The final step of the cycle is refactor. Here, both the tests and the implementation are cleaned up and reorganized into more logical components. It is important not to change the observable behavior of the code. Frequent test runs help ensure that refactoring does not introduce regressions. This step is essential for keeping the codebase scalable and readable.

2.2.2 Benefits and challenges of TDD

The TDD framework was popularized by Martin Fowler as a way to introduce order into the software-development process and to provide a structured learning curve—especially for newcomers to the field. By enforcing sound practices, the framework aims to raise the overall quality of the written code. A few advantages include naturally good test coverage, clear guidance for writing code, and the continual improvement that comes from regular refactoring [11].

Although a number of studies have examined TDD, their findings are often mixed [22]. The main criticism is that TDD is difficult to learn, slowing down developers who are trying to adopt the new framework. Moreover, it is easy to focus too much on simple, pointless tests, which can further complicate and slow down software projects [22]. Because the success of TDD still depends heavily on the interpretation and skill of the engineers, it can be difficult to quantify the exact value that TDD can give.

2.3 Use of TDD practices with LLMs

LLMs can assist at various stages of the TDD process, enhancing productivity and reducing manual effort. Different methods aim to automate parts of the workflow, but they share the common goal of maximizing the correctness of the generated code.

2.3.1 Prompt engineering

Prompt engineering is the simplest way to guide the generation process. One particularly effective method is to include test cases directly within the prompt [5]. When test cases are used, the model benefits from clear and specific constraints, reducing ambiguity in the instructions. Consequently, the generated code more closely matches the specified requirements [5]. This approach helps overcome the ambiguity common in textual prompting, as test cases provide explicit examples of desired outcomes.

Several strategies exist to improve prompt effectiveness under different scenarios. This section briefly introduces some key strategies and terms used throughout this thesis.

Zero-shot prompting

Zero-shot prompting is a method in which the model is given only a direct question or task without any additional examples. This approach generates the most general responses, as the model treats the input as a broad question. Although zero-shot prompting is simple, it can lead to inconsistent results, particularly for tasks that require complex reasoning.

Few-shot prompting

For more complex tasks, especially those requiring detailed reasoning, adding a few representative examples significantly enhances the model's performance [23]. Few-shot prompting enables in-context learning by providing the model with specific examples, guiding the generation towards the desired outcome.

Chain-of-thought prompting

When handling complex tasks like detailed calculations or logical reasoning, simply providing correct examples may not be sufficient. Instead, explicitly describing each reasoning step can greatly improve model understanding. This approach, known as chain-of-thought (CoT) prompting, helps models better grasp challenging logical and mathematical concepts [23]. CoT prompting can be combined effectively with few-shot prompting by including multiple detailed examples, further enhancing performance.

Iterative prompting

Iterative prompting involves repeatedly refining and modifying prompts based on earlier outputs. This method is similar to interacting with chatbots: if an initial answer is insufficient or incorrect, further prompts clarify or correct the model's response. Typically, iterative prompting requires manual intervention, where human input or feedback is necessary to guide improvements. However, recent studies have explored automating this iterative process by having multiple LLM agents collaboratively refine prompts and responses to achieve higher accuracy and reliability [5, 6].

2.3.2 Iterative feedback loop with test failures

The iterative feedback loop with test failures mirrors the Red-Green-Refactor cycle of TDD, using an iterative approach to improve code step by step. First, the LLM generates an initial implementation based on the problem statement and provided test cases. That code is executed against the tests; any failures or error messages are captured and fed back to the model [5]. The LLM corrects the issues, produces a revised version, and the process repeats until all tests pass. In this way, the model incrementally refines the solution, following the usual developer workflow of debugging and refinement.

This strategy boosts the reliability and correctness of the output, as the LLM systematically adapts to feedback. Its effectiveness, however, depends on the quality of the initial test suite and leaves little room for human intervention, since engineers remain outside the loop.

2.3.3 LLM-generated test cases

Another way to use an LLM is to generate test cases from a given problem description. This aligns with the core principle of TDD, where tests are written first to clarify requirements and expected behavior. By prompting the LLM to produce test cases, developers can automate the creation of input–output pairs and edge cases, guiding the LLM toward better code.

This method reduces the manual effort of writing tests and accelerates the TDD process, especially for simple to moderately complex problems. However, the quality and completeness of the generated tests are critical: incomplete or incorrect cases can lead to flawed implementations [24, 25, 26]. Human review or additional validation steps are therefore needed to ensure adequate coverage and accuracy.

2.3.4 Multi-agent systems

Multi-agent systems adopt a collaborative framework in which specialized agents—each assigned a distinct role in the TDD process—work together to enhance LLM-driven code generation. Typically, several LLM agents simulate a TDD workflow: a test agent produces test cases from the problem description, a code agent implements code to satisfy those tests, and a validation agent runs the tests, validates the code, and feeds back any failures [5, 6]. This division of responsibilities mirrors the staged nature of a conventional TDD process.

The aim of these systems is to maximize the potential of LLMs, and they have achieved strong results on coding benchmarks such as MBPP [13], HumanEval [14], and others [5, 6]. Nonetheless, most existing multi-agent frameworks remain prototype research tools optimized mainly for benchmark performance. Further work is needed to generalize them for everyday, real-world applications.

2.3.5 Automatic systems with human feedback

For a more guided approach, human actors can take on supervisory roles within otherwise automated workflows. This setup combines the speed of automated code generation with human expertise at critical points.

Implementations vary, but the human role typically involves verifying intermediate results or specifying additional criteria. The LLM then uses this feedback to adjust its output and satisfy those requirements, producing more reliable code [27, 7, 28, 8].

2.3.6 Research gaps

Research has demonstrated that applying TDD principles to code generation yields good results for more simple problems. However, a significant gap remains in clear guidance on how to use these models effectively in everyday work, especially for highly complex software-development challenges. Existing studies are often confined to benchmark tasks and tend to overlook practical engineering practices for deploying the available tools and models.

This thesis aims to identify methods that maximize the value of LLM-based code-generation tools.

3. Solving simple problems

To start exploring how prompts affect code generation, we first look at simpler problems. This approach is faster since many public datasets are available for LLM evaluation. These benchmarks clearly show the strengths and weaknesses of language models by offering problems of different difficulties. By checking how often the generated code passes tests, we can easily compare different prompting methods.

In this chapter, we examine how different prompt creation methods influence the quality of generated solutions. We focus on methods similar found in TDD, using unit tests and the red-green-refactor cycle. We look at solutions created with varying detail levels in problem descriptions and also study how well models improve solutions when given error feedback.

3.1 Evaluation setup

The two chosen benchmarks are MBPP [13] and HumanEval [14], both commonly used to measure and compare the performance of LLMs. These benchmarks contain a variety of problem statements with increasing complexity. This makes them useful for comparing the success rates of different models in a consistent and quantitative way.

3.1.1 Example problem statements

To understand how different inputs influence code generation, we first compare the problem statements and test suites provided by each benchmark.

MBPP benchmark

The Mostly Basic Python Programming (MBPP) benchmark contains 974 crowd-sourced Python problems [13]. Each task is written as a short, free-form description, mirroring the way entry-level programmers often phrase questions (Listing 3.1).

```
Write a function to find the minimum cost path to reach (m, n) from (0,
    0) for the given cost matrix cost[][] and a position (m, n) in
   cost[][].
```

Listing 3.1. MBPP problem statement

Every MBPP problem comes with a small test suite of three assertions on average. The tests contain no comments, only direct logic checks (Listing 3.2).

```
"assert min_cost([[1, 2, 3], [4, 8, 2], [1, 5, 3]], 2, 2) == 8",
"assert min_cost([[2, 3, 4], [5, 9, 3], [2, 6, 4]], 2, 2) == 12",
"assert min_cost([[3, 4, 5], [6, 10, 4], [3, 7, 5]], 2, 2) == 16"
```

Listing 3.2. MBPP test assertions

HumanEval benchmark

HumanEval, created by OpenAI, consists of 164 Python programming problems designed for LLM evaluation [14]. Unlike MBPP, each HumanEval task is structured like a function signature and includes a comment with examples (Listing 3.3).

```
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
\"\"\" Check if in given list of numbers, are any two numbers closer to
    each other than given threshold.
>>> has_close_elements([1.0, 2.0, 3.0], 0.5)
False
>>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
True
\"\"\"
```

Listing 3.3. HumanEval problem statement

The accompanying tests follow the same simple assertion style as MBPP (Listing 3.4). However, HumanEval provides an average of 7.7 test cases per problem, which is more than twice the number in MBPP. That said, the tests provided do not provide any descriptive information, having even the test names as "candidate".

```
assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3) == True
assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.05) == False
assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.95) == True
assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.8) == False
assert candidate([1.0, 2.0, 3.0, 4.0, 5.0, 2.0], 0.1) == True
assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 1.0) == True
assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 0.5) == False
```

Listing 3.4. HumanEval test assertions

3.1.2 Evaluation methods

In this thesis, two aspects of code generation were examined in the context of TDD practices. First, we studied how the amount of detail in the input prompt influences the resulting code. We compared solutions generated from a textual description only, unit tests only, and a combination of both. Second, we investigated whether an iterative feedback loop can correct a failed output. Each failed solution is executed against the test suite, and any failure messages are fed back into the model for a new generation.

Because the two benchmarks contain more than one thousand tasks in total, all experiments were run with an automated evaluation script.

3.2 Evaluating aspects of TDD efficiency

In this section, we describe how the automated evaluation was created and configured.

3.2.1 Test run setup

To handle thousands of test cases across several cycles, we wrote a script that automatically pulls each problem statement and its tests from the benchmark datasets. The script was integrated with OpenAI API, using the model 4o-mini [2]. The model choice was taken on base of low cost to make it sensible to run thousands of generations.

The written functionality allowed to generate solutions based on different inputs and also run real time test validation on generated code. This allowed to support evaluating both methods of generation and using the test feedback as an input for correcting generations.

3.2.2 Different methods of generation

From the benchmarks there are in large two sets of available information: problem description and the tests. To evaluate how the input creation affects the generation, the available information was used to make five different categories of inputs. Those are:

1. Problem description.
2. Test cases.
3. Problem description plus 1 test case.
4. Problem description plus 2 test cases.
5. Problem description plus all test cases.

For each problem, these five categories were used to generate the outputs, which could later be used to evaluate the importance of the input.

3.2.3 Feedback loop efficiency

For iterative prompting analysis with the test feedback, additional functionality was added to the run setup. For this, the setup needed to first run the tests automatically and also allow the test run information to be fed back to the LLM.

For base case, the problem generation from category problem description with all available tests was used. For any failing test runs, the feedback system used both the current state of the code and the test run information as an input for the LLM. Each test problem was allowed up to 10 feedback generation before failure was allowed. As each generated iteration is also saved, it is possible to afterwards compare the correctness by the feedback generation.

3.3 Gaps in analyzing the simple cases

While quantitative research on simpler problems provides valuable information about the effectiveness of various code-generation methods, these studies mainly focus on maximizing model capabilities. Because of this, the research overlooks how to incorporate code generation into engineers' everyday work. In real-world scenarios, problems are rarely simple enough to be solved by a single function and usually require integrating or modifying existing logic.

To better understand how code generation can fit into daily engineering tasks, it is important to identify and test practical approaches. This involves studying which ways of applying code generation give the most value for more complex problem statements, including problems that current models cannot consistently solve from the description alone.

In the following paragraph, we focus on an iterative approach to solving more complex problems, drawing inspiration from the TDD framework. The goal is to develop a workflow in which code-generation tools work hand in hand with developers, enabling them to tackle increasingly complex tasks. This approach has the potential to improve efficiency significantly.

4. Solving complex problems

Moving towards more real world like problems first we need to find a problem to actually solve. And while it is possible to find some famous examples from literature, it is quite likely that the more known problems have been used in the training of the LLM models as either the problem statement itself or from it's solution available in the internet. To prevent this, the problem statement was generated only for this thesis, reflecting a real-world programming problem.

To analysis the problem and how to solve this, four different approaches were taken. The in-depth analysis was possible as the qualitative research gave a high degree of freedom to manipulate the input of the code generation.

4.1 Evaluation setup

In this section, we describe the setup and steps taken to carry on the evaluation of solving the programming problems in multiple different approaches. In the evaluation we will confirm previously analyzed prompt engineering methods (Chapter 3) and propose an iterative generation workflow to improve the generation results.

For all the generation, the OpenAI model 4o-mini [2] was used.

4.1.1 Problem statement

In the evaluation we will focus on two different real-world inspired problem statements. For both, the topic is parking price calculations, which was chosen as a familiar topic for the author to create sensible scenarios.

The two problems are with varying complexity, allowing to evaluate the methods with raising complexity. The less complex problem, requires a simple parking calculation with fixed price (Appendix 2). The more complex problem tries to generalize the parking calculations by allowing a more flexible parking structure (Appendix 3). To show the difference in complexity multiple different methods could be applied. In this case the Simple Function Point (SFP) method is used.

Problem	TF	DF	Total
Parking with discount	3	1	19
Generalized parking with reservation	5	2	34

Table 1. SFP summary table for problem statements

Evaluating complexity

The SFP has shown to be a valid method of evaluating the complexity of problems on the requirements [29]. The SFP approach allows us to evaluate the two primary types of software functions: transactional functions (TF) and data functions (DF). TF is defined as inputs, outputs, and queries, and DF is defined as internal and/or external data interactions.

The total score is calculated as follows:

$$SFP = (\text{Number of TF} \cdot 4) + (\text{Number of DF} \cdot 7) \tag{4.1}$$

Let's add all the data from the problem statements to the table (Table 1).

While the total score does not give a definite answer on the overall complexity of the problem statements, we can verify that compared to each other, the problems have a different complexity, with the more generalized one being more complex (Table 1).

4.1.2 Evaluation methods

The evaluation was largely divided into two: iterative and zero-shot approach. In the iterative approach, the problem statement was solved using the proposed TDD based workflow. In the zero-shot approaches, the same problems were analyzed in a similar methodology as described analyzing the different benchmarks for simple problems (Chapter 3).

TDD based iterative generation

In the iterative TDD based generation, a step wise approach is taken for the generation. The generation was started with only one test case, which was used for the input. CoT was used to provide additional descriptive information and the previously generated solutions are also added as context for the input to help understand the state of the code. Throughout the iterative steps, the TDD Red-Green-Refactor cycle is used as the basis of development. The process was stopped once all initial requirements were fulfilled.

As a result, test cases were implemented with corresponding functionality for both of the problems. In the approach all the code was written by the LLM, with the developer only manipulating the input prompt in various different ways. These include the use of CoT, context augmentation and refactoring of the test cases.

Zero-shot generations

In the zero-shot generations, the methods follow a similar pattern as was done for the benchmark evaluations (Chapter 3). The following different prompt creation methods were used:

1. Problem description.
2. Test cases.
3. Problem description plus test cases.

The test cases created during the iterative approach were reused for zero-shot generations. This decision was intentional, as these tests had already been verified to generate working solutions and covered the intial requirements. Additionally, this allowed for direct comparison of the iterative approach with zero-shot methods using the exact same input.

To ensure a fair comparison between zero-shot and iterative approaches, the zero-shot approaches were executed multiple times. The number of generations matched the number of generations used in the final solutions walkthroughs of the iterative approach, ensuring a similar generation cost. Since generation outputs from the model are not deterministic, this approach enables a more equitable comparison, considering that iterative methods require multiple generations.

4.2 Iterative approach walkthrough: Parking with discount

In this section the iterative approach is described in the step like manner as it was developed. This shows both the successes and failures of the process with steps on how to handle different situations. The section is loosely divided into different subsection to highlight certain technique or process.

4.2.1 Triangulation

From the problem statement, there are three requirements: input, output and constraints put on the parking price calculation. These are:

- Input: parking start and end time (in HH:MM 24-hour format).
- Output: total parking cost.
- Constraint: discount functionality.

To start with the iterative process, the most simple parking case is taken. This includes a simple parking with a cost (Listing 4.1). Additionally the test already specifies a wanted architectural decision of having additional class `ParkingSession`, which holds the information about the session length. The idea here is to abstract the parking information into a separate class to limit the functionality of the `ParkingCalculator` to only the calculations.

```
@Test
fun underTwoHourRegularParking() {
    // Parking up to 2 hours should cost $3.00
    val session = ParkingSession("09:00", "10:30")
    val parkingCalculator = ParkingCalculator()
    val result = parkingCalculator.calculateParkingCost(session)
    val expected = 3.00
    assertEquals(expected, result)
}
```

Listing 4.1. Parking initial test case

The results of the generation.

```
data class ParkingSession(val startTime: String, val endTime: String)

class ParkingCalculator {

    fun calculateParkingCost(session: ParkingSession): Double {
        // Parse the start and end times from the session
        val start = LocalTime.parse(session.startTime)
        val end = LocalTime.parse(session.endTime)

        // Calculate the duration in minutes
        val duration = Duration.between(start, end).toMinutes()

        // If the duration is less than or equal to 120 minutes (2 hours), return $3.00
        return if (duration <= 120) {
            3.00
        } else {
            // You can add more logic for longer durations if needed
            0.0 // Default case (this could be modified depending on additional requirements)
        }
```

```
    }
}
```

Listing 4.2. Parking calculator failure state

The solution input is taken into consideration and the generated model supports the object of `ParkingSession` and `ParkingCalculator` (Listing 4.2). The generated result is quite strictly made to the given example case and supports only one case of parking lasting up to 120 minutes. To get past that we use the triangulation method and add additional tests.

After two more iterations of similar tests with varying durations and pricing we generated the following code.

```
// ParkingSession class to hold start and end times
data class ParkingSession(val startTime: String, val endTime: String)

// ParkingCalculator class to compute parking cost
class ParkingCalculator {

    // Method to calculate the parking cost
    fun calculateParkingCost(session: ParkingSession): Double {
        // Define the time format for parsing the string to LocalTime
        val formatter = DateTimeFormatter.ofPattern("HH:mm")

        // Parse the start and end times into LocalTime objects
        val start = LocalTime.parse(session.startTime, formatter)
        val end = LocalTime.parse(session.endTime, formatter)

        // Calculate the duration in minutes
        val durationInMinutes = java.time.Duration.between(start, end).
    toMinutes()

        // Calculate the parking cost
        return when {
            durationInMinutes <= 120 -> 3.00  // Up to 2 hours
            durationInMinutes <= 360 -> 3.00 + (durationInMinutes -
    120) * 1.5 / 60  // Between 2 and 6 hours
            else -> 15.00  // Over 7 hours
        }
    }
}
```

Listing 4.3. Parking calculator corrected state

In the solution we can see that the code still quite strictly sticks to the input and does not make any generalizations (Listing 4.3). This is fine in our case as our tests cover the necessary functionality, giving precisely defined length and prices.

4.2.2 Feedback mechanism from test run failures

Next the discount functionality will be implemented. To make the the discount logic more understandable, CoT technique will be applied to describe the discount applying logic.

```
 @Test
fun discountParking() {
    // Parking where most of the duration is between 20:00 and 06:00 should have a discount of 20 percent
    val session = ParkingSession("20:00", "23:00")
    val parkingCalculator = ParkingCalculator()
    val result = parkingCalculator.calculateParkingCost(session)
    // Regular price would be 4.50 but as discount applies, it is lowered by 20 percent to 3.60
    val expected = 3.60
    assertEquals(expected, result)
}
```

Listing 4.4. Parking with discount test case initial

In the test both the discount timing and price calculation have been described with the CoT technic (Listing 4.4). This is important in this scenario as otherwise the discount period would not be defined elsewhere. The test results in the following code.

```
data class ParkingSession(val startTime: String, val endTime: String)

class ParkingCalculator {

    fun calculateParkingCost(session: ParkingSession): Double {
        // Convert the start and end times from string to LocalTime
        val start = LocalTime.parse(session.startTime)
        val end = LocalTime.parse(session.endTime)

        // Calculate the duration in minutes
        val durationInMinutes = Duration.between(start, end).toMinutes()

        // Calculate total hours (round up if necessary)
        val totalHours = (durationInMinutes / 60.0).let { if (it % 1 != 0.0) it + 1 else it }

        var cost = 0.0

        when {
```

```
            totalHours <= 2 -> {
                cost = 3.00
            }
            totalHours in 2.1..6.0 -> {
                cost = 3.00 + (totalHours - 2) * 1.50
            }
            totalHours > 7 -> {
                cost = 15.00
            }
        }

        // Apply discount if the parking is during the night time (between 8:00 PM and 6:00 AM)
        if (isNightTime(session)) {
            cost *= 0.8  // 20% discount
        }

        return cost
    }

    private fun isNightTime(session: ParkingSession): Boolean {
        val start = LocalTime.parse(session.startTime)
        val end = LocalTime.parse(session.endTime)

        // Check if most of the session is between 8:00 PM and 6:00 AM
        return (start.isAfter(LocalTime.of(20, 0)) || start.isBefore(LocalTime.of(6, 0))) &&
               (end.isAfter(LocalTime.of(20, 0)) || end.isBefore(LocalTime.of(6, 0)))
    }
}
```

Listing 4.5. Parking with discount initial failure

Here we see our initial failures. There are two tests failing, previously added pricing test `underTwoHourRegularParking` (Listing 4.1) and the newly added `discountParking` (Listing 4.4).

Let's use the feedback loop mechanism and add the test failures as an input for the LLM prompt. After first iteration we manage to reduce the error count by one. The remaining failing test is the newly added `discountParking` (Listing 4.4). As we saw from the paragraph 3, there is some potential benefit in allowing multiple iterations of allowing the LLM tool to fix the code. Let's run a second iteration with the new error code as an input.

After the second iteration we have the following generated code.

```
class ParkingCalculator {

    fun calculateParkingCost(session: ParkingSession): Double {
        // Convert the start and end times from string to LocalTime
        val start = LocalTime.parse(session.startTime)
        val end = LocalTime.parse(session.endTime)

        // Calculate the duration in minutes
        val durationInMinutes = Duration.between(start, end).toMinutes()

        // Calculate total hours (round up if necessary)
        val totalHours = (durationInMinutes / 60.0).let {
            // Round up to the next whole hour if there’s any remaining minutes
            if (it % 1 != 0.0) Math.ceil(it) else it
        }

        var cost = 0.0

        when {
            totalHours <= 2 -> {
                cost = 3.00
            }
            totalHours in 2.1..6.0 -> {
                cost = 3.00 + (totalHours - 2) * 1.50
            }
            totalHours > 7 -> {
                cost = 15.00
            }
        }

        // Apply discount if the parking is during the night time (between 8:00 PM and 6:00 AM)
        if (isNightTime(session)) {
            cost *= 0.8  // 20% discount
        }

        return cost
    }

    private fun isNightTime(session: ParkingSession): Boolean {
        val start = LocalTime.parse(session.startTime)
        val end = LocalTime.parse(session.endTime)

        // Check if the session overlaps with the night time (between 8:00 PM and 6:00 AM)
```

```
        return (start.isAfter(LocalTime.of(20, 0)) || start.isBefore(LocalTime.of(6, 0))) ||
               (end.isAfter(LocalTime.of(20, 0)) || end.isBefore(LocalTime.of(6, 0)))
    }
}
```

Listing 4.6. Parking with discount calculator initial successful state

This time all the tests pass. But from a quick analysis of the function `isNightTime` (Listing 4.6) we can determine that there are some shortcomings from the original criteria. To generalize the functionality, let use the triangulation method again by adding additional tests.

```
@Test
fun nonDiscountParkingEndingInDiscountPeriod() {
    // Parking where most of the duration is not between 20:00 and 06:00 should not have a discount of 20 percent
    val session = ParkingSession("18:00", "21:00")
    val parkingCalculator = ParkingCalculator()
    val result = parkingCalculator.calculateParkingCost(session)
    // Regular price would be 4.50 but as discount applies, it is lowered by 20 percent to 3.60
    val expected = 4.50
    assertEquals(expected, result)
}
```

Listing 4.7. Parking with discount additional test case

From the output let's only focus on the discount application functionality. This can be summed up with two of functions `isDicsountApplicable` and `getDurationInHours` (Listing 4.8).

```
// Simple function to parse time and calculate difference in hours
private fun getDurationInHours(session: ParkingSession): Double {
    val dateFormat = SimpleDateFormat("HH:mm", Locale.getDefault())
    val start = dateFormat.parse(session.startTime)
    val end = dateFormat.parse(session.endTime)

    val diffInMillis = end.time - start.time
    return diffInMillis.toDouble() / (1000 * 60 * 60) // Convert from milliseconds to hours
}

// Function to check if the parking time was during the discount period (20:00 to 06:00)
private fun isDiscountApplicable(session: ParkingSession): Boolean {
```

```
    val dateFormat = SimpleDateFormat("HH:mm", Locale.getDefault())
    val start = dateFormat.parse(session.startTime)
    val end = dateFormat.parse(session.endTime)

    val discountStart = dateFormat.parse("20:00")
    val discountEnd = dateFormat.parse("06:00")

    val totalDurationInMillis = end.time - start.time
    val discountDurationInMillis = (end.time.coerceAtMost(discountEnd.
   time) - discountStart.time).coerceAtLeast(0)

    return discountDurationInMillis.toDouble() / totalDurationInMillis
   > 0.5
}
```

Listing 4.8. Parking with discount functionality initial state

Test run fails again with the first discount test failing `discountParking` (Listing 4.4).

Again, let's first try the feedback loop mechanism. With the additional error message we generate the following code.

```
// Simple function to parse time and calculate difference in hours
private fun getDurationInHours(session: ParkingSession): Double {
    val dateFormat = SimpleDateFormat("HH:mm", Locale.getDefault())
    val start = dateFormat.parse(session.startTime)
    val end = dateFormat.parse(session.endTime)

    val diffInMillis = end.time - start.time
    return diffInMillis.toDouble() / (1000 * 60 * 60) // Convert from
   milliseconds to hours
}

// Function to check if the parking time was during the discount period
     (20:00 to 06:00)
private fun isDiscountApplicable(session: ParkingSession): Boolean {
    val dateFormat = SimpleDateFormat("HH:mm", Locale.getDefault())
    val start = dateFormat.parse(session.startTime)
    val end = dateFormat.parse(session.endTime)

    // Define discount period: 20:00 to 06:00
    val discountStart = dateFormat.parse("20:00")
    val discountEnd = dateFormat.parse("06:00")

    // Calculate total duration of the parking session in milliseconds
    val totalDurationInMillis = end.time - start.time

```

```
    // Find the overlap between the parking session and discount window (20:00-06:00)
    val overlapStart = maxOf(start.time, discountStart.time)
    val overlapEnd = minOf(end.time, discountEnd.time + 24 * 60 * 60 * 1000) // 06:00 on the next day

    val overlapDurationInMillis = maxOf(0, overlapEnd - overlapStart)

    // Check if more than 50% of the parking time is in the discount period
    return overlapDurationInMillis.toDouble() / totalDurationInMillis > 0.5
}
```

Listing 4.9. Parking with discount fuctionality final state

This time all tests pass.

4.2.3 Context augmentation

Comparing the last the code generation (Listings 4.8 and 4.9) we can see that the final iteration added the functionality to actually compare the overlaps between parking and discount period. To verify the functionality, let's add the specific case of an overnight parking session.

```
@Test
fun discountParkingEndingOverNight() {
    // Parking where most of the duration is between 20:00 and 06:00 should have a discount of 20 percent
    val session = ParkingSession("23:00", "02:00")
    val parkingCalculator = ParkingCalculator()
    val result = parkingCalculator.calculateParkingCost(session)
    // Regular price would be 4.50 but as discount applies, it is lowered by 20 percent to 3.60
    val expected = 3.60
    assertEquals(expected, result)
}
```

Listing 4.10. Parking with discount test case for overnight

In the test run we see again failing tests. Both initial `discountParking` (Listing 4.4) and newly added `discountParkingEndingOverNight` (Listing 4.10) fail. As previously, let's first try the LLM to correct the mistake by inputting the error message to the prompt.

After three iterations, the same errors however persist and we can not pass the tests without taking additional steps. For this let's add the latest working state (Listing 4.9) as a context. This way the LLM tool should stick more to the existing functionality and lower the chance of previous tests breaking. With the previous state and tests, we generate the following code.

```
data class ParkingSession(val startTime: String, val endTime: String)

class ParkingCalculator {

    private val discountRate = 0.2
    private val regularRatePerHour = 1.5
    private val baseRate = 3.0
    private val flatRate = 15.0

    // Simple function to parse time and calculate difference in hours
    private fun getDurationInHours(session: ParkingSession): Double {
        val dateFormat = SimpleDateFormat("HH:mm", Locale.getDefault())
        val start = dateFormat.parse(session.startTime)
        val end = dateFormat.parse(session.endTime)

        // Handle parking sessions spanning across midnight
        if (end.before(start)) {
            end.time += 24 * 60 * 60 * 1000 // Add 24 hours to end time to adjust for the next day
        }

        val diffInMillis = end.time - start.time
        return diffInMillis.toDouble() / (1000 * 60 * 60) // Convert from milliseconds to hours
    }

    // Function to check if the parking time was during the discount period (20:00 to 06:00)
    private fun isDiscountApplicable(session: ParkingSession): Boolean {
        val dateFormat = SimpleDateFormat("HH:mm", Locale.getDefault())
        val start = dateFormat.parse(session.startTime)
        val end = dateFormat.parse(session.endTime)

        // Define discount period: 20:00 to 06:00
        val discountStart = dateFormat.parse("20:00")
        val discountEnd = dateFormat.parse("06:00")

        // Handle parking sessions spanning across midnight
        if (end.before(start)) {
            end.time += 24 * 60 * 60 * 1000 // Add 24 hours to end time
        }
```

```

        // Calculate total duration of the parking session in milliseconds
        val totalDurationInMillis = end.time - start.time

        // Find the overlap between the parking session and discount window (20:00-06:00)
        val overlapStart = maxOf(start.time, discountStart.time)
        val overlapEnd = minOf(end.time, discountEnd.time + 24 * 60 * 60 * 1000) // 06:00 on the next day

        val overlapDurationInMillis = maxOf(0, overlapEnd - overlapStart)

        // Check if more than 50% of the parking time is in the discount period
        return overlapDurationInMillis.toDouble() / totalDurationInMillis > 0.5
    }

    // Function to calculate parking cost based on the session
    fun calculateParkingCost(session: ParkingSession): Double {
        val duration = getDurationInHours(session)

        val baseCost = when {
            duration <= 2 -> baseRate // up to 2 hours
            duration <= 6 -> baseRate + (duration - 2) * regularRatePerHour // between 2 and 6 hours
            duration > 7 -> flatRate // over 7 hours
            else -> baseRate // Default to base rate in other cases
        }

        // Apply discount if applicable
        return if (isDiscountApplicable(session)) {
            baseCost * (1 - discountRate)
        } else {
            baseCost
        }
    }
}
```

Listing 4.11. Parking with discount final state

All tests pass. Also analyzing the original requirements we can determine that the functionality is covering all necessary use cases.

4.3 Iterative approach walkthrough: Generalized parking with reservation

In this section the second problem of generalized parking with reservation is worked solved. The approach will remain the same from the previous example.

4.3.1 Triangulation

For the second problem we take the parking calculation problem and will generalize the pricing structure. To do this we have additional input parameter to define the pricing structure on a zone level. Also, there's additional reservation logic to add additional complex flows to the mix. The requirements are as followed.

- Input: zone, start time and end time (HH:MM format), reservation.
- Output: total parking cost.
- Constraint: reservation logic.

Let's start the implementation on a similar path as previously. But this time add the parking zone object as a parameter to define the importance of the more generalized approach (Listing 4.12).

```
@Test
fun oneHourRegularParking() {
    val parkingCalculator = ParkingCalculator()
    val result = parkingCalculator.calculateParkingCost(
        startTime = LocalTime.of(10, 0),
        endTime = LocalTime.of(11, 0),
        parkingZone = ParkingZone(
            name = "Zone A",
            periods = listOf(
                ParkingPeriod(
                    startTime = LocalTime.of(0, 0),
                    endTime = LocalTime.of(23, 59),
                    pricePerHour = 1.00,
                )
            )
        ),
    )
    assertEquals(1.00, result)
}
```

Listing 4.12. Generalized parking calculator initial test case

The generated code follows the objects exactly, creating multiple data classes for `ParkingPeriod`, `ParkingZone`, `ParkingPeriod` and `ParkingZone`. The `ParkingCalculator` was generated as follows (Listing 4.13).

```
class ParkingCalculator {

    fun calculateParkingCost(startTime: LocalTime, endTime: LocalTime, parkingZone: ParkingZone): Double {
        // Calculate the parking duration in minutes
        val durationInMinutes = Duration.between(startTime, endTime).toMinutes()

        // Round up to the nearest 15 minutes
        val roundedDurationInMinutes = ((durationInMinutes + 14) / 15) * 15

        // Calculate the duration in hours (for price calculation)
        val hours = roundedDurationInMinutes / 60.0
        val pricePerHour = parkingZone.periods.firstOrNull()?.pricePerHour ?: 0.0

        // Calculate the cost
        return hours * pricePerHour
    }
}
```

Listing 4.13. Generalized parking calculator calculator initial state

All tests pass. From the code we can see that the `ParkingZone` is indeed used for the parking, but there is no logic on how to pick the correct matching period. To force the generation to add the corresponding logic, let's use the triangulation and add tests for `ParkingZone` with multiple `ParkingPeriod` entities.

```
@Test
fun parkingInParkingZoneWithMultiplePeriods() {
    val parkingCalculator = ParkingCalculator()
    val parkingPeriod = ParkingZone(
        name = "Zone A",
        periods = listOf(
            ParkingPeriod(
                startTime = LocalTime.of(0, 0),
                endTime = LocalTime.of(12, 0),
                pricePerHour = 1.00,
            ),
            ParkingPeriod(
                startTime = LocalTime.of(12, 0),
                endTime = LocalTime.of(23, 59),
```

```
                pricePerHour = 2.00,
            )
        )
    )

    val result = parkingCalculator.calculateParkingCost(
        startTime = LocalTime.of(12, 0),
        endTime = LocalTime.of(13, 0),
        parkingZone = parkingPeriod,
    )
    assertEquals(2.00, result)
}
```

Listing 4.14. Generalized parking test case with multiple parking periods

In the test we specified multiple `ParkingPeriod` entities with the parking only starting in the second of two (Listing 4.14). This way we force a period change within the logic. We generate the following code (Listing 4.15).

```
class ParkingCalculator {

    fun calculateParkingCost(
        startTime: LocalTime,
        endTime: LocalTime,
        parkingZone: ParkingZone
    ): Double {
        // Calculate the duration of parking in minutes
        val durationInMinutes = Duration.between(startTime, endTime).
    toMinutes()

        // Round up the duration to the nearest 15 minutes
        val roundedDurationInMinutes = (ceil(durationInMinutes / 15.0)
    * 15).toLong()

        // Find the correct parking period based on the time
        var totalCost = 0.0

        for (period in parkingZone.periods) {
            if (!startTime.isBefore(period.endTime) && !endTime.isAfter
    (period.startTime)) {
                // Calculate hours and round them up
                val hoursParked = (roundedDurationInMinutes / 60.0).
    toInt()
                totalCost += hoursParked * period.pricePerHour
            }
        }

```

```
        return totalCost
    }
}
```

Listing 4.15. Generalized parking calculator with multiple period failure

From test run we get multiple failing tests for `oneHourRegularParking`, `roundUpToFifteenMinutes` and `parkingInParkingZoneWithMultiplePeriods`. This seems to be indicating that the new period choosing logic does not work as intended and also breaking the previously added tests. Let's use the feedback loop and allow the LLM model to correct the functionality.

```
class ParkingCalculator {

    fun calculateParkingCost(
        startTime: LocalTime,
        endTime: LocalTime,
        parkingZone: ParkingZone
    ): Double {
        // Calculate the duration of parking in minutes
        val durationInMinutes = Duration.between(startTime, endTime).
    toMinutes()

        // Round up the duration to the nearest 15 minutes
        val roundedDurationInMinutes = (ceil(durationInMinutes / 15.0)
    * 15).toLong()

        var totalCost = 0.0
        var remainingDuration = roundedDurationInMinutes

        for (period in parkingZone.periods) {
            // Check if the start time and end time fall within the
    current period
            val periodStartTime = period.startTime
            val periodEndTime = period.endTime

            // Calculate overlap between the parking duration and the
    current period
            val overlapStart = maxOf(startTime, periodStartTime)
            val overlapEnd = minOf(endTime, periodEndTime)

            // If there is an overlap, calculate the cost
            if (!overlapStart.isAfter(overlapEnd)) {
                val overlapDurationInMinutes = Duration.between(
    overlapStart, overlapEnd).toMinutes()
                val roundedOverlapDurationInMinutes = (ceil(
    overlapDurationInMinutes / 15.0) * 15).toLong()
```

```

                // Calculate the cost for this period
                totalCost += (roundedOverlapDurationInMinutes / 60.0) *
    period.pricePerHour
            }
        }

        return totalCost
    }
}
```

Listing 4.16. Generalized parking calculator with multiple period corrected

This time we see all tests passing. Also analyzing the code between the last two generations (Listing 4.15 and Listing 4.16) we see the additional logic of calculating and using the overlap between parking and period. This way we make sure that only the part of the parking which is inside the specific period, gets correctly calculated.

4.3.2 Refactoring test cases

Next we will add a more realistic parking scenario where the start and end times are not from full hours (Listing 4.17). This way we will test the case of parking duration rounding and pricing between multiple periods.

```
@Test
fun parkingInParkingZoneWithMultiplePeriodsWithoutFullCross() {
    val parkingCalculator = ParkingCalculator()
    val parkingPeriod = ParkingZone(
        name = "Zone A",
        periods = listOf(
            ParkingPeriod(
                startTime = LocalTime.of(0, 0),
                endTime = LocalTime.of(12, 0),
                pricePerHour = 1.00,
            ),
            ParkingPeriod(
                startTime = LocalTime.of(12, 0),
                endTime = LocalTime.of(23, 59),
                pricePerHour = 2.00,
            )
        )
    )

    // Parking is calculated by 15 minute intervals. New period price
   is applied only if the intervals starts in the new period
```

```
    val result = parkingCalculator.calculateParkingCost(
        startTime = LocalTime.of(11, 39),
        endTime = LocalTime.of(12, 9),
        parkingZone = parkingPeriod,
    )
    assertEquals(0.50, result)
}
```

Listing 4.17. Generalized parking test cases for period triangulation

This however is quite a difficult scenario all together as it would need to handle the parking duration rounding and period change together. From the generated solution we get failure for `parkingInParkingZoneWithMultiplePeriodsWithoutFullCross`.

After three iterations of using the error in a feedback loop we still do not get to a working solution. This would indicate that the current prompt is too ambigiuous, difficult or incorrectly created for the model at hand to fully comprehend. Therefore let's try to add additional context from the last state of working solution (Listing 4.16). We generate the following functionality.

```
class ParkingCalculator {

    fun calculateParkingCost(
        startTime: LocalTime,
        endTime: LocalTime,
        parkingZone: ParkingZone
    ): Double {
        // Calculate the duration of parking in minutes
        val durationInMinutes = Duration.between(startTime, endTime).toMinutes()

        // Round up the duration to the nearest 15 minutes (if it is not already a multiple of 15)
        val roundedDurationInMinutes = ceil(durationInMinutes / 15.0) * 15.0

        var totalCost = 0.0
        var remainingDuration = roundedDurationInMinutes

        // Handle overlapping periods
        for (period in parkingZone.periods) {
            val periodStartTime = period.startTime
            val periodEndTime = period.endTime

            // Calculate overlap between the parking duration and the current period
```

```
            val overlapStart = maxOf(startTime, periodStartTime)
            val overlapEnd = minOf(endTime, periodEndTime)

            // If there is an overlap, calculate the cost
            if (!overlapStart.isAfter(overlapEnd)) {
                val overlapDurationInMinutes = Duration.between(overlapStart, overlapEnd).toMinutes()

                // Round the overlap duration to the nearest 15 minutes (to handle fractional time)
                val roundedOverlapDurationInMinutes = ceil(overlapDurationInMinutes / 15.0) * 15.0

                // Calculate the cost for this period
                totalCost += (roundedOverlapDurationInMinutes / 60.0) * period.pricePerHour
            }
        }

        return totalCost
    }
}
```

Listing 4.18. Generalized parking failed generation

Test cases still fail. And even the feedback loop technique fails after three iterations. Looking at the last generated solution (Listing 4.18) we can see that the function complexity is quite high with having to both round the duration of the parking, handle period switch and interval creation. To simplify the logic we could abstract the creation of the intervals out from the calculator and have a separate entity.

Let's create a test suit for `ParkingSession` class and define the interval creation to the class.

```
@Test
fun getParkingIntervalsForNonRegularParkingLength() {
    val parkingSession = ParkingSession("21:00", "21:16")
    // 16 minute parking will create 2 intervals of 15 minutes lasting after the specified end time
    val parkingIntervals = parkingSession.getParkingIntervals()
    // Expected output: 2 intervals of 15 minutes. First interval: 21:00 - 21:15, Second interval: 21:15 - 21:30
    val expectedParkingIntervals = listOf(
        ParkingInterval("21:00", "21:15"),
        ParkingInterval("21:15", "21:30"),
    )
```

```
    assertEquals(expectedParkingIntervals, parkingIntervals)
}
```

Listing 4.19. Generalized parking test cases for parking session interval abstraction

In the test we specify the function of `getParkingIntervals` to handle the creation of the parking intervals (Listing 4.19). The idea behind this is if we can clearly define the creations of the intervals, `ParkingCalcuator` has a lower complexity to handle. We achieve the following `ParkingSession` class (Listing 4.20).

```
class ParkingSession(start: String, end: String) {
    private val startTime: LocalTime = LocalTime.parse(start, DateTimeFormatter.ofPattern("HH:mm"))
    private val endTime: LocalTime = LocalTime.parse(end, DateTimeFormatter.ofPattern("HH:mm"))

    fun getParkingIntervals(): List<ParkingInterval> {
        val intervals = mutableListOf<ParkingInterval>()
        var currentStartTime = startTime

        while (currentStartTime < endTime) {
            val currentEndTime = currentStartTime.plusMinutes(15)
            intervals.add(ParkingInterval(
                start = currentStartTime.format(DateTimeFormatter.ofPattern("HH:mm")),
                end = currentEndTime.format(DateTimeFormatter.ofPattern("HH:mm"))
            ))
            currentStartTime = currentEndTime
        }

        // Ensure intervals go beyond the end time
        if (currentStartTime < endTime) {
            val currentEndTime = currentStartTime.plusMinutes(15)
            intervals.add(ParkingInterval(
                start = currentStartTime.format(DateTimeFormatter.ofPattern("HH:mm")),
                end = currentEndTime.format(DateTimeFormatter.ofPattern("HH:mm"))
            ))
        }

        return intervals
    }
}
```

Listing 4.20. Generalized parking session interval abstraction

Going back to the `ParkingCalculator`, we first have to modify the existing tests to also accommodate to the new `ParkingSession` class.

```
@Test
fun parkingInParkingZoneWithMultiplePeriodsWithoutFullCross() {
    val parkingCalculator = ParkingCalculator()
    val parkingPeriod = ParkingZone(
        name = "Zone A",
        periods = listOf(
            ParkingPeriod(
                startTime = LocalTime.of(0, 0),
                endTime = LocalTime.of(12, 0),
                pricePerHour = 1.00,
            ),
            ParkingPeriod(
                startTime = LocalTime.of(12, 0),
                endTime = LocalTime.of(23, 59),
                pricePerHour = 2.00,
            )
        )
    )

    // For parking from 11:50 to 12:05, we should pay for 15 minutes in
     the first period and 0 minutes in the second period. This is
    val result = parkingCalculator.calculateParkingCost(
        parkingSession = ParkingSession(
            start = "11:50",
            end = "12:05",
        ),
        parkingZone = parkingPeriod,
    )
    assertEquals(0.25, result)
}
```

Listing 4.21. Generalized parking test cases for pricing with parking intervals

Also, let's add the additional context of the working state. So with the tests will pass `ParkingSession` and `ParkingCalculator` previous working state. We generate the following.

```
class ParkingCalculator {

    fun calculateParkingCost(parkingSession: ParkingSession,
   parkingZone: ParkingZone): Double {
        val intervals = parkingSession.getParkingIntervals()
        var totalCost = 0.0

        for (interval in intervals) {
```

```
            val intervalStart = LocalTime.parse(interval.start)
            val intervalEnd = LocalTime.parse(interval.end)

            val applicablePeriod = parkingZone.periods.firstOrNull { period ->
                intervalStart >= period.startTime && intervalStart < period.endTime
            }

            if (applicablePeriod != null) {
                val minutesParked = ChronoUnit.MINUTES.between(intervalStart, intervalEnd).toDouble()
                val hourlyRate = applicablePeriod.pricePerHour
                totalCost += (minutesParked / 60) * hourlyRate
            }
        }

        return String.format("%.2f", totalCost).toDouble()
    }
}
```

Listing 4.22. Generalized parking calculator with parking intervals

With the latest generation we can pass all the tests. This verifies that the abstraction of the functionality for parking interval creation was a success, lowering the complexity of the `ParkingCalculator` functionality. Furthermore we can now reuse the `ParkingSession` class on future features to encapsulate the complexity in smaller classes.

4.3.3 Applying CoT for complex scenarios

Let's create additional logic for reservations. The original requirement for reservations was that the price had to be locked to the initial price of the parking. Looking at the last working implementation (Listing 4.22) we can see that the interval start is only used. In this we could theoretically reuse the interval creation to cover the functionality by allowing the reservation intervals to be created for the full parking time. To test this, first create a test suit.

```
@Test
fun getReservationParkingIntervals() {
    val parkingSession = ParkingSession("21:00", "22:30", reservation = true)
    // For reservation parking, the interval is the total length of the parking session
    val reservedParkingIntervals = parkingSession.getParkingIntervals()
    val expectedReservedParkingIntervals = listOf(
```

```
        ParkingInterval("21:00", "22:30"),
    )
    assertEquals(expectedReservedParkingIntervals, reservedParkingIntervals)
}
```

Listing 4.23. Generalized parking test cases for reservation

In the test suit we use a new parameter `reservation` for the `ParkingSession` creation. This is an important syntax use as otherwise there would be no definition of the potential flag for the code generation. We generate the following code.

```
class ParkingSession(
    private val start: String,
    private val end: String,
    private val reservation: Boolean = false
) {

    fun getParkingIntervals(): List<ParkingInterval> {
        if (reservation) {
            // If it's a reservation, return a single interval
            return listOf(ParkingInterval(start, end))
        }

        val intervals = mutableListOf<ParkingInterval>()
        var currentStartTime = start
        val endMinutes = timeToMinutes(end)

        // Generate intervals of 15 minutes
        while (timeToMinutes(currentStartTime) < endMinutes) {
            val nextEndTime = addMinutesToTime(currentStartTime, 15)
            intervals.add(ParkingInterval(currentStartTime, nextEndTime))
            currentStartTime = nextEndTime
        }

        return intervals
    }

    // Helper function to convert time "HH:mm" to total minutes since "00:00"
    private fun timeToMinutes(time: String): Int {
        val (hours, minutes) = time.split(":").map { it.toInt() }
        return hours * 60 + minutes
    }

    // Helper function to add minutes to a time "HH:mm" string
```

```
    private fun addMinutesToTime(time: String, minutesToAdd: Int):
    String {
        val totalMinutes = timeToMinutes(time) + minutesToAdd
        val hours = totalMinutes / 60
        val minutes = totalMinutes % 60
        return String.format("%02d:%02d", hours, minutes)
    }
}
```

Listing 4.24. Generalized parking session with reservation

With the tests passing we now look at the `ParkingCalculator` side to accommodate the price differences between reservation and non-reservation parking.

```
@Test
fun parkingInParkingZoneWithReservation() {
    val parkingCalculator = ParkingCalculator()
    val parkingPeriod = ParkingZone(
        name = "Zone A",
        periods = listOf(
            ParkingPeriod(
                startTime = LocalTime.of(0, 0),
                endTime = LocalTime.of(12, 0),
                pricePerHour = 1.00,
            ),
            ParkingPeriod(
                startTime = LocalTime.of(12, 0),
                endTime = LocalTime.of(23, 59),
                pricePerHour = 2.00,
            )
        )
    )

    // Reservation has a base price of 5 and locks the price to the
    current period for the full parking length
    val result = parkingCalculator.calculateParkingCost(
        parkingSession = ParkingSession(
            start = "10:00",
            end = "20:00",
            reservation = true,
        ),
        parkingZone = parkingPeriod,
    )
    assertEquals(15.0, result)
}
```

Listing 4.25. Generalized parking test cases for pricing with reservation

In the test we specify multiple periods to verify that the reservation actually locks in the price for the entire session and does not switch. We generate the following code.

```
class ParkingCalculator {

    fun calculateParkingCost(parkingSession: ParkingSession, parkingZone: ParkingZone): Double {
        val intervals = parkingSession.getParkingIntervals()
        var totalCost = 0.0

        // Handle the reservation case separately
        if (parkingSession.isReservation()) {
            val applicablePeriod = parkingZone.periods.firstOrNull {
                val sessionStartTime = LocalTime.parse(parkingSession.start)
                sessionStartTime >= it.startTime && sessionStartTime < it.endTime
            }
            // Use fixed price for reservation
            if (applicablePeriod != null) {
                val duration = ChronoUnit.MINUTES.between(LocalTime.parse(parkingSession.start), LocalTime.parse(parkingSession.end)).toDouble()
                totalCost = (duration / 60) * applicablePeriod.pricePerHour
                totalCost = if (totalCost < 5.00) 5.00 else totalCost // Ensure minimum reservation cost
            }
        } else {
            // For regular parking sessions
            for (interval in intervals) {
                val intervalStart = LocalTime.parse(interval.start)
                val intervalEnd = LocalTime.parse(interval.end)

                val applicablePeriod = parkingZone.periods.firstOrNull { period ->
                    intervalStart >= period.startTime && intervalStart < period.endTime
                }

                if (applicablePeriod != null) {
                    val minutesParked = ChronoUnit.MINUTES.between(intervalStart, intervalEnd).toDouble()
                    val hourlyRate = applicablePeriod.pricePerHour
                    totalCost += (minutesParked / 60) * hourlyRate
                }
            }
```

```
        }

        // Round the cost to two decimal places
        return String.format("%.2f", totalCost).toDouble()
    }
}
```

Listing 4.26. Generalized parking calculator with reservation

In the initial try we get a syntax error with the `start` and `end` being a private member of `ParkingSession` (Listing 4.24) class. After one iteration of feedback loop to the code generator we have a working running solution.

Running tests we still see a failure for lastly added test `parkingInParkingZoneWith-Reservation`. After three iterations of the feedback loop we still cannot get a passing test run, so let's take a look back at the test case `parkingInParkingZoneWithReservation` (Listing 4.25). The test case leaves the pricing calculation quite ambiguous. To help the LLM tool to interpret the logic, let's add the logic with CoT technique (Listing 4.27).

```
@Test
fun parkingInParkingZoneWithReservation() {
    val parkingCalculator = ParkingCalculator()
    val parkingPeriod = ParkingZone(
        name = "Zone A",
        periods = listOf(
            ParkingPeriod(
                startTime = LocalTime.of(0, 0),
                endTime = LocalTime.of(12, 0),
                pricePerHour = 1.00,
            ),
            ParkingPeriod(
                startTime = LocalTime.of(12, 0),
                endTime = LocalTime.of(23, 59),
                pricePerHour = 2.00,
            )
        )
    )

    // Reservation has a base price of 5 and locks the price to the
   current period for the full parking length
    val result = parkingCalculator.calculateParkingCost(
        parkingSession = ParkingSession(
            start = "10:00",
            end = "20:00",
            reservation = true,
        ),
```

```
        parkingZone = parkingPeriod,
    )
    // Price is calculated as follows: 5.0 as a reservation fee and 1.0
     * 10 = 10.0 for the parking duration with a total of 15.0
    assertEquals(15.0, result)
}
```

Listing 4.27. Generalized parking test cases for reservation with CoT

The most straight forward is to add the entire calculation flow in a step like manner to to the test as a comment. In this case we want to specify that there is a 5 unit reservation fee in addition to the normal parking fee. We generate the following functionality (Listing 4.28).

```
class ParkingCalculator {

    fun calculateParkingCost(parkingSession: ParkingSession,
   parkingZone: ParkingZone): Double {
        var totalCost = 0.0

        // If the session is a reservation, calculate it with the fixed
     reservation fee
        if (parkingSession.isReservation()) {
            totalCost = 5.0 // Fixed reservation fee
            val parkingPeriod = parkingZone.periods.first { it.
   startTime <= LocalTime.parse(parkingSession.getStartTime()) && it.
   endTime > LocalTime.parse(parkingSession.getStartTime()) }
            val minutesParked = ChronoUnit.MINUTES.between(LocalTime.
   parse(parkingSession.getStartTime()), LocalTime.parse(
   parkingSession.getEndTime())).toDouble()
            totalCost += (minutesParked / 60) * parkingPeriod.
   pricePerHour
            return String.format("%.2f", totalCost).toDouble()
        }

        // Otherwise, calculate for each 15-minute interval
        val intervals = parkingSession.getParkingIntervals()
        for (interval in intervals) {
            val intervalStart = LocalTime.parse(interval.start)
            val intervalEnd = LocalTime.parse(interval.end)

            // Find the applicable period for the interval
            val applicablePeriod = parkingZone.periods.firstOrNull {
   period ->
                intervalStart >= period.startTime && intervalStart <
   period.endTime
            }
```

```

            if (applicablePeriod != null) {
                val minutesParked = ChronoUnit.MINUTES.between(intervalStart, intervalEnd).toDouble()
                val hourlyRate = applicablePeriod.pricePerHour
                totalCost += (minutesParked / 60) * hourlyRate
            }
        }

        // Round total cost to two decimal places
        return String.format("%.2f", totalCost).toDouble()
    }
}
```

Listing 4.28. Generalized parking calculator final state

This time all the tests pass and we have filled all the initial requirements.

5. Evaluation

In this chapter, we examine the results for both the simple and the complex problem sets. Our evaluation focuses on the differences observed among the various prompt-creation techniques and on the added benefits delivered by the TDD-based iterative approach. Finally, we outline the limitations imposed on this study and propose ideas for future research.

5.1 Evaluating simple problems

With the simple problems, the evaluation focuses on two aspects. First, we change the input prompt creation methods in which the problem statements are presented to see how input design affects the generated output. Second, we test the model's ability to correct its own mistakes through a feedback loop. The analysis explores whether LLMs can fix their own code and whether allowing multiple attempts yields additional gains.

5.1.1 Differences in input prompt creation

There where 5 different methods compared in the creation of the input prompt.

1. Problem description.
2. Test cases.
3. Problem description plus 1 test case.
4. Problem description plus 2 test cases.
5. Problem description plus all test cases.

Starting with the MBPP benchmark, we observe a clear trend that providing more initial information leads to better generation results (Figure 1). The "Problem description" setting performs worst, with a success rate of 48.25%. The "Test cases" and the "Problem description plus 1 test case" perform similarly, achieving success rates of 63.24% and 64.27%, respectively. Finally "Problem description plus all test cases" yields the best outcome, with a success rate of 77.31%.

Looking back on the example problem statement (Listing 3.1) and its corresponding tests (Listing 3.2), we conclude that well-defined tests provide greater value for simple problems. Because the assertion-function names describe the main idea, the LLM can understand the

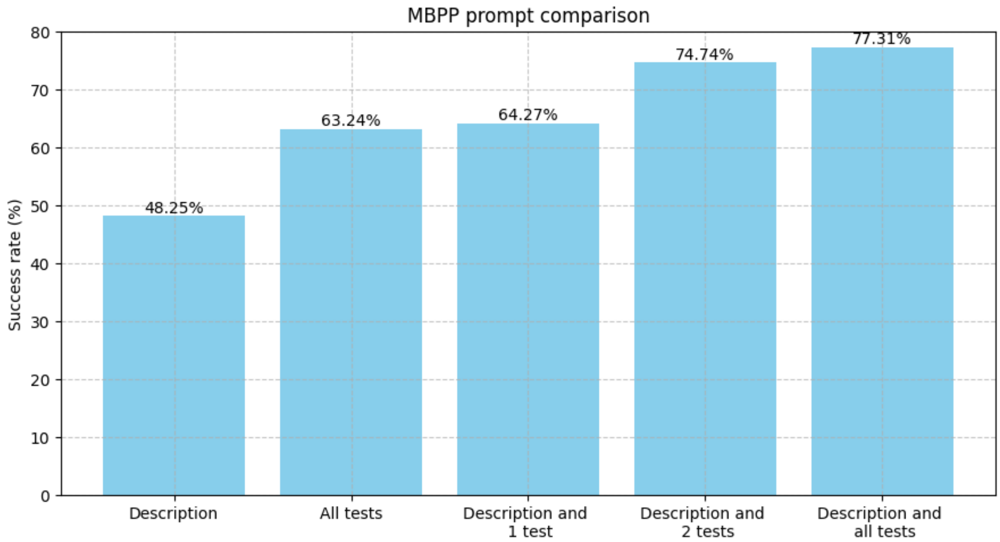

Figure 1. MBPP generation results with different methods

desired behavior directly from the test suite. Even so, combining the textual description with the full set of tests still outperforms using either source in isolation.

Furthermore, we observe a clear positive correlation between the number of tests supplied and generation accuracy. The success rate improves when moving from one to two tests, and again from two to three tests (Figure 1).

In the HumanEval benchmark the pattern changes. The best score comes from "Problem description" prompt: 75% of the solutions pass (Figure 2). Adding tests lower the success rate, as "Problem statement plus all test cases" have a success rate of 74.39%, "Problem description plus 2 test cases" is at 73.78%, and "Problem description plus 1 test case" at 72.56%. The "Test cases" method drops a lot lower at 40.85% success rate.

To explain the different outcome, first let's look back at the HumanEval benchmark problem statement example (Listing 3.3). The problem statement is much different from MBPP benchmark, defining a clear structure for the solution with example cases already added with descriptive comments. Therefor the "Problem description" method more corresponds to using the both the description and tests for generating the results, which also explains far better success rate from the tests only method.

Secondly, looking back at the test cases (Listing 3.4), the assertions do not even include the function name. This seems to be affecting the generation, not only making the "Test cases" method far worse, but also reducing the success rate of using description with tests compared to not using any tests at all. The results indicate that the differing function name is confusing the LLM and making the request more ambiguous.

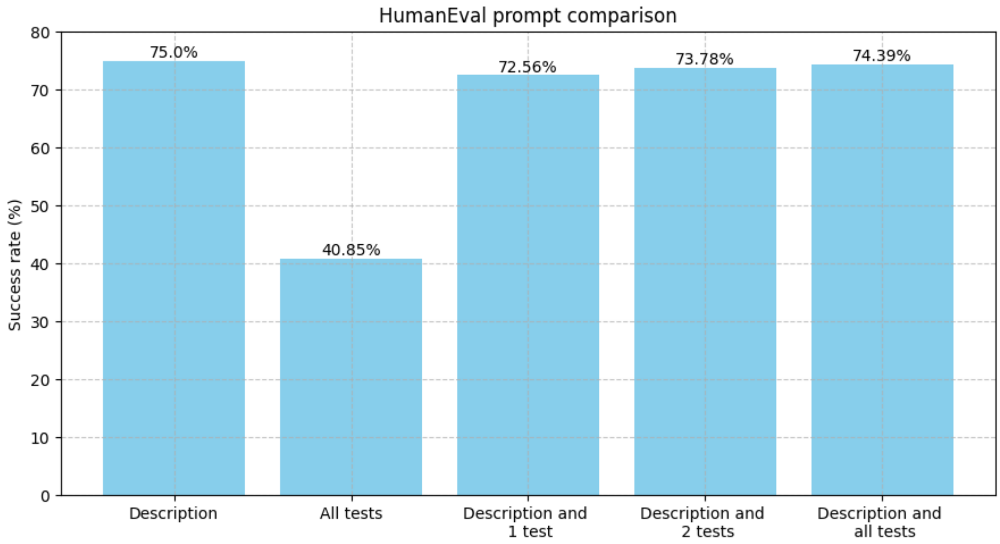

Figure 2. HumanEval generation results with different methods

From the results of both benchmark runs, it is clear that more contextual information produces a higher success rate for the generated solutions. This means providing descriptive information about the problem together with a set of corresponding tests. However, it is important not to include non-contextual information, as it can hurt the generation. As seen from the HumanEval example, poor naming of test cases can cause problems.

5.1.2 Differences in feedback mechanism iterations

The feedback mechanism was tested on results of the generations based on description and all tests, with the maximum of 10 iterations available.

For MBPP, the feedback iterations showed a gradual increase of the success rate of the solutions, with the original success rate of 77.31% raising all the way up to 81.62% (Figure 3). The biggest jump of the correctness was from the first iteration, where the success rate was increased by 1.95%. Starting from 7th iteration, the improvements stayed around 0.10% per iteration.

In HumanEval benchmark, the difference between iterations was much less gradual and had a greater effect on the success rate. The first iteration raised the success rate from 74.39% to 84.15%, second iteration to 84.76% and third to 86.59% (Figure 4). Starting from fourth iteration the success rate no longer changed.

Both benchmarks show clear improvements from using the test-run feedback as additional input for the failed runs. The success rate shows a 4–12% improvement for the same

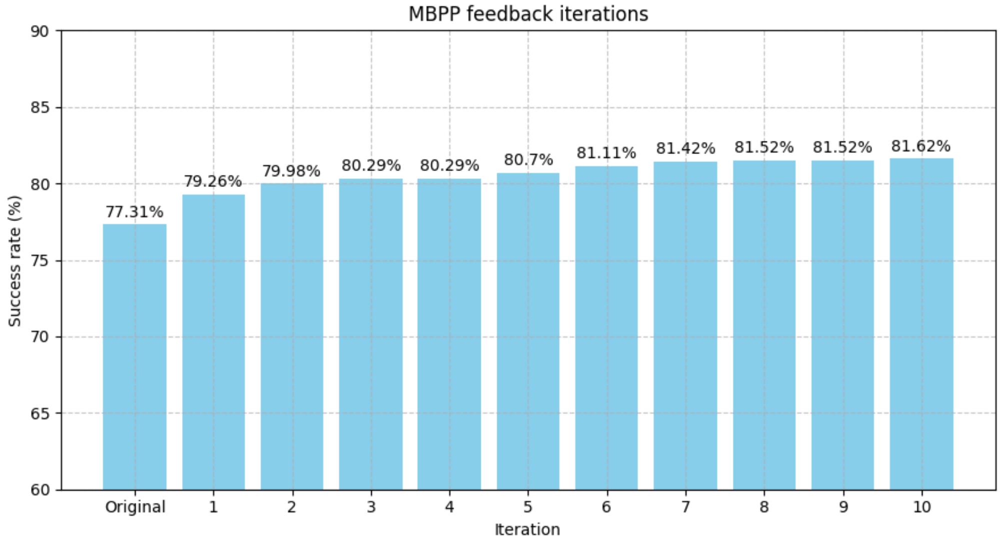

Figure 3. MBPP code generation with feedback iterations

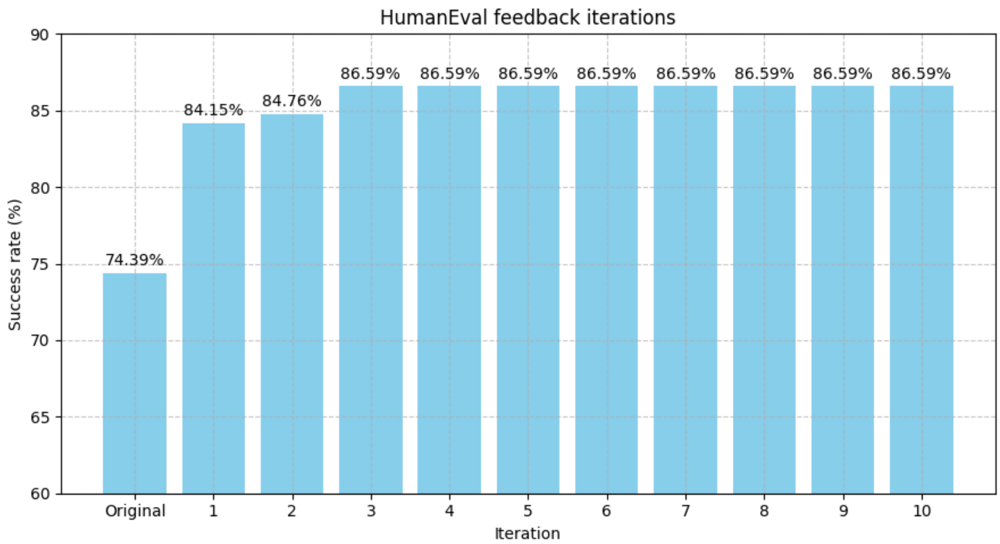

Figure 4. HumanEval code generation with feedback iterations

problem statement, with the first three iterations of the feedback session giving the majority of the improvement.

5.2 Evaluating complex problems

With the complex problems evaluation, we will first look at the iterative generation and define any learning points from the process. After that we will compare the success rate of different methods including the aforementioned iterative approach, to show the advantages and disadvantages of different methods.

5.2.1 Iterative generation summary

In this section we will focus on the iterative approach as described in the thesis (Chapter 4). The focus will be on what the approach adds to the generation process and what are the limitations met.

TDD methods

As the iterative approach relies on the Red–Green–Refactor cycle, all the enhancements are done in a step-like manner. This brings visibility to when the LLM is unable to generate a working solution as the tests start failing. And while sometimes allowing the LLM to correct its mistakes (Section 4.2.2), there might be a break point where the input needs to be refactored entirely (Section 4.3.2).

Furthermore, from the iterative approach it was visible that the test cases were always followed quite strictly, not providing additional logic or generalization if it was not requested. To still achieve generalization in the logic, triangulation showed good results in both examples (Section 4.2.1 and Section 4.3.1), where the complex logic was defined by using multiple test cases with varying parameters.

Finally, one of the outcomes of the method is to have a set of test cases covering most parts of the code. This is an excellent side-effect of the approach, as good test coverage is necessary for long-lasting functionality and any enhancements.

Context augmentation

It is not always necessary to add the previous states of code as context. If the input is definitive enough, the state could be generated from consistently again and again. The issue however raises with the more complex situations. With the raising complexity, the change of the LLM succeeding to generate code that covers all the test scenarios decreases.

This was also confirmed in the iterative approach, where the necessity to use context augmentation came later in the process, when a large portion of the logic was already implemented and a new feature was in works (Section 4.2.3).

Applying CoT

In some cases, a complex logic has to be precisely defined for it to be understood by the LLM. And while methods like triangulation work fine, in some cases it is not sensible to create tens of different tests to define all the possible logic edge cases. Instead it is much more convenient to describe the logic in a text. For TDD based generation a practical way is to use comments inside of the test cases (Section 4.3.3). This way the comments also remain with the code explaining the underlying logic that the initial implementation considered.

Refactoring test cases

In some cases, the code generation can still not provide a correct result even after using many other techniques provided in this chapter like CoT, context augmentation or modifying the tests to make the input less ambiguous. In these cases a more drastic measure has to be taken in the form of refactoring the test cases. This could include even the architectural side of the code, as it is possible to divide the problem between different classes or objects.

One example of this can be seen in the iterative approach (Section 4.3.2), where the logic part was defined for a separate class to direct the generation in a specific direction. For our case it worked greatly as the implementation was possible to be continued.

5.2.2 Code correctness comparison between methods

To compare the correctness of the different generation methods, a separate set of test suites was created to fully cover scenarios for all requirements. This ensured that the solutions were not solely tuned to the tests used during the generation process.

Parking with discount results

For the first parking problem, altogether 10 test cases covering the basic parking calculation and the discount were created. As the iterative approach used 10 generations in the workflow, the zero-shot approaches were also given in total 10 new generations. The tests showed a clear difference between the zero-shot and iterative approaches, with the TDD-based iterative approach being the only one to pass all the tests (Table 2).

Generation method	Passed tests (average)	Passed tests (median)	Total test count
Problem statement only	6.0	6.0	10
Tests only	6.8	6.5	10
Problem statement with tests	4.8	6.0	10
TDD based iterative	10	10	10

Table 2. Parking with discount test run results for different generation methods

Generation method	Passed tests (average)	Passed tests (median)	Total test count
Problem statement only	5.0	7.0	11
Tests only	5.7	6.0	11
Problem statement with tests	4.7	5.0	11
TDD based iterative	11	11	11

Table 3. Generalized parking with reservation test run results for different generation methods

Among the three zero-shot approaches, the success rate varied slightly. The best results were shown by the "Tests only" approach, with an average passed test count of 6.8, followed by the "Problem statement only" at 6.0 and finally by the combined solution of "Problem statement with tests" at 4.8 (Table 2). While there is a slight difference in the averages, the median of the results is even smaller, staying between 6.0-6.5 (Table 2). Therefor it is not reasonable to make many decisions on what zero-shot approach performed the best, as in the end, all of them failed.

With the "TDD based iterative" approach passing all tests, we can concur that there is additional value in the iterative process in itself. This is because the zero-shot approaches used the same set of tests in the generation, but failing at reproducing the results. That might be due to several reasons, but main one being in the constant evaluation and correction that comes from the iterative approach. Once a test fails, the necessary changes are done at the moment of failure, allowing to correct the exact problem at hand. With zero-shot approaches this is not possible, as there are can be many problems raising at the same time.

Generalized parking with reservation results

Continuing with the generalized parking problem we see a similar result. The zero-shot approaches again fail to achieve the same results to the TDD based iterative one. From the zero-shot approaches, the "Tests only" method achieves the best average passed tests count with 5.7, then "Problem statement only" at 5.0 and lastly the "Problem statement with tests" at 4.7 (Table 3).

The close average and median results again make it difficult to compare the zero-shot approaches to each other.

The "TDD based iterative" approach is again able to pass all the test cases with the same tests as an input. This again proves the method to be more efficient at generating results to complex problems compared to zero-shot approaches.

5.2.3 Code coverage

As TDD promotes the importance of tests, good code coverage naturally follows as a side effect of the practice. This also holds when using LLMs for code generation. As seen from the methods that use tests as input, the test coverage is quite high, covering in most cases 90–100 % of the generated code (Table 4 and 5). This indicates that the generation quite strictly follows the provided test cases on its own and does not introduce additional logic that might not be expected.

Generation method	Class (%)	Method (%)	Line (%)	Branch (%)
Problem statement only	-	-	-	-
Tests only	100	100	100	100
Problem statement with tests	100	100	96	66
TDD based iterative generation	100	100	97	92

Table 4. Test coverage of different generation methods

Generation method	Class (%)	Method (%)	Line (%)	Branch (%)
Problem statement only	-	-	-	-
Tests only	100	100	100	91
Problem statement with tests	100	100	100	81
TDD based iterative generation	100	100	100	85

Table 5. Test coverage of different generation methods

5.3 Limitations of the work

In this section we discuss about some limitations of the methods and approaches. The main limitations came from the choice of the generative model and the testing size for the iterative approach. As the methods used require a lot of manual work, a large quantitative study is not possible in this approach.

5.3.1 Limited model testing

For the purposes of showing differences in different prompt mechanisms, the examples were picked specifically for the model used, ChatGPT 4o-mini. The use of the model played a significant role in the complexity degree of the problem statements, as every model has a different capability in problem solving. Therefore, the same problem statements would not give much useful information for better or worse models. Better models could solve the issues with all generation methods the same, and worse models might not complete the solution at all.

Also, LLM research is right now moving at a very fast-paced rate. With the latest advancements coming in the use of reasoning models, which have changed the behavior of the models, this is untested in regard to the ways of working provided by the thesis.

5.3.2 Limited user research

The research provided a novel solution on how to use the existing limited models in a way that could solve issues otherwise not possible with the models. This, however, introduced a high level of dependency on the human side, which controls and manages the generation process. In this thesis, the impact of the way of working was not estimated by the value it could provide versus the time taken. Because of this, the true value of the method remains uncertain and would require user research to estimate the feasibility of the process in the real world.

5.4 Future works

To continue with the work there are multiple different directions that it could be taken as. From the research it has been verified that there are some aspects that do allow the generative model to achieve a better result, which are not yet applied in the available tools.

5.4.1 Feedback loop mechanism

In the quantitative analysis it was determined that the feedback gotten from test runs can help reduce issues with generations. This is however still quite a manual process, requiring to run the test cases and copying any explicit error into a prompt. If it would be possible to introduce a tool to allow an automatic run of such scenarios could help to achieve better results with the current models.

5.4.2 User research for iterative TDD practices

While the novel flow introduced in this thesis was proven to work on specific examples, it is still unknown how useful the tool would be for everyday use. To estimate the value an user research would be necessary where multiple engineers would need to evaluate the practicality of the method of work. The research should estimate both the practical value of the solution generation speed and ease of work to understand better its value for the engineers.

6. Summary

This thesis investigates the integration of Test-Driven Development (TDD) practices into Large Language Model (LLM)–based code generation workflows to enhance accuracy and reliability, especially for complex software tasks. We begun by evaluating prompt engineering strategies—textual descriptions, standalone test cases, and their combination—using established benchmarks (MBPP and HumanEval). This analysis demonstrated that combining natural-language descriptions with test cases yields more consistent generation accuracy than either method alone.

Building on these insights, we proposed an iterative, TDD-inspired approach in which LLMs generate code guided by a Red–Green–Refactor cycle. Develop takes the role of test-case creator and controls the generations, allowing an incremental refinement of the solution. Comparative experiments reveal that our iterative method consistently outperforms traditional zero-shot prompting, achieving near-complete test coverage with comparable generation costs.

A detailed case study on custom-designed parking scenarios further validates our approach in a real-world context. The iterative TDD workflow enables LLMs to handle complex requirements that zero-shot methods struggle to address effectively.

The key contributions of this work include a systematic evaluation of prompt-creation techniques that identifies best practices for combining natural-language descriptions with test cases, the design of a TDD-based iterative framework for effective LLM-based code generation, and empirical evidence from both benchmarks and complex case studies demonstrating the superior performance of the TDD-based methods over zero-shot prompting.

This thesis provides a clear instructions for software developers to use LLM capabilities more effectively, enabling the reliable automation of complex coding tasks. Future work may explore user experience research, multi-agent tools, and broader model research to generalize these techniques across different LLMs.

References

[1] *GitHub Copilot*. Accessed: 2024-12-18. URL: https://github.com/features/copilot.

[2] *OpenAi ChatGPT*. Accessed: 2024-12-18. URL: https://openai.com/index/chatgpt/.

[3] *Claude*. Accessed: 2024-12-18. URL: https://claude.ai/.

[4] Rasha Ahmad Husein, Hala Aburajouh, and Cagatay Catal. "Large language models for code completion: A systematic literature review". In: *Comput. Stand. Interfaces* 92 (2025), p. 103917. DOI: 10.1016/J.CSI.2024.103917. URL: https://doi.org/10.1016/j.csi.2024.103917.

[5] Md. Ashraful Islam, Mohammed Eunus Ali, and Md. Rizwan Parvez. "MapCoder: Multi-Agent Code Generation for Competitive Problem Solving". In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*. Ed. by Lun-Wei Ku, Andre Martins, and Vivek Srikumar. Association for Computational Linguistics, 2024, pp. 4912–4944. DOI: 10.18653/V1/2024.ACL-LONG.269. URL: https://doi.org/10.18653/v1/2024.acl-long.269.

[6] Dong Huang et al. "AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation". In: *CoRR* abs/2312.13010 (2023). DOI: 10.48550/ARXIV.2312.13010. arXiv: 2312.13010. URL: https://doi.org/10.48550/arXiv.2312.13010.

[7] Sarah Fakhoury et al. "Exploring the Effectiveness of LLM based Test-driven Interactive Code Generation: User Study and Empirical Evaluation". In: *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 390–391. DOI: 10.1145/3639478.3643525. URL: https://doi.org/10.1145/3639478.3643525.

[8] Zhao Tian and Junjie Chen. "Test-Case-Driven Programming Understanding in Large Language Models for Better Code Generation". In: *CoRR* abs/2309.16120 (2023). DOI: 10.48550/ARXIV.2309.16120. arXiv: 2309.16120. URL: https://doi.org/10.48550/arXiv.2309.16120.

[9] Moritz Mock, Jorge Melegati, and Barbara Russo. “Generative AI for Test Driven Development: Preliminary Results”. In: *CoRR* abs/2405.10849 (2024). DOI: `10.48550/ARXIV.2405.10849`. arXiv: `2405.10849`. URL: `https://doi.org/10.48550/arXiv.2405.10849`.

[10] Pietro Cassieri, Simone Romano, and Giuseppe Scanniello. “Generative Artificial Intelligence for Test-Driven Development: GAI4- TDD”. In: *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024, Rovaniemi, Finland, March 12-15, 2024*. IEEE, 2024, pp. 902–906. DOI: `10.1109/SANER60148.2024.00098`. URL: `https://doi.org/10.1109/SANER60148.2024.00098`.

[11] Kent L. Beck. *Test-driven Development - by example*. The Addison-Wesley signature series. Addison-Wesley, 2003. ISBN: 978-0-321-14653-3.

[12] Bertrand Meyer. *Agile!: The Good, the Hype and the Ugly*. Feb. 2014, pp. 1–170. ISBN: 978-3-319-05154-3. DOI: `10.1007/978-3-319-05155-0`.

[13] *MBPP Benchmark*. Accessed: 2024-12-18. URL: `https://paperswithcode.com/sota/code-generation-on-mbpp`.

[14] *HumanEval Benchmark*. Accessed: 2024-12-18. URL: `https://paperswithcode.com/dataset/humaneval`.

[15] Ashish Vaswani et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: `https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`.

[16] *OpenAi Codex*. Accessed: 2024-12-18. URL: `https://openai.com/index/openai-codex/`.

[17] *Elicit*. Accessed: 2024-12-18. URL: `https://elicit.com/`.

[18] *ResearchRabbit*. Accessed: 2024-12-18. URL: `https://www.researchrabbit.ai/`.

[19] *Tabnine*. Accessed: 2024-12-18. URL: `https://www.tabnine.com/`.

[20] *Notion AI*. Accessed: 2024-12-18. URL: `https://www.notion.com/product/ai`.

[21] *Grammarly*. Accessed: 2024-12-18. URL: `https://www.grammarly.com/`.

[22] Barbara Kitchenham and Pearl Brereton. “A systematic review of systematic review process research in software engineering”. In: *Information and Software Technology* 55.12 (2013), pp. 2049–2075. ISSN: 0950-5849. DOI: `https://doi.org/10.1016/j.infsof.2013.07.010`. URL: `https://www.sciencedirect.com/science/article/pii/S0950584913001560`.

[23] Jason Wei et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: 2201.11903 [cs.CL]. URL: https://arxiv.org/abs/2201.11903.

[24] Shreya Bhatia et al. "Unit Test Generation using Generative AI : A Comparative Performance Analysis of Autogeneration Tools". In: *LLM4CODE@ICSE*. 2024, pp. 54–61. DOI: 10.1145/3643795.3648396. URL: https://doi.org/10.1145/3643795.3648396.

[25] Shanthi Karpurapu et al. "Comprehensive Evaluation and Insights Into the Use of Large Language Models in the Automation of Behavior-Driven Development Acceptance Test Formulation". In: *IEEE Access* 12 (2024), pp. 58715–58721. DOI: 10.1109/ACCESS.2024.3391815.

[26] Max Schäfer et al. "An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation". In: *IEEE Trans. Software Eng.* 50.1 (2024), pp. 85–105. DOI: 10.1109/TSE.2023.3334955. URL: https://doi.org/10.1109/TSE.2023.3334955.

[27] Sanyogita Piya and Allison Sullivan. "LLM4TDD: Best Practices for Test Driven Development Using Large Language Models". In: *LLM4CODE@ICSE*. 2024, pp. 14–21. DOI: 10.1145/3643795.3648382. URL: https://doi.org/10.1145/3643795.3648382.

[28] Shuvendu K. Lahiri et al. "Interactive Code Generation via Test-Driven User-Intent Formalization". In: *CoRR* abs/2208.05950 (2022). DOI: 10.48550/ARXIV.2208.05950. arXiv: 2208.05950. URL: https://doi.org/10.48550/arXiv.2208.05950.

[29] Filomena Ferrucci, Carmine Gravino, and Luigi Lavazza. "Simple function points for effort estimation: a further assessment". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. SAC '16. Pisa, Italy: Association for Computing Machinery, 2016, pp. 1428–1433. ISBN: 9781450337397. DOI: 10.1145/2851613.2851779. URL: https://doi.org/10.1145/2851613.2851779.

Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis[1]

I Karmo Peetermann

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Evaluating Test-Driven Development for LLM code generation on complex problems”, supervised by Ants Torim
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

04.05.2025

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 - Parking with discount

Scenario:
A parking garage charges based on time tiers with discounts for off-peak hours and daily maximums.

- **Rates**:
  1. **0–2 hours**: \$3.00 flat rate (covers up to 2 hours).
  2. **2–6 hours**: \$1.50 per hour (in addition to the \$3.00 for the first 2 hours).
  3. **6+ hours**: A daily maximum rate of \$15.00 for any parking session that goes beyond 6 hours.
- **Off-Peak Discount**: If the majority of the parked time occurs between 8:00 PM and 6:00 AM, apply a 20% discount to the final cost.
  - *For example*, if someone parks from 10:00 PM to 2:00 AM (4 hours, mostly off-peak), they get the discount on their total bill.
- **Requirements**:
  1. **Input**:
     - Start time (in HH:MM 24-hour format).
     - End time (in HH:MM 24-hour format).
  2. **Output**: The total cost for the parking session, considering tiered rates and potential off-peak discount.
  3. **Constraints**:
     - The parking session cannot exceed 24 hours.
     - If the parking session crosses midnight, handle the date change accordingly (assume it is within the same 24-hour period).
     - The discount only applies if *more than 50%* of the parked time is within the off-peak window (20:00–06:00).

Appendix 3 - Generalized parking with reservation

Scenario:
A large city operates multiple parking zones, each with different base rates. Prices vary dynamically based on the time of day (e.g., morning rush, evening rush, overnight discount, etc.). Additionally, drivers can pre-book their parking space online at a fixed reservation fee, which will lock in a lower hourly rate.

Reservation System

- **Online Reservation Fee**: $5.00 flat fee.
- **Locked Hourly Rate**: Once a driver reserves, they pay the zone's rate *as of the reservation time block* for their entire stay—no time-of-day changes will apply mid-session.
- **No Reservation**: If a driver does **not** reserve, the rate for each hour (or partial hour) is determined by the time-of-day block in which that hour begins. If the parking session crosses multiple pricing blocks, they pay accordingly for each segment.

Time and Billing

1. **Billing Increment**: 15-minute increments, rounding up.
2. If the user **does not reserve** and crosses into a new time block (e.g., from Morning Rush to Midday) at the top of any hour, the rate for subsequent increments is recalculated based on the new time block's rules.
3. If the user **does reserve**, the rate remains constant throughout the stay, regardless of crossing different time blocks.

Requirements

1. **Input**:
   - Start time & end time (HH:MM format).
   - Whether the user made a reservation or not (if yes, they pay the reservation fee and lock the rate from the start time's pricing block).

- ParkingZone. Example format.

```
{
  "name": "Downtown",
  "periods": [
    { "startTime": "06:00", "endTime": "10:00", "pricePerHour": 6.00 },
    { "startTime": "10:00", "endTime": "16:00", "pricePerHour": 4.00 },

  ]
}
```

2. **Output**:

   - The final cost, factoring in:

     – Base zone rate.
     – Time-of-day dynamic pricing (for non-reserved sessions).
     – Reservation fee (if applicable).
     – Partial-hour billing increments.

3. **Constraints**:

   - **Rounding** to the next 15-minute increment applies at each segment if there's fractional usage.