**Enhancing Code Generation through**

**TDD and GenAI**

**Introduction**

**1.1 The Rise of LLMs in Code Generation**

Software development is changing fast nowadays. The modern software systems are quite complex. Tech companies want the development cycle to be faster and increase the quality of code. So for many years, people see automated code generation as a big goal. Since it can rapidly reduce repetitive workloads and improve overall productivity.

The rise of Large Language Models(LLMs) recently has made this goal reachable. Models like OpenAI’s GPT series, Codex (Chen et al., 2021), together with Google’s Gemini and PaLM, Meta’s CodeLlama, and DeepSeek-Coder (used in this project), have shown strong ability in code generation. They can take natural language as prompts and generate code in many different languages.

There are early models like CodeBERT (Feng et al., 2020). They were mainly designed to capture the unique features of programming languages. As a result, they learned how to build connections between natural language text and code. Later models were trained on billions of lines of code from GitHub. Because of this scale, they achieved strong performance on benchmarks and now support tools like GitHub Copilot (Chen et al., 2021). They are also being used for tasks such as code implementation, documentation, and software testing (Sarsa et al., 2022).

It is not always easy to access to the newest models. But still, it is clear that LLMs are transforming the way developers program.

**1.2 The Challenge: Correctness and Reliability of LLM-Generated Code**

Modern LLMs are fluent at code generation, but major problems like functional correctness and reliability still remain. They often produce code that looks correct (on the surface) and runs without syntax errors. However, it is still not guaranteed that the code really behaves as intended under all conditions. (Chen et al., 2021; Liu et al., 2023). Some of the key issues still persist.

First, natural language is the main way we instruct LLMs, but it is often not clear enough. Prompts often miss key details, like edge cases, error handling, performance limits and so on. LLMs might make assumptions or miss hidden needs, because they lack full understanding and cannot ask clear enough clarifying questions (Mu et al., 2023). This can lead to code that breaks in certain cases (Tian and Chen, 2024). It is still a main challenge to close the gap between high-level goals and low-level code.

Second, even code for simple tasks can hide small logic bugs, or something like off-by-one errors, wrong type handling, security holes, etc (Chen et al., 2021). It is especially hard to write complex algorithms or code that needs deep reasoning. It often needs lots of tests or careful review to find these issues.

The standard LLM process is feed-forward. The model writes code from the prompt, but it has no built-in way to check it against the requirements. Some work tries to fix code after generation using the run feedback(Gupta et al., 2020; Olausson et al., 2023; Chen et al., 2023; Zhang et al., 2023), but the baseline still skips this validation step. As a result, checking is left to the developer or later tests.

Third, it is hard to fully check functional correctness. Standard benchmarks often don’t have enough tests to catch hidden bugs (Liu et al., 2023). This can push up the scores and make the code seem more reliable than it really is. This makes people doubt the robustness. The models may not be reliable, even if they look strong on current benchmarks.

**Literature Review**

This section gives the background for the research in this thesis. It looks at three areas: First, the progress and challenges of Large Language Models (LLMs) in code generation. Second, the principles of Test-Driven Development (TDD). Third, the early work on combining testing methods like TDD with Artificial Intelligence (AI), especially LLMs. By reviewing this literature, I build the base for this project and point out the research gap it aims to fill.

**Large Language Models for Code Generation**

Automated code generation has changed a lot with the rise of LLMs. These models are usually based on the Transformer architecture. They are trained on very large datasets with both text and code. So, they can learn complex patterns and use context in useful ways. LLMs can now generate code that is clear and useful to the task. They are also used in other software engineering tasks, but code generation is one of the main applications.

Early work on code models saw code as ‘natural.’ This means code, like human language, has patterns that models can learn. Based on this idea, researchers built models designed for programming languages. One good example is CodeBERT (Feng et al., 2020).

It is a pre-trained model that learns from both natural language (NL) and programming language (PL) code samples. CodeBERT uses training tasks such as Masked Language Modeling (MLM) and Replaced Token Detection (RTD). It is trained on paired NL-PL data and on code-only data. The model showed strong results in tasks like code search and documentation generation. This work showed the value of training models to connect natural language and code. Building on this, research soon moved towards large LLMs trained on massive datasets.

Some early examples are GPT-Neo and GPT-J. They were trained on data like The Pile, which includes a lot of code from GitHub. These models already showed simple code generation skills. A major step came with OpenAI’s Codex (Chen et al., 2021). Codex is based on the GPT family but fine-tuned on billions of lines of public GitHub code. It showed strong results in generating Python code from docstrings and natural language. Codex became well known because it powered GitHub Copilot. At the same time, DeepMind’s AlphaCode showed that LLMs could also solve harder problems. It reached results close to human level in programming contests. This marked a turning point, as LLMs began to show both practical applications and competitive-level problem solving. After this, many other models appeared, such as CodeGen (Nijkamp et al., 2022), INCODER (Fried et al., 2022a), PolyCoder (Xu et al., 2022), CodeLlama, and the DeepSeek-Coder series used in this project. They all used large code datasets and different training setups, but the goal was the same: to make code generation more accurate and useful.

As these models grew stronger, a key challenge became how to evaluate their performance. Traditional language metrics such as BLEU were not enough, since two programs can look very different in code but still do the same thing (Chen et al., 2021). Because of this, researchers started using execution-based evaluation. Benchmarks like HumanEval and MBPP were introduced. They provide coding tasks where the output is tested by running unit tests on the generated code. The APPS benchmark (Hendrycks et al., 2021) expanded this idea with around 10,000 problems at different levels, similar to online coding contests. Another key step was the pass@k metric. It measures the chance that at least one of k code samples passes all tests. Unbiased estimators were also developed to make comparisons fair (Chen et al., 2021). These evaluation tools marked a big step forward, but important challenges remain.

Despite these advances, LLMs for code generation still face clear limits. They often struggle with tasks that need deeper reasoning, long chains of steps, or complex state management. Precise handling of variables and operations can also be difficult (Chen et al., 2021). Writing code that fully matches subtle or ambiguous natural language requests remains a challenge (Mu et al., 2023). Evaluation adds another problem. Studies have shown that common benchmarks like HumanEval may not include enough or diverse tests to check correctness well. When the HumanEval set was expanded with more unit tests, success rates for many top models dropped and even their rankings changed (Liu et al., 2023). This shows that stronger and more reliable benchmarks are needed. Access is another issue. Many of the most capable models, such as Codex or GPT-4, are only available through limited APIs. This makes it harder for researchers to study them in depth, adapt them, or compare them fairly. These challenges highlight that despite rapid progress, significant barriers remain for both practical use and academic study.

**Methodology**

I ran experiments with three prompting strategies to see how TDD principles can guide LLM code generation.

**Dataset and Tasks**

In this project, I used the HumanEval benchmark from OpenAI (Chen et al., 2021). This dataset is often used to test if LLMs can generate correct Python code. It includes 164 programming problems. Each problem gives a function signature and a short text description of what the function should do. The tasks include text understanding, writing basic algorithms, and solving small math problems. The difficulty is low to medium. It’s similar to an introductory programming course or simple coding interview.

Each HumanEval problem also provides a reference solution and human-written unit tests. In this project, I included both the problem description and the test cases in the prompt given to the model. The model then generated code based on this input.

All HumanEval problems are in Python, which fits the model I used. The focus of this project is on the process of code generation with different prompting methods (baseline vs. TDD-guided). It does not aim to build a full automated evaluation system. Other datasets like APPS(Hendrycks et al., 2021) are more complex, but HumanEval is simpler and more consistent. It is a pretty good starting point to see the effects of TDD guidance on code generation.

**Model and Setup**

In this project, I used chat model DeepSeek-V3-0324, called “deepseek-chat” for all my experiments. Then model could be accessed from the official API endpoint (<https://api.deepseek.com/v1/chat/completions>). This API followed the same format as OpenAI’s Chat Completions, so I could reuse OpenAI-style scripts with just very small changes.

Each request was sent in JSON format. With max\_tokens set to 1024 and *temperature* fixed at 0.0. This made sure that the outputs were consistent so easier to check. I chose to measure only pass@1, I didn’t sample multiple responses. Which means the model only produced a single solution for each task.

The scripts were written in Python 3.12.7, using standard libraries like *urllib*, *ssl*, and *json.* The *human\_eval* package was used to load tasks. These handled the API requests, retries, and output extraction. The experiments ran on my laptop, which was with Windows and WSL2 (Ubuntu 24.04.2 LTS, kernel 6.6.87.2). Since all computations were finished externally through the API with an API key, no GPU or other hardware was required.

For each HumanEval problem, my code created a prompt. Then sent it to the API, and collected the response. After the experiments started, the outputs were constantly shown in the terminal and also written into a JSONL result file. This setup was the same across all runs, so any differences in the results came only from the prompting strategies. I will describe next.

**Prompting Strategies**

I used three prompting strategies in this study. to compare how TDD principles affect the chosen model’s code generation.

First, Direct(Baseline). In this setting, I only gave the model the original HumanEval description. Each prompt contained just the function signature, and a short text describing the task. I didn’t add any extra hints or tests. To explain this, my goal was to see how good the model could solve problems under minimum input.

Second, manual TDD-guided. Here, I added human-written test cases from HumanEval. I set the prompt into three part: the problem description, the unit test, and a code block marker. It asked the model to write code that would pass the given tests, which basically followed a test-driven style. I used the tests directly from the dataset without any change.

Third, Auto TDD-guided. This strategy was a bit more complicated. I used two steps to achieve it. First, I asked the model to generate pytest-style tests for each HumanEval problem. Then, these tests were added to the original description. Later, the code prompted the model to write solutions that could pass them. It used the instruction: *“Implement the function so that these tests all pass.”* The purpose is to simulate a TDD cycle where tests are written before implementation.

Across all three strategies, the prompt format was kept quite simple. I used fenced code blocks(```python) to encourage the model to give clean outputs. In this way, the presence and source of test cases were the only difference among the strategies.

**Code Extraction and Saving**

After each API call, I extracted the Python code from the content field. At first, I looked for fence blocks starting with ```python, which kept only the code inside. If there were no Python block, I would check for a plain fenced block. If I could not find neither, I used the full text as a fallback. To make it ran directly, I also removed carriage returns, trimmed spaces, and cleaned the outputs.

For the auto TDD setting, I followed the same process twice. First, I extracted the model generated tests, then I added these tests to the original problem and asked the model to give code that would pass them. Again, I used the same method to extract the fenced solution. During execution, I displayed the results in the terminal and saved them in JSONL files. Each line contained the *task\_id* and the extracted completion. I used different files for each strategy. This kept results safe during runs, and made it easier for later comparison of the three strategies.

**Evaluation Metric (pass@1)**

I moved to the evaluation step once the outputs were gathered. Then I used the HumanEval pass@1 metric to check correctness for the code from each strategy (Chen et al., 2021). This metric checked if the model’s first output passed all unit tests for a problem. In practice, pass@1 shows the share of tasks solved correctly in one try. I chose it because real users usually expect the first response to work without sampling multiple outputs.

To compute pass@1, I used the official HumanEval evaluation script (*evaluate\_functional\_correctness*). After the results were stored in JSONL format, the script executed each completion in a sandbox. And it compared the outputs against the ground-truth tests. Each entry showed whether the completion passed, failed, or timed out in the output.

I then summed up the results to get a pass@1 score for each strategy. I didn’t evaluate pass@10 or pass@100, since I only generated one solution per task. Running more generations would need extra API calls, which cost me more and take way longer. By using only pass@1, the experiments stayed efficient. This still matched the goal of checking correctness.

Overall, pass@1 was a very easy and rather practical way to measure accuracy. It let me compare the three strategies, under the same conditions and without extra complexity. This balance made pass@1 a good choice for the project. With this, I compared the results across strategies.