**Enhancing Code Generation through**

**TDD and GenAI**

**Introduction**

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

Software development is changing rapidly. Modern systems are complex. Companies want faster development cycles and better code quality. Automated code generation has always been a big goal. It can reduce repetitive work and improve productivity.

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

Earlier models like CodeBERT (Feng et al., 2020) focused on the special features of programming languages. They learned links between text and code. Newer models are trained on very large datasets, often billions of lines of code from GitHub. These models perform well on benchmarks and power tools like GitHub Copilot (Chen et al., 2021). They are also used for many software tasks such as implementation, documentation, and even testing (Sarsa et al., 2022).

Access to the latest models is not always open. But it is very clear that LLMs are changing how developers program.

**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.

**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.