# Streamlining Software Development: An AI-Powered Code Generation Platform

Students Name: Abhinav AV (1), Johann Luke John (27), Kevin Tom Varghese (29), Thriambak S (58)

Group No: 6

Guide: Ms. Prathibha S Nair

# **Contents**

- Introduction
- Literature review
- Problem Statement
- Objective
- Methodology
- Results
- UML and Data Flow Diagram
- Conclusion
- Future Scope
- References

# Introduction

Advancements in artificial intelligence, particularly with LLMs like Claude, Codex,GPT-4 and Deepseek, have revolutionized how software is created. This project harnesses these models to create a web application capable of generating Python code from natural language prompts.

The platform offers an intuitive interface where users can specify their programming needs, generate code in real-time, and debug it immediately. Tailored to accommodate users of all skill levels, it simplifies software development by automating a huge majority of its various processes.

# Literature Review

No.	Title	Methodology	Results	Advantages and Disadvantages
1	Balancing Security and Correctness in Code Generation	<ul> <li>Evaluated five commercial LLMs (GPT-3.5, GPT-4, Claude Instant, Claude Opus, and Gemini 1.0).</li> <li>Designed prompts that intentionally generated security vulnerabilities based on Common Weakness Enumerations (CWEs).</li> <li>Applied different prompt strategies to improve security and assessed their impact on functionality.</li> <li>Generated 4,235 code samples and tested them using unit tests for correctness, security, and robustness.</li> </ul>	<ul> <li>Security-focused prompts significantly reduced vulnerabilities but often negatively impacted code correctness.</li> <li>GPT-4 produced the most secure and functional code, whereas Gemini 1.0 frequently rejected security-related prompts.</li> <li>Increasing temperature settings slightly improved security but had minimal effect on correctness.</li> <li>Some security measures introduced unnecessary complexity, leading to non-functional or difficult-to-debug code.</li> </ul>	Advantages  Enhanced Security: Prompt engineering significantly reduced the occurrence of CWEs in generated code.  Comparative Evaluation: Demonstrated the strengths and weaknesses of different commercial LLMs for secure code generation.  Practical Applications: Insights can be used to develop better prompt strategies for security-aware code generation.  Disadvantages  Reduced Correctness: Security improvements often came at the cost of program correctness.  Model Dependence: Different LLMs produced varying security and correctness levels, requiring careful model selection.  Overhead in Execution: More secure prompts led to increased response times and computational costs.

No.	Title	Methodology	Results	Advantages and Disadvantages
2	Code Generation using Machine Learning: A Systematic Review (2022)	<ul> <li>Search Approach: The study used the PRISMA framework to systematically review 37 papers from the IEEE Xplore and arXiv databases, applying strict inclusion and exclusion criteria.</li> <li>Application Categories: Papers were grouped into three paradigms: description-to-code, code-to-description, and code-to-code.</li> <li>Model Evaluation: Examined different machine learning (ML) models (e.g., RNNs, Transformers) and their performance on tasks like program synthesis, automatic program repair, and documentation generation.</li> <li>Data: Data sources included GitHub, StackOverflow, and manually created or synthetic datasets.</li> </ul>	<ul> <li>Performance Trends:         <ul> <li>Transformers generally outperformed RNNs for generating code and descriptions.</li> </ul> </li> <li>Applications: Tasks like code generation from natural language and documentation generation showed notable advancements.</li> </ul>	Advantages  Wide Applicability: ML models enable automatic software development tasks, enhancing productivity.  Adaptability: Models work on various paradigms (e.g., natural language to code).  Disadvantages  Data Limitations: Many datasets are mined automatically and lack quality, affecting model reliability.  Computational Costs: Training models like transformers demands significant computational resources.

No.	Title	Methodology	Results	Advantages and Disadvantages
3	Knowledge aware code generation using Large Language Models (2024)	<ul> <li>Data &amp; Knowledge Library: The authors create CodeF, a Python-based dataset (1,523 problems) from CodeForces, carefully split at September 2021 to avoid overlap with ChatGPT's training data. They also construct a Knowledge Library for 33 common algorithms and data structures, including short descriptions, pseudo-code, and step-by-step guides.</li> <li>Approach (KareCoder): The process has two main stages. First, a Prompt Engineering Stage: ChatGPT acts as a "prompt engineer," reading the problem plus the relevant knowledge to generate a problem-solving prompt. Second, a Coding Stage: ChatGPT uses that prompt to write the final code.</li> <li>Baselines &amp; Setup: They compare KareCoder with direct ChatGPT usage, Plan (Self-planning idea), SCOT (Structured Chain-of-Thought), and SCOT&amp;KareCoder. Pass@1, Pass@3, and Pass@5 are used as primary metrics.</li> </ul>	<ul> <li>On CodeF Post-2021: KareCoder notably boosts Pass@1 (first-try accuracy) compared to direct ChatGPT usage and other planning-based methods, demonstrating that injecting targeted algorithm/data-structure knowledge aids code generation on new tasks. However, ChatGPT alone can sometimes yield higher Pass@3 and Pass@5, since a single flawed intermediate prompt in KareCoder can affect all subsequent code attempts.</li> <li>On APPS (500 Problems): ChatGPT's performance remains strong, likely due to overlap with its training data. Even so, KareCoder often equals or exceeds other structured-prompt baselines, underscoring the value of integrating a Knowledge Library for tasks that ChatGPT has not seen.</li> </ul>	<ul> <li>Enhanced         Performance on         Unseen Problems:         Using algorithm         /data-structure         knowledge helps         solve new problems         ChatGPT has not         seen in training.</li> <li>Modularity: The         Knowledge Library         is easily updated;         the approach can         adapt to other         algorithm domains.</li> <li>Strong Pass@1         Gains: The method         particularly boosts         the first successful         generation rate         (Pass@1).</li> </ul>

No.	Title	Methodology	Results	Advantages and Disadvantages
4	Low-Cost Language Models: Survey and Performance Evaluation on Python Code Generation (Science Direct 2024)	<ul> <li>Dataset Creation:         <ul> <li>Introduced a new dataset of</li> <li>60 Python programming</li> <li>problems with varying</li> <li>difficulty levels.</li> </ul> </li> <li>Chain-of-Thought         <ul> <li>Prompting: Designed a</li> <li>prompt strategy to guide</li> <li>models in generating Python</li> <li>code.</li> </ul> </li> </ul>	Low-Cost Models: Some     CPU-friendly models (e.g.,     Mistral-7B, Llama-3.1, Phi-2)     achieved competitive results     compared to advanced models     like ChatGPT-3.5 and Gemini.      Quantization Impact: Quantized     models (e.g., 4-bit Mistral, 5-bit     Phi-2) performed well, with     minimal performance drop     compared to non-quantized     versions.	<ul> <li>Advantages</li> <li>Accessibility: Low-cost models enable Python code generation on standard CPUs, reducing resource requirements.</li> <li>Competitive Performance: Some models (e.g., Mistral-7B, Llama-3.1, Phi-2) rival advanced models in code generation tasks.</li> <li>Dataset Contribution: New dataset extends benchmarks, enabling deeper evaluation of model capabilities.</li> </ul>
		<ul> <li>Evaluation: Conducted semi-manual evaluation using GPT-3.5-Turbo for correctness assessment, validated with test cases.</li> <li>Comparison: Tested low-cost CPU-friendly models (e.g., Mistral, Llama, Phi) against advanced models (e.g., ChatGPT, Gemini) on HumanEval and EvalPlus</li> </ul>	<ul> <li>Dataset Performance: Low-cost models excelled in generating correct code but struggled with output format adherence.</li> <li>Inference Time: CPU-friendly models demonstrated efficient inference times, making them practical for standard machines.</li> </ul>	Disadvantages     Output Format Issues: Models like Dolphin-2.6-Mistral and Phi-2 struggled with adhering to specific output formats despite generating correct code.     Limited Scope: Focused on Python; underrepresented languages and paradigms were not evaluated.     Rapid Model Evolution: New models and updates may outpace the study's findings.

No.	Title	Methodology	Results	Advantages and Disadvantages
5	Performance Analysis of ChatGPT in Solving Programming Problems [IEEE 2023]	<ul> <li>Measured ChatGPTs ability to solve LeetCode problems of varying difficulty.</li> <li>Assessed ability to fix errors when provided with feedback.</li> <li>Analyzed computational performance in terms of execution time and memory usage.</li> </ul>	<ul> <li>71.875% of performance was solved in the first attempt.</li> <li>Only 36.5% of errors were corrected after feedback.</li> <li>Strong in structured problems but weak in Dynamic Programming and Greedy Algorithms.</li> </ul>	Advantages  Strong in structured problems.  High initial success rate for easier problems.  Disadvantages  Struggles with debugging and feedback.  Poor performance in complex algorithmic domains.

No.	Title	Methodology	Results	Advantages and Disadvantages
6	Methodology for Code Synthesis Evaluation of LLMs (ChatGPT vs Copilot) [IEEE 2024]	<ul> <li>Proposes a 4-step evaluation process:</li> <li>Create balanced prompts,</li> <li>Test for correctness by automated tests,</li> <li>Evaluate technical merit via static analysis</li> <li>Use human evaluation on code readability and acceptability.</li> <li>Demonstrated on 25 benchmark programming exercises in Java and C++.</li> </ul>	<ul> <li>ChatGPT solved 17 problems correctly against Copilot's 13.</li> <li>Both generated code with code smells identified by static analysis but ChatGPT generally performed better by human evaluators.</li> </ul>	<ul> <li>Advantages</li> <li>Comprehensive and generalizable framework.</li> <li>ChatGPT performed consistently better on many facets.</li> <li>Disadvantages</li> <li>Time-consuming because of human assessment.</li> <li>Task and language restricted diversity (Java, C++ only).</li> </ul>

No.	Title	Methodology	Results	Advantages and Disadvantages
7	No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT [IEEE 2024]	<ul> <li>Performs vast empirical testing using 728         LeetCode algorithmic problems and 18         CWE-founded security code bug vulnerabilities.</li> <li>Cross-checks correctness of code, complexity, security, and multi-round fixability by ChatGPT (GPT-3.5).</li> </ul>	<ul> <li>ChatGPT scored a 48.14% increase in pre-2021 vs post-2021 problem acceptance rate.</li> <li>Multi-round conversation enabled correcting 89% of dangerous code, but at times at a higher complexity level.</li> <li>Non-determinism of ChatGPT answers led to fluctuations being endured.</li> </ul>	Advantages  Demonstrates competency in familiar problem areas for it.  Multi-round discussion is suitable for repairing security problems.  Disadvantages  Poor performance of new tasks.  Repairing makes things complicated; non-determinism affects consistency.

No.	Title	Methodology	Results	Advantages and Disadvantages
8	RefleXGen: The Unexamined Code is Not Worth Using[IEEE 2025]	<ul> <li>Introduces RefleXGen, a self-reflective, Retrieval-Augmented Generation (RAG) framework.</li> <li>It works in three stages:         <ul> <li>(1) Generate code,</li> <li>(2) Reflect using a knowledge base and detect issues,</li> </ul> </li> <li>(3) Iteratively refine and integrate feedback into the KB for future tasks. Doesn't require fine-tuning the model.</li> </ul>	• Demonstrated security improvement across LLMs: +13.6% with GPT-3.5 Turbo, +6.7% with GPT-4o, +4.5% with CodeQwen, and +5.8% with Gemini. Continuous improvement via reflection loop.	Advantages  Effective in increasing security without retraining.  Lightweight and adaptable.  Disadvantages  Focuses mainly on security, not correctness or efficiency.  Reflection mechanism might increase latency.

# **Problem Statement**

Building projects from scratch is time-consuming and complex, especially for beginners. This project aims to develop an AI-powered platform that generates, executes and debugs Python code based on natural language prompts, simplifying the development process.

# Objective

#### The project's key objectives are:

- To develop a platform that generates code based on user prompts.
- To enable real-time error detection.
- To create an interactive environment for code generation, debugging, report generation and test case generation.

# Methodology

The Methodology for the project can be divided into 4 parts:

- 1. Requirement Gathering and Research
- 2. System Design and Architecture
- 3. Backend Development
- 4. Frontend Development

# 1. Requirement Gathering and Research

**Objective**: Evaluate Al models for automated code generation, debugging, and execution.

Outcome: Selected models based on accuracy, speed, and Python compatibility for backend integration.

#### Process:

- Tested publicly available LLMs via Together AI.
- Analyzed model performance with user prompts.
- Generated requirement docs to define software needs.

### Al Models Tested

- 1. Llama 3.3-70B-Instruct Turbo Free
- *Info*: A high-performance, FP8-quantized version of Meta's Llama 3.3 70B. Optimized for fast inference with a 128K token context window. Excels in multilingual dialogue and instruction-following tasks.
- Use: Generates accurate requirement docs and code with reduced latency.

#### 2. DeepSeek V3

- *Info*: A 671B parameter Mixture-of-Experts (MoE) model with 37B active parameters per token. Trained on 14.8T tokens, supports 128K context. Strong in reasoning, math, and coding.
- Use: Ideal for complex code generation and synthetic data tasks.

### continued...

#### 3. Gemma-2 Instruct 27B

- Info: Google's lightweight, open-source model with 27B parameters. Multimodal (text/image), 128K context, excels in reasoning and chat tasks.
- Use: Efficient for smaller-scale code generation with high accuracy.

#### 4. Qwen 2.5 72B

- Info: Alibaba's 72B parameter model from the Qwen series. Enhanced for code generation, reasoning, and math. Supports 128K context and multilingual tasks.
- Use: Powers robust code and test case generation (e.g., your script's generate\_code).

# 2. System Design and Architecture

**Objective**: Design a scalable system for automated code generation and documentation. **Outcome**: A modular system ready for web integration, supporting full-stack automation.

#### Features:

- Real-time code generation and execution.
- Automated error detection and correction.
- Structured documentation for requirements, specs, code, and tests.
- Python-centric design with extensible architecture.

### continued...

#### **Key Functionalities:**

- 1. **Prompt Processing**: Converts user input into structured requirements (generate req doc).
- 2. **Software Specification**: Defines tools and algorithms for implementation (generate\_software\_doc).
- 3. **Code Generation**: Produces Python code using LLMs (generate\_code).
- 4. **Self-Correcting Debugging**: Iteratively fixes errors with a validation loop (auto\_generate\_code).
- 5. **Test Case Creation**: Generates test cases to validate code (generate test cases).
- 6. **Execution**: Runs code and captures output/errors (execute\_code).
- 7. **Documentation**: Formats outputs into professional documents with headers, footers, and timestamps (create\_report).

# 3. Backend Development

**Objective**: Build a robust backend to support the web application's pipeline.

**Outcome**: A functional backend powering the pipeline, ready for frontend integration.

#### 1. Professional Interface (Chat-Based):

- Backend: Handles prompt submission (POST to /generate), processes it with LLMs, and returns responses to the chat box.
- Script Tie-In: Uses generate\_req\_doc for initial requirement generation.

### continued...

#### 2. Requirements Page:

- Backend: Generates requirement docs from user prompts and suggestions, saves as Requirements doc, and serves for download.
- Script Tie-In: generate\_req\_doc.

#### 3. Software Specification Page:

- Backend: Creates specs based on requirements, updates with feedback, and outputs Software Spec doc
- Script Tie-In: generate\_software\_doc.

#### 4. Code Generation Page:

- Backend: Generates Python code with framework options (e.g., Flask/Django), supports real-time preview and download (Code.py).
- Script Tie-In: generate\_code, auto\_generate\_code...

### continued..

#### 5. Test Cases Page:

- Backend: Produces test cases from generated code, updates with prompts, and saves as Test\_Cases.py.
- Script Tie-In: generate\_test\_cases.

# 4. Frontend Development

This website's frontend is built using a static, document-centric approach that emphasizes accessibility, clarity, and simplicity. Developed entirely with HTML and minimal use of JavaScript or external libraries, it is lightweight, easy to deploy, and highly portable. Each page—Project Overview, Requirements, SRS, Code, and Test Case Generation—is treated as a standalone document, allowing users to access well-structured content without complex navigation. The design follows a sequential, linear layout using standard HTML elements like headings, paragraphs, lists, and tables.

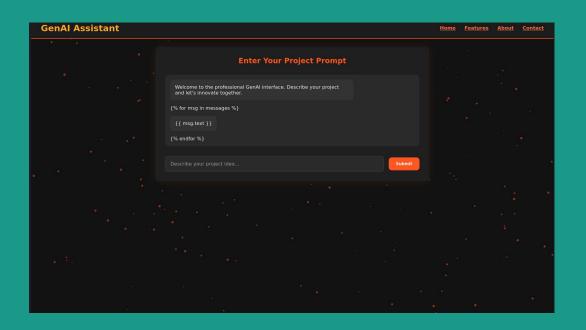
Each section serves a distinct role in the software development lifecycle. The **Project Overview** introduces the application's purpose, while the **Requirements** page outlines necessary system specifications. The **SRS** provides detailed functional and non-functional requirements in an organized format. The **Code** section showcases properly formatted code snippets for better readability. A key feature is the **Test Case Generation** module, where users can input functional prompts or code to generate test cases. It includes fields for prompt entry, customization settings, and a preview area displaying generated tests.

# Results

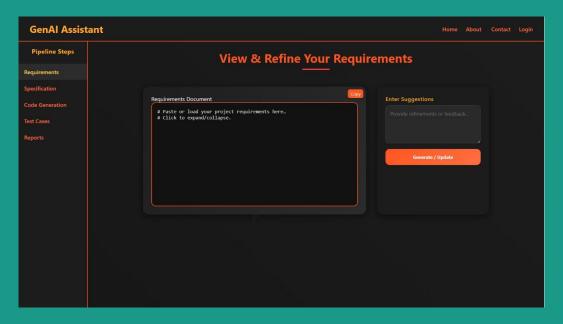
The goal of this project was to develop a platform that generates code based on user prompts, enabling real-time error detection. The platform was designed to provide an interactive environment for code generation, debugging, refinement, report generation, and test case creation, offering a seamless user experience across multiple interfaces.

#### 1. Receiving the prompt from the user

The user interface presents a intuitive dialog box designed to collect detailed input for code generation. At the top, a welcoming message invites users to describe their project idea. Beneath this, a dynamic message display area is reserved for ongoing conversational context, helping users keep track of their interactions. The primary input field allows users to enter a comprehensive prompt, including relevant project details and requirements. Once the prompt is ready, the user can click the **Submit** button to initiate the generation process.

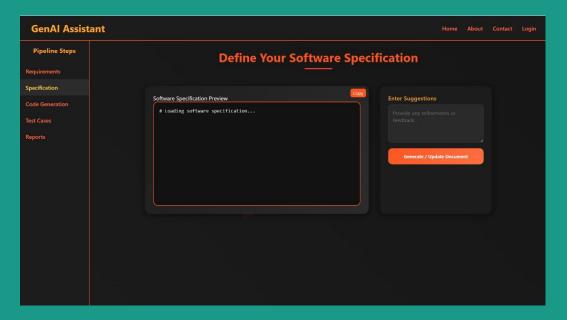


#### 2. Generation of Requirement document



The user is presented with a central editor where they can paste or write their project requirements in detail. This section supports a code-like format for structured input. On the right panel, users can provide specific suggestions, refinements, or feedback related to the requirements. Once the suggestions are entered, clicking the "Generate / Update" button processes the input and updates the requirements accordingly.

#### 3. Generation of SRS document



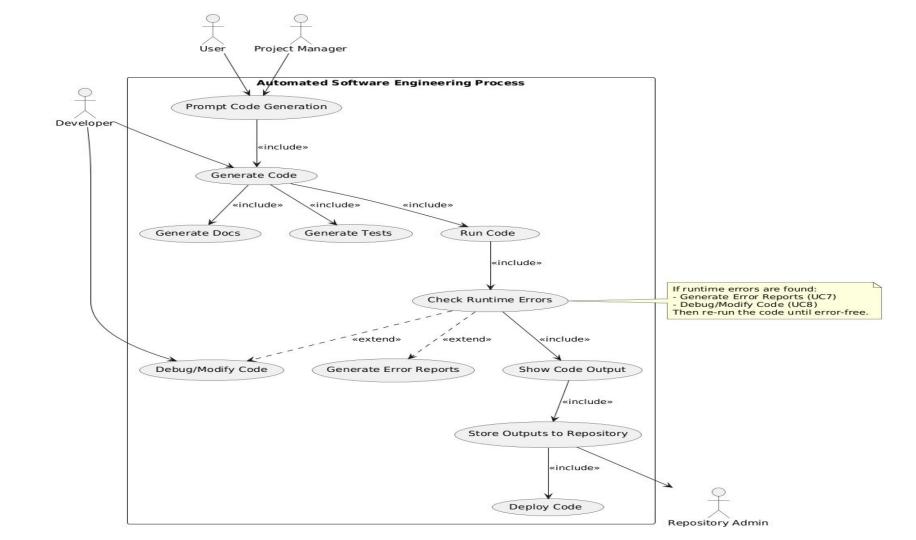
This interface allows users to define or refine their software specification. In the center, a preview panel displays the generated specification in a code-like format, currently showing a loading message. On the right, users can enter feedback or suggestions to improve the document. Once entered, clicking the "Generate / Update Document" button will regenerate the specification based on the input provided. The Copy button enables quick copying of the specification preview for external use or further editing.

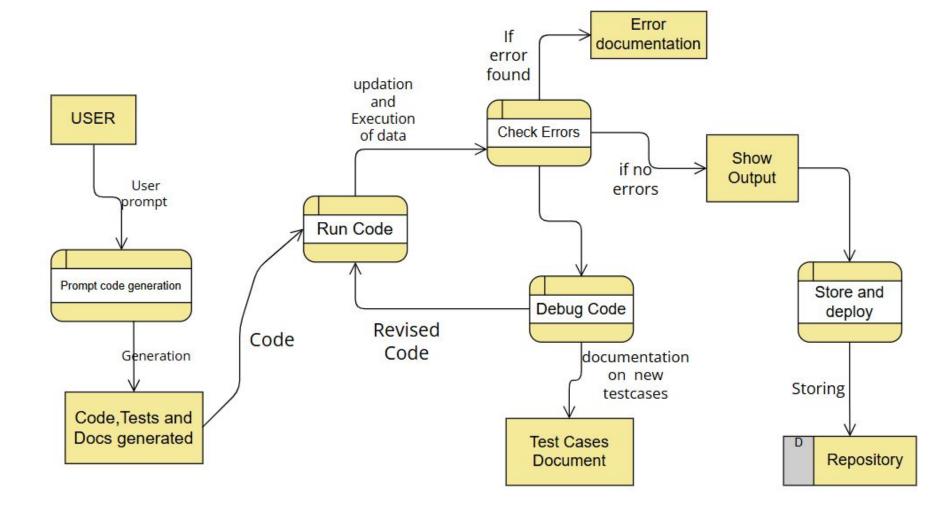
#### 4. Generation of Test Cases



This interface is designed for generating and previewing software test cases. The center panel displays the test cases in a formatted preview with a **Copy** button for quick copying. On the right, users can input additional prompts to customize the generation process. Two buttons—**Submit Settings** and **Explain Test Cases**—enable user-driven refinement. At the bottom, **Download Test Cases** allows saving the output, while **Proceed** moves the workflow forward based on the generated test cases.

# **UML and Data-flow diagram**





# Conclusion

This AI-powered platform will bridge the gap between coding expertise and accessibility by allowing users to generate code effortlessly.

By combining LLM-driven code generation with error debugging, report generation and test case generation, the platform offers a streamlined and intuitive solution for software development. This project has the potential to revolutionize how coding is taught, learned, and executed, ultimately making programming accessible to a broader audience.

# **Future Scope**

The project's future is to develop a single interface that can support code execution, thus enabling users to execute code within the application to accelerate the development process.

The interface will feature multi-language support to cater to the varied needs of developers and will include improved performance through profiling tools and automated tests to enhance execution speed and resource usage.

Additionally, the platform will incorporate real-time collaborative features, allowing multiple users to interact and work on the same codebase simultaneously. Finally, it will offer easy deployment options for the publication and management of applications across different environments.

### References

- [1] Balancing Security and Correctness in Code Generation

  Authors: Gavin S. Black, Bhaskar P. Rimal and Varghese Mathew Vaidyan
- [2] **RefleXGen: The Unexamined Code is Not Worth Using** *Authors*: Bin Wang, Hui Li, AoFan Liu, BoTao Yang, Ao Yang, YiLu Zhong, Weixiang Huang, Runhuai Huang, Weimin Zeng, Yanping Zhang
- [3] **Knowledge-Aware Code Generation with Large Language Models** *Authors*: Tao Huang, Zhihong Sun, Zhi Jin, Ge Li, Chen Lyu
- [4] Methodology for Code Synthesis Evaluation of LLMs (ChatGPT vs Copilot) Authors: Yu Feng, Rohan Padhye, Koushik Sen, George Necula

[5] No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT

*Authors*: Xudong Shang, Yashasvi Saini, Saikat Chakraborty, Ziyang Xu, Alessandra Gorla, Sarfraz Khurshid, Miltiadis Allamanis

[6] Low-Cost Language Models: Survey and Performance Evaluation on Python Code Generation

Authors: Jessica López Espejel, Mahaman Sanoussi Yahaya Alassan, Merieme Bouhandi, Walid Dahhane, El Hassane Ettifouri

- [7] **Performance Analysis of ChatGPT in Solving Programming Problems** *Authors*: Minda Li and Bhaskar Krishnamachari
- [8] Code Generation using Machine Learning: A Systematic Review *Authors*: Enrique Dehaerne, Bappaditya Dey, Stefan De Gendt and Wannes Meert

# Thank You!