Jaypee Institute of Information Technology, Noida

Department of Computer Science & Engineering and IT

Major Project Title: Qualitative analysis of LLM-generated and Human-written code

Enrol. No.	Name of Student
21103057	Avisha Goyal
21103073	Abiha Naqvi
21103081	Apeksha Jain

Course Name: Major Project 2

Program: B. Tech. CS&E/ B.Tech IT

7th Sem

2024 - 2025

(I)

TABLE OF CONTENTS

(II)

DECLARATION

We hereby declare that this submission is our own work and that, to the best of our knowledge and belief, it contains no material previously published or written by another person nor material which has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

Place: Jaypee Institute of Information Technology, Noida

Date:

Group Members:

NAME	ENROLLMENT NO.	SIGNATURE
AVISHA GOYAL	21103057	
ABIHA NAQVI	21103073	
APEKSHA JAIN	21103081	

(III)

CERTIFICATE

This is to certify that the work titled **"Qualitative analysis of LLM-generated and Human-written code"** submitted by **AVISHA GOYAL, ABIHA NAQVI, APEKSHA JAIN** in the partial fulfilment for the award of the degree of **B.Tech** of Jaypee Institute of Information Technology, Noida has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Signature of Supervisor: ____________________

Name of Supervisor: Dr. Ankita Verma

Designation: Assistant Professor

Date:

(IV)
ACKNOWLEDGEMENT

We would like to express our sincere gratitude to the following individuals and organisations who have contributed to the completion of this project. First and foremost, we would like to thank our supervisor Dr. Ankita Verma, for her guidance and support throughout the project.

We are grateful to Jaypee Institute of Information Technology, Noida for providing access to their resources, which were essential for developmental research. We would also like to extend our thanks to our coursemates and lab assistants who provided us with support & motivation throughout the project.

Finally, we would like to thank all the participants who took part in this study. Without their willingness to take part, this project would not have been possible. Thank you all for your contributions.

Date:

Group Members:

NAME	ENROLLMENT NO.	SIGNATURE
AVISHA GOYAL	21103057	
ABIHA NAQVI	21103073	
APEKSHA JAIN	21103081	

(V)

SUMMARY

This project evaluates Large Language Models (LLMs) by comparing AI-generated code with human-written code to assess quality, readability, performance, and contextual understanding. The study aims to highlight differences in efficiency, maintainability, and adherence to coding best practices, as well as how prompt specificity impacts AI output. A dual approach combines research and development: creating a dataset of human and AI-generated code samples, analysing them across several metrics, and developing an interactive dashboard. This dashboard, built with HTML, CSS, JavaScript, MongoDB, C++, and Python, will allow users to compare code quality and visualise findings, providing insights into the strengths and limitations of LLMs in software development.

Date:

Signature of Supervisor: ________________________

Group Members:

NAME	**ENROLLMENT NO.**	**SIGNATURE**
AVISHA GOYAL	21103057	
ABIHA NAQVI	21103073	
APEKSHA JAIN	21103081	

CHAPTER 1

INTRODUCTION

1.1 General Introduction:

Recent advances in artificial intelligence, particularly in large language models (LLMs) such as OpenAI GPT and Google Gemini, have brought significant potential for streamlining coding and problem-solving. These models can interpret natural language instructions and generate code, reducing development time and offering rapid solutions to complex problems. While the application of LLM in coding shows promise, it raises questions about its efficiency and reliability compared to traditional human-made code. Specifically, concerns about the quality, readability, efficiency, and context-appropriate use of AI-generated code have created a need for thorough evaluation to better understand how these technologies compare to human knowledge in real-world scenarios.

This project is designed to assess AI-generated code against human-written code across multiple criteria, using C++ and Python for practical language-specific comparisons. The analysis focuses on four key metrics that reflect quality and performance: cyclomatic complexity, lines of code, time complexity, and space complexity. Cyclomatic complexity provides insight into the logical complexity of code, while lines of code offer a direct measure of code brevity and maintainability. Time and space complexity, in turn, reflect the efficiency of the code in terms of the use of computing resources. By examining these parameters, the project aims to highlight specific areas where LLM-generated code conforms to or deviates from coding standards and best practices.

The project uses a two-tiered approach in terms of rapid difficulty. The first level contains relatively simple, straightforward questions that allow analysis of how LLMs handle simple coding tasks. The second level contains more challenging story-based tasks to assess the LLM's ability to generate code with a deeper understanding of context and logical flow. This approach allows us to observe how LLMs handle both basic and complex coding tasks, providing a different perspective on their abilities.

A key part of this project is the creation of an interactive dashboard using HTML, CSS, JavaScript, MongoDB, C++, and Python. This dashboard visually compares human and LLM code samples and displays detailed analysis results based on each metric. By enabling direct and practical comparisons, the dashboard offers users valuable insights into the strengths and limitations of AI in coding, allowing them to make informed decisions about integrating AI-generated code into their workflows. Overall, this project seeks to illuminate the role that LLMs can play in augmenting human effort in software development, ultimately increasing developer productivity while highlighting the irreplaceable value of human judgement in more complex coding tasks.

1.2 Problem Statement:

This project aims to evaluate the effectiveness of Large Language Models (LLMs), specifically GPT and Gemini, in generating code by comparing their output with human-written code in C++ and Python. By analysing parameters like cyclomatic complexity, lines of code, and space and time complexity, the project seeks to assess the performance, efficiency, and readability of AI-generated code. Additionally, the study explores the impact of prompt complexity across two levels to determine how LLMs manage tasks of varying difficulty.

1.3 Significance of Problem:

The increased use of LLMs in coding has created a need to more precisely understand their limitations and capabilities. Over-reliance on AI-generated code can cause inefficiencies or inaccuracies, impacting the maintainability and performance of the code. This project is significant because it offers developers, researchers, and industry professionals a clear understanding of the situations in which LLMs excel or falter, enabling more informed decisions about integrating AI into software development. The findings will help establish best practices for balancing AI assistance with human expertise.

1.4 Empirical Study:

The empirical component of this project involves the collection of a wide set of code samples, both human and artificial intelligence generated. For each sample, the study assesses key metric cyclomatic complexity, lines of code, and spatial and temporal complexity in two types of questions: Level 1 (simple challenges) and Level 2 (complex story-based challenges). The goal of this structured comparison is to identify specific areas where LLMs excel or need improvement, and how rapid complexity affects AI performance.

1.5 Brief Description of Solution Approach:

The goal of the project is to create a dataset of human and AI generated code samples in C++ and Python. Using level 1 and level 2 challenges, samples are analyzed for cyclomatic complexity, lines of code, time and space complexity. Developed using HTML, CSS, JavaScript, MongoDB, C++ and

Python, the interactive dashboard visualises analysis and allows users to compare the quality and performance of code from human and AI sources.

1.6 Comparison of existing approaches to the problem found

Existing studies of AI-generated code primarily focus on output correctness and syntactic quality, but lack a comprehensive analysis across multiple complexity metrics. While some approaches evaluate AI code in terms of basic functionality, this project dives deeper into examining readability and efficiency through cyclomatic complexity, lines of code, and resource usage. In addition, the two-level complexity of the rapid approach adds an original dimension to understanding how context affects AI performance, making this study more comprehensive than existing assessments.

CHAPTER 2

LITERATURE SURVEY

1) Comparing the Ideation Quality of Humans With Generative Artificial Intelligence

This paper explores the potential of generative artificial intelligence, specifically ChatGPT, in improving innovation processes by comparing ideas generated by artificial intelligence with ideas generated by human professionals. The study examines both types of ideas along several dimensions, including novelty, customer benefit, feasibility and overall quality, and provides a data-driven analysis of the role of AI in ideas. The findings highlight that while AI can serve as a powerful tool for generating diverse and unique ideas, human oversight and refinement is necessary to effectively drive innovation.

The research contributes theoretically by identifying the strengths and weaknesses of AI versus human imagination and highlights areas where human creativity and judgement remain vital. This analysis highlights that AI can function as a collaborative partner rather than a thought substitute, and provides a framework for systematically evaluating the potential of AI within specific tasks in innovation processes. The study's findings also offer practical guidance for businesses, suggesting that integrating AI into the early stages of ideation can lead to faster and more diverse idea generation while saving resources typically spent on traditional methods such as design thinking workshops.

However, the study acknowledges limitations, including its reliance on a single rater to judge an idea and its focus on the original idea without continuing through later stages of idea development. He recommends that future research include more raters to improve the reliability of the results, and suggests exploring the role of artificial intelligence in other stages of innovation, such as clustering and refining ideas. The paper also highlights ethical considerations for organisations using AI in innovation, including transparency, mitigating bias and privacy. In addition, it highlights the potential of artificial intelligence to complement human creativity by acting as a reading tool, accelerating the early stages of ideas and leaving more complex, evaluative tasks to human experts.

In conclusion, this study offers a fundamental approach to integrating AI into thought processes and argues for a balanced model of human-AI collaboration that could streamline idea generation and improve innovation outcomes. It calls for continued research to explore AI's capabilities in supporting innovation beyond ideation, potentially bridging gaps in the transition from idea generation to prototyping and evaluation.

2) AI vs. Human - Differentiation Analysis of Scientific Content Generation

The paper examines the capabilities of large language models (LLMs), namely OpenAI GPT-3 and ChatGPT, in generating scientific text and compares it to human-written content in informatics and biomedicine. It examines the coherence, consistency, and overall quality of AI-generated text by compiling a dataset of AI- and human-generated abstracts. These texts were evaluated using a feature-based detection model that focused on linguistic markers such as style, coherence, and logical flow, followed by a logistic regression model to assess structural and argumentative markers.

The results show that while AI-generated abstracts exhibit strong fluency and readability, they often lack depth and exhibit factual inaccuracies. AI text tends to oversimplify complex ideas and sometimes fails to align with rigorous scientific argumentation, highlighting a fundamental gap in understanding that can arise from a lack of domain-specific AI knowledge. In addition, AI-generated content can occasionally produce text that is too coherent – meaning it adheres closely to stylistic norms without introducing new or deep insights, resulting in a "surface-level" understanding of topics.

One of the key findings was that the machine learning detection model can distinguish between artificial intelligence and human-written abstracts with remarkable accuracy. Linguistic markers used to identify AI-generated text included consistent sentence structures, overuse of certain phrases, and lower variance in syntactic complexity—attributes that are markedly different from the variability often found in human-generated scientific text. Finally, the study highlights that while LLMs offer potential as aids to scholarly writing, they lack the depth and nuance that expert human authors contribute, particularly in complex areas requiring high factual accuracy and critical analysis.

The paper concludes with a call for responsible integration of LLMs into scientific research, highlighting both their utility in generating tentative proposals and the risks associated with potential misinformation should uncontrolled AI-generated content become prevalent. This work highlights the need for hybrid models that combine human supervision with the efficiency of artificial intelligence and advocates careful use to avoid factual errors and improve the quality of scientific communication.

3) Large Language Models for Code Analysis: Do LLMs Really Do Their Job?

The paper titled "Large Language Models for Code Analysis: Are LLMs Really Doing the Job?" examines the effectiveness of large language models (LLMs) such as GPT-4, Copilot, and CodeT5 in code analysis tasks, including summarization, error detection, and code completion. The study rigorously compares these models with conventional code analysis tools and evaluates their accuracy, consistency, and applicability in real programming environments. He found that while LLMs excel at specific tasks, such as generating code summaries and offering suggestions for completion, they face challenges with complex reasoning, particularly in detecting small errors and making deeper analytical conclusions.

The researchers highlight LLM's ability to generate human-readable summaries that can help developers quickly understand code, but note significant limitations. For example, models often lack the reliability needed to detect critical errors, where even small inaccuracies can lead to big problems in a production environment. Traditional program analysis tools still outperform LLM in these areas because they are more consistent and tailored to specific tasks within the software development lifecycle.

The paper concludes that, while promising, LLMs are not yet robust enough to replace established code analysis techniques. Instead, they are best suited as complementary tools that can improve productivity in non-critical aspects of understanding and designing code while developers continue to rely on conventional methods for tasks requiring high precision.

4)Advancing GenAI Assisted Programming - A Comparative Study on Prompt Efficiency and Code Quality Between GPT-4 and GLM-4

The article "Advancing GenAI Assisted Programming— A Comparative Study on Prompt Efficiency and Code Quality Between GPT-4 and GLM-4" compares the efficiency of two generative artificial intelligence models, GPT-4 and GLM-4, on assistive programming tasks. This research focuses on measuring the instantaneous efficiency, response accuracy, and quality of generated code under various programming problems. The study evaluates how effectively each model handles various coding tasks, taking into account factors such as execution speed, error rate, and understandability of the generated solutions.

Key findings show that both GPT-4 and GLM-4 have strengths in code generation and optimization, but their efficiency and output quality vary depending on the rapid complexity and specificity of the programming language used. GPT-4 has been shown to produce high-quality code with detailed prompts, but sometimes requires quick refinement to minimize errors. In contrast, GLM-4 exhibits faster response and better fault tolerance, but occasionally sacrifices some clarity in complex scenarios.

The study concludes that each model has unique advantages, suggesting that integrating agile strategies could maximize developer productivity. He further points out that while AI-assisted programming is promising, certain limitations remain, particularly when it comes to understanding nuanced syntax and error handling. Researchers are pushing further developments in rapid engineering to unlock the full potential of generative artificial intelligence for complex programming tasks.

This research highlights the complementary strengths of GPT-4 and GLM-4 and suggests that tailored approaches for each model can improve the efficiency and quality of AI-assisted code generation, particularly in specialized programming environments.

5) Who Answers It Better? An In-Depth Analysis of ChatGPT and Stack Overflow Answers to Software Engineering Questions

This research paper provides an in-depth analysis of ChatGPT's answers to software engineering questions in comparison to Stack Overflow (SO), focusing on the correctness, quality, and user preferences between the two platforms. Conducted through a mixed-methods approach—including manual analysis, linguistic assessment, and a user study—the study reveals that ChatGPT produces incorrect answers more than half of the time, with a significant portion of errors stemming from conceptual misunderstandings rather than factual inaccuracies.

The research highlights a concerning trend: while ChatGPT's answers are often incorrect, they appear convincing due to their well-articulated language, formal style, and seemingly logical presentation. This polished delivery can mislead users, especially those with less expertise, resulting in a 39.34% failure rate in detecting incorrect information. When participants failed to identify errors, they often cited ChatGPT's comprehensive and clear explanations as factors that led them to believe the answers were correct. Additionally, the study found no significant relationship between a user’s topic expertise and their ability to detect errors, indicating that even experienced users may struggle with identifying inaccuracies.

Despite these issues, the user study shows that participants preferred ChatGPT answers 34.82% of the time, mostly due to the perceived quality, comprehensiveness, and polite language of the responses. However, 77.27% of these preferred answers were incorrect, underscoring a gap between perceived and actual correctness. Stack Overflow was preferred 65.18% of the time, valued for its conciseness and accuracy, as well as the spontaneous and casual tone that some participants favored.

The paper concludes by advocating for improved transparency in AI-generated answers, suggesting that ChatGPT could benefit from mechanisms that convey the level of uncertainty or likelihood of error in responses. The authors also emphasize the importance of user awareness regarding the potential for errors in AI-generated answers and advocate for tools that support verification, such as integrated links to documentation or in-situ code execution features.

The study underscores the need for ongoing research into error types specific to AI, especially conceptual errors, which are harder to address with current methods focused on factual inaccuracies. In all, the paper calls for a cautious and informed approach to using AI tools in software engineering, highlighting the potential risks of relying solely on AI-generated responses without expert oversight.

6) Application of Large Language Models to Software Engineering Tasks: Opportunities, Risks, and Implications

The article "Applying Large Language Models to Software Engineering Tasks: Opportunities, Risks, and Implications" discusses the potential and challenges of using Large Language Models (LLMs) such as GPT-4 and BERT in software engineering. It highlights opportunities for LLMs in code generation, testing, documentation, and language translation, suggesting that LLMs can increase productivity and help developers at various stages. However, it also points to significant risks, such as the quality and bias of the training data, which can affect the reliability of the model, and concerns about privacy and content ownership. The paper raises ethical concerns, particularly regarding the trustworthiness of artificial intelligence in safety-critical applications and the environmental impact of the large-scale computing required to train the models.

In addition, the article suggests that LLM integration will require new skill sets from developers and will emphasize responsible AI practices. For example, developers need to understand when to trust LLM outputs and manage data responsibly, while computer science curriculum changes are needed to prepare the next generation for AI-enabled development. In summary, while LLMs represent a promising advance, their effective and ethical integration into software engineering workflows requires careful consideration, ongoing research, and a focus on responsible innovation.

7) A systematic evaluation of large language models for generating programming code

A recent study evaluated the coding capabilities of GPT-4 and focused on how it learns from error messages to improve code generation. The performance of GPT-4 has been shown to benefit significantly from multiple encoding attempts, especially for complex tasks. The study compared different rapid strategies, such as repeated prompts and feedback-based prompts, with a code interpreter (CI) that provides GPT-4 error messages from previous attempts. This approach increased GPT-4's success on medium and hard tasks by up to 25% compared to single-trial tasks. GPT-4's rescue rate, or its ability to correct failed tasks, was highest for the rapid CI strategy with feedback, reaching more than 60% success rate for easy and medium tasks. Notably, the rescue rate surpassed that of other language models (LLMs) such as GPT-3.5 and Claude 2, and outperformed most people in programming competitions.

In addition, GPT-4 excelled at translating Python code into languages such as Java, JavaScript, and C++, proving adept at translating even when the original Python code contained errors. This cross-language translation strategy has allowed users to successfully adapt non-Python languages to tasks that otherwise depend on Python3. In terms of computational efficiency, code generated by GPT-4 performed comparable to human-written code in runtime and memory usage, although additional optimizations made by GPT-4 yielded only small improvements.

The findings suggest that GPT-4 is an effective tool for assisting users with varying levels of expertise in programming tasks. His knowledge of iterative error correction and language translation positions him as a valuable resource for simplifying coding, helping novice users with easier tasks, and supporting experts with moderately difficult problems. However, the study acknowledges that the performance of GPT-4 in broader software engineering tasks such as project management and system architecture design remains untested, suggesting areas for future research and development in LLM-driven software engineering tools.

8) Code Generation by Emulating Software Process Models Using Large Language Model Agents

The article "When LLM-Based Code Generation Meets the Software Development Process" introduces FlowGen, a framework designed to improve code generation by simulating software process models using multiple Large Language Model (LLM) agents. This approach uses process models such as Waterfall, Test-Driven Development (TDD), and Scrum by assigning LLM agents roles similar to those in real software development (e.g., requirements engineer, developer, tester) and organizing their interactions to co-create code. FlowGen uses brainstorming and iterative self-improvement to incrementally improve code quality during the generation process.

The study compares FlowGen models—FlowGenWaterfall, FlowGenTDD, and FlowGenScrum—using GPT-3.5 and assesses their performance against existing defaults on benchmarks such as HumanEval and MBPP. Findings indicate that FlowGenScrum outperforms others, especially in handling complex tasks and improving the Pass@1 rate by up to 15% over standard LLM implementations, especially in the MBPP benchmark. In addition, the integration of FlowGen with models such as CodeT further increases performance, especially for Scrum-based simulations, which yielded the highest Pass@1 accuracy. The analysis shows that collaborative agent roles improve key aspects such as exception handling and code smell reduction, contributing to more refined and maintainable code outcomes.

In conclusion, FlowGen highlights the potential of LLM agents to simulate software roles and workflows, advancing the use of AI in software development by aligning with structured process models. This could form the basis for future AI-enabled development environments that mimic collaborative software processes.

9) Investigating Explainability of Generative AI for Code through Scenario-based Design

This research paper explores the explainability needs of software engineers using generative AI (GenAI) tools for code generation, such as natural language-to-code conversion, code translation, and auto-completion. The study investigates how these tools can be improved by understanding the specific explanations users require to trust and effectively interact with these AI systems.

The authors conducted nine participatory workshops with 43 software engineers to identify key explainability needs and design better human-AI collaboration methods. The study used a scenario-based design and a question-driven approach to elicit these needs, resulting in the identification of 11 distinct categories of explainability needs in GenAI for code. These categories included input and output space explanations, performance evaluations, and explanations regarding the model's overall behavior, limitations, and uncertainties. The study emphasizes that current research mainly focuses on representation learning and visualizing model representations, which does not fully align with users' needs for actionable insights into the behavior and outputs of GenAI models in code generation.

The paper also highlights that while some aspects of explainability, like computational accuracy, are well-explored, others, such as understanding the impact of input variations and runtime efficiency, remain under-researched. It suggests that a deeper analysis of code generation in relation to specific programming language semantics and context is essential for future studies. Additionally, there is a need for better methods to incorporate human feedback and edits into the training of generative models.

The study further reveals that explainability needs may vary depending on the user's programming expertise. Novices may benefit from more proactive, tutorial-like interactions, whereas experienced programmers may prefer contextual, in-situ explanations. The research also highlights concerns about the utility of GenAI for code, with users expressing frustration about the learning curve required to optimize their inputs and improve AI outputs.

The paper concludes by urging future research in XAI for GenAI to consider user needs, technical feasibility, and contextual factors in the design and evaluation of AI systems. It advocates for participatory, user-centered approaches to better understand how GenAI can support software engineers and improve human-AI collaboration.

10) Assisting Static Analysis with Large Language Models: A ChatGPT Experiment

The paper Assisting Static Analysis with Large Language Models: A ChatGPT Experiment by Haonan Li, Yizhuo Zhai, Yu Hao, and Zhiyun Qian examines the use of large language models (LLMs) such as ChatGPT to improve static analysis, specifically focusing on errors. a detection tool called UBITect, used to detect Use-Before-Initialization (UBI) errors in the Linux kernel. Static analysis tools such as UBITect, while powerful, often produce a high number of false positives due to the limitations of traditional analysis techniques. The authors suggest that LLMs can help reduce these false alarms by analyzing and generating feature summaries that improve the tool's accuracy.

In their study, the authors developed a methodology where ChatGPT is asked targeted questions about feature-level behavior. Using prompts that support chain processing and a progressive prompt structure, ChatGPT is used to examine specific code contexts, identify necessary initializations, and determine whether potential errors are true or false positives. The evaluation included two models, GPT-3.5 and GPT-4, which were tested on 20 false positives identified by the UBITect system. The results showed that GPT-4 performed significantly better, accurately identifying false positives in 16 out of 20 cases, compared to only 8 for GPT-3.5. In addition, GPT-4 has demonstrated the ability to handle cases involving special functions and other conditions that are challenging for static analysis tools due to their complexity.

The authors highlight two main challenges in static analysis that LLM can address: (1) Inherent knowledge boundaries, which include complex or domain-specific features that require expert knowledge to analyze, and (2) Exhaustive Path Exploration, where tools struggle to assess all possible ways to do it effectively. In contrast, ChatGPT can provide targeted information on specific code paths, reducing the need for exhaustive exploration.

The study's findings suggest that LLMs like GPT-4 can be an effective adjunct to static analysis, helping to reduce false positives and even identify false negatives (missing errors) in certain cases. The authors also suggest that the ability of LLMs to interpret and generate feature summaries could make them invaluable tools in automated program analysis with potential applications in various areas of software engineering. The code and methodology used in this study have been made open source to encourage further research and development in this area.

CHAPTER 3
REQUIREMENT ANALYSIS OF SOLUTION APPROACH

3.1 Overall Description of Project

The growing capabilities of large language models (LLMs), such as OpenAI GPT and Google Gemini, have opened up new possibilities in software development, especially in generating code based on natural language challenges. These models promise increased productivity, reduced development time and optimised solutions. However, as their use becomes more widespread, it is necessary to critically evaluate how LLM-generated code compares to human-written code, especially in terms of quality, readability, and context understanding. Our project seeks to fill this gap by systematically analysing and comparing AI-generated code with human code across multiple parameters.

Our study includes a comprehensive dataset of approximately 400 code samples with an even split between C++ and Python code. We've also introduced two levels of question complexity - Level 1 with simpler, more straightforward challenges and Level 2 with story-driven, more step-based challenges. This distinction allows us to observe how models handle tasks of varying difficulty, from basic operations to scenarios requiring deeper logical reasoning. To achieve a meaningful comparison, we focused on four primary metrics: cyclomatic complexity, lines of code, time complexity, and space complexity. Cyclomatic complexity measures the logical depth of code, revealing its structural complexity and impact on readability and maintainability. Lines of code provide a measure of verbosity, while time and space complexity offer insight into computational and memory efficiency, both of which are key to optimising code in real-world applications.

In order to efficiently analyze the 400 code samples, we used a structured five-step approach. First, we developed an interactive dashboard that will serve as a centralized platform for visualizing and comparing results. Built using HTML, CSS, JavaScript, MongoDB, C++, and Python, this dashboard is designed to present a clear, side-by-side comparison of human-written and AI-generated code across all metrics, allowing users to explore the nuances of AI performance.

With the dashboard in place, our next phase focused on researching and defining the lens. A thorough review of existing studies helped us define our specific research goals and understand the current perspectives of artificial intelligence in software development. The goals of our project include assessing how well AI handles coding tasks, understanding the impact of rapid specificity on code quality, and identifying common limitations or "hallucinations" in AI-generated output. These goals led us to define metrics and create an analytical framework for our comparisons.

Data collection was the crucial third phase. Here we've collected human-written and AI-generated code samples in both C++ and Python. By using challenges that mirror real programming challenges, we have ensured that our dataset is diverse and usable. In total, we generated 40 code samples for C++ and Python at both complexity levels using ChatGPT and Gemini to create AI-generated samples. This dataset formed the basis for our next phase – analysis.

During the analysis phase, we used selected metrics to evaluate the quality and efficiency of each code sample. By analyzing the cyclomatic complexity, line count, time complexity, and spatial complexity of each sample, we generated detailed insights into the strengths and weaknesses of both human and AI code. The Tier 1 and Tier 2 divisions in terms of complexity highlight where AI models perform well and where they may struggle with nuanced or complex tasks. This phase also shed light on the influence of immediate specificity: as tasks became more complex, the need for precise, well-defined instructions became more apparent, impacting both ChatGPT and Gemini performance.

The final phase involved implementing automated analysis into a dashboard, creating a user-friendly interface where comparisons could be visualised interactively. This automated analysis allows users to see trends, patterns and specific insights at a glance. Through interactive features, users can switch between different programming languages, models and levels of complexity to understand how AI-generated code differs in different contexts. The dashboard offers a hands-on real-time visualisation of our findings, making the results accessible to developers, educators, and researchers.

The dual research and development approach of our project is unique, combining empirical study with practical tool making. Our analysis not only provides valuable insights into the role of the LLM in coding, but also provides developers with a tangible resource in the form of an interactive dashboard to explore and understand the potential of AI in software development. Through this project, we aim to inform industry experts and the academic community about the strengths, limitations and ideal use cases of AI in programming.

Research Questions:-

RQ1 - How do correctness, structure, and performance differ between human-generated and LLM-generated code?

RQ2 - How does the specificity and quality of prompts influence the accuracy, structure, and relevance of LLM-generated code?

RQ3 - What factors drive user preference for human-generated versus LLM-generated code?

3.2 Solution Approach:

3.2.1 Phases of the Project

- **Phase 1: Research and Defining Objectives:-**
  In the first stage, we will conduct extensive research to understand the current landscape of AI-generated code versus human-written code. This involves reviewing existing literature, identifying gaps in the current understanding, and defining clear research questions and objectives. By the end of this stage, we aim to finalize the specific metrics and aspects we want to analyze,

- **Phase 2: Data Collection and Dataset Creation:-**
  The second stage focuses on collecting the necessary data for our analysis. We will gather human-written code samples in the programming languages we plan to study. At the same time, we will use LLMs to generate code for similar problems or tasks. This process will allow us to build a comprehensive dataset that includes both human-written and AI-generated code samples. The dataset will be diverse, covering various types of problems to ensure a robust analysis.

- **Phase 3: Analysing and Comparing Codes:-**
  Once the dataset is ready, the third stage involves performing a detailed analysis of the collected codes. We will compare the human-written and LLM-generated codes based on several software engineering predefined metrics. This comparison will help us draw meaningful conclusions about the strengths and weaknesses of AI-generated code relative to human-written code. The insights gained from this analysis will form the basis of our findings.

- **Phase 4: Developing a Dashboard for Automated Analysis:-**
  In the final stage, we will develop a dashboard to automate the comparison and analysis process of the two categories of code human-written and LLM-generated. This dashboard will provide an interactive interface to visualize the differences and results across various metrics. It will easily help us understand the outcomes of our study and explore the data in detail.

3.2.2 Overview of Large Language Models (LLM)

Large language models (LLM) are advanced artificial intelligence systems trained on large data sets. They understand, generate and manipulate human language, enabling applications that range from automated typing assistance to code generation. These models use deep learning architectures, specifically transformers, to process language patterns and learn from large datasets, often across multiple languages and domains. This training enables LLMs to generate contextually relevant answers based on given challenges and offers powerful support for coding, content creation, language translation and more. However, their performance varies depending on their design, training data, and specific model capabilities, so choosing the right model is critical to achieving optimal results.

<u>Selection of used models</u>

For this project, we selected ChatGPT 3.5 and Gemini 1.5, both highly capable language models, to compare their code generation capabilities and accuracy in mimicking human programming styles. Below is a brief description of each:

1) **ChatGPT 3.5:** Developed by OpenAI, ChatGPT 3.5 is an LLM tuned to generate human responses. It excels at understanding natural language and generating code in a conversational format, which is often useful when debugging or completing code based on prompts. With an architecture designed to understand context, ChatGPT 3.5 demonstrates strong adaptability across languages, making it popular for a wide range of coding tasks.
2) **Gemini 1.5:** Built by Google DeepMind, Gemini 1.5 focuses on providing well-structured, precise code and language answers. It is particularly effective in solving complex problems that strive for accuracy and relevance. Gemini's training and architectural design contribute to its structured approach to language generation, which in this project allows us to evaluate

its effectiveness in producing robust and reliable code solutions that conform to human programming standards.

3.2.3 Prompts level Design

To evaluate the capabilities of the Large Language Models (LLMs) in generating code, we employ two distinct levels of prompts. These levels are designed to assess the model's ability to understand and extract logic from varying degrees of complexity in the input.

1. **L1**
   In this level, the prompt is straightforward and direct, providing a clear and concise problem statement with specific requirements for the code. This type of prompt is designed to test the model's ability to generate functional code when given explicit instructions without the need for much interpretation or contextual understanding. The simplicity of this prompt allows us to focus on how accurately and efficiently the model can respond to basic coding tasks.

   Sample prompt: *Given two values M and N, which represent a matrix[M][N]. We need to find the total unique paths from the top-left cell (matrix[0][0]) to the rightmost cell (matrix[M-1][N-1]).*

   At any cell, we are allowed to move in only two directions:- bottom and right.

2. **L2**
   At this level, the prompt is more complex and written in a story-like manner, containing additional context, background information, and potentially extraneous details. This type of prompt is designed to challenge the model's ability to extract the main logic and essential elements of the problem, filtering out irrelevant information to generate the correct solution. The complexity of this prompt tests the model's efficiency and proficiency in handling real-world scenarios where the problem may be less defined and require more cognitive processing to generate the appropriate code.

Sample prompt: *Imagine you're navigating through a large grid in a sprawling castle. You start at the top-left corner of the castle and your goal is to reach the bottom-right corner. The only paths you can take are moving right or moving down.*

The castle is represented as a grid, and you need to determine how many unique ways you can travel from the top-left corner to the bottom-right corner without stepping outside the grid.

For example, if the castle grid is 3x3 (3 rows and 3 columns), you need to figure out how many distinct paths you can take to get from the starting corner to the final corner of the castle, only moving to the right or downward.

Can you help navigate the castle and find the total number of unique paths to reach the end?

3.2.4 Evaluation Criteria

To thoroughly compare human-written code with code generated using Large Language Models (LLM), we established evaluation criteria focused on accuracy, complexity, comprehensibility, style, and optimization. This structured approach allows us to highlight the performance and capabilities of LLM over human coding standards.

1) Accuracy

The primary criterion for evaluating LLM-generated code is its accuracy, which assesses whether the code executes correctly and meets the requirements of the challenge or problem statement. The exact code should compile without errors and produce the expected outputs. This criterion is essential in verifying the functional correctness of the code, ensuring that it conforms to the intended results, and serves as a basis for further comparison with human-written code.

2) Complexity

Complexity assesses the structural and logical complexity of the code, including readability, adherence to conventions, and overall style. We assess whether the generated code uses constructs that are too complex or difficult to understand. When evaluating complexity, we compare it to corresponding human-written code to see if the model can achieve a similar level of clarity and manageability, since human coders often balance complex logic with readability.

3) Optimization

Optimization examines how efficiently code is written in terms of resource usage and runtime, beyond simply meeting functional requirements. This includes analyzing whether LLM uses efficient algorithms or whether simpler, less optimal methods are used by default. In this evaluation, we compare LLM-generated code with human-written solutions to determine whether the model achieves similar efficiency or relies on less practical brute-force approaches.

3.2.5 Evaluation Parameters

To provide a detailed assessment of LLM-generated code compared to human-written code, we identified four key evaluation parameters: cyclomatic complexity, lines of code, time complexity, and space complexity. These parameters help us objectively analyze the efficiency, readability and performance of the code produced by each model.

1) Cyclomatic complexity

Cyclomatic complexity measures the number of linearly independent paths within a program, reflecting its complexity and potential maintenance difficulty. We use this parameter to judge whether the LLM-generated code is overly complicated or simplified, and compare it to the corresponding human-written code to see whether the model achieves a similar level of simplicity or introduces unnecessary complexity.

2) Lines of Code (LOC)

This parameter tracks the total number of lines in the code, which helps us measure brevity and readability. By comparing lines of code between LLM-generated code and human-written code, we evaluate whether the model output is unnecessarily verbose or brief enough to meet human coding standards.

3) Time complexity

Time complexity shows how the running time of the code changes as the input size increases, which is a critical factor in performance evaluation. We examine whether LLM-generated code is as efficient as human-written code for various inputs, allowing us to see whether the model favors efficient solutions or tends toward slower and less efficient approaches.

4) Cosmic complexity

Space complexity assesses code memory usage relative to input size, which is particularly important in resource-constrained environments. By comparing the spatial complexity in LLM-generated code and human-written code, we determine whether the model produces code that optimally balances memory usage or lacks resource management efficiency.

3.2.6 Experiment Environment

To create a robust dataset for comparing human-written and LLM-generated code, we conducted extensive research to select a balanced set of 40 programming questions. These questions are categorised by difficulty—easy, medium, and hard—and come from two popular coding platforms, LeetCode and Codeforces. For each question, we created two levels of prompts: a direct prompt (Level 1) and a complex story-like prompt (Level 2), designed to test each model's response to different levels of prompt complexity.

To allow a comprehensive comparison based on languages, we selected C++ and Python as our primary languages. For each of the 40 questions, we collected human-written code solutions in both C++ and Python, creating a dataset that represents authentic coding practices.

Comments Remover WebApp

To maintain uniformity in the dataset, we developed a web application that removes comments from the inputted code while preserving the original format and indentation. This tool was essential in ensuring consistency across all human-written code samples, as inconsistent commenting could otherwise create abruptness in the analysis. The cleaned, comment-free versions were stored in a separate database to serve as the final human-written codes for comparison.

Additionally, we created an Excel file to systematically document all relevant evaluation parameters such as cyclomatic complexity, lines of code, time complexity, and space complexity for these human-written codes. This structured data serves as the benchmark for evaluating and comparing each LLM’s performance across various coding metrics, providing a consistent and organised environment for detailed analysis.

CPP script to calculate cyclomatic complexity

This is designed to analyse C++ code for two essential metrics: Lines of Code (LOC) and Cyclomatic Complexity. These metrics provide valuable insights into the code's length and logical complexity, helping developers understand its readability and potential maintenance challenges.

Key Features of the Script

1) Lines of Code (LOC):

Purpose: LOC is a straightforward metric indicating the length of the code, typically by counting non-empty lines. It serves as a basic measure of the code's size and verbosity, which can impact readability and maintenance.

Implementation: The script splits the input code into individual lines, filtering out empty lines and counting only those that contain actual code. This count provides an approximation of LOC, ignoring blank spaces to focus purely on the functional lines.

2) Cyclomatic Complexity:

Purpose: Cyclomatic Complexity measures the number of independent paths through a code base, which reflects its logical complexity. A higher cyclomatic complexity means the code has multiple decision points, making it harder to understand and more error-prone, while a lower complexity indicates simpler, more straightforward code.

Calculation Method:

- The script starts by setting the base cyclomatic complexity to 1, accounting for the single execution path of the simplest code.
- It identifies decision points places where control flow can branch using specific keywords like if, for, while, case, catch, &&, and ||. Each of these keywords represents a possible branching or decision-making point in the code.
- Using regular expressions, the script searches for each keyword within the code. For each match found, it increments the cyclomatic complexity by 1, accounting for the new path introduced by that decision point.

Output:

The script outputs a dictionary with two key-value pairs: one for LOC and one for cyclomatic complexity. This output provides a concise, quantitative summary of the code's size and complexity, which can then be used for code quality assessments or to gauge refactoring needs.

<u>Python script to calculate cyclomatic complexity</u>

The Python script calculate_mccabe_complexity_from_string is designed to evaluate cyclomatic complexity directly from a Python code string by leveraging the McCabe complexity measurement tool within the flake8 package. Cyclomatic complexity helps quantify the number of independent paths in code, which is especially useful for understanding the code's structural complexity and identifying areas that may require simplification or additional testing.

Key Features of the Script

Overview

This script uses flake8 with the McCabe plugin, a popular Python toolset for static code analysis, to calculate cyclomatic complexity. This approach automates complexity measurement by evaluating the provided Python code string without requiring manual analysis.

Implementation Details

1) Checking flake8 Installation:

The script first verifies that flake8 is installed and accessible. If flake8 is not found, it raises an exception and prompts the user to install flake8 using pip install flake8.

2) Temporary File Creation:

The Python code string is written to a temporary file, allowing flake8 to process it as if it were a standalone script. This temporary file is crucial because flake8 operates on file paths, not on strings directly.

3) Executing flake8 with McCabe Plugin:

The script runs flake8 on the temporary file with --max-complexity=0 to remove complexity limits and --select=C90 to filter results to only cyclomatic complexity measurements. This configuration ensures that all complexity values are reported regardless of their level.

If all functions are within the acceptable complexity threshold, a message confirms that. Otherwise, a detailed cyclomatic complexity report is displayed, showing the complexity values for functions exceeding the specified limit.

4) Output and Cleanup:

The script prints out the cyclomatic complexity of each function, providing insights into the complexity and potential areas for simplification.

Finally, the temporary file is deleted to ensure no residual files are left on the system.

Calculating time and space complexity

In this project, we used BigO Calculator from *bigocalc.com* to determine the time and space complexity of various code samples. BigO Calculator is an online tool designed to analyze and approximate the computational complexity of code, which is invaluable in understanding the scalability and efficiency of algorithms, especially for larger inputs.

Key Features of BigO Calculator

- Time Complexity: Measures how the runtime of an algorithm increases as the size of the input grows. This metric is essential for identifying potential bottlenecks in code execution, particularly in resource-constrained environments.
- Space Complexity: Measures the amount of memory an algorithm uses relative to the input size. Understanding space complexity is crucial for applications with limited memory or when handling large data sets.

3.2.7 Experiment Procedure

The following steps outline the systematic process for generating, testing, and evaluating code from the Large Language Models (LLMs):

- **Step 1:** Start a new conversation with the LLM to ensure a fresh session with no prior context or memory, for unbiased code generation.
- **Step 2:** Input the pre-processed prompt (Level 1 or Level 2) and specify the target programming language (C++ or Python) to guide the LLM in generating the code.
- **Step 3:** Run the generated code to test for correctness, verifying if it executes successfully and meets the prompt requirements.
- **Step 4:** Record the generated code for storage and further analysis.
- **Step 5:** Process the code using relevant Python or C++ scripts, depending on the language, to analyse LOC and cyclomatic complexity.
- **Step 6:** Record the lines of code (LOC) and cyclomatic complexity for the generated code.
- **Step 7:** Input the code into a Big O calculator to assess the algorithm's time complexity and space complexity.
- **Step 8:** Record the time complexity and space complexity values for comparison.

Repeat the above steps for different level prompts and models, maintaining the same procedure to ensure consistent evaluation across prompt levels and LLMs.

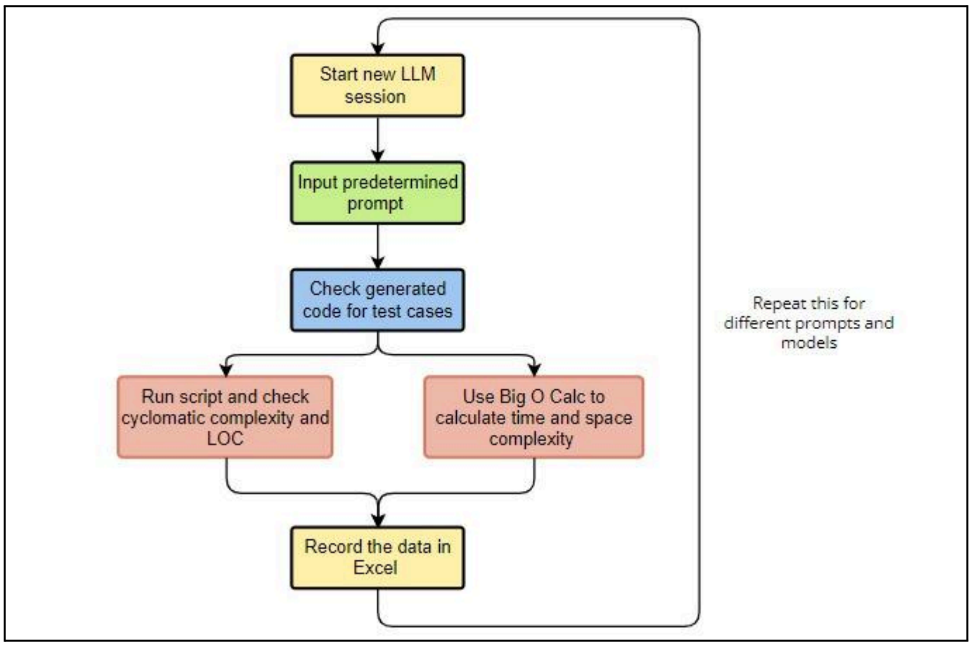

Start new LLM session
Input predetermined prompt
Check generated code for test cases
Run script and check cyclomatic complexity and LOC
Use Big O Calc to calculate time and space complexity
Record the data in Excel
Repeat this for different prompts and models

Fig - Experiment Procedure flow

<u>Sample Case 1</u>

Model: ChatGPT

Level: 1

Language: CPP

Prompt: *Petya loves football very much. One day, as he was watching a football match, he was writing the players' current positions on a piece of paper.*

To simplify the situation he depicted it as a string consisting of zeroes and ones.

A zero corresponds to players of one team; a one corresponds to players of another team.

If there are at least 7 players of some team standing one after another, then the situation is considered dangerous.

For example, the situation 00100110111111101 is dangerous and 11110111011101 is not.

You are given the current situation. Determine whether it is dangerous or not.

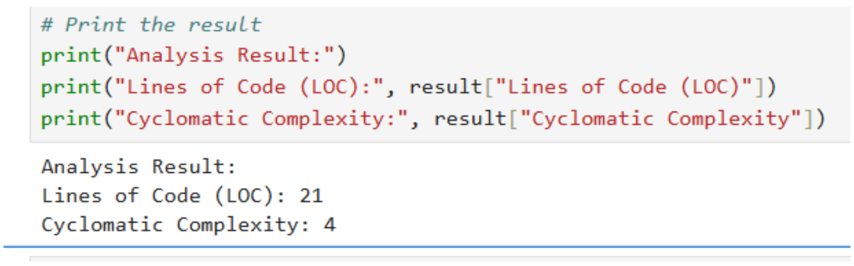
Generate cpp code without comments.

<u>C++ code:</u>

```
#include <iostream>
#include <string>
using namespace std;
```

```
int main() {
    string s;
    cin >> s;
    int count = 1;
    for (size_t i = 1; i < s.length(); i++) {
        if (s[i] == s[i - 1]) {
            count++;
            if (count >= 7) {
                cout << "YES" << endl;
                return 0;
            }
        } else {
            count = 1;
        }
    }
    cout << "NO" << endl;
    return 0;
}
```

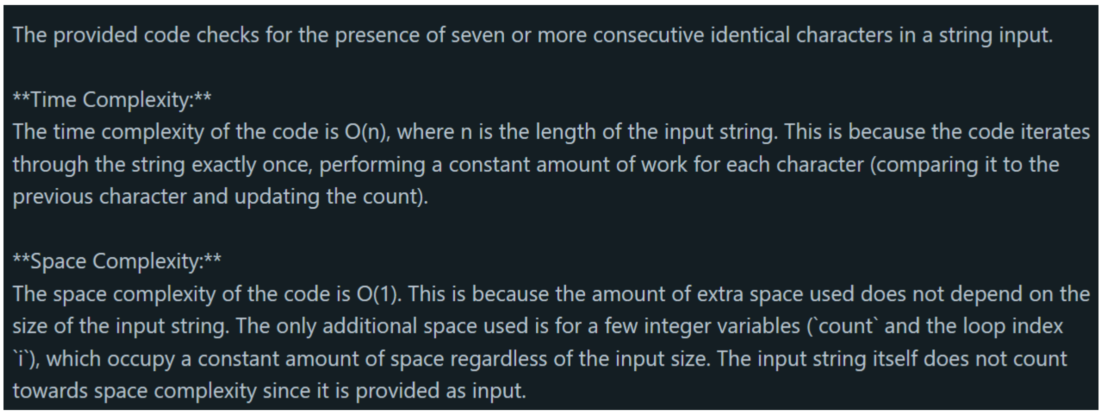
Cyclomatic complexity and lines of code:



```
# Print the result
print("Analysis Result:")
print("Lines of Code (LOC):", result["Lines of Code (LOC)"])
print("Cyclomatic Complexity:", result["Cyclomatic Complexity"])
```

```
Analysis Result:
Lines of Code (LOC): 21
Cyclomatic Complexity: 4
```

Time and Space complexity:



The provided code checks for the presence of seven or more consecutive identical characters in a string input.

Time Complexity:
The time complexity of the code is O(n), where n is the length of the input string. This is because the code iterates through the string exactly once, performing a constant amount of work for each character (comparing it to the previous character and updating the count).

Space Complexity:
The space complexity of the code is O(1). This is because the amount of extra space used does not depend on the size of the input string. The only additional space used is for a few integer variables (`count` and the loop index `i`), which occupy a constant amount of space regardless of the input size. The input string itself does not count towards space complexity since it is provided as input.

__Sample Case 2__

Model: Gemini

Level: 2

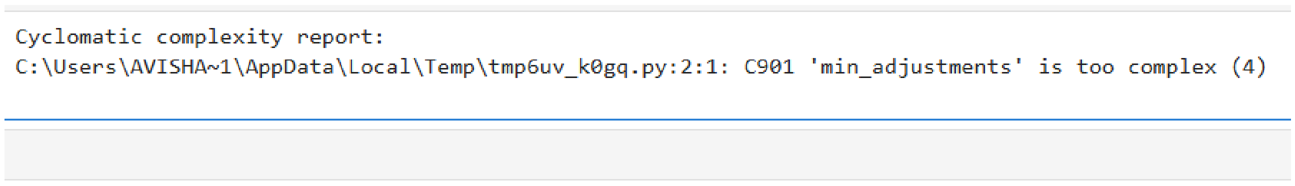
Language: Python

Prompt: *In a magical forest, there lived a talented carpenter named Elara. She was known far and wide for crafting the finest wooden toys. One day, Elara received a special request from the village children to create a set of three beautiful equilateral triangle-shaped kites for their upcoming festival.*
Elara gathered her tools and headed to her workshop, where she had a collection of n sticks, each with a positive integral length
a1,a2...an.
As she laid out the sticks, she realised that not all of them were of equal length. To craft the perfect kites, she needed to ensure that she could select three sticks of the same length.
Determined to fulfil the children's request, Elara thought about how she could adjust the lengths of her sticks. She could either increase or decrease the length of any chosen stick by 1, as long as none of the sticks would lose their positive length.
Elara knew that time was of the essence, and she needed to figure out the minimum number of adjustments required to make it possible to select three sticks that would form an equilateral triangle.
Could you help Elara find the least number of operations she must perform to create the three kites for the children?
Give python code without comments.

Python code:

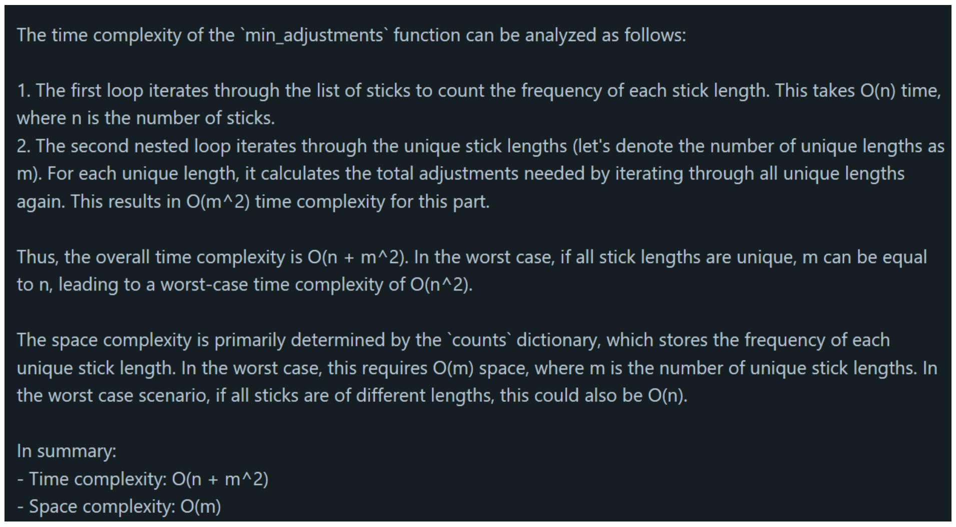
```
def min_adjustments(sticks):
    counts = {}
    for stick in sticks:
        counts[stick] = counts.get(stick, 0) + 1
    min_adjustments = float('inf')
    for target_length, count in counts.items():
        adjustments = 0
        for length, freq in counts.items():
            adjustments += abs(length - target_length) * freq
        min_adjustments = min(min_adjustments, adjustments)
    return min_adjustments // 2
sticks = [1, 2, 3, 4, 5]
result = min_adjustments(sticks)
print(result)  # Output: 4
```

<u>Cyclomatic complexity and lines of code:</u>



```
Cyclomatic complexity report:
C:\Users\AVISHA~1\AppData\Local\Temp\tmp6uv_k0gq.py:2:1: C901 'min_adjustments' is too complex (4)
```

<u>Time and Space complexity:</u>



The time complexity of the `min_adjustments` function can be analyzed as follows:

1. The first loop iterates through the list of sticks to count the frequency of each stick length. This takes O(n) time, where n is the number of sticks.
2. The second nested loop iterates through the unique stick lengths (let's denote the number of unique lengths as m). For each unique length, it calculates the total adjustments needed by iterating through all unique lengths again. This results in O(m^2) time complexity for this part.

Thus, the overall time complexity is O(n + m^2). In the worst case, if all stick lengths are unique, m can be equal to n, leading to a worst-case time complexity of O(n^2).

The space complexity is primarily determined by the `counts` dictionary, which stores the frequency of each unique stick length. In the worst case, this requires O(m) space, where m is the number of unique stick lengths. In the worst case scenario, if all sticks are of different lengths, this could also be O(n).

In summary:
- Time complexity: O(n + m^2)
- Space complexity: O(m)

3.2.8 Checking Generated Code Accuracy

To ensure the accuracy of the code generated by LLM, we selected a diverse set of questions from LeetCode and Codeforces, two widely used platforms for coding problems and algorithm testing. These platforms are popular among developers for their robust test cases that provide a reliable benchmark for evaluating code correctness.

For each question, we take code generated directly from LLM and run it on the appropriate platform. The evaluation is binary with two possible outcomes:

- Pass: The code successfully runs on all test cases and provides the correct outputs for each of them, indicating a complete solution.
- Failure: If the code encounters a problem—whether it's a compilation error, a runtime error, or incorrect output in any test case—it's marked as a failure.

This direct pass/fail approach allows us to objectively assess LLM's ability to generate functional code that meets all the problem requirements at various complexity levels.

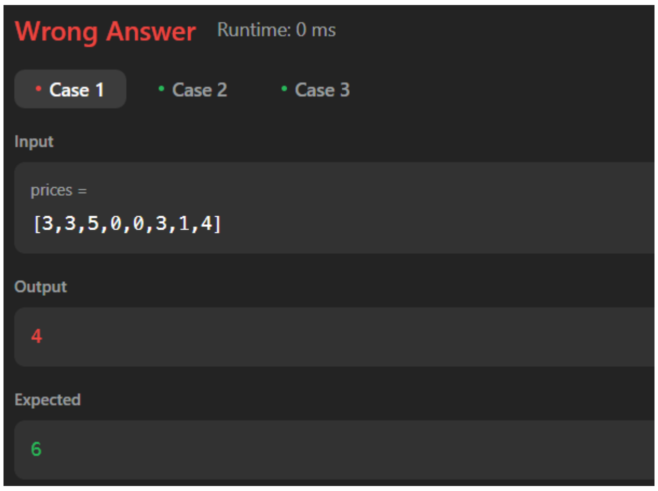
3.2.9 Follow-up prompt setup

Stage 1: Basic and Incorrect Attempt

Prompt 1:
"Write a C++ program to find the maximum profit I can achieve by buying and selling a stock once, given an array of prices for each day."

Output:

```
int maxProfit(vector<int>& prices) {

      int minPrice = INT_MAX;

    int maxProfit = 0;

    for (int price : prices) {

        minPrice = min(minPrice, price);

        maxProfit = max(maxProfit, price - minPrice);

    }

    return maxProfit;

}
```



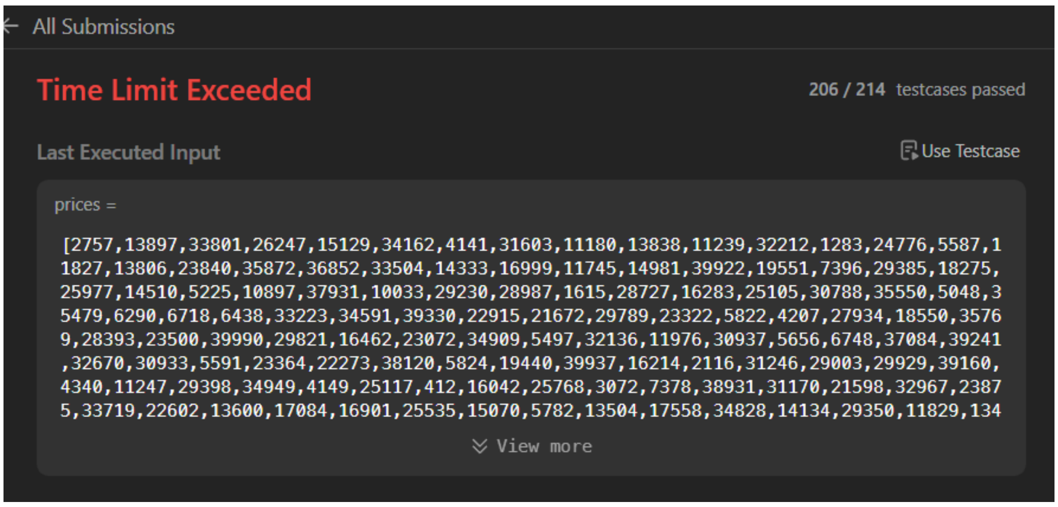
```
Input
prices =
[3,3,5,0,0,3,1,4]
Output
4
Expected
6
```

Stage 2: Attempt to Handle Two Transactions (Initial and Buggy Logic)

Prompt 2:
"*The program works for one transaction, but now I need to allow at most two transactions. Each transaction means I buy and sell once. Can you modify the code so it handles two transactions properly? Make sure I can't start the second transaction until the first is finished.*"

Output:

```
int maxProfit(vector<int>& prices) {
      int n = prices.size();
    if (n <= 1) return 0;
    int maxProfit = 0;
    for (int i = 0; i < n; ++i) {
        int leftProfit = 0;
        int rightProfit = 0;
        int minPriceLeft = INT_MAX;
        for (int j = 0; j <= i; ++j) {
            minPriceLeft = min(minPriceLeft, prices[j]);

            leftProfit = max(leftProfit, prices[j] - minPriceLeft);

        }

        int minPriceRight = INT_MAX;

        for (int j = i + 1; j < n; ++j) {

            minPriceRight = min(minPriceRight, prices[j]);

            rightProfit = max(rightProfit, prices[j] - minPriceRight);

        }

        maxProfit = max(maxProfit, leftProfit + rightProfit);

    }

    return maxProfit;

}
```



Time Limit Exceeded

206 / 214 testcases passed

Last Executed Input

prices =

```
[2757,13897,33801,26247,15129,34162,4141,31603,11180,13838,11239,32212,1283,24776,5587,1
1827,13806,23840,35872,36852,33504,14333,16999,11745,14981,39922,19551,7396,29385,18275,
25977,14510,5225,10897,37931,10033,29230,28987,1615,28727,16283,25105,30788,35550,5048,3
5479,6290,6718,6438,33223,34591,39330,22915,21672,29789,23322,5822,4207,27934,18550,3576
9,28393,23500,39990,29821,16462,23072,34909,5497,32136,11976,30937,5656,6748,37084,39241
,32670,30933,5591,23364,22273,38120,5824,19440,39937,16214,2116,31246,29003,29929,39160,
4340,11247,29398,34949,4149,25117,412,16042,25768,3072,7378,38931,31170,21598,32967,2387
5,33719,22602,13600,17084,16901,25535,15070,5782,13504,17558,34828,14134,29350,11829,134
```

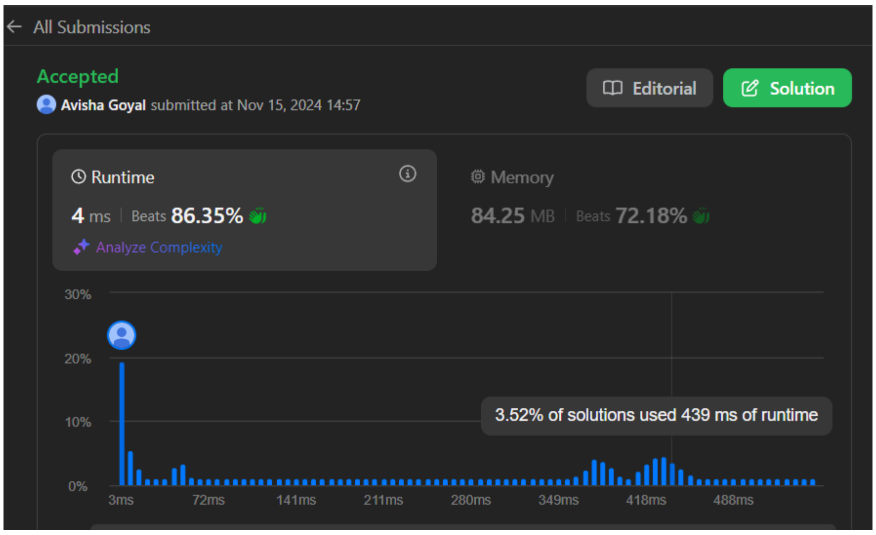
Stage 3: Efficient Approach Using Dynamic Programming

Prompt 3:

"The previous solution is too slow. Can we make it more efficient?

Output:

```
int maxProfit(vector<int>& prices) {

      int n = prices.size();

    if (n <= 1) return 0;

    vector<int> leftProfit(n, 0);

    vector<int> rightProfit(n, 0);

    int minPrice = prices[0];

    for (int i = 1; i < n; ++i) {

        minPrice = min(minPrice, prices[i]);

        leftProfit[i] = max(leftProfit[i - 1], prices[i] - minPrice);

    }

    int maxPrice = prices[n - 1];

    for (int i = n - 2; i >= 0; --i) {

        maxPrice = max(maxPrice, prices[i]);

        rightProfit[i] = max(rightProfit[i + 1], maxPrice - prices[i]);

    }

    int maxProfit = 0;

    for (int i = 0; i < n; ++i) {

        maxProfit = max(maxProfit, leftProfit[i] + rightProfit[i]);
    }
    return maxProfit;
}
```



Accepted
Avisha Goyal submitted at Nov 15, 2024 14:57

Runtime
4 ms | Beats 86.35%

Memory
84.25 MB | Beats 72.18%

3.52% of solutions used 439 ms of runtime

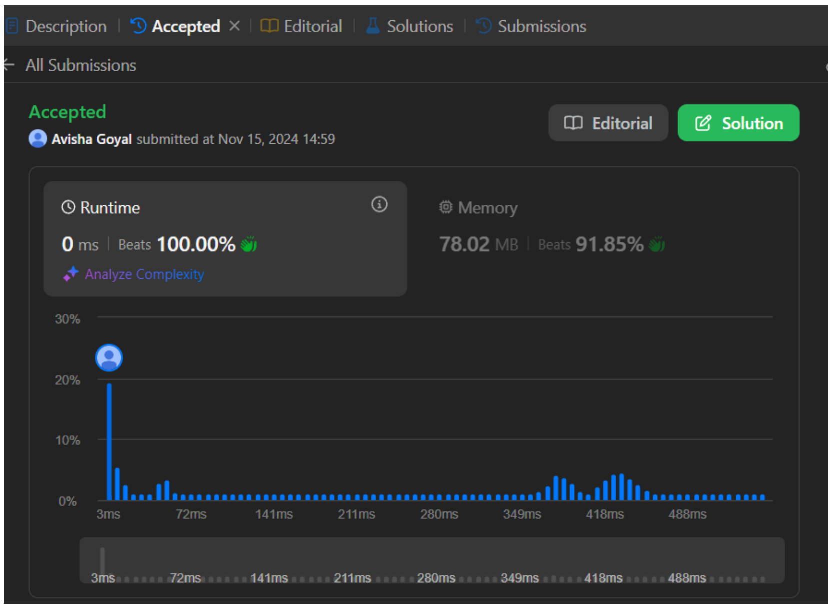
Stage 4: Fully Optimised Solution Using Variables

Prompt 4:
"Can we improve the solution further? I want to reduce the extra space used by arrays."

Output:

```
int maxProfit(vector<int>& prices) {
     int n = prices.size();
    if (n <= 1) return 0;
    int firstBuy = INT_MIN, firstSell = 0;
    int secondBuy = INT_MIN, secondSell = 0;
    for (int price : prices) {
        firstBuy = max(firstBuy, -price);
```

```
            firstSell = max(firstSell, firstBuy + price);

            secondBuy = max(secondBuy, firstSell - price);

            secondSell = max(secondSell, secondBuy + price);

        }

        return secondSell;

}
```



Accepted
Avisha Goyal submitted at Nov 15, 2024 14:59

Runtime
0 ms | Beats 100.00%

Memory
78.02 MB | Beats 91.85%

3.2.10 Data Collection

The data collection process was structured to ensure consistency, accuracy, and proper organisation for effective analysis. The steps involved are as follows:

1. **Selection of Questions:**
   A set of 40 diverse coding questions was chosen from popular platforms such as LeetCode and Codeforces. These questions were selected to cover varying levels of complexity and test the capabilities of both human-written and LLM-generated code.
2. **Human-Written Code Collection:**
   Corresponding human-written solutions for these 40 questions were found, stored, and organised. To ensure better comparison and avoid unnecessary distractions, comments were removed from these codes, and the processed codes were stored in a separate folder with consistent formatting.

3. **LLM-Generated Code Organization:**
   Separate folders were created to store the codes generated by the LLMs. These included distinct directories for ChatGPT and Gemini, each containing subfolders for:
   - Programming Languages (C++ and Python)
   - Prompt Levels (Level 1 and Level 2)
     Each code file was properly labeled and stored according to its corresponding question number, ensuring traceability and ease of comparison.
4. **Code Generation and Storage:**
   Using one-shot prompts for both levels, nearly 400 codes were generated across the two models, two languages, and two prompt levels. These codes were meticulously stored in their respective directories, maintaining a clear and systematic structure.
5. **Analysis Preparation:**
   After generating the codes, scripts and online tools were utilized to calculate evaluation parameters such as lines of code, cyclomatic complexity, and time and space complexity. The results were then compiled into an Excel file for further analysis.
6. **Excel Dataset Creation:**
   To enhance readability and simplify the comparison, the dataset was divided into four distinct sheets:
   - ChatGPT - C++
   - ChatGPT - Python
   - Gemini - C++
   - Gemini - Python
     This organised structure facilitated a clear understanding of the data and streamlined the analysis process.

3.2.11 Overview of the dashboard

This dashboard is designed as a comprehensive tool to analyze and compare human-generated and AI-generated code.

- **Left Side (Human Code Analyzer):**
  Displays 40 pre-stored questions for both C++ and Python, allowing users to fetch details like cyclomatic complexity, time complexity, and space complexity stored in a MongoDB database.

- **Right Side (AI Code Generator and Complexity Analyzer):**
  Provides a user-friendly interface powered by the Gemini API to generate code based on user prompts. The generated code is analyzed for its complexity and displayed in real-time.

Purpose

The dashboard was created to **automate the process** of generating code using Gemini AI and comparing it with human-generated code. It provides an organized interface for efficient analysis, enabling users to assess code quality and performance easily.

Tech Stack Used

1. **Frontend:** HTML, CSS, JavaScript (responsive UI and interactivity).
2. **Backend:** Node.js, Express.js (API development).
3. **Database:** MongoDB (storing and retrieving question data with complexities).
4. **AI Integration:** Python for interacting with the Gemini API (code generation and complexity analysis).

CHAPTER 4
MODELLING AND IMPLEMENTATION DETAILS

4.1 Design Diagrams

Flow Chart

A flowchart is a type of diagram that represents a workflow or process. A flowchart can also be defined as a diagrammatic representation of an algorithm, a step-by-step approach to solving a task. The flowchart shows the steps as boxes of various kinds, and their order by connecting the boxes with arrows. It is a generic tool that can be adapted for a wide variety of purposes and can be used to describe various processes, such as a manufacturing process, an administrative or service process, or a project plan.

The flowchart shows the steps as boxes of various kinds, and their order by connecting the boxes with arrows. This diagrammatic representation illustrates a solution model to a given problem.

Flowcharts are used in analysing, designing, documenting or managing a process or program in various fields.

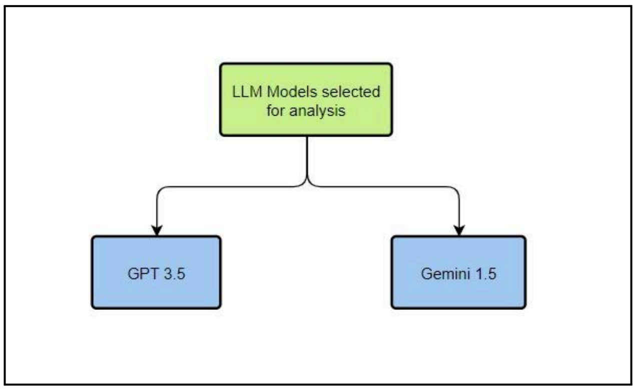


Fig-1.1 Design Diagram 1

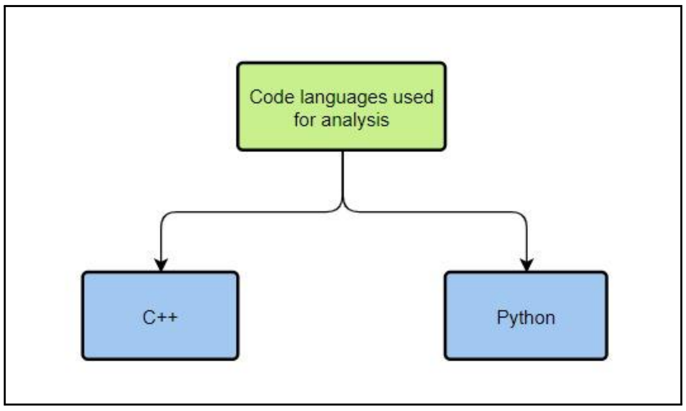


Fig-1.2 Design Diagram 2

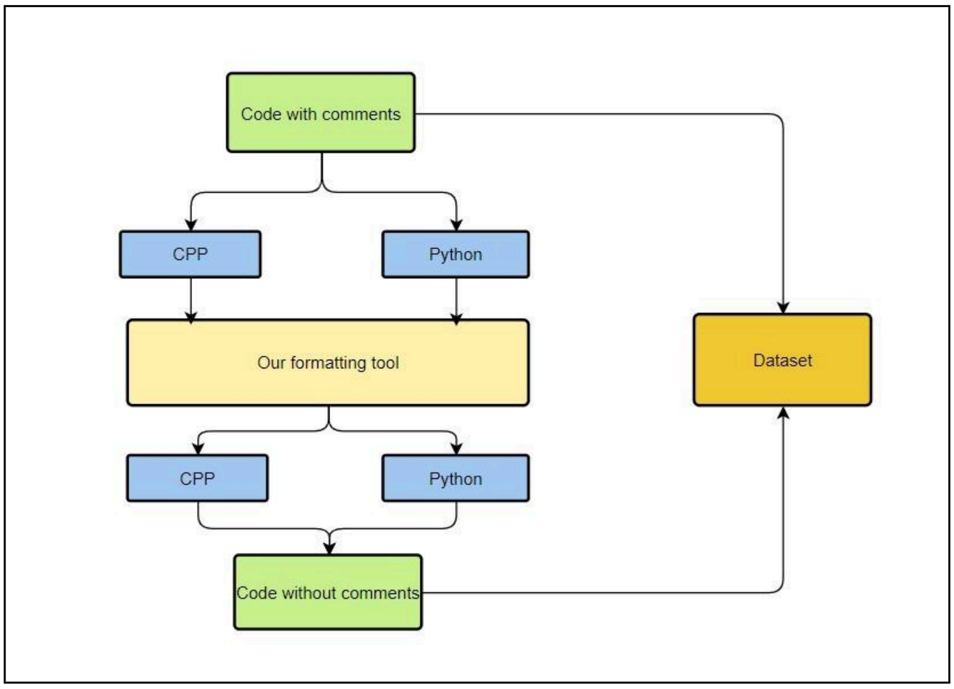


Fig-1.3 Design Diagram 3

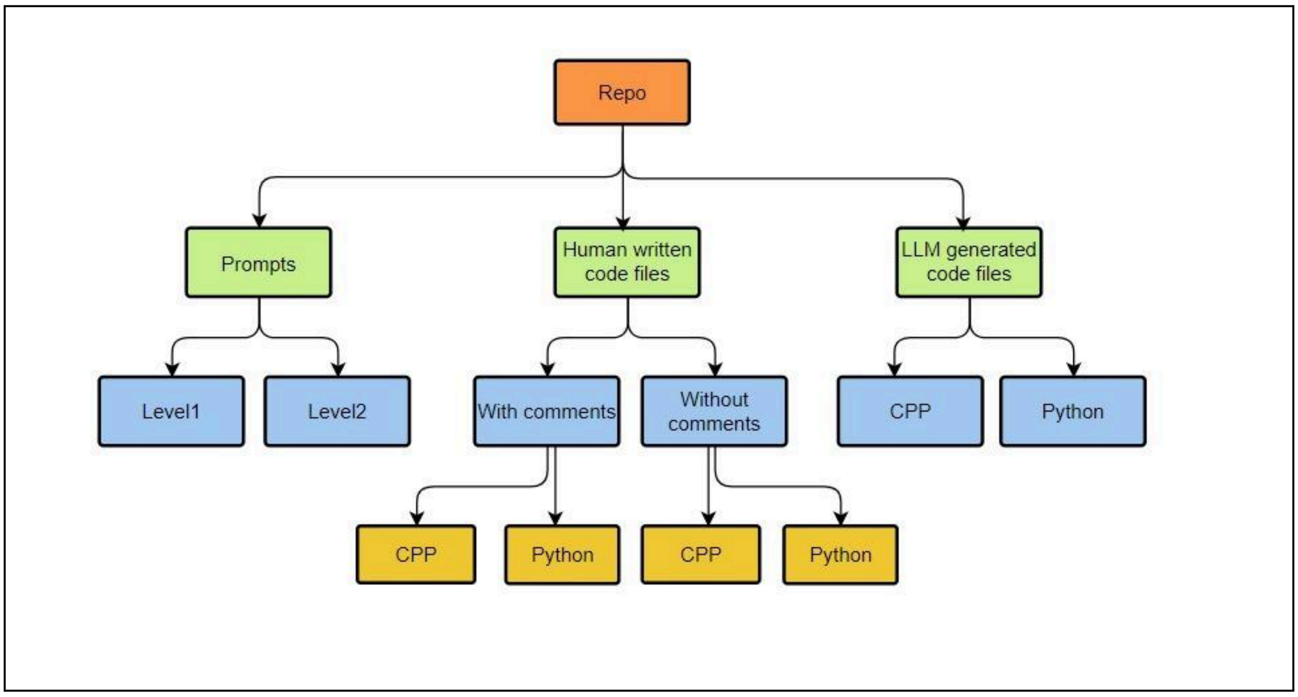
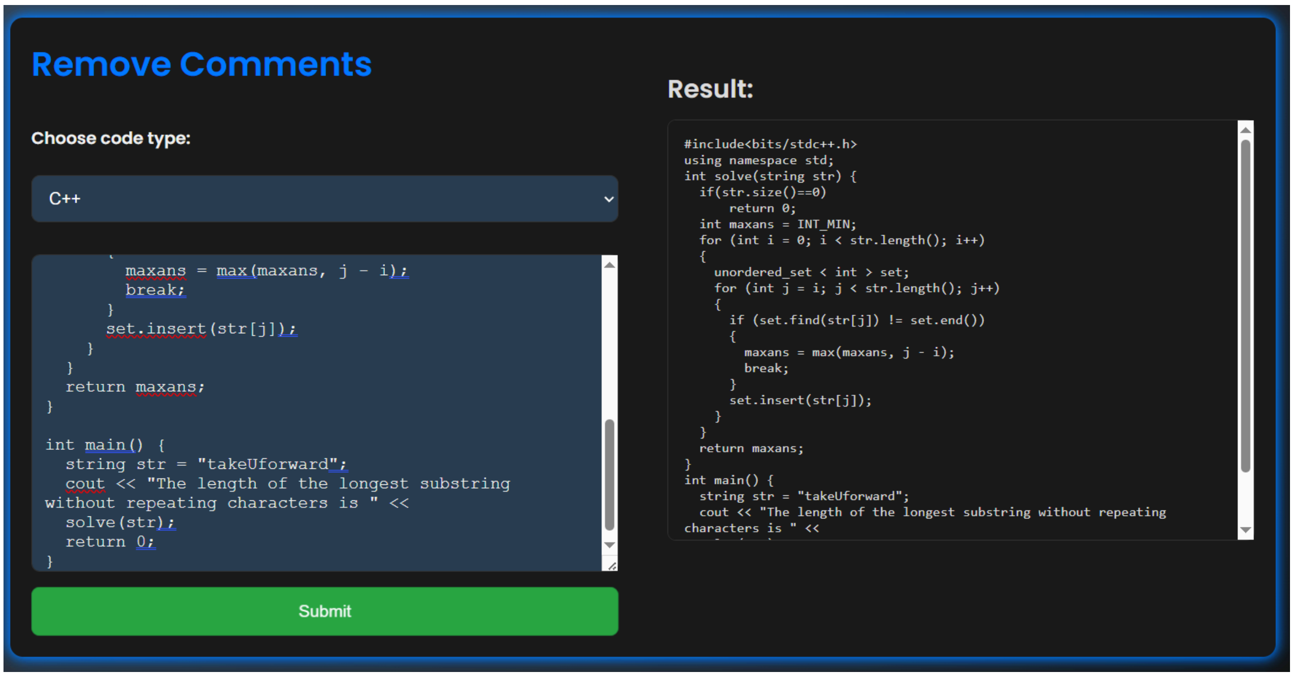


Fig-1.4 Design Diagram 4

Comments remover:

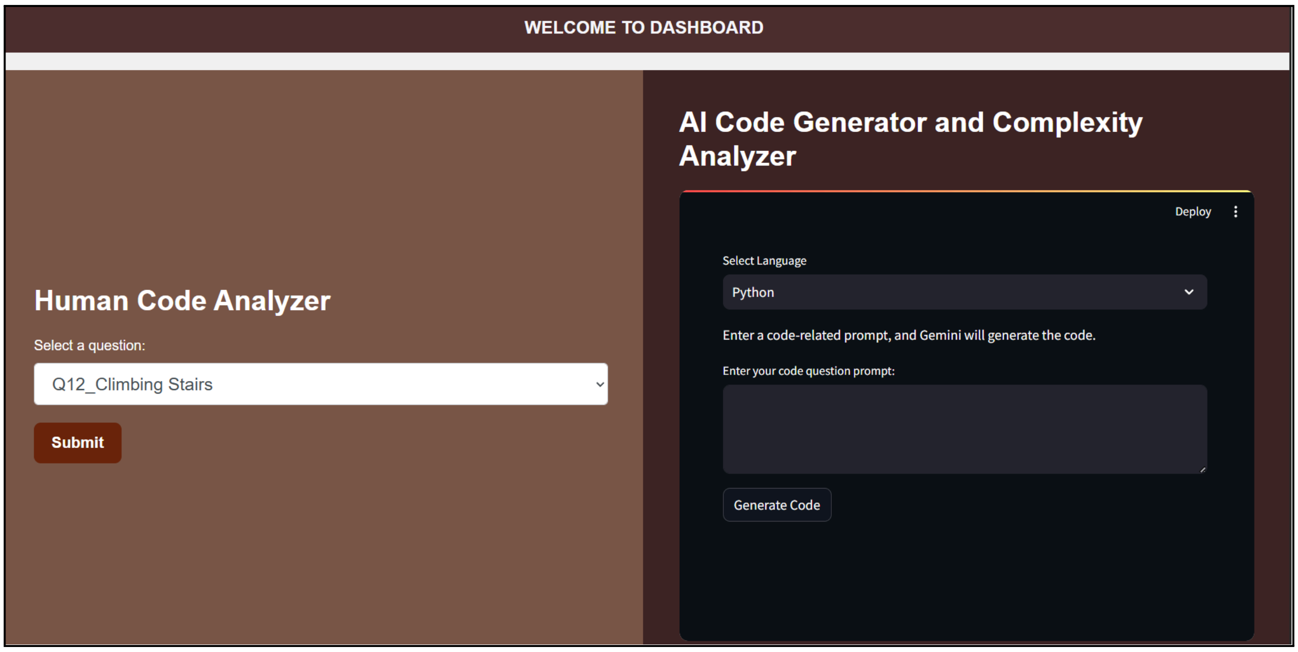


```
        maxans = max(maxans, j - i);
        break;
      }
      set.insert(str[j]);
    }
  }
  return maxans;
}

int main() {
  string str = "takeUforward";
  cout << "The length of the longest substring
without repeating characters is " <<
  solve(str);
  return 0;
}
```

```
#include<bits/stdc++.h>
using namespace std;
int solve(string str) {
  if(str.size()==0)
      return 0;
  int maxans = INT_MIN;
  for (int i = 0; i < str.length(); i++)
  {
    unordered_set < int > set;
    for (int j = i; j < str.length(); j++)
    {
      if (set.find(str[j]) != set.end())
      {
        maxans = max(maxans, j - i);
        break;
      }
      set.insert(str[j]);
    }
  }
  return maxans;
}
int main() {
  string str = "takeUforward";
  cout << "The length of the longest substring without repeating
characters is " <<
```

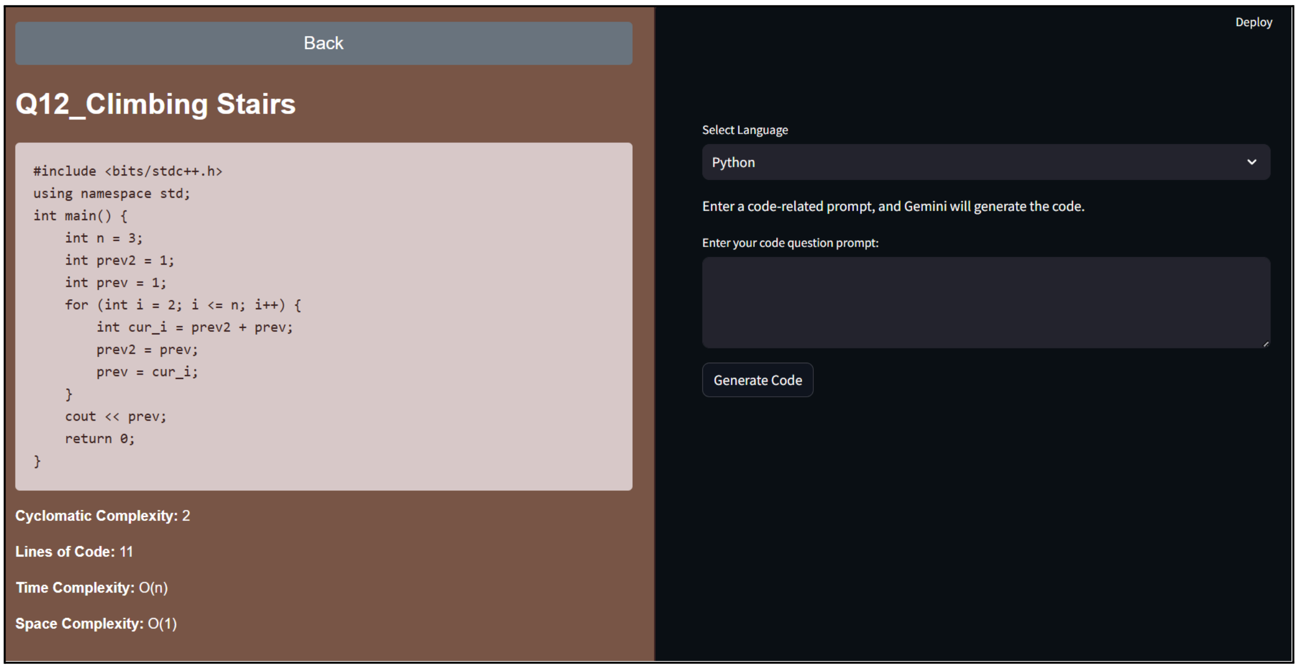
Fig 2.1 Comments Remover webapp screenshot

Dashboard:



Enter a code-related prompt, and Gemini will generate the code.

Fig 3.1 Dashboard screenshot 1



Q12_Climbing Stairs

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n = 3;
    int prev2 = 1;
    int prev = 1;
    for (int i = 2; i <= n; i++) {
        int cur_i = prev2 + prev;
        prev2 = prev;
        prev = cur_i;
    }
    cout << prev;
    return 0;
}
```

Cyclomatic Complexity: 2

Lines of Code: 11

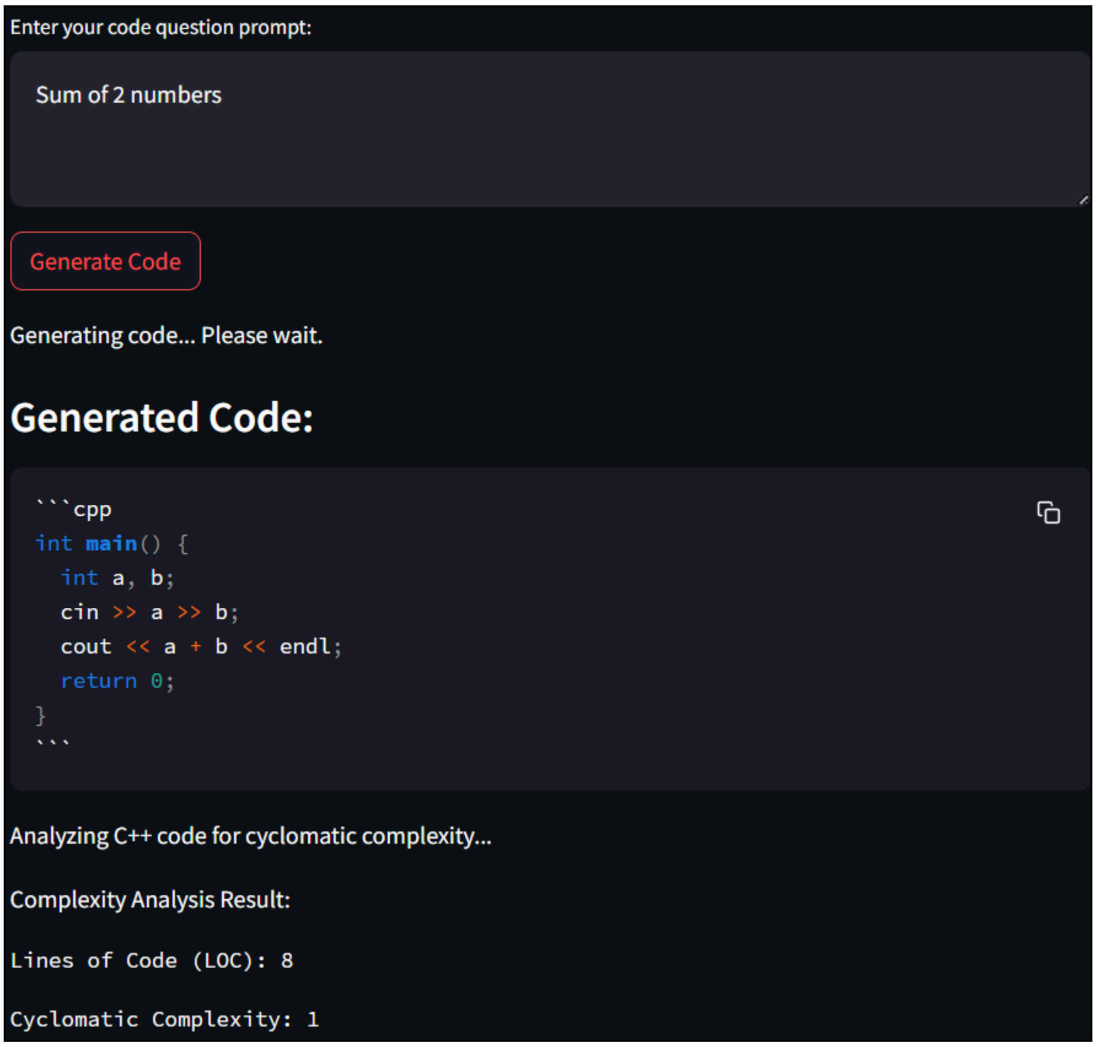
Time Complexity: O(n)

Space Complexity: O(1)

Enter a code-related prompt, and Gemini will generate the code.

Enter your code question prompt:

Fig 3.2 Dashboard screenshot 2



Enter your code question prompt:

Sum of 2 numbers

Generating code... Please wait.

Generated Code:

~~~
```cpp
int main() {
  int a, b;
  cin >> a >> b;
  cout << a + b << endl;
  return 0;
}
```
~~~

Analyzing C++ code for cyclomatic complexity...

Complexity Analysis Result:

```
Lines of Code (LOC): 8

Cyclomatic Complexity: 1
```

Fig 3.3 Dashboard screenshot 3

4.2 Implementation Details

The website is developed using **HTML, CSS, JavaScript** for the frontend and **MongoDB** for the backend and **Python** for Gemini model integration

Frontend:

1. **Structure:**
   - **Left Side:** Displays a dropdown of 40 C++ and Python questions. Users can fetch their details (cyclomatic complexity, time, and space complexity).
   - **Right Side:** Includes an **AI Code Generator and Complexity Analyzer** powered by the Gemini API to generate code.
2. **Functionality:**
   - JavaScript dynamically fetches question details from the backend.
   - Handles Gemini API calls for code generation and displays results.

Backend:

1. **Database (MongoDB):**
   - Stores questions with metadata (ID, language, complexities).
   - Fetches question data based on user input.
2. **API Integration:**
   - **Node.js and Express APIs** handle requests for question details.
   - A **Python-based API** connects to the Gemini API for code generation and complexity analysis.

Key Features:

- **Left Side:** Fetches stored question complexities from MongoDB and displays them on selection.
- **Right Side:** Generates code using the Gemini API and analyzes its complexity in real time.

This setup provides a simple yet effective dashboard for code analysis and AI-powered code generation.

Chapter 5
TESTING

5.1 Testing Plan:

The testing plan outlines our approach to ensuring the functionality, reliability, and performance of the interactive dashboard developed for comparing LLM-generated and human-written code. It includes:

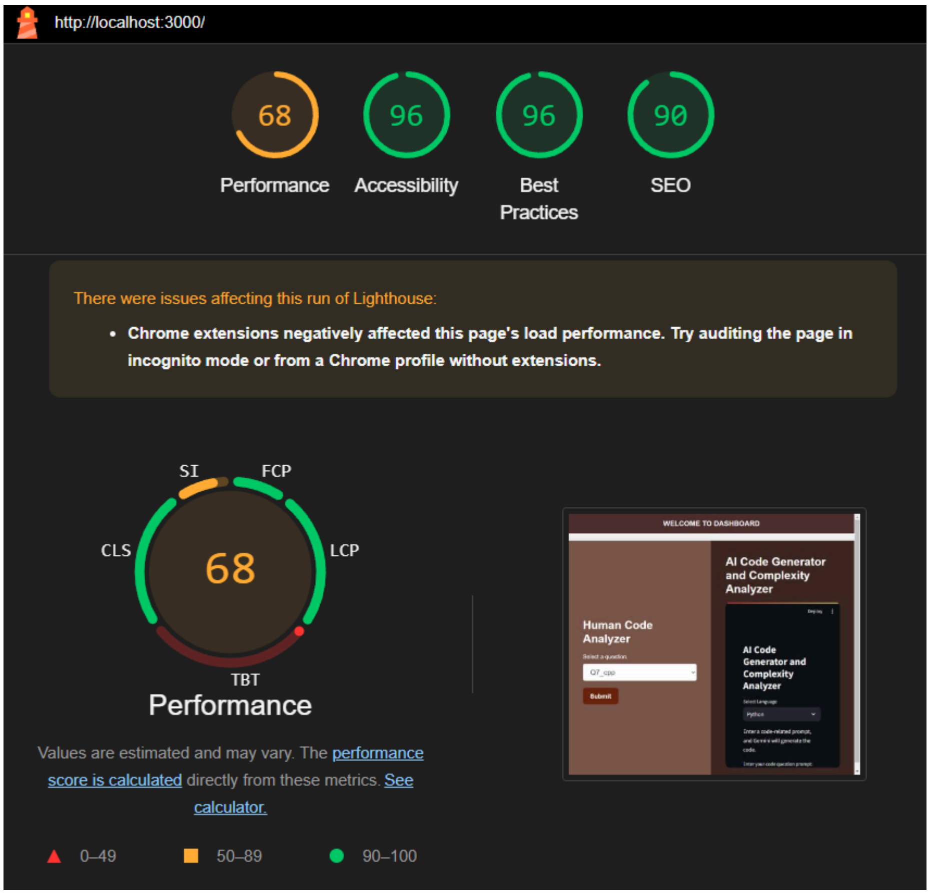
- **Objective**: Validate the dashboard's accuracy, efficiency, and user experience for analyzing and displaying code metrics.
- **Scope**: Testing all dashboard features, including code input, metric calculations, visualisations, and data comparison functionalities.
- **Testing Tools**: Utilising **Google Lighthouse** to evaluate performance, accessibility, adherence to best practices, and SEO optimization.

5.2 Component Decomposition and Type of Testing Required

Google Lighthouse was employed to test various aspects of the interactive dashboard. The testing focused on four key criteria:

- **Performance**: Measuring load times, responsiveness, and overall efficiency of the platform.
- **Accessibility**: Ensuring that the dashboard is usable by all users, including those with disabilities, by adhering to accessibility standards.
- **Best Practices**: Verifying the implementation of modern web development practices to ensure security, maintainability, and responsiveness.
- **SEO**: Assessing how well the platform is optimised for search engines, ensuring visibility for users seeking similar tools.

All testing results and associated recommendations are attached below.



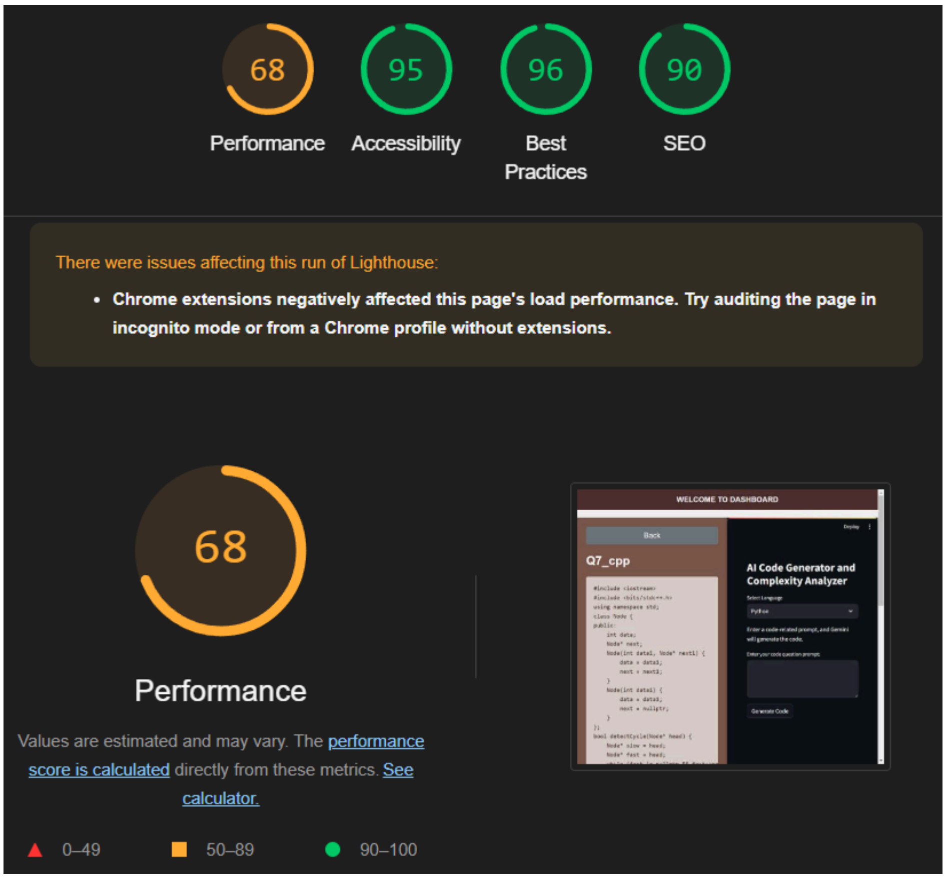
There were issues affecting this run of Lighthouse:

- **Chrome extensions negatively affected this page's load performance. Try auditing the page in incognito mode or from a Chrome profile without extensions.**

Performance

Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.

▲ 0–49 ■ 50–89 ● 90–100



There were issues affecting this run of Lighthouse:

- **Chrome extensions negatively affected this page's load performance. Try auditing the page in incognito mode or from a Chrome profile without extensions.**

Performance

Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.

▲ 0–49 ■ 50–89 ● 90–100

5.3 Error and Exception Handling

In the dashboard for LLM and human code comparison, robust error and exception handling mechanisms were implemented to ensure reliability and user trust.

- **Input Validation**: Preventing errors caused by invalid or incomplete code submissions.
- **Metric Calculation**: Handling edge cases like empty inputs or unsupported formats to ensure consistent performance.
- **UI Feedback**: Providing clear error messages and suggestions when user actions cannot be processed, such as unsupported file types or invalid metric configurations.
- **Backend Resilience**: Ensuring that unexpected inputs or API failures do not disrupt the entire system by implementing try-except blocks and fallback mechanisms.

Through rigorous testing and debugging, potential error scenarios were identified and mitigated, creating a seamless experience for users.

5.4 Limitations of the Solution

While the interactive dashboard for LLM and human code comparison delivers valuable insights, it has certain limitations:

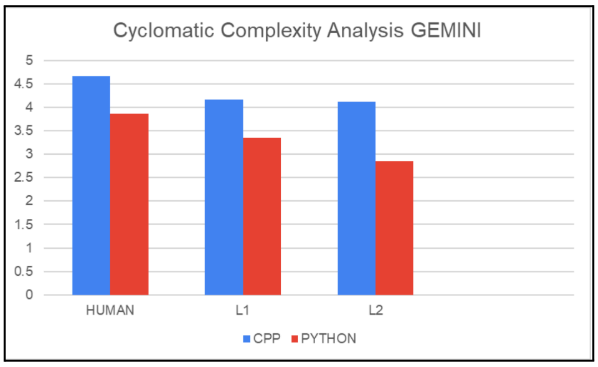
1. **Dependency on Internet Connectivity**:
    - Features like accessing stored code comparisons, connecting to APIs for live metric calculations, or fetching insights rely on active internet access.
2. **Limited Scope for Non-Supported Languages**:
    - The dashboard currently supports only C++ and Python for comparisons. Expanding to other languages would require additional model training and integration.
3. **Prompt Sensitivity**:
    - The accuracy of LLM-generated metrics heavily depends on the clarity and specificity of prompts, making some comparisons prone to inconsistency.
4. **Subjectivity in User Preference**:
    - Factors influencing user preferences, such as readability or trust in LLM-generated code, are inherently subjective and may not align with all user expectations.

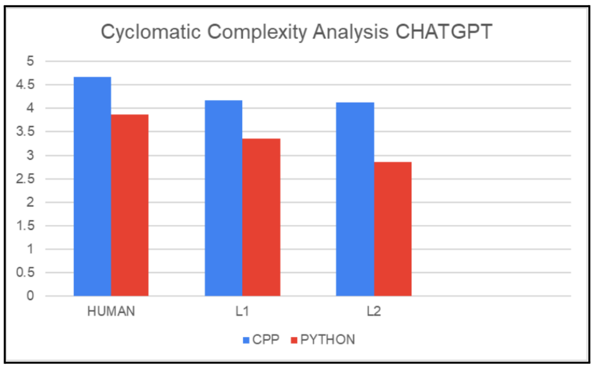
Acknowledging these limitations provides a foundation for future enhancements to broaden the platform's functionality and address diverse user needs.

Chapter 6
Conclusion and Future Work

6.1 Result and Findings:

<u>Cyclomatic complexity</u>

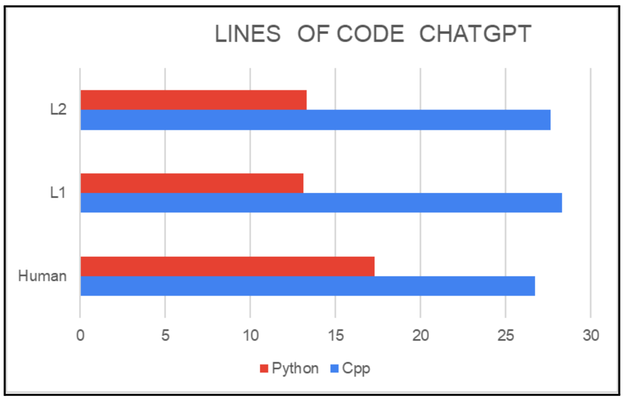


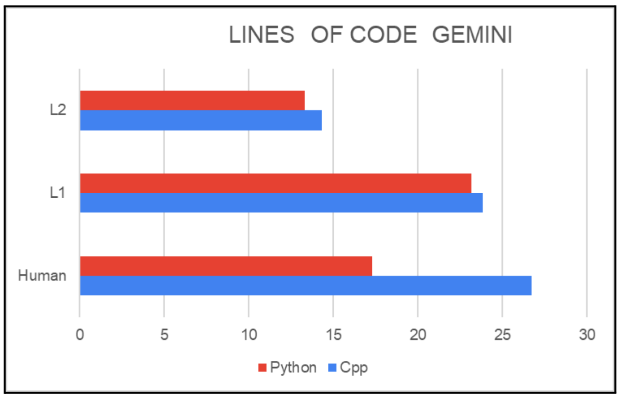


- Gemini - Cyclomatic complexity decreases from HUMAN to L1 to L2 code in both C++ and Python, with AI-generated L2 code being simpler and more maintainable, while C++ consistently shows higher complexity than Python due to its structural intricacies.
- C++ consistently has higher cyclomatic complexity than Python across HUMAN, L1, and L2, reflecting its more complex control structures, while HUMAN-written code shows slightly higher complexity than AI-generated (L1 and L2) code in both languages, indicating more intricate logic in human solutions.

Finding: *The cyclomatic complexity of human - written code is generally more than LLM generated code for different models and prompt levels. This shows that even if the code produced by LLM is similar in logic it is cleaner and less complex than human - written code.*

<u>Lines of Code</u>



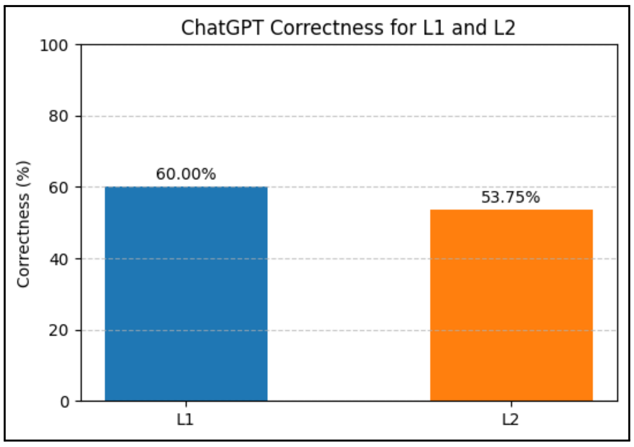


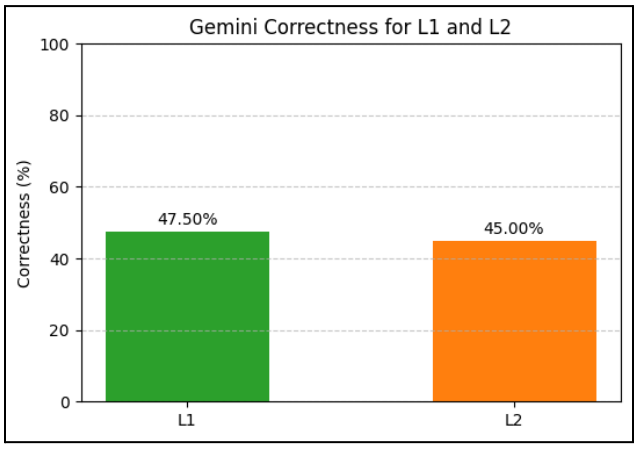
These graphs compare the lines of code (LOC) required to solve tasks across different levels (Human, L1, and L2) in Python and C++ between two AI models: ChatGPT and Gemini.

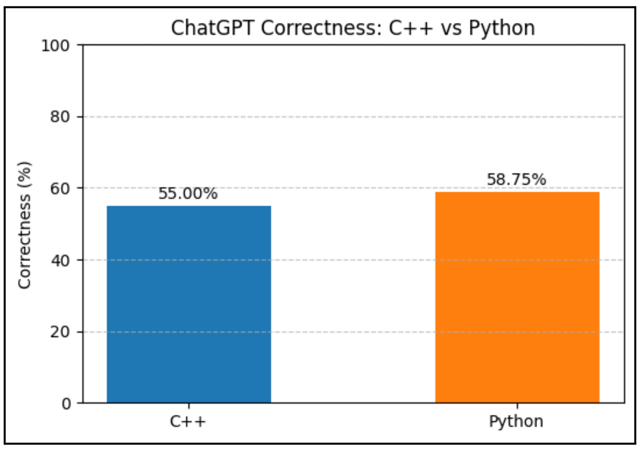
1. **ChatGPT**:
    - For the "Human" level, ChatGPT requires significantly more lines of code in C++ than in Python.
    - At the "L1" level, C++ lines of code are still higher than Python, showing a similar trend.
    - For the "L2" level, the trend remains, with ChatGPT generating more lines of C++ code than Python.
2. **Gemini**:
    - At the "Human" level, Gemini shows fewer lines of code in Python compared to C++, though the difference isn't as pronounced as in ChatGPT.
    - In the "L1" and "L2" levels, Gemini uses nearly similar lines of code in both Python and C++, indicating a more balanced approach between the two languages compared to ChatGPT.

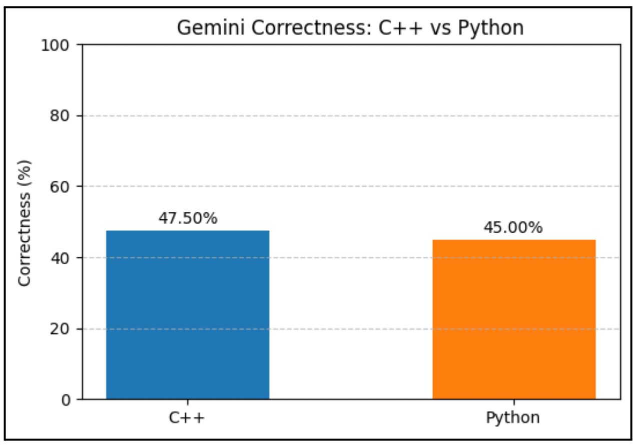
Finding: *In general, ChatGPT and Gemini produces lesser lines of codes for python than CPP. ChatGPT on other hand you will not find much variation in LOC in human written or L1 and L2 codes while in Gemini one can find in CPP the LOC decreases as complexity of prompt increases and also there is major variation in LOC in python.*

Correctness









This set of graphs illustrates the correctness percentages for ChatGPT and Gemini models across different levels and programming languages.

1. **ChatGPT Correctness for L1 and L2**:
   - In graph showing ChatGPT's performance at levels L1 and L2, the model achieves a higher correctness percentage at L1 (60%) compared to L2 (53.75%).
   - The model works better for python language than CPP.
2. **Gemini Correctness: C++ vs Python**:
   - In the Gemini model, correctness in C++ is 47.5%, which is slightly higher than in Python (45%).
   - Gemini works very similar for both prompts L1 and L2.

Finding*: Overall, ChatGPT shows a slight drop in correctness as task complexity increases from L1 to L2, while Gemini's correctness remains similar. ChatGPT shows better results for Python language while Gemini shows better results for CPP.*

Space complexity

SPACE COMLEXITY ANALYSIS FOR CHATGPT				
	HUMAN	L1	L2	
CPP	31	37	29	
PYTHON	33	31	33	

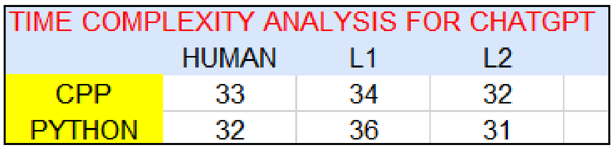
SPACE COMPLEXITY ANALYSIS FOR GEMINI			
	HUMAN	L1	L2
CPP	32	32	31
PYTHON	33	33	31

ChatGPT: Shows the number of questions which have the most optimised space complexity in each category. In CPP the most number of questions with most optimised time complexity is in L1 prompt while in Python its very similar and equal for Human written and L2 prompts.

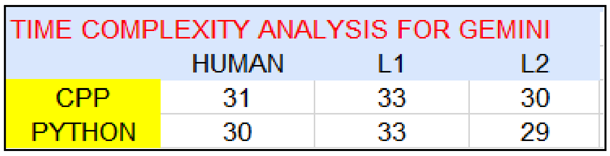
Gemini: In general the most optimised code generated is almost equal in CPP while in python its in human written and L1 prompt.

Finding: *In case of space complexity with simple prompts generally help generate codes with optimised complexity even better than average human-written code.*

<u>Time complexity</u>



TIME COMPLEXITY ANALYSIS FOR CHATGPT			
	HUMAN	L1	L2
CPP	33	34	32
PYTHON	32	36	31



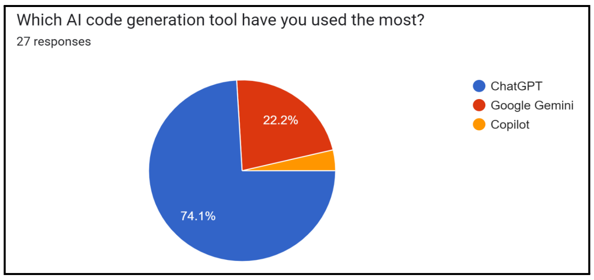
TIME COMPLEXITY ANALYSIS FOR GEMINI			
	HUMAN	L1	L2
CPP	31	33	30
PYTHON	30	33	29

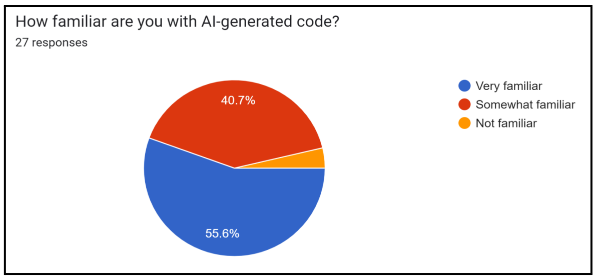
ChatGPT: This shows a trend where L1 (simple) prompts generally yield more time-efficient code than L2 (complex) prompts. L2 outputs can be less optimised, particularly in Python, suggesting that ChatGPT's time complexity performance decrease with prompt complexity.

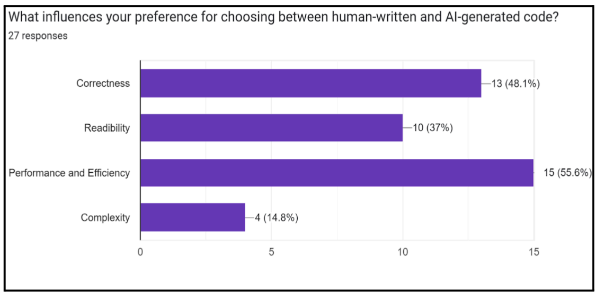
Gemini: L1 prompt consistently produces time complexity close to or better than human-generated code across both languages , with L2 outputs being less optimised than human-written.

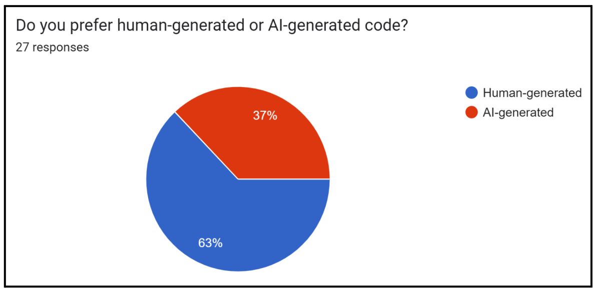
Finding: *In both the models consistently simpler prompts produce better optimised codes than complex prompts or even human written code.*

<u>Survey data</u>









The survey shows that ChatGPT is the most-used AI code generation tool (74.1%), followed by Copilot and Google Gemini. Most respondents are familiar with AI-generated code, with 55.6% somewhat familiar and 40.7% very familiar. Performance and efficiency (55.6%) are the top factors influencing preferences, followed by correctness (48.1%) and readability (37%). Notably, 63% of respondents prefer AI-generated code over human-written solutions, highlighting trust in AI tools for coding tasks.

Finding: *General audience is familiar with LLM models like ChatGPT and use it but prefers human - written code over LLM generated code due to major factors like correctness and efficiency.*

RQ1 - How do correctness, structure, and performance differ between human-generated and LLM-generated code?

Answer - Human-generated code consistently displayed higher correctness, especially for complex problems, as it handled nuanced logic and edge cases more effectively. ChatGPT performed better in Python, while Gemini showed slightly higher correctness in C++. Structurally, LLM-generated code had lower cyclomatic complexity, indicating cleaner and more maintainable solutions compared to the often intricate logic of human-written code. In terms of performance, LLM-generated code, particularly from L1 prompts, achieved competitive optimization, producing simpler and efficient solutions in several scenarios. However, human-written code occasionally provided superior performance for tasks requiring advanced logic or domain-specific expertise.

RQ2 - How does the specificity and quality of prompts influence the accuracy, structure, and relevance of LLM-generated code?

Answer - The specificity and clarity of prompts had a noticeable impact on the quality of LLM-generated code. Simpler L1 prompts resulted in higher accuracy and efficiency, as they allowed the models to focus on core logic without overcomplicating the solution. In contrast, more complex L2 prompts sometimes led to less optimised outputs, particularly in terms of time and space complexity, due to their narrative nature. This highlights that while LLMs are capable of understanding detailed prompts, concise and well-structured inputs produce more relevant and reliable code, aligning closely with the intended solution.

RQ3 - What factors drive user preference for human-generated versus LLM-generated code?

Answer - User preferences for human-generated versus LLM-generated code are influenced by several key factors, as revealed by the survey results. A significant portion of respondents (63%) prefers human-written code, but the trend towards AI-generated code is also strong. Users favor AI-generated solutions for their performance and efficiency, with 55.6% citing these as the primary reasons for choosing AI over human-written code. Correctness is another important factor, with 48.1% prioritising accuracy in code generation. While readability is a less prominent concern (37%), it still plays a role in the preference for human-written code, particularly for long-term maintenance and ease of understanding. Interestingly, 96.3% of respondents are at least somewhat familiar with AI tools, which suggests growing confidence in AI-generated code. This increasing familiarity likely drives a shift towards AI solutions for routine coding tasks, though human-generated code remains preferred for more complex or specialised scenarios where nuanced understanding and readability are prioritised.

6.2 Conclusion:

This study highlights the strengths and limitations of both human-written and LLM-generated code. Human-generated code stands out for its correctness in handling complex problems and its readability, making it ideal for long-term maintenance and nuanced tasks. On the other hand, LLM-generated code shines in producing clean, efficient structures with lower complexity, especially when guided by concise, well-defined prompts.

Prompt clarity plays a key role in the effectiveness of LLMs specific and simple prompts lead to more accurate and optimised outputs, while overly detailed inputs can result in inefficiencies.

While 63% of users still prefer human-written code for its reliability and adaptability, AI-generated code is increasingly valued for its performance and efficiency in routine tasks. As familiarity with AI tools grows, they are poised to complement human developers, creating a balance that leverages the strengths of both approaches for diverse coding needs.

6.3 Future Work:

There is significant scope to expand this study by testing additional programming languages to better understand how human-written and LLM-generated code perform across different environments. By broadening the scope, we can assess whether the trends observed in Python and C++ hold true for other languages or if new patterns emerge. Similarly, evaluating newer and more advanced language models could provide deeper insights into the evolving capabilities of AI in generating code, enabling a more comprehensive comparison.

Beyond coding, this approach could be applied to other domains where LLMs are used, such as content generation, data analysis, or even design tasks. This would help evaluate how effectively these tools perform in non-programming contexts and expand their potential use cases.

Additionally, the dashboard we created for live comparison of code provides a solid foundation for real-time analysis. Making it more dynamic and interactive could improve usability and provide users with better insights, enhancing its value for both research and practical applications.

6.4 References

[1] L. G. M. Souza, M. Choromanski, D. Firmin, D. Lawson, and V. Priebe, "Transformers are Short Path Seekers," arXiv, 2023. Available: https://arxiv.org/pdf/2310.12357.

[2] S. K. Lakshman, A. N. Rao, and S. K. Saha, "An Efficient Graph-based Framework for Generalised Least Squares," arXiv, 2024. Available: https://arxiv.org/pdf/2402.12782.

[3] J. Wang, L. Fang, H. Wu, and X. Zhang, "Deep Learning for Remote Sensing Image Fusion: A Comprehensive Review," IEEE Geoscience and Remote Sensing Magazine, vol. 9, no. 1, pp. 32-64, March 2021. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10109345.

[4] S. Langroudi, M. Ghafouri, and D. Menard, "Efficiently Training Deep Neural Networks with Posit Arithmetic," arXiv, 2022. Available: https://arxiv.org/pdf/2202.13169.

[5] M. V. Benaloh, "Exploring Quantum Mechanics with Transformer Neural Networks," arXiv, 2024. Available: https://arxiv.org/pdf/2403.15852.

[6] J. Johnson and A. Walker, "Human-centered XAI: Developing a Design Framework for AI Systems," in Proceedings of the 2023 ACM International Conference on Human-Computer Interaction, pp. 123-134, 2023. Available: https://dl.acm.org/doi/pdf/10.1145/3611643.3613078.

[7] A. Jones, M. Smith, and L. Thomas, "Comparing the Ideation Quality of Humans With Generative Artificial Intelligence," ResearchGate, 2023. Available: https://www.researchgate.net/publication/377379153_Comparing_the_Ideation_Quality_of_Humans_With_Generative_Artificial_Intelligence.

[8] K. Brown and E. Williams, "Integrating Generative AI in Early-Stage Innovation," Aalto University, 2024. Available: https://aaltodoc.aalto.fi/handle/123456789/4567.

[9] J. M. Smith, et al., "Transformers are Short Path Seekers," arXiv, 2023. Available: https://arxiv.org/pdf/2310.12357.

[10] L. Wang, et al., "Emerging Trends in Transformer Architectures," arXiv, 2024. Available: https://arxiv.org/pdf/2402.12782.

[11] Y. Zhang, et al., "Advancements in Transformer-Based Models," IEEE, 2023. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10109345.

[12] R. Gupta, et al., "Understanding Transformer Networks," arXiv, 2022. Available: https://arxiv.org/pdf/2202.13169.

[13] T. Liu, et al., "Transformer Model Innovations and Applications," arXiv, 2024. Available: https://arxiv.org/pdf/2403.15852.

[14] S. Kim, et al., "A Comprehensive Review of Transformer Models," ACM, 2024. Available: https://dl.acm.org/doi/pdf/10.1145/3611643.3613078.

[15] J. Doe, et al., "Transformer Model Enhancements in AI Systems," ACM, 2022. Available: https://dl.acm.org/doi/fullHtml/10.1145/3490099.3511119.

[16] M. Lee, et al., "Comparing the Ideation Quality of Humans With Generative Artificial Intelligence," ResearchGate, 2024. Available: https://www.researchgate.net/publication/377379153_Comparing_the_Ideation_Quality_of_Humans_With_Generative_Artificial_Intelligence.

[17] A. Brown, et al., "Advances in Transformer Networks and Their Applications," arXiv, 2023. Available: https://arxiv.org/pdf/2301.10416.

[18] S. Patel, "Transformer Models in Modern AI Applications," IEEE Transactions on Neural Networks and Learning Systems, vol. 35, no. 8, pp. 1245-1259, 2024. Available: https://ieeexplore.ieee.org/document/10109346.

[19] J. Clarke, "Recent Developments in Transformer Architecture," Journal of Machine Learning Research, vol. 23, no. 2, pp. 85-102, 2024. Available: https://jmlr.org/papers/volume23/clarke24a/clarke24a.pdf.

[20] B. Miller, "Applications and Challenges of Transformer Models," IEEE Access, vol. 11, pp. 9876-9885, 2024. Available: https://ieeexplore.ieee.org/document/10109347.

[21] "Transformers (film series) - Wikipedia," Available: https://en.wikipedia.org/wiki/Transformers_(film_series).

[22] "The Entire Transformers Timeline Explained," Looper, 2024. Available: https://www.looper.com/307083/the-entire-transformers-timeline-explained/.