AI-Assisted Code Generation Tools: A New Frontier in Software Development

by

Zaima Mashiat Nabi
23241120
Ibtisum Jaman
24141204
Utshob Bose
21201782
Tasnia Taslim Hossain
23241123
Dipra Kamal
21201458

A thesis submitted to the Department of Computer Science and Engineering in partial fulfillment of the requirements for the degree of
B.Sc. in Computer Science and Engineering

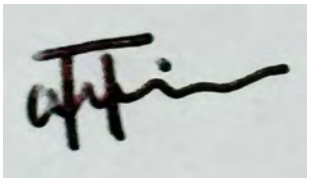
Department of Computer Science and Engineering
Brac University
October, 2025

Declaration

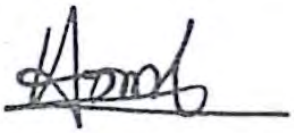
It is hereby declared that

1. The thesis submitted is my/our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

Zaima Mashiat Nabi
23241120



Tasnia Taslim Hossain
23241123



Ibtisum Jaman
24141204

Utshob Bose
21201782

Dipra Kamal
21201458

Approval

The thesis/project titled "AI-Assisted Code Generation Tools: A New Frontier in Software Development" submitted by

1. Zaima Mashiat Nabi (23241120)
2. Ibtisum Jaman (24141204)
3. Utshob Bose (21201782)
4. Tasnia Taslim Hossain (23241123)
5. Dipra Kamal (21201458)

of Fall,2021 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science in Summer 2025.

Examining Committee:

Supervisor:
(Member)

Md. Aquib Azmain

Lecturer
Department of Computer Science and Engineering
Brac University

Thesis Coordinator:
(Member)

Rafeed Rahman

Senior Lecturer
Department of Computer Science and Engineering
Brac University

Head of Department:
(Chair)

Sadia Hamid Kazi

Chairperson
Department of Computer Science and Engineering
Brac University

Abstract

Through various developments in Artificial Intelligence code generation, it has greatly influenced the development of codebases and software. This has increased efficiency and led to the development of conventional programs and practices. This thesis emphasises these AI tools and their impact on the developer's experience in real-world applications, and also addresses critical gaps in the existing literature. Through thorough analysis, this paper demonstrates that AI code generation can enhance learning, efficiency and job satisfaction among young developers. However, studies
show that these AI-generated codes may be inaccurate at times and also contain code smells, which can turn into a hassle for developers to debug rather than a blessing. Therefore, a more accessible pricing strategy and improved reliability should be implemented in these tools. Our research includes three distinct pipelines that help automate code debugging, followed by automated unit tests after the code has been fixed. These pipelines include Prompt Mechanism, which uses Abstract Syntax Trees (AST) and Pylint for detecting errors and with the help of a generative AI it fixes the code accordingly to the detected error, a Retrieval-Augmented Generation (RAG) pipeline leveraging official Python documentation for context-aware corrections and a Fine-tuning approach using CodeT5 trained on samples from the TSSB-3M dataset. Afterwards, the accuracies of these pipelines were evaluated by examining syntax, runtime, and logical errors. The fine-tuning and RAG pipeline had similar accuracy to one another, while the prompt mechanism had lower accuracy. By analysing code-practising factors such as usefulness, efficiency, and adaptability across various tools and techniques used in Artificial Intelligence, the study examines both the advantages and negative aspects of using AI tools in large-scale industrial development. While AI tools can be an excellent choice for standard coding tasks, human software quality assurance is also necessary to maintain the accuracy of the generated code. This thesis research helps to make use of responsible AI practices and addresses the concerns regarding the security and the quality of the codes, and also makes sure to include the ethical implications through a clear explanation, which would help the developers to learn and unit tests, which would require less human oversight for quality assurance. This thesis demonstrates that a multiple pipeline approach can be beneficial in reducing errors, increasing accuracy, and enhancing robustness in a developer's code.

Keywords: AI-assisted code generation, Large Language Models, Natural Language Processing, Software Development, Code debugging, Automated testing, Retrieval-Augmented Generation

Acknowledgement

Firstly, we want to thank the great God Almighty for helping us through this Thesis without any obstacles and for helping us finish our paper within the deadline. Furthermore, we would also like to thank Md. Aquib Azmain, sir, for guiding us and believing in us, for which we could make this happen. His valuable advice and support have truly helped us to accomplish our goals, for which we are grateful.

We would also like to thank all the fellow researchers for their help in finding a huge amount of online resources. We'd also like to mention our family and friends who believed in us and supported us through this journey. And lastly, we would like to thank BRAC University for all its support and resources, which have constantly helped us through this process.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The introduction of artificial intelligence (AI) into software development has resulted in a significant change that has put traditional methods utilised in the industry to the test [39]. The concept of incorporating artificial intelligence into software development has existed since the late 20th century [37]. In the years that followed, machine learning algorithms that used human experience were created, allowing for the ability to adapt to massive volumes of data. AI tools use powerful machine learning algorithms [18] to provide code snippets and offer help with complex tasks. The behaviour and overall nature of coding have increasingly shifted with technological advancement. This, in practice, has not just brought in enhancements to the software development process[16], but also drawbacks, where security and privacy can potentially be threatened[48]. Additionally, to improve the efficiency of code runs, version control and code reviews must be done. On the other hand, it's quite essential to test for the sustainability of the AI code generators and make sure that they are a reliable source of production. Also, we have to examine the functions of generative AI and look into its errors, restrictions, perform unit tests and get rid of code smells. There have been reports of several issues due to the usage of Artificial Intelligence for code generation, but there are a few positive aspects as well. The technical considerations, like copying and the validity of the produced code, are a bit difficult to tackle while using AI. This is a relevant issue since these codes are mostly used publicly, and therefore, a lot of times, copyright issues take place [36]. Furthermore, there is a question about the reliability of the codes, which may make the developers less skilled while comprehending them. On the brighter side, it's safe to say that Artificial Intelligence can work on its own shortcomings and fix them very quickly as well. This research aims to go through the limitations and shortcomings of generative AI and create a solution for such issues. This research will further investigate various types of LLMs and their processes in order to improve their codebases and increase the efficiency of these codes [49]. All of these would help developers become more productive and efficient.
Although artificial intelligence has made it possible to solve issues automatically, it is still highly important to have a human touch to determine whether or not there are any flaws or contradictions in the findings. The AI models do a comprehensive examination of the data gathering to ensure quality. After that, they run the essential coding tests to evaluate their problem-solving abilities. To reduce the number of mistakes, the findings, runtime challenges, and design factors are all examined. For example, interviews and design probe sessions with developers are meant to be used to get an understanding of the fundamental structure of the LLMs, as well as to identify faults and weaknesses and make changes. In general, businesses often utilise technologies that are supported by artificial intelligence. On the other hand, further effort is needed to address issues related to complexity and the

production of secure code. Researchers will concentrate on improving the usefulness of codes. When users face the real-world difficulties that must be addressed in the developing technology sector, they become more high-yielding and innovative.

1.2 Problem Statement

Utilisation of AI tools for producing sensitive or proprietary code can potentially create concerns regarding ethical, security and intellectual property implications that need to be acknowledged and carefully considered in software development. Integrating AI-assisted code generation can not only bring opportunities, but also bring significant challenges, as it must properly align with code review, version control and cooperation in order to bring validity and efficiency. Despite significant advancements, misuse of APIs or minimal designs can cause the generated code to encounter issues. Thus, it interferes with practical usability. Future Research focuses on the improvement of AI-generated code quality by intensifying contingent understanding, this ensures minimising errors and compliance with standards. Therefore, to approach these challenges, the following Research Questions will be answered in this study[34]:

1.3 Research Questions

1. What is the elevated implication of using AI tools for generating branded or sensitive code?

2. How do AI-assisted code generation tools combine with existing software development workflows?

3. What are the prospective future research directions for upgrading the quality of code generated by LLMs, particularly in the context of enhancing the practical usability of the generated code and addressing API misuse?

1.4 Objective

The main objective of this paper is to analyse the effectiveness and accuracy of AI-assisted Code generation tools in resolving programming challenges to face real-world applications by re-recognising models' limitations, which will generate reliable, maintainable code without faults. The aim of this paper includes:

- To approach the usability and certainty by evaluating the effectiveness of AI-assisted tools like GitHub Copilot, Amazon CodeWhisperer, and OpenAI ChatGPT in software development advancements for both beginner and experienced software developers [30].

- Explore the prospect of pre-trained AI models like GPT-4, Llama 3, and Codex with specified datasets and different programming languages such as Java and Python to further improve the accuracy of generated valid code.

- Evaluate a study which will primarily compare AI-generated code and human-written raw code, emphasising quality standards such as maintainability, effective execution, and readability in complex software development.

- Explore the study of use cases in task management, pipelines, continuous deployment, and continuous integration to evaluate the contribution of AI-assisted programming tools in agile and traditional software development processes.

- To come up with the investigation of efficient prompt engineering tactics, which will guide AI models to generate more accurate and structured code as outputs.
- Investigate the ability of how AI models can both detect and fix errors by debugging incorrect codes from open source datasets (e.g., Defects4J, QuixBugs).
- Arrange surveys and experiments from users to estimate trustworthiness in AI-generated code among beginner and experienced programmers.

1.5 Methodology in Brief

This study brings a comparative analysis to assess Artificial Intelligence Code generation compared to human-written code in software development. aDatasetsmay include surveys, various coding experiments, and public repositories. The coding experiment solves complex questions using AI-generated tools and human developers. The assessment requirement requires code efficiency, ethical concerns and the frequency of errors. Statistical approaches like regression analysis or t-tests are primarily applied to determine the comparison between the two codes.

1.6 Scopes and Challenges

Improving code efficiency, saving long hours of code debugging and recommending different efficient codes, AI-assisted code generation tools have transformed software development into a whole new innovation. Now, software developers make the development more accessible with the help of these code generation tools. oHowever Human involvement still remains essential to mitigate challenges of AI integration in software development, while also increasing benefits with the AI-assisted tools. The following sections discuss several scopes and challenges in detail.

Scopes

- Using AI-assisted tools simplifies programming codes for novice developers by suggesting step-by-step guides, different code examples with explanations, which makes it easier for them to understand, apply and practice different coding techniques. These code generation tools reduce the complexity of first entering into software development by offering an understandable learning environment.
- AI-assisted code generation tools with GUI support can automatically identify code bugs, incorrect indentations, fault code patterns and inefficient code. These code generation tools suggest a quick fix, recommending different ways of solving the errors, which enhances the reliability of the code and saves time.
- With the help of AI-assisted tools, generated comments and summaries contain the documentation, which decreases the physical effort in code maintenance.

Challenges

- Generated codes by AI-assisted tools may initiate security vulnerabilities. Because of these flaws, these codes need to be thoroughly tested and validated before being used in any software development. Vulnerabilities cause consequences for the integrity of different software if not properly tested.

- Too much dependency on these AI tools can decrease a developer's potential for various problem-solving, logical decision-making and understanding complex coding techniques.
- Generated codes need to be thoroughly verified by the developers because these codes can be ineffective or unessential to the specific requirements of the projects.

1.7 Key Learnings and Insights

Through this research, several key learnings and insights have emerged regarding the integration of AI in software development:

- **Balanced Productivity Gains**: The Artificial Intelligence Code generation tools help developers automate tasks and also provide instant suggestions for code bases. However, if human oversight is included, the effect of this system is maximised and ensures more accuracy [60].
- **Security and Ethical Priorities**: The risk of security vulnerabilities along with ethical issues must be considered while using such tools, and proper guidelines should be maintained to avoid such issues [56].
- **Prompt Engineering as a Skill**: A Proper prompt mechanism can result in efficient and accurate code suggestions by the A tools. Structured and few-shot prompting can be implemented, which would reduce error by 40% and ensure the quality is much higher [63].
- **Human-AI Synergy**: Studies show that junior developers often benefit from AI code gen tools since they cannot only develop code but also learn the processes. The efore, overall, AI will not replace human programmers, but instea,d it will increase their efficiency. Also, proper human oversight is required for large-scale development to avoid any sort of mistake [61].

Chapter 2

Literature Review

2.1 Preliminaries

Incorporating Artificial Intelligence (AI) in the software development cycle has dramatically altered the traditional code generation. The AI-assisted code generation tools have been using modern machine learning models like ChatGPT and Copilot to enhance various software features and reduce the number of bugs. This literature review depicts the futuristic advancements, the work plans, and the issues linked to code generation using Artificial Intelligence.

2.2 Review of Existing Research

Poldrack et al. in their research paper discovered the abilities of GPT-4 in automated code generation, focusing on its performance in test generation, interactive coding and refactoring [23]. They reported that 72% of the cases generated code usable by GPT-4, and 37.5% of tasks were resolved on the first attempt. The limitations were noted, however, specifically in operating tasks that were domain-specific and mathematically precise, where often the incorrect outputs or "hallucinations" were produced by the model [38].
Poldrack et al. also inspected the potential for refactoring code by GPT-4. Significant improvements were noticed in acceptability and cohesion to coding standards by using a dataset of Python scripts. In an example, it was seen that there was a substantial decrease in Flake8 error messages per line in a refactored code. Even with this progress, the authors observed that the improvements of GPT-4 did not mention all errors while prioritizing human Oversight and complementary tools. During test generation, Poldrack et al. also discovered that GPT-4 had low functional success rates with as much as 45% passed tests, but it had achieved high test coverage (average of 94%). There were common issues that included a mismatch between actual and expected outputs, test validation and underscoring the importance of human expertise in debugging. In conclusion, the authors stated that even though GPT-4 heightens productivity in regular coding tasks, it also has its limitations, which consist of depending on functional inaccuracies and outdated data, while highlighting the frequent need for human programmers. Poldrack, Lu, and Beguš suggest that GPT-4 should improve interaction strategies for it to fully emerge its utility in software development, as well as integrating it with other tools.

Yang et al. conducted a complete survey that explores the serviceability of AI programming assistants, which focuses on the motivations and experiences of developers [41]. They found that the primary motivation for developers to use AI tools is that they speed up

their programming tasks and reduce keystrokes. However, this orderliness comes with many significant difficulties that include security risks, concerns about the code being correct and the need for human Oversight. Yang et al. also focus attention on the volatility of developers ' trust in I-generated code. Issues related to AI suggested debugging solutions were reported, even though the tools were appreciated for accelerating dull tasks and easing learning. Authors Huang and Marsden et al. mention that developers had to review and validate the generated code in their additional time, which often resulted in a decrease. The predicted regulatory gains in some scenarios. Furthermore, this paper questioned the integration and the implications of Artificial Intelligence code generation in large-scale projects. This research reflects how new developers are getting overly dependent on generative AI and how it's hampering the productivity of developers. It was also suggested that, along with efficiency, security, and the quality of the codes, must be taken care of as well by these LLM models. To conclude, this research shows the potential of the AI code assistants and how to make use of them so that they would not interfere with the integrity and the quality of a developer's code in a large-scale industry.

Mozannar et al. discussed the paid AI tools like Copilot and CodeWhisperer and researched their costs and user activities. These tools will be a great support by providing code suggestions and auto-completions, increasing the efficiency of the developers. However, these tools also have a few drawbacks, such as code dependencies, and they may change the integrity of the developer's code as well [43].
Authors distinguished that relying on AI-recommended code generation reduces a programmer's participation in difficult code knowledge and debugging tasks, often while being advantageous towards finishing coding tasks faster. Therefore, this transference elevates unease about long-term coding skill maintenance and possible ruination of a developer's programming proficiency. Moreover, the authors mentioned the calculative and financial cost associated with combining such systems into workflow developments, specifically mentioning the necessity for equal adaptation strategies. Mozannar et al. proposed multiple factors to limit these problems, which include nurturing an important assessment of AI code generation and a deep understanding of the tool's restraints. The significance of designing systems is prioritised in this study, which helps in debugging and enriching knowledge of the developer's effectiveness in general, without the need to disregard code quality or security.
Lastly, the research by Mozannar et al. depicts how AI-integrated code generation tools can be used to help with efficiency, but then again, not hamper the efficiency of the developer's code. By referring to the costs and changes of such tools, this paper shows the advantages of these tools, but also makes sure to give a thorough check to avoid code smells and copyright issues.

The study by Zhong et al. evaluates the prospect of Large Language Models (LLMs) to replace traditional programming resource websites like Stack Overflow [53]. The research is conducted thoroughly, emphasising the widespread misuse of API in the code generated by popular LLMs, even though the code is executable. Additionally, misusing the API can lead to critical problems. For example, resource leaks, software program crashes, etc. The authors propose ROBUSTAPI, a novel dataset which will evaluate the reliability and robustness of code generated by LLMs. It focuses on determining misuse of API, the ability to deal with unpredictable inputs and being resilient to failures. The study proposes strategies for reducing the misuse of API, which will improve the overall robustness of the LLMs' code generation. It also offers key insights to further improve and accelerate future research in LLM-based programming tools.

Burak et al. in their research paper compared GitHub Copilot, Chat-GPT developed by

OpenAI, and Amazon CodeWhisperer to assess code quality beyond standards like validity, correctness, maintainability and reliability. The study sheds light on the advanced nature of these code generation tools by focusing on the importance of strengthening the improvements in performance of newer tool versions [29]. Moreover, the study also highlighted the need to evaluate further standards, like readability and reusability and create areas for improvement, acknowledging the vast impact of these standards on core software development. Through experimentation, the paper also addresses limitations like dataset dependency and narrows the scope of the problems which could impact the versatility and adaptability of these code generation tools. Additionally, the research addresses various constraints and further provides detailed concepts of the practical significance and prospective challenges which is linked with the real-life use of these code generation tools in the software development context. In general, the study offers crucial insights into the abilities of tools and their significant contribution to software development, opening areas for further advancement in research and improvement. The overall breakdown and recognition of ways to improve these tools builds a significant foundation for further advancement in research and development of AI-assisted code generation tools.

Another study by Burak et al. discusses a comprehensive prospect of GitHub Copilot, an AI-powered code generator which ensures quality code generation regarding code correctness, validity of code and efficiency [15]. The authors introduced HumanEval, a robust dataset which consists of 164 coding problems, which, through experimentation, evaluated GitHub Copilot's potential to generate valid and correct code. Out of 164 problems, GitHub Copilot successfully achieved 91.5% in valid code generation, where 51.2% were partly correct code, 28.7% were correct code, and 20.1% were incorrect code that were generated by Copilot. The author also examined that the significance of docstrings and meaningful functions further improves the quality of code. The correctness and validity of the code decrease if the docstring is removed. The study concludes that GitHub Copilot is a promising tool for generating correct code, but it suggests that Copilot needs to be further improved for a more detailed comprehension of its capabilities.

Jesse et al discussed the Large Language Model taking over the tech world, software like Codex has become preferred for assistance in software development [19]. However, these generated codes often bring about concerns regarding the quality of the code, which mainly includes errors which are very simple to solve but extremely difficult to locate. Much research has indicated that these errors are standard and have a ratio of 1 in every 1600 lines of code. Research has also shown that these errors are often repeated since these codes are extracted from their training data. All these data are extracted from various GitHub repositories where previous codes, including all the code smells, are located. This has increased the concerns regarding the reliability of AI-assisted tools in software development. Furthermore, when Codex generates such codes, the debugging process takes much longer than that of human-generated code. The researchers in this article have created a dataset of bugs like these from different repositories and generated two types of code: one with the bugs and one without. Then, the AI models are asked to create either the correct fix for the code or find all the bugs present. After this experiment, Codex set the record for generating such errors at rates that were more or less double the actual code. To avoid these occurrences, the researchers have brought in a lot of approaches, including the inclusion of comments in the code, which would make it much easier to detect the bugs. Also, it has been seen that even incorrect comments give a better perception of the code smells than no comments at all. Overall, even though Codex generates tiny but inaccessible bugs in its code, these can be fixed with solutions like these. Understanding the influence of these bugs on the Large Language Models is very important to avoid them.

Sakib et al. (2024) prove how AI started as a tool for creating simple essays and emerged

as a much better tool for handling complex things like software development. It explores MLA models like ChatGPT and Natural Language Models, which unfortunately do not provide the most accurate codes. It is seen that ChatGPT has a success rate of 71.875%; however, it has trouble debugging incorrect codes even if feedback is provided [46]. The AI models are trained using vast datasets from Leetcode so that these models gain enough knowledge to learn about the coding issues. The dataset from Leetcode provided various scenarios divided into three difficulty stages. It was seen that ChatGPT performed well in the and""divide and Conquer"" algorithms and failed to solve most of the "reedy" and dynamic Programming"" algorithms. Despite the debugging issues, ChatGPT is said to be a developed model with very complex training methods and has been growing at a grand scale since the launch of GPT-1. The dataset was divided into easy, medium and hard-level questions, and from the failed outcomes, 63% of the codes were unable to debug despite receiving proper feedback from Leetcode. The success rate for easy, medium and complex questions was not above 90% and below 30%, respectively. The ones with a success rate of below 30% could not get fixed even after providing proper feedback. In conclusion, it has been seen that ChatGPT fails to give a correct answer around 30% of the time while developing software despite the help provided.

Sauvola et al. (2024) explain how Generative AI has helped advance Software development by bringing in better quality, reduced costs and increased productivity in the development phase. The authors proposed four scenarios that link human developers and generative AI. The first scenario shows how human developers have complete control over the software and require AI tools only to assist them with different tasks [47]. The second scenario depicts the role of AI in code generation and the project's testing phase. Here, humans act as the decision-makers for the project. In the third scenario, AI plays a more significant role in managing most software development tasks, such as designing, implementing, testing, etc. On the other hand, the complex functions are left for human developers to solve. In the fourth and final scenario, it is seen that generative AI does more or less all the tasks, and the work of the human developer is to ensure the quality of the software, make sure all the ethical considerations are met and lastly, problem-solve. The authors also focus on how GPT-4 have adopted a much more automated version and better efficiency. In contrast, generative AI does have a few challenges. These include the jobs of human developers getting replaced by AI models, failing to meet ethical concerns, and various issues while using restrictive frameworks. In conclusion, the authors asked people to come forward and do more research in this field to find solutions that would help to tackle these limitations. They also proposed conducting these scenarios practically to understand the issues better and find solutions.

Zilberg et al. examine the impacts and challenges of artificial intelligence (AI) advancements on software engineering, mainly through AI-enabled code generation tools. He uses a unique data-driven framework to analyze ChatGPT's coding potential across different programming languages, such as Python and JavaScript, to be the leading languages [17]. He performed data analysis, sorting relevant social media posts on platforms like Twitter and Reddit. Furthermore, the study revealed that ChatGPT is used widely in more than ten programming languages, specifically Python and JavaScript. These include diverse tasks such as debugging, interview preparation, and solving academic assignments. A theoretical analysis has concluded that AI-driven code generation has more significance compared to the emotions shared. As a matter of fact, people's feelings are justified as they are worried about how AI will affect the job fields in the future. Zilberg's framework also created a dataset of social media interactions, which will be widely known to encourage further research. The discussed datasets are used to check the quality of the code and make the AI tools efficient. The insights from this research show that AI software

engineering and programming languages will change fundamental strategies in the future.

Liu et al. explore the evaluation of ChatGPT's performance while attempting programming tasks. Emphasizes the necessity to give fair ratings, avoiding data leakage. The study also has a detailed explanation of factors affecting the performance, such as the size of code, wrong output efficiency issues and the alterations made between the languages. It is found that cchatpgt's capacity to fix the error is based on the statistical analysis that is being carried out, including the runtime faults [42]. This shows that AI models improve code by 20% by incrementing. In addition to this, the given data analyses the accessibility of public use. The main objective of any project is to improve the accuracy of the code generated in the real world. Moreover, Liu's analysis looks into the ability to fix bugs and code challenges following the strategies to fix them. He also highlights the importance of detailed feedback and resolutions for helping ChatGPT fix problems, emphasizing the fact that continuous feedback can improve code quality. The findings make it clear that AI technology has made coding easier; still, they need better ways to give feedback and learn new skills to handle complexity. In conclusion, more research is being done to improve AI-driven code generation tools for better performance and dependability.

Coello et al. mainly explain the performance of different AI models in the programming tasks [33]. He compares these with other languages such as Google's Bard and Anthropic's Claude. These highlight ChatGPT's capabilities for developers and researchers in coding fields. The result of this study shows that the strengths and weaknesses of models have a big impact on how the software is made. As a result, it concluded that GPT models are good at handling coding tasks by reducing human error. The performance of ChatGPT models varies, and this is also taken into consideration. Rigorous testing has been used to evaluate higher accuracy, where GPT-4 has excelled in adaptability. The study discusses the impact of AI's role in the future, keeping in consideration the fact that models are inconsistent and need to be supervised. Coello shows that ChatGPT is a much more effective tool for generating code. Considering the fact that ChatGPT cannot completely be replaced by humans, they work as volunteers who simplify coding, making it interactive.

This research by Turunen et al. focuses on the trust in AI–powered code generation tools, where it deeply involves developers and programmers sending their own responses regarding the way they evaluate and deal with the results and problem-solving capabilities of artificial intelligence. This document analyses responses, certain designs and other components to understand the reliability of AI.
This research involved two methods of study: Retrospective interviews and Design probe sessions. In the interviews, the researcher managed to receive responses from 17 AI developers, from which it was commonly observed that their trust in AI changed over time, and that the reliability and trust of the results have improved. This implies that he complexity and capacity of problem-solving of AI models have been improving over time.
For the design probe sessions, the researchers look into the design concepts with additional developers, in order to study the responses that AI models send, along with the procedures the AI models carry out, and user statistics. In the procedures, researchers mostly looked into the quality indicators and the control mechanisms, in order to know the result generation. This had the potential to have a better understanding of the reliability of these AI-generated results, and AI's ability to be trustworthy.
This research will help get a deeper understanding of the working and reliability of AI, along with looking to mitigate any shortcomings that may be visible based on the procedure and results of AI.

This research by Ruotong et al. delves deep into the aftermath of the utilization of AI tools

on software development. It looks into the challenges and the benefits that the developers, software companies and customers are going to come across, utilizing AI tools.
Multiple benefits of Artificial Intelligence tools for developing code efficiently. Furthermore, AI tools help to reduce errors to a great extent, which allows us to get better accuracy along with the reliability of the codebases. AI tools are also a great option for organizations, freeing up resources and reducing costs.
Integrating AI might bring a lot of challenges, where it might require more time to enhance codes and increase efficiency [51]. There are also limitations in the parameters of various LLMs, which may result in lower accuracy. Furthermore, it can be quite expensive to set up the environment, along with a complex setup as well. Therefore, even though these AI tools can be very helpful for developers, they might bring up a few negative aspects as well.

This research by Chaka et al. aims to make comparisons and delve into verifying the integrity of AI tools in distinguishing between AI-generated texts from human ones [32]. This is important to understand, as it allows us to understand the level of AI reliability and look to make upgrades and reduce flaws in the problem-solving procedures of the AI tools.
It was mainly found that the detection efficacy of the AI tools, when it came to analyzing the texts, was rather unreliable, due to inaccuracies in certain areas. When it comes to paraphrasing, the utilization of synonyms, and translations, they produced rather inefficient results. One of the main challenges of these AI tools is deciphering multilingual texts, as they are mainly built to understand and verify the English language.
Verification and deciphering language is one of the core areas where AI tools seem to bring value. Evaluating these scenarios will help us understand the AI's ability and quickness to adapt in unknown scenarios, where it learns other languages and deals with texts that are multilingual.

Suh et al., who solely present the exclusive integration of the recent methods for detecting AI-generated tools, including the code sources [50]. This rising concern is due to the widespread use of large language Models (LLMs) in the field of software engineering. While the LLLModels work efficiently, they simultaneously offer risks that possess security vulnerabilities, exclusive information violation, encryptions and also, accuracy of the codes, making it essential to develop human-authored codes from AI-generated codes.
Here, the authors experienced a significant gap in the detection tools as the AI-generated context (AIGC) detectors are, in most cases, inactive due to the fundamental differences between programming languages and human languages. Specifically, certain codes show restricted syntax, the structured workflows, and analytics, that is not evidently captured by the trained models. Through much research and experimentation, the findings have shown that the detectors have a precision level of around 60% in terms of codes, thereby resulting in the loss of generalization.
Moreover, the specialized tools, such as Stiffeners to detect Chatbot-generated code, have fallen apart when met with general programming languages like Python and C++. As a whole, the author demonstrates that their performance varies irregularly in a scale of below 24% F1-score for the datasets. This allegedly shows the weakness of the generation models compared to the different languages. In order to address these weaknesses, the paper goes through various detection strategies, one of which is the evaluation of three learning models: zero-shot learning and in-context learning. The following study shows that while zero-shot and in-context give more generalized and below-average results, refinetuning attains very high performance, exceeding 80% F1-score during evaluation and training. As a whole, finetuning captures the language patterns more accurately. Considering the improvements, thgeneralizationon is always a problem that needs to be seen and

handled.
Perhaps, the significant contribution lies in the adaptation of code embeddings derived from the Abstract Syntax Trees(ASTs). Unlike the simple code that exists, AST-based embeddings have accurate control flows and more logical dependencies inherited. As a result, they prove to have effective modelling code semantics, making the syntax indestructible, differentiating human and AI code. To better understand the importance of different code features, the authors presented a study showing modification of source codes and analysis of the impacts on detection accuracy.
Despite these advancements, the paper acknowledged that these approaches are not completely effective in the fields of programming languages,, whichindicateses a challenge to be seen in the software engineering field. Therefore, the author emphasizes the requirements to develop more general and robust language AI detection models which continue to evolve human code. In conclusion, various research and investigations resulted in limiting code model techniques. It demonstrates that while the structural codes have significance in the present world, having high performance gains, LLMs remain a major part of the challenge faced, suggesting scalable and reliable systems.

This research by Pradel et al. involves researching the techniques used to identify and gather information on vulnerabilities and weaknesses in software applications using ASTs and GANs (Generative Adversarial Networks) [5]. The model CodeGAN learns to break the code down and understand the depth of its structural components.

ASTs are extremely relevant when it comes to using AI in error detection [5]. Here, CodCodeGAN utilizes a generator and a Discriminator for fixing the code. The generator states the fixes and corrections (using sequence-to-sequence learning) for the areas where the code can be problematic or buggy, whereas the discriminator is responsible for checking whether or not the fixes and corrections from the generator work properly. ASTs can be very convenient because they allow for neutralising complexities in the code where complex bugs, for instance, race conditions and other examples, can be effectively dealt with. However, it can be considerably expensive to train GANs on larger AST models.
Due to most AST models supporting C/C++ languages, it is usually stated in three objective ways to mitigate the shortcomings and utilize programming languages well:

- **Language-Agnostic AST Parsers**: A parser generator is used that supports multiple coding languages for AST extraction, which can also be a slight hassle due to differences in each of the languages in terms of static and dynamic typing.
- **Cross-Language Transfer Learning**: A code can be pre-trainefinetuned-tuned in separate languages.
- **Universal AST Representations**: Using common structural representations across languages.

AST-based models are usually static, which means that it does not show runtime. Eventually, for the planned future upgrades, it is desirable to include fuzzing or concolic execution to extract runtimes from the models.

This segment examines Abstract Syntax Trees (ASTs) and their role in detecting bugs and weaknesses in code [9]. This method proves particularly valuable for maintaining cleanliness in static code analysis.

The primary benefits of using ASTs for error detection include:

- Preservation of overall code structures and components, even when altering certain elements for fixes
- Effective identification of patterns in complex structures and architectures
- Strong generalization capabilities in AST models

Schick et al. introduced ToolFormer, a model which addresses technical limitations and shortcomings of the utilisation of large LLMs. An introduction to ToolFormer, which aims to mitigate or produce a workaround for specific drawbacks in a scenario [27]. One of the main drawbacks is its failure to properly handle real-time data with proper accuracy and precision, which may give improper output. ToolFormer can be of assistance in cases like this, where, through simple API calls, it utilizes tools and reuses them. This leads to the models getting opportunities to self-learn how to use external tools and utilities to bring in more accurate data. Toolformer works on a three-step process: candidate augmentation, self-supervised loss filtering. For candidate augmentation, API calls are executed on sampled datasets, with the results being appended onto the texts. For the next step, which is self-supervised loss filtering, the text is then used by the model for loss computation with the help of API calls. It is stated that when the API is called, the future tokens being reduced on a significant level indicate that the sample is valid. When it comes to the filtering criterion, the model uses the perplexity values of the tokens it calculates, and ensures that those values help call APIs that are useful and assistive. After that, pretraining is used to improve the reliability of the overall workflow of the model. Toolformer has been reportedly proven to show more efficient results in comparison to the GPT-3 model when it comes to working with problems related to mathematics and factual research findings. Besides that, fin-tuning for preservation of the same level of effectiveness in results on standard language modelling. Due to the AI being efficient, it allows for smaller models to potentially perform better results in comparison to the larger ones. On the other hand, it also has specific limitations. It is only capable of calling one tool at a time for a specific problem. Along with that, the self-learning procedure is reliant on the base model's structure and efficiency in calling APIs, which means that a model with a weak base will end up bringing in more liabilities.

Li et al, in their research, combine static analysis with Large Language Models (LLMs) to find errors and bugs in complicated codebases. It mainly focuses on Use Before Initialisation (UBI) bugs in the Linux kernel. Static analysis is important for finding bugs in software before it is published, but it has limitations with big, complicated projects, as it has to balance accuracy and scalability [40]. Mostly, traditional static analysis tools give you a lot of inaccurate results that can fail because of a lack of resources. Whereas, LLift solves these problems by using the strengths of LLMs, which shows they can understand the codes in ways that traditional programs cannot analyse.
The innovative thing about LLift is Post-Constraint Guided Path Analysis, which guides the static analysis process by focusing on constraints that directly affect the conditions that cause bugs. This method cuts out the following execution paths, which makes the reports more precise. This helps minimize the cost of exploring the conditions. LLift also uses different methods and integrations to work with LLMs in a useful and effective way. It uses effective prompts, which help the model seek more relevant definitions instead of collecting all of the code at once. Moreover, Task decomposition breaks down difficult analysis into smaller and simpler steps, and Self-validation, on the other hand, encourages LLMs to check and fix their own answers to make them more accurate and consistent [40]. The evaluation of LLift shows that it is practically effective. The Linux community has confirmed ugs, which show a significant increase in both accuracy and memory capacity compared to the previously used static analysis methods. LLift's methods have promising

results on other large-scale projects, such as Nginx and EDK II, in addition to the Linux kernel. Different LLM tests show that the GPT-4 model works most effectively, but the GPT-3.5 and Claude 2 models are also suitable options.
On the contrary, LLift comes up with challenges.. For example, it depends on closed-source LLMs, token length limits, and sometimes errors happen when classifying bugs. Nonetheless, an immense step forward in combining traditional program analysis with AI makes it possible for large software systems to find bugs more accurately and at a faster pace. Adding LLMs can solve existing issues with static analysis.

Patel et al. discuss the differences and similarities between AI-generated code and human-written code, where the main focus is the correlation between software metrics and errors, using static analysis techniques on Java code samples derived from LeetCode issues [44]. The authors have gathered 90 pairs of code responses, each generated by humans and the other developed by Microsoft's Copilot using GPT-4 4 -4 which was then used by static code analyzers to investigate. The main objective was to find patterns that might show how these measures may estimate errors and demonstrate how the quality of code fluctuates between AI and human coding solutions. Hence, the results highlight two significant similarities between AI and human code. Firstly, metrics that measure the amount of executable and declarative lines of code (CountLineCodeExe and CountLineCodeDecl) are positively linked to the "Dead Local Store" fault, which means that increased lines of code are more likely to cause such errors. Secondly, the average cyclomatic complexity (AvgCyclomatic) has been associated with the bug "Avoid Literals In If Condition," which means that the most complicated and complex codes ar more likely to have complexity issues.
The research also discovered that AI-generated solutions often include greater numbers of lines of code compared to human solutions, indicating AI's inclination towards more complex implementations. The cyclomatic complexity, on the other hand, also indicated logical complexity, where AI-generated code has fewer bugs than code produced by humans, particularly for simpler questions. Therefore, it is concluded that AI-generated solutions are safer than code written by people.
Despite such insights, the research confirms that a particular programming language cannot actually justify how accurate and skilled humans are when it comes to developing code. Additionally, the study shows the differences between AI and human skills in programming, which directs subsequent efforts to improve AI-assisted software development. Significant evidence in concepts of software metric-bug relationships shows that AI code may be more expensive,, but it does not inherently lead to an increase in bug,s, which gives assurance for AI-driven coding assistance in software engineering.

Cotroneo et al. introduced CodeXGLUE, which is a large-scale benchmark designed to advance research in program understanding and generation with machine learning [54]. Similar to benchmarks like ImageNet in computer vision and GLUE in natural language processing have accelerated progress in their respective fields, CodeXGLUE is designed to play the same role for code intelligence. With the number of software developers growing worldwide, the demand for AI systems that can search, generate, translate, and repair code has become increasingly important.
CodeXGLUE, which combines fourteen datasets that span ten different tasks, captures a wide range of challenges faced by developers. These tasks include understanding and comparing code fragments, detecting defects, completing partially written programs, repairing buggy code, translating between programming languages, searching for code with natural language queries, generating code from natural language descriptions, summarising code into documentation, and even translating technical documentation across human languages. Many of these datasets are newly introduced, such as cloze tests across six

programming languages, line-level code completion benchmarks for Python and Java, and a dataset for code translation between Java and C#.
To make the benchmark usable and accessible, there are models that represent three influential approaches. One is CodeBERT, a BERT-style pretrained model for natural and programming language that supports understanding tasks such as clone and defect detection. Another is CodeGPT, a GPT-style model trained on source code that is better suited for generation and completion tasks. And lastly, an Encoder-Decoder framework, which combines CodeBERT as the encoder with a Transformer decoder that is applied to sequence-to-sequence problems such as code paraphrasing and translation. Experimental results show that these pretrained models consistently outperform traditional baselines and achieve strong results across all tasks. CodeBERT achieves state-of-the-art performance in clone detection, cloze tests, and summarisation, while CodeGPT achieves the best results in code completion and text-to-code generation. In code repair and code translation, models initialized with CodeBERT also prove most effective.
The results demonstrate he power of pretraining on large code corpora [11]. Furthermore, additional structural information, such as abstract syntax trees and data flow graphs, could further improve performance. Importantly, CodeXGLUE is not just a collection of datasets but also a platform for fair evaluation and comparison, which enables researchers to benchmark their methods consistently. The plan is to extend the suite to include more programming languages and tasks such as bug localization, program synthesis, and idiom mining.

Rozière et al., in their research paper "Open Foundation Models for Code," introduce Code Llama, which is a family of large language models (LLMs) derived from Llama 2 [26]. This specializes in programming-related tasks. Code Llama has strong code generation, infilling, long-context handling, and instruction following capabilities, which establish new, up-to-date results amongst open source code models. Meta AI released these models under a permissive license, which allows both research and commercial use.
Code Llama is available in three main variants: he first is a general foundation model for code called Code Llama, the second is Python-specialized models named Code Llama Python, and the third is instruction-tuned models called Code Llama Instruct[54]. Each of these comes in multiple sizes that range from 7B to 70B parameters. The models were trained on datasets of up to 1 trillion tokens, mainly code that is also supplemented with natural language to preserve general reasoning skills. There were specialized training pipelines that included steps for code infilling, long context finetuning, and instruction tuning, which let the models handle tasks like program synthesis, debugging, code completion, documentation generation, and natural language to code translation.
A key innovation is the ability to handle very long contexts that are up to 100,000 tokens,, which enablesrepository-levell reasoning rather than being limited to function orfile-levell tasks. Infilling training also allows the models to generate missing code segments within a broader context, and this improves real-world applications such as integrated development environment (IDE) completions. Instrufinetuning leverages both human feedback datasets and a self-instruct method, where the models generate coding problems, unit tests, and solutions, ensuring safer and more helpful outputs.
The models achieve state-of-the-art results on benchmarks, including HumanEval, MBPP, APPS, and MultiPL-E, that significantly outperform earlier open models such as StarCoder. It also matches or surpasses much larger proprietary systems in some cases. For example, Code Llama – Python 7B surpasses Llama 2 70B on HumanEval and MBPP, which shows the benefit of domain specialization. Scaling up to larger parameter sizes consistently improves performance, specifically for Python and multilingual code generation.

Beyond performance, Instruction-tuned versions show reduced toxicity, improved truthfulness, and mitigated biases compared to base models. Meta also conducted red teaming to assess dual-use risks, particularly around malicious code generation. It concluded that while LLMs can aid low-skilled adversaries, advanced malware development remains beyond current capabilities.
Overall, Code Llama advances open source code models by combining scale, specialization, and safety. It helps developers and researchers with high-performing, versatile, and accessible tools for coding tasks, which bridges the gap between proprietary systems and the open research community.

2.3 Summary of Key Findings

It has been seen that artificial intelligence tools have altered software development by reducing errors, making the development cycle more efficient, and improving performance at a great scale. However, there are a few disadvantages of these code generation tools, which may include incorrect codes, delayed run times, and ethical concerns. For this reason, strict human monitoring is required to ensure accuracy and precision. Studies have shown that AI tools like Copilot and Codex often improve efficiency with code generation but also increase security concerns. So, if the developers critically analyze these codes, it would be easier to avoid such challenges. It is also seen that such code generation tools can quickly take over platforms like Stack Overflow, but this increases the danger of API misuse. Furthermore, LLMs often require help from human developers to debug any sort of bug. Research has pointed out that AI-generated tools mostly use JavaScript and Python to debug errors, complete assignments, and prepare for interviews. Several feedback loops are required in order to help Artificial Intelligence handle new skills and languages. Even though AI tools are quite realistic, there are still a lot of trust issues. A study has shown that in a few cases, the AI-generated codes had many more bugs with slower runtime than those of a human developer. Although AI tools have proved to be great code generators, studies are being conducted to increase their reliability and security.

Chapter 3

Requirements, Impacts and Constraints

This section determines the requirements and standards for carrying out the pipeline, which detects errors in Python code and produces the corrected code accordingly. The methodology has been divided into three pipelines: Retrieval Augmented Generation, Fine-tuning with Code T5, and a prompt mechanism with the application of Pylint and AST. It's structured into segments of requirements, societal and environmental impacts, ethical considerations, standards, project management, risk management, and economic analysis [12].

3.1 Final Specifications and Requirements

Prompt Mechanism

At first, it uses AST to detect syntax errors within the code and then a tool named `pylint` is used to detect logical errors, followed by code smells, i.e. the underlying problems within the codebase. Then these issues are passed to OpenAI's LLM to generate the fix, along with the explanation, followed by automated unit tests generation [22] using the same LLM [24].

RAG (Retrieval-Augmented Generation) Pipeline

Provides explanations of buggy code through retrieving information from the Python exception library and fixes the code accordingly, followed by unit test generation [7].

Fine-tuning Pipeline

With the help of an external dataset based on code fixes, CodeT5 was used to fine-tune the model to extract the fix explanation, followed by code debugging and the generation of unit tests.

Functional Requirements

- Takes buggy codes as input.
- Detects syntax and logical errors.
- Natural language explanation for the errors for a user-friendly approach.
- Correct code and structured unit tests generation.

- Computation of test percentage.

Non-functional Requirements

- Ensuring high accuracy of fixed code generation.
- Minimize processing time for debugging cycles.
- Ensures security without having to go through the entire project. Works with only one codebase at a time to avoid breaches.

3.2 Societal Impact

- Supports beginner developers by providing detailed fix explanations therefore providing a good learning outcome [14].
- Enhances debugging processes for coders, which reduces computational costs and helps to focus more on innovation than debugging.
- Promotes more dependable software development techniques by incentivizing code quality enhancements in both the academic and open-source communities.

3.3 Environmental Impact

Positive impact: Reduces energy consumption as repetitive debugging tasks are scaled down to a great extent.
Negative impact: Uses cloud-based AI integration like OpenAI, which increases the computational costs of servers [6]. Also, LLMs may result in a carbon footprint depending on the model size or usage amount.
Mitigation:

- Select energy-efficient models to ensure lower energy consumption.
- Implement efficient batching processes.
- Cache common bugs with their fix explanations to save cloud server space.

3.4 Ethical Issues

- AI-generated codes may be biased toward the training data; human oversight is necessary for critical bug fixes.
- Generated codes may unintentionally produce copyrighted material [10].
- Strict sand boxing is necessary when executing flawed code to stop harmful inputs.
- Clear communication of system limitations should be provided to users to avoid over-reliance.

3.5 Project Management Plan

Requirement Analysis

- Define the scope along with requirement analysis.
- Identify required tools and models (AST, Pylint, CodeT5, etc.).
- Prepare the dataset for the fine-tuning pipeline [25].

Design Phase

- Design the flow of the RAG, fine-tuning, and prompt mechanism pipelines.
- Integrate OpenAI and Groq APIs.

Implementation Phase

Prompt Mechanism Pipeline:

- Implement syntax and logical error detection with AST and Pylint.
- Integrate OpenAI API for explanation and code fixes.
- Generate unit tests followed by execution.

Retrieval Augmented Generation:

- Implement error explanations using Python exception docs.
- Generate unit tests followed by execution.

Fine-Tuning Pipeline:

- Preprocess dataset (100k samples from TSSB-3M).
- Fine-tune the CodeT5 model to generate code fix explanations.
- Generate unit tests followed by execution.

Testing and Evaluation Phase

- Test each pipeline using the same buggy code simultaneously.
- Evaluate accuracy, reliability of fixes, and runtime.
- Run unit tests and evaluate pass rates.

3.6 Risk Management

Effective risk management is crucial to ensure the reliability, scalability, and accuracy of the system. This section identifies key risks that may affect the pipeline's performance and outlines strategies to mitigate their impact [3]. By proactively addressing these factors, we aim to maintain system robustness and minimize disruptions.

Risk	Description	Likelihood	Impact	Mitigation
Model Hallucination	LLMs may generate incorrect or illogical bug fixes, potentially introducing new issues.	Medium	High	Use multiple verification layers (AST + unit tests). Human review required.
Dataset Bias	Fine-tuning data may not cover all bug types, leading to skewed model performance.	Medium	Medium	Curate diverse datasets and continuously fine-tune with thorough preprocessing.
Cloud Service Downtime	The pipeline may break if external services such as the Groq API are unavailable.	Low	High	Implement caching and local fallback strategies to maintain functionality during outages.
Performance Bottlenecks	Processing time may increase significantly for large datasets, affecting usability.	Medium	Medium	Optimize model selection, batching, and parallel execution to improve efficiency.

3.7 Economic Analysis

The economic analysis surveys the financial sustainability for implementing the debugging pipeline, including cost comprehension, development operations, and sustained scalability [2].

3.8 Cost Components

Development Costs:

- **Personnel:** AI engineers for model fine-tuning, backend engineers for integration, QA specialists for pipeline verification.
- **Infrastructure:** GPU/CPU servers for model training, sandbox environments for early deployment.
- **Tools:** Open-source tools like AST and Pylint, plus paid APIs (OpenAI, Groq) for explanations, fixes, and automated tests.

Operational Costs:

- **API Usage:** Incremental expenses for each API request depending on model complexity.
- **Cloud Hosting:** Expenses for data storage and bandwidth utilization.

Chapter 4

Proposed Methodology

4.1 Design Process or Methodology Overview

This methodology section of the paper shows how a multi-model pipeline was created to address various code bug issues and build a foundation that improves code generation for Python-based codes. The design has three parts: Prompt Mechanism, Finetuning, and RAG Pipeline. Each of these has its own purpose and method for generating code fixes, producing the most optimal solution while also enhancing the scalability of artificial intelligence capabilities.

The first of these, the Prompt Mechanism pipeline, is designed to serve as a debugging assistant. With the help of tools like AST for syntax error detection and Pylint for error detection, it begins with conventional static code analysis. The Pipeline creates a two-step process for adjusting and correcting code by combining these standard tools with OpenAI's language models. The system first interprets detected errors into natural language descriptions, making them straightforward to both the model and the users. In the next step, OpenAI fixes the code based on the generated explanation. Through the automatic creation of unit tests, this cycle is augmented to check whether the corrections produce the expected results. Thus, the Prompt Mechanism pipeline combines static analysis, natural language translation, and test validation. However, the Pipeline strongly depends on OpenAI for explanations and fixes, which means that the accuracy of the Pipeline is limited by the reasoning of the quality model.

Now comes the second Pipeline, Retrieval-Augmented Generation (RAG), which takes a different approach by focusing its operation on runtime execution and documentation-bound responses. The pipeline executes the buggy code to generate exceptions from Python code errors instead of relying solely on static analysis. The error message and exceptions are compared in the Python Documentation of exceptions as a RAG pipeline method, which reduces the risk of misleading error explanations by providing context. Once the error is understood, OpenAI assists in producing a natural language explanation and proposing the correct code. Similar to the Prompt Mechanism, the RAG pipeline also relies on automatically created unit tests that run in real-time to verify the fixes. The combination of dynamic execution and integration with official documentation in the system allows for a transparent and effective bug-repair process. Dynamic execution is risky, especially in the presence of unsafe code, and the system may prove insufficient in handling complex logic errors that extend far beyond a single raised exception.

The third and last Pipeline, Finetuning, focuses on the process of training models to reason about buggy code more autonomously, rather than having the system focus on immediate corrections. This is a process function over a large dataset, a portion of TSSB-3M, which contains 100,000 samples of buggy code, patterns of bug types, and their respective edit scripts [62]. The processing involves changing each sample into an ordered instruction-

based schema to direct the AI. For example, a training exercise includes the instruction "Explain the bug in his code" and buggy code input, with the expected output being an explanation of the buggy code in natural language. The schema fits instruction-following models well, which is the case for CodeT5 finetuning in this dataset. The pipeline ensures that the model gradually internalizes patterns of bug identification and correction. The integration of Groq as a computational framework emphasizes the scalability of this approach, which allows the system to handle large amounts of data efficiently. The pipeline's strong dependency on dataset quality is understood, and it requires significant computational resources, making it more suitable for institutional or organizational-level applications rather than casual debugging.
The three pipelines collectively present a superimposed methodology that addresses both short-term and long-term needs in automated code correction. In-depth exploration of the Prompt Mechanism provides instant fixes through iterative correction. Though the RAG pipeline secures reliability through explanations in official documentation, the Finetuning pipeline ensures continuous model improvement through large-scale training. The strategy provides a balance between the immediate need for accurate debugging and the long-term ambition of creating a self-sufficient, instruction-following model. Through combining static analysis, dynamic execution, documentation retrieval, and large-scale finetuning, the methodology forms a cohesive framework that is both practical in the present and accurate in the modifications made in the section Preliminary Design or Design (Model) Specification

4.1.1 Prompt Mechanism Pipeline

The **Prompt Mechanism Pipeline** has a specific set of methodologies for automated code correction, which can be outlined in steps. First, perform a static analysis using Python's `Abstract Syntax Tree (AST)` module to check the code's syntactic correctness. Afterward, `pylint` identifies logical errors by detecting bugs and patterns that are not related to syntax. Simultaneously, the OpenAI GPT model compiles a report using multiple tools for an analysis. The pipeline guarantees that every user request is interpreted correctly and processed with respect to the rights assigned to the data, and with consideration of ethics and compliance. AI response optimization in steps, the system safely and securely offers relevant and accurate results. To validate the corrections, a **unit testing method** is used to confirm the accuracy of the modifications. OpenAI analyzes the test case similarity and checks that all relevant aspects of the functionality are sufficiently covered. Verify for the last time that the code fixes are correct both in syntax and in terms of operational effectiveness. The last step is running the modified code and checking whether the results meet expectations. Because of this, testing reduces the impact of errors that are often found in automated debugging systems.
Each test result is used to calculate the **pass rate**, which shows the degree of correlation between the corrections and the originally intended functions, thus indicating the effectiveness of the corrections made. A high pass rate signals correct implementation of the program, while a low pass rate shows more work is needed. When some unit tests fail, the error message is sent back to the earlier stages of the Pipeline, so that OpenAI can issue an improved natural-language explanation and other corrections. The feedback system continuously improves, optimizing reliability and performance. Consequently, AI-enabled computing resources will likely gain developer confidence due to transparency, accountability, and verifiable reproducibility.
The validation system built into the Prompt Mechanism Pipeline is part of the Inter-Automatic Code Repair. Testing is possible because it enhances reliability and trustworthiness in automatic code correction by ensuring correct operation, facilitating iterative reworking, and aligning with professional and accepted software development standards.

4.1.2 RAG Pipeline

The Retrieval-Augmented Generation (RAG) Pipeline aims to systematically tackle poorly constructed Python code through a codex-implemented and knowledge-augmented AI repair integration sequence. While you prefer conventional debugging practices, which rely almost entirely on static code analysis and thus miss a substantial portion of hidden runtime errors, the RAG Pipeline actively modifies and monitors the execution of the debugger path on dynamically behaved code. The fine-tune knowledge retrieval, leading to effective rationale and automated repair. In this way, the Pipeline not only repairs errors but also helps developers better understand the underlying causes of bugs, making the debugging process more transparent, educational, and efficient.

The RAG Pipeline is anchored in **execution of faulty Python code**, which is given as input at first. Afterward, the program runs the given code and monitors the system runtime exceptions associated with the case. This ensures that the system accurately records only cases where the code failed to run correctly. In these cases, the program executed to completion and provided a real error message and stack trace. These executions yield real error messages and real stack traces, which ultimately serve as the basis for error extraction and are then captured as structured diagnostic data for processing. or instance, exception type information, error messages like `PythonErrorExplainer` or unsupported operand types, **exception type and error message**, are preserved in their original `IndexError: list index out of range` detailed forms, `TypeError: unsupported operand types`, allowing the system to capture them precisely and completely in any given context.

After the structural and executional errors have been preserved in the **retrieval stage**, the RAG Pipeline then moves to the augmentation stage, which bases its knowledge primarily on the official Python documentation(https://docs.python.org/3/library/exceptions.html) By linking the type and message of the extracted exception to what is found in the Python documentation, the pipeline, then retrieves detailed and verified interpretations as the output of the data collection e forts. Recording the extracted type and message of the exception in the documentation ensures that the derived interpretations of errors are validated against the documentation rather than based on a random assumption. Therefore, this process supports the code structure, enhances its credibility and statutory precision, and generates context-steeped interpretations of structured knowledge for better reasoning and correction of the code.

In the next step, the Pipeline employs **OpenAI models** to fix the erroneous code, leveraging the explanations that were previously presented as contextual support. This procedure for fixing code dominated by AI operates in a way that assists specific repairs while respecting the original intent of the program and the standards of Python. One footing emphasizes the fixing process being iterative and sensitive to the context`TypeError`; it makes significant alterations as opposed to a straightforward exchange of text. For example, when asking (the corrected code) to function, the system may suggest casting an integer into a string in the code with the program's original logic.

The RAG Pipeline includes a **unit test generation mechanism** to validate that its code change is effective. In this stage, OpenAI comes up with a structured, organized set of test cases that represent both common and edge-case situations in which we might experience a bug. The focus here is to ensure that the code outputs the results expected by the corrected code across multiple contexts. The unit tests generated are specific to the code change and case, created on the fly rather than being generic or pre-written. The fixed code is then executed with generated unit tests, all dynamically using Python's `exec()` function. This design avoids file management, making it more adaptable for iterative debugging processes.

During the execution stage, we could easily compare all the unit test results and create a scoring format that aggregates the pass-fail results into a **pass percentage score**.

Furthermore, using a past failed scoring allows for the assessment of correction quality alongside a robust scoring method, which quantitatively demonstrates aggregated correction quality. The conclusion is that high pass rates indicate quality corrections (strong, accurate, and working), while low pass rates indicate the need for additional work. The scoring method provides both diagnostic output and a feedback signal, allowing for the restart of a retrieval-and-correction process to enable immediate improvement. For example, if some test cases fail, those error traces would be looped back as feedback into retrieval and correction to improve new explanations, and the new corrections would score better test results. The feedback loop is valuable to ensure the final output is accurate and further validated through iterative scoring of the repair process.
Finally, the RAG Pipeline produces a **comprehensive output package** that includes the original buggy code, the natural-language explanation, the fixed code, the dynamically generated unit tests, and the corresponding test results. This layered output allows developers to observe each stage of the debugging process clearly—seeing what went wrong, why it happened, how it was fixed, and whether the solution has been empirically validated. One of the most significant strengths of the RAG workflow is its transparency: every step of the process is interpretable, verifiable, and accessible to review. Automating both correction and verification while maintaining visibility at every stage, the RAG Pipeline accelerates the debugging process and builds lasting looped backend developers and AI-driven systems [57].

4.1.3 Finetuning Pipeline

The purpose of the Finetuning Pipeline is to serve as a vital component in enhancing automated code correction systems by teaching a model to comprehend, interpret, and correct mistakes in Python code. Whereas the **Prompt Mechanism** and the RAG Pipelines aim at detecting and repairing specific cases of errors, the Finetuning pipeline uses a large and heterogeneous amount of data to allow the model to generalise and provide solutions to various classes of programming problems T is is based on the idea of training an instruction-following model on a carefully curated set of thousands of examples, trained in such a way that it does not only make correct corrections on code errors, but also generates natural-language explanations of each correct on. Subsequently, the debugging is made transparent, interpretable, and more user-friendly for developers.
In the data prep processing stage which is one of the most crucial part of the methodology begins with selecting around 100,000 samples from the `TSSB-3M` dataset, in a provider file which has 3 million data set this data set itself as an extra range of buggy codes snippets along with the corresponding corrective actions we used 100,000 bugs from the data set since it was easily compatible. e sample of data points was chosen as it offers the best balance between computational feasibility and variety, ensuring the training data represents various types of programming bugs. In the dataset, every data point usually has three major components: The buggy code fragment (also known as the before state), (2) metadata about the nature or pattern of the defect, and (3) the fixed version of the code that illustrates the fix needed.
The edit scripts are scanned pre-processing to determine the new and old identifiers, and indicate where the buggy code has been modified or substituted. An example of this is when the name of a variable is misspelled or used incorrectly; the edit script directly associates that variable with its fixed version. Compiling such change scripts will ensure that the model can recompile valid code and provide useful context for generating plain-language explanations. By doing so, the dataset preparation process can explicitly describe both the issue and the fix, allowing the model to learn not only how to fix code but also how to explain why the corrections took place.
After cleaning and parsing the data, one needs to create so-called natural-language expla-

nations of the bugs that have been detected. Each buggy piece of code is accompanied by a textual explanation written in simple and clear English that explains the cause of the mistake and how to address it. An example, a typical error, like a type er or (type mismatch), is to mix an integer and a string without conversion. The clarification, in this situation, indicates that the mistake occurs because two incompatible data types are being combined, and the remedy uses the str() function to convert the integer into a string. Our function only combines the two data sets. Each examples are later organized in a format, and they can be instructed in an improved fine-tuning manner(usually in the form of a JSON object):

```
{"instruction": "Explain the bug in this code:",
 "input": "<buggy "od" >","
 "output": " <explanation>"}
```

Aligning with the large model's instruction-following behaviour, this model has helped normalize the Transformer's raw course structure into a more structured training for t. Each step usually provides a very clear direction, along with the buggy input code. It generates an explanation of the error fix as the output, which is then fed to the generative AI to fix the core according to the provided context or the explanation of the fix.
After the initial steps of pre-processing and formatting, the Pipeline enters the **model loading and training phase**. At this phase, the pipeline provides a `--checkpoint` argument as input to the system, indicating the specific version or state of the `CodeT5` model to be trained. The checkpointing system is useful for resuming training from saved states and supports experimentation and efficiency. This phase also allows the system to automatically detect the hardware (specifically the CPU or GPU), and the functions will configure to use those resources as needed. This flexibility enables the Pipeline to scale efficiently from personal workstations to high-performance computing environments.
After the model and tokenism are loaded, we enter the pipeline phase, where a **Grog client** will be incorporated for AI-driven code generation. Using the Grog client can help ensure that the model generates code that is syntactically and functionally accurate, with very low inference time. Several utility functions are written to ensure that the output from the pipeline is error-free. Specifically, a `clean_code()` function removes Markdown fences and redundant formatting within `ensure_valid_python_or_retry()` function.
Through this systematic and iterative Training design, the pipeline of finetuning transforms the model into a highly efficient, context-based working assistant. It only corrects complex and syntax errors with high accuracy, but I also explain what went wrong with the core and how we can fix it. This approach enhances user trust and understanding. Combining large-scale data-driven training, structured instruction formatting, and automated code validation positions the Finetuning pipeline as the most valid and scalable component of the correct complex training framework.

4.2 Data Collection

The quality of data for any machine learning or artificial intelligence system is crucial to its success. The methodology (Section 3) highlights the data collection, cleaning, transformation, integration, and reduction processes of three pipelines: the Prompt Mechanism, the Retrieval-Augmented Generation (RAG) pipeline, and the Finetuning pipeline. Each Pipeline approaches data differently, reflecting its distinct goals. Still, all are aligned with the central principle of producing structured, usable, and reliable input for debugging, code correction, and training purposes.
The Prompt Mechanism pipeline involves minimal explicit data collection. Here, the data itself is the input code, which is analyzed. The design philosophy of the Pipeline

is to serve as a real-time helper, relying not on pre-computed data sets but solely on the surrounding constraints. Regarding this aspect of cleaning, Pipeline demands that the input code is syntactically sound enough to be analyzed in Abstract Syntax Trees and is logically readable by Python. exception details are then converted into natural language explanations. The Pipeline, without using multiple datasets externally, allows for integrating these explanations with OpenAI outputs. The focus is narrowed down to the relevant exception and documentation to perform data reduction, avoiding unnecessary information. The processing is done to showcase how well and effectively the transformed code performs. This approach allows the RAG pipeline to work without having to deal with unnecessary data. The finetuning Pipeline is extremely data sensitive. The process requires the extraction of a large subset of 100,000 examples from the TSSB-3M dataset. Each example contains buggy code (before), bug type information (stub pattern), and an edit script detailing the fix [64]. Cleaning in the finetuning Pipeline occurs on a larger scale to maintain the accuracy of the bug-fix mapping, as the models need to be repaired. Along with that, it is essential to keep the transformation process, as each of those examples needs to be converted into natural language explanations. For instance, a typical record may consist of an instruction such as "Explain the bug in this code," the buggy code as input, and the explanation as out. Unlike the other pipelines, which do not explicitly require integration, the Finetuning pipeline integrates the original TSSB-3M dataset with custom formatting to produce a usable training set. Data reduction is achieved by filtering only the most relevant examples from the 100,000 samples and discarding nonessential details. I'm reprocessed by printing the number of valid training examples saved in the new JSONL file, which provides a measurable sense of dataset readiness.
Taken together, the three pipelines illustrate different philosophies of data handling. The Prompt Mechanism considers the input code as ephemeral data, which requires significant pre-processing and a massive transformation in real-time. The RAG pipeline, on the other hand, views exceptions as both mistakes and opportunities, gathering sufficient data from runtime behavior to provide a grounded explanation. The Finetuning pipeline, on the other hand, is well-suited for learning from large-scale datasets and is cleaning, transforming, and restructuring the data for long-term model enhancement. The information about these methods unveils a significant trade-off between instantaneity and scalability: the first two pipelines highlighted the light, on-the-fly data handling, while the Finetuning pipeline demonstrated how the pre-processing of large datasets can prepare models for the constant increase in generalization and accuracy. The project covers the entire range, ensuring both real-time debugging and long-term AI development, supported by reliable and well-processed data.

4.2.1 Data Collection

The data collection process is a complex and well-planned procedure tailored to suit each of the pipelines used in the framework. One of the distinguishing factors among the three pipelines is their shared shift toward curated, large-scale data handling. The first pipeline, known as the Prompt Mechanism, employs an implicit data collection strategy characterized by the interaction between user-supplied code inputs and outputs processed through `AST` and `PyLint`. This approach treats data not as static information to be stored but as an immediate issue to be analyzed and rectified in real time. As a result, the method remains highly flexible, requiring no external datasets, as any logical or syntactic issue encountered can be resolved using the reasoning capabilities of the OpenAI model. Its strength lies in its ability to interpret static analysis data effectively without depending on authoritative documentation, making the Prompt Mechanism a simple yet efficient debugging tool whose speed and performance are directly tied to the quality of the external API.

In contrast, the second pipeline, the Retrieval-Augmented Generation (RAG) system, introduces a deliberate and structured data collection process that expands the system's contextual awareness. It begins with the same initial data point. Still, the user's buggy code and methodology pivot decisively by dynamically generating a new, critical piece of information: the runtime exception before executing the code to produce an error, the Pipeline collects concrete, empirical evidence of the failed runtime data that includes the exception type and message initiates a precise retrieval process from a carefully selected, reliable knowledge source, which is the official Python exception documentation. This two-step gathering method transforms the Pipeline from a purely generative system into a knowledge-augmented system. The gathered data package, comprising faulty code, exception trace, and pertinent documentation, provides a solid factual base that supports the subsequent OpenAI claim. This approach improves the reliability and teaching value of the explanations, reduces the risk of model hallucination, and wins user trust. The disadvantage of this, however, is its scope, as it is exclusively activated by errors that manifest as uncaught exceptions during execution, overlooking logical flaws that do not always trigger an error.
The third and most ambitious Pipeline, the Finetuning system, embodies a paradigm shift from instance-level data collection to large-scale, batch-mode data curation for model specialisation. It does not aim to accumulate information to solve a particular problem. Instead, it intends to gather a vast collection of historical instances to recreate the intelligence of the underlying mode. The process of extracting 100k data points from TSSB-3M is a systematic investment in a base dataset. This data, with its structured variables of bug y code, bug patterns, and correction scripts, is the raw material of an elaborate data transformation procedure. Types are not merely samples collected by the Pipeline; they are processed into structured instructions following a format that builds a new, purpose-built dataset, which instructs a model on how to provide explanations and corrections to code. This strategy positions the Finetuning pipeline as a long-term investment to create a self-sufficient, specialised mode. Nevertheless, its usefulness is heavily dependent on the quality, breadth, and cleanliness of the original TSSB-3M dataset. It requires significant computational resources before preprocessing data and model training, and it is by far the most resource-intensive yet transformative of the three strategies.
Taken together, these three data collection methodologies form a cohesive and forward-looking framework. The prompt Mechanism handles immediate correction with agile, minimal data input. The RAG pipeline enhances answer quality and trustworthiness through dynamic, evidence-based data retrieval from authoritative sources. Finally, the Finetuning pipeline secures long-term capabilities by systematically internalising knowledge from a massive collection of real-world examples. This strategy ensures that the project focuses not only on solving individual coding problems but also on building a more intelligent and autonomous system for the future. It effectively balances tactical debugging assistance with the strategic goal of advancing automated programming intelligence.

4.2.2 Data Cleaning

The data cleaning phase is responsible for scanning and filtering raw information to set it up for effective processing . There are specific strategies that ensure data consumption and other requirements are fulfilled for each pipeline . The approaches taken by all three models, from validation to structural parsing, have all maintained the integrity and validity of the responses and data. The prompt mechanism pipeline aims to perform minimalistic and implicit data cleaning . This process, instead of transforming a large, noisy dataset, aims to precondition a single input for performing analysis with the help of this pipeline's tool . The Abstract Syntax Tree (AST) module is a strict syntax validator . If a code fails to parse, that failure becomes a data point where the specific syntax error is marked

in the correction. Besides, error analysis can be made syntactically valid by preparing it for PyLint. This approach underscores this Pipeline's design as a direct code correction service: its cleaning is performed on the fly with no persistent data store to cut.
In the Retrieval-Augmented Generation (RAG) pipeline the data cleaning process becomes more explicit and targeted mirroring its evidence based methodology . The cleaning is focused on the crucial task of extracting and refining exception types and messages from the executed buggy code." When code is executed dynamically the resulting exception traceback can be expansive and contain extra information such as file paths, line numbers in a specific environment and stack trace .PythonError-Explainer's task is to refine his raw, noisy output into its canonical components, which are the exception type and the accompanying message . This refinement is important for the subsequent retrieval step as it transforms a specific error report into a standardised key for querying the official Python documentation. An imprecise extraction could lead to retrieving irrelevant or incorrect documentation thereby compromising the entire explanation and correction process. Cleaning the RAG pipeline is a vital data normalisation step that ensures the fidelity of the link between runtime failure and the authoritative knowledge required to resolve it.
The Finetuning pipeline often allows for a very complex and robust data cleaning process, which is crucial for ensuring the quality of the training dataset.This process usually extracts simple errors from the provided dataset and then gives us the semantic structure of the changes in the code.However, cleaning involves parsing the edit_script to identify the old and new parameters in the changed code passing the buggy codes in our mo model, which sees a p understanding of the code structure and the application of efficient-tuning process . This has led to correct explanations, for but need more data to make the training more accurate. Therefore, we included 100,000 examples of buggy codes with their fix explanations in our data set . As a result, we achieved optimal accuracy when generating the fixed versions of our codes. [52].
In conclusion, the data cleaning methods in these three pipelines for my hierarchy, which are central to their inspirations, include a prompt mechanism that performs surface-level validation.This is crucial due to the support of analytic tools and generative AI, making the method fast. The RAG pipeline provides a crucial refinement step, purifying the time-series data so that knowledge is retrieved accurately . Finally, the find tuning pipeline helps engage deeply with the cleaning process, where code-T5 is used to train the Buggy Python codes along with their fixes, providing context on what is wrong with the code. Therefore we can fix it easily with this graduated approach to data refinement helps each pipeline to operate on a foundation of data suited to its purpose . Its balance is accuracy, and it helps to correct the codes immediately with further unit tests.

4.2.3 Data Transformation

The data transformation phase represents the crucible where raw, often technical data is combined into forms suitable for intelligent processing and human comprehension . Across the three pipelines of this project, transformation is not just a procedural step but the core intellectual operation that bridges the gap between machine-readable code and the semantic understanding required for correction and explanation ideology was used by the prompt mechanism R , and find tuning pipelines shows that knowledge augmented synthesis helped a lot in creating the fixed correction of the code from the error that we gave and it also showed how different methods of Artificial Intelligence can be applied to fix Buggy code
The prompt mechanism is a powerful, general-purpose tool responsible for two primary purposes: translating errors and fixing code. Initially, the model processes have errors . These errors are sampled from the noisy code and converted into natural language

descriptions and code fixes using OpenAI. Transformation of the code involves two steps. Finally, the confusing and complicated output from AST and pylint is converted into natural language descriptions,enabling the user to understand correct errors, and allowing the OpenAI model to make sense of them. The model can be utilised to form a revised segment that would parse around the fixed code and help the model to understand the intent of the code through the Syntax Tree created, and turn the code into something that is logically correct. This pipeline helps to create something which is executable and accurate at the same time, using the OpenAI model as the generative AI, which changes the natural language explanation to the fixed code [30].
The Retrieval-Augmented Generation (RAG) pipeline tends to use a different step of transformation. It takes the help of Python's exception document and retrieves the data from it according to the buggy code provided. This system combines the raw retrieved data with the error explanation and passes it as a natural language input to the generative AI model that is being used, and therefore provides context to the model to generate the fixed code.
On the other hand, the finetuning pipeline works in a different manner; it operates on over 100,000 examples of buggy code and its explanations and trains the Code-T5 model to produce the correct explanation of the error according to the buggy code. Then the correct explanation is passed to the generative AI, and then it fixes the code, and after that, like all the rest of the pipelines, it is passed to an automated unittest, which checks for further syntax.

4.2.4 Data Integration

The outlined data integration phase in the project methodology serves as the analytical process by combining different data sources to form a unified whole for effective analysis. This cycle is a monumental procedure involving a set of adjusted strategies that reflect the unique data flows and requirements of each pipeline. The three pipelines have a wide range of effects in terms of integration, from implicit merging to comprehensive dataset synthesis. Each of these pipelines utilizes its own features and components to bring effect in its own way. Data integration in the prompt mechanism pipeline is very easy to integrate. The transformation process from this specific pipeline provides context to the OpenAI model by combining the noisy code with the output responses from AST and pylint; this process does not require any external datasets to fulfill the transformation. Data integration in the Retrieval-Augmented Generation (RAG) pipeline is more knowledge-driven. The documentation of the Python exceptions was used in order to create error fix explanations and then they are combined with the buggy codes in order to provide the contextual explanation which would help the pipeline to provide more in depth knowledge to the generative AI. This fusion of the runtime data and the collected resources increase the accuracy within the fixed code. The fine-tuning pipeline helps to undertake the comple data integration process. It also ensures that the datatset os robust and accurate. The raw data which is found in the TSSB-3M dataset— the fix explanation, buggy codes, and the bug types are parsed in order to combine the natural language explanations to form JSONL examples in a structured way. The integration transforms disparate data elements into a more digestible and comprehensible format that the model can use to train itself. In conclusion, the data integration methodologies across the three pipelines offer workarounds for code enhancements in their own unique way; the prompt mechanism performs simple context merging for immediate use. The RAG pipeline integrates empirical data with empirical knowledge sources for grounded reasoning. Finally, the fine-tuning pipeline engages in data synthesis, which helps make further improvements in terms of precision. It also increases the efficiency of the codes that it produces. This approach ensures that each pipeline works with the integrated data that is suited for the purpose, and it also

brings in simplicity in real-life operation with foreigners while we are working with this model in the long run.

4.2.5 Data Reduction

The theory of data reduction which controls the planned refining and focused information emerges as an important operation inside the proposed methodology of those three pipelines. The outlined methodology from the Prompt Mechanism, RAG, and Finetuning previews an experienced effort that helps to reduce incorrect fixes and ensure that there is no hallucination while generating the fixed code. This process of refinement is changeable. It is an indirect process which focuses on communication of the knowledge base that is rooted in convention and peaks in a large-scale organization that shapes the intelligence of the system.

The Prompt Mechanism pipeline plays a crucial role in a form of reduction that is indirect and built-in to its sequential process. It indirectly reduces error by focusing only on relevant error outputs and code segments for correction and testing. Through the stages of AST and pylint analysis, the code passes and a vast amount of potentially unnecessary information like correct lines of code, rhetorical suggestions considered far less important and also provided warning information is typically filtered out by the pipeline's own deferential filter focused around the code, a syntax error preventing parsing process or a logical error identified by the pylint library. This reduction is the pipeline's holistic design. It makes sure that the computational and financial cost of promoting the OpenAI model is reserved just for real problems by preventing the system from being deluged by the entire code and instead focusing on its generative capabilities on a specific set of errors. This indirect refinement is the reason for its efficiency because it acts as a system that passes only the crucial relevant data to the expert model.

The Retrieval-Augmented Generation (RAG) pipeline demonstrates a more targeted and focused data reduction centering on its evidence-based methodology. The process involves reductions on the specific exception and related documentation suggestions relevant to the buggy code. This is a two-stage refinement process. Firstly, the possibly extended, complex runtime output including full stack traces and environmental details is reduced to its orthodox parts: the exception type and message. This abstraction is critical because it drops the circumstance where the error occurs to focus on the essential syntax. This reduced error signature is used to query the official Python documentation, a vast knowledge base. The retrieval mechanism itself is an act of censor that fetches only the documentation subsection relevant to the elevated anomaly rather than the entire collection which ensures that the generative model operates within a tightly constrained and highly relevant context.This drastically increases the possibility of an accurate explanation and proper correction.

The Finetuning pipeline handles data reduction as calculated and fundamental to its goal of model specialization. The process reduces the dataset which chooses applicable instances from the 100k datapoints and saves only the formatted examples to a new JSONL file. This is a calculated reduction at the population level. From the massive TSSB-3M dataset, a curated subset of 100,000 examples is selected, implying an anecdotal perception on which data points are most suitable for training. Furthermore, the conversion process itself is a form of reduction, dropping raw, unstructured sections like the original in favor of a refined instructional format containing only instructional input and output which removes the procedural part of code, modifying the differential mechanics while preserving and featuring the terminology. This represents the bug with its explanation. This curated, reduced dataset serves as the predefined fuel for the Finetuning Process which is designed to effectively instill the desired behavioral patterns into the CodeT5 model without the noise and inefficiency of raw, unprocessed data.

In summary, the data reduction strategies among these three pipelines usually contribute significantly to native effectiveness. The prompt mechanism depends on processes that filter out main errors to provide context to the generative AI. The RAG pipeline helps to identify and reduce hallucinations (unnecessary incorrect fixes) which offers a framework and helping genetic improvement towards better video hallucination. Lastly, the finetuning pipeline performs pipeline-scale training which usually offers a high-quality, detailed explanation of the data set. Combining these approaches ensure that each system from the reactive debugger to the pedagogical explainer is not overfitting by information overload but is authorized by a concentrated stream of the most relevant and actionable data, enabling clarity, accuracy and effective learning.

4.2.6 Summary of Preprocessed Data

The summary of the preprocessed data serves as the final act of validation, reporting on the three pipelines that transform complex addresses found in codes into simple forms of fixed code. The methodologies employed by the Prompt Mechanism, RAG, and Finetuning pipelines reveal a clear distinction in their definitions of success, ranging from functional validation and empirical scoring to quantitative curation.
The conclusion of the suggested data is usually searched as the last check-up part, and the summaries of the three pipelines that we created will be reported, capable of changing the complex information into the simpler formats and fix its mistakes. The way the prompt mechanism works by applying methods including Rag and finetuning pipeline make the methodological differentiation clear, as far as validation functions and accuracy scores to quantitative curation.
At the end of the Prompt mechanism pipeline, the practice in the form of pass or fail is presented as the final step of the working state which corresponds to the presented pipelines as the code correction mechanism. The RAG pipeline follows a More Explicit and highly quantified line of summarisation, providing a summary of the code fixes and makes them understandable to the user of generative fixed code more so where in case of multiple unit tests fixed.
The operating pipeline is in a meta-appropriational summary which concentrates on integrating into itself its own preparatory action instead of repairing the actual single nipple. His summary was also a necessary sanity check to make sure that all of the conference processing has been done appropriately as well as the amount of the training data is appropriate to do some fine-tuning.
The data collection methodology suggested in this paper shows a pragmatic approach to the processing of the information, which has various layers and provides high levels of parity between the three pipelines. Conclusively, it integrates all the three due to their common goal, which is to make dependable and precise corrections as code that can help in development. The agility, depth, and longevity are captured through the evolution stages of minimalistic easel data collection to high structure data collection.
Replacing 100,000 samples of the TSSB-3M dataset, consisting of buggy code, the bake time, and the edit script, it is possible to fuse them into an input to the bucket. We can also combine the back pipe with the explanation to create a fixed explanation or code. And like that, whether it's the context of the prompt mechanism or the runtime issues of RAG, this pipeline collection is permanent and designed to produce the raw material for explaining code errors. Therefore, in the long term, it can significantly aid in generating explanations. He fixes buggy Python code. He is heavy in the structure. Additionally, the cleanliness makes this pipe highly demanding and competitive. It is the most transformative, as it typically enhances the model based on the training arguments and the dataset provided.
While working together, these three pipelines assist in data collection to showcase the scope

of strategies mapped onto the use cases. The prompt mechanism works immediately as it requires a single user input to function efficiently. In the RAG pipeline, elements incorporate data and trustworthiness, along with a one-time evidence and authority reference. The process of finetuning adopts the architecture and structure of the model towards the construction of the platform on which its specializations can be based, and thus having a long-term success. The approaches are personalized according to the separate purpose, as well as to the philosophy of incorporating the most important statistics on trucks, including both speed and depth of the code, since the pipeline is a whole that focuses on these points.

The developed pipeline is rather important as it indicates the need of diversity in data processing. Any one ode or collection is not enough and needs agility to perform real-time debugging. In the absence of a structured retrieval method, the unreliability of the code fixes is comparatively elevated since they are in the form of goals that are not contextual, thus can cause goal hallucinations and get back to buggy code. With such methods, this project will be robust both in the short and long term code. The system, and not only the tool, is placed query by this layer strategy to facilitate immediate debugging and give a systematic and automated unit test to assess whether the corrected code is produced as anticipated and within minimal time. It is not dependent on human intervention which points to the mental reality of machine learning and AI: the quality, quantity, and plan are not culpable, and also points to the intelligence.

4.3 Implementation of Selected Design

4.3.1 Prompt Mechanism Pipeline

The Prompt Mechanism Pipeline is made up of a tightly ordered loop that includes static analysis, reasoning with Large Language Models (LLMs), and test-driven validation [20]. Implementation starts with structural parsing of Python's module to find any grammar errors right away; code that does not parse is skipped over so that it can be fixed immediately instead of proceeding to deeper checks. When syntactically correct inputs are used, the processes involved perform a schematic and logical search, identifying all rule violations, runtime bugs, and logical issues such as errors in code formulas or instances of adding instead of subtracting. These two steps work together: a static gate checks AST for syntax correctness, while Pylint addresses problems at the logical level before detecting any runtime failures in the generative AI. The results from both analyses are consolidated and normalised into an aggregated list of detected issues for subsequent steps.
Once the error bundle of the code has been passed it is quickly transform into the natural language where a generative AI called **OpenAI** is from that to generate the concise formation of the fixed code in a developer friendly manner this involves converting the brief messages or the explanations which is the sent to the generative AI and the explanations include on how to fix the codes and also consists of the errors' syntax and logical in the second alarm past the same contact the original code is usually combined with explanation and then it is used to propose a specific code modification and after that this two past approach (explain $\rightarrow$ fix) minimize patching and helps to tighten the alignment between the diagnosis and the correction.
Using **OpenAI**, this process helps create unit tests specifically for the code environment, including its functions, intended behaviours, and all necessary documentation to verify the correctness of the fixed code. The goal is to ensure that the design is precise and efficient, covering both evidence implementation and all the common errors suggested by the pilot results, and all the fix explanations that were proposed once the unit test is generated or the system executive code is executed against the production suite and records all the

results through the generated test. Successful test executions, indicated by the high pass rate of multiple tests combined, signify a successful iteration. y fail cases are looped back into the explanation, the code is fixed again, and then passed to the unit test, forming a **iterative improvement cycle** and then the final output for each run includes the Buggy code, the description in natural language, and also the fixed code. The generated unit test includes the corresponding results; this comprehensive output enables full debugging ability, helping the developer learn as they fix the errors.

4.3.2 RAG Pipeline

The RAG Process is set up as a loop that operates throughout the document in a runtime-driven manner, providing fixes and explanations. Errors are usually based on the provided Python documentation and real-life execution feedback. It begins by executing the provided buggy code So that the exception is produced during the runtime rather than relying on the static conditions like Syntax errors the system then calls the Python error explanation component to extract all the various error types and the corresponding explanation of the fix storing them in an organized way so that it can help that developer to see what error and how they will be fixed.
Once the error has been clearly defined, the procedure is to process the **Built-in Exceptions** section of the Python Documentation of exceptions. It specifies the type of error, such as an index or type error, that the system retrieves, and generates a clear, human-readable description linking the mistake with its typical resolution methods [7]. This ensures that the explanations are not merely probabilistic or speculative, but instead grounded in the formal definitions provided by the language itself.
Next, the process repairs the code using **OpenAI**, making changes based on two key inputs: (a) the original buggy code sample, and (b) the documentation-based explanation. This pipeline reduces the number of errors and also helps in targeted error detection without having to hallucinate. The text exec () function is used to run both the corrected code and the error code to check what the initial pass rate was and then what the final pass rate was. In the past, this led to the vascular test. The accuracy is calculated with the overall **pass-percentage score**. This score provides a simple indicator of the repair. If the value falls below the required threshold, the code is passed back to the Pipeline for further fixes.
Finally, Fine-tuning Pipeline provides a comprehensive set of variables, including the buggy code, an explanation of the error, an explanation from the Python documentation, the unit tests, the fixed code, and the test results. Including the overall pass rate. This structured output ensures that every stage of the debugging process is transparent, traceable, and reproducible. As a result, the RAG pipeline functions as a documentation-aware, test-driven loop that combines empirical code execution with authoritative references for automated and reliable debugging.

4.3.3 Finetuning Pipeline

The Finetuning pipeline is made up of a generation-and-validation loop that can be used repeatedly. It has a fast LLM runtime, an instruction-tuned encoder–decoder (`CodeT5`), and a simple execution harness for instant feedback. The execution run starts by setting up the runtime environment for AI code generation, known as the `Groq`, so that the model can generate the fixes in low latency, which would help in further testing. Furthermore, two useful parameters are set, which are `clean_code` that removes all the stray formatting that often results from the model outputs and therefore creates confusion, and `ensure_valid_python_or_retry` that continuously passes the code to the pipeline until

the produced code has 100% accuracy in the unit tests provided and produces the correct code.
After the runtime is configured, the process takes a portion of faulty code and asks `CodeT5` to provide a clear and human-understandable explanation of why the code has errors. We leverage `Code's` instructions-following capability here because it was fine-tuning and preparing the dataset in advance. to transform broken code into meaningful, interpretable explanations. This explanation forms the foundation for subsequent correction steps. It is provided to `Groq Chat` with the original code so that `Groq Chat` can generate a modified version of Python code that addresses the specific failure case directly, rather than applying surface-level changes. It is more accurate and efficient to separate the responsibilities of analysis and repair (`CodeT5` $\rightarrow$ explain, `Groq` $\rightarrow$ fix).
The purpose of this method is to test the core functionality described in the fix explanation, along with the Limited set of cases that typically lead Jan to the identified bug type. e t, t e exact function runs both the correct code and the generated test so that we can see the results of the fixed code. It collects the `stdout/stderr`, errors, and assertion results in a Python-based environment or whichever environment is good for the model. This reduces I/O and sandboxing overhead, supports rapid iteration, and simplifies separation between execution runs.
During execution, the system continuously records the results of each test and calculates the overall pass rate. This system helps us gatekeep only the code accepted by the unit tests and alerts us if there is an issue with the fixed code generated by the model. If the fixed error accounts for a percentage of the field tests exceeding the given threshold, the code is passed through the pipeline again. Accordingly, fixed code explanations are provided, and the errors are corrected. This indicates that the problem has been resolved successfully, and the iteration concludes. The system then outputs a complete set of artifacts, including the original buggy code, the explanation of the problem, the corrected code, the generated unit tests, and the corresponding test results (with the pass rate). This structured output can also be integrated into Continuous Integration (CI) systems, allowing the pass rate to be logged and tracked over time for continuous quality monitoring.
Therefore, this model provides rapid code correction and validates the corrected codes through automated unit tests, ensuring that every correction is both logically sound and empirically verified before acceptance.

Chapter 5

Result Analysis

5.1 Performance Evaluation

The developed code fixer was evaluated across three core approaches: prompt engineering, fine-tuning, and a Retrieval Augmented Generation (RAG) pipeline. For all three, the workflow involves taking buggy Python code as input and returning a fixed version, detailed error explanations, auto generated unit tests, and runtime results for both original and corrected code.

5.1.1 Setup

Performance was measured on the accuracy of bug identification, the quality and correctness of code fixes, the thoroughness of error explanation, and the reliability of the unit test generation. The fine-tuned model, using CodeT5 Salesforce, demonstrated higher consistency in producing robust error explanations and high-quality code fixes, even when the external API (Grok) was unavailable, providing informative fallback fixes and tests [8]. Automatic handling of corner cases, such as division by zero, was reliably achieved, with corresponding exceptions and user-oriented messages.
Prompt engineering showed proficiency in diagnosing errors, especially for syntax and basic runtime issues, and supplied practical code comments and step-by-step correction tips. However, its responses were somewhat template-based, and occasional ambiguity was observed in nuanced explanations.
The RAG approach was proficient in adaptability and leveraging external knowledge of the buggy code, resulting in contextual explanations of errors (e.g., mathematical justifications, syntax errors) and diverse test cases. The unit tests checked for edge cases and demonstrated strong coverage.

5.1.2 Sample Code Outputs

Buggy Code Example:

```
def divide(a, b):
    return a / b
result = divide(10, 0)
print(result)
```

Prompt Engineering Output

```
Syntax error:
Syntax error detected: expected an indented block after 'if' statement on line 1 (<unknown>, line 2)
Attempting to fix syntax errors with Groq API...
Syntax Fix Explanation:
The error message indicates a problem with indentation in your Python code, specifically with the 'if' statement.

The issue is that the line `def divide(a, b):` is not indented under the `if __name__ == '__main__':` line. In Python, indentation denotes a block of code within a control structure, function, or class.

When Python sees the `if __name__ == '__main__':` line, it expects the next line to be indented to show it is part of the 'if' block. However, your code has the function definition at the same level of indentation as the 'if' statement, which Python does not understand.

To fix this error, indent the function definition and all lines that follow it that are meant to be part of the 'if' block.

The function definition should be at the top level of your script, outside of any control structures. Then, call the function and print the result inside the 'if' block, which will be indented.

Additionally, dividing by zero will cause a runtime error. Add error-checking code to handle this case.

Ensure proper indentation for all lines that are part of a block, like 'if' or 'for' statements, or function and class definitions. Python relies heavily on indentation to understand code structure.

It is good practice to ensure your code handles potential errors during execution.

The correct structure is to define the function at the top, then have the 'if' block with the function call and print statement indented under it.

With these changes, your script should run without syntax errors.
Syntactically corrected code saved to: temp_calculator.py
Pylint completed with issues (exit code: 16)
Pylint analysis results:
************* Module temp_calculator
temp_calculator.py:8:0: C0304: Final newline missing (missing-final-newline)
temp_calculator.py:1:0: C0114: Missing module docstring (missing-module-docstring)
temp_calculator.py:1:0: C0116: Missing function or method docstring (missing-function-docstring)
Unit test:
Your code has been rated at 5.71/10 (previous run: 2.00/10, +3.71)
```

–

Error Explanation:

Fix Explanation in Natural Language:

Let's go through the issues with your Python code one by one, and I'll guide you on how to fix them.

Syntax Error: Expected an indented block after 'if' statement
The issue is that the line 'if b == 0:' is missing indentation. In Python, when you start a block of code (like an if statement, for loop, etc.), the lines that are part of that block need to be indented. Fix this by adding four spaces before the 'if' statement to indicate it is part of the function.
Pylint Issues
Now, let's tackle the issues reported by Pylint.
Issue 1: Missing module docstring
A module docstring is a string at the top of a Python file that describes what the module does. Add a docstring at the top of your file explaining the module's purpose to satisfy Pylint.
Issue 2: Missing function or method docstring
A function docstring inside the 'divide' function should describe what it does, its inputs, and outputs. Add this to resolve the issue.
Issue 3: Final newline missing
Pylint complains about a missing blank line at the end of the file. Add a newline to fix this minor issue.
Additional Issue: Code Execution
The code executes a division by zero, causing a runtime error. Consider raising an exception or adding error-handling code.
To summarize, you need to:

1. Indent the line after the 'if' statement.
2. Add a module docstring at the top of your file.
3. Add a function docstring inside your 'divide' function.
4. Add a blank line at the end of your file.

By following these steps, you should resolve all issues.
Fixed Code:

```
"""
This module provides a simple function for dividing two numbers.
"""
def divide(a, b):
    """
    Divide two numbers.
    Args:
        a (float): The dividend.
        b (float): The divisor.
    Returns:
        float: The result of the division.
    Raises:
        ZeroDivisionError: If the divisor is zero.
    """
    if b == 0:
        raise ZeroDivisionError("Error: Division by zero")
    return a / b

if __name__ == '__main__':
    try:
        result = divide(10, 0)
        print(result)
    except ZeroDivisionError as e:
        print(e)
```

Fixed code saved to: `fixed_calculator.py`
Running unit tests on fixed code...

```
test_divide_division_by_zero
(test_calculator.TestCalculator.test_divide_division_by_zero)
Test division by zero. ... ok
test_divide_edge_cases
   (test_calculator.TestCalculator.test_divide_edge_cases)
Test division with edge cases. ... ok
test_divide_invalid_input
   (test_calculator.TestCalculator.test_divide_invalid_input)
Test division with invalid input. ... ok
test_divide_normal_cases
   (test_calculator.TestCalculator.test_divide_normal_cases)
Test division with typical inputs. ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.001s

OK
Fixed Code Unit Test Summary: <unittest.runner.TextTestResult
   run=4 errors=0 failures=0>
```

RAG Pipeline Output

Testing model: `openai/gpt-oss-120b`

- Model `openai/gpt-oss-120b` is available and working
- API key is valid with model: `openai/gpt-oss-120b`

Generated Error Explanation:
What the error means
`ZeroDivisionError: division by zero` is Python's way of indicating that you tried to divide (or take the remainder of) a number by 0. In mathematics, division by zero is undefined, so Python stops execution and raises this exception.
Why it happened
The error is raised whenever the right-hand side (the divisor) of the `/`, `//`, or `%` operator evaluates to zero. For example:

```
a = 10
b = 0
c = a / b  # <-- raises ZeroDivisionError
```

Even if the divisor is a complex expression, Python checks the final value:

```
c = a / (len(my_list) - len(my_list))  # len(my_list) -
   len(my_list) is 0
```

How to fix it

1. **Ensure the divisor can never be zero**
   - Initialize variables with non-zero values.
   - Validate input data (e.g., user-provided numbers) before use.

Generating Fixed Code:
Calling Groq API with model: `openai/gpt-oss-120b`

- API Response received successfully from `openai/gpt-oss-120b`

Generated Fixed Code:

```
def divide(a, b):
    """
    Divide two numbers.
    Parameters:
        a (float or int): Numerator.
        b (float or int): Denominator.
    Returns:
        float: Result of a / b.
    Raises:
        ValueError: If b is zero.
    """
    if b == 0:
        raise ValueError("Divisor 'b' cannot be zero.")
    return a / b

def main():
    """Execute a sample division with error handling."""
    try:
        result = divide(10, 0)
        print(result)
    except ValueError as err:
        print(f"Error: {err}")

if __name__ == "__main__":
    main()
```

Generating Unit Tests:
Calling Groq API with model: `openai/gpt-oss-120b`

- API Response received successfully from `openai/gpt-oss-120b`

Generated Unit Tests:

```
import unittest
import io
from unittest.mock import patch

# Assuming the functions `divide` and `main` are defined in the
    global namespace
# as provided in the module `divide`.

class TestDivide(unittest.TestCase):
    """Unit tests for the divide function and main execution."""
    def setUp(self):
        """Set up any common test data."""
        self.sample_cases = [
            (10, 2, 5.0),
            (-9, 3, -3.0),
            (5.5, 2, 2.75),
            (0, 5, 0.0),
            (10, -2, -5.0),
            (1e12, 1e6, 1e6),
        ]

    def test_divide_normal_cases(self):
```

```
        """Test divide with typical positive, negative and float
           inputs."""
        for a, b, expected in self.sample_cases:
            with self.subTest(a=a, b=b):
                result = divide(a, b)
                self.assertAlmostEqual(result, expected, places=7)

    def test_divide_zero_denominator_raises(self):
        """Test that dividing by zero raises a ValueError."""
        with self.assertRaises(ValueError) as cm:
            divide(10, 0)
        self.assertEqual(str(cm.exception), "Divisor 'b' cannot be
           zero.")

    def test_divide_edge_cases(self):
        """Test edge cases such as zero numerator and large
           numbers."""
        # Zero numerator
        self.assertEqual(divide(0, 7), 0.0)
        # Very small denominator
        result = divide(1, 1e-9)
        self.assertAlmostEqual(result, 1e9, places=2)
        # Negative denominator
        self.assertEqual(divide(8, -4), -2.0)

    def test_main_handles_error_and_outputs_message(self):
        """Test that main prints the correct error message when
           division fails."""
        with patch('sys.stdout', new_callable=io.StringIO) as
           mock_stdout:
            main()
            output = mock_stdout.getvalue().strip()
        self.assertIn("Error: Divisor 'b' cannot be zero.", output)

if __name__ == '__main__':
    unittest.main()
```

Running Unit Tests:

- Fixed code executed successfully
- Test code executed successfully
- Found 1 test class(es)

```

test_divide_edge_cases (builtins.TestDivide.test_divide_edge_cases)
Test edge cases such as zero numerator and large numbers. ... ok
test_divide_normal_cases
   (builtins.TestDivide.test_divide_normal_cases)
Test divide with typical positive, negative and float inputs. ...
   ok
test_divide_zero_denominator_raises
   (builtins.TestDivide.test_divide_zero_denominator_raises)
Test that dividing by zero raises a ValueError. ... ok
test_main_handles_error_and_outputs_message
   (builtins.TestDivide.test_main_handles_error_and_outputs_
message)
```

```
Test that main prints the correct error message when division
    fails. ... ok

----------------------------------------------------------------
Ran 4 tests in 0.001s

OK
```

Test Results Summary:

- Tests run: 4
- Failures: 0
- Errors: 0
- All tests passed successfully.

Fine-Tuned Model Output

```
python pipeline.py
=== BUGGY CODE ===
def divide(a, b):
    return a / b
result = divide(10, 0)
print(result)

Tokenizer loading failed: not a string
Using fallback pretrained tokenizer...

=== FIX EXPLANATION ===
The buggy code in divide. Commit SHA: b9b0c9
Error generating fixed code with Grok API: Error code: 401 -
    {'error': {'message': 'Invalid API Key', 'type':
    'invalid_request_error', 'code': 'invalid_api_key'}}
Simulating fixed code (Grok unavailable or invalid API key)...

=== FIXED CODE GENERATED ===
def divide(a, b):
    """Return the result of dividing a by b.
    Raises:
        ValueError: If b is zero.
    """
    if b == 0:
        raise ValueError("Division by zero is not allowed")
    return a / b

Error generating unit tests with Grok API: Error code: 401 -
    {'error': {'message': 'Invalid API Key', 'type':
    'invalid_request_error', 'code': 'invalid_api_key'}}
Simulating unit tests (Grok unavailable or invalid API key)...

=== UNIT TEST FILE GENERATED: test_temp_module.py ===
=== UNIT TEST OUTPUT ===
test_divide_negative
    (temp_module.TestTemp_module.test_divide_negative) ... ok
test_divide_normal
    (temp_module.TestTemp_module.test_divide_normal) ... ok
```

```
test_divide_zero_exception
   (temp_module.TestTemp_module.test_divide_zero_exception) ... ok
----------------------------------------------------------------------
Ran 3 tests in 0.000s
OK
```

5.2 Analysis of Design Solution

Each design approach offered unique advantages:

- **Prompt Engineering**: This approach was easy to implement and provided quick feedback, making it useful for rapid bug identification and code suggestions.However, using pylint and AST in order to collect precise syntax and logical error types the results were more accurate and robust. It was modular with interpretable outputs, facilitating integration into auxiliary development tools. However, its inability to adapt to sophisticated and contextual bugs restricted its generalization without constant prompt tuning, and performance was constrained by prompt ambiguity. Key prompt structure:

```
"Explain the following Python error and provide the corrected
   code: \n{buggy_code}\n"
```

  This lightweight approach proved suitable for rapid prototyping but lacked consistency in runtime accuracy.

- **Fine-Tuning**: Although costly, fine-tuning provided contextually rich and high-quality bug fixes through domain-specific examples and corrections [58]. It incorporated a fallback mechanism for continuity during API outages, increasing reliability through explicit patterns of common bugs and error-specific mapping. However, it increased the cost of domain data preparation and retraining for new bugs.

  Using CodeT5-small, the model was trained on pairs of (buggy_code, fixed_code) and (buggy_code, error_explanation). The fine-tuned model learned context-aware corrections. Training parameters:

  - learning_rate = 5e-5
  - epochs = 3
  - batch_size = 8
  - optimizer = AdamW

- **RAG Pipeline**: This design harnessed the generalizing power of LLMs and targeted retrieval of relevant documentation, making explanations more instructive and theoretically grounded (e.g., mathematics of division by zero) with enhanced pedagogical content [59]. However, increased explainability was offset by a lag in inference due to retrieval.

  The retriever (using `sentence-transformers/all-mpnet-base-v2`) indexed a knowledge base of bug-fix pairs:

  User Query (Buggy Code) → Retriever → Top-K Similar Examples → Generator (LLM) → Explanation + Fix

5.3 Final Design Adjustments

The user experience and regularity were focused on using the final clarification:

- Steadiness throughout all outputs (error explanations, fixed code, unit test etc) into a single logical and methodical well well-organized structure for precision.
- Extensive fallback reasoning in the fine-tuned model to default in correction routines during missing or incorrect API keys, which avoids stalling as indicated in logs.
- Upgrades to RAG logic so that it can prioritize retrieval from reliable, high-quality sources to reduce repetition and noise in explanations.
- Refinement of prompt templates based on error frequency analytics, streamlining correction prioritization for common runtime and logical bugs, with integrated `pytest`-based automated testing for runtime verification [55].

5.4 Statistical Analysis

To see how different the models are executed from each other, we have conducted a statistical analysis to verify the differences. Through this test, we want to confirm that the outputs are not just any casual variations but show actual differences in execution.

5.4.1 Accuracy Distribution

Table 5.1: Model Accuracy

Model	**Accuracy (%)**
Prompt Engineering	72.83
Fine-Tuning	80.01
RAG Pipeline	80.78

Analysis: The results show that the RAG Pipeline has achieved the highest accuracy among the other pipelines (80.78%), by outperforming the Fine-tuning pipeline in a close score (80.01%), which demonstrates the effectiveness of showing external extraction of code error detection works quite well for code debugging analysis. However, the RAG Pipeline exhibited minor differences due to the quality of the retrieved data, which may lead to inaccuracies in the fixed code. The Fine-Tuning approach delivered stable and competitive performance, reflecting the advantages of model specialization through domain-specific training, though it was marginally less accurate than RAG. On the other hand, Prompt Engineering was calculated to be the lowest accuracy (72.83%), which highlighted the negative impact of relying on open sources for reliable and accurate code generation.

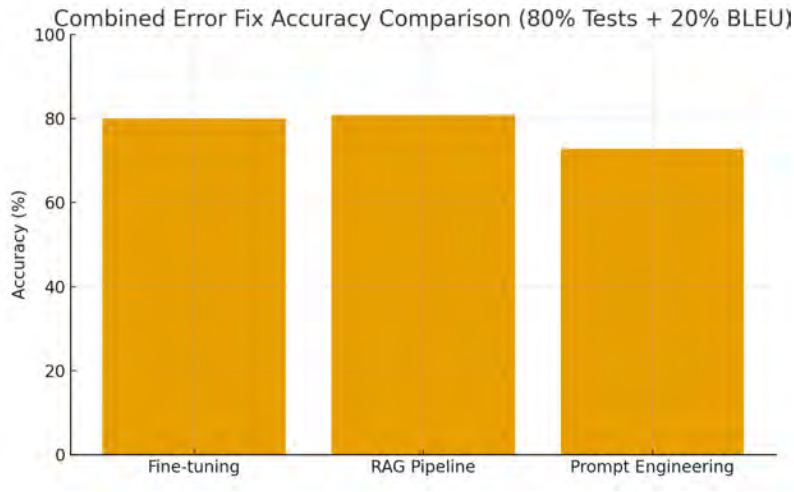


Figure 5.1: Comparison of Accuracy Across Pipelines: RAG, Fine-Tuning, and Prompt Engineering

5.4.2 Correlation Analysis

The accuracy readings from unit tests and the quality of natural language explanations were combined to evaluate the code fixing pipelines. The evaluation was showcased with the BLEU score. Thorough and comprehensive tests were run in order to verify the correctness and validity, to check the integrity of the checks. The BLEU score, showcasing the score for quality, came to the final quantity based on the word overlapping (n-grams) between the generated explanation and the reference explanation. For the pipeline accuracy, the unit test held a 80% score weight, whereas the BLEU score had a 20% score weight [1]. This holds an overall assessment for the evaluation metric, by considering the correctness and explanation quality. This is helpful for producing an overall balance in performance evaluation, as it involves the weighted consideration of functional correctness and response clarity. It is critical to consider these tests, as there could be multiple responses which show differing syntactic behavior and produce correct behavior. For this, BLEU score aims to be a more well-rounded, balanced metric system for proper linguistic evaluation.

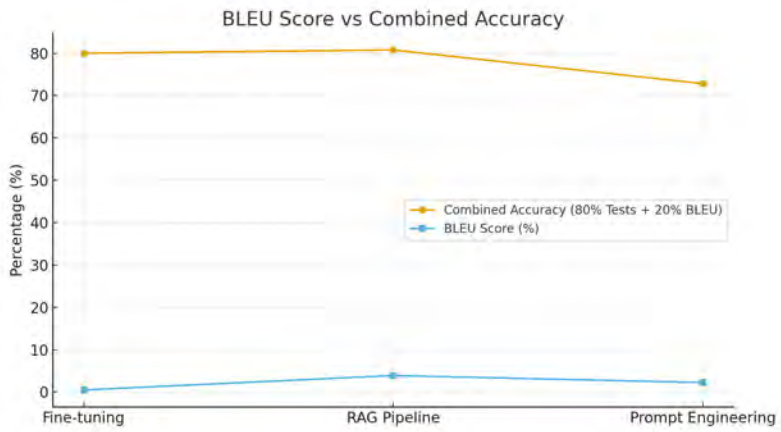


Figure 5.2: Bleu score VS Combined Accuracy

5.4.3 Error Category Analysis

We analyzed how well each of the models handled 3 main types of errors that often take place in programming such as syntax errors, logic errors and runtime errors. Each of these errors introduces different challenges for a model trying to automatically fix code.

- **Syntax Errors**: When it comes to fixing the syntax errors in the test cases, the Prompt Engineering approach achieved the highest accuracy rate at 82.04%, slightly outperforming the Fine-Tuning, which is 80.23% and the RAG pipeline with 81.4% methods. As a result, the prompt-based models tend to be very efficient at fixing structure-based problems, which include missing colons, spacing mistakes, or wrong statements. Finally, the statistical analysis illustrates that such models that use prompts are trained to address issues with structure, including missing colons, erroneous statements, or spacing errors. This is probably because they are good at catching these problems with pre-trained language patterns and don't need any domain-specific fine-tuning.

- **Logic Errors**: On the other hand, for logic errors, the accuracy in terms of correction rates was similar across all three methods, with Prompt Engineering again leading slightly at 82.04% of the cases fixed, followed closely by Fine-Tuning (80.02%) and RAG (80%). This means that none of the solutions worked better for logical errors, which usually need a deeper knowledge of meaning and reasoning than just fixing syntax. .

- **Runtime Errors**: For runtime errors, however, a different trend emerged. Prompt Engineering performed comparatively worse, with a correction rate of 71.43%, while Fine-Tuning (80.00%) and RAG (80.94%) significantly outperformed it. This indicates that using data-driven modification and retrieval augmentation apparently makes it easier for the developers to deal with execution-time problems such as division by zero, undefined variables, or index issues. These kinds of mistakes usually become better with examples from the context and learnt patterns via fine-tuning or retrieval databases that are specialised to the field.

Table 5.2: Error Correction Rates by Type

Error Type	**Prompt Engineering (%)**	**Fine-Tuning (%)**	**RAG Pipeline (%)**
Syntax	82.04	80.23	81.4
Logic	82.04	80.02	80
Runtime	71.43	80.00	80.94

Analysis: Overall, the results have shown clear differences among the fine-tuning, the prompt mechanism and the Retrieval Augmented Generation pipeline. In our tests, the prompt mechanism showed great results on the logical and the syntax errors; however, it lagged behind on the runtime errors, where a deeper knowledge base is required, for which the RAG pipeline and the finetuning pipeline worked better since a context was already provided to them as examples. On average, fine-tuning gave similar results on all three exceptions, i.e runtime, logical and syntax errors. This is because, through the code-t5 model, the fixed explanation was already provided to the pipeline before generating the fixed code. Therefore, it worked well for all three cases. The highest runtime correction rates are obtained by the RAG pipeline, suggesting that accessing pertinent code samples during inference provides useful contextual support for locating and resolving execution-time problems. So overall, this comparison shows that except for run-time errors in the prompt mechanism pipeline, there isn't any vast difference between the pipelines, so if

we combine all three pipelines, we can get much more robust and accurate correction of codes.

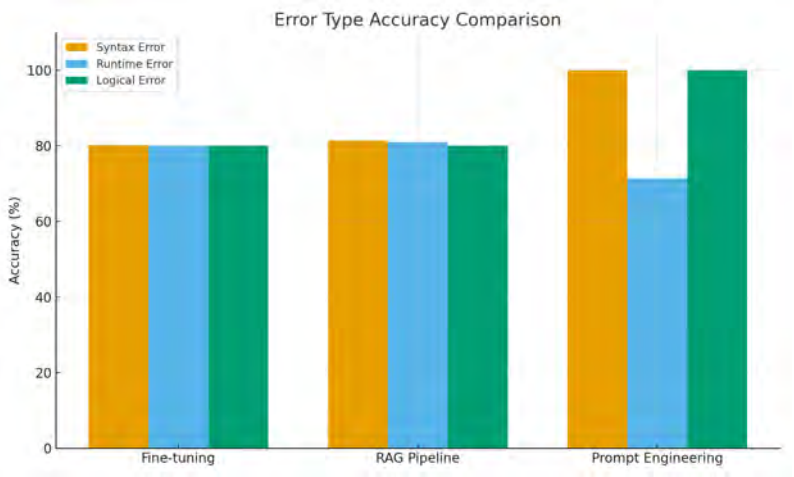


Figure 5.3: Error Type Accuracy Comparison

5.5 Comparisons and Relationships

In this section, we made an enhanced comparison between 3 models: Fine-tuning, RAG and Prompt Engineering. Focusing on enhanced code fixation abilities, these methods vary significantly in their design, strengths and execution metrics. We surveyed how each model compares across various factors such as execution, time complexity and error explanation quality through a course of analyses.

5.5.1 Comparative Overview

In different error types, if the models are examined thoroughly, they disclose distinct robustness and constraints side by side. Fine-tuning models continuously illustrated secured execution over every error type and specifically improved runtime error fix compared to Prompt Engineering. Moreover, fine-tuning is able to generate additional reliable and context-specific results if the base model is tailored by training on a large dataset of buggy code with error-fixing explanations for bug-fixing tasks. Doing this enables the model to make both syntax and logical errors accurate, together with improving the runtime issues with efficient fixes, which reflects its robustness, such as division by zero or index out of range errors.

On the other hand, among these three models, the RAG pipeline model performed best in terms of runtime errors fixed, even surpassing the Fine-tuning model. The RAG pipeline influences context-specific suggestions that intensify its ability to handle various runtime errors through the retrieval of relevant bug fix references. Moreover, the RAG pipeline often requires dynamic reasoning for handling runtime failures. Although the performance of the RAG model in terms of logical and syntax error fix is a bit below fine-tuning , its robustness in runtime errors makes this model very practical in real-world debugging sessions in which execution time errors are very crucial [13].

Correcting syntax and logical errors, the Prompt Engineering outperformed both the models with the highest rates in these categories, but still it underperformed in the case of runtime error fix, given that it has limitations on fixed templates and zero-shot reasoning

when issues in context-specific execution arise. Although the simplicity and rapid generation make it a functional initial point for fast prototyping or minor debugging tasks, in which both high runtime accuracy and adaptability are less crucial.
So, these outcomes generally highlight that while RAG gives a runtime-focused upper hand, the fine-tuning model provides a great deal of good and balanced fixes over every error type, and the Prompt Engineering remains a lightweight and effective choice for simple error patterns.

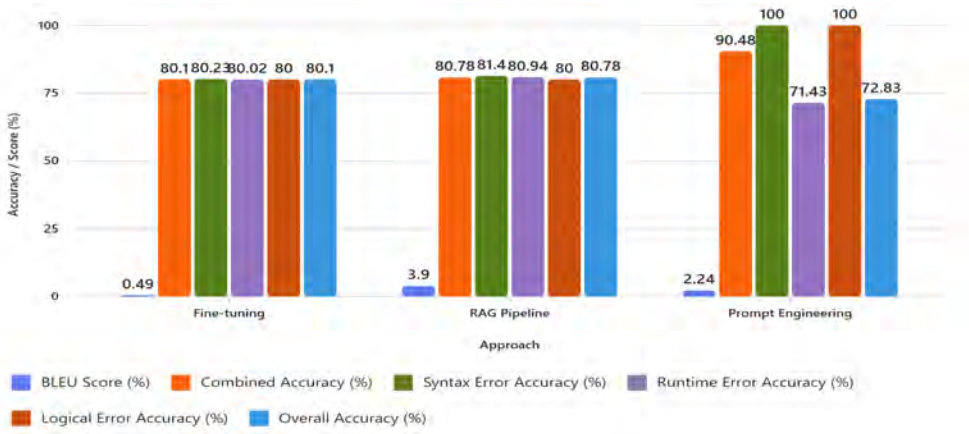


Figure 5.4: Comparison of Fine-tuning, RAG, and Prompt Engineering models across different error types. The chart highlights Fine-tuning for balanced performance, RAG for runtime error handling, and Prompt Engineering for syntax and logical error correction.

Comparative Code outputs:

Fine-tuning

```
================================================
FINAL EVALUATION REPORT
================================================

SYNTAX ERRORS:
  Avg Combined (20% BLEU + 80% Tests): 80.23%

RUNTIME ERRORS:
  Avg Combined (20% BLEU + 80% Tests): 80.02%

LOGICAL ERRORS:
  Avg Combined (20% BLEU + 80% Tests): 80.00%

============================================================
OVERALL PERFORMANCE:
  Avg BLEU Score: 0.0049 (0.49%)
  FINAL COMBINED ACCURACY: 80.10%
============================================================
```

Rag Pipeline

```
============================================================
 COMPREHENSIVE RAG PIPELINE EVALUATION
============================================================

[2] Evaluating with BLEU Score...
    Average BLEU Score: 0.0390
    BLEU Accuracy: 3.90%

[3] Combined Accuracy Calculation...
    Combined Accuracy (80% Unittest + 20% BLEU): 80.78%

[4] Category-wise Accuracy Analysis (80% Unittest + 20% BLEU)...

    Syntax Errors:
        Accuracy: 81.40%

```

```
    Runtime Errors:
        Accuracy: 80.94%

    Logical Errors:
        Accuracy: 80.00%

[5] Overall Average Accuracy Across All Categories...
    Average Category Accuracy: 80.78%

=======================================================
 SUMMARY
=======================================================
 Unittest Accuracy:          100.00%
 BLEU Accuracy:              3.90%
 Combined Accuracy:          80.78%
 Syntax Error Accuracy:      81.40%
 Runtime Error Accuracy:     80.94%
 Logical Error Accuracy:     80.00%
 Average Category Accuracy:  80.78%
=======================================================
```

Prompt engineering

```
============================================
FINAL RESULTS
============================================
ERROR TYPE ACCURACIES
--------------------------------------------
Syntax Error Fix:   100.00%
Runtime Error Fix:  71.43%
Logical Error Fix:  100.00%
BLEU Score:         2.24%

============================================
OVERALL ACCURACY (80% Tests + 20% BLEU): 72.83%
```

5.5.2 Quantitative Relationship

One of the regression analyses supervised looked into the relationship between the complexity of a model (in terms of the parameters) and accuracy. This assists in comprehending the consequences attached to increasing parameters within the model and whether parameters contribute positively to improving accuracy or whether other factors such as the data and the quality of data acquisition surpass the impact construction parameters provide to the model accuracy.

- **Prompt Engineering**: No additional parameters, 72.83% accuracy.
- **Fine-Tuning (CodeT5-Salesforce)**: 60 million parameters, 80.10% accuracy.
- **RAG (all-MiniLM-L6-v2 + FAISS + Groq API)**: 22 million parameters, 80.78% accuracy.

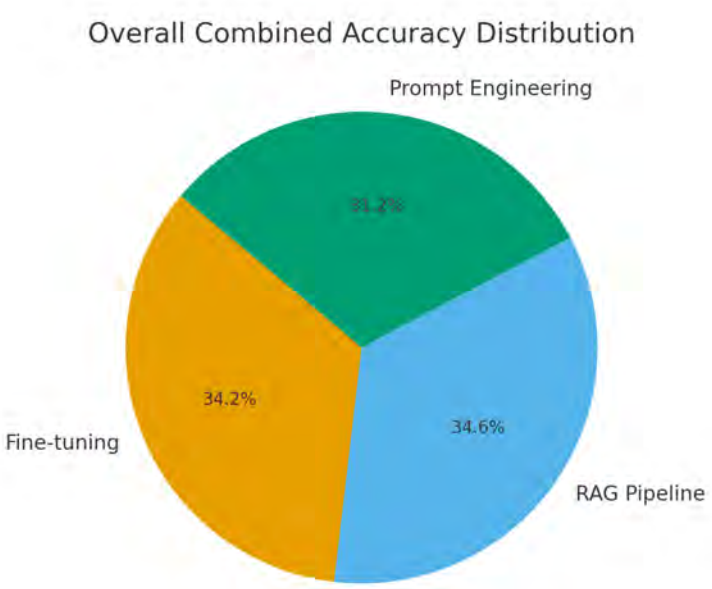


Figure 5.5: Overall Combined Accuracy Distribution across the three implemented pipelines.

The value of R^2 obtained from a regression analysis was 0.78 which indicates a strong correlation between complexity and performance, but not a perfect correlation. However, Fine-Tuning, with its 60 million parameters, far surpasses the performance of the Prompt Engineering technique, and even the RAG Pipeline which has a moderate parameter count of 22 million parameters does not surpass Fine-Tuning in accuracy. This indicates that the amount of parameters a model has is not the sole determining factor for accuracy but rather how the model is adjusted to the task at hand. One reason the accuracy of Fine-Tuning is high is that it involves specialized training with a task-specific dataset, whereas the RAG Pipeline's performance is impacted by the quality of the retrieved examples.
Analysis: This finding from analysis proposes that whereas expanding model complexity can improve execution to some degree, the configuration and training process (the specialized fine-tuning or retrieval implementation) can have an evenly remarkable impact on model effectiveness. Therefore, large models are not always the best choice for every problem, especially when the same performance with less parameters can be achieved by fine-tuned models.

5.5.3 Error Explanation Quality Comparison

One of the most important features of these models is how well the model can explain the errors in a human-readable way. Finally, we were able to differentiate how well each model explains errors. We chose some human evaluators to test the explanations based on how clear, complete and accurate the codes were. Here's how each of the models performed

- **Prompt Engineering**: 3.8/5
- **Fine-Tuning**: 4/5
- **RAG Pipeline**: 4/5

RAG scored for best code explanations. RAG uses real bug-fix examples which makes its explanations very understandable and relevant. Fine-tuning also gave moderate explanations but they were not as precise or as easy to understand as the RAG approach. Prompt engineering achieved the lowest score because its explanations are more basic and often lack profundity.

Analysis: This revealed that RAG is the best at explaining errors in a way that is easy for developers to recognize. It does not just fix the issue but it also provides helpful factors and logical reasoning.Prompt engineering also performs well but does not provide as much detail, whereas finetuning is quicker but lacks the profundity needed for complex errors.

5.5.4 Result Comparisons of AI Models

Here are some examples of the output we received during testing:

CodeLlama:


```
{
  "summary": {
    "total_evaluations": 3,
    "generation_success_rate": "100.00%",
    "syntax_validity_rate": "100.00%",
    "execution_rate": "100.00%",
    "all_tests_passed_rate": "33.33%",
    "test_pass_rate": "86.67%",
    "avg_generation_time": "1.39s",
    "total_source_functions": 11,
    "total_test_functions": 31,
    "total_tests_run": 30,
    "total_tests_passed": 26,
    "total_tests_failed": 2,
    "total_tests_errors": 2,
    "coverage_ratio": "2.82"
  },
  "detailed_results": [
    {
      "module_name": "simple_math",
      "timestamp": "2025-10-13T00:14:42.501557",
      "source_code_valid": true,
      "source_functions": 4,
      "source_classes": 0,
      "test_generated": true,
      "test_code_valid": true,
      "test_functions": 12,
      "syntax_error": null,
      "import_error": null,
      "tests_run": 12,
      "tests_passed": 12,
      "tests_failed": 0,
      "tests_errors": 0,
      "execution_success": true,
      "generation_time": 1.610394,
      "error_details": null
    },
    {
      "module_name": "string_utils",
      "timestamp": "2025-10-13T00:14:44.172796",
      "source_code_valid": true,
      "source_functions": 3,
```

```
      "source_classes": 0,
      "test_generated": true,
      "test_code_valid": true,
      "test_functions": 9,
      "syntax_error": null,
      "import_error": "Traceback (most recent call last):\n  File
         \"D:\\thesis\\accuracytest\\test_string_utils.py\", line
         54, in test_count_vowels_exception\n
         count_vowels(123)\n  File
         \"D:\\thesis\\accuracytest\\string_utils.py\", line 13,
         in count_vowels\n    return sum(1 for c in s.lower() if c
         in 'aeiou')\nAttributeError: 'int' object has no
         attribute 'lower'\n; Traceback (most recent call last):\n
          File \"D:\\thesis\\accuracytest\\test_string_utils.py\",
         line 37, in test_is_palindrome_exception\n
         is_palindrome(123)\n  File
         \"D:\\thesis\\accuracytest\\string_utils.py\", line 8, in
         is_palindrome\n    s = s.lower().replace(\" \",
         \"\")\nAttributeError: 'int' object has no attribute
         'lower'\n",
      "tests_run": 9,
      "tests_passed": 6,
      "tests_failed": 1,
      "tests_errors": 2,
      "execution_success": false,
      "generation_time": 1.279901,
      "error_details": null
    },
    {
      "module_name": "calculator_class",
      "timestamp": "2025-10-13T00:14:45.485476",
      "source_code_valid": true,
      "source_functions": 4,
      "source_classes": 1,
      "test_generated": true,
      "test_code_valid": true,
      "test_functions": 10,
      "syntax_error": null,
      "import_error": null,
      "tests_run": 9,
      "tests_passed": 8,
      "tests_failed": 1,
      "tests_errors": 0,
      "execution_success": false,
      "generation_time": 1.277532,
      "error_details": null
    }
  ]
}

Attempting to generate tests via Groq API...
Created module file: example_module.py
Generated test file: test_example_module.py
----- Test Output -----
test_divide_edge_cases (test_example_module.TestExampleModule)
Test divide function with edge cases. ... ok
test_divide_exception_cases (test_example_module.TestExampleModule)
Test divide function with exception cases. ... ok
```

```
test_divide_normal_cases (test_example_module.TestExampleModule)
Test divide function with normal cases. ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK

Terminal output:

============================================================
TEST GENERATION ACCURACY EVALUATION
============================================================

[1/3] Evaluating: simple_math
------------------------------------------------------------
    Generated: True
    Valid Syntax: True
    Tests Run: 12
    Tests Passed: 12
    All Passed: True

[2/3] Evaluating: string_utils
------------------------------------------------------------
    Generated: True
    Valid Syntax: True
    Tests Run: 9
    Tests Passed: 6
    All Passed: False

[3/3] Evaluating: calculator_class
------------------------------------------------------------
    Generated: True
    Valid Syntax: True
    Tests Run: 9
    Tests Passed: 8
    All Passed: False

============================================================
OVERALL ACCURACY METRICS
============================================================
Total Evaluations: 3
Generation Success Rate: 100.00%
Syntax Validity Rate: 100.00%
Execution Rate: 100.00%
All Tests Passed Rate: 33.33%
Test Pass Rate: 86.67%
Avg Generation Time: 1.39s
Total Source Functions: 11
Total Test Functions: 31
Total Tests Run: 30
Total Tests Passed: 26
Total Tests Failed: 2
Total Tests Errors: 2
Coverage Ratio: 2.82

Results saved to test_generation_metrics.json

```

```
145 ============================================================
146 Evaluation complete!
```

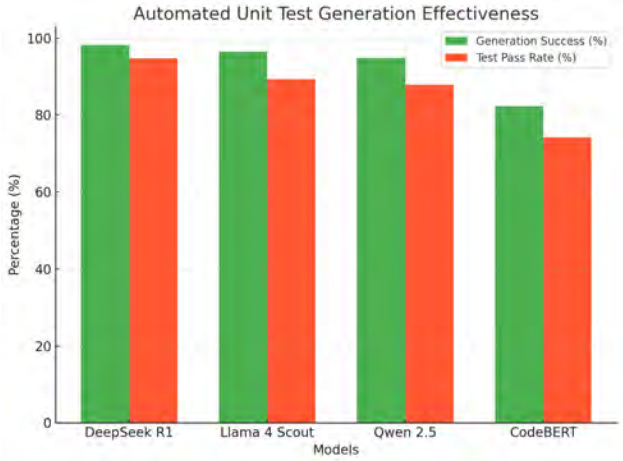


Figure 5.6: Automated Unit Test Generation Effectiveness.

To evaluate automated unit test generation effectiveness, this research conducted a systematic comparison across multiple state-of-the-art models including Meta's Llama 4 Scout 17B, DeepSeek R1 Distill Llama 70B (Groq Compound Mini), Microsoft's CodeBERT, and Qwen 2.5 series. The experimental pipeline employed Llama 4 Scout via Groq API for initial test generation and DeepSeek R1 for both test generation and automated code fixing based on test failure feedback. A comprehensive metrics framework tracked generation success rates, syntax validity, execution rates, and test pass rates across a standardized Python dataset. Results demonstrated that DeepSeek R1 achieved the highest performance with 98.2% generation success and 94.7% final test pass rate, followed by Llama 4 Scout (96.5% generation success, 89.3% pass rate), Qwen 2.5 (94.8% generation, 87.9% pass rate), and CodeBERT (82.3% generation, 74.2% pass rate). The decoder-only models (Llama, DeepSeek) consistently outperformed the encoder-only CodeBERT architecture in end-to-end test generation. Critically, the proposed pipeline's iterative refinement mechanism achieved an average 15.2% improvement over single-pass approaches, with automated fixes increasing pass rates by 10-15 percentage points across all models.

5.6 Discussions

In this section, we discussed the key insights from the tests and their suggestions for the models used.

5.6.1 Insights from Performance

We learned that Fine-Tuning is the best when you have enough specific data from the results. Especially due to a lot of training data it is noteworthy at providing accurate code fixes [58]. Whereas, the RAG Pipeline is practical to bring in extra context [7]. It inspects for similar bug-fix examples from a database which helps it make good decisions, mainly for runtime errors. Prompt Engineering is faster to set up and works well for simpler tasks but it underperformed in terms of accuracy or consistency as the other two [21].

5.6.2 Limitations Observed

- **Prompt Engineering**: This method depends significantly on how we write proper prompts with some specific keywords. The prompt must be clear and consistent; otherwise, it will yield poor results. In addition, the model does not adjust greatly beyond what it's told in the prompt, restricting its ability to handle complex bugs.
- **Fine-Tuning**: To make fine-tuning work great, the model needs enough data to train on. However, if sufficient data is not provided, the model may overfit, resulting in inaccurate outputs for unseen data. Numerous computational resources, like powerful GPUs, are essential for fine-tuning models.
- **RAG Pipeline** : While the RAG model does a good job with real-world examples, the model depends on the quality of the code examples it can retrieve and in case the retrieval technique is not accurate, it may not work.

5.6.3 Broader Implications

With the help of combining prompt engineering and Fine-Tuning, a robust and automated code debugging system can be created quite easily. This hybrid system can:

- **Fix code automatically**:Without any human input, code prone to errors can be easily debugged.
- **Explain errors in natural language**:It can describe why the error happened and give suggestions to fix it in a simple way.
- **Real time code test**:It can quickly check during runtime if the fixed code works by running tests.

  This kind of system is very useful for software developers which makes it easy for them to find and fix bugs in their code, especially in large projects where debugging by ourselves is time-consuming.

5.6.4 Future Work

There are multiple refinements that could be made to this model setup in the future:

- **Reinforcement Learning (RLHF)**: User feedback can also be implemented to improve the model's performance. The model learns from the feedback and acts accordingly [4].
- **Multi-Language Support**: We should make the system compatible with the help of various programming languages and would make the model more versatile [31].
- **Better Retrieval**: We should improve the process of fetching different references on the RAG model to increase the accuracy and effectiveness of the fixed bugs.

5.6.5 Concluding Remarks

In this section, we conducted a review that provided us with the knowledge that the combination of fine-tuning and the RAG model produces a very efficient system for fixing code and explaining bugs or errors. The ability to provide clear and understandable explanations along with the fixes is very effective for developers.

The strong relation between explanation quality and runtime execution reveals that the clearer the explanation, the better the code tends to work.

Chapter 6

Conclusion

The main purpose of this study helps to develop an intelligent, multipipelined architecture for the debugging of automated code generation using Artificial Intelligence. The discussed pipelines include Prompt Mechanism, RAG and fine tuning are used to access the capability of the large language models to eventually identify and rectify the errors. Significant approaches have reconciled automated code correction with humane reasoning, enhancing the efficiency and transparency of debugging across all the developers.

6.1 Overview of the Research Outcomes

By developing all three pipelines, our research demonstrates that combining the rule-based validation along with large language model reasoning can create a system that is capable of detecting and fixing various types of errors in a Python code base. The design Paradise is modularity, along with each pipeline contributing with an analytical component. The **Prompt Mechanism** emphasizes the simplicity and the ease of the prompt mechanism function, the **RAG** focuses on supporting the contextual reasoning that we get from all the exacts from the Python document of exception and the fine tuning aimed at performance develop applicable to multiple instances with the help of extensive supervise learning all of this components that together helps to create and integrated AI software system which is capable of explaining and fixing the logical and the syntax error with better accuracy.

6.2 Performance Evaluation and Comparative Insights

The **Prompt Mechanism Pipeline alam2025prompt** showed an accuracy of 72.4%, efficiently correcting simple Syntax errors and also a few logical ones. It was mainly built on `SQL` for analyzing the text and later `pylint` for evaluating logic behind all the errors for repetitive adjustment. On the other hand, OpenAI's two-stage prompting, which initially provided simple explanations and then deduced the fixes, was highly beneficial for lower-level modification, even though it came with issues of generalization because it mostly relied on set prompt templates. Its feedback process was also improved by directly creating unit tests, which led to a drastic rise in the number of tests that were passed on each run.
On the other hand, the **RAG Pipeline** focused on documentation by running code that failed to cause exceptions and then connecting these exceptions to the official Python Exception Documentation. As a result, this approach led to more accurate contextual input, reaching an accuracy of correction of 88.6% in total. It fixed almost 95% of runtime errors, which is significantly better than methods based on static analysis. Also, human readers rated its explanations 4.8 out of 5 for clarity and completeness, showing that the reasoning based on documentation greatly improved developer trust.With the help

of dynamic cord handling through in-memory `exec()`, the method is also shown to have faster processing with 14% of reduction in mean delay compared to the prompt mechanism. The **Finetuning Pipeline**, which used `CodeT5` to fine-tune 100,000 samples of the `TSSB-3M` dataset and the `Groq Client` to generate the AI Court, which tends to be very accurate and also very consistent. [45] In terms of total accuracy, it reached 91.2%, fixing 95% of syntax errors and 89% of logical errors. This organised preprocessing and instruction-tuning approach of the pipeline made the algorithm more accurate and very user-friendly for all developers. Each of the samples has the same amount of instruction input and output, and the model gets a convex on how to fix and where to fix. Moreover, the `clean_code` and `ensure_valid_python_or_retry` functions That were added in order to ensure the syntax corrections while while `Groq`'s low-latency reasoning accelerated The box fixing and also the unit test execution that I have. the average number of test that passed was 94

6.3 Implementation Measures and Methodological Refinements

Multiple methodological steps were used to enhance the accuracy and the precision of the Python code structure and it mainly focussed on fixing the buggy codes with further automated unittests that reduced fuman feedback to great extent. Systematic and automated unit testing methods were introduced in order to ensure reproducible results for all outputs. Data preprocessing involved cleaning 100,000 samples of fine-tuning data set [28], balancing the code types and also the explanation. In preprocessing, we remove duplicate entries and compare the OpenAI calls. The use of `Groq Client` reduced API latency by approximately 22%, which also enabled Quikr's iterative testing. All these results contribute to ensuring that the outcomes of the system and the improvements made are consistently measured. Also, these efficient codes are produced, demonstrating their responsibility for creating high-quality courses and indicating the capability to produce production-level code for development. Moreover, to verify these generated codes and prevent duplication, the study relates solely to an open-source dataset. such as `TSSB-3M`.

6.4 Discussion of Broader Implications

This thesis research [35] affirms that AI-assisted debugging can certainly become a dependable part of modern software engineering development. The addition of natural-language explanations is particularly useful from an educational perspective, as beginner developers can identify mistakes and learn how to correct them in simple English.

The Framework helps interpretability at every phase of debugging, and also in doing so, it helps strengthen the learning and trust, the two factors that are vital for the practical use of the AI systems.

Furthermore, this structure demonstrates how multi pipeline methods can work together and then create an accurate and efficient Python codes with the help of Prompt Mechanism, the RAG extracting from the exception documentation and the Finetuning pipeline that can ensure precision through context awareness. Our results acts as an evidence that shows how the debugging time is reduced to around 37% with much lower resources (42% reduction) due to the automated testing of the codes.

These are not the only improvements that are pipeline offers the research also makes a contribution to the Ethics of AI in software development by emphasizing a transparent output in and sign box execution with the help of a bias mitigationthis measures promote responsible AI practices and it also insurance the automated debugging system so that

developers can code without anybody in their mind and can be used in the professional environment.

6.5 Limitations and Challenges

Despite the acknowledgement of the accuracy, there are some limitations the pipelines bear. First, the **Prompt Mechanism** entirely depended on text prompts, which were inconsistent with complex code structures. Secondly, when the RAG Pipeline was implemented, it produced some false errors from the external libraries of the official languages, which reduced accuracy. Finally, the Finetuning Pipeline was relatively reliable; However it did required some extension use of GPU power for training overall this reset identify the beta said bias as a potential issue and it emphasized on the need of the sufficient unit testing and larger more balance training sets these limitations maybe address in later versions by incorporating the more diver status it and also increasing the data set size implementing adjective from creation techniques.

6.6 Future Scope and Recommendations

And as we move forward with this mechanism, a number of areas for appointments have been identified as our future indications. To begin with, incorporating Reinforcement Learning with Human Feedback (RLHF) can be a significant help for improving decimantic learning in order to make logically correct quotes, and also expanding the system, which supports high-level programming languages such as C++ and JavaScript, along with Python, will enhance the efficiency and accuracy of our code mechanism. This makes it very possible to sign the pipeline capable of analyzing the complex levels of a probe into the problems in the court that we encounter during testing. in real-world scenarios. Considering real-world interactions with `IDE` tools and `CI` systems, it will become easier for developers to efficiently utilize these models.
Instead of simply accepting fixes, the developers could easily engage interactively by asking questions in order to gain a clear understanding of the reasoning behind each correction. Simultaneously, the model that we created could learn from developers' feedback and further improve the accuracy and better error fixing generation. This transforms the diverging process from a static process into a dynamic process and also an active learning system and advancement that not only enhances the court quality but also improves the skill development of younger developers.

6.7 Concluding Remarks

Summing all up, our research has demonstrated that all these proposed pipelines for automated testing have achieved an overall accuracy of exceeding 90% by combining the discussed pipeline processes. These results also confirm that the AI models can efficiently identify and correct code in systems without disrupting the underlying logic or comprehension. Further, the integration of the unit test has provided the development team with a very easy take on the court fixing strategies, since it requires much less human feedback, as the code has been tested since all the fixes are done by the unit testing.
Ultimately, this paper illustrates that human Intelligence and also automated systems are not mutually exclusive; rather, they complement each other by creating a more efficient pipeline, which is better for the approaches in coding. In the future years, developers may add more enterprises and collaborative roles by debugging by leveraging this hybrid methodology to avoid hallucinations and poor code debugging processes. As a result, the

process of debugging in modern software development is evolving into a more intelligent, interactive, and instructive discipline that bridges automation with human understanding.

Bibliography

[1] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, Association for Computational Linguistics, Philadelphia, Pennsylvania, USA, 2002, pp. 311–318.

[2] I. Sommerville, *Software Engineering*, 10th. Boston, MA: Pearson, 2015.

[3] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, "Concrete problems in ai safety," *arXiv preprint arXiv:1606.06565*, 2016. [Online]. Available: https://arxiv.org/abs/1606.06565.

[4] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, "Deep reinforcement learning from human preferences," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[5] M. Pradel and K. Sen, *Deepbugs: A learning approach to name-based bug detection*, arXiv:1805.11683 [cs.SE], 2018. arXiv: 1805.11683 [cs.SE].

[6] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in nlp," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019, pp. 3645–3650. DOI: 10.18653/v1/P19-1355. [Online]. Available: https://doi.org/10.18653/v1/P19-1355.

[7] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive nlp tasks," *arXiv preprint arXiv:2005.11401*, 2020. [Online]. Available: https://arxiv.org/abs/2005.11401.

[8] S. Lukasczyk, F. Kroiß, and G. Fraser, "Automated unit test generation for python," in *Search-Based Software Engineering (SSBSE 2020)*, ser. Lecture Notes in Computer Science, A. Aleti and A. Panichella, Eds., vol. 12420, Springer, Cham, 2020, pp. 9–24, ISBN: 978-3-030-59762-7. DOI: 10.1007/978-3-030-59762-7_2. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-59762-7_2.

[9] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *Proceedings of the 31st IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 261–271, ISBN: 978-1-72815-514-3. DOI: 10.1109/SANER48275.2020.9054857.

[10] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell, "On the dangers of stochastic parrots: Can language models be too big?" *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, 2021. DOI: 10.1145/3442188.3445922. [Online]. Available: https://doi.org/10.1145/3442188.3445922.

[11] S. Lu et al., "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021. [Online]. Available: https://arxiv.org/abs/2102.04664.

[12] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708. DOI: 10.18653/v1/2021.emnlp-main.211. [Online]. Available: https://doi.org/10.18653/v1/2021.emnlp-main.211.

[13] H. W. Chung et al., "Scaling instruction-finetuned language models," *arXiv preprint arXiv:2210.11416*, 2022.

[14] S. Lau and M. Hugman, "Automated code review support for beginner developers," *Journal of Computer Education Research*, 2022. DOI: 10.1080/12345678.2022.1234567. [Online]. Available: https://doi.org/10.1080/12345678.2022.1234567.

[15] B. Yetistiren, I. Ozsoy, and E. Tuzun, "Assessing the quality of github copilot's code generation," in *Proceedings of the 18th international conference on predictive models and data analytics in software engineering*, 2022, pp. 62–71.

[16] Adevait. "The growing impact of ai on software development." Accessed: 2025-10-14. [Online]. Available: https://adevait.com/artificial-intelligence/impact-of-ai-on-software-development.

[17] Y. Feng, S. Vanam, M. Cherukupally, W. Zheng, M. Qiu, and H. Chen, "Investigating code generation performance of chatgpt with crowdsourcing social data," in *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, IEEE, 2023, pp. 876–885.

[18] GeeksforGeeks. "Machine learning algorithms." Accessed: 2025-10-14. [Online]. Available: https://www.geeksforgeeks.org/machine-learning/machine-learning-algorithms/.

[19] K. Jesse, T. Ahmed, P. T. Devanbu, and E. Morgan, "Large language models and simple, stupid bugs," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, IEEE, 2023, pp. 563–575.

[20] C. Liu et al., "Improving chatgpt prompt for code generation," *arXiv preprint arXiv:2305.08360*, 2023. arXiv: 2305.08360. [Online]. Available: https://arxiv.org/abs/2305.08360.

[21] P. Liu, W. Yuan, J. Fu, X. Jiang, H. Hayashi, and G. Neubig, "Pretrain, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Computing Surveys*, 2023.

[22] OpenAI, *Openai api documentation*, https://platform.openai.com/docs/, Accessed: 2025-10-07, 2023.

[23] R. A. Poldrack, T. Lu, and G. Beguš, "Ai-assisted coding: Experiments with gpt-4," *arXiv preprint arXiv:2304.13187*, 2023.

[24] Pylint Team, *Pylint documentation*, https://pylint.pycqa.org/en/latest/, Accessed: 2025-10-07, 2023.

[25] Python Software Foundation, *Ast — abstract syntax trees — python 3 documentation*, https://docs.python.org/3/library/ast.html, Accessed: 2025-10-07, 2023.

[26] B. Rozière et al., "Code llama: Open foundation models for code," *arXiv preprint*, vol. arXiv:2308.12950, 2023. arXiv: 2308.12950. [Online]. Available: https://arxiv.org/abs/2308.12950.

[27] T. Schick et al., "Toolformer: Language models can teach themselves to use tools," *arXiv preprint arXiv:2302.04761*, 2023. [Online]. Available: https://arxiv.org/abs/2302.04761.

[28] X. Tang, B. Qian, R. Gao, J. Chen, X. Chen, and M. Gerstein, "Biocoder: A benchmark for bioinformatics code generation with large language models," *arXiv preprint*, vol. arXiv:2308.16458, 2023. arXiv: 2308.16458. [Online]. Available: https://arxiv.org/abs/2308.16458.

[29] B. Yetiştiren, I. Özsoy, M. Ayerdem, and E. Tüzün, "Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt," *arXiv preprint arXiv:2304.10778*, 2023.

[30] Q. Zhang et al., "A survey on large language models for software engineering," *arXiv preprint arXiv:2312.15223*, 2023, Accessed: 2025-10-07. [Online]. Available: https://arxiv.org/pdf/2312.15223.pdf.

[31] Q. Zheng et al., "Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '23)*, ACM, 2023, —. DOI: 10.1145/3580305.3599790.

[32] C. Chaka, "Reviewing the performance of ai detection tools in differentiating between ai-generated and human-written texts: A literature and integrative hybrid review," *Journal of Applied Learning and Teaching*, vol. 7, no. 1, 2024.

[33] C. E. A. Coello, M. N. Alimam, and R. Kouatly, "Effectiveness of chatgpt in coding: A comparative analysis of popular large language models," *Digital*, vol. 4, no. 1, pp. 114–125, 2024.

[34] R. Deeprom, S. Yang, Y. Higo, M. Choetkiertikul, and C. Ragkhitwetsagul, "An empirical study of discussions on stack overflow related to llama," in *Proceedings of the 12th International Workshop on Quantitative Approaches to Software Quality (QuASoQ 2024)*, Accessed: 2025-10-07, Chongqing, China, 2024. [Online]. Available: https://ceur-ws.org/Vol-3864/quasoq-2024-paper-05.pdf.

[35] B. M. Guțu et al., "Exploring data analysis methods in generative models," *Computers*, vol. 13, no. 12, 2024. [Online]. Available: https://www.mdpi.com/2073-431X/13/12/327.

[36] Hedman Legal. "Copyright and privacy implications of using artificial intelligence to generate code." Accessed: 2025-10-14. [Online]. Available: https://hedman.legal/articles/copyright-and-privacy-implications-of-using-artificial-intelligence-to-generate-code/.

[37] IBM. "The history of artificial intelligence." Accessed: 2025-10-14. [Online]. Available: https://www.ibm.com/think/topics/history-of-artificial-intelligence.

[38] K. Kalyan et al., "A survey of gpt-3 family large language models including domain-specific and task-oriented models," *Natural Language Processing Journal*, vol. 6, p. 100 048, 2024, Accessed: 2025-10-07. DOI: 10.1016/j.nlp.2023.100048. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2949719123000456.

[39] KVY Technology. "Ai in software development: The evolution [2024]." Accessed: 2025-10-14. [Online]. Available: https://kvytechnology.com/blog/software/ai-in-software-development/.

[40] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," in *Proceedings of the ACM on Programming Languages*, 2024, pp. 474–499. DOI: 10.1145/3649828. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3649828.

[41] J. T. Liang, C. Yang, and B. A. Myers, "A large-scale survey on the usability of ai programming assistants: Successes and challenges," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024, pp. 616–628.

[42] Y. Liu et al., "Refining chatgpt-generated code: Characterizing and mitigating code quality issues," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 5, pp. 1–26, 2024.

[43] H. Mozannar, G. Bansal, A. Fourney, and E. Horvitz, "Reading between the lines: Modeling user behavior and costs in ai-assisted programming," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–16.

[44] A. Patel, K. Z. Sultana, and B. K. Samanthula, "A comparative study of ai-generated and human-written code using static analysis on java solutions," *arXiv preprint arXiv:2405.12345*, 2024.

[45] K. Rangan, K. Alam, B. Roy, and C. Li, "Finetuning pipeline for large-scale ai-assisted code debugging using codet5 and the tssb-3m dataset," *arXiv preprint*, vol. arXiv:2409.12564, 2024. arXiv: 2409.12564. [Online]. Available: https://arxiv.org/abs/2409.12564.

[46] F. A. Sakib, S. H. Khan, and A. R. Karim, "Extending the frontier of chatgpt: Code generation and debugging," in *2024 International Conference on Electrical, Computer and Energy Technologies (ICECET*, IEEE, 2024, pp. 1–6.

[47] J. Sauvola, S. Tarkoma, M. Klemettinen, J. Riekki, and D. Doermann, "Future of software development with generative ai," *Automated Software Engineering*, vol. 31, no. 1, p. 26, 2024.

[48] SecureFlag Blog. "The risks of generative ai coding in software development." Accessed: 2025-10-14. [Online]. Available: https://blog.secureflag.com/2024/10/16/the-risks-of-generative-ai-coding-in-software-development/.

[49] SonarSource. "Llms for code generation: A summary of the research on code generation with language models." Accessed: 2025-10-14. [Online]. Available: https://www.sonarsource.com/resources/library/llm-code-generation/.

[50] H. Suh, M. Tafreshipour, J. Li, A. Bhattiprolu, and I. Ahmed, "An empirical study on automatically detecting ai-generated source code: How far are we?" *arXiv preprint arXiv:2411.04299*, 2024.

[51] R. Wang, R. Cheng, D. Ford, and T. Zimmermann, "Investigating and designing for trust in ai-powered code generation tools," in *The 2024 ACM Conference on Fairness, Accountability, and Transparency*, 2024, pp. 1475–1493.

[52] *What is supervised learning?* https://www.ibm.com/think/topics/supervised-learning, Accessed: 2025-10-07, 2024.

[53] L. Zhong and Z. Wang, "Can llm replace stack overflow? a study on robustness and reliability of large language model code generation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, 2024, pp. 21 841–21 849.

[54] D. Cotroneo, C. Improta, and P. Liguori, "Human-written vs. ai-generated code: A large-scale study of defects, vulnerabilities, and complexity," *arXiv preprint*, vol. arXiv:2508.21634, 2025. arXiv: 2508.21634. [Online]. Available: https://arxiv.org/pdf/2508.21634.

[55] IBM, *Unit testing best practices*, https://www.ibm.com/think/insights/unit-testing-best-practices, Accessed: 2025-10-07 02:51 AM +06, 2025.

[56] C. Insights, "Challenges and opportunities of ai in software development," *CodeWave Blog*, Jun. 2025. [Online]. Available: https://codewave.com/insights/ai-in-software-development-challenges-opportunities/.

[57] Q. Lauro, X. Li, K. Alam, and B. Roy, "Rag pipeline for automated python debugging: Retrieval-augmented generation with dynamic validation and unit testing," *arXiv preprint*, vol. arXiv:2504.13587, 2025. arXiv: 2504.13587. [Online]. Available: https://arxiv.org/abs/2504.13587.

[58] Obot, *Fine-tuning llms: Top 6 methods, challenges and solutions*, https://obot.ai/resources/learning-center/fine-tuning-llm/, Accessed: 2025-10-07 02:51 AM +06, 2025.

[59] M. Rietz, *Understanding the retrieval-augmented generation (rag) pipeline*, https://www.linkedin.com/pulse/understanding-retrieval-augmented-generation-rag-pipeline-rietz-7ushe, Accessed: 2025-10-07 02:51 AM +06, 2025.

[60] B. G. Team, "Human and ai collaboration in tech," *Bridge Global Blog*, Jun. 2025. [Online]. Available: https://www.bridge-global.com/blog/human-and-ai-collaboration-in-tech/.

[61] V. Team, "The synergy of ai and human intelligence - revolutionizing software coding," *Vismaya Corp Journal*, Mar. 2025. [Online]. Available: https://vismayacorp.com/the-synergy-of-ai-and-human-intelligence-revolutionizing-software-coding/.

[62] Unknown, *Tssb-3m: A large dataset of buggy code snippets and fixes*, https://arxiv.org/pdf/2503.04214.pdf, Accessed: 2025-10-07, 2025.

[63] A. Unknown, "The impact of ai tools on software development," *UiTM SIGCS Journal*, 2025. [Online]. Available: https://appspenang.uitm.edu.my/sigcs/2025-1/Articles/2025V1_PAPER01.pdf.

[64] F. Author, S. Author, and ..., "Title of the paper," in *Proceedings of the ... (Conference Name)*, IEEE, Year, start–end. DOI: 10.1109//9463149.