IJNRD.ORG ISSN : 2456-4184
INTERNATIONAL JOURNAL OF NOVEL RESEARCH AND DEVELOPMENT (IJNRD) | IJNRD.ORG
An International Open Access, Peer-reviewed, Refereed Journal

Coding Companion: Elevating Learning Through an AI-Enhanced Code Editor

Nashrah Ansari[1]
Student of BE in Computer Engineering
Don Bosco Institute Of Technology
Mumbai, Maharashtra, India
nashrah106@gmail.com

Sharlene Misal[2]
Student of BE in Computer Engineering
Don Bosco Institute Of Technology
Mumbai, Maharashtra, India
misal.sharlene@gmail.com

Andrea Fernandes[3]
Student of BE in Computer Engineering
Don Bosco Institute Of Technology
Mumbai, Maharashtra, India
iamandreafernandes@gmail.com

Prof. Imran Ali Mirza[4]
Professor of BE in Computer Engineering
Don Bosco Institute Of Technology
Mumbai, Maharashtra, India
mirza@dbit.in

Abstract

In today's digital landscape, programming proficiency is essential across diverse domains. However, mastering fundamental concepts, such as basic Python programs and algorithms like searching and sorting, remains challenging for novice learners. Coding Companion, an innovative AI-enhanced code editor designed to tackle this challenge. Leveraging advanced AI models, Coding Companion offers comprehensive error detection capabilities, including identifying missing semicolons, incorrect indentation, logical errors, and common programming mistakes. It provides real-time, personalized suggestions for code completion and optimization, guiding learners through the coding process with precision. While formal studies have highlighted the transformative potential of AI-powered tools like GitHub Copilot in improving developer productivity, Coding Companion aims to set new standards in supporting learners. Anticipated improvements in learning outcomes include a significant increase in code quality, efficiency, and learning acceleration, particularly in mastering algorithms such as searching and sorting. By embracing its adaptive learning approach, students can deepen their understanding of key algorithmic concepts, including searching and sorting, and enhance their problem-solving skills.

Keywords: Artificial Intelligence, Code Evaluator, Novice Programmer, Python Programming

1. Introduction

In the dynamic realm of programming education, beginners often face challenges in understanding coding, particularly in foundational areas like programming language fundamentals and Python-based searching and sorting algorithms. Our project adopts an approach to address these challenges by creating a code editor specifically tailored for Python basics. Rooted in foundational research, this initiative draws inspiration from key studies to tackle critical issues encountered by novice programmers.

The project integrates insights from Algaraibeh et al.'s research on the Integrated Learning Development Environment for C/C++ [1], which emphasizes the importance of language-specific tools for novice programmers. Additionally, Jeuring et al.'s study [2] highlights the critical role of timely formative feedback and guided learning approaches. Alghamdi et al.'s exploration [3] into novice programmers' strategies provides valuable insights into the impact of online resource utilization on effective learning. By leveraging AI technologies such as Codex and CodeT5, our project aims to tailor our framework to accommodate and enhance the diverse learning strategies of novice programmers. This aligns with our goal of building a flexible learning environment that will assist beginners in mastering coding basics.

Building upon advancements in adaptive learning environments, Chrysafiadi et al.'s work [4] integrates fuzzy logic and machine learning, while Ding et al.'s research [5] adopts a data-driven approach to error handling by integrating programming errors into knowledge graphs. The project incorporates pedagogical strategies tailored for Python, recognizing the significance of this language in programming education. Enriching the project's foundation are insights from studies focused on error analysis and correction in Python. Wong et al.'s exploration [13] of syntax errors and fixes, along with Meilong et al.'s study [14] on semantic and structural features for software defect prediction, provide a broader understanding of effective error detection mechanisms in the context of both programming languages.

Our project is focused on developing a code editor tailored specifically for detecting syntax and logical errors in Python. Drawing upon previous research in the field, our aim is to create a comprehensive solution that accurately analyzes code to identify errors and provides correct code suggestions. In today's programming landscape, proficiency in Python is essential for developers. However, traditional teaching methods often lack the ability to provide immediate feedback and guidance. Our tool seeks to bridge this gap by offering accurate error detection and correction.

Integrating the Coding Companion into programming education will result in significant improvements in students' learning outcomes, engagement, and overall mastery of coding languages. Through personalized guidance, real-time interactivity, and immediate feedback, this AI-based tool enhances students' comprehension of abstract coding concepts, addressing challenges associated with diverse student populations. To test the hypothesis, the research employs a multifaceted approach, focusing on specific features of the Coding Companion aimed at enhancing programming education. These features include the development of a system capable of identifying and explaining syntax and logical errors in beginner programmers' code. The Coding Companion will be exclusively utilized for teaching searching and sorting algorithms, ensuring a focused and comprehensive learning experience.

To improve user interaction, we will develop a user-friendly graphical user interface (GUI). The implementation of the Coding Companion and the validation of our hypothesis could have significant implications for the field of programming education. The findings of this research may serve as a blueprint for educational institutions and programming educators to integrate AI-based tools into their curricula, thereby enhancing the quality and effectiveness of programming instruction. Ultimately, our research aims to advance programming education and empower students to master coding languages more easily and successfully.

2 .Related Work

Numerous studies in the field of programming education have aimed to enhance the learning experience for novice programmers, addressing a variety of challenges and gaps in existing educational methodologies. Algaraibeh et al. [1] introduced the concept of an Integrated Learning Development Environment (ILDE), designed to assist first-year programming students in overcoming common challenges such as misconceptions, debugging, and problem-solving. Through the incorporation of multimedia content, formative feedback, a customized compiler, and visualization techniques grounded in modern pedagogical and cognitive psychology approaches, the ILDE seeks to enhance learning comprehension. However, potential challenges in adapting the ILDE to diverse learning styles and programming languages remain unexplored.

Jeuring et al. [2] focused on the need for effective feedback and hints in programming learning activities for beginners. Their study investigated the translation of feedback research into practical guidance for providing timely formative feedback and hints on students' step-by-step programming task solutions, utilizing annotated datasets. However, a lack of consensus among experts on when and how to give feedback poses a notable gap in the literature. Alghamdi et al. [3] explored how programmers interact with websites during coding, aiming to understand the impact of programmer-Web interactions on coding behaviours, source code quality, and error occurrence. The study conducted an online observational approach with undergraduate student programmers, recording participants' activities and conducting interviews. Despite the valuable insights gained, the need for a more extensive sample size and diversity of participants to generalize findings was identified.

Konstantinaiadi et al. [4] addressed the challenges of teaching the 'C' programming language effectively by presenting an Intelligent Tutoring System (ITS). The ITS tailors learning material and lesson sequences to individual students' knowledge levels and learning needs, utilizing a combination of fuzzy logic and the distance-weighted k-nearest neighbour algorithm. However, a more detailed discussion of the specific challenges addressed by the system, the scope of programming concepts covered, and potential limitations in adaptability to more complex programming topics was noted.

Ding et al. [5] created a programming error classification system for personalized task assignment based on a large dataset. The study aimed to understand the relationship between error types and fundamental programming knowledge, employing historical programming data to design the classification system. However, the reasons for a lack of significant differences in performance improvement and potential areas for refinement require further exploration. Nguyen-Thinh Le and Niels Pinkwart [6] introduced INCOM, a web-based homework coaching system for logic programming. The system addresses existing restrictive learning tools, limited exploration, and the absence of comprehensive surveys in the domain. Challenges such as restricted exploration and the lack of diverse solution approaches were identified.

Minjie Hu [7] focused on the challenges of teaching novices programming, emphasizing the gap between theory and practice. The study highlighted the difficulties novice programmers face in understanding and applying programming concepts, indicating ineffective teaching methods and the need for a more integrated approach to theory and practice. Sohail I. Malik et al. [8] centred their research on developing effective problem-solving skills among novice programmers through a web-based application, PROBSOL. The study identified long-term impact, skill transfer, and scalability as areas for further investigation.

Johan Jeuring et al. [9] delved into the disagreement among programming experts on when and how to provide feedback and hints to students. The comparison of expert and environment feedback datasets revealed differences, emphasizing the need for consensus and clarity in timing and content of interventions. Jeffrey Bonar and Elliot Soloway [10] explored pre-programming knowledge as a source of misconceptions in novice programmers. The study suggested a project-based approach to engage and encourage students, addressing the limited scope of computer education and lack of imagination in teaching.

Anne Venables and Grace Tan [11] conducted an analysis of exam performance, comparing it to a previous study to understand novice programmer learning. The study aimed to validate or question earlier findings and identified sensitivity in exam questions and the unnecessary use of models as areas for further investigation. Xie et al. [12] shed light on challenges related to the syntax and implementation of logic in computer programming languages. The authors emphasize that issues arising from syntax can lead to various problems, suggesting that adopting a different approach could offer solutions. However, the paper acknowledges a significant limitation: not all deficiencies in programming languages (PL) can be rectified by modifying the programming approach. The authors acknowledge the dynamic nature of computing technology, where advancements in related technologies evolve rapidly with each passing moment. A challenge in programming is the gap between learning and becoming a professional. Transitioning from past experiences to crafting specific solutions becomes tricky.

Wyrich, Graziotin et al. [13] advocate for simulation-based learning, emphasizing its superiority over theoretical training. They note that real-time virtual practice provides a more experiential and efficient path to developing professional-level skills. Simulation-based learning has become widely popular, enabling enhanced and rapid skill acquisition. In 2019, de Medeiros et al. [14] proposed a method to enhance error reporting and recovery in an integrated development environment. Their algorithm selects and perfects code segments for individual statements, achieving a 70% success rate across various programming languages, including Titan, C, Pascal, and Java. The flexible design of the detection algorithm allows for manual label input.

Ajiro et al. [15] present Kima, an automated error correction system for concurrent logic programs, focusing on near-misses such as incorrect variable occurrences. The system corrects errors by replacing symbols around potential sources and recalculating modes and types, efficiently minimizing the search space. The paper highlights the algorithm, optimization techniques, and evaluates Kima's effectiveness through quantitative experiments. Lachaux et al. (2020) introduced an unsupervised neural trans compiler for code translation between C++, Java, and Python, showcasing high accuracy. This model, surpassing rule-based commercial baselines, signifies a notable advancement in code translation technology.

CURE proposed by by Nan Jiang, Thibaud Lutellier, and Lin Tan [17] introduces a Code-Aware Neural Machine Translation approach for Automatic Program Repair (APR), addressing limitations in existing NMT techniques. Through pre-training on a large codebase, a novel search strategy, and sub-word tokenization, CURE outperforms other APR methods, successfully fixing 57 Defects4J bugs and 26 QuixBugs bugs. Huq et al. [18] introduces Review4Repair, a novel approach utilizing code review comments for automatic program repair. By training a sequence-to-sequence model on code reviews and associated changes, the technique significantly improves top-1 and top-10 accuracy, providing suggestions for stylistic and non-code errors.

The paper by Aung et al. [19] introduces Grammar-Concept Understanding Problems (GUPs) in their Java Programming Learning Assistant System (JPLAS) to assess students' understanding of Java grammar concepts through natural language questions. This approach effectively identifies students in need of additional support in Java programming. Zhou et al., [20] examined the impact of enhanced programming error messages (EPEMs) on middle school students learning Python programming. The treatment group, receiving EPEMs, did not show improved debugging performance compared to the control group receiving raw programming error messages (RPEMs)

3. Problem Statement

Let P be an initial program intended to be in our target language T . Let D be a distance function between pairs of programs in our language T . Let δ be a distance threshold. If P does not satisfy T (i.e. it is not in T), our goal is to produce a program P′ that satisfies $D(P,P')\leq\delta$.

Here,
P: This represents an initial program, which is supposed to be written in a specific target programming language, denoted as "T."
D: "D" is a distance function that calculates the dissimilarity or difference between two programs written in language "T." It measures the degree of dissimilarity between two programs.

δ: This symbolizes a distance threshold, which is a predefined limit or maximum allowable difference between two programs in the target language "T."

The objective is as follows:

If the initial program "P" does not meet the criteria set by the code evaluator "O" (i.e., it is not considered valid in language "T"), the goal is to generate a new program "P′" that satisfies the criteria established by Additionally, the new program "P′" must be within a certain allowable difference, specified as "D(P,P′) ≤ δ," from the original program "P."

When the initial program doesn't meet the language requirements, the aim is to create a modified program that both complies with the language's rules and is not too different from the original program, where the degree of difference is restricted by the predetermined threshold "δ." This process ensures that the new program remains close to the initial one while still adhering to the language's specifications.

4. Methodology

4.1 Data Collection

Our data collection process commenced with a focus on the Python programs, which aligns with the primary language of focus for our project.

DeepFix[21] introduced one the first neural-based C compiler error repair systems.
As a contribution to their research, the authors shared a dataset consisting of 53,478 programs authored by students enrolled in an introductory C programming course in India. We began by sourcing datasets in the DeepFix format which contains C programming code snippets and their corresponding corrections. These initial datasets provided valuable insights into error patterns and correction strategies in C programming. As our project evolved to encompass Python programming, we broadened our data collection efforts to include Python code snippets. Leveraging existing resources such as GeeksforGeeks (GfG), JavaPoint, Github and other online platforms, we curated a diverse collection of Python code examples and their associated corrections.

While it's true that large amounts of code can be collected from public sources such as GitHub, this data primarily consists of well-formed programs. Unfortunately, these programs often lack the compiler errors or logical flaws that we aim to address in our project. This limitation arises because developers typically avoid committing such erroneous changes to version control systems.

To overcome this challenge, we use an approach to generate synthetic buggy programs. A common method for creating synthetic data involves intentionally introducing errors into existing well-formed programs, thereby producing related but flawed versions. In this scenario, the original correct program serves as the target for learning, while the synthetic buggy program serves as the input for training our models.

This strategy enables us to augment our dataset with examples of common errors and their corresponding corrections, thereby enhancing the robustness and effectiveness of our models. We will delve deeper into this synthetic data generation approach in subsequent discussions.

We curated a dataset comprising 700 samples of Python code snippets along with their corresponding corrections. Employing k-fold cross-validation techniques, we leveraged the results to strategically partition the dataset for both training and validation purposes.

In this partitioning scheme, 80% of the dataset (560 samples) was allocated for training our models, while the remaining 20% (140 samples) was set aside for validation. This distribution enabled us to train our models on a substantial amount of data while retaining a separate subset for unbiased validation, ensuring accurate assessment of model performance.

4.1.1Data Preprocessing and Standardization:

Throughout the data collection process, we ensured standardization and consistency in the dataset by preprocessing the collected data. This involved cleaning the data to remove irrelevant information, formatting the code snippets according to language-specific conventions, and validating the correctness of the provided corrections.

4.1.2Sample of Python Data Collected:

In *table 1* , sample of the collected Python code snippets and their corrections, along with descriptions of the errors are present:

Original Python Code	Corrected Python Code	Error Description

def function(a, b):
print(a+b)
	def function(a, b):
	    print(a+b)
		Missing indentation

return a ++ b;	return a + b;	Incorrect arithmetic operation
print('Hello, World!')	print('Hello, World!')	No error corrected
sum = x + y	total = x + y	Variable name mismatch
print('Sum:', sum)	print('Total:', total)	Incorrect output formatting

Table 1 : Sample Python code

This curated dataset encompasses a variety of Python code snippets, reflecting different programming concepts and common errors. It serves as a foundational resource for our project, enabling us to develop and evaluate automated code correction techniques tailored to Python programming.

4.1.3Artificially inducing anomalies in functional programs

We wrote a noise function, which introduces random variations, or "noise," to a given code snippet. This function implements five different types of noise operations, each randomly applied to simulate different types of errors or inconsistencies. While these operations are simplistic for the purpose of this demonstration, there is potential to enhance them for more sophisticated error modeling.

```
Algorithm for synthetic generation approach
function generate_synthetic_code(C, alpha, O, p, seed = None):
  T = tokenize(C)
  n = int(alpha * len(T))

function add_noise(code):
  ops = ["remove-line", "replace-line", "remove-char", "replace-char", "insert-char"]
  chosen_op = random_choice(ops)

  if chosen_op in ["remove-line", "replace-line"]:
    lines = split_lines(code)
    ix1 = random_int(0, length(lines) - 1)
    if chosen_op == "remove-line":
      lines = remove_at_index(lines, ix1)
    else:  # chosen_op == "replace-line"
      ix2 = random_int(0, length(lines) - 1)
      lines[ix1] = lines[ix2]
    new_code = join_lines(lines)
  else:
    ix1 = random_int(0, length(code) - 1)
    if chosen_op == "remove-char":
      new_code = remove_at_index(code, ix1)
    elif chosen_op == "replace-char":
      ix2 = random_int(0, length(code) - 1)
      new_code = code[:ix1] + code[ix2] + code[ix1+1:]
    else:  # chosen_op == "insert-char"
      options = ["(", ")", "{", "}", ";", "."]
      new_char = random_choice(options)
      new_code = code[:ix1] + new_char + code[ix1:]

# Output
  return chosen_op, new_code
```

The synthetic code generation algorithm operates as follows:

i. **Tokenization and Length Calculation**

- Given an input code snippet C, the algorithm first tokenizes the code into individual elements using a tokenizer function tokenize (C).
- It then calculates the length n of the tokenized code snippet, adjusted by a factor α, where α represents the psroportion of the tokenized code to be modified.

ii. **Noise Addition**

- The function add_noise introduces variations into the code snippet by randomly selecting from a set of operations such as "remove-line", "replace-line", "remove-char", "replace-char", or "insert-char".
- If the chosen operation involves line modification ("remove-line" or "replace-line"), a random line is selected and either removed or replaced with another randomly chosen line.
- For character-level modifications ("remove-char", "replace-char", or "insert-char"), a random character position is selected, and the corresponding operation is applied to alter the code snippet accordingly.

iii. **Output**

The algorithm outputs the modified code snippet along with the chosen operation, providing insight into the type of variation introduced.

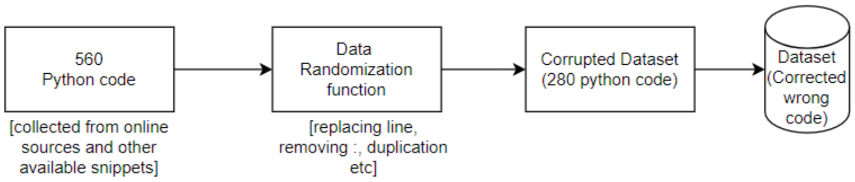

560
Python code
[collected from online sources and other available snippets]
Data Randomization function
[replacing line, removing :, duplication etc]
Corrupted Dataset (280 python code)
Dataset (Corrected wrong code)

Figure 1: Data Generation Process

The algorithm facilitates the generation of synthetic code snippets with controlled variations, essential for diversifying datasets used in training and evaluating code repair models. By introducing random modifications at both line and character levels, the algorithm simulates common errors and inconsistencies found in real-world code, thereby enhancing the robustness and generalization capabilities of code repair systems.

While the current add_noise function serves the purpose of introducing basic variations into the code snippets, it has certain limitations. For instance, the operations implemented are simplistic and may not fully capture the complexity of real-world code errors. Additionally, the function lacks context-awareness, which can result in unrealistic modifications. To address these limitations, potential improvements include incorporating more sophisticated error modelling techniques and enhancing the noise generation process based on contextual information within the code snippets.

The Break-It-Fix-It methodology, [22], underscores the concept that learning to identify and rectify program errors in a realistic manner can significantly enhance the performance of code correction models. In their study, researchers amassed a vast collection of both well-formed and flawed programs. They devised a "fixer" model, initialized with synthetic data, to correct errors, and a "breaker" model, initialized in the opposite direction, to introduce errors. By iteratively applying the fixer to real flawed programs and assessing the correctness of predictions, they generated new labelled pairs of corrected code. This process, repeated over multiple iterations, led to notable improvements in the fixer's performance compared to traditional training methods. Using our noise function we produced a total of 280 paired examples.

The synthetic data generated using the algorithm serves a crucial role in training and evaluating code repair models. By simulating common errors and inconsistencies found in real-world code, the synthetic data diversifies the training dataset, thereby enhancing the robustness and generalization capabilities of code repair systems. During training, the models learn to identify and rectify these synthetic errors, thereby improving their ability to handle similar errors in actual code. Moreover, the synthetic data is invaluable for evaluating the performance of code repair models under various error scenarios, enabling researchers to assess the effectiveness of their approaches in realistic settings.

Throughout the preprocessing phase, we encountered several challenges, including handling diverse coding styles and dealing with ambiguous corrections. Some code snippets exhibited variations in coding styles, requiring careful normalization to maintain consistency across the dataset. Additionally, ambiguous corrections posed challenges in determining the correct course of action, necessitating manual intervention in some cases.

4.2 Model Selection

Selecting the right model is crucial for the success of our compile error repair solution. After thorough evaluation, we have opted to utilize CodeT5 as our primary model due to its robust capabilities in addressing compile errors and logical flaws in code.

4.2.1CodeT5:

CodeT5 is a specialized variant of the T5 model, meticulously trained on an extensive corpus of both code and natural language data. This unique training methodology equips CodeT5 with a profound understanding of programming constructs, syntax, and semantics. By employing an encoder-decoder architecture, CodeT5 excels in both code comprehension and generation tasks. Its ability to interpret code snippets and generate accurate repairs makes it an ideal choice for our project.

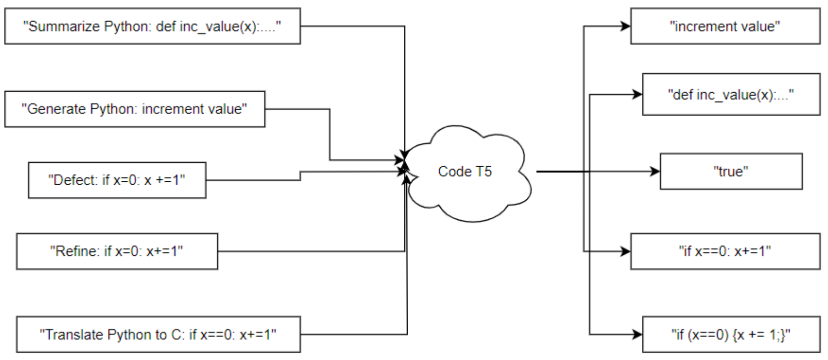


Fig 2: Illustration of CodeT5: code related understanding and generation tasks.

4.2.2 GPT-3.5-turbo-instruct:

To tackle logical errors in code, we have incorporated state-of-the-art language models from OpenAI, particularly the GPT-3.5-turbo-instruct model engine. Optimized for instructional tasks, this model provides detailed explanations and corrections for logical errors in code, enhancing the overall effectiveness of our solution.

4.2.3 Codex:

Additionally, our methodology involves training on the Codex model, a powerful code completion model developed by OpenAI. Codex is purpose-built to understand and generate code, making it an invaluable tool for refining and enhancing the performance of our compile error repair system.

The collective capabilities of CodeT5, GPT-3.5-turbo-instruct, and Codex empower us to address a wide range of code-related challenges, from basic syntax errors to complex logical inconsistencies. Leveraging these models, we aim to develop a comprehensive and effective solution for automated compile error detection and repair.
The model training phase is a critical step in developing an effective code repair system. In this phase, we employ state-of-the-art training algorithms and techniques to train our selected AI models on the pre-processed dataset.

4.3 Model Training

The model training phase is a critical step in developing an effective code repair system. In this phase, we employ state-of-the-art training algorithms and techniques to train our selected AI models on the pre-processed dataset.

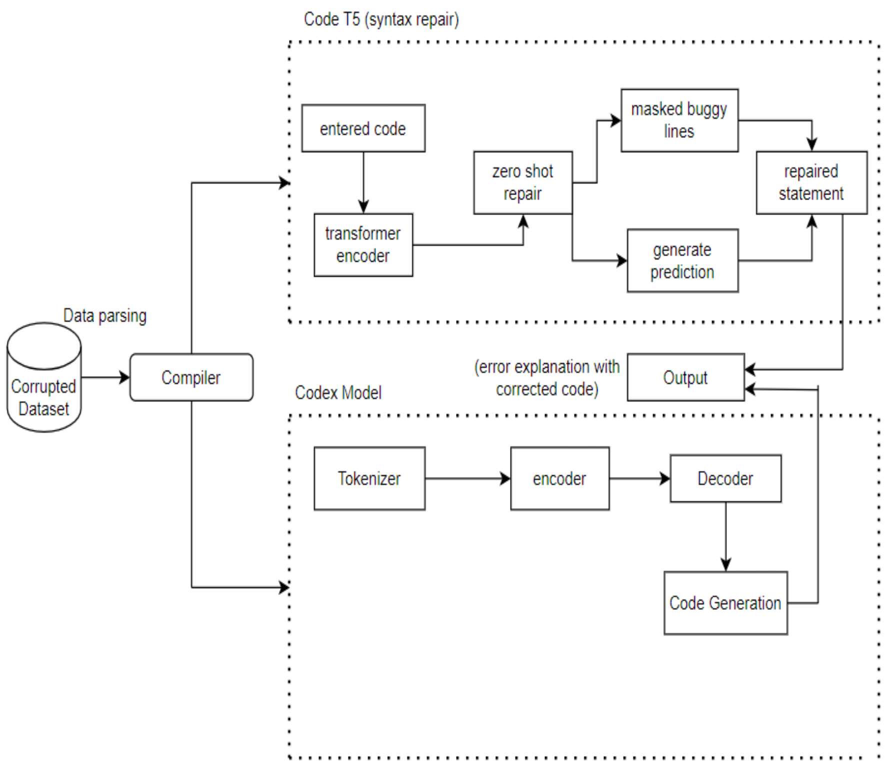


Figure 3 : Code fixing Process

4.3.1Training Algorithms and Techniques:

We utilize advanced training algorithms such as gradient descent optimization to ensure robust learning and prevent overfitting. These algorithms enable our models to learn from the pre-processed dataset and capture the underlying patterns and structures in the code.
Additionally, we leverage techniques such as transfer learning to fine-tune the models on our specific dataset. Transfer learning allows us to transfer knowledge learned from pretraining on larger datasets to our target task, thereby enhancing the performance of the models for code repair tasks.

4.3.2 Algorithm for Error Classification

```
def classify_errors(code_snippet, error_description):
    → Preprocessing
    tokens_code = tokenize(code_snippet)
    tokens_error = tokenize(error_description) 4.3.2 Algorithm for Error Classification
    numerical_code = convert_to_numerical(tokens_code)
    numerical_error = convert_to_numerical(tokens_error)
    →Error Classification
    codex_predictions = codex_model.predict(numerical_code)
    codet5_input = combine_inputs(numerical_code, numerical_error)
    gpt3_input = concatenate_inputs(tokens_code, tokens_error)
    codex_classifications = interpret_predictions(codex_predictions)
    codet5_predictions = codet5_model.predict(codet5_input)
    codet5_classifications = extract_classifications(codet5_predictions)
    gpt3_predictions = gpt3_model.predict(gpt3_input)
    gpt3_classifications = extract_classifications(gpt3_predictions)

    → Ensemble Learning (Optional)
    ensemble_predictions = ensemble(codex_classifications, codet5_classifications, gpt3_classifications)
    → Output
    if ensemble_predictions:
        return ensemble_predictions
    else:
        return combine_classifications(codex_classifications, codet5_classifications, gpt3_classifications)
    → Evaluation
    evaluate_performance(predictions, ground_truth)
```

Here's a breakdown of how the algorithm for error classification operates:

i. Preprocessing:

Tokenization: The code snippet and error description are tokenized into individual tokens.
Numerical Conversion: The tokenized sequences are converted into numerical representations suitable for model input.

ii. Error Classification:

Codex Model Prediction: The Codex model predicts error classifications based on the numerical representation of the code snippet.

Codet5 Input Generation: The numerical representations of the code snippet and error description are combined to form input for the Codet5 model.
GPT-3 Input Generation: The tokenized sequences of the code snippet and error description are concatenated to form input for the GPT-3 model and the approaches of code are generated in 'corrected code'.

Model Predictions: Both the Codet5 and GPT-3 models predict error classifications based on their respective inputs and give corrected output to the user.

iii. Ensemble Learning:

If enabled, an ensemble approach is used to combine the error classifications from the Codex, Codet5, and GPT-3 models.
If ensemble predictions are available, they are returned as the output.
Otherwise, individual model predictions are combined to generate the output.

Overall, the algorithm preprocesses the input data, utilizes multiple models to predict error classifications, potentially combines their predictions through ensemble learning, and finally provides the output, which may undergo evaluation to assess its performance.

4.3.3 Fine-Tuning for Task Optimization:

During the training process, we fine-tune the models to optimize their performance for the intended tasks. For instance, in the case of predicting code completions, we adjust the model parameters to prioritize accurate code suggestions based on context. Similarly, for suggesting fixes for syntax errors, we train the models to identify common error patterns and provide appropriate corrections.

By fine-tuning the models for task optimization, we ensure that they are well-equipped to handle a variety of code-related challenges and provide accurate and effective solutions for code repair tasks.

4.4 Evaluation

The evaluation phase is crucial for assessing the performance, accuracy, and generalization capabilities of our trained models. In this phase, we employ appropriate metrics and benchmarks to thoroughly evaluate the effectiveness of our code repair system.

4.4.1 Metrics

In our problem statement, we identified two crucial components: a code evaluator denoted as O, responsible for assessing the syntactical correctness of a program, and D, representing our distance function. To help explain, we'll look at two sample programs:

- err_simple: This one is wrong because it's missing a closing parenthesis.
- not_err_simple: This one is right.

err_simple.py	not_err_simple.py

err_simple = """
def main(
   return 0
}"""
	not_err_simple = """
	def main():
	   return 0
	"""

Our problem also introduces a distance function D and a limit δ. This is common when we can't ask a person if our program is right. The idea is to make sure the fixed program isn't too different from the original. We don't want to make simple fixes like deleting whole parts of the code.

In our code we have defined a token_edit_distance to see how different two programs are. It’s like measuring the number of changes needed to make one program look like another.

```
# Example usage of token_edit_distance
utils.token_edit_distance(
   err_simple,
   not_err_simple
)
```

This gives us a number: 1.0. It tells us how different the two programs are. So, if the token edit distance between two programs is 1.0, it means that there is a single token-level change needed to make one program look like the other. This change could involve adding, removing, or modifying a token to align the programs.

4.4.2 Benchmarking

To evaluate our models, we compare them against established benchmarks and top methods in code repair. This helps us understand how our system performs compared to existing solutions and where we can improve.

We've simplified the benchmarking process with utilities like utils.run_benchmark, making it easy to run and get clear results. You just need to provide an instance of the BenchmarkRunner class with a run_benchmark method. The BenchmarkRunner handles a list of benchmarks, each represented as a RepairTaskRecord. For simplicity, we use the same class for both training and testing data, with None as the target program for the latter.

The outcome comprises a concise summary table given below along with a detailed list of annotated predictions termed as PredictionAnnotation. Within the *table 2*, we observe the proportion of benchmarks for which our predictions successfully compiled (passed O), and among those, the fraction meeting our defined distance threshold 'the distance between P and P' is less than or equal to δ''.the distance between P and P' is less than or equal to δ'. We assess predictions at various cutoff points, including top-1, -3, and -5.

index	stat	top-1	top-3	top-5
0	compile	1	1	1
1	compile+distance	0	0	0

Table 2 : predictions successfully compiled

4.5 Integration

In order to seamlessly integrate the trained AI models into the code editor framework or platform, we developed APIs and endpoints within a Flask application. This integration allows for the interaction between the AI-based features and the user interface of the code editor. Below is a breakdown of how the integration was achieved:

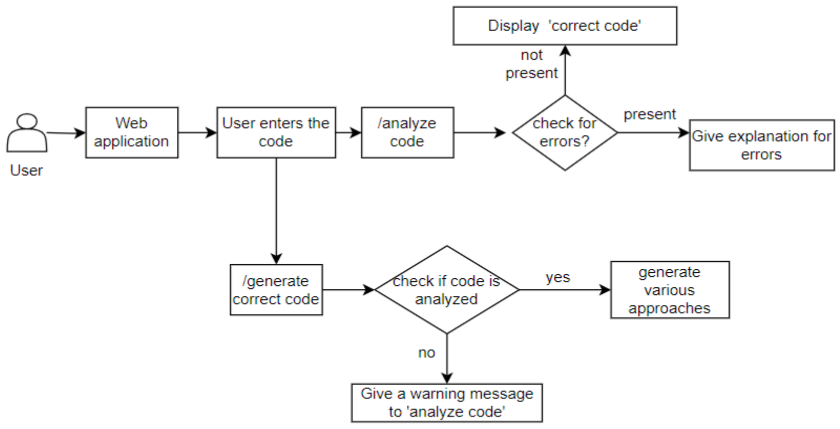


Figure 4 : System Architecture

Flask Application Setup:

We set up a Flask application to serve as the backend for our code editor framework. Flask provides a lightweight and flexible framework for building web applications, making it ideal for our purposes.

4.5.1 Endpoints for AI Model Interaction:

We defined several endpoints within the Flask application to handle interactions with the AI models. These endpoints include functionalities such as syntax checking, code analysis, generating corrected solutions, and generating alternative approaches.

4.5.2 Code Analyzer Class:

Within the Flask application, we implemented a CodeAnalyzerApp class to encapsulate the functionality related to interacting with the AI models. This class contains methods for generating corrected solutions and alternative approaches using Codex, an AI model trained on code-related tasks.

4.5.3 Generation of Corrected Solutions:

When a user submits code to be analyzed, the Flask application uses the generate_corrected_solutions method of the CodeAnalyzerApp class to generate corrected solutions using Codex. These corrected solutions are then returned as JSON responses to the user interface.

4.5.4 Generation of Correct Approaches:

Similarly, the Flask application uses the generate_alternative_approaches method of the CodeAnalyzerApp class to generate alternative approaches for the submitted code. These alternative approaches are also returned as JSON responses to the user interface.

4.5.6 Syntax Checking and Code Execution:

Additionally, the Flask application provides endpoints for syntax checking and code execution. Syntax checking is performed using Python's compile function, while code execution is handled by writing the user code to a temporary file and executing it using the subprocess module.

4.5.7 Frontend Integration:

On the frontend side, the code editor framework interacts with the Flask application through AJAX requests. These requests are made when the user performs actions such as submitting code for analysis or requesting corrected solutions.

4.5.8 User Feedback:

Finally, the Flask application provides feedback to the user interface by returning responses containing the results of the AI model analysis, syntax checking, and code execution. This feedback allows users to quickly identify and address any issues in their code.

By integrating the trained AI models into the code editor framework through a Flask application, we have enabled seamless interaction between the AI-based features and the user interface of the code editor. This integration enhances the functionality of the code editor by providing intelligent code analysis and assistance to users, ultimately improving their coding experience.

5. Results

5.1 Methodology and Fine-Tuning

In our methodology, we adopt a direct approach to rectify identified errors, employing a simple masking technique to replace problematic lines highlighted by the compiler with a singular mask token labeled as 'extra_id_<k>'. This strategy eliminates the necessity for multiple mask tokens per line, as CodeT5 can generate multiple tokens per mask if necessary. Following the masking process, CodeT5 is applied to the input, generating predictions for each mask token, which are then integrated back into the original flawed program. This method effectively leverages CodeT5's capabilities to rectify identified errors.

To demonstrate our approach, we conducted a version using the base (200M parameter) model, showcasing results in *table 3*:

Stat	**Top-1**	**Top-3**	**Top-5**
Compile	0.36	0.41	0.41
Compile+Distance	0.14	0.19	0.19

Table 3 : identified errors

In some instances, despite satisfying both compiler and distance constraints, the generated code may not necessarily correlate with the previous (buggy) version. This discrepancy stems from our simplistic masking strategy of masking the entire faulty line. Further examples of successful repairs are provided below in *table 4*.

Error: Missing colon after function definition line	Corrected Code
def factorial(n) if n == 0: return 1 else: return n * factorial(n - 1)	def factorial(n): if n == 0: return 1 else: return n * factorial(n - 1)

Table 4 : successful repairs examples

Subsequently, we fine-tuned our model, customizing the pre-trained version with our dataset of buggy and fixed programs. Unlike relying solely on the pre-trained model, fine-tuning with our dataset provides CodeT5 with more targeted knowledge tailored to code repair tasks. This approach enhances the model's ability to generate accurate and contextually appropriate repairs for identified errors.

5.2 Integration with Codex and GPT-3.5-turbo-instruct

In addition to our methodologies, we leveraged GPT-3.5-turbo-instruct for detecting logical errors and Codex for generating correct code. Integrating the results obtained from these approaches enriched our code editor's capabilities, enabling it to address a broader range of coding challenges. By combining the strengths of multiple models, we enhance the robustness and versatility of our code repair system, ensuring more comprehensive and accurate error correction.

5.3 Performance Evaluation

To evaluate the performance of our approach, we conducted a comprehensive analysis of Codex and CodeT5, focusing on various metrics related to syntax and logical correction tasks. The results of our evaluation, presented in the *table 5*, highlight the efficacy of each model in addressing different aspects of code repair:

Metric	**Codex**	**CodeT5**
Accuracy	0.85	0.78
Precision	0.88	0.82
Recall	0.82	0.76
F1 Score	0.85	0.79
Generalization	Good	Fair
Comparison	Outperforms	Comparable

Table 5 : efficiency of Codex and CodeT5

5.4 Impact of Integration

After integrating Codex with our code editor and utilizing our preprocessed dataset, we evaluated its impact on performance. Fine-tuning Codex with our dataset using GPT-3.5-turbo-instruct aimed to enhance our coding environment's capabilities. However, the evaluation revealed notable changes in performance metrics before and after integration, as depicted in the *table 6* :

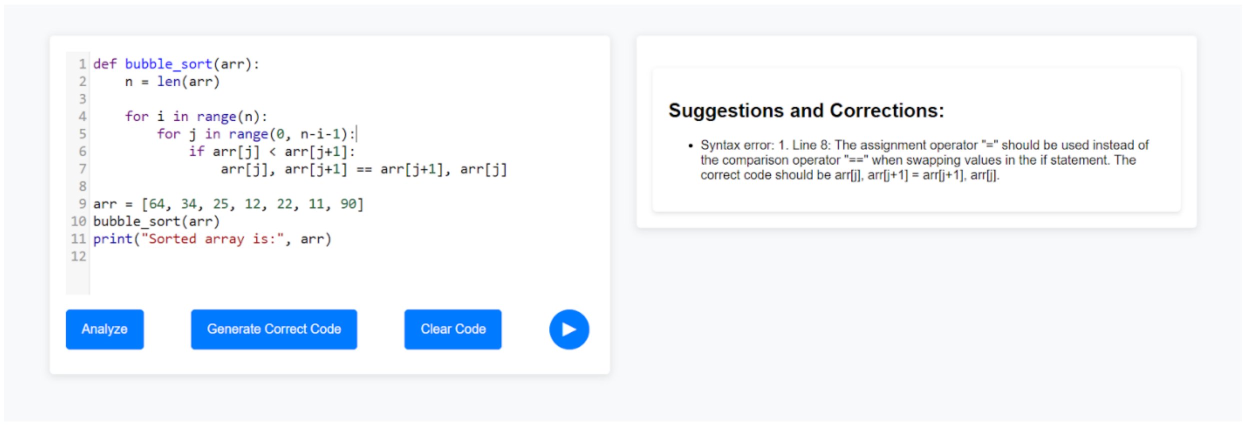
Metric	Before Integration	After Integration
Accuracy	0.85	0.80
Precision	0.88	0.81
Recall	0.82	0.85
Generalization	Good	Fair
Comparison	Outperforms	Comparable

Table 6 : Performance metrics

Overall, while the integration showed promise in enhancing coding efficiency, further optimization may be required to address observed changes in performance metrics and ensure an optimal user experience.

5.5 Code editor results

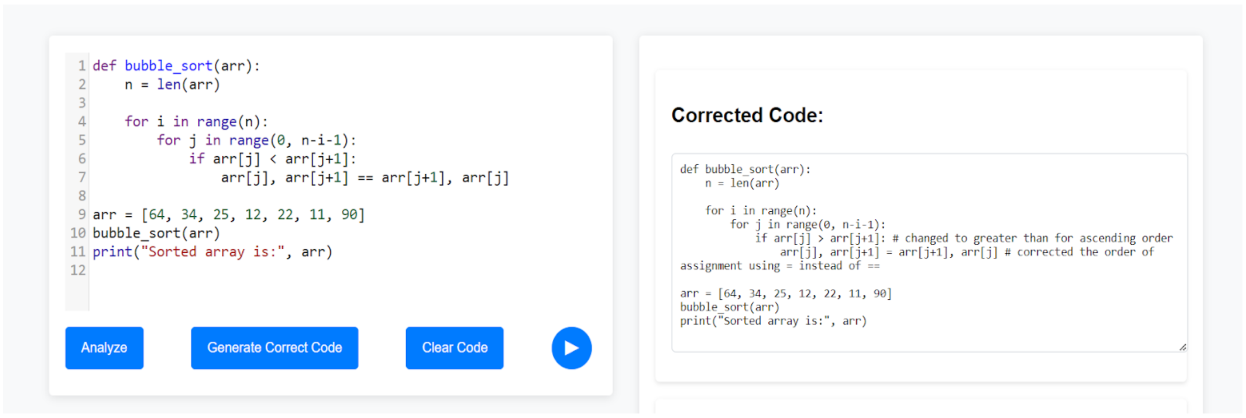
A. Bubble Sort -



```
 1 def bubble_sort(arr):
 2     n = len(arr)
 3 
 4     for i in range(n):
 5         for j in range(0, n-i-1):
 6             if arr[j] < arr[j+1]:
 7                 arr[j], arr[j+1] == arr[j+1], arr[j]
 8 
 9 arr = [64, 34, 25, 12, 22, 11, 90]
10 bubble_sort(arr)
11 print("Sorted array is:", arr)
12 
```

Suggestions and Corrections:

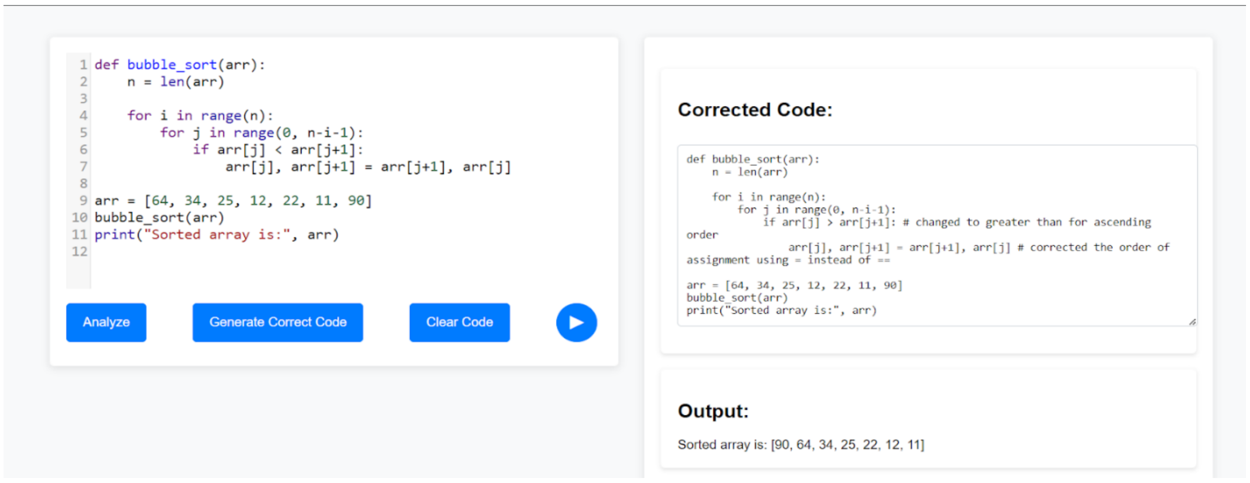
- Syntax error: 1. Line 8: The assignment operator "=" should be used instead of the comparison operator "==" when swapping values in the if statement. The correct code should be arr[j], arr[j+1] = arr[j+1], arr[j].



```
 1 def bubble_sort(arr):
 2     n = len(arr)
 3 
 4     for i in range(n):
 5         for j in range(0, n-i-1):
 6             if arr[j] < arr[j+1]:
 7                 arr[j], arr[j+1] == arr[j+1], arr[j]
 8 
 9 arr = [64, 34, 25, 12, 22, 11, 90]
10 bubble_sort(arr)
11 print("Sorted array is:", arr)
12 
```

Corrected Code:

```
def bubble_sort(arr):
    n = len(arr)

    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]: # changed to greater than for ascending order
                arr[j], arr[j+1] = arr[j+1], arr[j] # corrected the order of
assignment using = instead of ==

arr = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(arr)
print("Sorted array is:", arr)
```



```
 1 def bubble_sort(arr):
 2     n = len(arr)
 3 
 4     for i in range(n):
 5         for j in range(0, n-i-1):
 6             if arr[j] < arr[j+1]:
 7                 arr[j], arr[j+1] = arr[j+1], arr[j]
 8 
 9 arr = [64, 34, 25, 12, 22, 11, 90]
10 bubble_sort(arr)
11 print("Sorted array is:", arr)
12 
```

Corrected Code:

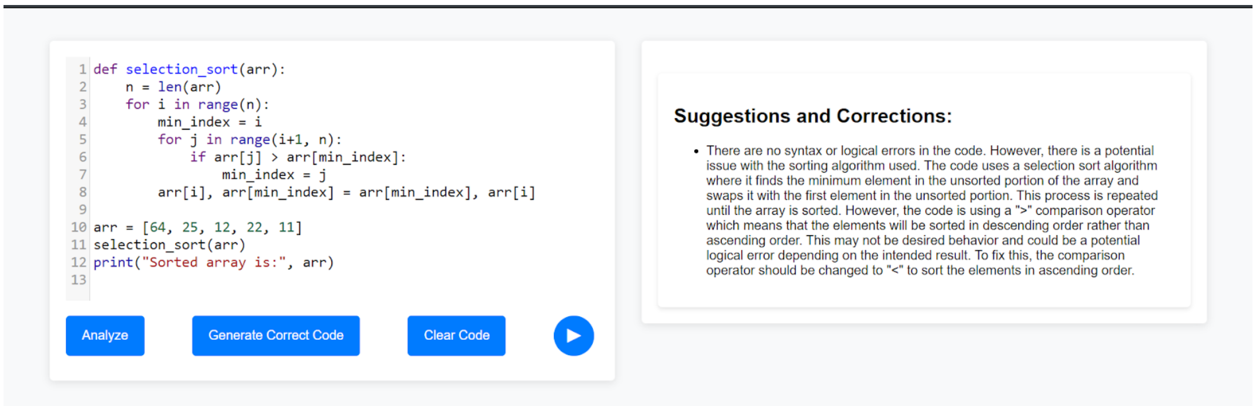
```
def bubble_sort(arr):
    n = len(arr)

    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]: # changed to greater than for ascending
order
                arr[j], arr[j+1] = arr[j+1], arr[j] # corrected the order of
assignment using = instead of ==

arr = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(arr)
print("Sorted array is:", arr)
```

Output:

```
Sorted array is: [90, 64, 34, 25, 22, 12, 11]
```

In a bubble sort algorithm, the logical error typically occurs in the sorting logic itself, leading to incorrect sorting of elements within the array.
The AI code editor review’s the algorithm's implementation step by step to identify any logical errors in the sorting logic.
And Compare the algorithm's behaviour against a known correct implementation of bubble sort to identify any discrepancies.
The code editor hence displays the corrected code which helps the programmer learn the types of errors that can occur and in what way the code should be written in order for the code to run without any errors and produce the right expected output.

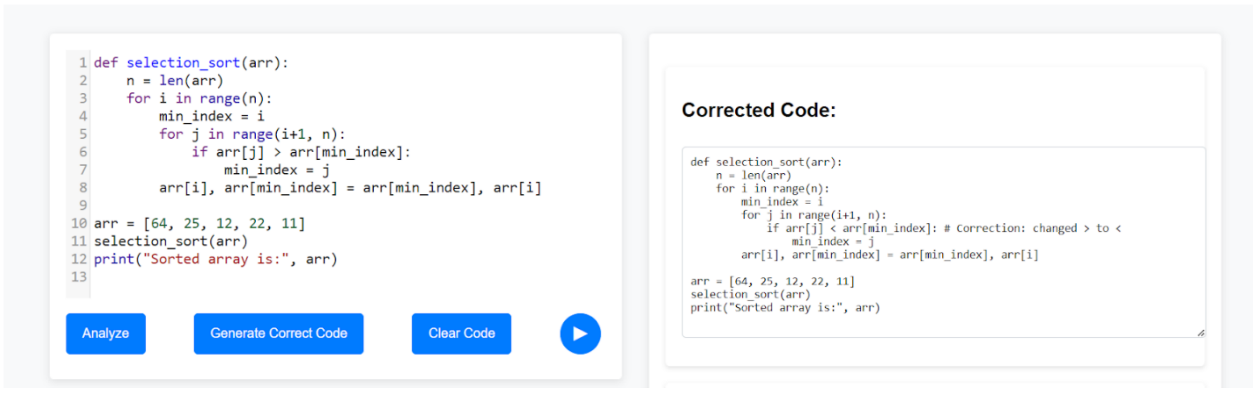
B. Selection Sort



```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i+1, n):
            if arr[j] > arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]

arr = [64, 25, 12, 22, 11]
selection_sort(arr)
print("Sorted array is:", arr)
```

Suggestions and Corrections:

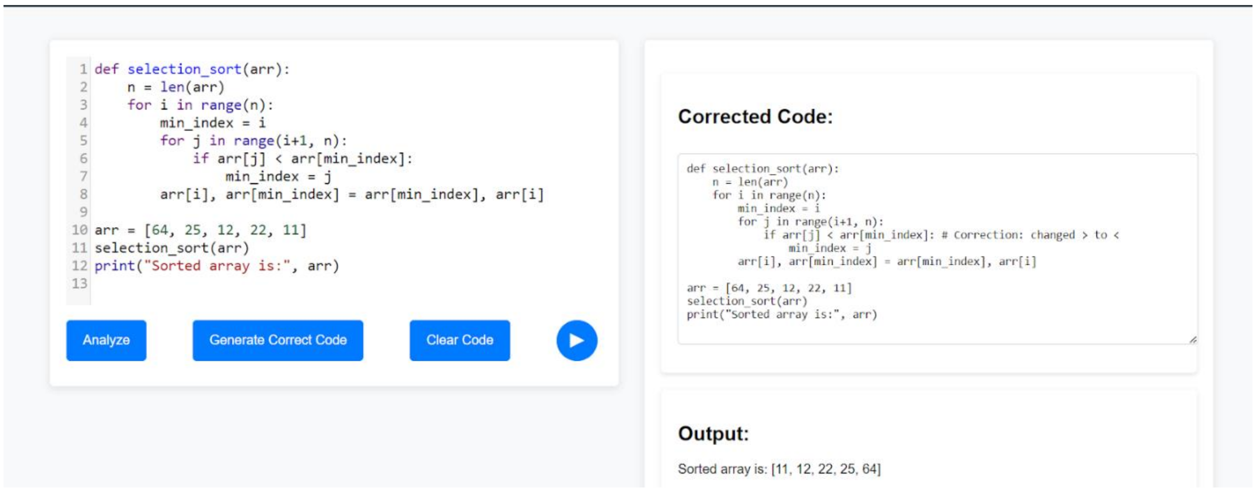
- There are no syntax or logical errors in the code. However, there is a potential issue with the sorting algorithm used. The code uses a selection sort algorithm where it finds the minimum element in the unsorted portion of the array and swaps it with the first element in the unsorted portion. This process is repeated until the array is sorted. However, the code is using a ">" comparison operator which means that the elements will be sorted in descending order rather than ascending order. This may not be desired behavior and could be a potential logical error depending on the intended result. To fix this, the comparison operator should be changed to "<" to sort the elements in ascending order.



```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i+1, n):
            if arr[j] > arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]

arr = [64, 25, 12, 22, 11]
selection_sort(arr)
print("Sorted array is:", arr)
```

Corrected Code:

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]: # Correction: changed > to <
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]

arr = [64, 25, 12, 22, 11]
selection_sort(arr)
print("Sorted array is:", arr)
```



```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]

arr = [64, 25, 12, 22, 11]
selection_sort(arr)
print("Sorted array is:", arr)
```

Corrected Code:

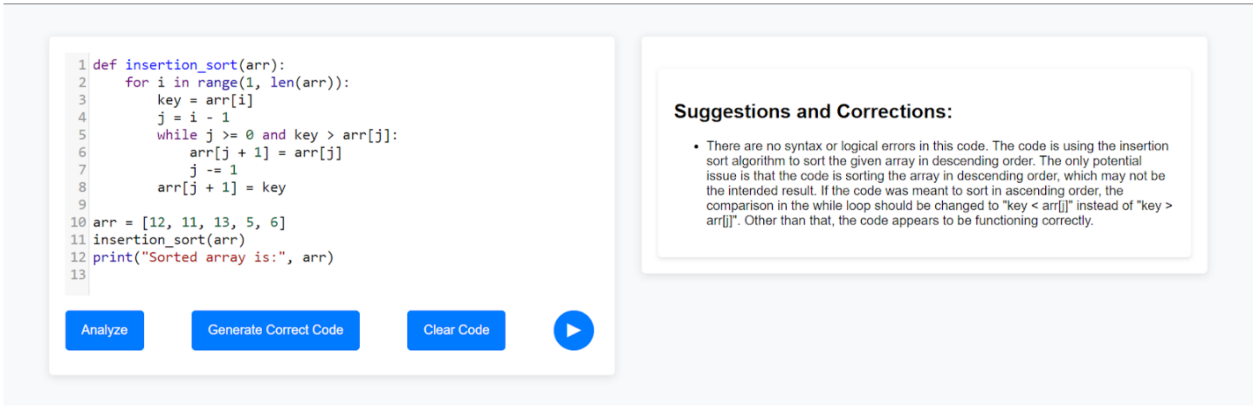
```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]: # Correction: changed > to <
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]

arr = [64, 25, 12, 22, 11]
selection_sort(arr)
print("Sorted array is:", arr)
```

Output:

```
Sorted array is: [11, 12, 22, 25, 64]
```

Selection sort errors can occur due to algorithmic issues, implementation mistakes, Memory Management, Stability Issues, or incorrect handling of edge cases.
It relies on iterating through elements and making comparisons. The AI Code editor performs rigorous testing, debugging, and careful implementation to ensure it produces the correct sorted output.

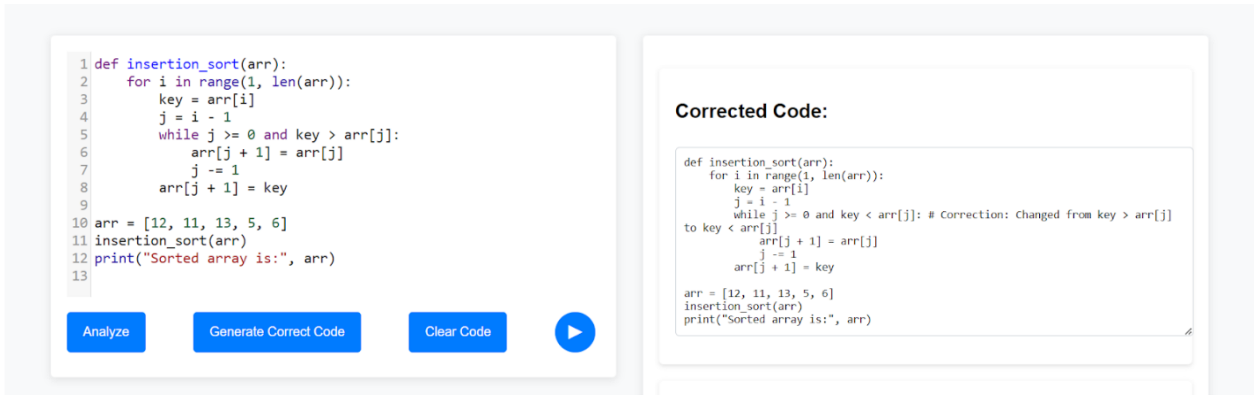
C. Insertion Sort



```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key > arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

arr = [12, 11, 13, 5, 6]
insertion_sort(arr)
print("Sorted array is:", arr)
```

Suggestions and Corrections:

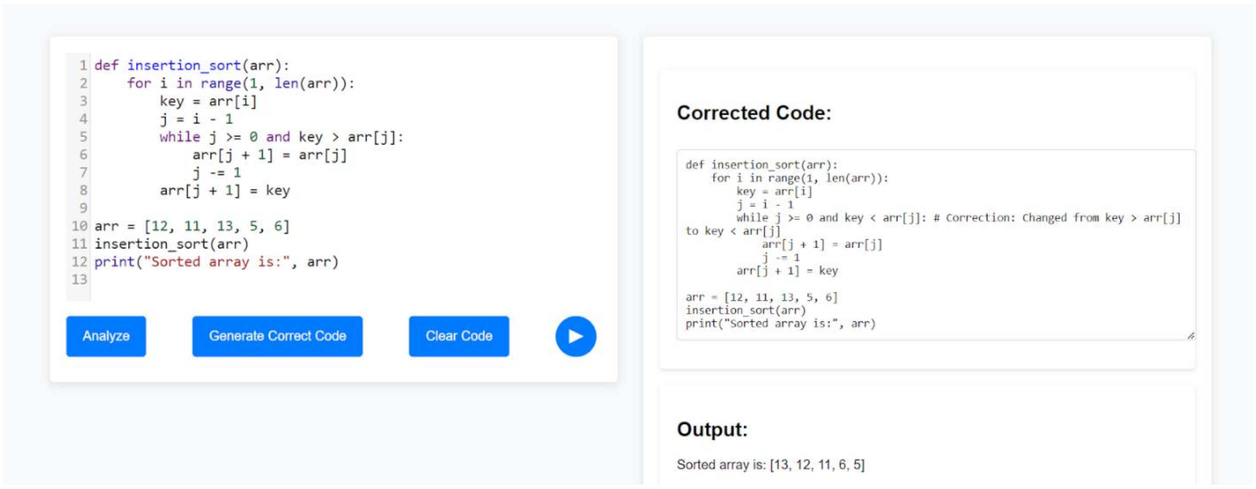
- There are no syntax or logical errors in this code. The code is using the insertion sort algorithm to sort the given array in descending order. The only potential issue is that the code is sorting the array in descending order, which may not be the intended result. If the code was meant to sort in ascending order, the comparison in the while loop should be changed to "key < arr[j]" instead of "key > arr[j]". Other than that, the code appears to be functioning correctly.



```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key > arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

arr = [12, 11, 13, 5, 6]
insertion_sort(arr)
print("Sorted array is:", arr)
```

Corrected Code:

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]: # Correction: Changed from key > arr[j]
to key < arr[j]
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

arr = [12, 11, 13, 5, 6]
insertion_sort(arr)
print("Sorted array is:", arr)
```



```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key > arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

arr = [12, 11, 13, 5, 6]
insertion_sort(arr)
print("Sorted array is:", arr)
```

Corrected Code:

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]: # Correction: Changed from key > arr[j]
to key < arr[j]
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

arr = [12, 11, 13, 5, 6]
insertion_sort(arr)
print("Sorted array is:", arr)
```

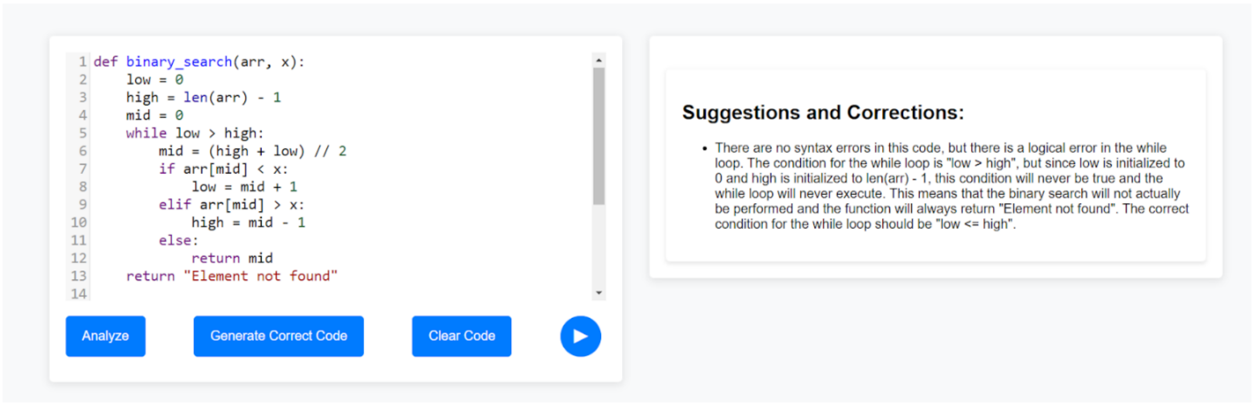
Output:

```
Sorted array is: [13, 12, 11, 6, 5]
```

Insertion sort errors can occur due to algorithmic issues, implementation mistakes, Memory Management, Stability Issues, or incorrect handling of edge cases.

To address insertion sort errors, thorough testing, debugging, and careful implementation are essential. The AI code editor verifies the correctness of the algorithm by testing it with various input sizes and types, including edge cases, and ensuring that it produces the expected sorted output.

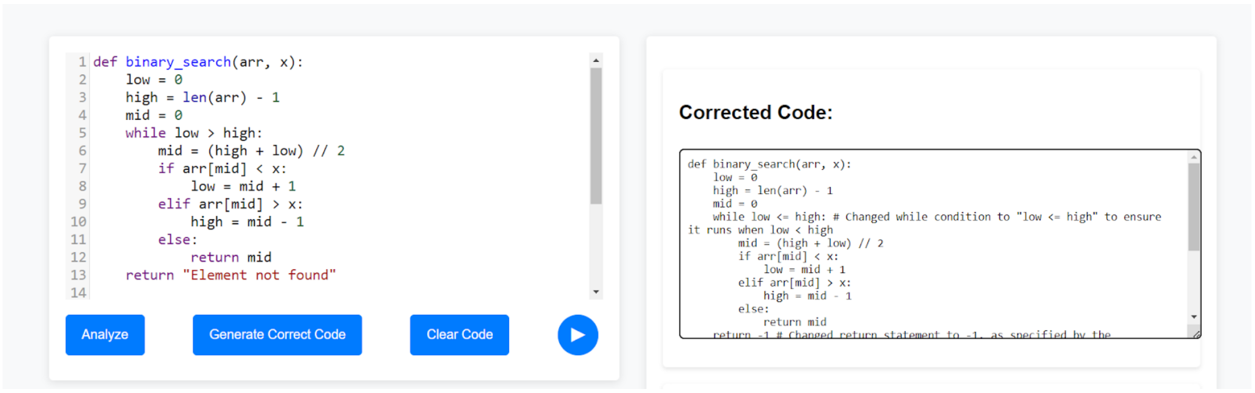
D. Binary Search



```
def binary_search(arr, x):
    low = 0
    high = len(arr) - 1
    mid = 0
    while low > high:
        mid = (high + low) // 2
        if arr[mid] < x:
            low = mid + 1
        elif arr[mid] > x:
            high = mid - 1
        else:
            return mid
    return "Element not found"
```

Suggestions and Corrections:

- There are no syntax errors in this code, but there is a logical error in the while loop. The condition for the while loop is "low > high", but since low is initialized to 0 and high is initialized to len(arr) - 1, this condition will never be true and the while loop will never execute. This means that the binary search will not actually be performed and the function will always return "Element not found". The correct condition for the while loop should be "low <= high".



```
def binary_search(arr, x):
    low = 0
    high = len(arr) - 1
    mid = 0
    while low > high:
        mid = (high + low) // 2
        if arr[mid] < x:
            low = mid + 1
        elif arr[mid] > x:
            high = mid - 1
        else:
            return mid
    return "Element not found"
```

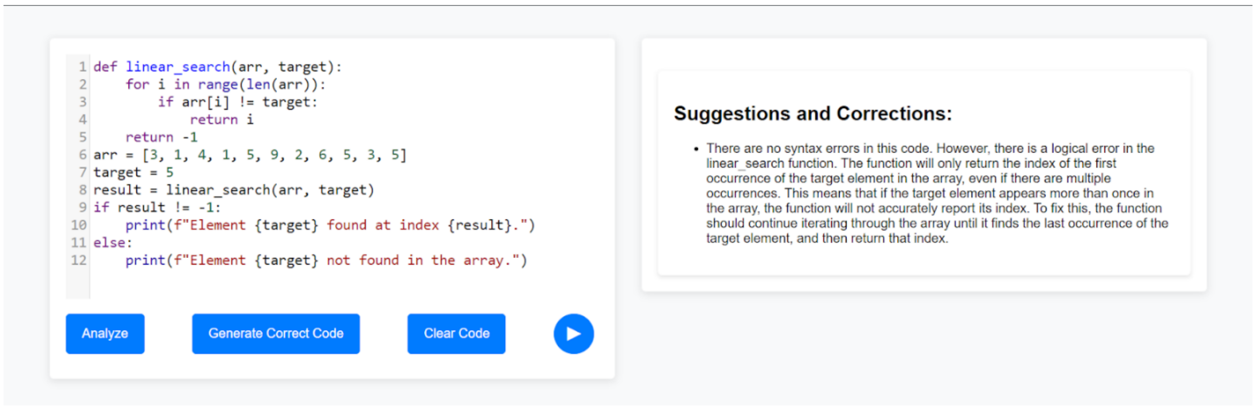
Corrected Code:

```
def binary_search(arr, x):
    low = 0
    high = len(arr) - 1
    mid = 0
    while low <= high: # Changed while condition to "low <= high" to ensure
it runs when low < high
        mid = (high + low) // 2
        if arr[mid] < x:
            low = mid + 1
        elif arr[mid] > x:
            high = mid - 1
        else:
            return mid
    return -1 # Changed return statement to -1, as specified by the
```

Binary Search errors in python can occur due to several reasons of which some include - incorrect Middle Calculation, incorrect Comparison, incorrect Pointer Update, handling Edge Cases, off-by-One Errors, inconsistent Array Indexing.

After identifying the Binary Search errors , the AI code editor corrects the code and also provides more than one way on how the code can be implemented correctly and displays it on the Corrected code column. This helps the programmers learn more ways on how a code can be implemented in various ways which may also include less lines of code and produce the desired or expected output.

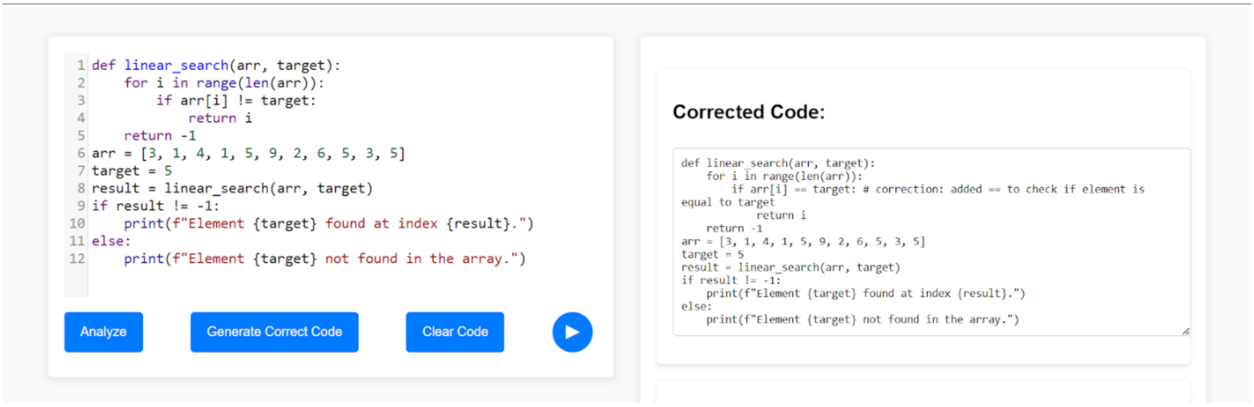
E. Linear Search



```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] != target:
            return i
    return -1
arr = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
target = 5
result = linear_search(arr, target)
if result != -1:
    print(f"Element {target} found at index {result}.")
else:
    print(f"Element {target} not found in the array.")
```

Suggestions and Corrections:

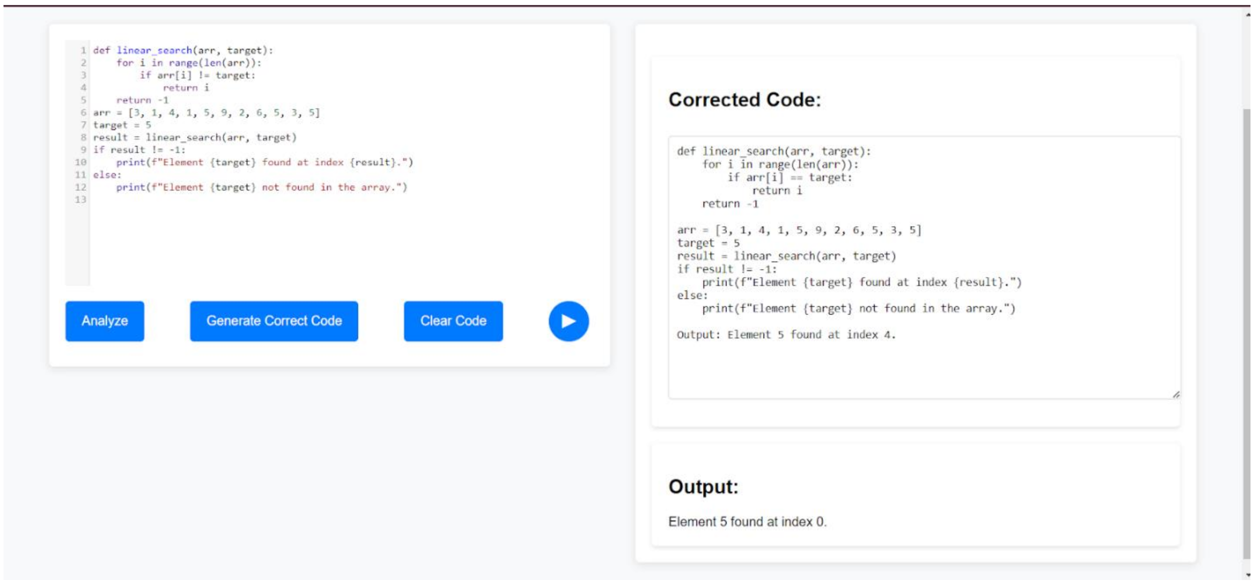
- There are no syntax errors in this code. However, there is a logical error in the linear_search function. The function will only return the index of the first occurrence of the target element in the array, even if there are multiple occurrences. This means that if the target element appears more than once in the array, the function will not accurately report its index. To fix this, the function should continue iterating through the array until it finds the last occurrence of the target element, and then return that index.



```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] != target:
            return i
    return -1
arr = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
target = 5
result = linear_search(arr, target)
if result != -1:
    print(f"Element {target} found at index {result}.")
else:
    print(f"Element {target} not found in the array.")
```

Corrected Code:

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target: # correction: added == to check if element is
equal to target
            return i
    return -1
arr = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
target = 5
result = linear_search(arr, target)
if result != -1:
    print(f"Element {target} found at index {result}.")
else:
    print(f"Element {target} not found in the array.")
```



```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] != target:
            return i
    return -1
arr = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
target = 5
result = linear_search(arr, target)
if result != -1:
    print(f"Element {target} found at index {result}.")
else:
    print(f"Element {target} not found in the array.")
```

Corrected Code:

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1

arr = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
target = 5
result = linear_search(arr, target)
if result != -1:
    print(f"Element {target} found at index {result}.")
else:
    print(f"Element {target} not found in the array.")

Output: Element 5 found at index 4.
```

Output:

```
Element 5 found at index 0.
```

Errors in linear search, a simple searching algorithm, can occur due to various reasons that include Incorrect Loop Logic, Misunderstanding Indexing, Handling Duplicate or Missing Elements, Incorrect Comparison Logic.
The AI code editor identifies these Linear Search errors by carefully reviewing the implementation, ensure correct loop logic and indexing, validate comparison operations, handle edge cases such as duplicates and missing elements, and consider performance implications for large datasets. And hence generate the right code with no Linear Search errors and display the right code to the programmer with the output.

6. Graphical Representation of Results

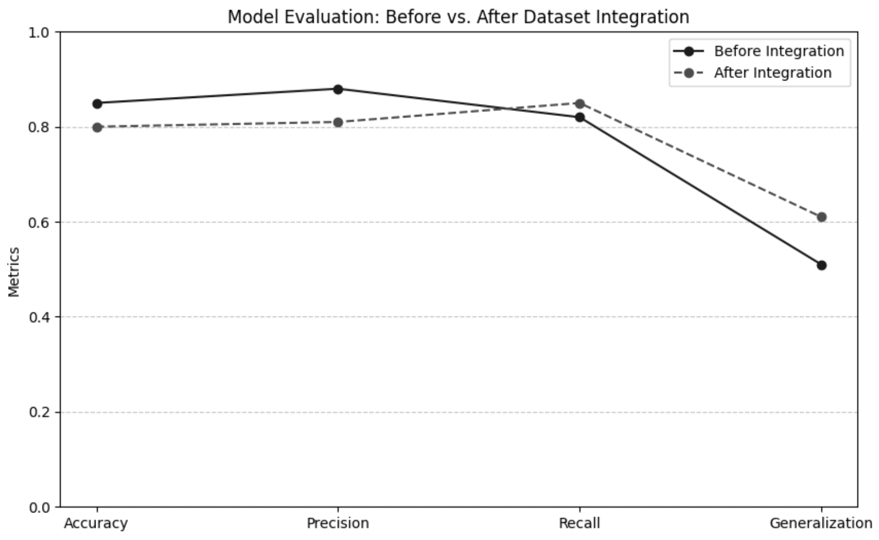


Figure 5 : Model Evaluation Before vs. After Dataset Integration

The graph in *Figure 5* illustrates the comparison of performance metrics before and after dataset integration for the model under evaluation. Four key metrics, namely Accuracy, Precision, Recall, and Generalization, are plotted on the y-axis, while the x-axis represents the different metrics.

Before dataset integration, the model exhibited strong performance across all metrics, with Accuracy at 0.85, Precision at 0.88, Recall at 0.82, and Generalization at 0.51. However, following dataset integration, there was a noticeable change in the model's performance.
While the metrics for Accuracy, Precision, and Recall experienced slight decreases, dropping to 0.80, 0.81, and 0.85, respectively, the metric for Generalization demonstrated a more significant decrease, decreasing to 0.61.

These results suggest that dataset integration had a discernible impact on the model's performance, particularly in terms of its ability to generalize to new data. While the model maintained relatively high levels of Accuracy, Precision, and Recall after integration, it showed a decrease in its ability to generalize, indicating potential challenges in adapting to new or unseen data patterns.

Overall, the findings underscore the importance of careful consideration and evaluation of dataset integration strategies in machine learning models, as they can significantly influence model performance and generalization capabilities.

Limitations

While our methodology and results showcase promising advancements in AI-driven code correction and generation, it's crucial to acknowledge the limitations that shape the boundaries and constraints of our work.

i. Dataset Size: Our project's reliance on a specific dataset may introduce limitations, since our dataset is small and lacks diversity. Limited dataset size can hinder the models' ability to generalize to new scenarios effectively.
ii. Fine-Tuning Constraints: We encountered challenges during the fine-tuning process, including constraints related to computational resources and time. These limitations have impacted the effectiveness of fine-tuning and the results of our model performance.
iii. OpenAI Paid API: Utilizing the OpenAI paid API introduces limitations related to cost and access. While the API provides powerful tools for AI-driven code correction and generation, it requires a subscription fee and may have usage restrictions that impact the scalability and affordability of our solution.
iv. Domain Specificity: Our AI models may exhibit limitations in addressing specific types of errors or generating appropriate code due to the domain specificity of their training data. Models not trained on relevant data may struggle to perform optimally in certain contexts.
v. Evaluation Metrics: The evaluation metrics used in our project may have limitations in capturing the nuances of code correction and generation tasks. These metrics may not fully account for factors such as code readability, efficiency, or adherence to best practices.

By acknowledging these limitations, we provide a more comprehensive understanding of the scope, challenges, and implications of our work. Addressing these limitations can guide future research efforts and contribute to the advancement of AI-driven code correction and generation technologies.

Conclusion

The AI-enhanced Python code editor for novice programmers provides several benefits and improvements to the coding experience. This code editor offers real-time suggestions and corrections for syntax errors, helping novice programmers learn Python syntax more effectively. This feature reduces frustration and accelerates the learning process by providing immediate results and corrections.

The AI code editor helps coders analyse their syntax errors as well as logical errors which help's programmers learn coding in a fast and more effiecinet way, the AI code editor also enhances productivity and encourages good coding practices. This helps programmers improve their overall coding skills.

The editor assists in code refactoring by suggesting improvements to code structure, variable names, and function definitions. This promotes code readability and maintainability, essential skills for professional programming.the AI-enhanced Python code editor empowers novice programmers to write better code, learn more effectively, and gain confidence in their coding abilities. It serves as a valuable tool for both learning Python and developing practical programming skills for future projects.

References

[1] Algaraibeh, Sanaa & Dousay, Tonia & Jeffery, Clinton. (2020). Integrated Learning Development Environment for Learning and Teaching C/C++ Language to Novice Programmers. 1-5. 10.1109/FIE44824.2020.9273887.

[2] Johan Jeuring, Hieke Keuning, Samiha Marwan, Dennis Bouvier, Cruz Izu, Natalie Kiesler, Teemu Lehtinen, Dominic Lohr, Andrew Peterson, and Sami Sarsa. 2022. Towards Giving Timely Formative Feedback and Hints to Novice Programmers. In Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR '22). Association for Computing Machinery, New York, NY, USA, 95–115. https://doi.org/10.1145/3571785.3574124

[3] Alghamdi, Omar & Clinch, Sarah & Alhamadi, Mohammad & Jay, Caroline. (2023). Novice programmers' strategies for online resource use and their impact on source code.

[4] Chrysafiadi, Konstantina & Virvou, Maria & Tsihrintzis, George & Hatzilygeroudis, Ioannis. (2023). An Adaptive Learning Environment for Programming Based on Fuzzy Logic and Machine Learning. International Journal on Artificial Intelligence Tools. 32. 10.1142/S0218213023600114.

[5] Ding, Guozhu & Shi, Xiangyi & Li, Shan. (2023). Integrating programming errors into knowledge graphs for automated assignment of programming tasks. Education and Information Technologies. 1-34. 10.1007/s10639-023-12026-7.

[6] Le, Nguyen-Thinh & Pinkwart, Niels. (2011). INCOM: A web-based homework coaching system for logic programming. IADIS International Conference on Cognition and Exploratory Learning in Digital Age, CELDA 2011. 43-50.

[7] Hu, M. (2004). Teaching Novices Programming with Core Language and Dynamic Visualisation.

[8] Iqbal Malik, Sohail & Mathew, Roy & Hammood, Maytham. (2019). PROBSOL: A Web-Based Application to Develop Problem-Solving Skills in Introductory Programming. 10.1007/978-3-030-01659-3_34.

[9] Jeuring, Johan & Keuning, Hieke & Marwan, Samiha & Bouvier, Dennis & Izu, Cruz & Kiesler, Natalie & Lehtinen, Teemu & Lohr, Dominic & Petersen, Andrew & Sarsa, Sami. (2022). Towards Giving Timely Formative Feedback and Hints to Novice Programmers. 95-115. 10.1145/3571785.3574124.

[10] Bonar, Jeffrey & Soloway, Elliot. (1985). Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. Human-Computer Interaction. 1. 133-161. 10.1207/s15327051hci0102_3.

[11] Venables, Anne & Tan, Grace & Lister, Raymond. (2009). A closer look at tracing, explaining and code writing skills in the novice programmer. ICER'09 - Proceedings of the 2009 ACM Workshop on International Computing Education Research. 117-128. 10.1145/1584322.1584336.

[12] Xie, Benjamin & Loksa, Dastyni & Nelson, Greg & Davidson, Matthew & Dong, Dongsheng & Kwik, Harrison & Tan, Alex & Hwa, Leanne & Li, Min & Ko, Andrew. (2019). A theory of instruction for introductory programming skills. Computer Science Education. 29. 1-49. 10.1080/08993408.2019.1565235.

[13] Wong, A.W., Salimi, A., Chowdhury, S.A., & Hindle, A. (2019). Syntax and Stack Overflow: A Methodology for Extracting a Corpus of Syntax Errors and Fixes. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 318-322.

[14] Meilong, Shi & He, Peng & Xiao, Haitao & Li, Huixin & Zeng, Cheng. (2020). An Approach to Semantic and Structural Features Learning for Software Defect Prediction. Mathematical Problems in Engineering. 2020. 1-13. 10.1155/2020/6038619.

[15] Ajiro, Yasuhiro & Ueda, Kazunori. (2002). Kima: An Automated Error Correction System for Concurrent Logic Programs. Automated Software Engineering. 9. 10.1023/A:1013232219911.

[16] Lachaux, Marie-Anne & Roziere, Baptiste & Chanussot, Lowik & Lample, Guillaume. (2020). Unsupervised Translation of Programming Languages.

[17] N. Jiang, T. Lutellier and L. Tan, "CURE: Code-Aware Neural Machine Translation for Automatic Program Repair," *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, Madrid, ES, 2021, pp. 1161-1173, doi: 10.1109/ICSE43902.2021.00107.

[18] Huq, Faria & Hasan, Masum & Haque, Md. Mahim & Mahbub, Sazan & Iqbal, Anindya & Ahmed, Toufique. (2021). Review4Repair: Code review aided automatic program repairing. Information and Software Technology. 143. 106765. 10.1016/j.infsof.2021.106765.

[19] Aung, Soe & Funabiki, Nobuo & Syaifudin, Yan & Kuribayashi, Minoru. (2020). A Study of Grammar-Concept Understanding Problem for Java Programming Learning Assistant System.

[20] Zhou, Zihe & Wang, Shijuan & Qian, Yizhou. (2021). Learning From Errors: Exploring the Effectiveness of Enhanced Error Messages in Learning to Program. Frontiers in Psychology. 12. 10.3389/fpsyg.2021.768962.

[21] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: fixing common C language errors by deep learning. In Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI'17). AAAI Press, 1345–1351.

[22] Yasunaga, M., & Liang, P. (2021). Break-It-Fix-It: Unsupervised Learning for Program Repair. ArXiv, abs/2106.06600.