PROJECT REPORT ON

PyPAF:Python Program,Algorithm and Flowchart Generator

Submitted By

FATHIMA NOURIN S U (CEC21CS045)

HARITHA KRISHNA R (CEC21CS050)

HIBA HUSSAIN (CEC21CS051)

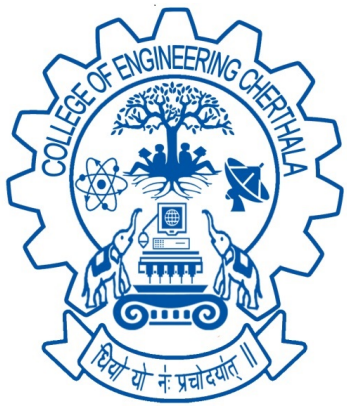
SURYA ANN JOSHY (CEC21CS095)

under the esteemed guidance of

Mrs.Fathima N

Assistant Professor

Department Of Computer Engineering



APRIL 2025

DEPARTMENT OF COMPUTER ENGINEERING
COLLEGE OF ENGINEERING, PALLIPPURAM P O, CHERTHALA,
ALAPPUZHA PIN: 688541,
PHONE: 0478 2553416, FAX: 0478 2552714
http://www.cectl.ac.in

PROJECT REPORT ON

PyPAF:Python Program,Algorithm and Flowchart Generator

Submitted By

FATHIMA NOURIN S U (CEC21CS045)

HARITHA KRISHNA R (CEC21CS050)

HIBA HUSSAIN (CEC21CS051)

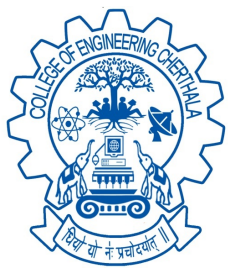
SURYA ANN JOSHY (CEC21CS095)

under the esteemed guidance of

Mrs. Fathima N

(Assistant Professor)

In partial fulfillment of the requirements for the award of the degree
of
Bachelor of Technology
in
Computer Science and Engineering
of
APJ Abdul Kalam Technological University



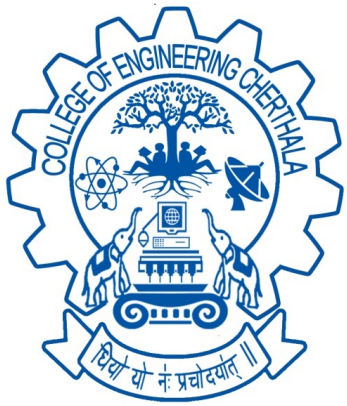
APRIL 2025

DEPARTMENT OF COMPUTER ENGINEERING
COLLEGE OF ENGINEERING, PALLIPPURAM P O, CHERTHALA,
ALAPPUZHA PIN: 688541,
PHONE: 0478 2553416, FAX: 0478 2552714
http://www.cectl.ac.in

DEPARTMENT OF COMPUTER ENGINEERING

COLLEGE OF ENGINEERING CHERTHALA

ALAPPUZHA-688541



C E R T I F I C A T E

This is to certify that the project report titled **PyPAF: Python Program, Algorithm and Flowchart Generator** is a bonafide record of the **CSD415 Project** presented by **FATHIMA NOURIN S U** (CEC21CS045), **HARITHA KRISHNA R** (CEC21CS050), **HIBA HUSSAIN** (CEC21CS051), and **SURYA ANN JOSHY** (CEC21CS095), Eight Semester B.Tech Computer Science & Engineering students, under our guidance and supervision, in partial fulfillment of the requirements for the award of the degree, **B. Tech. Computer Science & Engineering** of **A P J Abdul Kalam Technological University**.

Guide	**Co-ordinator**	**HoD**
Mrs. Fathima N	**Mr.Jayakrishnan R**	**Dr. Preetha Theresa Joy**
Assistant Professor	Assistant Professor	Professor
Dept Of Computer Engg	Dept Of Computer Engg	Dept Of Computer Engg

ACKNOWLEDGEMENT

This work would not have been possible without the support of many people. First and the foremost, we give thanks to the Almighty God who gave us the inner strength, resource and ability to complete our project successfully.

We would like to thank **Dr. Jaya V L**, our Principal, who has provided with the best facilities and atmosphere for the project completion and presentation.We would also like to thank our HoD **Dr. Preetha Theresa Joy** (Professor, Department Of Computer Engineering) , our project coordinator **Mr.Jayakrishnan R** (Assistant Professor, Department Of Computer Engineering), and our guide **Mrs. Fathima N** (Assistant Professor, Department Of Computer Engineering) for the help extended and also for the encouragement and support given to us while doing the project.

We would like to thank our dear friends for extending their cooperation and encouragement throughout the project work, without which we would never have completed the project this well. Thank you all for your love and also for being very understanding.

ABSTRACT

In the evolving landscape of programming education, automation tools play a crucial role in enhancing the teaching and learning experience. This project presents an intelligent code generation and evaluation system that transforms user input into structured pseudocode, which serves as the foundation for generating Python code, algorithms, and flowcharts. The system automates code creation while ensuring adherence to coding standards and algorithmic correctness. Additionally, it includes a code evaluation feature designed for educators, enabling them to assess student submitted code by comparing it with the system generated code. This comparison provides valuable insights into code accuracy and efficiency, assisting educators in evaluating student's programming skills. By integrating automation with structured evaluation methodologies,the project serves as a powerful tool for improving programming education and assessment.

Contents

List of Figures

Chapter 1

INTRODUCTION

In the evolving landscape of digital education, the ability to transform ideas into functional code, structured algorithms, and visual representations plays a crucial role in programming and computational thinking. However, individuals with limited programming expertise often struggle to convert their concepts into executable solutions. This challenge affects both students, who require guided learning tools and educators, who face difficulties in assessing programming assignments efficiently.To bridge this gap, this project introduces an automation-driven web application that facilitates programming education by generating pseudocode, structured algorithms, flowcharts, and Python code from user-provided queries. The system enables students to input logic in simple language, which is then parsed and converted into meaningful outputs, aiding in their understanding of programming concepts. Additionally, an automated grading system is integrated to evaluate code correctness and similarity, allowing educators to assess student submissions with greater accuracy and efficiency.

The application enhances interactive learning by automating pseudocode to algorithm conversion, flowchart generation,code generation and also code assessment, reducing the manual effort required in programming education. Furthermore, by simplifying complex computational processes, the system ensures that both beginners and experienced users can engage with programming concepts more effectively.By leveraging automation and AI-driven techniques, the project aims to improve accessibility, streamline programming assessments, and foster a more inclusive and engaging approach to programming education.

Chapter 2

PROBLEM STATEMENT

2.1 PROBLEM STATEMENT

The process of learning programming presents significant challenges for students, particularly in understanding fundamental concepts and translating them into practical solutions. Many learners struggle to grasp essential programming constructs such as loops, conditionals, and data structures, often finding it difficult to structure logical solutions to problems. This difficulty is compounded when they attempt to convert real-world problems or natural language queries into structured algorithms, visualize them as flowcharts, and ultimately implement them as functional Python code. The lack of intuitive tools to bridge these gaps frequently leads to errors, confusion, and a steep learning curve. As a result, students experience frustration, disengagement, and slower progress in developing the skills necessary to thrive in a technology-driven world.

For educators, the challenges are equally pressing, especially when it comes to evaluating student performance. During lab examinations or assignments, teachers are tasked with assessing a large volume of code submissions within tight time constraints, making manual grading an inefficient and daunting process. This approach is not only time-consuming but also prone to inconsistencies, as it relies heavily on subjective judgment and limited resources to thoroughly review each piece of code. Consequently, providing timely, accurate, and constructive feedback becomes a significant hurdle, hindering students' ability to learn from their mistakes and improve. There is a clear need for an automated system that alleviates these burdens by simplifying the

learning process for students and streamlining the assessment process for educators, ensuring both efficiency and effectiveness in programming education.

2.2 OBJECTIVE

The project focuses on a web-based automation tool that empowers students to overcome the challenges of learning programming by seamlessly converting natural language queries into structured programming artifacts. Specifically, the tool aims to guide learners through the educational sequence of algorithm design, flowchart visualization, and Python code generation, fostering a deeper understanding of fundamental programming concepts. By leveraging advanced technologies such as a Huggingface model for code generation and visualization libraries like Graphviz for flowchart creation, the system seeks to simplify the process of translating ideas into executable code. This will reduce the frustration and disengagement often experienced by students, enabling them to build problem-solving skills and logical reasoning more effectively and efficiently.

Additionally, the project aims to support educators by integrating an automated grading system that alleviates the burden of evaluating large volumes of student submissions. The tool will incorporate code similarity detection techniques, such as Abstract Syntax Tree (AST) comparison, to assess the correctness and quality of students' code against generated solutions, providing accurate and consistent feedback.

By offering a built-in environment for code execution, testing, and debugging, the system ensures that learners can validate their work independently, while teachers benefit from a streamlined assessment process. Ultimately, the objective is to create an intuitive, comprehensive platform that enhances programming education for both students and educators, bridging the gap between conceptual learning and practical application.

Chapter 3

LITERATURE REVIEW

Published in September 2018 in the International Journal of Research - Granthaalayah **"An Extensible Approach to Generate Flowcharts from Source Code"** by Damitha D. Karunarathna and Nasik Shafeek of the University of Colombo School of Computing introduces a novel method to assist developers in visualizing program logic as flowcharts, enhancing code comprehension and maintenance. The researchers propose a three-stage compiler-based process that employs an Abstract Syntax Tree (AST) as an intermediate representation, transforming source code—tested with PHP—into flowcharts using a custom parser, Dot Language, and the open-source Graphviz tool. This approach abstracts language-specific syntax into a universal AST, allowing extensibility to other languages through new front-end translators, while its modular structure separates parsing, translation, and rendering for independent updates, utilizing existing tools to minimize development effort. However, the prototype remains confined to PHP, demands compiler expertise for expansion, lacks a user interface for novices, and generates unoptimized Dot Language output, potentially resulting in cluttered flowcharts. Experiments with constructs such as if-else, loops, and switches validated its feasibility, yielding accurate and compliant flowcharts. The authors suggest future improvements, including support for languages like Java and Python, development of a user-friendly interface, and optimization of Dot Language for clearer diagrams, positioning their method as a promising, scalable solution despite current limitations [1].

Wu Wen et al.'s paper,**"Code Similarity Detection using AST and Textual Information"** published in the International Journal of Performability Engineering in October 2019, investigates

code plagiarism in programming education and proposes a hybrid algorithm to detect similarities across student submissions. The authors target prevalent plagiarism methods—such as exact copying, comment changes, identifier renaming, code reordering, and refactoring—by employing a dual approach: they utilize the Simhash algorithm for textual analysis, creating fingerprints for word comparisons, and Abstract Syntax Tree (AST) analysis with the Zhang-Shasha algorithm for structural edit distance, combining these into a weighted similarity score. Their experiments with Python code from MoocCode and LeetCode confirmed the method's efficacy, yielding high similarity scores (e.g., above 0.8) for plagiarized work and lower scores (e.g., below 0.6) for originals, thus supporting fairer educational assessments. The approach excels in detecting varied plagiarism tactics and leverages real-world student code, yet it faces challenges with computational inefficiency, absence of functional checks, and lack of ready-to-use implementation. The authors plan future enhancements to improve its efficiency for larger-scale applications, aiming to amplify its educational value [2].

The paper **"Automatic Code Generation for C and C++ Programming"** (IRJET, May 2021) details an initiative by authors Sanika Patade, Pratiksha Patil, Ashwini Kamble, and Prof. Madhuri Patil to simplify programming through a desktop application that transforms user-drawn flowcharts into C and C++ code. Built with Eclipse and JDK1.5.0, their GUI-based tool fits into a wider landscape of code generation research focused on easing coding efforts. It demonstrates advantages such as saving time, minimizing syntax errors, and supporting beginners via a visual interface, with positive feedback from both experts and general users affirming its utility. Nonetheless, the tool's dependence on predefined flowchart shapes, restriction to C/C++ languages, and outdated technology limit its ability to address complex, nested structures, reflecting drawbacks seen in similar studies. While the paper provides a practical, education-oriented contribution, the field is advancing toward AI-driven solutions—like prompt-based pseudocode and multi-language code generation—suggesting a future need for more flexible, intelligent, and updated systems to tackle these constraints [3].

The study **"On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot"** (2023) by Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo

Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota investigates the robustness of GitHub Copilot for code generation. The researchers generated 892 Java methods using Copilot, testing it with both original and paraphrased descriptions. They employed two paraphrasing techniques—PEGASUS, a deep learning-based method, and Translation Pivoting (TP)—to examine how variations in input descriptions impact the quality and accuracy of Copilot's code outputs.The study highlights that input description quality significantly affects code generation results, offering valuable insights for improving code recommendation systems. However, the variability in Copilot's output can hinder usability, as different phrasings of the same intent may lead to inconsistent or incorrect recommendations. Future work will involve conducting in vivo studies with real developers to better understand their experiences and refining automated paraphrasing for software-specific text to enhance reliability and accuracy in code generation tools [4] .

The paper **"Leveraging Pre-Trained Language Models for Code Generation"** (2024) by Ahmed Soliman, Samir Shaheen, and Mayada Hadhoud explores the use of pre-trained language models like BERT, RoBERTa, ELECTRA, and LUKE for generating code. Using Marian as a decoder, the researchers fine-tuned these models on two datasets—CoNaLa and DJANGO—after preprocessing the data through tokenization and normalization steps. The setup relies on Python, the PyTorch framework, and Hugging Face tools, with a focus on enhancing code generation quality, as measured by BLEU scores and exact match rates.The advantages of this approach include improved code quality, faster coding processes, and better contextual understanding within generated code. However, the study is limited by the small dataset size, impacting generalizability, and by a focus on single-line code generation rather than complex structures. Significant computational resources are also required, and only a subset of possible models was tested, which leaves room for future improvements. Future work includes exploring multimodal code generation with visual or audio inputs and integrating explainable AI features, like attention mechanisms, for greater interpretability [5].

Aryaman Darda and Reetu Jain's study, **"Code Generation from Flowchart using Optical Character Recognition & Large Language Model,"** April 24, 2024, introduces a tool designed to aid novice programmers by converting flowchart images into Python code. They developed an

intuitive user interface with the Python library Gradio, allowing users to upload flowchart images and receive corresponding Python code accompanied by a brief explanation. The process entails extracting text from the images using EasyOCR, which leverages deep learning models like VGG and ResNet for feature extraction, Long Short-Term Memory (LSTM) networks for sequence labeling, and Connectionist Temporal Classification (CTC) algorithms for decoding. The extracted text is then transformed into a structured query and provided to the 70-billion-parameter Llama-2-Chat large language model, accessed through the Replicate API, to generate the code. Tested with a diverse set of flowchart images, the application demonstrated its ability to produce executable Python code, proving effective even for challenging cases involving noisy or complex flowcharts, thus offering a valuable educational resource to simplify the shift from visual algorithm design to functional code for beginners [6].

Chapter 4

SOFTWARE REQUIREMENT SPECIFICATION (SRS)

4.1 Introduction

4.1.1 Purpose

The purpose of this project is to enhance programming education by developing an AI-powered system that automates code generation and grading. The system aims to assist students in understanding programming concepts by converting problem statements into structured pseudocode, algorithms, flowcharts, and executable Python code. Additionally, it provides an automated grading mechanism for educators, enabling efficient and accurate assessment of student submissions during lab exams. By leveraging artificial intelligence, the platform streamlines the learning process, reduces manual evaluation efforts, and ensures fairness in assessment.

4.1.2 Project Scope

This web-based automation tool assists students in understanding programming concepts, generating structured algorithms, and visualizing flowcharts. For educators, it provides an efficient evaluation system that automatically assesses student submissions using similarity detection techniques. The system streamlines the learning process by maintaining a structured approach: Algorithm−>Flowchart −> Code

4.1.3 References

- HuggingFace AI models for code generation. [7]
- AST Similarity algorithms for code evaluation. [2]

4.2 Overall Description

4.2.1 Product Perspective

The AI-powered Code Generation and Grading System is a standalone web-based application designed to assist students and educators in programming education. It integrates AI-driven code generation, algorithm ,flowchart visualization, and an automated assessment system to streamline the learning and evaluation process. Unlike traditional Integrated Development Environments (IDEs), this system provides an end-to-end solution by translating problem statements into structured pseudocode, algorithms,flowcharts and Python code while also offering automated grading. The system leverages Natural Language Processing (NLP) models for prompt interpretation, similarity detection techniques for evaluation, and AI-based automation to minimize manual effort in both learning and assessment. By integrating these features into a single platform, the system enhances programming comprehension, optimizes evaluation efficiency, and ensures fairness in grading.The platform is divided into two main modules:

4.2.1.1 Student Module

- Students input a problem statement.
- The system generates: Pseudocode,Algorithm,Flowchart,Python code
- Students can execute and verify the generated code.

4.2.1.2 Teacher Module

- The system generates a reference solution using code generation model.
- Student code is compared with the reference solution using similarity-checking algorithms.
- Automatic grading assigns grades with an option for manual review.

4.2.2 Product Functions

The system comprises the following core functionalities:

- **Prompt-Based Input:** Users provide logic in natural language, which is converted into pseudocode.
- **Algorithm & Flowchart Generation:** The system translates pseudocode into structured algorithms and visual flowcharts.
- **Code Generation:** AI models generate Python code from pseudocode.
- **Code Execution & Debugging:** Users can test and validate their code within an integrated execution environment.
- **Automated Grading:** The system evaluates student submissions using similarity detection and assigns scores based on correctness.

4.2.3 User Characteristics

Targeted users include:

- **Students:** Users who require assistance in learning programming logic, generating algorithms, and converting them into executable code.
- **Teachers:** Educators who need an automated grading system for evaluating programming assignments and lab exams.

4.2.4 Constraints

The system must address:

- Requires an internet connection for accessing AI models and executing pseudocode generations.
- The correctness of generated Python code relies on the precision of pseudocode translation.

- Limited to Python programming language in the initial implementation phase.
- Ensures clarity and correctness in automatically generated flowcharts.
- The accuracy of student code assessment depends on predefined comparison metrics and evaluation criteria.

4.2.5 Assumptions and Dependencies

- Assumes the availability of a reliable pseudocode generation mechanism.
- Depends on accurate translation of pseudocode into Python code, algorithms, and flowcharts.
- Requires tools like Graphviz for generating flowcharts.
- Assumes predefined comparison metrics for assessing student-submitted code.

4.3 Specific Requirements

4.3.1 Functional Requirements

- **FR1**: Accept user input and generate structured pseudocode.
- **FR2**: Generate a step-by-step algorithm from the pseudocode.
- **FR3**: Create a flowchart representation of the pseudocode.
- **FR4**: Convert pseudocode into Python code automatically.
- **FR5**: Allow educators to upload student code for evaluation.
- **FR6**: Compare student-submitted code with system-generated Python code and provide similarity analysis.
- **FR7**: Display generated outputs (Python code, algorithm, flowchart, and evaluation results) in an interactive interface.
- **FR8**: Provide options to copy,run and download the generated outputs.

4.3.2 Non-Functional Requirements

- **NFR1**: Ensure high accuracy in pseudocode generation and code conversion.
- **NFR2**: The system should generate Python code within 2 minutes of receiving input.
- **NFR3**: The UI should be intuitive and user-friendly.
- **NFR4**: Ensure reliable and efficient code evaluation for educators.
- **NFR5**: The system should provide clear explanations for generated algorithms and flowcharts.

4.4 System Features

4.4.1 Input Processing Module

Allows users (students or teachers) to interact with the system through a web-based interface.

- Accepts problem descriptions or programming tasks in natural language.
- Normalizes and tokenizes the input to ensure clarity before processing.

4.4.2 Pseudocode Generation Module

Converts natural language descriptions into structured pseudocode.

- Uses the BlackBox API for pseudocode generation.
- Stores generated pseudocode for further processing.
- Ensures logical structuring of pseudocode to serve as a foundation for algorithm and code generation.

4.4.3 Algorithm Generation Module

Transforms pseudocode into a step-by-step structured algorithm.

- Extracts algorithmic steps from the generated pseudocode.
- Provides a clear, human-readable format for better understanding.

4.4.4 Flowchart Visualization Module

Generates a graphical representation of the pseudocode.

- Uses Graphviz to create interactive flowcharts.
- Visualizes the logical flow of the program for better comprehension.

4.4.5 Code Generation Module

Converts pseudocode into executable Python code.

- Utilizes the Hugging Face model for code generation.
- Ensures the generated Python code adheres to programming standards.
- Provides options to copy and run the generated code within the platform.

4.4.6 Code Evaluation Module

Assesses student-submitted code by comparing it with the system-generated reference code.

- Teachers input problem statements, and the system generates Python code as a reference solution.
- Student code is compared with the reference code using similarity metrics like AST Similarity and Levenshtein Distance.
- Automatically assigns marks based on the similarity score and test pass ratio, providing objective assessment.

4.4.7 User Interface Module

Provides an interactive and user-friendly interface.

- Allows users to input problem descriptions and receive outputs.
- Displays pseudocode, algorithm, flowchart, Python code, and evaluation results.
- Provides options for copying, downloading, and running the generated outputs.

4.5 Other Non-functional Requirements

4.5.1 Performance

The system should generate pseudocode, algorithms, flowcharts, and Python code within seconds for short to medium-length inputs.

4.5.2 Reliability

The system must consistently produce accurate outputs without crashes, ensuring dependable operation for educators and students.

4.5.3 Usability

The user interface should be intuitive and accessible for users with varying levels of programming expertise, including teachers and students.

4.5.4 Scalability

The architecture should support multiple users simultaneously, ensuring smooth operation even with increased usage.

4.5.5 Maintainability

The system should allow for easy updates, particularly for improving pseudocode generation, code evaluation models, and user interface enhancements.

4.6 Feasibility Study

This feasibility study evaluates the potential and practicality of developing a system that automates the generation of pseudocode, algorithms, flowcharts, and Python code based on user descriptions. It also examines the feasibility of an integrated code evaluation system for educators. The study addresses economic, technical, operational, and social feasibility.

4.6.1 Economic Feasibility

This assesses the financial viability of the project, considering investments in software tools, cloud resources, and development time. While initial costs may be high due to computational requirements, the system reduces manual effort in programming education, making it a cost-effective solution in the long term. The study concludes that the project is economically feasible.

4.6.2 Technical Feasibility

This examines the availability and reliability of the required technology. The project leverages Python, Flask for web development, and libraries such as Graphviz for flowchart generation. With access to automation tools and computational resources, the system is technically feasible.

4.6.3 Operational Feasibility

This evaluates whether the system can function effectively in an educational environment. The system features a user-friendly interface, structured code generation, and automated evaluation for student submissions. With efficient workflow design, the project is operationally feasible and beneficial for both students and educators.

4.6.4 Social Feasibility

This examines user acceptance and societal impact. The system aids teachers in explaining programming concepts and assessing student code efficiently. By reducing manual effort in grading and improving learning outcomes, it enhances programming education. The study concludes that the project is socially feasible and valuable for academic use.

Chapter 5

SYSTEM DESIGN

Design is the process of defining the architecture, modules, interfaces, and data for a system to satisfy specified requirements. Here we have many modules that perform specific functionalities.The design of the system includes mainly two steps:

- System design
- Detailed design

In system design a structural framework for the entire system is created.It is done in such a way that related part come under particular groups.Thus after the system design,a network of different groups is obtained. In detailed design,each group is studied in detail and the internal operations are decided.Based on this,the data structures and programming languages to be used are decided.

5.1 MODULES

5.1.1 User Selection

The user selection module determines whether the system is being accessed by a student or a teacher. This classification is crucial as it defines the workflow and functionalities available to the user. If a student accesses the system, they are guided through the pseudocode generation, algorithm conversion, and flowchart creation processes. Teachers, on the other hand, gain access

to evaluation tools that allow them to review and assess student submissions. This module ensures that each user type experiences a tailored interface optimized for their specific needs.

5.1.2 Pseudocode Generation

The Pseudocode Generation Module is responsible for converting natural language descriptions into pseudocode using the BlackBox API. The system sends the user input to the API, retrieves the generated pseudocode, and stores it for further processing. This module ensures that user inputs are transformed into structured programming logic, serving as the foundation for algorithm and code generation. The implementation is handled using Python and the Requests library for seamless API communication.

5.1.3 Algorithm Generation

The Algorithm Generation Module takes the generated pseudocode and converts it into a structured step-by-step algorithm. The objective of this module is to create a clear and human-readable format, allowing users to understand the logical flow of the solution before proceeding to code implementation. A rule-based mapping approach is employed, where predefined syntax rules guide the structuring of the final algorithm. This module is implemented using Python with custom transformation logic to extract structured algorithmic steps from the pseudocode.

5.1.4 Flowchart Visualization

This module extends the capabilities of the system by converting pseudocode into a graphical representation of logic flow. After parsing the pseudocode, it classifies logical statements into nodes such as decision points, input/output steps, and processing steps. A flowchart construction layer arranges these classified nodes based on control flow logic, creating a structured visual representation. This module enhances the learning experience by enabling students to comprehend the flow of execution in an intuitive manner.

5.1.5 Code Generation

The Code Generation Module is responsible for converting pseudocode into executable Python code. The generated pseudocode is passed to the Hugging Face model, which translates it into syntactically correct and functional Python code while ensuring adherence to programming standards. The generated code is then presented to the user for execution and verification. To enhance usability, the interface includes a Copy button, allowing users to quickly copy the generated code for external use, and a Run button, enabling them to execute the code directly within the platform. This functionality provides immediate feedback on the correctness and functionality of the generated code, ensuring a seamless and efficient coding experience.

5.1.6 Code Evaluation

The Code Evaluation Module is designed to assess student-submitted Python programs by comparing them with a reference solution. A reference code is first generated using a Hugging Face model based on a given problem statement. The submitted code is then evaluated for similarity using Levenshtein Distance and SequenceMatcher, ensuring structural and functional accuracy. The module also utilizes Abstract Syntax Tree (AST) parsing to analyze the code's structure and includes a timeout mechanism to prevent infinite loops or excessive execution time. The final evaluation considers functional correctness, structural similarity, and efficiency, providing an automated and objective assessment.

5.2 SYSTEM ARCHITECTURE

The overall architecture of the system revolves around two types of users: Students and Teachers, each interacting with the system for different purposes.The process begins with User Selection, where the user identifies themselves as either a student or a teacher.

For Students, the system takes a User Prompt, which serves as an input query for pseudocode generation. The Pseudocode Generation module is responsible for translating the prompt into structured pseudocode. This pseudocode undergoes a Parsing stage, which extracts key com-

ponents and generates an Algorithm and a Flowchart for better visualization. Additionally, the Code Generation Model processes the pseudocode to generate corresponding executable code.

For Teachers, the process is slightly different. Teachers provide a User Prompt, which is directly processed by the Code Generation Model to generate Reference Code. This reference code is then compared with student-submitted code in the Evaluation phase. The evaluation process assesses the correctness and quality of the student's code against the reference code. Finally, the system generates a Report, providing feedback and performance analysis of the student's submission.

This architecture ensures that students receive structured outputs like algorithms, flowcharts, and code, while teachers can assess student work efficiently through automated evaluation and reporting.

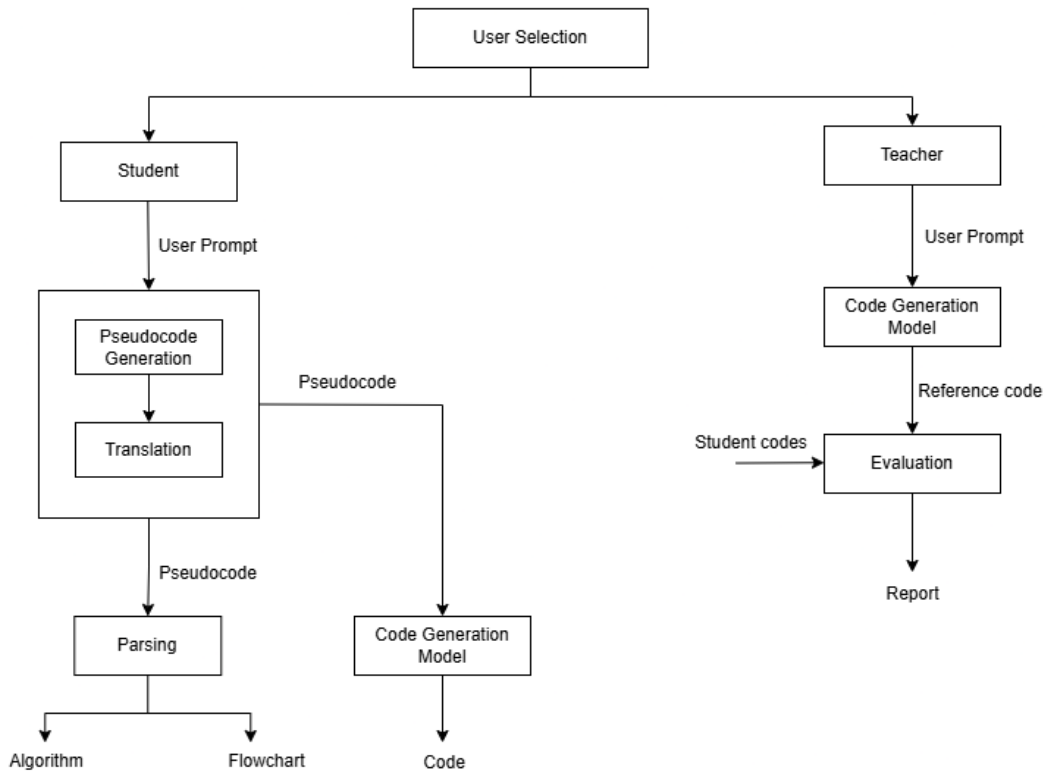


Fig. 5.1: System Architecture

5.2.1 Code Generation Architecture

The proposed model is a Transformer-based decoder architecture designed to convert pseudocode into structured code. It follows a step-by-step approach, processing input pseudocode through tokenization, embedding, sequential decoding, and output generation. This architecture leverages self-attention mechanisms and deep learning principles to ensure accurate and efficient code generation.

5.2.1.1 Tokenization and Embedding Layer

Before processing, the input pseudocode is tokenized into smaller components such as keywords, variables, and symbols. These tokens are then mapped to numerical representations using a predefined vocabulary that contains commonly used programming terms like "begin", "end", "OUTPUT", and "if".Once tokenized, the tokens pass through a Token Embedding Layer, where each token is converted into a dense vector representation. This transformation helps the model understand semantic relationships between tokens. Additionally, a Position Embedding Layer is applied to retain the sequence order of tokens, ensuring that the model correctly interprets the structure of the input.

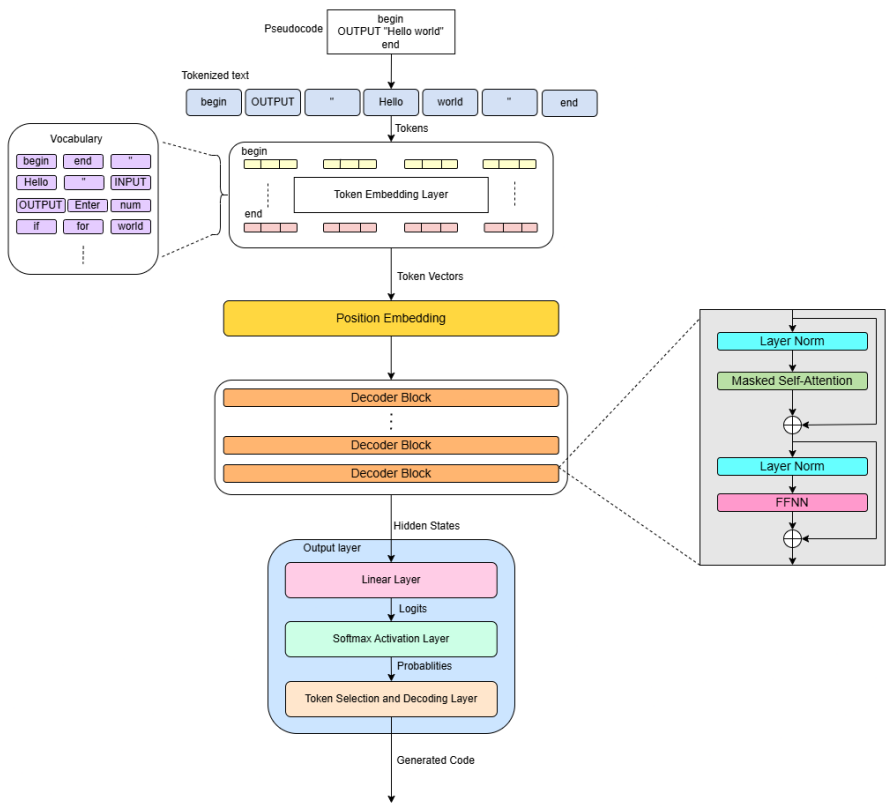


Fig. 5.2: Code Generation Architecture

5.2.1.2 Decoder Blocks and Processing

The decoder consists of multiple stacked decoder blocks, each responsible for processing token embeddings and generating meaningful representations. Each decoder block contains three primary components:

Masked Self-Attention Mechanism: This mechanism ensures that each token can only attend to previously generated tokens, maintaining an auto-regressive approach to decoding. By masking future tokens, the model prevents cheating by ensuring predictions are based only on past information. The attention scores are computed using the Scaled Dot-Product Attention, where queries, keys, and values are projected using learned weight matrices. These attention scores determine the importance of each past token when predicting the next one. The softmax function is applied to normalize the scores before weighted summation.

Layer Normalization: To stabilize training and improve convergence, layer normalization is applied at multiple points in the decoder block—both before and after the self-attention mechanism. This normalization helps prevent internal covariate shift, ensuring that the activations remain within a stable range. By normalizing across the feature dimension, the model maintains consistent scaling of representations, which is particularly useful in deep architectures.

Feed-Forward Neural Network (FFNN): This component further refines token representations by applying non-linearity. The FFNN consists of two linear layers separated by an activation function, typically a ReLU (Rectified Linear Unit). The first layer expands the input dimensions, capturing complex transformations, while the second layer projects it back to the original size. This non-linearity enables the model to learn richer representations beyond what attention alone can capture.

Each decoder block incorporates residual connections, which help in maintaining the flow of gradients across layers. These connections ensure that important information is retained while reducing the risk of vanishing gradients, making training more efficient

5.2.1.3 Output Processing Layer

Once the hidden states pass through the decoder blocks, they undergo a final transformation in the Output Processing Layer, which converts the processed token representations into meaningful output code. This stage ensures that the generated code is syntactically and semantically correct. The main components of this layer include:

a. Linear Layer: The hidden states produced by the decoder blocks are first passed through a Linear Transformation Layer. This layer maps the hidden state vectors to a higher-dimensional space, where each dimension corresponds to a token in the predefined vocabulary. Essentially, this transformation assigns a numerical score (logit) to each possible token, indicating how relevant that token is for the current position in the output sequence.

b. Softmax Activation Layer: Since the output from the linear layer consists of raw scores, they need to be converted into probabilities to determine the likelihood of each token appearing in the final output. This is achieved through a Softmax Activation Function, which normalizes the logits into probability values.

c. Token Selection and Decoding Layer: After obtaining the probability distribution over all tokens, the Token Selection and Decoding Layer determines the final output by selecting tokens sequentially.The decoding process continues iteratively until a special termination token (e.g., "end") is generated, signaling the completion of the output sequence. This ensures that the generated code has a well-defined stopping point, preventing infinite loops or unnecessary token generation.

By integrating these components, the Output Processing Layer effectively translates the learned representations from the decoder into well-structured, syntactically correct, and semantically meaningful code.

5.2.2 Algorithm Archietecture

This architecture transforms pseudocode into a structured algorithm by following a sequence of well-defined steps. Each step ensures that the logic from the pseudocode is correctly interpreted and converted into a formal algorithmic structure.

5.2.2.1 Pseudocode Input

The system starts by receiving pseudocode as input, which contains step-by-step instructions written in a structured but informal manner. The pseudocode follows a logical sequence but lacks strict syntax rules required for direct execution.

5.2.2.2 Preprocessing

Before further processing, the pseudocode undergoes a cleaning phase where unnecessary elements like extra whitespace, special symbols, and comments are removed. Then, it is tokenized, breaking it down into fundamental units such as keywords, variables, operators, and control structures. This step standardizes the input, making it easier for further analysis.

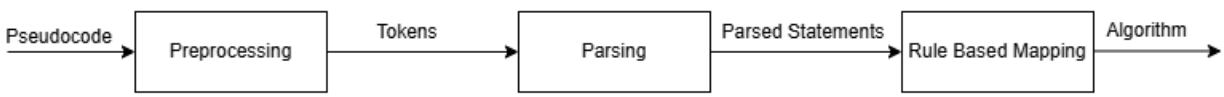


Fig. 5.3: Algorithm Generation Architecture

5.2.2.3 Parsing

The tokenized pseudocode is analyzed to construct a structured representation. This involves identifying loops (for, while), conditions (if-else), function calls, and assignments. A syntax tree or an intermediate representation is created, which organizes these components in a hierarchical manner to preserve execution order and relationships between statements.

5.2.2.4 Rule-Based Mapping

Once parsed, the system applies predefined transformation rules to convert the structured representation into an algorithmic format. Each conditional statement, loop, and function call is mapped to its corresponding representation in the algorithm. The mapping ensures logical accuracy while maintaining clarity.

5.2.2.5 Algorithm Output

Finally, the system generates a structured algorithm that follows a clear, stepwise format. The output is a well-organized algorithm that maintains the intent of the original pseudocode.

5.2.3 Flowchart Generation Architecture

This part of the architecture follows the same initial steps as the Algorithm Generation Architecture, including pseudocode input, preprocessing, and parsing. After parsing, the system maps the extracted logical statements into classified nodes with labels, such as decision points, input/output steps, and processing steps. These classified nodes are then processed in the Flowchart Construction Layer, where they are arranged and connected based on control flow logic. Finally, the system generates the flowchart output, providing a structured visual representation of the pseudocode's logic.

Fig. 5.4: Flowchart Generation Architecture

5.3 USE-CASE DIAGRAM

Use case diagrams model the functionality of a system using actors and use cases. Use cases are a set of actions, services, and functions that the system needs to perform.

The below mentioned use case diagram has two actors :

- Student
- Teacher
- System

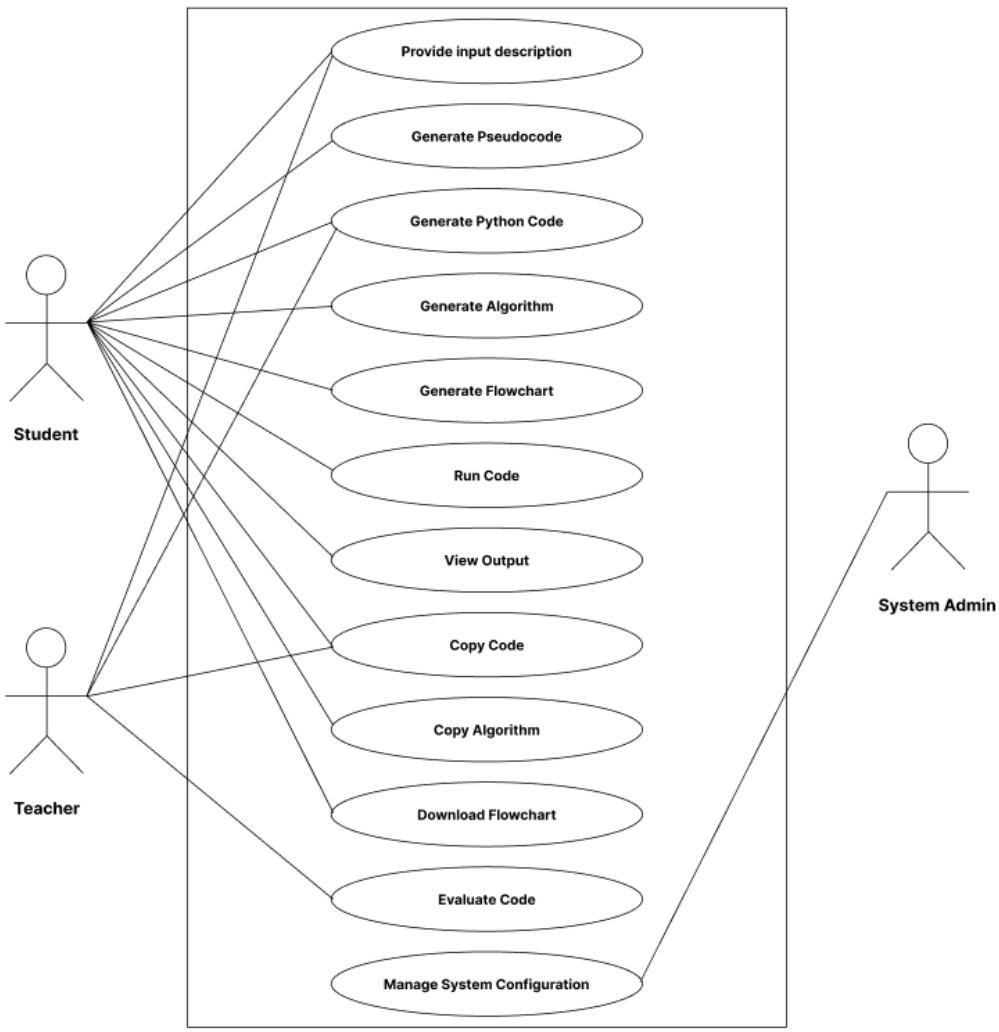


Fig. 5.5: Use Case diagram

5.4 DATA FLOW DIAGRAM

The data flow diagram (DFD) is used for classifying system requirements to major transformation that will become programs in system design. This is starting point of the design phase that functionally decomposes the required specifications down to the lower level of details.

• Bubbles: Represent the data transformations.

• Lines: Represent the logic flow of data.

Data can trigger events and can be processed to useful information. Systems analysis recognizes the central goal of data in organizations.

5.4.0.1 Description

- Process: Describes how each input data is converted to output data
- Data Store: Describes the repositories of data in a system.
- Data Flow: Describes the data flowing between process, Data stores and entities.
- Source: An external entity causing the origin of data
- Sink: An external entity, which consumes the data

5.4.1 Level 0

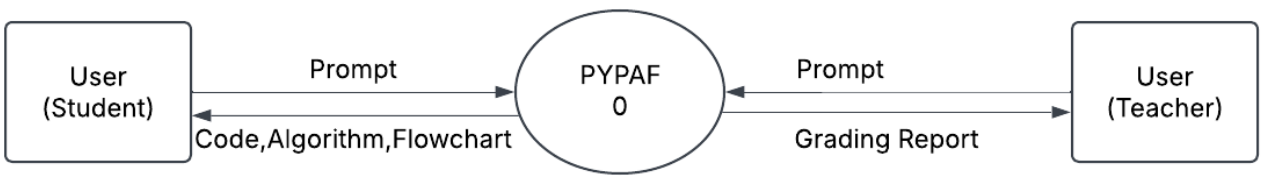


Fig. 5.6: Level 0 DFD

5.4.2 Level 1

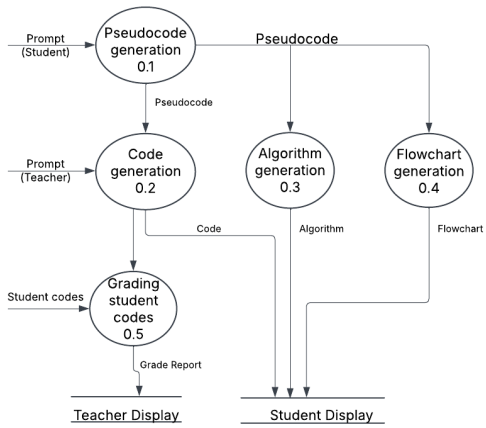


Fig. 5.7: Level 1 DFD

5.4.3 Level 2

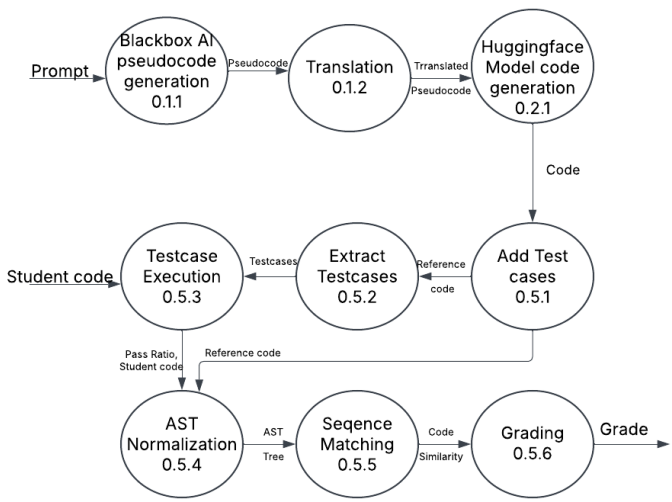


Fig. 5.8: Level 2 DFD

Chapter 6

IMPLEMENTATION

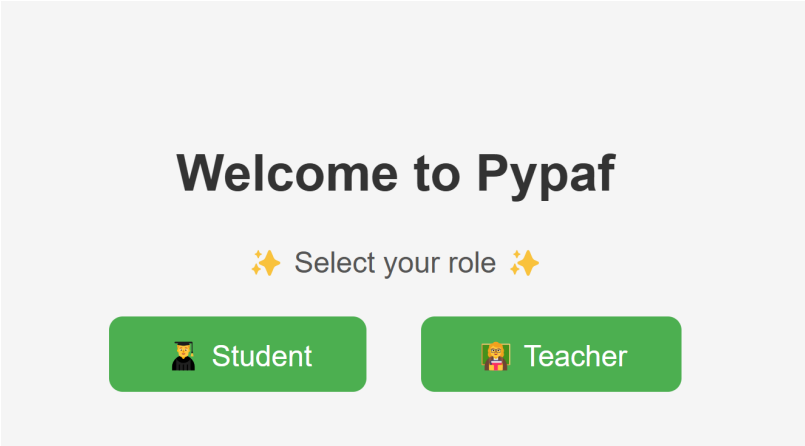


Fig. 6.1: User Interface

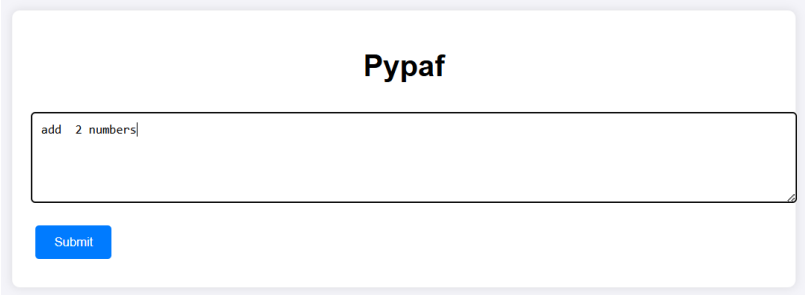


Fig. 6.2: Student-Input

Algorithm:

Step 1: Start
Step 2: Declare the variable 'num1, num2, sum as integer'.
Step 3: Display the message 'enter the first number:' and store to the variable 'num1'.
Step 4: Display the message 'enter the second number:' and store to the variable 'num2'.
Step 5: Assign value 'num1 + num2' to the variable 'sum'.
Step 6: Display the message 'addition results:", sum'.
Step 7: Stop

Fig. 6.3: Algorithm

Flowchart:

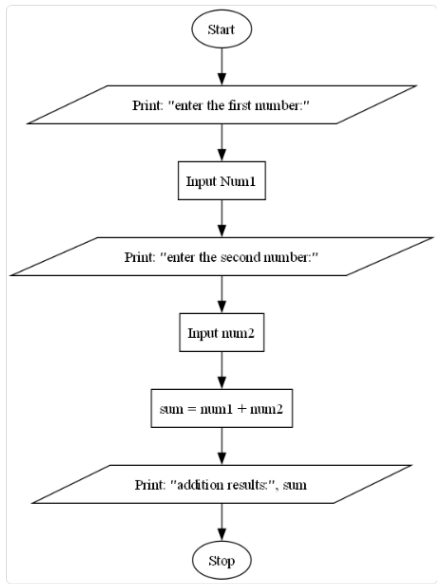


Fig. 6.4: Flowchart

Generated Python Code:

```
# variable declaration to store two numbers
Num1, Num2, Sum = 0, 0, 0

# request input from the user for two numbers
print("Enter the first number:")
Num1 = int(input())
print("Enter the second number:")
Num2 = int(input())

# add up both numbers
```

Copy Code Run Code

Fig. 6.5: Code Generated

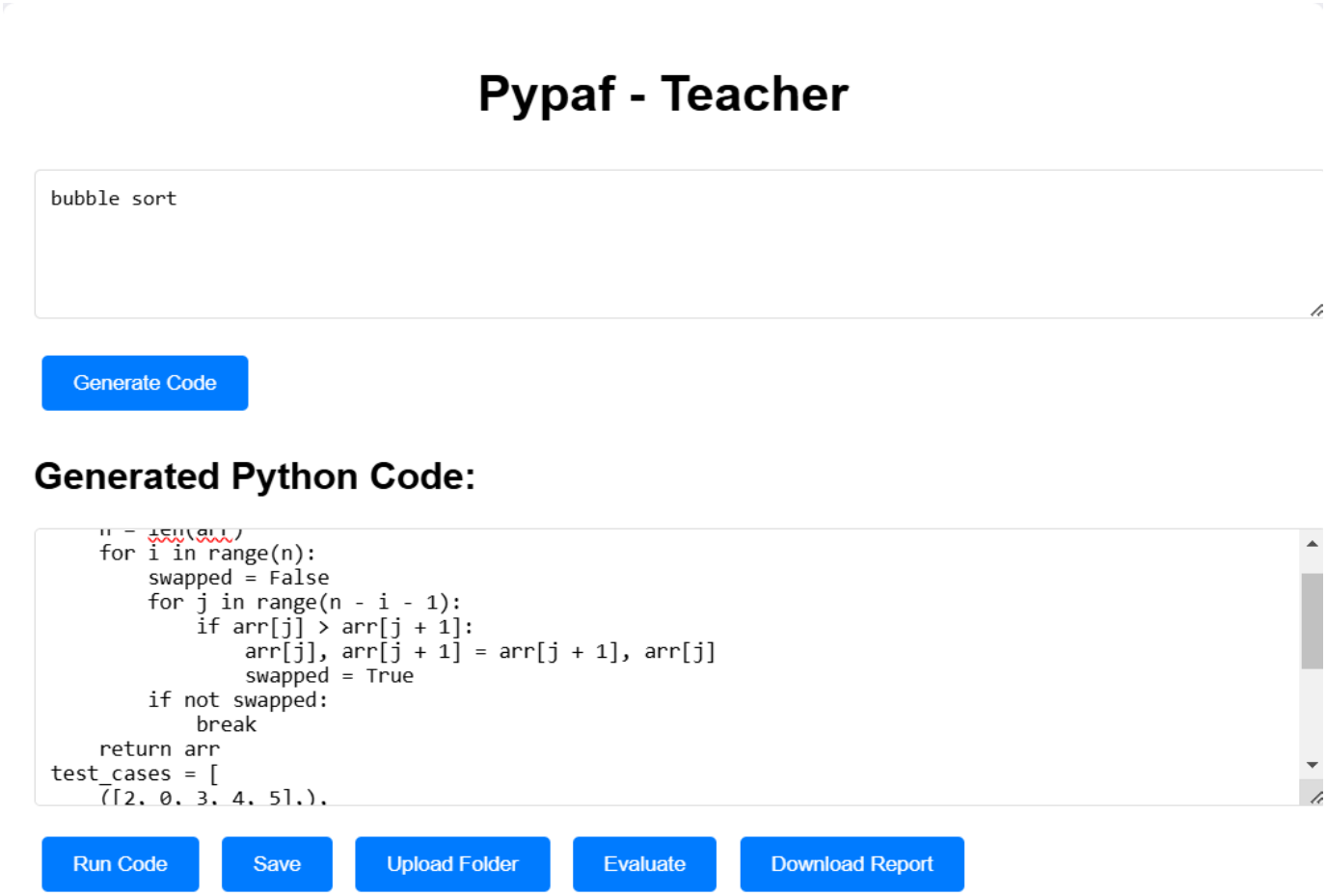


Fig. 6.6: Teacher-Input

Evaluation Results

Student Name	Algorithm Similarity (100)	Test Case Passed	Total Test Cases	Final Score (100)	Grade	Error
student1.py	100.00	0	6	40.00	C	Infinite Loop Detected
student2.py	95.80	0	6	38.32	C	Syntax Error: expected ':' (student2.py, line 4)
student3.py	100.00	6	6	100.00	S	
student4.py	94.91	3	6	67.97	B+	
student5.py	43.34	6	6	77.33	D	Algorithm does not follow expected structure
student6.py	100.00	6	6	100.00	S	
student7.py	82.73	6	6	93.09	S	
student8.py	100.00	6	6	100.00	S	
student9.py	0.00	0	6	0.00	F	Algorithm does not follow expected structure; Missing functions: bubblesort; No expected function found (expected: bubblesort); Missing return statement

Fig. 6.7: Evaluation Report

Chapter 7

CONCLUSION AND FUTURE SCOPE

The system for automatic code generation, algorithm creation, and flowchart visualization provides an efficient and intelligent solution for simplifying programming tasks. By automating the transformation of natural language input into pseudocode, algorithms, flowcharts, and executable code, the system enhances productivity and learning. The inclusion of an automated evaluation mechanism further ensures accurate and unbiased grading, reducing the workload for educators while offering students a structured and objective assessment of their code. This tool not only serves as a valuable resource for students looking to improve their coding skills but also provides an efficient way for teachers to evaluate assignments with minimal effort.

Looking ahead, several enhancements can further improve the system's functionality, making it more versatile and accessible. Expanding the tool to support multiple programming languages beyond Python will allow a broader audience to utilize its features effectively. Additionally, flowchart generation for function-based structures can be introduced, providing a clearer representation of modular programming concepts. The integration of AI-driven debugging, error detection, and code optimization will further refine the accuracy and efficiency of generated code, offering real-time recommendations to users.

Strengthening support for complex algorithms and data structures will make the system even more useful for tackling advanced programming challenges. Furthermore, incorporating cloud-based collaboration and online platform integration will enhance accessibility, allowing users to work remotely, share code, and collaborate in real time.

With these advancements, the system has the potential to become an indispensable tool for automated code generation, algorithmic learning, and programming education. By continuously evolving to incorporate new features and improvements, it will empower users to develop and refine their programming skills more effectively while streamlining the overall coding process.

References

[1] D. D. Karunarathna and N. Shafeek, "An Extensible Approach to Generate Flowcharts from Source Code," *International Journal of Research - Granthaalayah*, vol. 6, no. 9, pp. 505-519, 2018. https://doi.org/10.5281/zenodo.1465019.

[2] W. Wu, X. Xue, Y. Li, P. Gu, and J. Xu, "Code Similarity Detection using AST and Textual Information," *International Journal of Performability Engineering*, vol. 15, no. 10, p. 2683, 2019. https://doi.org/10.23940/ijpe.19.10.p14.26832691.

[3] S. Patade, P. Patil, A. Kamble, and M. Patil, "Automatic Code Generation for C and C++ Programming," *International Research Journal of Engineering and Technology (IRJET)*, vol. 8, no. 5, May 2021.

[4] A. Mastropaolo, L. Pascarella, E. Guglielmi, M. Ciniselli, S. Scalabrino, R. Oliveto, and G. Bavota, "On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot," arXiv, Feb. 2023. 10.48550/arXiv.2302.00438.

[5] A. Soliman, S. Shaheen, and M. Hadhoud, "Leveraging Pre-trained Language Models for Code Generation," *Complex Intelligent Systems*, vol. 10, pp. 3955–3980, 2024. https://doi.org/10.1007/s40747-024-01373-8.

[6] A. Darda and R. Jain, "Code Generation from Flowchart Using Optical Character Recognition and Large Language Model," 2024.

[7] HuggingFaceTB, "SmolLM2-1.7B-Instruct," 2024. Available: https://huggingface.co/HuggingFaceTB/SmolLM2-1.7B-Instruct.