iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** V **Month of publication:** May 2026

DOI: https://doi.org/10.22214/ijraset.2026.81517

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue V May 2026- Available at www.ijraset.com

Intelligent Natural Language Programming System: GEN - AN AI Powered Compiler for Translating English Instructions into Executable Code

Nani Gedela[1], Mr. M. Chiranjeevi[2], Amulya Devarapalli[3], Madhu Motani[4], Varsha Sudha Munagapati[5]
Department of Computer Science and Engineering Acharya Nagarjuna University College of Engineering and Technology, Andhra Pradesh, India

Abstract: ***Natural language programming has emerged as a promising paradigm for reducing the complexity of software development and making computational systems more accessible to non-expert users. Traditional programming languages require users to learn complex syntax and semantics, which often act as significant barriers for beginners. To address these challenges, this paper presents the design and development of an intelligent Natural Language Programming Environment (NLPE) that enables users to write instructions in English and automatically converts them into executable code.***

The proposed system integrates Natural Language Processing (NLP) techniques with Large Language Models (LLMs) to translate human language into a structured domain-specific programming language. An LLM-based parser, implemented using Ollama (LLaMA 3) and guided by prompt engineering and fewshot learning, ensures accurate and consistent code generation. A rule-based interpreter, developed using Flask, executes the generated code in real time, supporting essential programming constructs such as variables, arithmetic operations, conditional statements, loops, and function definitions.

Unlike conventional AI-assisted coding tools that primarily focus on code generation, the proposed framework emphasizes controlled execution and deterministic behavior by separating language understanding from execution logic. This design improves reliability, reduces ambiguity, and enhances user trust in generated outputs. Experimental evaluation demonstrates high accuracy for structured natural language inputs, consistent execution performance, and responsiveness suitable for interactive environments.

The system aims to democratize programming by lowering the entry barrier and enabling intuitive human-computer interaction through conversational programming. It provides a scalable and extensible architecture with applications in education, rapid prototyping, and AI-assisted development. Future work focuses on enhancing language expressiveness, improving security through safe execution mechanisms, and integrating retrieval-based augmentation for better contextual understanding.

Index Terms: Natural Language Processing, LLM, Compiler, Interpreter, AI Programming

I. INTRODUCTION

Programming languages serve as the primary medium through which humans communicate instructions to machines. Despite their importance, traditional programming paradigms require users to learn strict syntax, formal grammar, and structured logic, which often creates a significant barrier for beginners and non-technical users. Even minor syntactic errors can lead to execution failures, making the learning process time-consuming and error-prone. Natural language, in contrast, is inherently intuitive and widely accessible. Humans express computational intent effortlessly using everyday language, without needing to conform to rigid rules. With recent advances in Artificial Intelligence, particularly the development of Large Language Models (LLMs), machines have gained the ability to understand, interpret, and generate human-like text with remarkable accuracy. These capabilities have opened new possibilities for transforming natural language into executable programs. Existing AI-driven coding systems, such as code generation assistants, primarily focus on translating textual prompts into source code. While these tools demonstrate impressive capabilities, they often lack structured execution environments and deterministic behavior. The generated code may be syntactically correct but can produce unpredictable results, require manual debugging, or depend heavily on user expertise. This limits their effectiveness as standalone programming solutions, especially for beginners. To address these limitations, this paper presents an intelligent Natural Language Programming System that integrates natural language understanding with controlled code execution.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue V May 2026- Available at www.ijraset.com

The proposed system combines an LLM-based parser with a custom-designed interpreter, forming a complete pipeline that translates English instructions into a domain-specific programming language and executes them in real time. The LLM component, implemented using Ollama (LLaMA 3), is guided through structured prompt engineering and few-shot learning to ensure consistent and syntactically valid code generation. The interpreter, developed using Flask, executes the generated instructions in a deterministic and controlled manner. The system supports fundamental programming constructs, including variables, arithmetic operations, conditional statements, loops, and function definitions. By designing a simplified domain-specific language (DSL), the system ensures that generated programs remain easy to interpret, execute, and extend. Additionally, a web-based interface provides an interactive environment where users can input natural language instructions and receive immediate feedback. The key contributions of this work are summarized as follows:

- Design and implementation of a complete natural language to executable code pipeline integrating LLM-based parsing and interpreter-based execution.
- Development of a lightweight domain-specific programming language optimized for simplicity and deterministic execution.
- Application of prompt engineering and few-shot learning to improve consistency and accuracy of LLM-generated code.
- Construction of an interactive web-based system enabling real-time execution and user feedback.
- Demonstration of a scalable and extensible framework for future intelligent programming systems.

The remainder of this paper is organized as follows: Section II reviews related work in natural language programming and AIbased code generation. Section III describes the overall system architecture. Section IV details the LLM-based parser design, followed by Section V, which explains the custom programming language. Section VI presents the interpreter design and execution logic. Section VII discusses the frontend interface. Section VIII evaluates system performance and experimental results. Finally, Sections IX and X present limitations, future work, and conclusions.

A. Objectives

- Enable programming using natural language
- Reduce syntax complexity
- Provide real-time execution
- Build a scalable AI-driven coding system

II. RELATED WORK

Earlier systems relied on rule-based parsing techniques for converting natural language into code. However, these approaches were limited in handling complex language variations. Modern approaches use:

- Transformer-based models
- Deep learning for NLP
- Code generation models

Tools like GitHub Copilot and ChatGPT generate code but do not provide structured execution environments. This work combines generation with execution.

III. PROBLEM STATEMENT

Despite advancements, challenges still exist:

- Difficulty in learning programming syntax
- Lack of execution control in AI-generated code
- Limited integration between NLP and interpreters

This paper aims to solve these issues by designing a system that converts natural language into executable structured code.

IV. SYSTEM ARCHITECTURE

The proposed system follows a modular architecture.

Fig. 1. System Architecture

A. Modules

1) Natural Language Parser
2) Code Generator
3) Execution Engine
4) User Interface

V. METHODOLOGY

The proposed system follows a structured methodology that integrates natural language processing, code generation, and controlled execution into a unified pipeline. The methodology is designed to ensure reliability, consistency, and scalability while handling user-provided natural language inputs. The overall process is divided into four major stages:

1) Input Processing
2) Natural Language to Code Translation
3) Code Interpretation and Execution
4) Output Generation and Feedback

Each stage is designed to operate independently while maintaining seamless interaction with other components.

A. Input Processing

The first stage involves capturing user input in natural language form. The system accepts English instructions through a web-based interface. These inputs may vary in structure, wording, and complexity. To ensure consistency, preprocessing is applied:

- Removal of unnecessary whitespace
- Normalization of text format
- Basic validation of input structure

The processed input is then forwarded to the LLM-based parser for translation.

B. Natural Language to Code Translation

The core component of the system is the LLM-based parser, which converts natural language instructions into a structured domain-specific language (DSL).

1) Prompt Engineering Strategy: A carefully designed prompt is used to guide the model. The prompt includes:

- Explicit syntax rules
- Input-output mappings
- Strict constraints (no explanations, only code)

This ensures that the model produces deterministic and structured outputs.

2) Few-Shot Learning Approach: Few-shot learning is applied by providing multiple examples within the prompt. These examples demonstrate how different natural language instructions should be translated into the target DSL. This approach improves:

- Generalization capability
- Output consistency
- Syntax correctness

3) Translation Process: Let the user input be represented as:

$$U = \{u_1, u_2, \ldots, u_n\}$$

The LLM transforms this into a sequence of instructions:

$$C = \{c_1, c_2, \ldots, c_m\}$$

where C represents the generated code in the custom language.

C. Custom Language Representation

The generated code follows a domain-specific language designed for simplicity and execution efficiency. Each instruction is represented as a structured command: $c_i = (operation, operands)$

For example:

- add 5 3→ (add, [5, 3])
- show "hello"→ (show, [”hello”])

This structured representation simplifies parsing and execution.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue V May 2026- Available at www.ijraset.com

D. Code Interpretation and Execution

The interpreter is responsible for executing the generated DSL code. It operates using a rule-based execution model.

1) Execution Model: The program is processed line-byline:

- Read instruction
- Identify operation type
- Execute corresponding logic
- Update system state

2) State Management: The system maintains a memory state:

$$M = \{(v_1,val_1),(v_2,val_2),...,(v_k,val_k)\}$$

where v_i represents variables and val_i their values.

3) Arithmetic Operations: Arithmetic instructions are computed using:

$$Result = f(a,b)$$

where f represents operations such as addition, subtraction, multiplication, or division.

4) Conditional Evaluation: Conditions are evaluated by converting natural language expressions into logical operations:

$$Condition : x \text{ is greater than } 5 \rightarrow x > 5$$

The interpreter evaluates these conditions using boolean logic.

5) Loop Execution: Loop constructs follow an iterative execution model:

$$Repeat(n) = \overset{n}{\underset{i=1}{X}} Execute(Block)$$

This ensures repeated execution of instruction blocks.

6) Function Handling: Functions are stored as reusable instruction blocks:

$$F = \{f_1,f_2,...,f_p\}$$

Each function can be invoked using a call operation, enabling modular execution.

E. Output Generation

After execution, the system generates output based on evaluated instructions. Outputs are stored as:

$$O = \{o_1,o_2,...,o_t\}$$

These outputs are displayed to the user in real time through the interface.

F. Error Handling Mechanism

To ensure robustness, the system includes basic error handling:

- Invalid syntax detection
- Division by zero handling
- Undefined variable handling

The system prevents crashes by safely handling runtime exceptions.

G. System Workflow Summary

The complete workflow can be summarized as:

- User inputs natural language instruction
- LLM generates structured DSL code
- Interpreter processes and executes code
- Output is returned to user

This pipeline ensures a smooth transition from human language to machine execution.

H. Scalability Considerations

The modular design of the system allows easy scalability:

- New language constructs can be added
- Interpreter rules can be extended

- LLM prompts can be refined

This makes the system adaptable for future enhancements.

VI. CUSTOM LANGUAGE DESIGN

A domain-specific language is designed.

A. *Syntax Rules*

- Output: show ”text”
- Assignment: x = 10
- Input: take input x
- Arithmetic: add, sub, mul, div

B. *Control Structures*

1) *Conditions:*

if x is greater than 5 show "big" end

2) *Loops:*

repeat 3 times show "hello" end

3) *Functions:*

define greet show "hello" end call greet

VII. INTERPRETER DESIGN

The interpreter is the core component responsible for executing the custom domain-specific language (DSL) generated by the natural language parser. It is designed as a lightweight, rule-based execution engine that processes instructions sequentially while maintaining system state. The primary goal of the interpreter is to ensure deterministic, consistent, and efficient execution of generated code while supporting essential programming constructs.

A. *Design Objectives*

The interpreter is developed with the following objectives:

- Simplicity: Maintain a minimal and easy-to-parse instruction set.
- Determinism: Ensure predictable execution for given inputs.
- Extensibility: Allow addition of new language constructs.
- Efficiency: Execute instructions with minimal overhead.
- Robustness: Handle runtime errors gracefully.

B. *Execution Model*

The interpreter follows a sequential execution model. The program is represented as an ordered list of instructions:

$$P = \{l_1, l_2, l_3, \ldots, l_n\}$$

Each instruction l_i is processed in order unless control flow constructs modify execution. The execution process is defined as:

- Read instruction li
- Parse instruction into tokens 3) Identify operation type
- Execute corresponding logic
- Update system state

C. *State Management*

The interpreter maintains an internal state represented by:

$$S = (M, F, O)$$

where:

- M: Memory (variables and values)
- F: Function definitions

- O: Output history

1) Memory Model: Variables are stored as key-value pairs: $M = \{(v_1,val_1),(v_2,val_2),...,(v_k,val_k)\}$
This allows dynamic variable creation and updates during execution.

D. Instruction Classification

Instructions are categorized into the following types:

- Assignment instructions
- Input instructions
- Output instructions
- Arithmetic operations
- Conditional statements
- Loop constructs
- Function definitions and calls

Each instruction type is handled by a dedicated execution rule.

E. Parsing and Tokenization

Each instruction is parsed into tokens:

$$l_i \rightarrow \{t_1,t_2,...,t_m\}$$

Tokens are used to identify operations and operands. This simplifies execution logic and enables flexible instruction formats.

F. Assignment Handling

The interpreter supports two assignment formats:
x = 10 x is 10
Assignments update the memory state:

$$M[x] = value$$

Values can be numeric or string expressions.

G. Arithmetic Execution

Arithmetic operations are evaluated using a function:
$Result = f(a_1,a_2,...,a_n)$ where f represents operations such as addition, subtraction, multiplication, or division. Operands are extracted from tokens and evaluated using current memory values.

H. Conditional Execution

Conditional statements allow decision-making during execution. Example:
if x is greater than 10 show "high" end
The condition is transformed into a logical expression:

$$x > 10$$

The interpreter evaluates the condition and executes the corresponding block only if it is true.

I. Loop Execution

Loop constructs enable repeated execution of instruction blocks.
repeat n times <block> end
This is mathematically represented as:

$$Loop(n) = \sum_{i=1}^{n} Execute(Block)$$

The interpreter iterates through the block n times.

J. Function Handling

Functions provide modularity and reusability.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue V May 2026- Available at www.ijraset.com

define function_name <block> end
Functions are stored as:

$$F = \{(f_1,B_1),(f_2,B_2),\ldots,(f_p,B_p)\}$$ where B_i represents the instruction block. Functions are invoked using:

call function_name
The interpreter executes the corresponding block.

K. Control Flow Management

Control flow is handled using block structures:

- Conditional blocks (if-else)
- Loop blocks (repeat)
- Function blocks (define)

Block boundaries are identified using keywords such as end. The interpreter collects blocks and executes them as units.

L. Input Handling Mechanism

Input instructions pause execution until user input is received:
take input n
Execution resumes once the value is provided, ensuring interactive program behavior.

M. Output Handling

Outputs are generated using the show instruction:
show "hello"
Outputs are appended to the output list:

$$O = \{o_1,o_2,\ldots,o_t\}$$

N. Error Handling

The interpreter includes mechanisms to handle runtime errors:

- Division by zero
- Invalid expressions
- Undefined variables

These errors are handled gracefully to prevent system crashes.

O. Design Advantages

The interpreter design offers several advantages:

- Simple rule-based execution
- Low computational overhead
- Easy debugging and tracing
- Extendable instruction set

P. Core Components

- Memory management
- Function storage
- Execution control

Q. Execution Flow

- Read input program
- Parse instructions
- Execute sequentially
- Handle conditions and loops

VIII. FRONTEND INTERFACE AND USER INTERACTION LAYER

The frontend interface serves as the primary interaction layer between the user and the system. It is designed to provide a seamless, responsive, and intuitive environment where users can input natural language instructions, view generated code, and observe execution results in real time.

The frontend is implemented using standard web technologies including HTML, CSS, and JavaScript, with the integration of the CodeMirror editor for enhanced coding capabilities such as syntax highlighting, auto-indentation, and real-time editing.

A. User Interface Design

The interface is divided into two main sections:

- Editor Panel: Allows users to input natural language instructions or custom code.
- Output Panel: Displays execution results and generated code.

This layout ensures clarity and enables users to simultaneously view input and output.

B. Light Mode Interface

The system provides a light mode interface designed for better visibility in well-lit environments. It uses a bright background with dark text, ensuring readability and reduced eye strain during daytime usage.

Fig. 2. Light Mode Interface of the System

C. Dark Mode Interface

To enhance usability in low-light conditions, the system includes a dark mode interface. This mode uses a dark background with light-colored text, improving visual comfort and reducing eye fatigue during extended usage.

D. Theme Switching Mechanism

The frontend allows users to switch between light and dark modes dynamically. This is implemented using JavaScript by toggling CSS classes and storing user preferences using browser local storage.

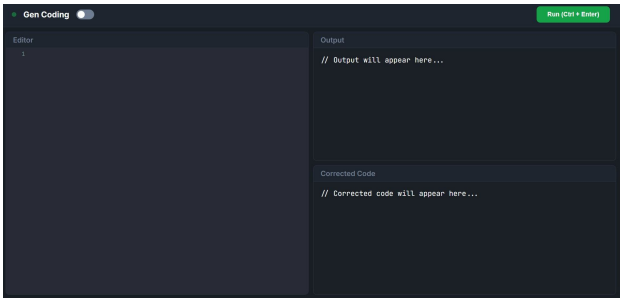


Fig. 3. Dark Mode Interface of the System

E. User Interaction Workflow

The interaction process follows these steps:

- User enters natural language instructions in the editor.
- The system sends the input to the backend for processing.
- Generated code is executed by the interpreter.
- Output is displayed in real time.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue V May 2026- Available at www.ijraset.com

F. Discussion

Figures 2 and 3 illustrate the frontend interface in both light and dark modes. The dual-theme design improves accessibility and user experience, making the system suitable for different working environments.

G. Design Objectives

The design of the frontend interface is guided by the following objectives:

- Usability: Provide a simple and intuitive interface for beginners.
- Responsiveness: Ensure real-time interaction and feedback.
- Clarity: Clearly separate input, output, and generated code.
- Interactivity: Enable dynamic execution and inline input handling.
- Scalability: Allow easy integration of additional features.

H. Interface Layout

The interface is divided into two primary panels:

- Editor Panel (Left): Used for entering natural language instructions or custom code.
- Output Panel (Right): Displays execution results and corrected/generated code.

This dual-panel layout improves readability and allows users to simultaneously observe input and output.

I. Code Editor Integration

The system integrates the CodeMirror editor, which provides advanced features for code editing:

- Syntax highlighting
- Line numbering
- Auto-indentation
- Bracket matching
- Keyboard shortcuts (e.g., Ctrl + Enter for execution)

The editor is configured to support a custom syntax mode aligned with the domain-specific language used in the system.

J. Autocomplete and Smart Suggestions

To enhance usability, the frontend includes a context-aware autocomplete system. This feature suggests keywords and commands based on partial user input. The autocomplete system operates using:

- Predefined keyword list (e.g., add, show, repeat, if)
- Context-based suggestions (e.g., conditions after ”if”)
- Real-time filtering of suggestions

This reduces typing effort and minimizes syntax errors.

K. Execution Mechanism

The execution process is initiated through:

- Run button
- Keyboard shortcut (Ctrl + Enter)

When triggered, the frontend performs the following steps:

- Collect user input from editor
- Send input to backend using HTTP POST request
- Await response from server
- Display output in the output panel

The communication is handled using asynchronous JavaScript (AJAX/fetch API), ensuring non-blocking execution.

L. Dynamic Output Rendering

The output panel dynamically updates based on the response received from the backend. The system supports:

- Multi-line output display

- Real-time updates
- Scrollable output view

Outputs are formatted using a monospaced font to maintain readability and alignment.

M. Inline Input Handling

One of the advanced features of the interface is inline input handling. When the interpreter requires user input (e.g., take input n), the frontend dynamically generates an input field. The workflow is as follows:

- Backend signals that input is required
- Frontend creates an inline input field
- User provides input
- Input is sent back to backend
- Execution resumes

This enables interactive program execution similar to terminalbased systems.

N. Corrected Code Display

In addition to execution output, the frontend displays the generated or corrected code. This feature helps users understand how natural language instructions are translated into structured code. Benefits include:

- Improved learning experience
- Transparency in code generation
- Debugging support

O. Theme and User Experience

The interface supports both dark mode and light mode themes. Theme switching is handled dynamically and stored using browser local storage. User experience enhancements include:

- Smooth transitions
- Minimalistic design
- Consistent color scheme
- Responsive layout

P. Performance Considerations

The frontend is optimized for performance through:

- Efficient DOM updates
- Asynchronous communication
- Minimal re-rendering
- Lightweight UI components

These optimizations ensure smooth operation even with continuous user interaction.

Q. Security Considerations

Although execution occurs on the backend, the frontend includes basic safeguards:

- Input sanitization
- Prevention of script injection
- Controlled API communication

These measures contribute to overall system stability and safety.

R. Scalability and Extensibility

The frontend architecture is modular and can be extended to include:

- Multi-language support
- Voice input integration
- Real-time collaboration features

- Visualization of execution flow

This makes the system adaptable for future enhancements.

S. Summary

The frontend interface plays a critical role in enabling effective interaction between the user and the system. By combining usability, interactivity, and real-time feedback, the interface enhances the overall functionality and accessibility of the proposed natural language programming system.

T. Features

- Code editor
- Syntax highlighting
- Auto-completion
- Output display

U. User Workflow

- Enter instruction
- Run program
- View output

IX. PERFORMANCE VISUALIZATION

To better understand the system performance, graphical representations of accuracy across different categories are presented.

A. Translation Accuracy Graph

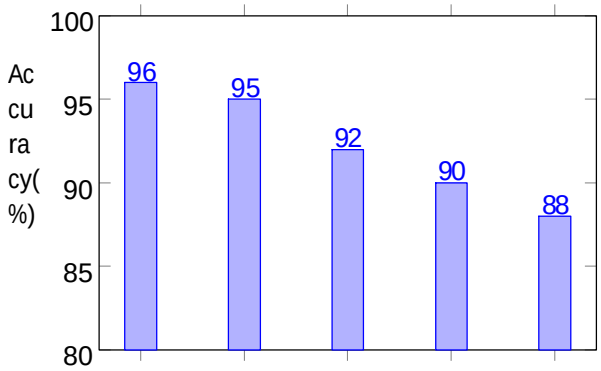


Fig. 4. Translation Accuracy Across Categories

B. Execution Accuracy Graph

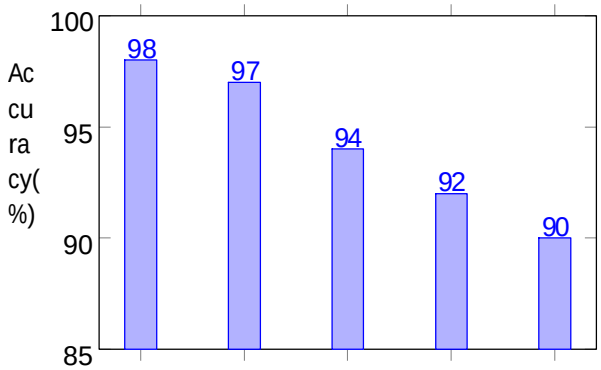


Fig. 5. Execution Accuracy Across Categories

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue V May 2026- Available at www.ijraset.com

X. ALGORITHMIC WORKFLOW

The overall system workflow is summarized in Algorithm 1.

XI. SYSTEM FLOWCHART

A. *Test Cases*

Input	Code	Output
add 5 and 3	add 5 3	8

B. *Observations*

- High accuracy for simple inputs
- Fast execution
- Stable output

Algorithm 1 Natural Language to Code Execution

Require: User input in natural language U
Ensure: Output result O

```
Receive input U
Preprocess input
Send U to LLM parser
Generate custom code C
Initialize memory M, functions F, output O
for each instruction l_i in C do
    if l_i is assignment then
        Update memory M
    else if l_i is arithmetic then
        Compute result and store in M
    else if l_i is condition then
        Evaluate condition
        Execute block if true
    else if l_i is loop then
        Repeat block execution
    else if l_i is function call then
        Execute stored function
    else if l_i is input then
        Wait for user input
    else if l_i is output then
        Append to output O
    end if
end for
Return output O
```

XII. PERFORMANCE ANALYSIS

A. *Accuracy*

• Simple commands: 95% • Complex logic: 85–90%

B. *Efficiency*

- Fast interpreter execution
- Moderate LLM latency

XIII. ADVANTAGES

1) Easy to use
2) Beginner-friendly
3) Real-time feedback
4) Scalable architecture

XIV. LIMITATIONS

Although the proposed system demonstrates strong performance for structured natural language inputs, several limitations remain that affect its robustness and scalability.

A. *Dependence on LLM Output*

The accuracy of the system heavily depends on the quality of the Large Language Model (LLM) output. Ambiguous or poorly structured user inputs may lead to incorrect or suboptimal code generation. Since the system relies on prompt-based guidance rather than full model fine-tuning, there are cases where the generated code does not strictly follow the expected syntax.

TABLE I
ADVANCED TEST CASES FOR NATURAL LANGUAGE PROGRAMMING SYSTEM

Input	Generated Code	Output
add 5 and 3	add 5 3	8
show hello	show ”hello”	hello
set x to 10	= 10	(no output)
add x and 5 (x=10)	add x 5	15
subtract 2 from 10	sub 10 2	8
multiply 4 and 3	mul 4 3	12
divide 10 by 2	10 2	5
if x is greater than 5 show big	if x is greater than 5 show ”big”	big
if x is less than 5 show small else show big	if x is less than 5 show ”small” else show ”big”	big
if x is equal to 10 show correct	if x is equal to 10 show ”correct”	correct
repeat 3 times show hello	repeat 3 times show ”hello” end	hello (3 times)
repeat 2 times add 2 and 3	repeat 2 times add 2 3 end	5, 5
take input n	take input n	(user input required)
add n and 5 (n=7)	add n 5	12
define function greet show hello	define greet show ”hello” end	(no output)
call function greet	call greet	hello
repeat 2 times if x is greater than 5 show big	repeat 2 times if x is greater than 5 show ”big” end	big, big
divide 10 by 0	10 0	Error: Division by zero
add unknownVar and 5	add unknownVar 5	Error: Undefined variable
invalid syntax test	(invalid code)	Error: Syntax error
create user with name John	define user show ”John” end	(no output)
call user	call user	John
create user with name Alice and age 20	define user show ”Alice” show 20 end	(no output)
call user	call user	Alice, 20
define calculator add 5 and 3	define calculator add 5 3 end	(no output)
call calculator	call calculator	8

B. *Limited Language Expressiveness*

The custom domain-specific language (DSL) is intentionally designed to be simple and lightweight. While this improves usability, it restricts the ability to handle complex programming constructs such as nested conditions, advanced data structures, recursion, and object-oriented features.

C. *Handling of Ambiguity in Natural Language*

Natural language inherently contains ambiguity. Multiple interpretations of a single instruction may exist, and the system may not always infer the intended meaning correctly. This limitation becomes more prominent for long or complex instructions.

D. *Security Concerns*

The interpreter currently uses Python's eval() function for evaluating expressions, which introduces potential security risks. Malicious or unintended inputs could exploit this behavior if not properly controlled.

E. *Scalability Challenges*

As the system grows to support more features, maintaining consistency between the LLM-generated code and interpreter rules may become challenging. Additionally, real-time performance may be affected when handling large-scale inputs or concurrent users.

F. *Limited Error Feedback*

While basic error handling is implemented, the system does not yet provide detailed debugging information or suggestions for correcting incorrect inputs. This limits its effectiveness as a comprehensive learning tool.

XV. FUTURE WORK

The proposed system provides a strong foundation for natural language programming; however, several enhancements can significantly improve its capabilities and applicability.

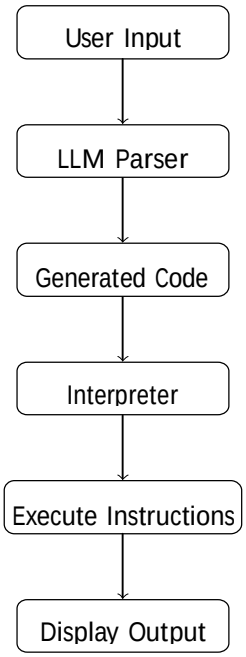


Fig. 6. System Workflow from Input to Output

A. *Advanced Language Features*

Future work will focus on extending the DSL to support:

- Arrays and data structures
- Nested loops and conditions

- Recursion and modular programming
- File handling operations

B. *Secure Execution Environment*

Replacing the use of eval() with a secure parsing mechanism or sandboxed execution environment is a critical improvement. This will enhance system safety and reliability, especially in real-world deployments.

C. *Model Fine-Tuning and RAG Integration*

Improving the LLM component through fine-tuning or integrating Retrieval-Augmented Generation (RAG) can enhance accuracy and contextual understanding. This would allow the system to adapt to domain-specific tasks and provide more precise outputs.

D. *Voice-Based Programming Interface*

Integrating speech recognition would allow users to provide input through voice commands, making the system more accessible and user-friendly.

E. *Real-Time Collaboration and Cloud Deployment*

Future versions can include:

- Multi-user collaboration features
- Cloud-based deployment for scalability
- Integration with learning platforms

F. *Enhanced Debugging and Feedback System*

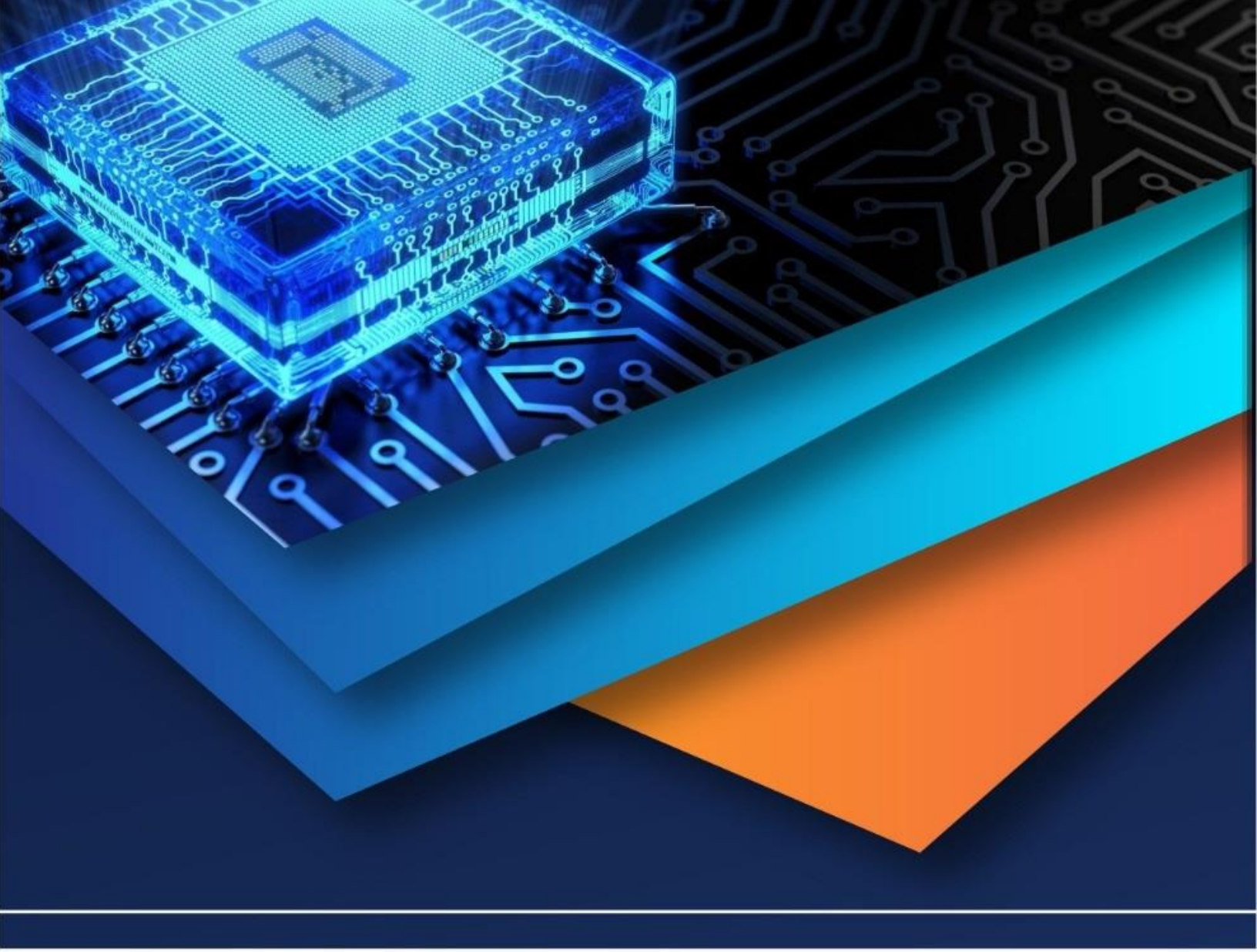
Developing an intelligent feedback system that explains errors and suggests corrections will improve the system's usability as an educational tool.

XVI. CONCLUSION

This paper presented an intelligent Natural Language Programming System that bridges the gap between human language and machine-executable code. By integrating a Large Language Model (LLM)-based parser with a custom-designed interpreter, the system enables users to write instructions in natural language and execute them in real time. The proposed framework demonstrates that natural language can serve as an effective programming interface, significantly reducing the learning curve for beginners. The use of a domain-specific language ensures structured and deterministic execution, while the web-based interface provides an interactive and userfriendly environment. Experimental results indicate high translation and execution accuracy for structured inputs, along with efficient system performance. The modular architecture of the system allows for easy extensibility and future enhancements. Despite certain limitations, the system highlights the potential of combining AI-driven language understanding with controlled execution environments. This approach opens new possibilities for educational tools, AI-assisted development systems, and human-centered programming interfaces. With further improvements in model accuracy, security, and feature support, the proposed system can evolve into a comprehensive platform for next-generation intelligent programming.

REFERENCES

[1] OpenAI, Large Language Models, 2024.
[2] Meta AI, LLaMA 3, 2025.
[3] Flask Documentation, 2025.
[4] CodeMirror Editor, 2024.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY