INTERNATIONAL JOURNAL OF RESEARCH, EXPLORATION & KNOWLEDGE (IJREK)
Official Scholarly Journal Portal: **https://www.ijrek.org** | DOI: **https://doi.org/10.5281/zenodo.23857516**

AI-BASED CODE GENERATION AND SELF-TESTING SYSTEM

Dr.Taware G.G[1], Thorat Tejaswee[2]

[1] Faculty Guide & Supervisor | [2] Dattakala Group of Institutions, Faculty of Engineering, Pune, Maharashtra, India, India

Corresponding author: **tejasweethorat@gmail.com**

ARTICLE ID	ARTICLE TYPE	RESEARCH AREA	PUBLICATION
IJREK-2026-00182	Research Article	Artificial Intelligence	Volume 1, Issue Issue 1 (2026)
RECEIVED	**REVISED**	**ACCEPTED**	**PUBLISHED**
30 October 2026	02 November 2026	31 October 2026	03 November 2026

Abstract

Software development often requires programmers to write repetitive code, understand unfamiliar requirements, and spend considerable time identifying and correcting errors. Recent advances in artificial intelligence have enabled systems to generate source code from natural-language instructions. However, generated code may contain syntax errors, incorrect logic, insecure patterns, or incomplete handling of edge cases. This paper proposes an AI-Based Code Generation and Self-Testing System that accepts a programming task in natural language, generates a candidate solution, prepares or selects test cases, executes the code in a controlled environment, and analyses the results. The proposed workflow combines prompt interpretation, code generation, static validation, test-case generation, sandboxed execution, and feedback-based refinement. The system is intended to support developers and learners by reducing repetitive effort and providing an initial validation cycle. It does not replace code review or guarantee correctness; execution safety, test coverage, model limitations, and human verification remain important. This paper presents a proposed methodology and evaluation plan. Quantitative performance claims should be added only after implementation and testing.

Keywords: *Artificial intelligence; code generation; automated testing; natural-language processing; software engineering; test-case generation; program validation*

I. INTRODUCTION

A. Background / Context Modern software applications are developed using multiple programming languages, frameworks, and libraries. Developers frequently translate requirements into code, debug errors, and prepare test cases to verify program behaviour. These activities can be time-consuming, particularly for beginners or when working with unfamiliar APIs. AI-assisted code-generation tools can convert natural-language descriptions into candidate code, while automated testing techniques can execute programs against defined inputs and expected outputs. Combining these capabilities can create a workflow in which generated code is checked and improved before it is presented to the user. B. Problem Statement A code-generation model may produce code that appears plausible but does not satisfy the stated requirements. It can fail to compile, return incorrect results, omit boundary conditions, or use unsafe operations. Reviewing every generated solution manually reduces the time-saving benefit of automation. A useful system therefore needs to validate generated code, create meaningful tests, execute it safely, identify failures, and provide actionable feedback without claiming that a limited set of tests proves complete correctness. C. Aim and Objectives • Develop a workflow that generates program code from structured natural-language instructions. • Interpret requirements and identify the target programming language, inputs, outputs, and constraints. • Generate or derive test cases, including normal, boundary, and invalid-input scenarios. • Compile or execute candidate code in a restricted environment and capture results and errors. • Use test feedback to support controlled code refinement and re-testing. • Evaluate code quality and testing effectiveness using measurable criteria. D. Significance and Scope The proposed system focuses on educational and general-purpose programming tasks for supported languages. It aims to assist with first-draft code generation and preliminary validation. The initial version should clearly define supported languages, execution limits, test formats, and the types of tasks it can handle. Generated code must be reviewed before use in production, especially when it accesses files, networks, databases, credentials, or other sensitive resources.

II. LITERATURE REVIEW / RELATED WORK

A. Overview of Existing Research Research in automated programming includes program synthesis, neural code generation, large language models, static analysis, and automated software testing. Earlier approaches used templates, grammar-based generation, and search over program structures. More recent systems use machine-learning models trained on code and natural-language pairs to generate candidate programs. Separately, software-testing research provides methods such as unit testing, property-based testing, mutation testing, and coverage analysis to assess program behaviour. B. Key Research Themes Natural-language understanding is used to extract requirements from user prompts. Code-generation models predict source-code tokens or structured program representations. Static analysis can detect selected syntax, style, type, and security issues without executing the program. Dynamic testing evaluates actual behaviour by running code with selected inputs. Test generation may use examples, boundary-value analysis, equivalence classes, constraints, or model-assisted suggestions. Combining these methods provides broader validation than relying on generated code alone. C. Research Gap Code-generation quality and test quality are closely related: weak or incomplete tests may allow incorrect generated programs to pass. Generated tests can also reflect the assumptions of the same model that produced the code, creating a risk that both share the same misunderstanding of the requirement. Execution environments introduce additional concerns, including resource exhaustion, unsafe system calls, dependency handling, and language-specific behaviour. A practical student-level system should therefore separate generation from evaluation where possible, use explicit expected outputs or independent test oracles, enforce sandbox limits, and report the scope of validation. D. Contribution of the Proposed Work This paper proposes a modular pipeline that links requirement interpretation, code generation, test preparation, controlled execution, result analysis, and iterative refinement. It emphasizes traceable test cases, execution safeguards, and transparent reporting of limitations. Any claims about accuracy, pass rate, or time savings must be based on experiments conducted with a documented task set.

III. PROPOSED SYSTEM

The proposed system accepts a programming task written in natural language and identifies its requirements. A code-generation component produces a candidate solution. The system then performs basic validation, prepares test cases, and executes the candidate inside a restricted environment. Test outcomes, compiler messages, and runtime errors are analysed and returned as feedback. If refinement is enabled, the system may request a revised candidate and repeat the validation cycle up to a configured limit. Component Main responsibility User input interface Accepts task description, language, constraints, and sample input/output. Requirement parser Extracts functional requirements, data types, and constraints. Code generation module Produces candidate source code using an AI model or configured generator. Static validation Checks syntax, formatting, and selected safety rules before execution. Test-case module Creates or stores normal, boundary, and invalid-input test cases. Execution sandbox Runs code with time, memory, filesystem, and network restrictions. Feedback and refinement Summarizes failures and supports a bounded re-generation cycle. Result interface Displays code, test outcomes, warnings, and validation limitations. A. Basic Workflow • Receive a natural-language programming task and selected language. • Extract requirements, constraints, sample inputs, and expected outputs where provided. • Generate a candidate program and perform syntax or static checks. • Prepare test cases using examples and additional boundary or invalid-input cases. • Execute the candidate in a restricted sandbox and collect outputs, errors, and resource usage. • Compare actual results with expected results and summarize failed tests. • Optionally refine the candidate using failure feedback and repeat within a fixed retry limit. B. System Architecture Programming Task / Requirements ↓ Requirement Parsing and Validation ↓ AI Code Generation ↓ Static Checks and Test-Case Preparation ↓ Restricted Code Execution ↓ Test Result Analysis and Feedback ↓ Validated Candidate Code and Test Report

Fig. 1. Proposed AI-based code generation and self-testing architecture. C. Main Functional Requirements The system should accept supported task descriptions, generate code for configured languages, validate input and generated output, run tests in a restricted environment, and present clear results. It should handle empty prompts, unsupported languages, compilation failures, timeouts, and missing dependencies gracefully. The system should preserve the association between each test case and its expected result. D. Non-Functional Requirements The system should be secure, maintainable, responsive for typical tasks, and reproducible. Execution limits must be enforced consistently. API keys and user code should be protected. Logs should avoid storing sensitive source code or credentials unless explicitly required and appropriately protected. The architecture should allow supported languages and test strategies to be added without redesigning the complete workflow.

IV. METHODOLOGY

A. Requirement and Task Collection The system should receive a clearly described programming task. A task record may include a problem statement, target language, input format, output format, constraints, examples, and expected results. The evaluation dataset should contain tasks with reference solutions or independently prepared test oracles. The source and permitted use of any benchmark dataset should be documented. Field Example format Purpose

PROBLEM STATEMENT

Natural-language description Defines the requested behaviour. Programming language Python / Java / C++ Selects the generation and execution toolchain. Input specification Integer, string, array, etc. Defines accepted input. Expected output Exact value or rule Supports automated result comparison. Constraints Ranges, limits, conditions Guides implementation and test design. B. Prompt Processing and Code Generation The requirement parser should identify the task objective, language, input and output definitions, constraints, and examples. These details are assembled into a structured generation request. The code generator returns source code and, where supported, a short explanation or assumptions. The system should preserve the original task and record the model or generator configuration so that experiments can be reproduced. Generated output should be checked for malformed formatting before it is passed to later stages. C. Test-Case Generation Test cases should cover typical inputs, boundary values, empty or invalid inputs where applicable, and cases likely to expose logical errors. User-provided examples can be included, but should not be the only tests. Expected outputs should come from the task specification, a trusted reference implementation, or independent manual preparation. When an expected result cannot be determined reliably, the system should label the test as exploratory rather than report it as a definitive pass or fail. D. Static Validation and Sandboxed Execution Before execution, the system can perform syntax checks, language-specific compilation, and selected static rules. Candidate programs should run in an isolated environment with restricted permissions, execution time, memory, process creation, filesystem access, and network access. The system should capture standard output, standard error, exit status, compilation messages, and timeout events. Untrusted generated code must never be executed directly on the application host with unrestricted privileges. E. Result Analysis and Iterative Refinement For each test, actual output is compared with the expected result using a documented comparison rule. The result analyser classifies tests as passed, failed, or not evaluated and summarizes relevant errors. If refinement is enabled, only the failure summary and necessary task context should be supplied to the generator. A maximum retry count should prevent endless loops. The final report should distinguish the original candidate from revised candidates and retain test outcomes for each attempt. F. Responsible Use and Limitations • Passing a finite test suite does not prove that a program is fully correct or secure. • Generated code may reproduce errors, insecure patterns, or unsupported assumptions. • Test cases generated by the same model may fail to detect the model's own mistakes. • Sandbox restrictions, dependency versions, and language runtimes affect execution results. • Human review is required before generated code is used in important or production systems.

V. SYSTEM DESIGN AND IMPLEMENTATION PLAN

Layer Possible student-level choice Frontend HTML, CSS, JavaScript, or a simple React interface. Backend Python Flask or FastAPI for request handling and orchestration. AI integration A configured code-generation model/API or a local model where feasible. Static checks Language compiler, parser, formatter, and selected lint/security rules. Testing Unit-test runner and custom input/output comparison module. Execution Docker or another isolated sandbox with strict resource limits. Storage Local files or database for permitted task metadata and test reports. Deployment Local demonstration or approved hosting with execution isolation. A. Main Modules • Input module: receives the task description, target language, and constraints. • Requirement module: converts the description into structured task information. • Generation module: requests and receives candidate source code. • Validation module: performs syntax checks and selected static analysis. • Test module: prepares test cases and expected results. • Sandbox module: executes candidate code under configured restrictions. • Analysis module: compares outputs, classifies failures, and prepares feedback. • Report module: presents source code, test summary, warnings, and limitations. B. Implementation Steps • Select supported programming language(s), task types, and generation method. • Prepare a documented set of programming tasks and reference test cases. • Implement requirement parsing and candidate code generation. • Integrate syntax checks, test-case preparation, and a restricted execution environment. • Build output comparison, failure reporting, and bounded refinement. • Test error handling, sandbox restrictions, repeatability, and user-interface behaviour. C. Data Flow Input Processing Output Task description Requirement parsing Structured requirements Structured requirements AI code generation Candidate source code Candidate code Static checks and sandbox execution Execution logs and test results Test results Comparison and feedback analysis Pass/fail summary and suggested refinement Final candidate Result presentation Code and validation report D. Interface and Deployment A simple interface can include a task-description field, language selector, generate action, source-code display, test summary, and error details. The backend should separate model calls from code execution. The execution service should use disposable isolated environments and should not expose host credentials, private files, or unrestricted network access. Deployment should be attempted only after sandbox behaviour and resource limits have been verified. E. Security and Maintenance The application should validate request sizes, protect model credentials, restrict execution privileges, and record dependency versions. Generated code and user prompts should be handled as untrusted input. Logs should be access-controlled and retention should be limited. Model changes, prompt changes, compiler updates, and sandbox configuration changes should be versioned because they may affect output and evaluation results.

VI. RESULTS AND EVALUATION PLAN

Results must be obtained from the implemented system using a documented task set. No code-generation accuracy, test pass rate, or time-saving values are asserted in this proposed paper. Evaluation should report task difficulty, programming language, model configuration, test-oracle source, execution environment, and failure categories. Test ID Test case Expected result T1 A valid task with clear input/output requirements is submitted. System generates candidate code and displays it. T2 The prompt is empty or contains no usable task. System requests a valid programming task. T3 Generated code contains a syntax error. Static check or compilation reports the error. T4 A test case produces an incorrect output. System marks the test as failed and records the mismatch. T5 Candidate execution exceeds the time limit. Sandbox stops execution and reports a timeout. T6 The task includes boundary or invalid inputs. Applicable tests are executed and results are reported. T7 The refinement retry limit is reached. System stops retries and presents the latest result. A. Evaluation Metrics • Task success rate: proportion of tasks for which the final candidate passes the defined test suite. • Test pass rate: proportion of executed test cases that pass under the documented oracle. • Compilation or syntax success: proportion of generated candidates accepted by the language toolchain. • Execution failure rate: proportion of runs ending in runtime errors, crashes, or timeouts. • Refinement effectiveness: change in test outcomes between the initial and final candidate. • Generation time: elapsed time from task submission to candidate output, reported with the test configuration. • Test coverage: statement or branch coverage where supported by the selected language and tools. B. Discussion The discussion should interpret measured outcomes and investigate why tasks fail. A high test pass rate may reflect a limited or weak test suite rather than general correctness. Exact output comparison may be unsuitable for tasks where multiple outputs are valid or ordering is unspecified. Results should separate failures caused by generation, compilation, runtime behaviour, test-oracle quality, and sandbox configuration. Any comparison between models or settings should use the same task set and evaluation procedure. C. Comparative Analysis Parameter Manual coding and testing Code generation only Proposed generation + self-testing Initial effort Developer writes code and tests Prompting and reviewing generated code Prompting plus automated validation cycle Error detection Depends on developer tests and review Often relies on user inspection Runs defined tests and reports failures Feedback Manual debugging process May provide explanation on request Uses test and execution feedback for refinement Main risk Human oversight or incomplete tests Plausible but incorrect generated code Weak tests, sandbox gaps, or shared model assumptions

VII. ADVANTAGES AND LIMITATIONS

Advantages Limitations Can reduce repetitive first-draft coding effort. Generated code may be logically incorrect or incomplete. Provides a repeatable initial testing workflow. Passing tests does not prove complete correctness. Can identify selected syntax and runtime failures early. Test quality and expected-output accuracy are critical. Feedback can help guide bounded code refinement. Repeated generation may not resolve ambiguous requirements. Can support programming education and experimentation. Safe execution requires careful sandbox design and maintenance.

VIII. APPLICATIONS

• Educational programming environments for learning and practice. • Rapid prototyping of small functions and algorithmic solutions. • Initial validation of programming exercises and coding challenges. • Developer assistance for generating unit-test candidates and debugging hints. • Demonstrations and research in AI-assisted software engineering. • Internal tools that support code review when combined with human approval.

IX. FUTURE SCOPE

• Support additional programming languages and framework-specific environments. • Add property-based testing, fuzz testing, mutation testing, and coverage-guided test generation. • Use independent test generation or reference solutions to reduce shared-model blind spots. • Integrate static security analysis and dependency vulnerability checks. • Improve requirement clarification by asking users about ambiguous specifications. • Explore retrieval-augmented generation using approved documentation and API references. • Evaluate performance across task difficulty levels and real-world maintenance scenarios.

X. CONCLUSION

This paper proposes an AI-Based Code Generation and Self-Testing System that combines natural-language task interpretation, candidate code generation, static validation, test-case preparation, restricted execution, and feedback-based refinement. The approach aims to make AI-assisted programming more useful by checking generated code against defined tests instead of presenting generated output without validation. Its effectiveness depends on requirement clarity, generation quality, independent and meaningful test cases, reliable expected outputs, and secure execution. The proposed system should be treated as a developer-assistance tool rather than a guarantee of correctness or security. Future work can extend test-generation strategies, language support, security analysis, and evaluation on broader programming tasks. The final study should report only measurements obtained from the actual implementation.

REFERENCES

[1] M. Chen et al., “Evaluating Large Language Models Trained on Code,” arXiv:2107.03374, 2021.

[2] D. Hendrycks et al., “Measuring Coding Challenge Competence With APPS,” Advances in Neural Information Processing Systems, vol. 34, 2021.

[3] J. Austin et al., “Program Synthesis with Large Language Models,” arXiv:2108.07732, 2021.

[4] M. Allamanis, M. Brockschmidt, and M. Khademi, “A Survey of Machine Learning for Big Code and Naturalness,” ACM Computing Surveys, vol. 51, no. 4, 2018.

[5] P. Ammann and J. Offutt, Introduction to Software Testing, 2nd ed., Cambridge University Press, 2016.

[6] A. Zeller, Why Programs Fail: A Guide to Systematic Debugging, 2nd ed., Morgan Kaufmann, 2009.

[7] ISO/IEC/IEEE 29119-1:2022, Software and Systems Engineering—Software Testing—Part 1: General Concepts, 2022.

Publication archival note: Official scholarly contribution published under CC BY 4.0 International License. Permanent record on **https://www.ijrek.org** | Article ID: IJREK-2026-00182