



International Journal of Innovative Research in Science Engineering and Technology (IJIRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)



Impact Factor: 9.935

Volume 15, Issue 10, October 2026

Design and Implementation of NextIDE: A Web-Based AI-Powered Algorithm-to-Code Platform

Aousekar Anjum Samir¹, Jahgirdar Syed Noufil Syed Mujahid², Razvi SayyedSaad Nihal³,
Zartar Arshiya Irshad Husain⁴, Patel Mohammad Zaki Rafique Ahmed⁵, Prof. M. N. Gaikwad⁶

Department of Computer Science and Engineering, Brahmdevdada Mane Institute of Technology, Solapur,
Maharashtra, India¹⁻⁵

Project Guide, Department of Computer Science and Engineering, Brahmdevdada Mane Institute of Technology,
Solapur, Maharashtra, India⁶

ABSTRACT: Students and beginner programmers often understand the logic of an algorithm but find it difficult to express it as syntactically correct code in a specific programming language. This paper presents NextIDE, a web-based software prototype that uses generative artificial intelligence to convert natural-language algorithm descriptions into source code. The system accepts a description and a target language, and uses Google Gemini 2.5 Flash to return source code, an explanation, time and space complexity estimates, assumptions, and potential edge cases. It also provides AI-assisted test-case generation, code execution through Judge0 CE, suggested error corrections, authentication, and project history. The application is built with React, Next.js, TypeScript, and Tailwind CSS, with MongoDB (Mongoose) for persistence and NextAuth for authentication, and supports Java, Python, JavaScript, TypeScript, C++, C, and Go. The prototype was designed around three representative problems (a restaurant bill splitter, a shopping discount calculator, and a fuel cost calculator) covering arithmetic and conditional logic. However, systematic test records, measured generation latency, and a code-generation success rate across all supported languages are not available; therefore, no overall accuracy or pass rate is reported. NextIDE is intended as an educational aid for generating and examining algorithmic solutions, and its generated code must be reviewed and tested before use. Further systematic testing and independent evaluation are required to assess its reliability.

KEYWORDS: NextIDE; AI code generation; natural-language programming; Gemini 2.5 Flash; Judge0; Next.js; programming education.

I. INTRODUCTION

Programming involves converting a problem-solving approach into instructions that a computer can execute. Differences between programming languages, input/output requirements, debugging, and the handling of edge cases increase the effort needed to turn an understood algorithm into working code. Recent large language models (LLMs) can interpret written instructions and produce candidate implementations in several languages, but generated code may contain logical errors or incorrect assumptions and must still be reviewed and tested.

NextIDE was developed as a web-based prototype that brings several programming-assistance activities into one application: describing an algorithm in English, generating code in a chosen language, reading an explanation with complexity estimates, generating test cases, executing the program, requesting error-correction suggestions, and saving project history.

The project addresses a software-integration problem: connecting a generative AI model, a code-execution service, a database, and an authentication layer behind a single interface. It does not establish that the generated code is correct for arbitrary problems. Published work on code-generation models illustrates why implementation and evaluation should be reported separately. Chen et al. evaluated Codex on the HumanEval benchmark and noted limitations on complex instructions [1]; Austin et al. showed that performance depends on model scale and task complexity [2]. These results concern their own models and benchmarks and are not attributable to NextIDE.

The aim of this paper is to describe the design and implementation of NextIDE as a web-based algorithm-to-code platform, reporting the software work actually completed without making unsupported performance claims.

Aim: To design and implement NextIDE, a web-based prototype that converts natural-language algorithm descriptions into source code and supports explaining, testing, executing, and correcting the generated implementation.

Objectives:

1. To generate code from natural-language instructions using AI.
2. To support multiple programming languages through a common workflow.
3. To explain the generated code, including time and space complexity.
4. To generate test cases and execute code through an external execution service.
5. To help users fix errors in their programs.
6. To maintain projects and code-generation history.
7. To identify the testing and independent evaluation still needed before making claims about reliability.

II. LITERATURE REVIEW

AI-based code generation has progressed from code-completion tools to models that produce whole programs from natural-language descriptions. Chen et al. introduced Codex, a language model adapted for programming, and evaluated functional correctness on HumanEval; they reported both the promise of the approach and difficulty with complex instructions and consistent correctness [1]. Austin et al. studied program synthesis with large language models and introduced benchmarks for generating code from problem descriptions, showing that performance depends on model scale, training, and task complexity [2]. Nijkamp et al. presented CodeGen, a family of models for program synthesis, and explored multi-step synthesis in which a task is broken into stages [3].

These studies support the central idea behind NextIDE, that a user can describe intended logic in ordinary language and obtain an initial implementation. They also imply that the clarity of the description matters: vague instructions can lead to incorrect assumptions. NextIDE therefore requests structured output that includes assumptions and edge cases alongside the code.

Existing approaches to turning algorithms into code can be compared as follows.

Approach	Main purpose	Limitation
Manual programming	Converts logic into code through direct implementation	Requires knowledge of syntax and language rules
Online tutorials and examples	Provide reference solutions and explanations	Examples may not match the user's exact problem
Code completion tools	Suggest code while a programmer is typing	Often depend on surrounding code and workflow
AI chat-based assistants	Generate and explain code through conversation	May require repeated prompting and manual organisation
NextIDE	Converts written algorithm logic into structured code and supports execution	Output quality depends on prompt clarity and model reliability

AI-generated code does not guarantee correctness. Programs may contain syntax errors, incorrect calculations, unsuitable input handling, or logic that misses edge cases, and ambiguous instructions (for example, whether a discount applies before or after a service charge) can produce different implementations. Successful execution also does not prove that an algorithm is correct for all inputs, so test cases and user review remain essential.

The reviewed work establishes that language models can generate code from natural-language descriptions. A practical educational tool must additionally consider usability, language coverage, code execution, debugging assistance, and

organisation of earlier work. NextIDE is presented as a software prototype addressing these requirements together; claims about its code-generation accuracy would require independent evaluation on labelled problem sets.

III. SYSTEM OVERVIEW AND REQUIREMENTS

NextIDE is a web-based platform in which users describe an algorithm in natural English, select a target language, and receive generated code with supporting information. The platform provides natural-language algorithm input, code generation in seven languages, explanation and complexity information, test-case generation, execution with standard input and output, AI-assisted error resolution, project and generation history, and user authentication with saved preferences.

The functional requirements are summarised below.

ID	Requirement	Description
FR-01	User authentication	Allow users to access the application through the authentication system
FR-02	Algorithm input	Accept an algorithm description in natural language
FR-03	Language selection	Allow selection of a supported programming language
FR-04	Code generation	Generate source code using the Gemini model
FR-05	Explanation	Display the explanation and complexity information
FR-06	Test-case generation	Generate test cases for the selected problem
FR-07	Code execution	Submit code and input to Judge0 CE and display the result
FR-08	Error resolution	Provide suggested corrections from code and error information
FR-09	Project management	Save and retrieve projects and generated code
FR-10	History management	Preserve earlier generations and execution results

The non-functional design goals are usability for students, reasonable responsiveness subject to external service times, handling of external-service errors, server-side validation and authentication, maintainability through modular code and TypeScript types, browser compatibility, and extensibility to further languages. These are design goals; formal performance, security, and compatibility benchmarks would be needed to show that each has been achieved. This paper does not include a supported browser/device matrix or formal acceptance criteria; these are identified as future work.

IV. MATERIALS AND METHODS

4.1 Study design

This work is a software development and implementation report. It describes the interface, backend, and integrations of the prototype. It is not a controlled study of code-generation accuracy or a user study.

4.2 Architecture

The application follows a web-client/API/service structure. The frontend, built with React and Next.js and styled with Tailwind CSS, handles user interaction. Next.js API routes process requests on the server, call Gemini 2.5 Flash for generation, test-case creation, and error resolution, and submit programs to Judge0 CE for execution. MongoDB, accessed through Mongoose, stores user and project data, and NextAuth manages authentication. TypeScript types define requests, responses, projects, and execution results. The database is accessed by the application server as required; AI and execution results do not all pass through it.

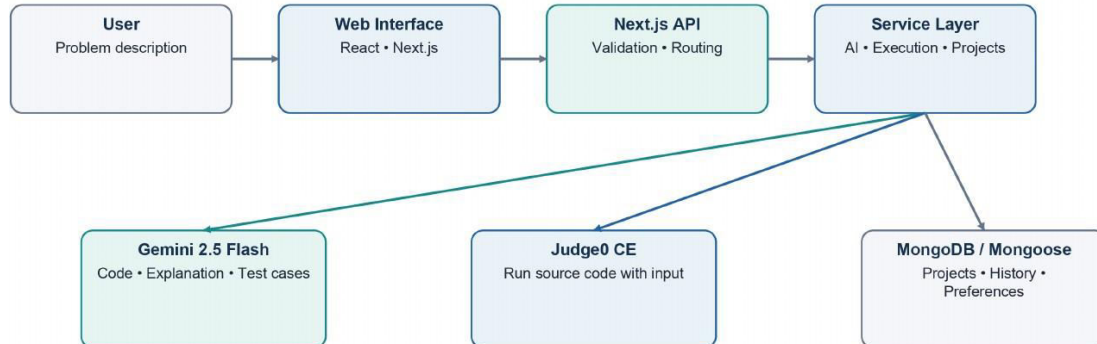


Figure.1. High-level architecture of NextIDE

The code base is organised into an app directory (pages and API routes), components, a lib directory separating AI, authentication, database, execution, utility, and validation logic, and a types directory (ai.ts, execution.ts, project.ts).

4.3 Software modules

User interface. Users enter algorithm descriptions, select a language, view generated code, supply program input, and inspect execution results.

Authentication. NextAuth associates saved projects and history with the correct user. Protected operations should verify the session and authorisation on the server so that a user cannot reach another user's project by changing an identifier.

AI code generation. The module builds a prompt from the description and selected language, requests a structured response from Gemini 2.5 Flash, validates it, and returns it for display and storage. The response (type `GenerateCodeResponse`) contains code, explanation, timeComplexity, spaceComplexity, assumptions, and edgeCases.

Code execution. The module submits source code and optional standard input to Judge0 CE after mapping the selected language to the service's language configuration. The `ExecuteResponse` includes standard output, standard error, exit code, optional execution time, and a status of success, error, timeout, or pending. A `RunnerConfig` type defines timeout and memory limits, whose enforcement depends on the external service.

Error resolution. Given the code, language, and error output, the module asks the model for a correction. The `ResolveErrorResponse` contains `fixedCode`, `updatedLogic`, `explanation`, `changesSummary`, and `logicChanged`. Suggestions must be reviewed and re-run before acceptance.

Test-case generation. The module requests test cases, each with input, `expectedOutput`, and description. Because both the code and the expected outputs may be AI-generated, independently verified examples are valuable.

Project management. A project stores its title, algorithm description (`englishLogic`), selected language, tags, status, generated code versions (including explanation, complexity, test cases, version number, acceptance status, and last execution result), suggested fixes, and timestamps.

4.4 Workflow and configuration

The general workflow is: the user signs in; enters algorithm logic; selects a language; the frontend submits the request to a Next.js API route; the server validates it and calls Gemini 2.5 Flash; the response is processed and displayed; the user optionally submits the code to Judge0 CE; and the result is displayed and, where configured, saved in project history. Test-case generation and error resolution follow the same pattern with different request data.

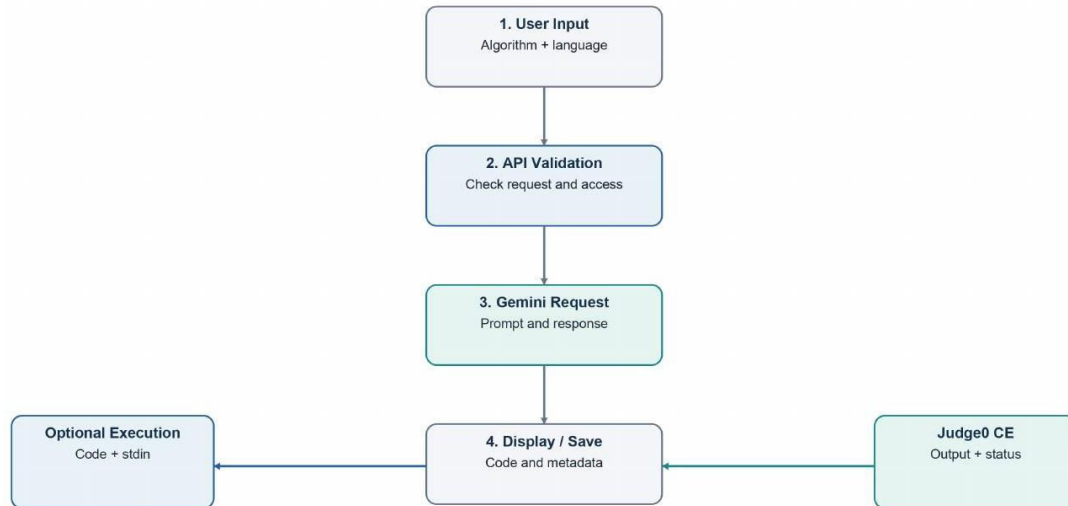


Figure.2. Data flow for code generation and execution

Supported language identifiers are java, python, javascript, typescript, cpp, c, and go. Because application identifiers can differ from those expected by Judge0, correct language mapping is required for execution. Secrets such as the database connection string, Gemini API key, and authentication secret are held in environment variables and are not exposed in client-side code. The exact environment variable names, route paths, HTTP methods, payload schemas, and software versions should be confirmed from the implementation and the execution environment.

4.5 Testing and outcome measures

Testing used representative programming problems with manually prepared inputs. The procedure was to enter the description, select a language, generate the code, review the code and explanation, provide input, execute through the integrated service, and compare the observed output with the expected result. This combines manual review with execution-based functional testing and is not exhaustive.

Three problems were used:

Test ID	Scenario	Sample input	Expected result
TC-01	Restaurant bill splitter: total dish price plus 5% service charge, divided equally	Dishes 100 and 200; 3 people	Total 315; 105 per person
TC-02	Shopping discount calculator: 10% discount if total exceeds 1000	Items 600 and 500	Total 1100; final bill 990
TC-03	Fuel cost calculator: fuel required and total fuel cost	150 km; 15 km/litre; 100 per litre	10 litres; cost 1000

The expected values are independently calculated for illustrative inputs. They are not measured outputs from a recorded execution. Additional boundary checks were identified: a total of exactly 1000 in TC-02 (discount should not apply) and a mileage of zero in TC-03 (division by zero must be handled).

V. RESULTS

5.1 Implemented software

NextIDE was implemented as a web-based algorithm-to-code platform. The application provides natural-language input, selection among seven languages, structured generation output (code, explanation, complexity, assumptions, edge cases), test-case generation, execution through Judge0 CE, AI-assisted error resolution, authentication, and persistent project history. These observations describe implemented functionality and do not establish the correctness of generated code.

5.2 Representative system outputs

The three problems in Section 4.5 cover different logic types: arithmetic and equal division (TC-01), arithmetic with a conditional statement (TC-02), and division and multiplication (TC-03). They illustrate the kinds of descriptions that can be supplied to the platform.

5.3 Functional testing

The main application functions can be evaluated with the following checklist.

Test ID	Function	Expected behaviour
FT-01	Language selection	The selected language is used for generation
FT-02	Code generation	A structured response is returned
FT-03	Explanation display	Explanation and complexity fields are displayed
FT-04	Code execution	The program is submitted and a result is returned
FT-05	Invalid program	Error output or unsuccessful status is displayed
FT-06	Test-case generation	Test cases include input, expected output, and description
Test ID	Function	Expected behaviour
FT-07	Error resolution	A suggested correction and explanation are returned
FT-08	Project persistence	Saved project information can be retrieved
FT-09	Authentication	Protected operations require valid authentication
FT-10	Execution timeout	A timed-out execution is represented appropriately

Complete test-case records, including the number of tests executed and their pass/fail outcomes, were not available at the time of writing. Therefore, an overall functional pass rate is not reported.

5.4 Generation evaluation

The present project did not measure code-generation accuracy, generation latency, execution time, or success rate across the seven languages. These metrics are therefore not reported.

VI. DISCUSSION

NextIDE is presented as an integrated software prototype for algorithm-to-code assistance rather than as a validated code-generation system. Its implementation combines a structured AI generation module, an external execution service, a database, and authentication in a single workflow, which the literature identifies as a practical need beyond raw code generation [1]–[3].

The distinction between software implementation and model evaluation is important. Published benchmarks evaluate specific models on defined problem sets, and their results cannot be attributed to NextIDE, which relies on an external model (Gemini 2.5 Flash) through its API [4]. Its output quality depends on prompt clarity and model reliability.

Several design choices support responsible use. Structured output exposes assumptions and edge cases for review. Integrated execution encourages users to validate programs instead of trusting the AI response alone. Error-resolution suggestions are explained and must be re-run before acceptance. Test cases should be checked against the problem statement because both the implementation and the expected outputs may be AI-generated. Security practice includes server-side session validation, project-level authorisation, keeping secrets out of client code, and relying on Judge0's isolation and resource limits instead of running submitted code on the application server [6].

An educational concern is that generated code may let users obtain a solution without understanding the algorithm. The explanation and complexity output are intended to counter this, but the effect on learning has not been studied.

VII. LIMITATIONS

This work has several limitations. First, only a small set of representative problems is documented, and systematic test records with pass/fail results are not available. Second, no measured generation latency, execution-time benchmark, or statistically established success rate exists. Third, comprehensive testing across all seven supported languages is not documented. Fourth, AI-generated explanations, complexity estimates, and test cases may themselves contain errors. Fifth, the system depends on the availability and limits of the Gemini and Judge0 CE services, including time and memory limits. Sixth, security, load, and concurrent-user testing have not been performed, and no user study of educational benefit was conducted. Accordingly, NextIDE should be regarded as a software prototype whose generated output requires user review.

VIII. CONCLUSION

NextIDE demonstrates the implementation of a web-based platform that converts natural-language algorithm descriptions into source code in seven programming languages and supports explanation, test-case generation, execution, error-resolution suggestions, and project history. It integrates React, Next.js, TypeScript, Tailwind CSS, Gemini 2.5 Flash, MongoDB, Mongoose, NextAuth, and Judge0 CE in a unified workflow.

The prototype demonstrates software integration; however, complete functional-testing records were not available and generation accuracy was not measured. The outputs should not be interpreted as evidence of overall correctness or performance. Future work should include testing all languages with independently verified inputs and expected outputs, boundary-value and invalid-input tests, automated comparison of actual and expected outputs, advanced debugging support, collaborative features, and evaluation of latency, failure rates, and concurrent use.

IX. DECLARATION

Ethics Approval and Consent to Participate: This study describes the development of a web-based software prototype. No human participants were recruited for testing.

Data Availability: No independent dataset was created for evaluating code-generation accuracy. Test problems are described in Section 4.5.

Software Availability: NextIDE was developed as a software prototype. A public source-code repository is not available at the time of submission.

Competing Interests: The authors declare no competing interests.

Funding: This project received no external funding.

Author Contributions: [To be completed by the authors: state each author's contribution.] All authors reviewed and approved the final manuscript.

Acknowledgements: The authors thank their project guide, Prof. M. N. Gaikwad, for his guidance and support.

REFERENCES

- [1] M. Chen et al., “Evaluating Large Language Models Trained on Code,” arXiv preprint arXiv:2107.03374, 2021. doi: 10.48550/arXiv.2107.03374.
- [2] J. Austin et al., “Program Synthesis with Large Language Models,” arXiv preprint arXiv:2108.07732, 2021. doi: 10.48550/arXiv.2108.07732.
- [3] E. Nijkamp et al., “CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis,” in Proc. International Conference on Learning Representations (ICLR), 2023.
- [4] Google, “Gemini API Documentation.” Available: <https://ai.google.dev/gemini-api/docs>
- [5] Vercel, “Next.js Documentation.” Available: <https://nextjs.org/docs>
- [6] Judge0, “Judge0 Documentation and API Reference.” Available: <https://ce.judge0.com/>
- [7] MongoDB, Inc., “MongoDB Documentation.” Available: <https://www.mongodb.com/docs/> ;
- Mongoose, “Mongoose Documentation.” Available: <https://mongoosejs.com/docs/>
- [8] Auth.js, “Auth.js Documentation.” Available: <https://authjs.dev/>



INTERNATIONAL
STANDARD
SERIAL
NUMBER
INDIA



INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH

IN SCIENCE | ENGINEERING | TECHNOLOGY

 9940 572 462  6381 907 438  ijirset@gmail.com



www.ijirset.com

Scan to save the contact details