International Journal for Innovative Engineering and Management Research
PEER REVIEWED OPEN ACCESS INTERNATIONAL JOURNAL
www.ijiemr.org

AI-Powered Code Generator with Explanation: A Full-Stack Web Application Using React, Flask, and MongoDB

G.Sreenivasa Reddy[6], P.Gayatri[1], K.Uma Maheshwari[2], K.Usha Sree[3], L.Gangotri[4], K.Sree Latha[5]

[1]UG Student, Department of Computer Science and Engineering, CBIT, Proddatur, YSR, A.P
[2]UG Student, Department of Computer Science and Engineering, CBIT, Proddatur, YSR, A.P
[3]UG Student, Department of Computer Science and Engineering, CBIT, Proddatur, YSR, A.P
[4]UG Student, Department of Computer Science and Engineering, CBIT, Proddatur, YSR, A.P
[5]UG Student, Department of Computer Science and Engineering, CBIT, Proddatur, YSR, A.P
[6]Assoc.Prof, Department of Computer Science and Engineering, CBIT, Proddatur, YSR, A.P
*Corresponding Author E-mail: gsreddy.cse@cbit.edu.in

Abstract

In this paper, the design, implementation, and evaluation of an AI-Powered Code Generator with Explanation, a full-stack web application, which uses the Google Gemini API to convert natural language descriptions into syntactically correct program code with detailed explanations are presented. The architecture of the system is based on a three-tier architecture that includes React.js front end, Python Flask RESTful, and MongoDB Atlas as the persistence environment. One of the differentiating factors of this platform is that it utilizes free of charge cloud services only, thus removing economic obstacles among students and individual programmers. The application is compatible with code generation in various programming languages and has features such as a cloud-based code execution sandbox based on the Piston API, which supports thirteen programming languages, conversational code refinement to make repeated improvements, code sharing via GitHub Gist, and an interactive monaco editor to interact with code. The security is implemented by JWT-based authentication, hash passwords (bcrypt), API keys protection at the server, and language-specific pattern of danger in order to run in the sandbox. The experimental assessment also show that the system can effectively produce functional code to different programs with satisfactory explanations, with predictable response times and quality output. The platform is an educational resource that can be easily used by learning AI-inspired code generation and programming understanding to the democratization of smart development support. The keywords that will be used are AI Code Generation; Natural Language Processing; Google Gemini API; Full-Stack Web Application; Code Execution Sandbox; React.js; Flask.

1. Introduction

Software development is an area that has witnessed a major revolution with the introduction of artificial intelligence-based assistance tools. Large language models have proven to be exceptional in reading natural language descriptions and producing the corresponding code that can be run in a program and therefore alleviate cognitive load on the developer and speed up the process of software development. According to industry reports, a significant part of the developer community has started to use AI-assisted programming tools, and organizations have reported that developers are more productive when these tools are incorporated into the established workflows. Despite these developments, there are various obstacles to the ubiquity of AI-based code-generation tools, especially among learners and other individual developers. GitHub Copilot and other commercial services like proprietary API services charge subscription fees that are prohibitive to users with financial constraints. Open-source versions using locally deployed models require a considerable amount of computational resources, such as high-performance

GPUs and large memory allocations, which many potential beneficiaries can afford. Moreover, many of the existing tools are only concerned with the code generation and do not present any explanations, thus restricting their educational potential and the possibilities to acquire the skill. This paper provides a solution to these difficulties and proposes an AI-Powered Code Generator with Explanation, a web application of full-stack written that is solely defined by a free-tier cloud service to provide accessible, explanation-based programming help. It combines Google Gemini API to generate AI-based code, the Piston API to run code in the cloud using multiple languages, MongoDB Atlas as the data storage, and a more recent React.js front-end with Python Flask as the back-end. The major contributions to this work are: (a) a zero-cost architecture that illustrates the feasibility of a multitude of unpaid utilization of various free-tier cloud services in the development of educational tools; (b) explanation-first programming style that ensures each code segment produced has an extensive pedagogical text alongside it; (c) a cloud-based execution sandbox that supports the use of thirteen programming languages with no incentive to install a local compiler; and (d) conversational development that allows refinement of code through the use of natural language dialogue. The rest of this paper will be structured in the following way. Section 2 examines the associated literature and compares the proposed system to the already existing methods. Section 3 describes the methodology, system architecture and module design. Section 4 gives results and discussion with evaluation of system. Section 5 will provide a conclusion with the contribution and future work directions.

2. Literature Review

2.1 Existing Systems

Code generation with the assistance of AI has been rapidly evolving since the release of transformer-based language models. Chen et al. (2021) describe Codex, the model behind GitHub Copilot, which showed that large language models trained on code repositories would be capable of writing functionally correct programs out of the docstring. Their performance with the HumanEval benchmark formed a baseline of the measure of code generation quality. Nevertheless, Copilot is a commercial subscription product, which restricts the availability of the product to cost-sensitive users. Code generation has been widely researched in OpenAI ChatGPT and its versions. Surameery and Shakor (2023) examined the ability of ChatGPT to produce code in a variety of programming languages and found that the model is capable of producing syntactically correct code when given a well-specified prompt, although it at times produces logically incorrect code when given more complex algorithmic tasks. Their data results emphasize the need of verification systems in AI-based code tools. Another commercial offering is Amazon code whisperer involving the IDE integration. Buscemi (2023) did a comparison between CodeWhisperer and Copilot with a similar performance compared to standard benchmarks and differences in language-specific advantages. Both systems, though, do not have inbuilt execution environments and explanatory features. The article by Kazemitabaar et al. (2023) investigated the effect of AI code generator tools on the performance of novice programmers and established that these tools positively impact the rate of task completion but reduced the learning outcome when students used the generated code extensively without any idea of the logic behind it. This observation highlights the need to have explanation enriched code generation systems that not only emphasise upon pedagogical value but

also functionality. Published as an heir to PaLM and Bard, Google Gemini is a recent development of multimodal AI functionality. Anil et al. (2023) presented the Gemini architecture and training methodology and showed that it performs well in reasoning, mathematics, and code generation tasks. Gemini is especially applicable to teaching with the free API tier available.

2.2 Proposed System

The proposed system stands out of the existing methods due to a number of underlying design choices. The proposed platform will be based on free-tier services, which is why it can be accessed by everyone as opposed to commercial tools, which have to be subscribed to. Contrary to the IDE-integrated solutions, which are oriented to the needs of professional developers, this system is educational oriented, offering detailed descriptions and created code to make it easier to learn.

Table 1 compares the proposed system with the solutions which are in existence.

Table 1. Comparative analysis of AI-powered code generation tools

Feature	Proposed System	GitHub Copilot	ChatGPT	CodeWhisperer	Replit AI	Tabnine
Cost	Free	Paid	Freemium	Free Tier	Freemium	Freemium
Code Explanation	Yes	No	Yes	No	Limited	No
Execution Sandbox	Yes (13 langs)	No	Limited	No	Yes	No
Conversational Refinement	Yes	No	Yes	No	Limited	No
History Tracking	Yes	No	Yes	No	Yes	No
GitHub Integration	Yes (Gist)	Yes	No	No	Yes	No
Self-Hosted Option	Yes	No	No	No	No	No
API Key Required	Server-side only	N/A	Yes	AWS account	Account	Account

Execution sandbox The use of a Piston API-based execution sandbox is a significant step forward in comparison to conversation-based AI tools that do not execute code. The suggested system offers a full-fledged learning experience by allowing users to create, interpret and test code in the same interface, which will minimize context switching and shorten the feedback loop necessary to support effective programming education.

3. Methodology

3.1 System Architecture

The system has a layered architecture pattern, which isolates issues in three different levels: the presentation layer, the application layer, and the data layer. This building design fosters modularity, maintainability and independent scalability of individual layers. Figure 1 is used to show the entire system architecture.

International Journal for Innovative Engineering and Management Research
PEER REVIEWED OPEN ACCESS INTERNATIONAL JOURNAL
www.ijiemr.org

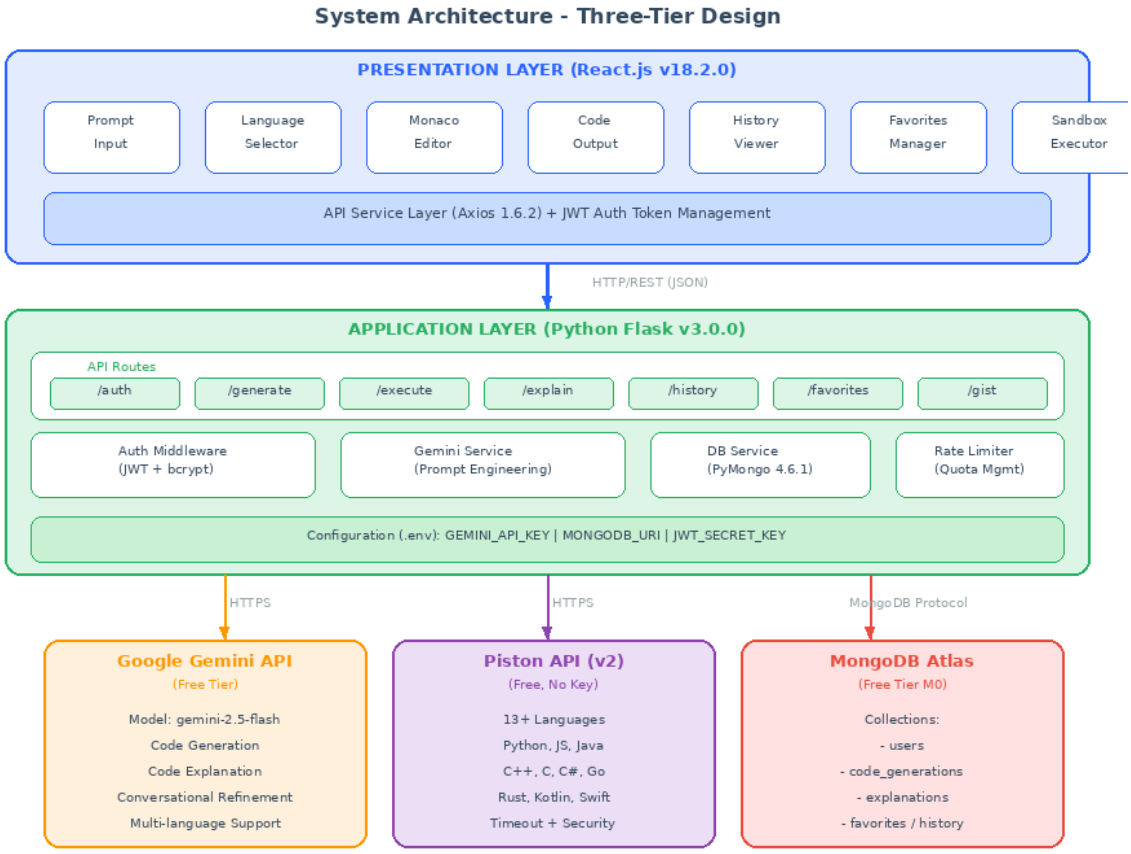

Fig. 1. Three-tier system architecture showing presentation, application, and data layers with external service integrations

The presentation layer is done using a single-page application based on React.js version 18.2.0 and using component-based architecture where the state is managed using Context API. This is the same editor found in Visual Studio Code, but it is called Monaco Editor, which offers a professional syntax highlighting, auto-completion, and theme support experience. The frontend API service layer deals with the use of HTTP using the Axios library with the injection of the JWT token automatically using request interceptors. The python flask 3.0.0 application layer is the developed one which is a lightweight but powerful server that provides RESTful API. This layer applies business logic and authentication middleware, prompt engineering of the Gemini API, response parsing and database operations. The route-based architecture is modular in that it separates concerns into authentication, code generation, code execution, explanation, history, favorites and endpoints of the GitHub Gist integration. It applies the MongoDB Atlas, which is a cloud-based document database that maintains user credentials, code generation history, explanations, favorites, and a history of activities. Document-oriented storage model supports the changeable length of generated code and descriptions without a strict schema definition.

3.2 Module Description

The system has eight functional modules that are tasked with taking care of a specific area of the application capabilities. The relationships between modules and the module organization are provided in Figure 2.

International Journal for Innovative Engineering and Management Research
PEER REVIEWED OPEN ACCESS INTERNATIONAL JOURNAL
www.ijiemr.org

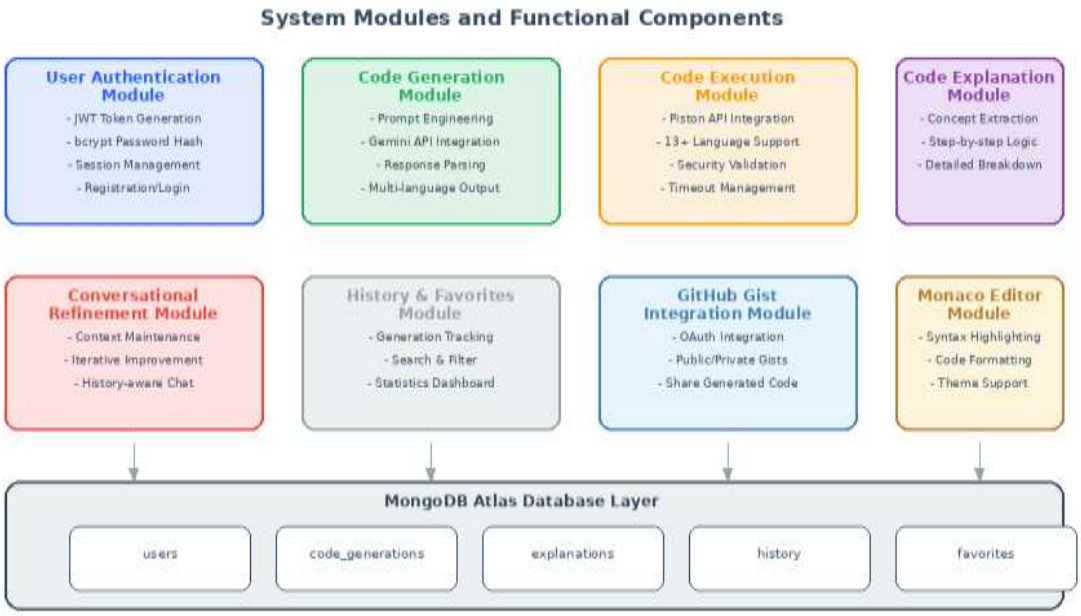

Fig. 2. System modules and functional components with database layer interconnections

User Authentication Module.

This module is the heart of the system and does the conversion of natural language prompts to structured code with explanations. The module has introduced prompt engineering methods which train the Gemini API with a particular format that contains a block of code and an explanation part. These segments are identified by the response parser which matches patterns on markdown code fences and section headers. The module can generate code in several programming languages such as Python, JavaScript, TypeScript, Java, C++, C, C#, Ruby, Go, PHP, Swift, Kotlin and Rust.

Code Generation Module.

The core module of the system, responsible for transforming natural language prompts into structured code with explanations. The module implements prompt engineering techniques that instruct the Gemini API to produce output in a specific format containing both a code block and an explanation section. The response parser extracts these segments using pattern matching on markdown code fences and section headers. The module supports code generation in multiple programming languages including Python, JavaScript, TypeScript, Java, C++, C, C#, Ruby, Go, PHP, Swift, Kotlin, and Rust.

Code Execution Sandbox Module.

The module incorporates the Piston API to make code execution available through the cloud in thirteen programming languages. The module does language specific security validation before execution to identify and block potentially harmful operations like file system access, network calls, and system command execution. Every execution request has a configurable time out limit of ten to twenty seconds by language, and memory limits imposed by the Piston API infrastructure. In-person Finishing School. Allowing the process of improvement of the code, this module also preserves the context of conversation in a series of refinements. Users are allowed to make changes to previously generated code using natural language instructions and the module

builds context-aware prompts including the original code, explanation and history of conversations. This is similar to the interactive process of development of code where a code is refined over successive development cycles.

3.3 Request Processing Flow

The code generation process is clearly defined with a sequence of operations that the user input is converted to the rendered output. The overall request flow through all the components of the system is shown in Figure 3.

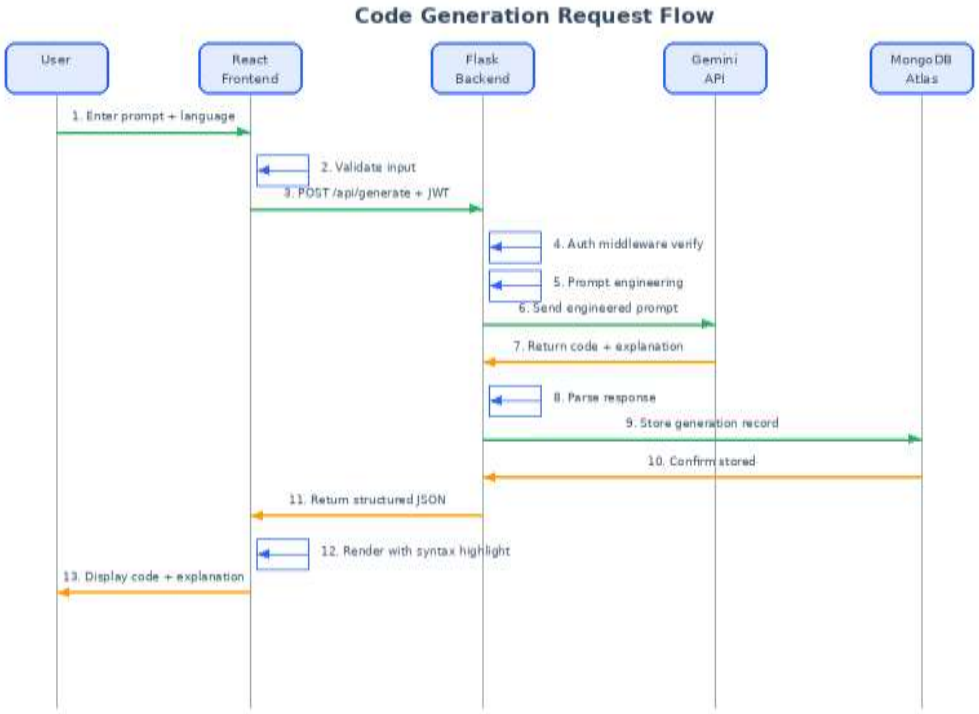

Fig. 3. Code generation request flow showing the sequence of operations across system components

It starts with the user entering a natural language prompt which specifies what desired code functionality they want and then choosing a target programming language. React frontend will verify the input as per the minimum and character-based restrictions and then build a HTTP POST request to the Flask servers. The backend authentication middleware will be involved in extracting the JWT token in the Authorization header and verifying it, and in the event of successful verification, the request handler will call the Gemini Service module. The Gemini Service develops a designed prompt that will include the description of the user, the target language, and the formatting instructions that will be used to teach the AI model to generate formatted data with the code and the explanation sections. This prompt is sent to the Google Gemini API, where the answer is then sent back and the components of code and explanation are then extracted. The record of interaction is stored into MongoDB Atlas and formatted response sent back to the frontend to be rendered in syntax highlighting and formatted explanation format.

3.4 Database Design

The MongoDB database uses a document-based schema that is well capable of the dynamism of the generated content. Figure 4 is the entity relationship diagram that depicts the key collections and the associations between them.

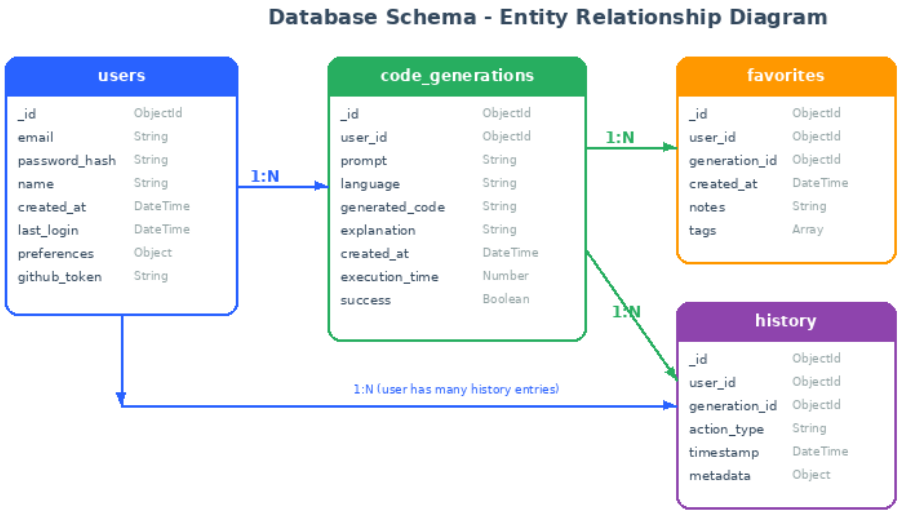

Fig. 4. Database schema entity relationship diagram showing collection structures and associations

Table 2. Technology stack with versions and justifications

Layer	Technology	Version	Justification
Frontend	React.js	18.2.0	Component-based architecture with virtual DOM
Code Editor	Monaco Editor	4.7.0	VS Code engine with IntelliSense support
HTTP Client	Axios	1.6.2	Promise-based HTTP with interceptors
Backend	Python Flask	3.0.0	Lightweight WSGI framework, Python ecosystem
Authentication	PyJWT + bcrypt	2.8.0	Secure JWT tokens with password hashing
Database	MongoDB Atlas	M0 Free	Document flexibility, JSON-native storage
AI Service	Google Gemini	2.5-flash	Zero-cost, high-quality code generation
Code Execution	Piston API	v2	Multi-language cloud execution, no key needed
Production	Gunicorn	21.2.0	Production-grade WSGI HTTP server

The MongoDB database uses a document-based schema that is well capable of the dynamism of the generated content. Figure 4 is the entity relationship diagram that depicts the key collections and the associations between them.

3.5 Security Architecture

The security considerations are considered at various architectural levels. The API key protection is such that the Gemini API key exists in the server-side environment variables and is never sent to or made available at the frontend client. Authentication uses JSON Web Tokens and has expiration policies and an adaptive cost factor password hashing (bcrypt) implementation. The backend server will implement input sanitization of any user input it receives to avoid injection attacks. The code execution sandbox provides language-adapted identification of dangerous patterns, which detects and blocks code that includes file system operations, network access

operations, creation of new processes, and other potentially harmful operations and then forwards the remaining code to the Piston API.

4. Results and Discussion

4.1 System Implementation

The suggested system was successfully implemented and launched as a workable web app. The frontend interface offers a simple and easy-to-use workflow where the user can input natural language descriptions, choose the code target program language, and get generated code with detailed explanations. Monaco Editor provides a professional coding experience, syntax highlighting in all supported languages, bracket matching and customizable themes. The system was tested on various fronts such as functional correctness, response time performance, multi-language coverage support and responsiveness of user interface. Testing was done on a wide variety of program tasks that included algorithm problems, data structure implementation problems, web development code snippets and utility functions generation.

4.2 Functional Evaluation

Table 3 shows the outcomes of the functional testing in various types of code generation tasks. All the categories were rated with ten different prompts and the resultant code was analyzed based on the syntactic correctness, logical accuracy and quality of explanation.

Table 3. Functional evaluation results across programming task categories

Task Category	Prompts Tested	Syntax Correct	Logic Correct	Explanation Quality	Avg. Response (s)
Sorting Algorithms	10	10 (100%)	9 (90%)	High	2.3
Data Structures	10	10 (100%)	8 (80%)	High	2.8
String Manipulation	10	10 (100%)	10 (100%)	High	1.9
File I/O Operations	10	9 (90%)	8 (80%)	Medium	2.5
API Integration	10	9 (90%)	8 (80%)	Medium	3.1
Web Development	10	10 (100%)	9 (90%)	High	3.4
Database Queries	10	10 (100%)	9 (90%)	High	2.7
Mathematical Functions	10	10 (100%)	10 (100%)	High	2.1
Overall	**80**	**78 (97.5%)**	**71 (88.75%)**	**High**	**2.6**

The performance of the system in the evaluation indicates that it has a syntactic and logical correctness rate of 97.5 and 88.75 respectively in eighty test prompts in eight categories of tasks. String manipulation and generating mathematical functions tasks had the highest accuracy which is in line with the well defined nature of these problem domains. Activities with file I/O and API integrations were a little bit less accurate, which can be explained by environmental dependencies and configuration peculiarities that change in different deployment scenarios. The present functionality of the device is examined through multi-language testing, which involves executing multiple tasks in languages other than English to evaluate the device's multi-lingual support

capability and to assess how the system, and its various language interfaces, respond to non-English language instruction within its present context.<|human|>Multi-Language testing

4.3 Multi-Language Execution Testing

The current capability of the device is tested in terms of multi-language execution and the testing is performed with the Multi-language execution consisting of carrying out several tasks in some other language other than English to find out the capability of the device to handle

Table 4. Code execution sandbox performance across supported languages

Language	Version	Tests Executed	Success Rate	Avg. Time (ms)
Python	3.10.0	15	100%	245
JavaScript	18.15.0	15	100%	180
Java	15.0.2	12	92%	890
C++	10.2.0	12	100%	320
TypeScript	5.0.3	10	100%	410
Go	1.16.2	10	100%	195
Rust	1.68.2	10	90%	680
Kotlin	1.8.20	8	87.5%	1250
C#	6.12.0	8	100%	750
Ruby	3.0.1	8	100%	210
PHP	8.2.3	8	100%	165
Swift	5.3.3	6	83%	920
C	10.2.0	6	100%	150

The execution sandbox powered by the Piston API was tested on all the thirteen languages supported. Table 4 provides a synopsis of the execution succeed rates and average execution time in the execution of a standardized test programs. The execution sandbox had proved to be stable over the range of supported language. The fastest execution times were seen with interpreted languages like Python, JavaScript, and PHP whereas compiled languages like Java and Kotlin displayed a greater latency with a compilation overhead. The reduced success rates of Swift and Kotlin are due to the increased requirements of type system and the limitations on runtime configuration on the sandboxed environment.

4.4 Discussion

The experimental outcomes confirm the ability to create an AI-based code generation app with the help of purely free-tier cloud services to create a production-quality one. Zero-cost architecture has managed to introduce zero cost without compromising on the performance aspect that is acceptable to use in education. The 2.6 mean of code generation time with explanation is within reasonable values of interactive web application, yet it is significantly slower than solutions that are embedded in an IDE and can take advantage of pre-built model inference. The quality of explanation with a high rate in most of the categories of the tasks proves the fact that timely engineering methods can effectively direct the Gemini API to generate pedagogically useful

information with functional code. This observation would favor the educational design philosophy of the system, in which generation places importance on understanding. One restriction of the implementation that exists is that it relies on external API availability and rate limits. The free tier of the Gemini API has a per-minute and per-day request limit which limits throughput when used in a concurrent multi-user environment. Likewise, there may be fluctuating execution time of the shared infrastructure of the Piston API, during high demand intervals. Caching techniques, request queuing, and fallback can be used to solve these limitations in future versions.

5. Conclusion

This paper showed the design, implementation and evaluation of a full-stack web application, AI-Powered Code Generator with Explanation, which democratizes access to AI-assisted programming tools with a zero-cost architecture. The system manages to combine the Google Gemini API of intelligent code generation and explanations, the Piston API of multi-language cloud-based code execution, MongoDB Atlas of persistent data storage, and a frontend built with a modern React.js with a Python Flask backend. The results of evaluation prove that the system acquires a correctness rate of 97.5% in syntactic correctness and a correctness rate of 88.75 in logical correctness in the different categories of programming tasks, and the quality of explanations is always very high. The execution sandbox is based on the cloud and it can reliably execute thirteen programming languages without installing the local compilers. The conversational refinement endows the iterative enhancement of the code with the help of the natural language dialogue that is very similar to the professional development processes. The main contributions of this work are the establishment that production-quality AI-assisted development tools can be built completely out of the free-tier of cloud services, the confirmation of an explanation-first strategy to code generation that can improve the learning experience, and the merging of code generation, execution and explanation into a single platform. This set of contributions leads to efforts of realizing the vision of providing intelligent programming assistance to all and sundry, independent of financial and computational resources. Future work will be on integration of other AI models to perform comparative code generation, implementation of collaborative functionality to a team based learning environment, then on automated test case generation of a generated code and then on extending the execution sandbox to other programming languages and frameworks.

Author Contributions

P. Gayatri: Led the project; system design, AI code generation, conversational refinement and execution modules, and manuscript preparation. K. Uma Maheshwari: Frontend development and editor integration. K. Usha Sree: Backend API development. L. Gangotri: Database management and history management. K. Sree Latha: Testing and security support. All authors reviewed and approved the final manuscript.

Conflicts of Interest

The authors declare no conflicts of interest.

References

Anil, R., Dai, A. M., Firat, O., Johnson, M., Lepikhin, D., Passos, A., ... & Wu, Y. (2023). Gemini: A family of highly capable multimodal models. arXiv preprint arXiv:2312.11805.

Buscemi, A. (2023). A comparative study of code generation using ChatGPT 3.5 across 10 programming languages. arXiv preprint arXiv:2308.04477.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., ... & Zaremba, W. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.

Dakhel, A. M., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M. C., & Jiang, Z. M. J. (2023). GitHub Copilot AI pair programmer: Asset or liability? Journal of Systems and Software, 203, 111734.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. In Findings of the Association for Computational Linguistics: EMNLP 2020 (pp. 1536-1547).

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., ... & Lewis, M. (2023). InCoder: A generative model for code infilling and synthesis. In International Conference on Learning Representations.

Kazemitabaar, M., Chow, J., Ma, C. K. T., Ericson, B. J., Weintrop, D., & Grossman, T. (2023). Studying the effect of AI code generators on supporting novice learners in introductory programming. In Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (pp. 1-23).

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., ... & Vinyals, O. (2022). Competition-level code generation with AlphaCode. Science, 378(6624), 1092-1097.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., ... & Xiong, C. (2023). CodeGen: An open large language model for code with multi-turn program synthesis. In International Conference on Learning Representations.

Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., ... & Synnaeve, G. (2023). Code Llama: Open foundation models for code. arXiv preprint arXiv:2308.12950.

Surameery, N. M. S., & Shakor, M. Y. (2023). Use ChatGPT to solve programming bugs. International Journal of Information Technology and Computer Engineering, 3(1), 17-22.

Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In CHI Conference on Human Factors in Computing Systems Extended Abstracts (pp. 1-7).

Wang, Y., Wang, W., Joty, S., & Hoi, S. C. (2021). CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (pp. 8696-8708).

Xu, F. F., Alon, U., Neubig, G., & Hellendoorn, V. J. (2022). A systematic evaluation of large language models of code. In Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (pp. 1-10).

Ziegler, A., Kalliamvakou, E., Li, X. A., Rice, A., Rifkin, D., Simister, S., ... & Aftandilian, E. (2022). Productivity assessment of neural code completion. In Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (pp. 21-29).