WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal Homepage: www.ijrpr.com ISSN 2582-7421

Bug-Zero AI-Powered Code Bug Finder and Fix Suggestion Platform

R. Nithin, R. Navin Chandra, S. Krishna Prasath, Dr. S. Prakash*

Student, *Project Guide:
Department of Information Technology, Sri Shakthi Institute of Engineering and Technology, Coimbatore, Tamil Nadu, India
Email ID: nithin54267@gmail.com

ABSTRACT

Finding bugs in source code is a slow task, especially for students and beginners, because compiler messages are often cryptic and traditional linters work for only one language or follow fixed rules. This project presents BUG-ZERO, a web-based application that uses a Large Language Model (LLM) to find bugs in source code written in any programming language. The application combines an HTML, CSS and JavaScript frontend with a Python Flask backend and an SQL database. Users register and log in, paste or upload code into an editor with syntax highlighting, and receive a structured report that lists each bug with its line number, type, severity, a plain-language explanation, and a suggested fix, together with a complete corrected version of the code. An optional problem statement can be supplied; in that case BUG-ZERO lists each requirement, marks it as met or not met, and reports every mismatch as a bug. The backend uses password hashing and JWT-based sessions, sends the code to the Google Gemini API through a structured prompt, retries when the model is busy, and saves every report so that users can search, reopen, and delete their history. The supplied test script runs 47 automated checks with a simulated model response, and all 47 pass. No live-model accuracy measurement is included in the project files, so the quality of the detected bugs depends on the LLM and should be reviewed by the user. BUG-ZERO is intended as a learning and debugging aid, not as a replacement for code review, testing, or compilation.

Keywords: Bug Detection, BUG-ZERO, Large Language Model, Google Gemini, Flask, JWT Authentication, SQL Database, Code Analysis, Web Application.

1. INTRODUCTION

Software bugs are among the most common and costly problems in software development. Developers, and especially students and beginners, spend a large share of their time reading code, running it repeatedly, and searching the internet to find the cause of an error. Some bugs, such as logical errors, security flaws, and poor coding practices, do not produce any error message at all and are easily missed. A tool that can examine code, explain the problem in simple language, and suggest a correction can therefore save time and help users learn from their mistakes.

Recent progress in Large Language Models (LLMs) has made it possible for a single model to read and reason about source code in many programming languages. Instead of matching fixed patterns, an LLM can use the surrounding context of the code and, when it is given a task description, the intent of the program. Web technologies can make such a model accessible through a browser. In this project, BUG-ZERO combines an LLM-based bug analysis feature, secure user accounts, and a saved history in one web application.

1.1 Existing challenges

Most linters and debuggers are language-specific, for example Pylint for Python or ESLint for JavaScript, so a developer working in several languages needs several tools. Rule-based static analysers detect only predefined patterns and cannot understand the intent of the code. Compiler errors are often cryptic and do not explain why a bug occurred or how to fix it. Many free tools also provide no user accounts, so users cannot save, revisit, or track the bugs they have found over time. An LLM can also make mistakes or report a bug that does not exist. Therefore, a responsible system should present its output as suggestions, avoid executing submitted code, validate the input, and encourage the user to verify every fix.

1.2 Role of large language models and web technologies

A Large Language Model is a neural network trained on large collections of text and code that can produce a structured answer from a written instruction. In the project, the Google Gemini API is called through its REST interface with a system prompt that defines the role of the reviewer, the rules for reporting bugs, and the exact JSON format of the reply. Flask handles backend requests, authentication, and database access. PyJWT creates session tokens and werkzeug hashes passwords. The frontend is built with HTML, CSS and JavaScript, and CodeMirror provides the code editor with line numbers and syntax highlighting. The frontend communicates with the backend through API calls for registration, login, analysis, and history.

1.3 Problem statement and objectives

The project addresses the need for a single, easy-to-use platform that can analyse code in any programming language, identify bugs, explain them in plain language, and suggest corrected code, while keeping a personal history for each user. The system is not designed to replace compilers, test suites, or human code review. Its output is produced by an LLM, and its quality has not been measured against a benchmark in the supplied project files.

The objectives of BUG-ZERO are:

i. To build a bug detection system that supports multiple programming languages through an LLM API.

ii. To provide a secure registration and login system using password hashing and JWT sessions.

iii. To design an interactive interface with a code editor, syntax highlighting, file upload, and a clear results panel.

iv. To report each bug with its line number, type, severity, plain-language explanation, and suggested fix.

v. To return a complete corrected version of the submitted code.

vi. To check code against an optional problem statement and report each requirement as met or not met.

vii. To store users, submitted code, and bug reports in an SQL database with search and delete options.

viii. To validate inputs, handle busy or failing model responses, and verify the system with an automated test script.

2. MATERIALS AND METHODS

2.1 Development approach

The project was developed through problem definition, requirement identification, database design, backend implementation, frontend development, integration, and testing. The backend was implemented in Python using Flask and is divided into routes, services, middleware, models, and utility modules. The frontend was developed with HTML, CSS and JavaScript and contains separate login, registration, editor, and history pages. The final application connects these components through HTTP API requests. Flask also serves the frontend files, so the whole application runs from a single command.

2.2 System architecture

BUG-ZERO follows a client-server architecture. The browser frontend provides the user interface and sends requests to the Flask backend. For code analysis, the backend verifies the JWT token, checks that the code and the optional problem statement are within the size limits, sends them to the LLM service, saves the returned report in the SQL database, and returns the report to the frontend as JSON. For history requests, the backend reads or deletes only the reports that belong to the logged-in user. The frontend displays the returned bug list, requirement checks, and corrected code.

Layer / component	**Responsibility**
Frontend (HTML, CSS, JavaScript)	Provides the login, registration, editor, and history pages, file upload, and the results panel.
API helper (api.js)	Stores the JWT token, sends requests to the /api endpoints with the Authorization header, and handles errors.
Editor module (CodeMirror)	Provides line numbers and syntax highlighting for 15 languages, file upload, and highlighting of the lines that contain bugs.
Flask backend	Receives requests, validates inputs, applies authentication, calls the services, returns JSON responses, and serves the frontend files.
Authentication middleware	Creates and verifies JWT tokens (HS256, 24 hours) and protects the analysis and history routes.
LLM service	Builds the structured prompt, calls the Gemini API with retries and optional backup models, parses the JSON reply, and fills missing fields.
Database module	Creates the users and reports tables and provides functions to create users and to save, list, search, open, and delete reports.

Table 1. Responsibilities of the BUG-ZERO system components.

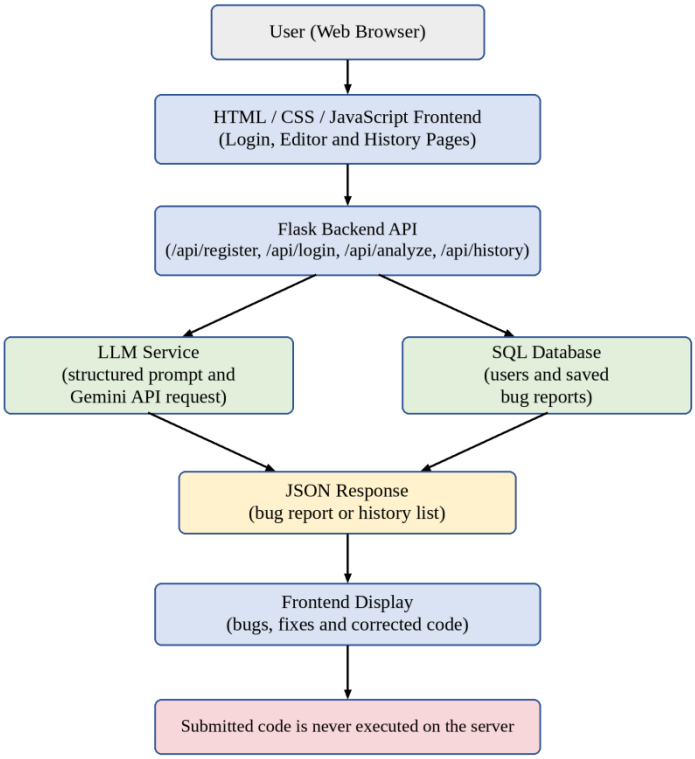


Fig. 1. System Architecture of BUG-ZERO.

2.3 Technology stack and development environment

Technology	**Use in the project**
Python	Backend implementation language.
Flask 3.0.3	Backend web framework and API routes.
Flask-CORS 4.0.1	Enables cross-origin API requests.
PyJWT 2.9.0	Creates and verifies JWT session tokens.
werkzeug	Hashes and checks user passwords (installed with Flask).
python-dotenv 1.0.1	Loads settings and secrets from the .env file.
requests	Sends HTTPS requests to the Gemini API.
SQLite / SQL	Stores users and reports (MySQL is planned for deployment).
Google Gemini API	LLM that analyses the code and returns the bug report.
HTML / CSS / JavaScript	Page structure, styling, and browser interactions.
CodeMirror 5.65.16	Code editor with line numbers and syntax highlighting, loaded from a CDN.

Table 2. Technologies used in the BUG-ZERO project.

The project README specifies that Flask serves both the frontend and the API on port 5000, so the application is opened at http://localhost:5000 after running python app.py inside the backend folder. Settings are read from a .env file that contains GEMINI_API_KEY, GEMINI_MODEL, an optional list of backup models, SECRET_KEY, and DATABASE_PATH. The .env file is excluded from Git, and only placeholders are kept in .env.example. If

the frontend is hosted separately, the API address in frontend/js/api.js must be changed. For a MySQL deployment, models/db.py must be adapted to a MySQL driver and the schema must be modified.

2.4 Supported languages and report structure

BUG-ZERO does not train its own model, so it does not use a dataset. The language-specific part of the interface is the editor configuration in editor.js, which defines syntax-highlighting modes for 15 languages and maps common file extensions to a language when a file is uploaded. The language list can also be left on automatic detection, in which case the LLM detects the language itself. Other languages can still be analysed through the LLM, although they are not highlighted in the editor. Each bug in the report has a line number, a type (syntax, logical, runtime, security, or performance), a severity (low, medium, high, or critical), a title, an explanation, and a fix. The report also contains the detected language, a summary, an overall severity, a requirements match section, and the corrected code.

No.	Language in editor	No.	Language in editor
1	Python	9	SQL
2	Java	10	HTML
3	C	11	CSS
4	C++	12	Go
5	C#	13	Rust
6	JavaScript	14	Ruby
7	TypeScript	15	Kotlin
8	PHP	16	Auto-detect (by LLM)

Table 3. Languages configured in the code editor.

2.5 Prompt design and response handling

The user message sent to the model contains the code inside <code> tags and, when provided, the problem statement inside <problem_statement> tags together with an optional language hint. The system prompt defines BUG-ZERO as an expert code reviewer, tells the model to detect the language, look for syntax, logical, runtime, security, and performance problems, and not to invent bugs. When a problem statement is present, the model must work out every requirement, check the code against each one, report each unmet requirement as a logical bug, and return corrected code that satisfies the statement. The prompt requires the reply to be only a JSON object in a fixed shape, and it states that the text inside the tags is untrusted input and never an instruction. The request uses a temperature of 0.2 and the JSON response type. The reply is parsed by a tolerant function that removes code fences and extracts the JSON object, and missing fields such as language, bugs, corrected code, overall severity, summary, and requirements match are filled with safe defaults.

2.6 LLM integration and resilience

The LLM service calls the Gemini generateContent REST endpoint using the requests library and the API key from the environment. The model name is read from the GEMINI_MODEL setting (the example configuration uses gemini-3.8-flash), and the request timeout defaults to 120 seconds. If the API answers with a temporary error (HTTP 429, 500, 502, 503, or 504), the service makes up to three attempts in total, waiting 3 seconds and then 6 seconds between attempts. A timeout is not retried and moves directly to the next model. If the main model still fails, the service tries each model listed in GEMINI_FALLBACK_MODELS in order. When all models fail, a clear error message is returned to the user with HTTP status 502.

The service does not run, compile, or evaluate the submitted code at any point; the code is only sent to the LLM API as text. The returned report is checked for structure but not for correctness, so the bugs and corrections must still be verified by the user. Each successful report is saved in the reports table together with the language, code, bugs, corrected code, overall severity, problem statement, and requirement-match result.

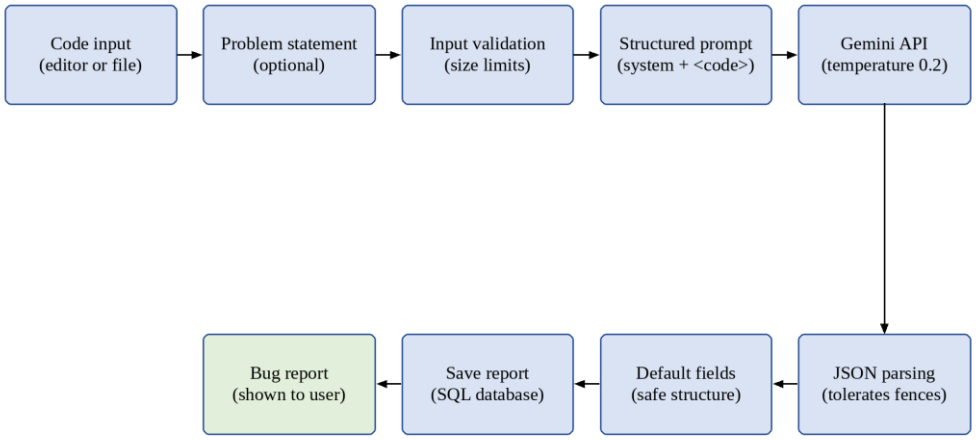


Fig. 2. Code analysis pipeline used in BUG-ZERO.

2.7 Analysis workflow

1. The user registers or logs in, and the browser stores the returned token.
2. The user pastes code into the editor or uploads a file, and optionally types a problem statement.
3. The frontend sends the code, optional language, and optional problem statement to POST /api/analyze with the token.
4. Flask verifies the token and checks that the code is not empty and is within 20,000 characters, and that the problem statement is within 5,000 characters.
5. The LLM service builds the structured prompt and sends it to the Gemini API, retrying or using a backup model when needed.
6. The service parses the JSON reply and fills any missing fields with defaults.
7. The backend saves the report in the SQL database and adds the report identifier to the response.
8. The frontend highlights the lines that contain bugs and shows the summary, requirement checks, bugs, and corrected code.
9. The user can open the history page later to search, reopen, or delete the saved report.

2.8 Authentication and history workflow

The application provides registration and login pages. During registration the backend checks that the username has at least 3 characters, the email address is valid, and the password has at least 8 characters, and it rejects a username or email that already exists. The password is stored only as a werkzeug hash. At login the user can enter either the email or the username, and the backend returns a JWT token that is valid for 24 hours. All analysis and history routes require this token as a Bearer token, and every history query includes the user identifier, so a user cannot read or delete the reports of another user. The history page lists the saved reports, supports searching by language, code, bug text, and problem statement, and allows a report to be reopened in the editor or deleted.

2.9 Backend API design

Endpoint	Method	Purpose
/api/register	POST	Creates a user from a username, email, and password and returns a JWT token.
/api/login	POST	Accepts an email or username with a password and returns a JWT token.
/api/analyze	POST	Requires a token; accepts code, an optional language, and an optional problem statement and returns the bug report.

Endpoint	Method	Purpose
/api/history	GET	Requires a token; lists the reports of the user and supports the q search parameter.
/api/history/<id>	GET	Requires a token; returns one saved report with its bugs and corrected code.
/api/history/<id>	DELETE	Requires a token; deletes one saved report of the user.
/ and /<path>	GET	Serves the frontend pages and static files from the same Flask server.

Table 4. Backend API endpoints of the BUG-ZERO application.

2.10 Input validation and safety

The backend rejects empty code, code longer than 20,000 characters, and a problem statement longer than 5,000 characters with an error response (400 or 413). Registration inputs are validated for length and email format, and duplicate users are rejected with a conflict response. Passwords are hashed and never stored as plain text, and the JWT secret is read from the SECRET_KEY setting; if it is not set, a default development value is used, so a long random key must be set before deployment. History queries use parameterised SQL statements and are limited to the logged-in user. Text returned by the model is escaped before it is inserted into the results panel, and the .env file that holds the API key is excluded from Git. These controls reduce common risks but do not by themselves establish that the application is production-secure; for example, the CORS policy currently allows all origins for the API routes, and the development server runs in debug mode. The bug report is informational and must be checked by the user.

2.11 Testing and evaluation method

The project includes an automated script, tests/check_requirements.py, that checks the application against the problem statement. The script replaces the LLM call with a fake response, so it needs no API key or internet connection, and it uses a temporary database. It checks the folder structure, registration and login, password hashing, protected routes, input limits, report content, database tables, problem-statement checking, retry and backup-model behaviour, history search and deletion, user isolation, and selected frontend features by reading the source files. The journal reports only the result of this script; no live-model accuracy, response-time, or user-study measurements are claimed.

3. RESULTS AND DISCUSSION

3.1 Implemented application features

The supplied project contains an HTML, CSS and JavaScript frontend with login, registration, editor, and history pages, a Flask backend, an LLM service for the Gemini API, authentication middleware, and an SQL database. The editor page has a language selector with automatic detection, a file upload button, a problem statement box, and a Find bugs button. The lines that contain bugs are highlighted in the editor, and the results panel shows the summary, the requirement verdict, each bug with its severity, type, line, explanation, and fix, and a Copy button for the corrected code. Fig. 3 shows the editor page after the analysis of a sample Python function. The report in the figure comes from a simulated model response and is shown only to illustrate the interface; a live analysis depends on the Gemini API. User satisfaction and the accuracy of live bug detection require separate evaluation.

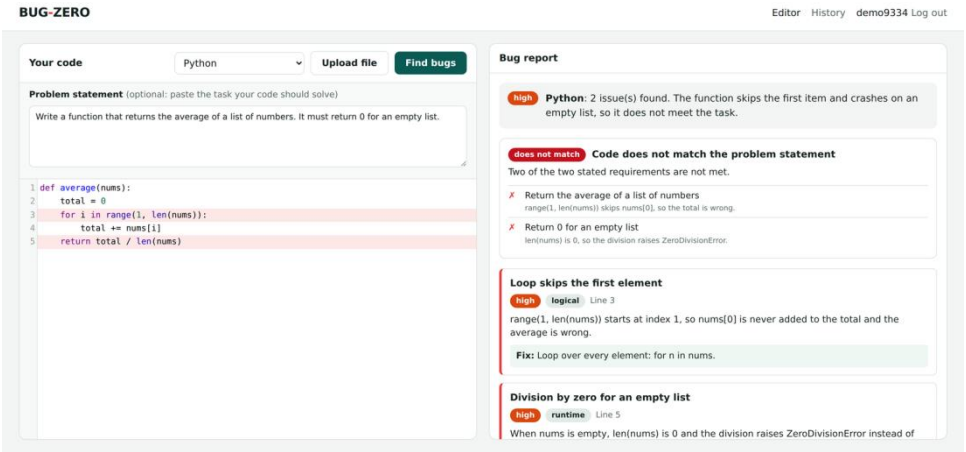


Fig. 3. Editor page of the BUG-ZERO web interface (sample analysis).

3.2 Automated requirement check

The supplied test script was executed on the project files and reports that 47 of 47 requirements are satisfied. The checks can be grouped into seven areas: project structure and configuration, authentication, code analysis, database and history, problem-statement checking, LLM service resilience, and frontend features. All groups passed. These results confirm that the implemented routes, validation, database design, and interface elements behave as specified. They use a simulated model response and therefore do not measure how accurately the LLM finds bugs.

Evaluation metric	**Reported value**
Automated checks run	47
Checks passed	47
Checks failed	0
Overall pass rate	100%
Test groups	7
Model responses in the tests	Simulated (fake LLM)
Database used in the tests	Temporary SQLite database
Live Gemini accuracy evaluation	Not included in the project files
Test script	tests/check_requirements.py

Table 5. Summary of the automated test results for the project.

3.3 Test-group observations

Test group	**Checks**	**Passed**	**Failed**	**Pass rate**
Structure and configuration	3	3	0	100%
Authentication	8	8	0	100%
Code analysis and report	9	9	0	100%
Database and history	9	9	0	100%
Problem-statement checking	7	7	0	100%
LLM service resilience	2	2	0	100%
Frontend features	9	9	0	100%
Total	47	47	0	100%

Table 6. Test results grouped by area, taken from the output of the supplied test script. The grouping of the individual checks into areas is made for this journal.

The results show that the project meets the requirements listed in its problem statement at the level of implementation. Users can register and log in, protected routes reject requests without a token, reports contain the line, type, severity, explanation, fix, and corrected code, and reports are saved and can be searched, reopened, and deleted only by their owner. The tests also show that a busy model is retried and that a backup model is used when the main model stays busy. However, because the model response is simulated, the tests cannot show that the LLM finds every bug or that its corrected code always runs. A more complete evaluation should run programs with known bugs in several languages through the live API, compare the detected bugs with the known ones, and record the number of missed and false reports and the response time.

3.4 Bug report output



Fig. 4. Bug report panel of BUG-ZERO showing the requirement check, two detected bugs, and the corrected code for the sample function. The report is a simulated model response used to illustrate the output format.

3.5 Functional test scenarios

Test ID	Module	Test scenario	Expected / observed handling
F1	Registration	Register with a valid username, email, and password	Returns status 201 and a JWT token; the password is stored hashed.
F2	Validation	Register with an invalid email, a short password, or an existing user	Returns 400 for invalid input and 409 for a duplicate user.
F3	Login	Log in with a wrong password	Returns 401; the correct password returns a token.
F4	Access control	Call /api/analyze and /api/history without a token	Returns 401 for every protected route.
F5	Analysis input	Submit empty code or code above 20,000 characters	Returns 400 for empty code and 413 for oversized code.
F6	Problem statement	Submit code together with a problem statement	Returns a verdict and requirement checks that are saved with the report.
F7	History	List, search, reopen, and delete reports as two different users	The owner can use the reports; another user gets an empty list or 404.
F8	Resilience	Model returns a busy error (503)	The request is retried; a backup model is used if the main model stays busy.

Table 7. Functional scenarios covered by the supplied test script (tests/check_requirements.py). The model response is simulated in these tests.

3.6 Discussion and limitations

BUG-ZERO demonstrates an integrated application that connects an LLM with secure user accounts, a saved history, and an interactive editor. The frontend, backend, LLM service, and database are separated into modules, so the interface, the model provider, or the database can be changed independently. The automated tests show that the application behaves as designed, but they do not measure the quality of the LLM output. An LLM may miss a bug, report a bug that does not exist, or return corrected code that still contains errors. The output must therefore be presented as a suggestion, and the user must test the corrected code.

The available project files do not include live-model test results, a comparison with existing tools, response-time measurements, usability feedback, or verified deployment results. These details should be added only after they are measured. The current limits of 20,000 characters of code and 5,000 characters of problem statement mean that large files and whole repositories cannot be analysed, and running or compiling the code, real-time collaborative editing, and automatic fixing of entire projects are outside the scope of this version. Sending code to an external API also has privacy implications, so users should not submit confidential code unless this is acceptable.

4. CONCLUSION

BUG-ZERO is a web-based AI code bug finder developed using HTML, CSS, JavaScript, Python Flask, an SQL database, and the Google Gemini API. The application allows users to register, log in, submit code in any programming language, and view each bug with its line, type, severity, explanation, and suggested fix, together with a corrected version of the code. An optional problem statement allows the application to check whether the code meets each stated requirement. The backend provides API routes for authentication, analysis, and history, with password hashing, JWT sessions, input limits, retries, and a backup-model option.

The supplied test script reports that all 47 automated checks pass, which supports the correctness of the implementation, although the model response is simulated in these tests. Future work includes measuring the accuracy of the live model on programs with known bugs in several languages, adding more automated tests for the frontend, supporting larger files and projects, deploying with MySQL and a production server, restricting the CORS policy, and collecting feedback from students on the usefulness of the explanations. With further evaluation, BUG-ZERO may serve as a practical learning and debugging aid that reduces the time spent on finding bugs and helps users learn from their mistakes.

5. ACKNOWLEDGMENTS

The authors express their sincere gratitude to Sri Shakthi Institute of Engineering and Technology for providing the academic environment and facilities required for this project. The authors thank the project guide, Dr. S. Prakash, for guidance, encouragement, and support throughout the development of BUG-ZERO. The authors also acknowledge the faculty members and friends who supported the learning and implementation process.

6. REFERENCES

1. Google, "Gemini API documentation." Available: https://ai.google.dev/gemini-api/docs
2. A. Vaswani et al., "Attention Is All You Need," in Advances in Neural Information Processing Systems (NeurIPS), 2017.
3. T. B. Brown et al., "Language Models are Few-Shot Learners," in Advances in Neural Information Processing Systems (NeurIPS), vol. 33, 2020.
4. M. Chen et al., "Evaluating Large Language Models Trained on Code," arXiv preprint arXiv:2107.03374, 2021.
5. M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," RFC 7519, Internet Engineering Task Force, 2015.
6. Flask, "Flask Documentation." Available: https://flask.palletsprojects.com/
7. PyJWT, "PyJWT Documentation." Available: https://pyjwt.readthedocs.io/
8. SQLite, "SQLite Documentation." Available: https://www.sqlite.org/docs.html
9. CodeMirror, "CodeMirror 5 User Manual." Available: https://codemirror.net/5/doc/manual.html
10. Project source code and test artifacts, "BUG-ZERO: AI-Powered Code Bug Finder," supplied project archive, BUG-ZERO.zip. Includes backend, frontend, database schema, README.md, docs/problem_statement.md, and tests/check_requirements.py.