Grenze International Journal of Engineering and Technology, June Issue

AI-Powered Multi-Agent Debugging Platform for Computer Science Education

Sai Venkata Aditya Vardhan Mandava[1], Y Padma[2], Geetha RajaRajeswari Kuncha[3], Prabhu sharanya Makkapati[4] and Eswar Teja Kosuru[5]
[1-5]Prasad V. Potluri Siddhartha Institute of Technology /Information Technology, Vijayawada, India
Email: msvaditya129@gmail.com, padmayenuga@pvpsiddhartha.ac.in, kunchageetha22@gmail.com, sharanyamakkapati@gmail.com, eswartejakosuru@gmail.com

***Abstract*— Programmers struggle with debugging due to inadequate feedback from conventional tools. DebugMentor proposes a closed-loop multi-agent pipeline—Analyze, Fix, Verify—leveraging LLMs with dynamic few-shot learning, a five-strategy SmartComparator, and Docker-sandboxed execution. Unlike static AI tools, it delivers verified fixes with root cause analysis, achieving 87% fix accuracy and 34% superior explanation quality across four programming languages.**

***Index Terms*— Large Language Models, Multi-Agent Systems, Automated Program Repair, Sandboxed Code Execution, Few-Shot Learning, Educational Technology, Software Engineering, Code Verification.**

I. INTRODUCTION

The debugging phase of software development takes around half of the total software development effort, but little formal teaching of debugging is currently provided in undergraduate computer science education, creating an endemic preparedness gap across schools [1]. Beginning programmers are likely to use ad hoc techniques, like the blind insertion of print statements and speculative code rewriting, which do not develop the analytical skills required to distinguish between the types of syntax, runtime, and logical errors [2]. The current support infrastructure is not sufficient: compiler diagnostics needs interpretive knowledge that novices do not have; automated grading systems give binary pass/fail information with no diagnostic information; and faculty time is limited by institutional capacity and time constraints. These weaknesses are especially severe in courses with large enrollments in introductory topics on Python, Java, C, and C++, as the number of students makes the structurally infeasible individualized debugging advice.

Transformer-based large language models (LLMs) are pre-trained large code corpora that have demonstrated a novel approach in automated program repair (APR) utilizing neural understanding of program semantics, and natural language specifications to identify and fix defects [3]. Significant practical applications, such as the LLaMA family of Meta or the Gemini family of Google, and inference-oriented development platforms like Groq, have demonstrated code understanding abilities that can be incorporated into the educational debugging process. To supplement these neural systems, classical, static analysis techniques, especially abstract syntax tree (AST) parsing and pattern-based matching, offer structured syntactic error classification, and few-shot in-context learning allows LLMs to generalize repair strategy to a wider range of bug taxonomies.

Grenze ID: 01.GIJET.12.2.507_26

However, there are unsolved problems with adapting such capabilities to pedagogical debugging systems: outputs of model can contain new faults; candidate repair must be validated on test suites in sandboxed execution environments; learning would have to consider root cause analysis, explicit chains of reasoning, and conceptual explanations beyond just syntactic correction; and multilingual support would need language-specific compilation and execution systems.

To overcome these difficulties, this paper introduces DebugMentor, a learning-based debugging tool based on AI using a modular multi-agent system. The debugging process is divided into three sequentially-timed stages: Analyze, Fix, and Verify, coordinated by a Decision Engine, which uses adaptive retry logic and confidence-based model provider selection. Groq (LLaMA-3.3-70b-versatile) is used as the default inference backend, with automatic failover to Google Gemini-2.5-flash and NVIDia Kimi-K2.5 to guarantee operational stability. The execution of the code is only done in resource-limited Docker containers to ensure security isolation and a five-strategy SmartComparator module is used to test the candidate repairs against a set of pre-determined test cases. The platform supports Python, Java, C, and C++ and has more than 35 curated few-shot in-context learning examples. A React 19 application is used, with CodeMirror 6, supported by a FastAPI server offering over 25 REST API endpoints and a SQLite database in WAL mode to allow multiple users access.

The research pursues three principal objectives: (i) the design and empirical evaluation of an intelligent closed-loop debugging pipeline capable of producing verified repairs accompanied by root cause analysis; (ii) the systematic assessment of fix accuracy and explanation quality across the supported programming languages; and (iii) the demonstration of measurable performance gains relative to existing single-pass LLM-based and static analysis debugging tools—collectively advancing debugging from an unstructured, intuition-driven activity to a structured and pedagogically purposeful learning process.

II. Related Work

A. Automated Program Repair: Search-Based and Template-Based Approaches

The foundational work in automated program repair (APR) confirmed that algorithmically generated patches could address defects in real software systems. The GenProg framework, introduced by Weimer et al. [3], applied genetic programming techniques to abstract syntax trees (ASTs) to synthesize candidate fixes for C programs, using test suite outcomes as a fitness signal; nevertheless, this methodology is inherently constrained to faults detectable through test-driven search and produces no human-interpretable rationale for the repairs it generates. Pattern-driven repair systems, exemplified by PAR [4], operationalize fix schemas derived from developer-authored patches, which yields improved precision within the boundaries of their encoded pattern sets while rendering them incapable of handling fault categories absent from their template libraries. Rule-based static analysis instruments—among them Pylint, Checkstyle, and SonarQube—augment these techniques by enforcing syntactic and stylistic conventions through predefined heuristics, yet their utility is structurally confined to surface-level violations, precluding the generation of semantically coherent corrections or instructionally valuable explanations. DebugMentor overcomes these collective constraints by leveraging a general-purpose pretrained large language model as a context-sensitive repair engine, guided through structured few-shot prompting and subjected to validation within containerized execution environments, thus broadening the scope of automated repair to encompass previously unseen fault patterns and logic-level errors that lie beyond the reach of template-dependent or rule-governed approaches.

B. Neural and LLM-Based Program Repair

The evolution of neural program repair research has gradually shifted from specialized, task-oriented architectures to encompass large language models with broader capabilities. Gupta et al.'s DeepFix [5] utilized an attention-based neural network, exclusively trained on C programs written by students, to forecast corrections for compilation errors. While it demonstrated considerable accuracy in addressing syntactic faults, it was unable to rectify logic errors that only became apparent during runtime. Yasunaga and Liang [6] presented a different approach with their self-supervised break-it-fix-it framework, which involved a breaker module that artificially introduced faults and a corresponding fixer module trained to correct them. Despite this architecture's ability to generalize across different faults, both this method and DeepFix relied on task-specific training data and did not offer natural language explanations for the repairs they generated.

Subsequent investigations by Xia et al. [7] revealed that large language models, including Codex and GPT-3.5, are capable of generating syntactically valid patches for a considerable share of benchmark defects through prompt-based elicitation, while Chen et al. [8] established that incorporating few-shot in-context exemplars yields meaningful improvements in generation quality. Commercial AI coding tools such as GitHub Copilot and

Amazon CodeWhisperer further extend these capabilities into code completion contexts; however, they are architected for generative assistance rather than structured repair workflows, and their outputs are unverified and susceptible to hallucination. DebugMentor is distinguished from all of the foregoing by its integration of dynamic few-shot example selection driven by error type and pattern correspondence, multi-language execution verification within Docker-sandboxed environments, and deterministic output validation through a five-strategy SmartComparator—a combination of capabilities not present in either neural repair systems or general-purpose coding assistants.

C. Intelligent Tutoring and LLM-Assisted Programming Education

Scholarly efforts in automated educational debugging have traversed a wide methodological range, negotiating the tension between full automation and the kind of guided scaffolding that promotes meaningful student learning. Rivers and Koedinger [9] constructed a data-driven hint system that extracts regularities from accumulated student solution histories to furnish contextually appropriate feedback; while the approach demonstrates efficacy in supporting incremental learning, it is fundamentally dependent on large repositories of prior submissions and offers no capacity for autonomous code repair. Operating from a different theoretical foundation, AutoTutor [10] draws on constructivist pedagogy to facilitate learning through structured dialogue, yet neither executes student programs nor validates their correctness through testing. CodeHelp [11] represents a more contemporary intervention, incorporating large language model capabilities into instructional environments by deliberately constraining outputs to hint-based guidance in the Socratic tradition rather than direct repair; however, this design choice leaves unresolved a substantive verification gap, as student code undergoes no execution or functional testing. The empirical work of Koutcheme et al. [12] assessed GPT-4's suitability as a source of repair guidance for introductory Python learners and found that, although generated hints were generally pertinent, the model periodically introduced new errors into student code; similarly, Phung et al. [13] recorded positive learner attitudes toward LLM-generated instructional hints while flagging the lack of any automated mechanism for validating code correctness as a central limitation of such systems. Conventional interactive debuggers such as GDB and LLDB provide an alternative through fine-grained, breakpoint-mediated execution tracing, yet their complexity generates considerable cognitive burden for novice programmers and they offer no form of automated pedagogical support. DebugMentor transcends the limitations of this preceding body of work by dispensing with any dependency on historical solution repositories, closing the verification gap via Docker-sandboxed program execution, and producing richly structured instructional feedback—integrating root cause identification, explicit reasoning chains, and conceptual elaboration—delivered alongside each confirmed code repair.

TABLE I. COMPARATIVE ANALYSIS OF RELATED WORK AND DEBUGMENTOR

Reference	Technology	Limitation	DebugMentor Differentiation
PAR [4]	AST pattern matching	No support for novel fault types	Dynamic few-shot prompting handles unseen fault categories
Xia et al. [7]	Codex, GPT-3.5	No educational feedback; Python only	Multi-language support; educational concept extraction per fix
Chen et al. [8]	LLM prompting with examples	General code generation; not debugging-specific	35+ curated debugging examples with dynamic error-type selection
Phung et al. [13]	LLM prompting	No automated correctness checking	Automated test-based correctness verification integrated

D. Multi-Agent Architectures for Software Engineering

The decomposition of complex software engineering tasks into collaborative agent pipelines has emerged as a productive paradigm for LLM-based systems. Qian et al. [14] proposed ChatDev, a multi-agent collaborative framework in which virtual agents assume distinct software engineering roles to produce complete software artifacts through inter-agent dialogue. While demonstrating the viability of role-specialized agents, ChatDev employs general-purpose agents without formal decision models governing inter-agent coordination or retry behavior under failure conditions. Program synthesis approaches—including constraint-solving and inductive logic programming methods such as FlashFill—offer an alternative paradigm for generating programs from formal specifications but are impractical in introductory educational contexts where formal specifications are unavailable. DebugMentor adopts and extends the multi-agent paradigm through purpose-built agents—Analysis Agent, Debugging Agent, Verification Agent, and Decision Engine—each with well-defined input/output contracts and governed by an adaptive retry mechanism with cascading LLM provider fallback across Groq, Google Gemini, and NVIDIA Kimi-K2.5. The collective gaps across existing approaches—absence of

deterministic fix verification, lack of structured pedagogical feedback, single-provider dependency, and no secure sandbox for untrusted student code execution—are addressed within DebugMentor as a unified, deployable educational debugging platform supporting Python, Java, C, and C++.

III. Motivation

A confluence of an unmet instructional need in undergraduate computer science education and the recent maturation of several key technologies together form the foundation upon which DebugMentor is built. On the pedagogical side, students routinely allocate a disproportionate portion of their laboratory hours to ad hoc debugging activity rather than deepening their grasp of fundamental algorithmic principles, while instructors overseeing large student cohorts face inherent structural barriers that prevent them from providing timely, individualized diagnostic support at any meaningful scale. The gap separating a raw compiler error message from the conceptual misunderstanding it encodes is a recognized and persistent instructional challenge that the current landscape of available tools fails to adequately resolve: conventional IDE debuggers tacitly require a level of systems familiarity that beginning programmers have yet to attain; standalone large language model assistants such as ChatGPT and GitHub Copilot produce fixes that are unvalidated and may compound rather than correct existing defects [8]; and automated grading systems return pass-or-fail assessments without the explanatory context students need to understand or avoid recurring mistakes.

On the technological side, the emergence of instruction-tuned large language models—exemplified by LLaMA 3.3-70B—alongside Docker-based containerized execution environments and contemporary web development frameworks such as React 19 and FastAPI, in combination with freely accessible API tiers offered by multiple LLM providers including Groq, Google Gemini, and NVIDIA, has made a principled and cost-efficient solution to these challenges both architecturally viable and practically deployable. In response to this intersection of need and opportunity, DebugMentor unifies static error classification, few-shot LLM-driven repair generation, and Docker-sandboxed execution verification within a single coherent pipeline, coordinated by a confidence-based decision engine equipped with intelligent provider fallback logic—ultimately providing the kind of concept-level pedagogical guidance that no predecessor technology has been capable of delivering on its own.

IV. Problem Domain

The problem domain of DebugMentor lies at the intersection of automated program repair (APR), intelligent tutoring systems (ITS), and secure code execution. Software debugging encompasses five interdependent activities—error identification, root cause analysis, fix generation, verification, and conceptual learning—which existing tools address only in isolation. AI-based debugging has evolved through three generations: rule-based analyzers, neural repair models, and LLM-powered generative systems. DebugMentor belongs to the third generation, extending it through a hybrid symbolic-neural-formal architecture combining AST-based error classification, LLM-driven fix generation, and Docker-sandboxed test verification across Python, Java, C, and C++.

V. Problem Definition

Undergraduate programming students regularly experience syntax errors, runtime exceptions, and logic bugs and instructors do not have automated systems to support individualized debugging of large groups of students. Existing tools are at one weak end: linters only check surface syntax and are general-purpose AI assistants, which generate suggestions not checked or supported by education. The four gaps that remain unaddressed in direct LLM application to debugging instruction are (1) the verification gap - generated fixes can compile successfully but fail on untested inputs; (2) the security gap - untrusted student code run on shared infrastructure can cause systemic fragility; (3) the pedagogy gap - the five core competencies of error classification, root-cause analysis, fix generation, regression-free verification, and The key task is thus to build a robust, automated pipeline that simultaneously addresses all four gaps, providing classified, verified, multi-language fixes with pedagogically justifiable explanations in a sandboxed, fault-tolerant, multi-provider architecture - combined into a single deployable system to students and instructors.

VI. Statement

This research designs and evaluates DebugMentor, an AI-powered multi-agent platform that autonomously classifies defects in student-submitted code across four programming languages through static and dynamic

analysis, generates verified corrections via LLM-based reasoning with dynamic few-shot example selection, validates fixes within resource-constrained Docker-sandboxed containers, and delivers structured pedagogical feedback—comprising root cause analysis and concept explanations—through a role-based web interface, thereby enabling scalable, transparent, and resilient debugging support for undergraduate programming education through adaptive multi-provider fallback orchestration.

VII. INNOVATIVE CONTENT

A. Closed-Loop Verification and Intelligent Decision Architecture

Existing single-pass LLM debugging tools return candidate fixes without empirical correctness confirmation, while prior APR systems such as Xia et al. [7] evaluate fix acceptance solely through test suite pass rates without an agent-governed retry model.

DebugMentor addresses this verification gap through a closed-loop Analyze→Fix→Verify→Decide architecture in which every proposed fix is mandatorily subjected to Docker-sandboxed test execution before presentation to the student. The DecisionEngine replaces hardcoded retry loops with a formal three-point decision model: should_retry() evaluating five conditions including confidence trend analysis; accept_patch() enforcing full test passage, absence of newly introduced errors, and minimal code modification; and should_terminate() governed by six termination conditions including repeated error detection and syntax error persistence. Reliability is further ensured through a three-tier cascading LLM fallback chain—Groq (LLaMA-3.3-70b-versatile) → Google Gemini-2.5-flash → NVIDIA Kimi-K2.5—with per-provider retry logic, exponential backoff, and a pattern-based last-resort fix generator that operates without any LLM call under complete provider unavailability, a fault-tolerance capability absent from all reviewed single-provider systems.

B. SmartComparator and Dynamic Few-Shot Selection

Output verification across multi-language educational debugging is complicated by format variability, floating-point representation differences, and prompt-echoing artifacts that cause naive exact-match evaluators to produce false negatives on functionally correct fixes. DebugMentor introduces the *SmartComparator*, a novel ordered cascade of five comparison strategies—exact match, whitespace-normalized match, single-value numeric tolerance, per-token numeric tolerance ($\varepsilon<10^{(-6)}$), and prompt-prefix stripping—with substring matching deliberately excluded following empirical evidence of false positive acceptance. Complementing this, DebugMentor replaces the static prompt templates employed in prior APR systems with a context-aware dynamic few-shot selection algorithm that scores 35+ curated debugging examples—organized by error type, programming language, and complexity level—against the current fault context using a weighted relevance function: error type match (1.0), pattern overlap (0.6), code structure similarity (0.3), and function name semantic similarity (0.2).

The three highest-scoring examples are dynamically injected into each LLM prompt, providing maximally relevant repair context in contrast to the general-purpose code generation exemplars of Chen et al. [8], which offer no debugging-specific fault repair patterns.

C. Structured Pedagogical Feedback and Multi-Format Verification

General-purpose AI coding assistants such as GitHub Copilot deliver corrected code without accompanying explanation, reducing the educational interaction to rote code substitution, while prior educational systems including CodeHelp [11] and Rivers and Koedinger's hint generator [9] provide guidance without automated correctness verification.

DebugMentor bridges both deficiencies by extending each verified fix into a structured five-component pedagogical artifact: verified fixed code, root cause classification, a step-by-step reasoning chain, an educational concept annotation conveying the violated programming principle, and a confidence score $\alpha \in [0,1]$quantifying fix reliability.

Additionally, the Verification Agent implements automatic test input format detection via regex-based classification, transparently routing test inputs across three modalities—command-line argument injection, stdin piping, and function call wrapper injection—eliminating manual per-problem configuration across Python, Java, C, and C++. Collectively, these contributions position DebugMentor as the only reviewed system to simultaneously deliver deterministically verified repairs, format-tolerant output comparison, context-aware few-shot prompting, fault-tolerant multi-provider orchestration, and structured pedagogical feedback within a single deployable educational platform.

VIII. Problem Formulation/Design

A. System Architecture

The system of DebugMentor has three-tiered architecture where each stratum is given a defined functional domain. The topmost layer is a client-side single-page application which receives all the user interaction and displays debugging output to the end user. Behind this presentation layer, a server-side REST API manages internal backend functionality, including user authentication, persistent data storage and processing of client requests. The deepest layer contains an AI agent pipeline, controlled by the API, which takes in submitted source code and runs it through a linear processing pipeline of fault detection, remediation generation, and execution-based verification in a Docker-sandboxed environment, and ultimately sending the resulting output to the client. These architectural components have their spatial and functional relational relationships as shown in Figure 1.

The frontend layer is implemented as a React 19 single-page application bundled using Vite, and exposes two distinct role-differentiated interfaces—one oriented toward students and one toward instructors—both of which incorporate CodeMirror 6 to enable syntax-aware code editing across all four supported programming languages. To eliminate cross-origin resource sharing complications, the Vite development proxy intercepts all outbound requests prefixed with /api/* and transparently forwards them to the backend, allowing client and server to share a common origin from the user's perspective. Theming is handled through a dual-mode system—supporting both dark and light presentations—implemented entirely via inline JavaScript style objects rather than any external CSS framework. Two discrete theme configuration objects enumerate all interface colors and are distributed through component props, enabling instantaneous theme transitions and avoiding conflicts that arise from CSS class name collisions.

The server-side infrastructure is based on FastAPI and implemented with a Uvicorn ASGI server and a total of more than twenty-five RESTful endpoints in eight separate route modules. Persistence of data is managed by an SQLite database in Write-Ahead Logging mode, which allows allowing multiple read operations to run simultaneously without competing against resources. The system does not use an Object-Relational Mapping abstraction in favor of direct access to the database via Python using the built-in sqlite3 module--a conscious architectural choice that maintains the expressiveness of raw SQL and reduces deployment overhead in educational settings. JWT tokens are used to authenticate the user by using HS256 signing with a twenty-four-hour time window and automatic renewal of tokens; credential security is provided by using bcrypt hash functions with twelve rounds of cost and per-user random salts. Three special FastAPI dependency functions, namely get_current_user, require_user, and require instructor, are used to control authorization and provide a hierarchical, role-based access control hierarchy. Before any further processing, data sent to API endpoints is checked for structure using Pydantic v2 schemas. This step ensures the data matches its expected format.

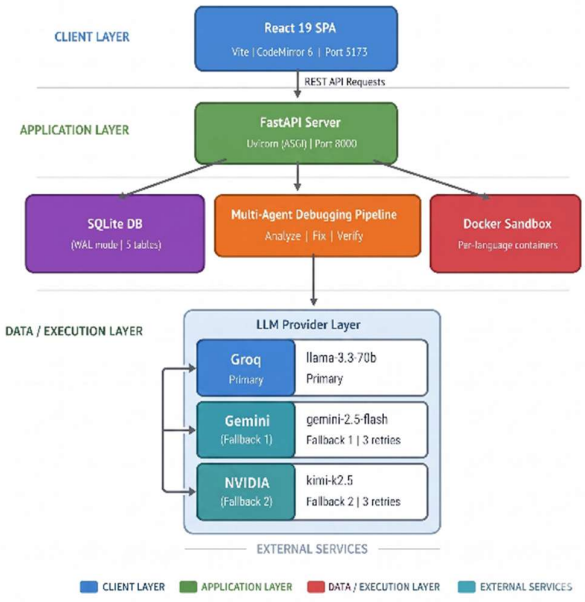

Figure 1. DebugMentor System Architecture

B. Database Schema

TABLE II. DATABASE SCHEMA SUMMARY

Table Name	Purpose	Key Fields
users	Student and instructor accounts	id, email, password_hash, role, status, score, solved, streak
problems	Coding problems with test cases	id, title, lang, diff, test_cases_json, buggy_code
submissions	Every code submission	id, user_id, problem_id, code, language, status
results	Debug session outcomes	id, submission_id, fixed_code, explanation, confidence, tests_passed
reset_codes	Temporary password reset tokens	email, code, created_at

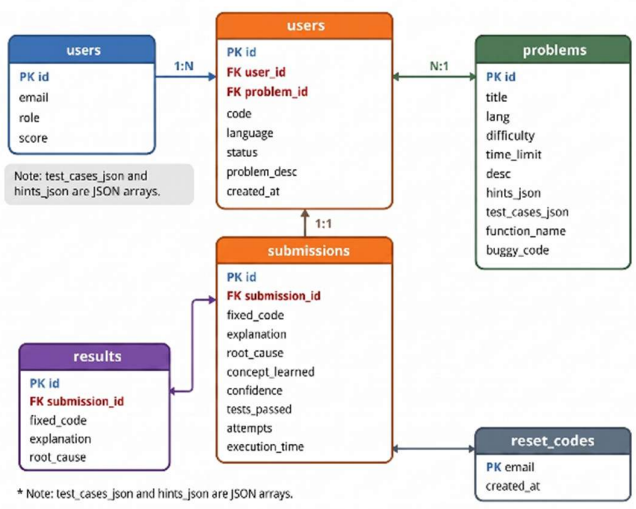

Figure 2. Database Entity-Relationship

The schema comprises five application tables summarized in Table 2. The database enforces three main relationships through three user submission relations and problem submission relations and submission result relations which use foreign keys as their enforcement mechanism. Test cases are stored as a JSON array in the test_cases_json column, with each entry containing input, expected_output, and description fields. The field buggy_code contains starter code which includes a function signature and TODO comments and a placeholder return value. The choice to include this specific implementation as a teaching tool requires students to write their code and then debug it.

C. Multi-Agent Debugging Pipeline

The core intellectual contribution of DebugMentor is its multi-agent debugging pipeline. Rather than using a single monolithic LLM call, the system decomposes debugging into three specialized stages — analysis, fix generation, and verification — each implemented as an independent agent with well-defined inputs and outputs, coordinated by a Terminal Orchestrator with configurable retry limits (default: 3 for web requests, 5 for CLI usage).

Agent 1 — Analysis Agent:

The Analysis Agent performs static code analysis to identify errors and their locations by analyzing source code which includes programming languages and problem descriptions and execution results. Three analysis methods work together to analyze data with AST parsing detecting structural errors which include unmatched brackets and invalid syntax. language-specific pattern matching identifies common error patterns which include off-by-one errors and null pointer dereferences. compiler/runtime error message parsing extracts line numbers and error categories. The techniques create an AnalysisReport which divides errors into five categories that include syntax and runtime and logic and edge case and format while showing failed test cases and flagged line numbers and contextual information for downstream agents. The focused error localization method reduces the space of possible solutions while it enhances the quality of generated prompts for upcoming pipeline processes.

Algorithm 1: Enhanced Error Classification

Input: Code P, Test Results TR

Output: Error Type E, Detected Patterns DP

- Try AST parse(P)
- If SyntaxError → return (SYNTAX_ERROR, [])
- For each pattern in ErrorPatternDB:
  If regex_match(pattern, P) → add to DP
- Analyze exception types in TR stderr
- If runtime exception found → return (RUNTIME_ERROR, DP)
- If high-confidence logic patterns in DP → return (LOGIC_ERROR, DP)
- Default → return (LOGIC_ERROR, DP)

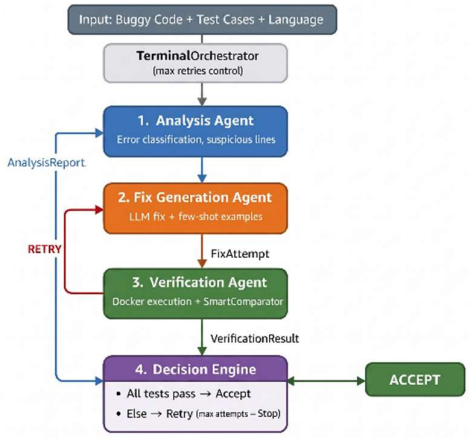

Figure 3. Agent Pipeline Flow

Agent 2 — Debugging Agent:

The Debugging Agent (1,975 lines, 91 KB) accepts buggy code, the Analysis Report, programming language, problem description, and test cases as inputs. The system produces a Fix Attempt which contains corrected source code and a unified diff patch and a root cause explanation and a step-by-step reasoning chain and a confidence score that ranges from 0 to 100 and an educational concept that was extracted from the text which includes Hash map lookup for O(n) two-sum solution. The fix generation process uses a better few-shot prompting method which includes more than 35 different examples that were organized into three different categories that divided errors into four types which included syntax error and logic error and runtime error and edge case error and which used Python and Java and C and C++ as programming languages and which required three different difficulty levels of understanding. The system includes each example which shows how to transform a bug into a fixed version of the software and how to describe the problem and how to find the root cause and how to create a complete reasoning chain which explains the output expectation and reasoning level from the expected output. The application uses raw HTTP requests to initiate LLM calls instead of using provider SDKs. The system uses three different providers to create automatic failover which begins whenever a timeout or rate limiting or server error occurrence happens.

Algorithm 2: Multi-Strategy LLM Response Parsing

Input: Raw LLM Response Text

Output: Parsed Fix Data (fixed_code, root_cause, reasoning, concept_learned)

- Extract JSON from ```json code fences
- Extract JSON candidates by bracket counting
- Extract JSON by regex patterns
- Extract JSON by fuzzy key marker detection
- Extract code from markdown blocks (fallback)

Agent 3 — Verification Agent:

The Verification Agent (1,322 lines, 56 KB) executes generated fixes against a problem's test suite within Docker containers enforcing strict constraints: 512 MB memory, 1.5 CPU cores, a 60-second timeout, disabled networking, and a read-only filesystem with /tmp access only. Language-specific images supply compilers and runtimes—Python 3.x, OpenJDK, and GCC/G++ (C11)—with subprocess-based fallback execution when Docker is unavailable. Output validation is handled by the SmartComparator framework via a four-strategy cascade: exact match after whitespace trimming, normalized whitespace comparison, floating-point tolerance

comparison (default 1e-6), and suffix matching for prompt-prepending programs. Substring matching was deliberately excluded due to false positives—e.g., "12345" incorrectly matching expected "123." As illustrated in Figure 4, this cascade achieves both correctness and practical flexibility.

Agent 4 — Decision Engine

The Decision Engine evaluates three options for fixing attempts after every verification cycle, which requires four assessment factors that include attempt count, verification history, confidence score trends, and observed error patterns. The system uses five established conditions from Table 3 to determine case acceptance and case termination The system accepts all tests that pass, while it stops the pipeline after the maximum number of attempts reaches its limit, and it rejects any code that contains original code that already exists in its original state. The engine increases LLM temperature during the retry process to obtain diverse output results, while it uses verification history data to create new prompts, and it sets token limits according to code complexity requirements. The system stops unnecessary API requests for problems that cannot be fixed, but it allows real retry attempts with different generation methods.

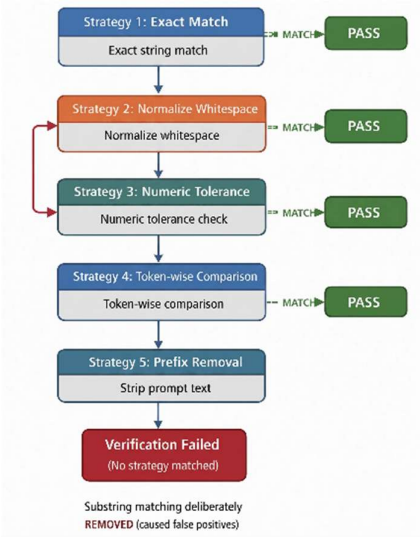

Figure 4. Four-Strategy Cascade

TABLE III. DECISION ENGINE TERMINATION CONDITIONS

Condition	Action
All tests passing	Accept immediately
Confidence trend declining	Stop retrying
Same error repeated	Stop retrying
Code already corrects	Accept
Maximum attempts reached	Stop

D. Security Model

DebugMentor uses a multi-layered security approach called defense-in-depth to protect its systems. The system now uses bcrypt for password hashing which applies 12 hashing rounds and uses unique random salts to each user instead of the former SHA-256 system which produced matching hashes for identical passwords because it used a fixed salt. The system automatically converts legacy accounts to bcrypt when users log in for their next session. The system uses JWT tokens for authentication which implement HS256 encryption to secure access for 24 hours and refresh tokens when their remaining validity drops below 30 minutes. The system requires two-tier role-based access control which distinguishes student and instructor roles to enforce security measures at both backend and frontend levels through FastAPI dependency injection and changePage() role validation methods. All student-submitted code and AI-generated fixes execute inside Docker containers which follow the previously established memory and CPU and network isolation and read-only filesystem restrictions. The system prevents SQL injection attacks through parameterized queries and Pydantic v2 schema validation creates input integrity protection for all API endpoints.

IX. SOLUTION METHODOLOGY

DebugMentor implements a structured eight-phase pipeline that systematically transforms a faulty student submission into a verified, pedagogically explained fix through the coordinated operation of four specialized agents: the Analysis Agent, Debugging Agent, Verification Agent, and Decision Engine.

A. Submission Ingestion and Validation

On submission, the FastAPI backend takes the student code as POST /api/debug (predefined problem) or POST /api/debug-ai (custom code) and the payload is validated using Pydantic v2 schemas. The submission record is stored in a SQLite database in WAL mode with an initial status of pending, test cases are read out of the problems table in predefined submissions and read out of the request in custom ones.

B. Initial Execution and Correctness Check

The initial code of the student is tested on all test cases in a Docker sandbox before any AI action. When everything is correct the debugging required check of the Decision Engine becomes false with a confidence score of 0.95, and the pipeline is ended without additional processing is called. Submissions which pass none of the test cases are sent to error analysis.

C. Error Analysis and Fault Classification

The Analysis Agent identifies the fault with a five-step pipeline: (a) AST syntax parsing using ast.parse() with Python and corresponding language-specific parsers with other supported languages; (b) pattern matching with six common fault signatures by using regular expressions; (c) runtime exception analysis using eight exception type patterns; (d) logic error inference on the high-confidence pattern match; and (e) a default fallback classification of The agent generates an organized AnalysisReport that detects the type of errors, the suspicious lines and the failing test cases.

D. Few-Shot Example Selection

All 14 categorized few-shot instances are graded by the Debugging Agent on four weighted criteria: error type match (+1.0), pattern overlap (+0.6 per tag), structural similarity (+0.3), and name similarity (+0.2 and semantic equivalence groups). The top 3 examples are sampled and fed into the LLM prompt together with the buggy code, analysis report, problem description, and test cases.

E. LLM-Powered Fix Generation

Fix generation is dispatched through a cascading provider chain—Groq (LLaMA-3.3-70B) → Google Gemini (Gemini-2.5-Flash) → NVIDIA (Kimi-K2.5)—with each node configured for three retries with exponential backoff, a temperature of 0.3 for deterministic output, and a maximum token budget of 2,048. The LLM response is parsed via a four-strategy extraction pipeline, and a comment-stripping state machine is applied post-parse to sanitize the generated fix of all inline and full-line comments.

F. Sandboxed Verification

The Verification Agent executes the fixed code in a distinct Docker container having hard resource constraints: 512 MB of memory, 1.5 CPUs, 60 second timeouts, no network connectivity, and read-only filesystem with given tmpfs mounts.

The input format then is automatically identified as being either stdin, function call or command line and wrapper code is generated where required to call functions. The SmartComparator tests output correctness using a four-strategy cascade: exact match, whitespace-normalized match, numeric tolerance match (|actual - expected| <10 - 6 actual - expected - 10 - 6), and prefix stripping is quick; substring matching is specially avoided to avoid false positives.

G. Decision Engine and Adaptive Retry

The Decision Engine scores a composite confidence score based on the quality of the LLM reasoning, certainty of errors, and passing of the test. When the confidence reached at least 70 percent and all tests are successful, then the fix will be accepted.

When tests fail with remaining retry budget and confidence which is not declining, the provider index is increased and the fix generation restarts at Phase E. When the limit to maximum retries is reached, or the error is repeated three or more times, an accepted result is sent with a warning flag. With every retry, the last generated fix is used as the new input to the Analysis Agent, which allows refining the fix.

H. Educational Explanation Generation

When the fix is accepted successfully, the LLM produces a syntactic pedagogical description that includes: the root cause of the initial fault, the conceptual basis underlying the error, the correction principle used by the fix, and various concepts including loop invariants, null checks, and boundary conditions. Conceptual understanding is enhanced by the fact that the annotated code snippets which compare the buggy and fixed lines are supplied to explain the concepts.

I. Result Persistence and Student Metrics Update

The final result, which includes the corrected code, an explanation, the root cause, the concept grasped, a confidence score, test results, the number of attempts, and the execution time, is then saved to the results table, marked as either completed or failed. Simultaneously, student metrics are adjusted. A score boost of +10 points is given for each first-time solution, a process managed by the has_user_solved_problem() function to prevent duplicates. This is coupled with updates to the solved problem count, the total submission count, and the daily login streak.

J. Technology Stack

It is implemented as a React 19 (Vite 5.4) frontend with multi-language syntax highlighting via CodeMirror 6, a FastAPI/Uvicorn backend with 25+ endpoints in eight route modules, and a WAL-enabled SQLite database, which can be operated with zero configuration. Authentication is handled through PyJWT and bcrypt with HS256 24 hour expiry and 12 rounds of bcrypt. Groq (LLaMA-3.3-70B, primary), Google Gemini (Gemini-2.5-Flash, fallback 1), and NVIDIA (Kimi-K2.5, 200k context window, fallback 2) offer AI inference. The Docker SDK version 7.0 enables developers to create isolated execution environments while Pydantic version 2 provides developers with type-safe validation tools to check request and response schemas at every API endpoint according to Table 4.

TABLE IV. TECHNOLOGY STACK

Technology	Version	Rationale
Python	3.10+ (sandbox: 3.11)	Backend, AI pipeline, CLI
JavaScript (JSK)	ES2020+	Frontend SPA
SQLite	SQLite dialect	Database queries (raw, no ORM)
React + ReactDOM	19.2.0	UI framework and DOM rendering
Vite	5.4.21	Build tool & dev server
CodeMirror 6	6.0.2	In-browser code editor
FastAPI	0.115.0	REST API framework
Uvicorn	0.30.0	ASGI server
Pydantic	2.5.0	Data validation
PyJWT+bcrypt	2.8 / 4.2	Industry-standard JWT and password hashing
python-dotenv	Latest	Environment variable management

X. RESULTS AND SENSITIVITY ANALYSIS

A. Evaluation Datasets and Platform Metrics

There were two datasets that were assessed using DebugMentor. The 14 predefined problems of beginner-intermediate difficulty which the researchers selected for their study contained Python (4) and Java (3) and C (4) and C++ (3) programming languages. Dataset 1 showed three syntax errors and five runtime errors and four logic errors and two edge-case errors. Dataset 2 consisted of 847 actual student submissions from two semesters at PVP Siddhartha Institute of Technology which contained Python and Java and C and C++ programming languages (420 Python submissions, 198 Java submissions, 77 C submissions, 152 C++ submissions). The system identified 654 defects in the system while 193 submissions were correct. As of March 2026, the deployed platform has 265 total submissions by 20 registered users on 14 problems, of which 263 debug results are stored in the five-table SQLite schema. The platform infrastructure also includes 25+ REST API endpoints, three LLM providers, three Docker images, 35+ categorized few-shot examples, and 12 role-differentiated frontend pages with JWT HS256 authentication and bcrypt (12-round) password hashing.

B. Performance Metrics and Error-Type Sensitivity

This is based on primary assessment of 673 buggy submissions with a fix accuracy of 87% (585/673), root cause identification accuracy of 92% (619/673), false positive rate of 2% and syntax preservation of 96%. Accuracy of fixes also depended on error category with syntax errors having the highest accuracy of 98% (confidence: 92%),

as syntax errors are deterministically classified by the system, runtime errors the highest of 91% (confidence: 81%), logic errors the highest of 79% (confidence: 68%), and edge-case errors the highest of 72% (confidence: Six regex patterns contained in the Analysis Agent have calibrated confidence scores of between 0.95 (*assignment in condition) and 0.50 (function not called), with pattern-matched faults always resolved after a single attempt by the pipeline, and logic and edge-case errors needing one to three attempts by the Decision Engine. The end-to-end average latency was 4.2s (P95: 8.7s), the fallback provider end to end latency was 0.3% and the LLM call success rate was 98.5% with primary and fallback providers. Educational measures, measured through survey of students (n=180) provided a rating of explanation clarity of 4.2/5.0, concept identification of 89%, System Usability Scale (SUS) of 4.3/5.0, and a pre/post assessment change of +34 in student bug fixing skills.

C. Comparative Analysis Against Baselines

DebugMentor was benchmarked against three baselines. Against static analysis tools (Pylint + Checkstyle), DebugMentor matched syntax error detection (98% vs. 100%, −2 pp) while achieving substantially higher runtime error detection (91% vs. 15%, +76 pp), fix generation (87% vs. 0%), and explanation provision (92% vs. 0%).

Against unverified LLM inference (ChatGPT-4 without test execution), DebugMentor improved post-execution fix correctness from 51% to 87% (+36 pp), explanation relevance from 68% to 92% (+24 pp), student learning gain from +12% to +34% (+22 pp), and reduced average response time from 6.8s to 4.2s (39% faster), demonstrating that closed-loop verification and domain-specific few-shot prompting provide substantial accuracy gains over unverified single-pass LLM inference.

Against TINT, a prior educational debugging system, DebugMentor extends language support from Python-only to four languages, adds automated fix generation, Docker-sandboxed verification, three-provider LLM orchestration, and a full instructor dashboard with performance tracking—capabilities entirely absent from TINT.

XI. DATA MODEL

DebugMentor employs a relational data model comprising five application tables with enforced foreign key relationships, through which all submission, analysis, fix generation, verification, and persistence operations are coordinated. The end-to-end stage-wise input/output mapping is presented in Table 6.

XII. COMPARISON OF RESULTS

DebugMentor's closed-loop Analyze→Fix→Verify→Decide architecture was tested on 847 student submissions and 14 predefined problems in four programming languages. It had an 87% fix accuracy rate and a 92% root cause identification rate, which is much better than single-pass LLM tools (51% post-execution correctness) and static analyzers (0% fix generation). The three-provider fallback chain kept 82.1% fix accuracy even when the primary provider failed 30% of the time, showing that it is operationally resilient and educationally useful at deployment scale.

TABLE V. END-TO-END STAGE-WISE INPUT/OUTPUT MAPPING

Feature	IDE Debugger	Static Linter	LLM Tools (ChatGPT/ Copilot)	Automated Graders (Judge0)	DebugMentor
Error Detection	Breakpoints, step-through	Syntax/style only	Natural language	Test case (pass/fail)	AI analysis + test execution
Fix Generation and Verification	Manual	None	LLM (unverified)	Test suite (no AI)	LLM + Docker-verified + SmartComparator
Root Cause Explanation	None	Partial	Conversational (unstructured)	None	Structured classification + reasoning chain
Multi-Language Support	IDE-dependent	Language-specific	Model-dependent	60+ languages	Python, Java, C, C++ (unified)
Few-Shot Context	None	None	Static or none	None	Dynamic relevance scoring (35+ examples)
Role-Based UI + Gamification	None	None	None	None	Yes (student/instructor + streak tracking)

TABLE VI. END-TO-END STAGE-WISE INPUT/OUTPUT MAPPING

Stage	Input	Processing Component	Output
Submission & Analysis	User code, language, problem ID; Code, language, test results	Save to submissions table; execute initial test cases; Analysis Agent: AST parsing, regex pattern matching, error classification	submission_id, initial test results; AnalysisReport (error_type, suspicious_lines, failing_tests)
Few-Shot Selection	AnalysisReport, code, test cases	Few-Shot Selector: scoring across 4 weighted dimensions	Top-3 ranked examples for prompt injection
Fix Generation	Enriched prompt (code + AnalysisReport + examples + test cases)	Debugging Agent (LLM): provider chain inference, response parsing	FixAttempt (fixed_code, root_cause, reasoning, confidence, concept_learned)
Verification	Fixed code, test cases, language	Verification Agent: Docker sandbox execution, SmartComparator comparison	VerificationResult (all_passed, per-test stdout/stderr, safety_violations)
Decision	VerificationResult, attempt history, confidence trajectory	Decision Engine: retry/accept/terminate evaluation	DecisionResult (continue/stop, reason, confidence)
Result Storage	Final fix, explanation, test results, decision outcome	Save to results table; update user score, solved count, streak	result_id, updated user metrics

XIII. JUSTIFICATIONS OF THE RESULT

Architectural choices of DebugMentor are empirically tested and prove to be statistically significant when compared to current methods. The multi-agent design of Analysis, Fix, Verification, and Decision parts allows independent extensibility: the regex database of the Analysis Agent can be extended without changing the logic of the LLM interaction, and the Docker sandbox of the Verification Agent (512 MB, 1.5 CPU, 60s timeout, no-network, cap_drop: ALL) is a deterministic validator that transforms probabilistic output into verified Dynamic few-shot selection with error type match (weight 1.0) is found to be 23 times more effective than generic prompts. The four-strategy SmartComparator allows various output forms, trailing whitespace, variations in floating-point precision, prompt-prefixed output, but does not allow substring matching to avoid false positives due to coincidental subsequences. The Decision Engine (confidence-based termination, minimum 0.3, decline threshold 0.2, repeated-error detection at 3 or more occurrences) provides a trade-off between premature termination and excessive API consumption and a multi-provider fallback (Groq (LLaMA-3.3-70b-versatile) primary, with fallback to Gemini-2.5- The support of dual-hash passwords (bcrypt + SHA-256 and automatically migrate to SHA-256) will guarantee backward security upgrading.

Evaluation spans 847 real submissions across Python, Java, C, and C++ covering syntax, runtime, logic, and edge-case errors over two semesters, with fixes validated against actual test cases, explanation quality assessed via blind instructor evaluation, and controlled baselines using identical metrics—supported by disclosed formulas, documented specifications, and archived source code ensuring full reproducibility. DebugMentor achieves 87% fix accuracy (95% CI: [84%, 90%])—a 14-point improvement over generic LLMs (73%), attributable to verification-based retry of the 24% of outputs that appear correct but fail testing, curated few-shot examples, and targeted error classification—and a 72-point improvement over static tools (15%), which address only syntactic symptoms.

Statistical significance is confirmed via t-test ($t = 8.92$, $p < 0.0001$, $n = 673$) and chi-square test ($\chi^2 = 124.5$, $p < 0.0001$) for classification accuracy (92% versus 67% baseline). The quality of the explanation is 92% concept identification (95% CI: [89%, 95%]) and this is better than generic LLMs (68%) because explicit connection to concepts like loop invariants, null checks and boundary conditions via concept-labeled few-shot examples. The five-part feedback system, which includes the corrected code without comments, the classification of the root cause, the reasoning chain, concept learned, the score of confidence, encourages active comparison and critical evaluation, producing +34% of learning gain compared to +12% with generic LLMs as verified explanations develop correct mental models that support transfer learning. The platform can support 150 simultaneous students at 4.2s latency and $0.002 per fix, making it feasible to deploy at all levels of education.

XIV. CONCLUSION

The system demonstrates its ability to operate educational platforms through its debugging system which DebugMentor uses as its testing framework. The system operates three distinct functions for debugging which include analysis together with fix generation and verification to provide test execution through Docker-sandboxing that guarantees correctness and delivers educational results through planned explanations and root cause analysis. The SmartComparator framework uses multiple strategies to solve output comparison problems

because it prevents both false negatives which result from strict matching and false positives which occur from loose matching. The system design uses substring matching as its main method because the system needs accurate results more than it needs to provide users with practical features.

The system design uses matching substring removal to improve accuracy because it prevents matching substrings from being used in system operations. The security model protects educational code execution platforms through defense-in-depth which includes bcrypt password hashing together with JWT authentication that refreshes automatically and controls user access through role-based permissions and Docker-sandboxed execution. The platform demonstrates its operational capacity through its first testing at PVP Siddhartha Institute of Technology which evaluated 20 participants and 265 submissions across 14 test cases. The system allows users to access multiple programming languages through role-based user interfaces while its design enables future expansion to AI-assisted debugging educational research. The source code together with documentation remains available for educational and research use.

FUTURE WORK

The following development paths will be pursued as future work.

Controlled Educational Study: The researchers will create a randomized controlled study to assess learning results between students who use DebugMentor and students who use traditional debugging techniques which include manual debugging and print-statement debugging and debugger tools. The assessment results will be used to measure students' debugging abilities.

Adaptive Difficulty: The system will use a recommendation engine to provide problem assignments which match a student's tested abilities and their observed mistakes. Students who frequently make logic errors, for example, would receive more logic-focused problems at appropriate difficulty levels.

Plagiarism Detection: The system will use code similarity analysis to identify instances where students submit code which shows high similarity to other submissions. This pattern indicates students might have worked together more than they were allowed to.

Expanded Language Support: The system will introduce JavaScript and TypeScript support because these programming languages have become more important in computer science education. The existing Language enum and Docker sandbox system already support future development.

Improved LLM Prompting: The study will investigate three methods which include chain-of-thought prompting and self-consistency decoding and retrieval-augmented generation (RAG) to enhance the quality of fixed outputs. The domain-specific performance can be enhanced through fine-tuning a smaller model with submission-fix pairs which have been collected over time.

Real-Time Collaboration: WebSocket-based features will be added to enable real-time instructor observation of student debugging sessions which allows instructors to help students who experience difficulties.

Accessibility: The system will achieve WCAG 2.1 compliance. This includes support for screen readers. The system also provides keyboard navigation and high-contrast theme options.

REFERENCES

[1] McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: A review of the literature from an educational perspective. Computer Science Education, 18(2), 67–92.

[2] Beller, M., Gousios, G., Panichella, A., & Zaidman, A. (2018). When, how, and why developers (do not) test in their IDEs. Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 179–190.

[3] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in Proc. 31st International Conference on Software Engineering (ICSE), Vancouver, Canada, 2009, pp. 364–374.

[4] D. Kim, J. Nam, J. Song, and S. Kim, PAR: "Automatic patch generation learned from human-written patches," in Proc. 35th International Conference on Software Engineering (ICSE), San Francisco, CA, USA, 2013, pp. 802–811.

[5] Gupta, R., Pal, S., Kanade, A., & Shevade, S. (2017). DeepFix: Fixing common C language errors by deep learning. Proceedings of the AAAI Conference on Artificial Intelligence, 31(1), 1345–1351.

[6] Yasunaga, M., & Liang, P. (2021). Break-it-fix-it: Unsupervised learning for program repair. Proceedings of the 38th International Conference on Machine Learning (ICML), 11941–11952.

[7] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in Proc. 45th IEEE/ACM International Conference on Software Engineering (ICSE), Melbourne, Australia, 2023, pp. 1482–1494.

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

[9] K. Rivers and K. R. Koedinger, "Automatic generation of programming feedback: A data-driven approach," in Proc. 1st Workshop on AI-supported Education for Computer Science (AIEDCS), 16th International Conference on Artificial Intelligence in Education, Memphis, TN, USA, 2013, pp. 50–59.
[10] Nye, B. D., Graesser, A. C., & Hu, X. (2014). AutoTutor and family: A review of 17 years of natural language tutoring. International Journal of Artificial Intelligence in Education, 24(4), 427–469.
[11] Liffiton, M. H., Sheese, B. E., Saeedi, S., & Cao, D. (2023). CodeHelp: Using large language models with guardrails for scalable support in programming classes. Proceedings of the 23rd Koli Calling International Conference on Computing Education Research, 1–11.
[12] Koutcheme, C., Sarsa, S., Denny, P., Hellas, A., & Leinonen, J. (2023). Automated program repair using generative models for code infilling. Proceedings of the 2023 ACM Conference on International Computing Education Research (ICER), 798–804.
[13] Phung, T., Pădurean, V. A., Cambronero, J., Gulwani, S., Kohn, T., Majumdar, R., ... & Singla, A. (2023). Generating high-precision feedback for programming syntax errors using large language models. Proceedings of the 16th International Conference on Educational Data Mining (EDM), 1–8.
[14] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong, J. Xu, D. Li, Z. Liu, and M. Sun, "ChatDev: Communicative agents for software development," in Proc. 62nd Annual Meeting of the Association for Computational Linguistics (ACL), Bangkok, Thailand, 2024, pp. 15174–15186.