International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 10 | Oct - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

CodeArena: A Smart Online Coding Judge for Automated Programming Assessment

[1]Manobi Barman
CSE-Computer Science and Engineering
Hyderabad Institute of Technology and Management (HITAM)
Hyderabad, Telangana, India.
email: manobibarman26@gmail.com

K. Sowjanya
CSE-Computer Science and Engineering
Hyderabad Institute of Technology and Management (HITAM)
Hyderabad, Telangana, India.
email: Kanapuramsowjanya31@gmail.com

Nabila Naaz
CSE-Computer Science and Engineering
Hyderabad Institute of Technology and Management (HITAM)
Hyderabad, Telangana, India.
email: naaznabila217@gmail.com

Durga Devi
CSE-Computer Science and Engineering
Hyderabad Institute of Technology and Management (HITAM)
Hyderabad, Telangana, India.
email: Durgadevi0761@gmail.com

Guided by: Dr. Dharmendra Roy

Abstract: CodeArena is a web-based online coding judge designed to combine programming practice, automated evaluation, and academic assessment in a single workflow. Students can register and authenticate, browse published programming problems, write source code, submit solutions, and review evaluation results. Administrators can create and manage problems, configure test cases, maintain contest data, and monitor submissions. The system uses a React-based frontend, Java Spring Boot REST services, and MySQL for persistent storage. When a submission is received, the backend validates the request, associates the source code with the selected problem, retrieves the configured test data, executes the evaluation process, records the outcome, and returns a clear result to the learner. The design separates user interaction, application logic, data management, and code evaluation so that each responsibility can be developed and maintained independently. CodeArena is intended to reduce repetitive manual checking, provide faster and more consistent feedback, preserve submission history, and support programming exercises and contests in an academic environment. The paper presents the system requirements, architecture, implementation workflow, functional testing approach, limitations, and possible extensions.

Keywords: online judge, automated assessment, programming education, code evaluation, Spring Boot, React, MySQL, REST API.

I. INTRODUCTION

Programming assignments become increasingly difficult to check manually as the number of learners and submissions grows. An instructor may need to inspect source code, identify compilation or runtime problems, run several inputs, and record the result for each attempt. Repeating these steps for many students consumes time and can make feedback inconsistent, especially when the same problem is submitted many times.

CodeArena addresses this workflow by shifting repeatable checking into software. A submitted program is associated with a selected problem and its test data, processed by the evaluation stage, and assigned a result that can be displayed immediately. The same application also gives administrators a place to maintain the material used for assessment, so problem creation and submission evaluation are handled within one system.

The design therefore connects three practical requirements. On the learner side, the interface should make problem selection, coding, submission, and result review straightforward. On the administration side, authorized users need controls for problems, test cases, contests, and user information. Between these two sides, the backend must preserve records and coordinate the request, database, and judging steps reliably.

The prototype combines React on the client side, Java Spring Boot for application services, REST communication, and MySQL for persistent records. These responsibilities are separated so that interface changes do not require rewriting database operations and judging logic is not placed in the browser. The same structure also leaves a path for later additions such as more languages, contest support, analytics, and stronger execution isolation.

The remainder presents related work, requirements, architecture, implementation, testing, security, discussion, limitations, and future work.

II. RELATED WORK

A. Online-Judge Foundations

Online-judge foundations. Kurnia, Lim, and Cheang [1] established the online-judge model as a way to support programming practice through automated submission and evaluation. Cheang et al. [2] further examined automated grading in an academic setting, showing how program execution and test-based checking can reduce repetitive manual assessment. These studies provide the foundation for CodeArena's submission-and-evaluation workflow, while our implementation applies the same principle to a compact academic web platform.

B. Architecture and Learning Support

Architecture and learning support. Wasik et al. [3] surveyed online-judge systems and identified recurring architectural elements such as problem repositories, submission processing, execution environments, and feedback. Rubio-Sánchez et al. [4] examined student perception and use of automated programming assessment, highlighting the value of usable feedback and repeated practice. Watanobe et al. [5] discussed requirements, architecture, and operational experience for online-judge systems. CodeArena relates to these contributions by combining problem management, user roles, REST services, persistent submission records, and result presentation within one academic-oriented prototype.

C. Assessment Quality and Problem Design

Assessment quality and problem design. Wang et al. [6] studied how problem difficulty and ordering can influence learning outcomes in online-judge environments, indicating that the assessment workflow should be considered together with the design of programming tasks. Liu et al. [7] investigated the quality of tests

used to judge submissions, emphasizing that the judge itself must be evaluated carefully. These findings motivate CodeArena's use of explicit test cases, recognizable evaluation outcomes, and administrator-controlled problem data rather than treating compilation alone as evidence of correctness.

D. Positioning of CodeArena

Position of the present work. In this paper, Manobi Barman, K. Sowjanya, Nabila Naaz, and Durga Devi present CodeArena as a practical integration of the contributions above. Existing research provides strong foundations for automated grading, online-judge architecture, student interaction, and test quality; CodeArena adds a focused implementation that connects student and administrator workflows, React-based interaction, Spring Boot REST services, MySQL persistence, submission history, and test-case evaluation in one academic system. The contribution is therefore not a new judging algorithm, but a clear, modular, and extensible implementation of established online-judge practices for programming education.

III. REQUIREMENTS AND PROBLEM ANALYSIS

A. Functional Requirements

CodeArena separates users into student and administrator roles. Students can register and authenticate, browse available problems, read requirements, enter source code, submit attempts, and review their recorded results. Administrators receive additional controls for maintaining problem statements, test cases, contests, user records, and other assessment data.

The central functional path is the submission lifecycle. After a student presses submit, the frontend sends the selected problem, source code, and required metadata to the REST backend. The backend checks the request, creates the submission record, passes the program to the evaluation stage, receives the judging outcome, updates the stored status, and sends the result back to the interface.

Problem records must contain enough information for both the learner and the evaluator, including the statement, instructions or constraints, expected input/output behavior, and associated test cases. Each test case is linked to its problem so that submissions are judged with the intended data. Contest support can then expose a selected group of problems within a particular assessment context.

B. Non-Functional Requirements

For normal academic workloads, the application should respond predictably, preserve consistent database records, and communicate failures in a form that users can understand. The separation between React components and Spring Boot services supports maintainability, while REST boundaries provide a clear interface between the client and server. Protected operations also require authentication and administrator authorization.

The evaluation path needs explicit failure handling. Compilation errors, runtime exceptions, wrong output, and execution timeouts should become recognizable submission states instead of terminating the surrounding web application. Treating evaluation as its own processing stage makes these outcomes easier to record, display, and test.

IV. SYSTEM ARCHITECTURE AND METHODOLOGY

A. Architecture Overview

The application is organized as a client–server stack. React supplies the visible pages, forms, problem screens, coding interactions, authentication views, and result displays. Spring Boot provides the REST endpoints and application logic for authentication, problem access, submissions, and administration. MySQL supplies durable storage for the structured records used by these services.

The normal request path can be represented as User Interface → REST API → Authentication/Business Logic → MySQL and Evaluation Stage → Submission Result → User Interface. The server first validates the incoming request and applies the relevant permission checks. Database operations read or update persistent records, while a code submission is handed to the judging workflow. After evaluation, the resulting status is returned through the API and rendered by the frontend.

The browser is intentionally not treated as the authority for judging. Source code, expected outputs, test cases, and administrator operations remain under backend control. This prevents a client-side request from deciding its own correctness result and keeps the evaluation rules in the server-side workflow.

B. Submission Evaluation Workflow

A submission starts when a learner chooses a problem and sends source code from the coding interface. The backend resolves the selected problem, loads its configured tests, validates the accompanying metadata, and creates a record for the attempt. The program is then prepared for execution using the evaluation environment supported by the prototype.

The evaluator processes the configured inputs and records the program output together with its execution state. The produced output is checked against the expected result using the applicable judging rule. When all required tests satisfy that rule, the submission can be marked accepted. Otherwise, the stored result identifies the observed failure, such as a compilation error, runtime failure, wrong answer, or timeout.

For internal evaluation, the project can calculate test-case pass rate as Pass Rate = (Passed Test Cases / Total Test Cases) × 100. The value is useful for diagnosing individual evaluation batches and for presenting additional feedback, although the final accepted/rejected decision remains governed by the configured judging rule.

C. Database Design

MySQL stores the persistent entities required by the workflow, including users, roles, problems, test cases, submissions, contests, and links between contests and problems. User records retain account and role information; problem records hold assessment content; test cases point to their parent problem; and submission records connect a user with a problem and preserve the resulting status for later history views.

The database relationships must prevent invalid references; for example, a submission should correspond to an existing user and problem, and a test case should belong to a valid problem. Frequently searched attributes such as user ID, problem ID, contest ID, and submission time can be indexed to keep problem lists and submission histories responsive as the amount of stored data increases.

V. IMPLEMENTATION

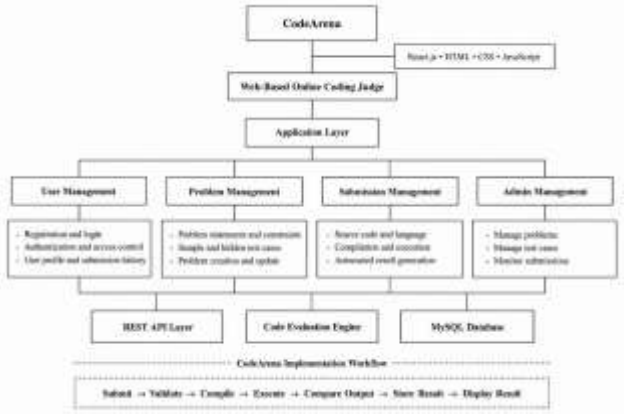


Fig. 1. CodeArena System Architecture and Implementation Workflow.

A. Frontend

The user interface is built with React and divided into reusable components. Navigation elements, forms, problem displays, submission views, and administrative controls can therefore be maintained separately while still participating in the same application state. This component-based structure also makes it easier to add or revise individual screens without rewriting the complete frontend.

The coding flow keeps the main task in one context: the learner opens a problem, reviews its requirements, enters the solution, chooses the required options, and submits. Once the server finishes processing the attempt, the returned status and available feedback are shown in the same workflow. Authentication screens control entry into student and administrator features.

B. Backend and REST Services

The server side is implemented in Java with Spring Boot. REST controllers receive requests from the React client, service classes coordinate application rules, and data-access components communicate with MySQL. Keeping these responsibilities separate gives the backend a clear path for validation, authorization, persistence, and submission processing.

Backend services validate incoming data, check permissions, perform database operations, and coordinate the submission lifecycle. Because the browser communicates through REST endpoints rather than directly with the database, the client remains independent of internal service implementation. The same endpoints can also provide a foundation for additional clients or integrations later.

C. Build and Database Layer

Maven manages the Java build configuration and project dependencies. MySQL provides the relational store used by CodeArena for accounts, problem data, test cases, contests, and submission records. Together, the build and database layers provide the supporting infrastructure around the React interface, Spring Boot services, and judging workflow.

The backend can be organized into configuration, controller, service, repository, entity/model, and utility responsibilities. This arrangement keeps presentation concerns outside persistence logic and gives new endpoints or business rules a defined place. As a result, changes to the interface can be made without coupling them directly to database implementation details.

VI. TESTING AND RESULTS

A. Testing Strategy

Testing covers several levels of the application. Focused checks can validate individual rules and methods, integration checks can exercise the REST, service, and database boundaries together, and functional tests can follow complete flows such as registration, login, problem access, submission, result display, and administrator actions. Separate evaluation tests address accepted programs, incorrect output, compilation problems, and runtime failures.

Failure-oriented testing is especially relevant because the platform accepts source code and user-controlled requests. Missing fields, invalid identifiers, unsupported submissions, malformed requests, and repeated actions should produce controlled responses without damaging stored data. The frontend must likewise turn backend failures into understandable messages instead of leaving the user with an unexplained error.

TABLE I. FUNCTIONAL TESTS

Test Area	Scenario	Expected Behavior
Authentication	Valid and invalid login	Valid credentials allow access; invalid credentials are rejected.
Problem Management	Create/update problem and test cases	Authorized administrator can persist valid problem data.
Submission	Correct solution	Submission is evaluated and receives an accepted result when all required tests pass.
Submission	Incorrect output	Submission is marked as a failure and feedback is returned.
Execution	Compilation/runtime failure	Failure is captured as a controlled evaluation status.
History	View previous submissions	Authenticated user can retrieve their recorded submission results.
Authorization	Student attempts admin action	Protected administrative operation is denied.

Table I summarizes representative functional scenarios used to verify the intended behavior of CodeArena. The cases focus on observable outcomes rather than measured throughput or latency.

B. Evaluation Metrics

Performance benchmarking should measure response time, execution time, test-case pass rate, compilation-failure rate, runtime-error rate, and database-query latency under normal and peak loads. Average response time is calculated as total response time divided by the number of submissions. Repeated benchmark runs can compare throughput, latency, and reliability across deployment configurations and submission volumes.

The main outcome of the prototype is the integration of the assessment loop in one application: administrators publish problems and test data, learners submit programs, the evaluator checks those programs, the result is stored, and the interface presents the outcome. This removes repeated manual checking from the normal workflow and gives students a consistent place to practice and review attempts.

VII. SECURITY, RELIABILITY, AND LIMITATIONS

A. Security Considerations

Submitted programs must be treated as untrusted data. Authentication should protect account access, while role checks should restrict administrator operations. Request values should be validated before database work is performed, and secrets or sensitive configuration should remain on the server side. Database access should use safe parameterized mechanisms provided by the application's data-access layer.

B. Reliability and Limitations

The architecture supports the basic automated-judging workflow, but its reliability still depends on the evaluator, compiler/runtime environment, and quality of the test data. Hidden tests should cover meaningful edge cases, and differences in compiler versions, language runtimes, operating-system behavior, or resource limits can affect whether an execution is reproducible.

The present implementation targets an academic-scale workflow rather than the infrastructure required by a large competitive-programming service. Higher concurrency would benefit from asynchronous queues, separate judging workers, caching, stronger sandboxing, and additional monitoring. Useful operational signals would include evaluator health, queue depth, failed executions, and API response time.

A second limitation comes from the assessment content itself. An automated judge can apply a defined rule consistently, but it cannot repair an incomplete problem statement or compensate for missing edge cases in the test set. Before publication, instructors should therefore review both the problem specification and its test cases.

VIII. DISCUSSION

A. Current Findings and Design Implications

CodeArena combines familiar web technologies around a dedicated programming-evaluation workflow. React handles learner and administrator interaction, Spring Boot coordinates server-side operations, MySQL preserves application state, and the judging stage connects submitted programs with the test data defined for each problem. The separation of these responsibilities makes the prototype easier to reason about and modify.

The useful part of the platform is the complete workflow rather than compilation alone. A learner can make repeated attempts and inspect the stored outcomes, while an administrator can maintain a controlled collection of problems and tests. Bringing these activities together reduces the amount of repeated checking required from an instructor and creates a persistent feedback path for learners.

The current structure also supports several extensions. Additional languages can be introduced through language-specific compiler and runtime settings. Contest views can summarize participation and problem-level results, while historical submissions can support later difficulty estimation. The result model can also distinguish compilation diagnostics, runtime failures, and incorrect output while keeping hidden test information private.

The project illustrates why the interface, service layer, data model, and evaluator should not be tightly coupled. Each layer has different responsibilities and different failure conditions. Keeping those boundaries explicit makes faults easier to isolate and reduces the chance that a modification in one part of the application will unintentionally alter another part.

B. Future Enhancements

The prototype can be extended with multiple programming languages, asynchronous judging, and richer learning analytics. Language-specific compiler and runtime settings would allow the same judging interface to support additional languages, while a submission queue would separate code execution from the browser request and improve concurrency. Submission history could also be summarized to identify repeated compilation errors, frequently failed tests, or unusually high attempt counts. Other practical additions include scheduled contests, problem tags, configurable scoring, notifications, similarity-analysis assistance, and deployment monitoring.

C. Practical Outcome

The completed prototype combines authentication, problem selection, coding, submission, and result review in a single application. Submission records retain the relationship between the learner and the selected problem, while administrator functions provide a controlled way to maintain the problems and test cases used during evaluation.

Because the interface, server, database, and evaluator have separate responsibilities, the system can be developed incrementally. A change to a React screen does not require the database schema to be redesigned, and judging rules remain behind server-side services. The same separation also makes later monitoring or scaling of individual components more practical.

The test scenarios cover the main paths through which the two user roles interact with CodeArena: authentication, problem administration, successful and unsuccessful submissions, compilation/runtime failures, submission history, and authorization checks. These cases provide a baseline test set that can be expanded when more languages or judging features are introduced.

D. Scalability Considerations

With many simultaneous users, judging becomes the main workload. The web layer should acknowledge submissions quickly and place them in an asynchronous task queue. Separate worker processes can consume queued attempts under configured resource limits, while the database stores each job's status for frontend retrieval. This keeps interactive requests separate from computation-heavy evaluation and supports controlled horizontal scaling.

Frequently requested problem information and public metadata can be cached to reduce repeated database reads. Indexes on user IDs, problem IDs, and submission timestamps can also support history and administration queries. If the database becomes large, older submission records can be archived separately from active judging data so that routine operations remain efficient.

The modular structure also makes it possible to compare alternative judging policies, feedback formats, execution limits, or scoring methods without rebuilding the complete application. That flexibility is useful in an academic project because different implementations can be tested while the student-facing workflow remains stable.

E. Integration and Extension Path

Future extensions can be introduced through modular adapters for languages, assessment policies, and administration. A language adapter can map source code, input, expected output, and resource limits to a common execution request; assessment adapters can support exact matching, numeric tolerance, or custom validators. The administrator layer can later add draft/publish states, reusable test sets, scheduled contests, and audit records. This path allows the interface, REST services, database, and evaluator to evolve independently while preserving the core CodeArena workflow.

IX. CONCLUSION

CodeArena brings the main stages of programming assessment into one web-based workflow. Students can work with published problems and submit solutions, while administrators maintain the assessment material. The backend stores the relevant records and coordinates the evaluation path, reducing the amount of repetitive checking that would otherwise be performed manually.

The implementation combines React, Spring Boot, MySQL, REST APIs, and Maven in a modular design. Its main strengths are separation of responsibilities, persistent submission history, explicit evaluation outcomes, and role-based administrative access. Testing covers successful paths and failure conditions because predictable behavior is essential for a coding judge. Future development should prioritize Docker-based sandboxing with CPU, memory, filesystem, and network limits; asynchronous task queues with independent judging workers; and benchmark tests measuring response latency, execution time, and throughput under peak loads. Additional languages, richer contest analytics, stronger observability, and similarity-analysis assistance can then extend the academic prototype toward an institutional programming-assessment service.

ACKNOWLEDGMENT

The authors gratefully acknowledge the academic guidance and support received throughout the development of CodeArena. They also acknowledge the official documentation and open-source resources used from React, Spring Boot, MySQL, and Apache Maven.

REFERENCES

[1] A. Kurnia, A. Lim, and B. Cheang, “Online Judge,” Computers & Education, vol. 36, no. 4, pp. 299–315, 2001, doi: 10.1016/S0360-1315(01)00018-5. Source

[2] B. Cheang, A. Kurnia, A. Lim, and W.-C. Oon, “On automated grading of programming assignments in an academic institution,” Computers & Education, vol. 41, no. 2, pp. 121–131, 2003, doi: 10.1016/S0360-1315(03)00030-7. Source

[3] S. Wasik, M. Antczak, J. Badura, A. Laskowski, and T. Sternal, “A Survey on Online Judge Systems and Their Applications,” ACM Computing Surveys, vol. 51, no. 1, Art. no. 3, pp. 1–34, 2018, doi: 10.1145/3143560. Source

[4] M. Rubio-Sánchez, P. Kinnunen, C. Pareja-Flores, and Á. Velázquez-Iturbide, “Student perception and usage of an automated programming assessment tool,” Computers in Human Behavior, vol. 31, pp. 453–460, 2014, doi: 10.1016/j.chb.2013.04.001. Source

[5] Y. Watanobe, M. M. Rahman, T. Matsumoto, U. K. Rage, and P. Ravikumar, “Online Judge System: Requirements, Architecture, and Experiences,” International Journal of Software Engineering and Knowledge Engineering, vol. 32, no. 6, pp. 917–946, 2022, doi: 10.1142/S0218194022500346. Source

[6] J. Wang, P. Lin, Z. Tang, and S. Chen, “How problem difficulty and order influence programming education outcomes in online judge systems,” Heliyon, vol. 9, no. 11, Art. no. e20947, 2023, doi: 10.1016/j.heliyon.2023.e20947. Source

[7] K. Liu, Y. Han, J. M. Zhang, Z. Chen, F. Sarro, M. Harman, G. Huang, and Y. Ma, “Who Judges the Judge: An Empirical Study on Online Judge Tests,” in Proc. 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 334–346, 2023, doi: 10.1145/3597926.3598060. Source