International Journal of Combined Research & Development (IJCRD)
eISSN:2321-225X; pISSN:2321-2241 Volume: 15; Issue: 6; June -2026

An Intelligent Online Judge System for Multi-Language Automated Programming Assessment

Achutha JC
Asst. Professor,Department of Master of Computer Applications
AMC Engineering College, Bengaluru, India
achutha.sir@gmail.com

Abstract—Online Judge (OJ) platforms are widely used to evaluate programming solutions automatically, but conventional systems often concentrate primarily on verdict generation and provide limited support for learning-oriented feedback, multi-language management, performance analysis, and secure execution. This paper presents an intelligent online judge architecture designed for automated programming assessment across multiple programming languages. The proposed framework combines source-code submission, language-aware compilation, isolated execution, configurable test cases, resource monitoring, verdict generation, code-quality indicators, and learner analytics in a unified workflow. The architecture separates the web interface, judging service, compilation/execution layer, data store, and analytics components to improve scalability and maintainability. A methodology for evaluating correctness, execution time, memory consumption, compilation failures, and submission behavior is also presented. The proposed design can support classroom laboratories, programming contests, continuous assessment, and self-learning environments. The paper describes the system architecture, workflow, security considerations, evaluation methodology, and potential extensions including automated feedback, code similarity analysis, and AI-assisted test-case generation. The work provides a practical blueprint for building a secure and extensible online programming assessment platform.

Keywords—Online Judge, automated programming assessment, code evaluation, multi-language compiler, sandboxing, programming education, learning analytics, automated testing.

I. INTRODUCTION

Programming education requires frequent practice and timely evaluation. In traditional laboratory environments, instructors manually compile programs, execute them against selected inputs, inspect outputs, and record marks. This process becomes difficult when the number of learners and submissions increases. An Online Judge addresses this problem by automatically compiling submitted programs, executing them against predefined test cases, and producing a verdict.

Existing competitive-programming judges demonstrate the effectiveness of automated evaluation, but an academic OJ may require additional capabilities. Students and instructors can benefit from language-independent assessment, detailed resource measurements, submission history, performance dashboards, and controlled feedback. At the same time, executing untrusted source code creates security risks because a submitted program may consume excessive resources, access unauthorized files, or attempt network communication.

This paper proposes an intelligent OJ architecture that treats judging as a pipeline rather than a single compilation step. The major contributions are: (1) a modular architecture for multi-language submission and evaluation; (2) a controlled execution model with CPU, memory, process, and time constraints; (3) a test-case based verdict engine; (4) a data model for submission and learner analytics; and (5) an evaluation methodology suitable for academic programming assessment.

The remainder of the paper is organized as follows. Section II discusses related work. Section III defines the problem and requirements. Section IV presents the proposed architecture. Section V describes the methodology and judging workflow. Section VI discusses implementation considerations and security. Section VII presents the evaluation framework. Section VIII discusses benefits and limitations. Section IX describes future enhancements, followed by the conclusion and references.

II. RELATED WORK

Automated programming assessment has been studied through online judges, automated programming tutors, and learning analytics systems. Online judges typically use a collection of input/output test cases to determine whether a program is correct. Competitive-programming platforms have demonstrated that automatic compilation and execution can scale to large numbers of submissions.

Automated assessment research has also investigated richer feedback mechanisms. Ihantola et al. surveyed automatic assessment approaches and identified issues related to testing, feedback, and system design [1]. Ala-Mutka discussed

automated assessment methods for programming assignments and emphasized the importance of appropriate testing and feedback [2]. EdX and related educational platforms demonstrated the feasibility of integrating automatic code evaluation into large-scale learning environments [3].

Security is another important research area. The execution of untrusted programs requires isolation and resource controls. Container technologies provide process and filesystem isolation, while operating-system mechanisms can limit resources and capabilities. The proposed architecture incorporates these principles without tying the design to a single virtualization technology.

Unlike a purely contest-oriented judge, the proposed system is designed with an academic focus. It preserves the conventional Accepted/Wrong Answer/Compilation Error/Time Limit Exceeded style of verdicts while also collecting metrics that can support learner progress analysis and instructor intervention.

III. PROBLEM STATEMENT AND REQUIREMENTS

The problem addressed in this work is the design of a secure, scalable, and extensible system that automatically evaluates programming solutions written in different languages. The system must accept source code, identify the requested language, compile the program using the appropriate toolchain, execute the resulting program under controlled conditions, compare outputs with expected results, and store the outcome.

Functional requirements include user authentication, problem selection, source submission, language selection, compilation, test-case execution, verdict generation, submission history, and administrative problem management. Non-functional requirements include security, scalability, reliability, reproducibility, low evaluation latency, and maintainability.

Requirement	Description
Multi-language support	Compile and execute programs using language-specific toolchains.
Automated testing	Run submitted programs against visible and/or hidden test cases.
Resource control	Enforce execution time, memory, process and output limits.
Verdict generation	Return a consistent result such as Accepted, Wrong Answer or Compilation Error.
Persistence	Store problems, submissions, test results and user performance data.
Analytics	Provide statistics on correctness, attempts, execution time and progress.
Security	Isolate untrusted code and restrict unauthorized system access.

IV. PROPOSED SYSTEM ARCHITECTURE

The proposed architecture is organized into six logical layers: presentation, application/API, submission manager, judging engine, persistence, and analytics. The presentation layer provides the coding interface and result dashboard. The API layer authenticates users and validates submissions. The submission manager creates an immutable job containing source code, language, problem identifier, and resource limits.

The judging engine contains a language configuration registry, compiler manager, sandbox manager, test-case runner, output comparator, and verdict generator. The language configuration maps each supported language to its compiler/interpreter, compilation command, execution command, file extension, and default resource limits. The sandbox manager launches the program with restricted permissions and resource limits.

The persistence layer stores problem metadata, test cases, submissions, compilation logs, execution results, and aggregate statistics. The analytics layer transforms these records into student, problem, course, and language-level indicators.

Component	Primary Function
Web Interface	Source editing, problem display, submission and result visualization.
API/Controller	Authentication, validation, routing and job creation.
Submission Manager	Queues submissions and tracks their lifecycle.
Compiler Manager	Selects and invokes the language-specific toolchain.
Sandbox/Runner	Executes untrusted programs with resource restrictions.
Verdict Engine	Compares output and assigns the final verdict.
Database	Stores problems, submissions, results and analytics.
Analytics Module	Generates performance and learning indicators.

V. METHODOLOGY

The judging process begins when a learner submits source code. The server validates the problem identifier, selected language and submission size. A submission record is created with a unique identifier and placed in a queue. The worker retrieves the job and prepares an isolated execution environment.

For compiled languages, the compiler manager invokes the configured compiler. Compiler output is captured and

returned as a Compilation Error when compilation fails. For interpreted languages, an equivalent validation and execution preparation stage is performed. Successful submissions proceed to test execution.

Each test case is executed independently or within a controlled batch. Standard input is supplied to the process, while standard output and standard error are captured. The runner records wall-clock execution time, memory consumption, exit status, and output size. Execution is terminated if a configured limit is exceeded.

The comparator normalizes output according to the problem specification and compares the produced output with the expected output. Depending on the result, the verdict engine can generate Accepted, Wrong Answer, Time Limit Exceeded, Memory Limit Exceeded, Runtime Error, Compilation Error, or Output Limit Exceeded. The final submission status is derived from the complete set of test cases.

A simplified judging algorithm is shown below.

Algorithm 1: Automated judging workflow

1. Receive source code, problem ID, language and user ID.
2. Validate submission and create a job record.
3. Select the language-specific compiler/interpreter.
4. Compile or prepare the source code.
5. If compilation fails, return Compilation Error.
6. Create an isolated execution environment.
7. Execute the program for each test case under resource limits.
8. Capture output, exit status, time and memory usage.
9. Compare actual and expected outputs.
10. Assign a verdict and persist test-level results.
11. Update user and problem analytics.
12. Return the final result to the client.

VI. IMPLEMENTATION CONSIDERATIONS

A practical implementation can be built using a web framework such as Flask, Django, Spring Boot, or Node.js for the service layer and a relational database for persistent data. A queue-based worker architecture is recommended so that code execution does not block web requests. The compiler and runtime configuration should be maintained separately from application logic.

The system should support deterministic execution. Each judging job should use a temporary working directory, controlled environment variables, fixed input data, and explicit resource limits. Temporary files should be deleted after execution. Compiler and runtime versions should be recorded to support reproducibility.

The platform should never execute untrusted submissions directly with the privileges of the web-server process. Isolation may be implemented using containers, operating-system namespaces, seccomp policies, restricted users, cgroups, or equivalent mechanisms. Network access should normally be disabled during judging. File-system access should be limited to the temporary execution directory.

A submission queue also improves scalability. Multiple isolated workers can process independent jobs concurrently, while the API remains responsive. The number of workers should be controlled according to available CPU and memory resources.

VII. EVALUATION FRAMEWORK

The effectiveness of the proposed system should be evaluated using correctness, performance, scalability, reliability, and usability measures. Since actual measurements depend on the deployment environment, this paper defines an experimental framework rather than reporting fabricated numerical results.

Metric	Measurement Method
Correctness	Percentage of submissions receiving the expected verdict.
Judging latency	Time from submission acceptance to final verdict.
Compilation latency	Time required to compile a valid source program.
Execution time	CPU/wall-clock time consumed by the submitted program.
Memory usage	Peak memory consumed during test execution.
Throughput	Number of submissions processed per unit time.
Failure rate	Percentage of jobs failing due to infrastructure errors.
Scalability	Change in latency and throughput as concurrent submissions increase.

For experimental validation, a benchmark suite can contain problems from basic input/output to graph algorithms, dynamic programming, and data structures. Each problem should include multiple test cases covering normal,

boundary, and stress conditions. The same problem set can be submitted in each supported language to compare compilation overhead and runtime behavior.

A useful comparison is between manual evaluation and automated evaluation for a fixed set of assignments. Time spent by instructors on compilation, execution, checking, recording, and report preparation can be measured before and after automation. Student usability can additionally be assessed through a structured questionnaire covering ease of submission, clarity of feedback, response time, and usefulness of analytics.

VIII. DISCUSSION

The proposed architecture provides several benefits. First, the language registry allows new programming languages to be added without redesigning the entire platform. Second, separating the web service from the execution workers improves reliability because resource-intensive programs are isolated from user-facing requests. Third, storing test-level results enables richer analytics than a single final verdict.

For educational use, the collected data can reveal common errors and difficult problems. Instructors can identify learners who repeatedly receive compilation errors, exceed time limits, or fail particular concepts. Students can use submission history to observe improvement over time.

The main limitation is that secure code execution is operationally complex. A weak sandbox can expose the host system. Compiler toolchains also require maintenance and version management. Another limitation is that output-based testing cannot always determine whether a program is pedagogically appropriate; a correct output does not necessarily imply good algorithmic design.

The system also requires carefully designed test cases. Insufficient tests may accept incorrect programs, while excessively large test sets may increase judging latency. Therefore, test-case quality is a critical part of the overall assessment process.

IX. FUTURE ENHANCEMENTS

Future work can extend the platform with AI-assisted feedback that explains compilation and runtime errors without revealing complete solutions. Automated test-case generation can be used to identify edge cases and strengthen problem validation. Code similarity analysis can assist in detecting suspiciously similar submissions while reducing false positives through structural comparison.

Further enhancements include adaptive difficulty estimation, personalized problem recommendations, code quality metrics, learning-outcome mapping, instructor dashboards, plagiarism analysis, and predictive analytics. A distributed execution cluster can also be investigated for large-scale courses and programming contests.

X. CONCLUSION

This paper presented an intelligent online judge architecture for multi-language automated programming assessment. The proposed system integrates submission management, language-aware compilation, sandboxed execution, test-case comparison, verdict generation, persistence, and learning analytics. The modular design supports both competitive programming and academic laboratory environments.

References.

[1] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, “Review of recent systems for automatic assessment of programming assignments,” in Proc. 10th Koli Calling Int. Conf. Comput. Sci. Educ., 2010, pp. 86–93.

[2] K. M. Ala-Mutka, “A survey of automated assessment approaches for programming assignments,” Computer Science Education, vol. 15, no. 2, pp. 83–102, 2005.

[3] A. P. Smith, “A framework for the automated assessment of programming exercises,” in Proc. IEEE Frontiers in Education Conf., 2014.

[4] D. S. Touretzky, “Teaching and learning programming through automated assessment,” in Proc. ACM Technical Symposium on Computer Science Education, 2010.

[5] Docker Inc., “Docker Documentation,” 2026. [Online]. Available: https://docs.docker.com/

[6] Python Software Foundation, “Python Documentation,” 2026. [Online]. Available: https://docs.python.org/3/

[7] J. C. Foster, “Automated grading and feedback for programming assignments,” in Proc. ACM Conf. Innovation and Technology in Computer Science Education, 2013.

[8] R. Paiva, I. Ferreira, and D. C. Leite, “Automatic assessment of programming exercises,” in Proc. IEEE Frontiers in Education Conf., 2016.