CSRS - Automated Assessment

Engineering Design Review (EDR)

Date: 31/10/2024

Introduction:
For institutions teaching Computer Science and related fields, there are often programming assignments provided as part of the curriculum. Typically, markers of such assignments are required to go through the tedious process of manually processing submissions provided by students. This mainly involves, but is not limited to: downloading submissions, checking for plagiarism, compiling the code, executing test cases, comparing the code found in submissions with that of the skeleton code (usually provided in the assignment instruction set). This process, whilst necessary for accurate assessment, is both labour-intensive and prone to inconsistencies due to marking errors and differences in local environments. As student numbers have been shown to consistently rise over the years, and thus, the same for assignment submission volumes, it is clear that the lack of automation for these tasks creates a bottleneck in the grading workflow, impacting resource allocation and turnaround times for feedback.

This EDR proposes an automated assessment and grading feature for CSRS, with the aim of making it easier for module organisers and the likes to configure a set of predefined actions – this could include code compilation, test execution, diff-checking with the skeleton code, integrating RewriteItIn for plagiarism checking and grading against a set criteria – that are automatically triggered upon submission or when all submissions from a specific cohort are collected. This enhancement aligns with our goals for improving operational efficiency, supporting academic workflows and, as a byproduct improving overall student satisfaction - something our customers would appreciate, as their needs for reliable and scalable academic solutions continue to change.

Goals and non-goals:
- Goal: Automate assessment processing with the option for manual intervention if required
- Goal: Reduce the time spent on routine assessment tasks by at least 50%
- Goal: Standardise assessment steps across all submissions to reduce variability in grading and ensure all submissions are evaluated under identical conditions
- Goal: Ensure the feature can handle high volumes of submissions
- Non-goal: Provide advanced code analysis, such as performance benchmarking or deep semantic error checking
- Non-goal: Cover all possible languages or frameworks

Design Overview:
The automated assessment feature will be composed of a series of backend services and an interface for configuration and results management. These services will handle the entire lifecycle of the assessment pipeline: from initial submission to test execution and result

collation. Each component, especially where directly involved in assessment, should be designed for flexibility, allowing assignment hosts to customise each step.

Existing data:
We anticipate the following data, especially submissions to be the most relevant when developing the pipeline. If possible, we will integrate this into the assessment feature, or refactor the current submissions system so that the system works properly.

Table	Relevant fields	Relevance
assignment	Assignment_id (pk), module_id(fk), host_id(fk), due, instructions	Stores assignment instructions, who it was set by (host) and due time
assignment_submission	Submission_id (pk), assignment_id (fk), student_id(fk), contents, time, late, mark	Stores submissions and who they were submitted by, time, mark, and lateness (which can be used to apply penalties)

Existing functions:
- Assignment page, and submissions collection via CSRS
- Role based access, markers have access to all submissions provided by students

A general outline of the system is as follows:

1. Submission handler
    a. Intercepts and processes the submissions from the CSRS interface, and triggers the assessment pipeline if configured to do so.
    b. (Optionally) checks plagiarism with existing RewriteItIn
    c. Integrates with CSRS's existing API, ensuring compatibility with its backend
2. Configuration Module
    a. The configuration module should allow module organisers to define specific steps for each assignment - this includes the configuring and ordering of the set of steps. For example, setting the programming language, providing a VM snapshot or system image for manual intervention, compilation instructions, test execution against marking criteria, and diff checking.
    b. Configurations are stored in a secure, dedicated configuration database and are tied to assignment IDs, allowing different modules to have unique setups.
3. Execution Engine
    a. Execution engine runs each step in the defined assessment pipeline
    b. W.r.t. Scalability, each task can be run within a containerised environment, allowing horizontal scaling to handle multiple submissions concurrently
    c. Job scheduling and load balancing additionally implemented to optimise latency and performance

4. Result Management (Interface)
    a. After processing, the results are stored and displayed alongside the submission, providing markers with access to the post-process evaluation, and the option to add qualitative feedback for each submission
    b. Results are accessible through a results API endpoint that the CSRS front-end can retrieve and display in a structured format
5. Error handling and logging
    a. Logs are stored to facilitate debugging and auditing.
    b. Logs will be made for each step of the assessment process in the case of failure to execute or unexpected results

Alternatives

Real-time processing
Allows for instant updates to the results page as assignments are marked on a job by job basis. However, this will significantly increase the server load, especially near submission deadlines. Additionally, markers/module organisers have no opportunity to fix any errors in the pipeline before results are finalised and are viewable by the student.

Manual Trigger by Marker
This option would allow markers to manually initiate assessment steps. Although this provides the option for debugging any errors within the pipeline, and allowing markers to selectively apply assessments to tailor feedback, it reintroduces the risk of variability in grading, risk of marker error, and ultimately defeats the purpose of automating assessment.

Milestones

Milestone 1:

Focus on establishing the foundational backend infrastructure for tracking and processing submissions. This includes creating the database schema to manage submission data, assessment configurations, and result tracking, as well as implementing secure file storage for handling submission files. An API will be developed to intake student submissions, logging metadata and consequently triggering the compilation step. Once these components are tested and confirmed to be reliable, the foundation will be evaluated to determine if subsequent development should proceed.

Develop the core backend infrastructure and set up databases for tracking and processing initial submissions. This includes setting up secure file storage for submission files. Build an API for this, and ensure that a simple compilation workflow can be established after handling the submissions. Test this, and if it is reliable and works with some security features implemented and shown to be working, then continue to further development.

Milestone 2:

A testing suite will be developed to run predefined unit tests on student submissions, capturing logs for errors and exceptions. A code comparison service will also be

implemented to analyse modifications between student code and provided skeleton code (could use some git functionality here). The database schema will be expanded to store detailed test results, compilation errors, and comparison results. A configuration interface will be created to allow module organisers to specify testing parameters, or upload test scripts and skeleton code. Integration tests will then be conducted to validate the system's functionality and determine readiness for the next phase.

Milestone 3:

Develop the marker dashboard and initial UI, which will provide hosts with an interface to review submission results, manage assessment configurations, and filter flagged submissions. The dashboard will display submission outcomes, test results, and flagged issues. A temporary UI will be presented to gather initial feedback from users; if the feedback is positive, development will proceed to final testing and deployment.

Milestone 4:

Perform end-to-end system testing and complete documentation to finalise the system for production. User documentation for markers and module organisers will be created, along with technical documentation for internal use, covering API use details. The deployment process will follow a staged approach, with a pilot phase to gather final user feedback and make any necessary adjustments. After reviewing pilot feedback and making final adjustments, the system will be prepared for full deployment.

Optional Milestone 1 will implement a notification feature that alerts markers when all assignments from a cohort have been marked, ensuring they are promptly informed of job completion.

Optional Milestone 2 will introduce real-time feedback for students immediately post-submission, providing them with initial results while incorporating safeguards to minimise risks of academic misconduct.

Dependencies
UI Team: Will need to design the marker dashboard (including configuration and results interface) and make changes to existing views to integrate new features
Database Team: Required for setting up new database tables and migrating old ones where needed for submission handling. They will also be needed to optimise these for bulk requests.
Notifications Team: Responsible for creating dynamic notification templates and ensuring they are sent in a timely and accurate manner to the correct people
Configuration Team: Will need to liaise with the UI team and ensure an intuitive configuration interface is built according to the needs of faculties/course organisers so that the pipeline is flexible enough to meet the needs of most users
Legal Team: Will review legal agreements with customers to ensure that the new features comply with regulatory requirements and do not expose the organisation to any new liabilities.
Containerisation Team: Will be responsible for setting up the containerisation infrastructure that compiles, executes, and tests the code.

Security Team: Ensure security is implemented to the best and latest standards to avoid attacks. Ensure that sandboxing is implemented correctly. Ensure that comprehensive audit logging mechanisms are in place and there is strict adherence to data protection regulations, and general institution regulations.

Cost

We do not anticipate any significant increase in operating costs as a result of the proposed enhancements to the assessment pipeline and notification system (with current assumed usage figures). The core functionalities, including the automated assessment steps and notification alerts for markers, will be executed within the existing infrastructure and should not substantially increase resource consumption.The job for compiling and marking assignments will be scheduled to run during off-peak hours, minimising any potential impact on system performance. Additionally, notification emails will be sent only once all assignments from a cohort have been marked, which will be a periodic task that can be managed effectively within our current email infrastructure. However, as the number of users and submissions increases, there may be additional cloud costs or server costs associated with scaling our infrastructure. This includes the need for more compute resources to handle increased job loads, as well as potential costs for storage if submission data grows significantly. If performance issues arise, we can optimise the frequency and resource allocation for these processes to ensure efficiency without incurring extra costs.

Privacy and security concerns

All student data used in the assessment pipeline and notification system is already accessible to authorised markers and module organisers, eliminating the need for new roles or permissions. To prevent unauthorised access and ensure security, we will adhere to established best practices for safeguarding against potential vulnerabilities. We will employ sandboxing techniques to isolate the execution of submitted code, preventing malicious activity. Proper audit logging will be established to monitor access to sensitive routes, utilising our existing audit event system, while maintaining strict data privacy and compliance with regulations such as GDPR

Risks

Risk	Mitigations
Malicious code execution	Implement sandboxing to isolate code execution and restrict access to system resources.
Performance Bottlenecks	Conduct load testing during development to identify and address bottlenecks.
Data Privacy Breaches	Strict access control will be enforced. We will perform routine security audits and penetration testing to find and fix vulnerabilities. .

User adoption resistance	Comprehensive training and support resources will be provided. Actively use feedback during pilot phases to address concerns and enhance UX
Results may not be reflective of submission quality (e.g, using the correct structure, syntactic correctness)	Implement a hybrid approach and deploy the feature in phases to address differences in manual against automated marking and configure the pipeline accordingly

Supporting material

- Refer to for good security practices: OWASP (2021). *OWASP Top Ten*. [online] Owasp.org. Available at: https://owasp.org/www-project-top-ten/.
- Chen, J.C., Whittinghill, D.C. and Kadlowec, J.A. (2010), Classes That Click: Fast, Rich Feedback to Enhance Student Learning and Satisfaction. Journal of Engineering Education, 99: 159-168. https://doi.org/10.1002/j.2168-9830.2010.tb01052.x