O.W.C.A

Jacques Klooster - u23571561
Aidan McKenzie - u23545080
Reece Jordaan - u23547104
Richard Kruse - u23538172
Luke Gouws - u23668882

Table of Contents

1. Introduction

Automated assessment systems have become indispensable in computer science education, streamlining the grading process and providing students with rapid feedback. At the University of Pretoria's Computer Science Department, our existing tool, FitchFork, has served this role admirably for several years. However, with the rapid evolution of both programming paradigms and academic integrity challenges, there's a clear need to rebuild and expand its capabilities. Our capstone project aims to not only replicate all of FitchFork's current features but also to integrate cutting-edge research from GATLAM, a new AI-powered engine for in-depth code evaluation and personalised feedback.

The motivation behind this rebuild is twofold. First, by working on "our nemesis," we get hands-on experience in software engineering, AI, and formal methods, all while directly contributing to an essential departmental tool. Second, by incorporating advanced features like plagiarism and AI-generation detection, we're ensuring that the system remains robust, fair, and scalable in the face of ever-more sophisticated student submissions.

2. User Stories / User Characteristics

Student User Stories

1) View Modules
   As a student,
   I want to view my enrolled modules and see all the details associated with them.

   Acceptance Criteria:
   - The student can see all enrolled modules
   - The student can click on a module and see a detailed description of it

2) View Assignments
   As a student,
   I want to view and download the assignment instructions so that I know what I need to do.

   Acceptance Criteria:
   - The student can select an assignment from a list in a module
   - The system allows file downloads

3) Submit Assignments
   As a student,
   I want to submit my assignment for a module,
   so that my code can be automatically marked.

   Acceptance Criteria:
   - The student can select an assignment from a list in a module
   - The system allows file uploads
   - The submission is stored and queued for auto-marking
   - The student is notified once marking is complete

4) View Feedback and Marks
   As a student,
   I want to see my marks and feedback after submitting my assignment.
   so that I can understand how I performed and improve.

   Acceptance Criteria:
   - The system displays marks out of a total score
   - Feedback includes test results, terminal output and comments
   - Students can view previous submissions and their respective feedback

Tutor User Stories

1) View Student Submissions
   As a tutor,
   I want to view student submissions for an assignment,
   so that I can assist students with their issues.

   Acceptance Criteria:
   - Tutors can access a list of student submissions for modules they are assigned to
   - Each submission shows the student's name, timestamp, and status
   - Tutors cannot modify the student's code or marks

2) View Terminal Output
   As a tutor,
   I want to see the terminal output of a student's submission,
   so that I can better understand what went wrong and provide help.

Acceptance Criteria:

- Tutors can view the console output (stdout/stderr) from a student's code execution
- Output includes test results and any runtime errors
- Tutors cannot rerun or modify the execution

3) Compare Terminal Output
   As a tutor,
   I want to be able to compare a student's output to the memo output in an easy to manage fashion

   Acceptance Criteria:
   - Tutors can view the student output next to the memo output to compare them easily

Lecturer User Stores

1) Manage Assignments
   As a lecturer,
   I want to create an assignment in a module,
   so that students can submit their work for marking.

   Acceptance Criteria:
   - The lecturer can choose the module and create a new assignment
   - Students linked to the module can see the new assignment

2) Submit Assignment Code
   As a lecturer,
   I want to upload my assignment code solution,
   So that the system can automatically generate a testing framework for student marking.

   Acceptance Criteria:
   - The lecturer can upload memo zipped code
   - The lecturer can upload makefile zipped code
   - The lecturer can upload main zipped code
   - The lecturer can upload a config file

3) Edit Mark Allocator

As a lecturer,
I want to edit the generated mark allocator so that I can specify how many marks get allocated for specific correct code implementation.

Acceptance Criteria:

- The Mark Allocator gets generated and displayed for the lecturer
- The Mark Allocator can be edit and saved

4) Choose Testing Algorithm
   As a lecturer,
   I want to choose between RNG, Code Coverage or GATLAM algorithm, so that the appropriate testing framework is generated for student submissions.

   Acceptance Criteria:
   - The lecturer is presented with the three algorithm options when creating an assignment
   - The system applies the selected algorithm to generate the testing framework

5) Detect Plagiarism
   As a lecturer,
   I want to view plagiarism reports for assignments,
   so that I can identify and address cheating among students.

   Acceptance Criteria:
   - A plagiarism report is available for each assignment after student submissions
   - The report highlights matching submissions and a similarity score

Admin User Stories

1) Access All Tools
   As an admin,
   I want access to all features available to students, tutors and lecturers, so that I can manage and oversee the entire system.

   Acceptance Criteria:
   - Admins can view and manage all modules, users, assignments, and submissions

2) Manage User Permissions
   As an admin,
   I want to change the roles and permissions of any user,
   so that I can ensure correct access levels are assigned.

   Acceptance Criteria:
   - Admins can view a list of all users with their current roles
   - Admins can change a user's role

3) Manage Modules
   As an admin,
   I want to create modules,
   so that I can assign new lecturers to them

   Acceptance Criteria:
   - Admins can create modules and assign lecturers to them

3. Functional Requirements

FR1: User Authentication and Roles

- **FR1.1**: System shall support user authentication for Admin, Lecturer, Tutor, and Student roles.
- **FR1.2**: System shall restrict access to features based on user roles.

FR2: Module Management

- **FR2.1**: Admin shall be able to create new modules.
- **FR2.2**: Admin shall be able to edit module details.
- **FR2.3**: Admin shall be able to delete modules.

FR3: Assignment Management

- **FR3.1**: Admin/Lecturer shall be able to create assignments for a module.
- **FR3.2**: Admin/Lecturer shall be able to edit assignment details.
- **FR3.3**: Admin/Lecturer shall be able to delete assignments.

FR4: Marking Script Management

- **FR4.1**: Admin/Lecturer shall be able to create marking scripts using:

    - **FR4.1.1**: GATLAM
    - **FR4.1.2**: Random Number Generator
    - **FR4.1.3**: Coverage-based algorithm
    - **FR4.1.4**: Manually
- **FR4.2**: Admin/Lecturer shall be able to delete marking scripts.
- **FR4.3**: Admin/Lecturer shall be able to edit marking scripts.
- **FR4.4**: Admin shall be able to upload custom interpreter
- **FR4.5**: Interpreter shall translate marking script into executable code depending on the marking script used
- **FR4.6**: Admin/Lecturer shall be able to set marking to manual mode

FR5: Mark Allocator

- **FR5.1**: A Mark Allocator shall be generated from the memo output
- **FR5.2**: Admin/Lecturer can edit the mark allocator
- **FR5.3**: Mark Allocator shall determine the weight of correct code for marking

FR6: Code Submission

- **FR6.1**: Students shall be able to upload their code files.
- **FR6.2**: Students shall receive marks and feedback for their submission

FR7: Reporting and Statistics

- **FR7.1**: System shall provide live statistics per assignment.
- **FR7.2**: Statistics shall be available as downloadable reports.
- **FR7.3**: Statistics shall be displayed in graph form.

FR8: Code Viewer and Runner

- **FR8.1**: System shall allow viewing code without downloading.
- **FR8.2**: System shall allow running code without downloading.
- **FR8.3**: System shall show output and stack trace of execution.

FR9: Execution Environment

- **FR9.1**: Student submissions shall be run in containerized environments.
- **FR9.2**: Student output shall be matched to marker output outside of the container.

FR10: Plagiarism Detection

- **FR10.1**: System shall support plagiarism detection per assignment.
- **FR10.2**: System shall compare ASTs before invoking MOSS.
- **FR10.3**: Plagiarism shall optionally be displayed in graph form

FR11: AI Assistance

- **FR11.1**: System shall provide AI-generated summaries of exceptions.
- **FR11.2**: System shall provide AI-generated summaries of incorrect outputs.

FR12: Gamification and Progression

- **FR12.1**: System shall support achievements and other gamified elements.
- **FR12.2**: System shall support unlocking tasks by completing previous tasks.

FR13: Grading System

- **FR13.1**: System shall calculate grades per assignment.
- **FR13.2**: System shall allow different grade weights per task.
- **FR13.3**: System shall display grades to students.
- **FR13.4**: System shall support time and space complexity analysis.

FR14: Submission Rules

- **FR14.1**: Admin shall be able to configure:
    - **FR14.1.1**: Submission deadlines (date and time)
    - **FR14.1.2**: Late submission policy
    - **FR14.1.3**: Submission count limit (including infinite)

FR15: Security

- **FR15.1**: System shall restrict student access to memo content.
- **FR15.2**: System shall isolate containers to prevent memo leakage.

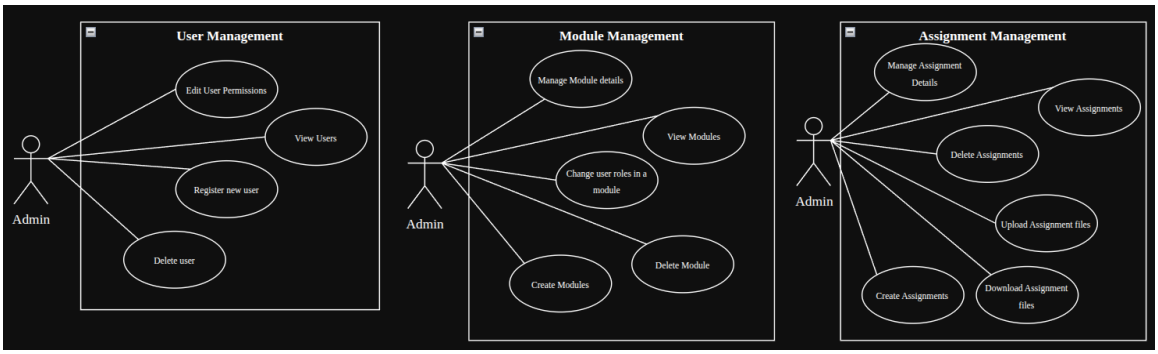
FR16: Support System

- **FR16.1**: System shall have a ticketing system (Feature Flag enabled).

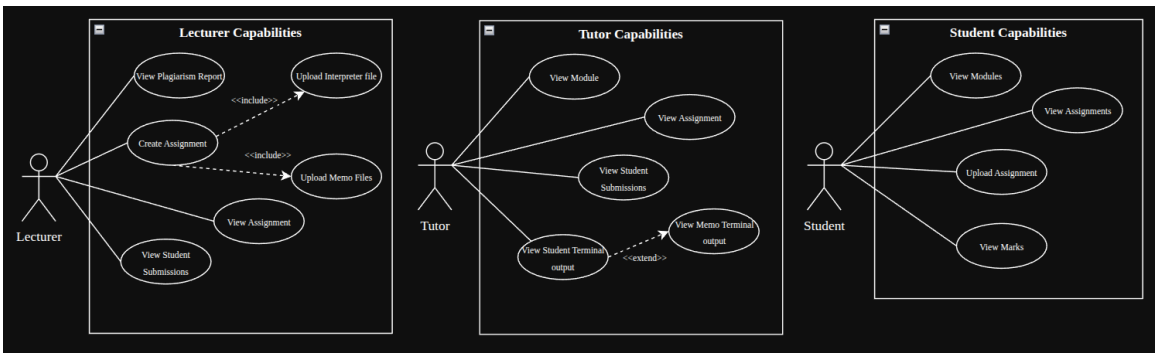
4. Domain Model

5. Use Cases

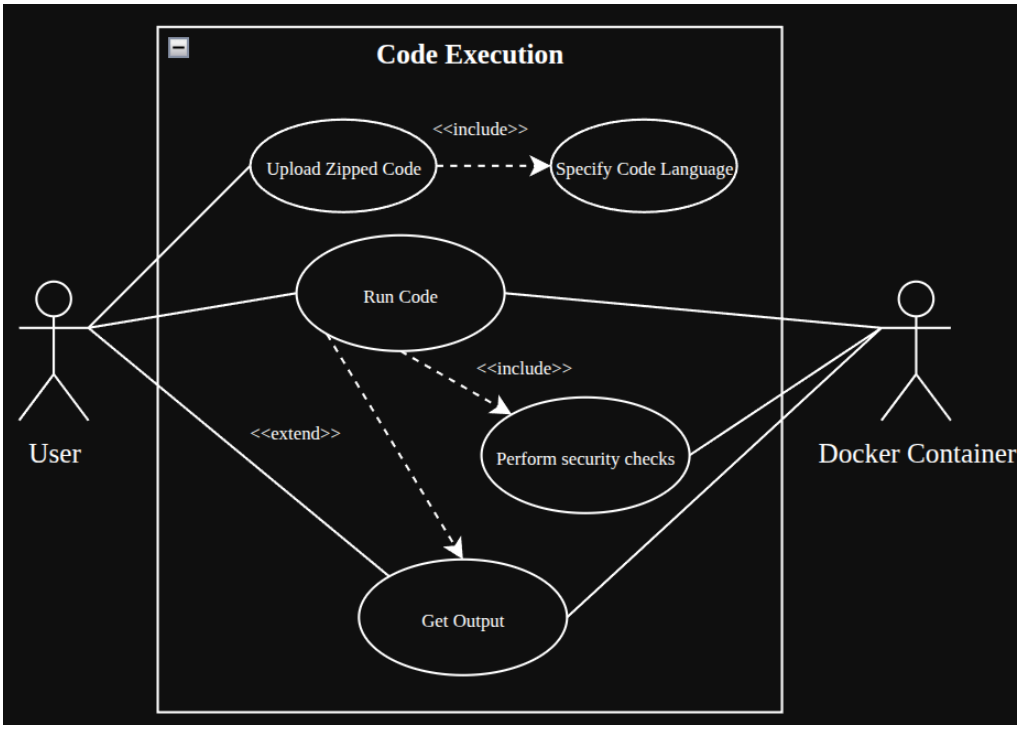
Admins



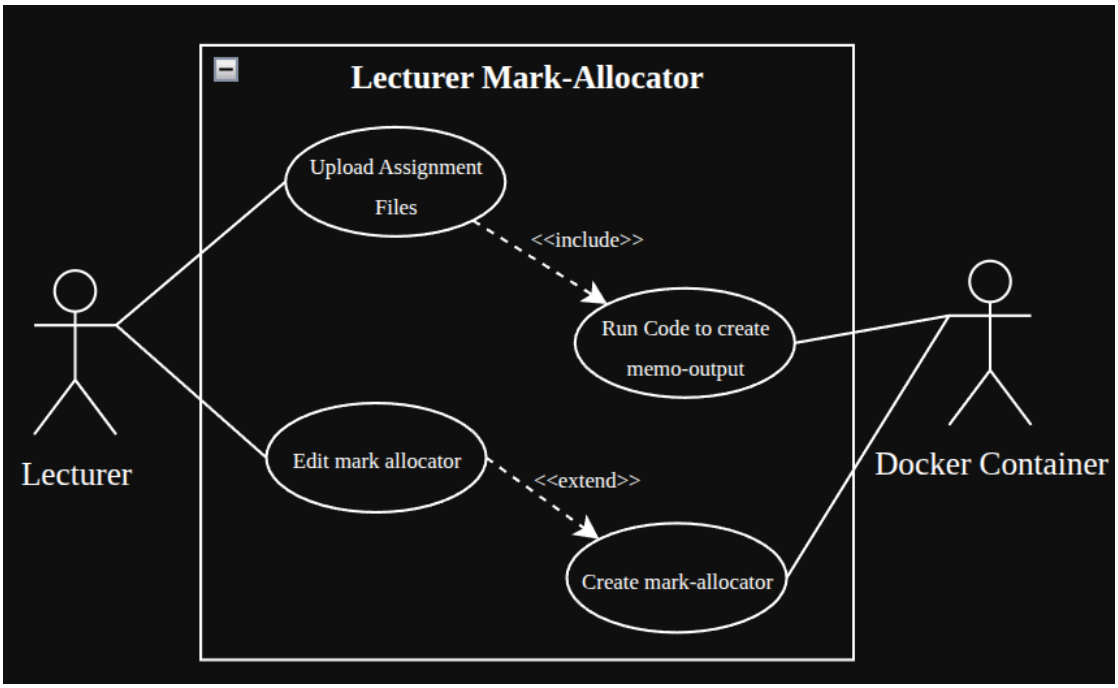
Lecturer/Tutor/Student



Code Execution



Mark Allocator Usage



3.6.1 Architectural Design Strategy

The design strategy we used is **Use-Case driven** design,

- The system is structured around clear functional requirements involving the multiple actors (lecturers, admins, students)
- Each core flow is mapped to concrete routes and services
- This helps us create a working product with the main functionality we require as soon as possible as we focus on actual user stories and applications

---

3.6.2 Architectural Styles

For our architectural style we used a **Layered Architecture** which is divided into three primary layers

- **Presentation Layer**: This consists of the React-based single-page application. It is responsible for rendering the user interface, handling routing, capturing interactions, and communicating with the backend via HTTP requests. This layer

is completely decoupled from the business logic and data management layers

- **Application Layer:** This is implemented using Rust and the Axum web framework. It acts as the orchestration layer of the system, it routes all the requires, enforces authorization and RBAC via guys and delegates tasks such as grading, code execution, and plagiarism detection to specialized internal components.

- **Data Layer:** The system uses SQLite for storage, interfaced through SeaORM and SQLx for type-safe and asynchronous database operations. The layer is abstracted from the rest of the system and has a single access point via a Singleton, making the code highly maintainable

---

3.6.3 Architectural Quality Requirements

QR1: Performance

- **QR1.1**: The system must have an average code submission time of less than 10 seconds
- **QR1.2**: The system must be able to process and store up to 250 code submissions over a 12-hour period without performance degradation
- **QR1.3**: Graceful timeouts, if a job takes too long it is ended

QR2: Scalability

- **QR2.1**: The system architecture must support horizontal scaling of compute resources (e.g., worker nodes or containers) to handle increased load.
- **QR2.2**: Stateless API layer

QR3: Availability

- **QR3.1**: The system must maintain at least a 99.5% uptime during the semester
- **QR3.2**: The system must include real-time health monitoring for core components

QR4: Usability

- **QR4.1**: The system must provide a user interface on mobile devices
- **QR4.2**: The system must provide an accessible user interface on differently scaled viewports
- **QR4.2**: The system must provide users with active responses for loading, errors or successful execution

QR5: Security

- **QR5.1**: Student code execution must happen in sandboxed containers
- **QR5.2**: Communication must be secured with TLS
- **QR5.3**: Passwords and student information must be protected via hashing
- **QR5.4**: System sections must only be accessible to the appropriate roles

3.6.4 QR Justification

- **QR1.1**: Due to the amount of student submissions and the limited containers submission turnaround must be fast
- **QR1.2**: Based on class sizes and estimated peak daily usage, this amount must at least be able to be handled
- **QR1.3**: Prevents runaway/infinite loops in student code from monopolizing resources
- **QR2.1**: Deadlines produce load spikes, meaning that workers need to be scalable when demand is high and return to normal when it is low
- **QR2.2**: Statelessness makes it easy to add/remove API servers later
- **QR3.1**: Students need predictable access during the semester, and to reduce load on tutors and lecturer in the event of an extension
- **QR4.1**: Many students may access their profile using a tablet, it also makes it much easier for a tutor during a session to look at a student's mark if it is available on mobile
- **QR4.2**: Students use varied devices, responsive design avoids broken layouts
- **QR4.3**: Immediate, clear feedback improves user experience for both students and lecturers, and prevents accidental duplicate submissions
- **QR5.1**: Student code must not compromise the host system or other students' submissions
- **QR5.2**: Prevents eavesdropping, especially across a public network such as on campus
- **QR5.3**: Protects students passwords and privacy in compliance with POPIA
- **QR5.4**: Ensures lecturers, tutors, and students only see the functions appropriate to their roles, preventing unauthorized access and tampering with data

---

3.6.5 Architectural Design and Pattern

We adopted the **Model-View-Controller** architectural pattern as it provides a clean and well-understood separation of concerns, the system is easy to understand and maintain. In the context of our project, MVC is applied most directly in the backend. Although we are not building a traditional monolithic web application, the principles of MVC map well onto our API-driven architecture.

- **Model:** This represents the database entities, managed via SeaORM. It encapsulates business entities such as users, modules, assignments, submissions, and tasks. All interactions with the database are centralized here.

- **View:** In a conventional web app, this would generate HTML. In our system, the equivalent role is played by the JSON API responses returned to the frontend. This allows us to independently evolve the backend from frontend changes.

- **Controller:** The controller layer is comprised of Axum route handlers. These endpoints handle incoming requests and invoke the appropriate business logic.

3.6.6 Architectural Constraints

The system must adhere to several constraints across business, technical, operational, and legal domains. Which have directly affected our architectural decisions and technology choices.

From a **business and scheduling perspective**, the functionality of the platform must align with the fixed due dates given by the university. Demos and key deliverables are fixed with no allowance for timeline extensions, meaning, we were constrained to create a minimum viable product and did so by prioritizing use cases.

In terms of the **development process**, the engineering team is completely new to the Rust programming language. As such, extensive documentation and trial and error were needed, losing valuable time. The development process must also be fully automated, leveraging CI/CD pipelines to ensure repeatable and tested deployments.

From a **legal and compliance** standpoint, the system must comply with the POPIA, requiring all student data to be encrypted and stored, and never leave the database in such a way that it can be traced back to said student (or at all), additionally, data must be encrypted both in rest and in transit. Transport security must use current cryptography standards

The **technology stack** is fixed with Rust and Axum web framework for the backend, SQLite as the primary data store, and RESTful APIs for system communication. This constraint allows us to decide how the system is modularized and how communication is structured

Security is another core requirement. All code must be executed in secure sandboxes to prevent malicious code execution, and RBAC must be enforced at both the infrastructure and application levels to constrain user capabilities.

Lastly, there is the constraint of **scalability and availability**, due to the importance of Fitchfork for assignments we need a system that supports a high uptime, specifically a 99.5% uptime during the semester. Additionally during high demand periods the system should be able to horizontally scale to handle it.

3.6.7 Technology Choices

Rust: The language we used for the backend is Rust, and we did so for the following reasons

- **High Performance:** Comparable to C++, with type safety. Since we are going to have computationally intense backend operations (such as code execution, plagiarism mapping, automated marking), all with limited resources we needed a language that could handle it without intensive overhead of a garbage collector

- **Strong Concurrency Model:** Rust's ownership model and fearless concurrency make it ideal for handling concurrent operations without facing any hard to debug conditions

- **Modern tooling Ecosystem:** Libraries like Axum, Tokio, and SeaORM enable easy development of asynchronous, type-safe RESTful APIs, which form a core part of the system architecture

React:

- **Large Ecosystem:** React benefits from an immense community that provides free third-party libraries and tools for components, which streamlines developments.

- **Familiarity:** The team members working on the frontend are extremely familiar with React which reduces any lost time for learning the framework

- **Seamless integration with REST APIs:** React integrates cleanly with the Rust/Axum backend via fetch/Axios, enabling a completely decoupled backend

- **Component Reusability:** The modular nature of create allows components to be reused which makes UIs consistent and easy to build as a project evolves.

Docker:

- **Student Code:** Docker helps contain student code and prevent malicious execution

- **Scaling:** As demand increases containers can simply horizontally scale to keep up with demand
- **Parallel Execution:** In addition, since containers run in parallel (and usually on different cores) containers themselves can run in parallel from each other which inherently supports faster marking

SQLite

- **Lightweight:** Minimal footprint and setup which makes it easy to integrate into the Rust backend
- **Single Node Deployment:** Easy to set up for our specific one node setup (where are PostgreSQL would not be)
- **ACID Compliance and Transactional Integrity:** Without requiring a separate database server, which reduces overhead

TLS

- **Confidentiality:** Of all HTTPS requests is ensured between client and server
- **Man-in-the-Middle:** Prevents against eavesdropping
- **Compliance:** TLS is needed when handling sensitive user data

Argon2 Hashing

- **Brute-force Protection:** Provides protection from brute forcing by purposefully delaying hashing generation if the passwords were ever to be attained
- **Configurable:** Allows the hashing process to not be insanely intensive but still protect the underlying data
- **Reliable:** Recommended by OWASP and regarded as one of, if not the, best hashing algorithm for password storage.