Cprog-ng Design Report

Supervisor: ir. D. M. Abeln

Mart Dikker
s2900556
m.dikker@student.utwente.nl

Maarten van Dort
s3174506
m.vandort@student.utwente.nl

Twan Kuipers
s3202127
t.h.kuipers@student.utwente.nl

Marcus de Lange
s3144569
m.l.delange@student.utwente.nl

Sander Marki-Pleym
s3174301
s.marki-pleym@student.utwente.nl

April 17, 2026

1 Summary

Programming in C is a critical skill for Electrical Engineering students, and providing consistent, automated feedback is essential for teaching these demanding concepts at scale. The original CProg application successfully fulfilled this role for over a decade, but eventually reached the end of its maintainable lifecycle. Accumulating technical debt, unpatchable security vulnerabilities, a lack of separation between course iterations, and an inefficient checker running on a periodic CRON job made the legacy system increasingly difficult to maintain and use.

To address these shortcomings, CProg-ng (CProg Next Generation) was developed as a complete, modern overhaul designed from the ground up to replace its predecessor. The new system leverages a decoupled client-server architecture to ensure scalability and separation of concerns. It features a responsive, component-driven frontend built with Next.js (React 19) , and a NestJS backend chosen specifically for its non-blocking, event-driven handling of I/O operations. All relational data is securely stored in a PostgreSQL database managed via TypeORM.

The system introduces several major architectural improvements over the legacy platform:

- **Secure Sandboxing:** The automated code checker pipeline was completely redesigned using NsJail. This lightweight Linux isolation tool securely compiles and evaluates student code, enforcing strict resource limits to prevent malicious attacks without the heavy performance overhead associated with Docker.
- **Flexible Course Management:** A new Canvas-inspired structure groups assignments into sets within distinct courses, fully resolving the previous system's inability to separate data between academic years.
- **Granular Access Control:** The legacy system's hardcoded roles were replaced with a highly flexible, permission-based access control model, allowing for fine-grained authorization for students, teaching assistants, teachers, and administrators.

Development was guided by a hybrid methodological framework that combined the predictable phasing of Waterfall with the collaborative and iterative practices of Agile. Requirements were elicited from the legacy system workflows and stakeholder input, then strictly prioritized using the MoSCoW method. To ensure fairness and reliability, the system underwent rigorous automated unit and end-to-end testing.

Ultimately, CProg-ng successfully preserves the core automated grading functionality of the legacy platform while delivering a secure, scalable, and highly maintainable application deployed via Docker Compose. It establishes a robust technical foundation that resolves past vulnerabilities and is well-equipped to support the evolving needs of the curriculum.

Contents

2 Introduction

Programming in low-level, procedural languages, such as C, remains a fundamental and highly critical skill for Electrical Engineering students. It provides the necessary bridge between hardware and software, which is essential for embedded systems design, hardware interfacing, and deterministic execution. To effectively teach these demanding concepts at scale, providing students with timely, consistent, and objective feedback is crucial.

The original 'CProg' application was developed to meet this exact need. By automating the grading process of student code, it ensures deterministic evaluation while significantly reducing the administrative and grading burden on teaching staff. For more than a decade, CProg successfully fulfilled this role, undergoing numerous infrastructure upgrades and continuous bug fixes to keep up with evolving course requirements and growing student cohorts.

However, like many legacy platforms, the original CProg eventually reached the end of its maintainable lifecycle. The inevitable accumulation of technical debt meant that implementing additional updates required disproportionate amounts of time and effort. Furthermore, the underlying technology stack became outdated, leading to an increasing number of security vulnerabilities that were difficult to patch, and a user experience that no longer aligned with modern web standards.

To address these critical shortcomings and future-proof the curriculum's infrastructure, a completely overhauled system was commissioned: CProg-ng (CProg Next Generation). Designed from the ground up to fully replace its ageing predecessor, CProg-ng leverages modern software engineering principles to deliver a secure, scalable, and highly maintainable automated grading platform.

This report comprehensively documents the design, development, and implementation of CProg-ng. It details the entire engineering process, providing insight into the transition from a legacy application to a modern software solution. The following sections will explore all critical aspects of the new system, including the modernization of the front-end user experience, the scalable back-end architecture, the specific frameworks chosen, and the development cycles employed. Finally, this document will reflect on the major technical decisions made throughout the project, the roadblocks encountered, and how they were overcome to deliver the final product.

3 Methodology

This section outlines the methodology used to guide the project, ensuring effective collaboration and a high-quality end result. It begins with the overarching methodological framework, followed by the methodology used during the system specification and design. After this, the implementation process is covered, followed by the testing and validation approach. Finally, the deployment strategy is discussed.

3.1 Methodological Framework

As overarching methodology, a hybrid method was used, combining a phase-oriented approach with several collaborative practices from other methods. As the project's main task was to create a new generation of existing software, requirements were expected to be clearly identifiable beforehand, making a linear, Waterfall-like[1] approach suitable for guiding the main phases of the development process due to the stability of the requirements. However, in order to support team coordination and flexibility, some practices from iterative and collaborative methodologies, like Agile[2] and DevOps, were also implemented. The work was organized into small manageable units and development was supported by automated quality checks and a structured review process. This combination enabled a good balance between predictability and adaptability, thus increasing development efficiency and product quality.

3.2 System Specification

The system requirements for CProg-ng were primarily derived from the established and proven functionality of the legacy system. Because the fundamental purpose of the application remained unchanged, user stories representing the core needs of teachers and students were extracted from the existing workflow. Stakeholder input was obtained through the system owner, who acted as a representative for end-user requirements. An initial set of requirements was formulated using the MoSCoW method[3] (Must have, Should have, Could have, Won't have) allowing for a clear distinction between essential and optional functionality. These requirements were refined and finalized through iterative feedback, with a newly added emphasis on system maintainability and architectural extensibility. Ultimately, the requirements were translated into actionable task tickets to guide the development phases.

3.3 System Design

The system design phase focused on defining a robust and maintainable architecture that supports the functional and non-functional requirements identified during specification. A decoupled client–server architecture[4] was chosen to enforce a clear separation of concerns between data processing and presentation. This approach improves system modularity, scalability, and security. To plan out the system and its behaviour, standard `UML` modelling techniques were applied[5]. Design artifacts, including class diagrams and sequence diagrams, were developed to document both the static and dynamic aspects of the system. These models were used to guide the development process.

3.4 System Implementation

The implementation phase translated the system design into a working software solution following a structured and collaborative development process. Version control practices such as conventional commits and feature branches[6] were used to support parallel development and controlled integration of features. A feature-based workflow enabled isolated

development of components while maintaining overall system stability. To ensure consistency and maintain a high standard of code quality, automated code analysis and formatting tools were used to check code alongside systematic peer review. These practices helped enforce coding standards, reduce defects, and improve maintainability across the codebase.

3.5 Testing and Validation

Testing and validation were done to ensure the correctness, reliability, and robustness of the system. A multi-level testing strategy was adopted, incorporating unit testing and end-to-end testing[7]. This approach allowed individual components, their interactions, and the system as a whole to be systematically verified. Both back end functionality and frontend components were validated using automated testing techniques. Where necessary, testing scope was adapted to account for environmental constraints, ensuring that core system behaviour could be tested reliably across different development setups. Lastly, usability tests for the core workflows of the system were performed. This allowed for the evaluation of whether the system met its objectives.

3.6 Deployment

The deployment strategy was designed to ensure consistency, reproducibility, and ease of maintenance across different environments. A containerized approach[8] was used to bundle system components and their dependencies, enabling reliable execution from development to production. Deployment processes were structured to minimize manual intervention and support efficient updates, contributing to overall system stability and maintainability.

4 System Analysis

This section will show an analysis of the existing CProg system, showing its implementation and functions. After that, the short-comings of the old system, which CProg-ng should improve upon, will be highlighted.

4.1 Overview Existing System

The CProg system was created in 2013 to automatically grade assignments for the Programming in C course in the Bachelor of Electrical Engineering. It was implemented in PHP 7 and consists of two main parts:

- **Frontend.** The frontend is the web interface through which users interact with the system. Since PHP is executed on the server when a webpage is requested, the frontend is closely tied to the database. The system uses Smarty templates to render webpages and populate them with data from the backend.
- **Backend.** The backend consists of the checker, which is periodically triggered through a CRON job. When it is executed, the checker attempts to process a queue of submissions regardless of if any are present. It compiles and tests each submission in turn. After a submission has been fully processed by the checker, the result is stored in the database.

4.2 Functionality

Groups Many assignments in the course are meant to be done in a group, for this CProg has groups in place that students can join, which allows students to hand in group assignments. After joining a group, CProg does not prevent the student from leaving the group or joining another altogether. While this is possible, it is not intended and students are simply told to not do so. Similarly, there exists assignments that are meant to be done in a group, but CProg does not prevent individuals from submitting these either.

Submitting Attempts To submit a solution to an assignment, a student must upload a ZIP file containing their solution files. Assignment submissions are either individual or on behalf of a group. Once submitted, the files are added to a queue for checking. A CRON job then periodically processes the queue and runs the checker on all submitted ZIP files.

Automatic Grading The backend periodically runs a checker based on a CRON job. It automatically compiles and tests all queued submissions. If any of the submissions do not pass the tests, it will be given no points. Should the submission pass all tests, an amount of points will be awarded. This amount is based on the total points available for the given assignment and the number of attempts it took to pass. The final grade of a student is the sum of all the points earned for each assignment.

Override Results Functionality exists for teacher to override the result of a submission. They can change the result to force a pass, a fail, or ignore the submission entirely. If a submission is overridden by accident or incorrectly, it can be undone. Upon overriding a submission, CProg will recalculate the amount of points awarded, and then recalculate the final grade to reflect the updated assignment status.

Export Grades In the CProg system, it is possible for teachers to export the calculated grades of the students as a file. This file can then be uploaded to other systems like Canvas to easily move the grades.

4.3 Improvements/Problems

Hard to maintain The CProg system is difficult to maintain and extend due to the way it has been upgraded over time. CProg was written in PHP many years ago, as such the system has undergone a few upgrades, though these changes were not always supported by proper version-control practices or systematic refactors. As a consequence, the codebase contains multiple sections of large commented-out code blocks and other remnants of earlier implementations. This has made navigating the codebase rather difficult and the amount of time and effort required for any changes let alone overhauls, has only grown over the years. In addition many important values are hard-coded rather than managed using configuration files, which reduces flexibility and makes updates more prone to errors. CProg also lacks basic robustness measures, such as automatically creating missing directories when needed. Altogether, these issues sum up to a fragile system that is hard to upgrade and more time consuming to maintain.

Hard to change Assignments Assignments are difficult to modify since they are configured through multiple files inside the system rather than through a centralized and structured mechanism. As a result, even small changes often require changes in multiple places. This increases both the effort involved and the chance of introducing mistakes. The problem is made even worse by the lack of separation between course iterations, since changing an assignment can also affect submission from previous years. Thus, any modification has to be made carefully and be documented thoroughly, adding additional mainenance overhead.

Non-standard Login Method The student login mechanism was implemented in ad hoc fashion and is not based on widely used authentication standards, especially given modern standards. This made for a less robust and harder to maintain approach. Additionally the convoluted implementation makes it harder to assess the overall security.

No Separation between Iterations A significant limitation of CProg is that the different iterations of the course are not clearly separated. Data from one year remain partially mixed with data from earlier years. This makes long term retention within the same system impractical. To start a new iteration, the existing data must first be manually migrated to separate storage. This creates unnecessary and recurring work for the maintainers.

Checker on a CRON job The checker is triggered periodically through a CRON job instead of being started immediately when a submission is made. This means that students may have to wait for their submission to be processed, which delays feedback. At the same time, the checker may still be triggered even though no new submissions are pending, resulting in unnecessary system activity and less efficient use of resources.

Hardcoded Roles Access control is based on a small number of hardcoded roles that are directly referenced throughout the codebase. Because of this, changing an existing role or adding a new one requires changing the code itself rather than updating a flexible permission structure. This makes authorization difficult to adapt and increases maintenance efforts, and the risk of inconsistencies increase on role related changes.

Stakeholder	Interest	Influence	Key Needs
Students	High	Low	Usability, fast feedback
Teachers	High	High	Grading control, insights
Teaching Assistants	Medium	Medium	Efficient workflows
Administrators	High	High	Security, maintainability

Table 1: Overview of stakeholder interest, influence, and key needs

5 System Specification

We have transitioned from a role-based system to a fully permission-based access control model. In this new approach, system behaviour is determined by fine-grained permissions rather than hard-coded roles. Roles such as Admin, Teacher, Teaching Assistant (TA), and Student are now treated as predefined collections of permissions rather than distinct system types.

An Admin represents a user with all available permissions across the system. A Teacher represents a user with permissions related to course management, grading, and student interaction. A TA represents a user with a subset of teacher permissions, limited to actions such as viewing submissions. A Student represents a user with only the permissions necessary to participate in courses, such as submitting assignments, viewing their own submissions, and joining a group.

This permission-based interpretation of these roles is used throughout this document to define and organize the system requirements.

5.1 Stakeholder Analysis

Since CProg-ng is a remake of an already existing system, the primary stakeholders could largely be reused. However, their roles, interests, and influence on the system were analysed to better guide design decisions.

The system involves the following stakeholders:

- **Students**: End-users who submit assignments and receive feedback. Their main interests are ease of use, fast feedback, and clear error reporting. They have high interest but low influence on system design.
- **Teachers**: Responsible for course management, grading, and monitoring progress. They require ease of use and clear insights into student performance. Teachers have both high interest and high influence.
- **Teaching Assistants**: Support teachers in grading and student interaction. Their focus is on access to student submissions. They have moderate influence.
- **Administrators**: Manage the system, users, and configuration. Their primary concerns are security, stability, and maintainability. They have high influence on the system.

To better compare stakeholders, their relative interest and influence are summarised in Table 1.
These stakeholder characteristics directly influenced several design decisions. For example, the need for fine-grained control by teachers and administrators led to the adoption of a permission-based access control system. Similarly, the requirement for fast and consistent feedback for students was taken into account while outlining the checker requirements.

5.2 Requirement Engineering

5.2.1 Requirements Elicitation

Requirements were mostly extracted from the existing system by analysing the core workflows. Core functionalities from the old system were kept the same in order to preserve existing user workflows, while additional requirements were obtained through stakeholder input and iterative refinement. This approach ensured the system would preserve the old functionality and improve it in the places that were requested.

5.2.2 Requirements Prioritization

To structure and prioritize the identified requirements, the MoSCoW method[3] was applied. The MoSCoW method made it possible to clearly distinguish between essential functionality (Must Have), important but non-critical features (Should Have), optional enhancements (Could Have), and explicitly excluded features (Will Not Have). As a result, development tasks could be prioritized accordingly during the implementation of the system.

5.2.3 Functional Requirements

System Management (Must Have)

- The system MUST allow administrators to manage users.
- The system MUST allow administrators to manage roles.
- The system MUST enforce permission-based access control.
- The system MUST support authentication via Single Sign-On (SSO).
- The system MUST allow administrators to disable student login.
- The system MUST allow administrators to disable the submission checker.

Course Management (Must Have)

- The system MUST allow administrators to create and edit courses.
- The system MUST allow administrators to lock and unlock courses.
- The system MUST allow administrators to archive courses.

Assignment and Grading (Must Have)

- Assignments MUST be graded in points.
- Assignments MUST support automated evaluation through unit tests.
- Assignment sets MUST contribute to the final course grade.
- Courses MUST compute a final grade based on assignment sets.
- The system MUST support the disabling point-deductions for specific assignments.
- The system MUST support mandatory assignments and mandatory assignment sets.
- The system MUST allow assignment sets to require a minimum number of completed assignments.

Submission System (Must Have)

- Students MUST be able to submit solutions to assignments as a ZIP file.
- Submissions MUST include source code and related files.
- The system MUST display submission results, including compilation errors, runtime errors, and test results.
- The system MUST display submitted source code with syntax highlighting.
- Submissions MUST include a status indicator.
- Students MUST be able to view their previous submissions.

Feedback and Evaluation (Must Have)

- Teachers and teaching assistants MUST be able to view submissions.
- Teachers and teaching assistants MUST be able to leave comment for students.
- Teachers MUST be able to override submission results (e.g., pass, fail, voided).

Group Management (Must Have)

- The system MUST support group-based assignments.
- Students MUST be able to join a group.
- Teachers MUST be able to manage groups.

Code Checker (Must Have)

- The system MUST automatically execute the code checker upon submission.
- The checker MUST process submitted files and execute defined test steps.
- The checker MUST provide compilation output, runtime output, and test results.

Functional Requirements (Should Have)

- The system SHOULD support assignment templates.
- The system SHOULD allow exporting of grades.
- The system SHOULD support importing of users.
- The system SHOULD provide system activity logs.
- The system SHOULD support modular checker steps.
- The system SHOULD allow administrators to make announcements to all users.
- The system SHOULD have a memory analysis step in the checker.
- The system SHOULD display memory analysis and unused files within a submission.

Functional Requirements (Could Have)

- The system COULD support configurable grading rules.
- The system COULD support overdue submissions.
- The system COULD support assignment categorization.
- The system COULD show logs to teachers.
- The system COULD support unit tests that expect output over stderr.

Functional Requirements (Will Not Have)

- The system WILL NOT support plagiarism detection.
- The system WILL NOT support dependency-based assignment unlocking.
- The system WILL NOT support hidden comment (teacher/ta only).

5.2.4 Non-Functional Requirements

Security (Must Have)

- The system MUST ensure secure authentication and session management.
- The system MUST prevent unauthorized access and permission escalation.
- The system MUST isolate execution environments to prevent sandbox escape.
- The system MUST mitigate denial-of-service attacks caused by malicious submissions.

Performance and Reliability (Must Have)

- The system MUST handle concurrent submissions without data loss or corruption.
- The system MUST enforce configurable resource limits on code execution to prevent system exhaustion.

Maintainability (Must Have)

- The system MUST provide structured logging for critical operations.
- The system MUST support configuration through environment variables.
- The system MUST include automated testing.
- The system MUST provide developer documentation for setup and deployment.
- The system MUST ensure a modular code structure.
- The system MUST enforce consistent code style via linters and formatters.

Non-Functional Requirements (Should Have)

- The system SHOULD provide a user-friendly interface.
- The system SHOULD provide clear and actionable feedback to users.
- The system SHOULD include accessible documentation.
- The system SHOULD provide audit logging of user activity.
- The system SHOULD maintain acceptable response times under normal load.
- The system SHOULD be containerized for portability.

Additional Non-Functional Requirements

- The system COULD support customizable themes.
- The system WILL NOT support non-containerized production deployment.
- The system WILL NOT provide dedicated mobile device support.

6 System Design

This section outlines the design choices for the system, accompanied by both a high-level overview of the system and a more detailed analysis per component.

6.1 Design Choices

This chapter will highlight the design decisions made for the technical aspects of the project accompanied by the reasoning behind them.

6.1.1 Framework Choices

The first major design choice we had to make was regarding which framework to use for the application. The main choices we considered were the following:

- NestJS + NextJS
- Django
- Ruby on Rails
- Laravel

All of these are mainstream, well-tested, and accepted frameworks for web applications. Each has their own pros and cons.

NestJS + NextJS Ultimately, we decided to proceed with the combination of NestJS for the backend and NextJS for the frontend. NestJS is a progressive Node.js framework for building efficient, reliable, and scalable server-side applications. It uses modern JavaScript and is built with and fully supports TypeScript.

The primary motivation for this choice lies in its architecture and handling of I/O operations. Unlike the blocking nature observed in the Django subprocess execution, Node.js operates on a non-blocking, event-driven architecture. This is critical for our use case that involves the 'NsJail' sandbox. When a student submits code, the server must spawn a child process to run the checker. NestJS allows us to spawn these processes asynchronously without halting the main thread. This ensures that the application remains responsive to other users even while multiple heavy-duty checking processes are running in the background.

Furthermore, NestJS provides a modular architecture that enforces a strict structure, making the codebase highly extensible and maintainable.

Django Django is a framework known for its quick turnaround time, easy prototyping, and fast deployment. It is used by large websites such as Instagram, Mozilla, Pinterest, and more. Django uses the Python programming language and has a simple framework that at its core includes models and views. A model is a database object, and a view is a webpage. The reason we did not choose Django is due to one major setback when using Python. In Python, when a subprocess is spawned, it blocks the thread when executing [9]. If many students submit their solutions at the same time, the server could get overwhelmed, and as a result users would experience slow responses and long checker times.

Ruby on Rails Ruby on Rails is a mainstream web application framework used by many big websites such as GitHub, GitLab, Canvas, and Airbnb. It is famous for its "Convention over Configuration" philosophy, which allows for rapid development by assuming what the developer needs to do rather than requiring endless configuration files.

However, we decided against using Ruby on Rails primarily due to the team's technical expertise. Although the framework is robust, our team lacks familiarity with the Ruby ecosystem. Adopting it would have introduced a significant learning curve without offering a decisive advantage over the Node.js ecosystem that would justify the initial loss in velocity.

Laravel Laravel is a web application framework with expressive, elegant syntax, built on PHP. It follows the Model-View-Controller (MVC) architectural pattern similar to Django and Ruby on Rails. It is currently one of the most popular frameworks for web development, boasting a vast ecosystem and powerful features like the Eloquent ORM.

Similarly to our decision regarding Ruby on Rails, we chose not to use Laravel due to the team's lack of familiarity with the PHP tech stack. While Laravel is a capable and mature framework, it did not present any unique architectural advantages for our specific requirement of non-blocking process spawning that would outweigh the benefits of using a JavaScript or TypeScript stack, which the team is already proficient in.

6.1.2 Code Checker Choices

The code checker is one, if not the most important, aspect of the Cprog-ng project. We need to be able to automatically check the students submissions against a preconfigured set of unit tests. Since we allow the student to execute arbitrary code on the server, we put an emphasis on security and sandboxing. Due to this requirement, we were limited to only a few options, of which we considered the following three:

- Docker
- NsJail
- WebASM

Docker Docker is a mainstream containerization application used by developers for local development and for deployments all around the world. A Docker container allows for sandboxing, process management, resource limitation, and much more. However, a Docker container is a heavy and expensive process. Spinning up a container for every unit-test or student submission would introduce significant latency. This startup overhead would create a bottleneck during high-load periods, making it an inefficient choice for a real-time checking system where speed is essential.

NsJail NsJail is a lightweight Linux isolation and sandboxing tool originally developed by Google. We chose this tool as our code checker solution because it strikes the perfect balance between security, performance, and resource limitation [10].

Unlike Docker, NsJail leverages Linux namespaces and cgroups directly to isolate processes without the overhead of a container daemon. It is not heavy or expensive to run; it allows us to spawn a securely isolated process in milliseconds. This efficiency enables the system to handle concurrent submissions and run multiple test cases per submission without exhausting server resources or blocking the thread, perfectly aligning with our choice of a non-blocking NestJS backend.

WebASM WebAssembly (WebASM or Wasm) is a binary instruction format that allows code to run in a memory-safe, sandboxed execution environment. It is often used to run high-performance applications in web browsers.

We considered compiling student code to WebASM to execute it safely. However, we decided against this option due to the complexity it adds to the compilation pipeline. To run C code in WebASM, it must be cross-compiled using tools like Emscripten, which

behaves differently than a standard GCC or Clang compilation on Linux. This could lead to discrepancies where a student's code works in the checker but fails locally (or vice versa), specifically regarding system calls and file I/O. Furthermore, the overhead of this cross-compilation step provided no clear benefit over the native speed and simplicity of NsJail.

6.1.3 Course System Choices

The course system is another important part of our new CProg-ng system. The course system is used to separate assignments and submissions from different years, inspired by Canvas. In our course system, a single course has many assignment sets, where a single assignment set has many assignments, and a single assignment has many submissions. This course system has several benefits compared to other systems, which we can best explain by covering the flaws of the other options we considered.

Global Assignments One option we considered was having global assignments like the old CProg system. In this case, there would be global assignments that everyone has access to, and all would be used for grading students. However, there is a problem with this system: there is no separation between years. As a result someone who is redoing the course will see their submissions from the previous times they did this course. Furthermore, any assignment changes will also break previous submissions. In the old CProg system, this problem was mitigated by emptying the system after moving all data to a backup server. However, we wanted to solve this problem, so we disregarded this as a viable option.

Global Assignments with Revisions Another option we considered was global assignments like the old CProg system, but with revisions for each year. This improves on the Global Assignments by defining revisions to which the assignments belong, allowing changes to newer iterations of the assignments and separating the submissions of students by each revision. Although this option does solve our problem, it is too restricted for our liking, and we would prefer a more flexible system.

Courses, AssignmentSets, and Assignments To solve the problem of annual revisions and to make our system more flexible than a simple annual revision approach, we had another option with courses, inspired by Canvas. The system would have multiple courses, each of which has assignment sets, containing assignments. This solves the annual revision problem by allowing multiple versions of a course to exist: Programming in C 2025, Programming in C 2026, and so on. Creating a new course for a new revision of a course is also done on Canvas, so teachers will be familiar with this system. This system also allows for more flexibility as this also would allow for more courses than just the Programming in C course, whilst also allowing courses to group their assignments in assignment sets, so that they can customize their course to their heart's content.

6.1.4 Role and Permission Choices

The role and permission system is an important part of our new CProg-ng system. The role and permission system is used to determine what each user is allowed to view and do inside our system. We had several discussions about how to structure our role and permission system, as we disagreed about how it could best be achieved.

Fixed Roles or Configurable Roles One major discussion point in our group was whether we should have a fixed set of roles: student, ta, teacher, and admin, or whether we should leave the roles more configurable, providing the same set of roles as a basis

that teacher/admins can configure how they see fit. This discussion mainly revolved around whether the added functionality of configurable roles outweighed the difficulty of implementing the roles.

Fixed roles would mean that we could simply define a simple enum type for the roles, which could then be assigned to users and which can easily be checked. Configurable roles would mean that we would have to make separate objects for each of the roles, where each has its own list of permissions. Checking whether a user would then be allowed to do an action would be more complex, as we would need to check whether the user has a role that has a given permission. Furthermore, we would also need to implement an API and add an interface to the frontend for creating, editing, managing the permissions of, and deleting roles.

This discussion was ultimately resolved by our client, as he expressed interest in being able to configure the roles within the system (section 11.1.2).

Global Roles or Course Specific Roles Another major discussion point in our group was whether the roles should be global or whether they should be course specific. Again the discussion revolved around whether the functionality outweighed the complexity.

Global roles would mean that we would only have to check whether a user has a given role or permission, whilst for course specific roles we would also need to add a way of discovering which course each request belongs to. However, course specific roles would give better control over what each user can do, because a user can be a student in one course, while at the same time being a ta in another course. Such functionality cannot be achieved with global roles.

Similarly to our discussion about fixed or configurable roles, we asked our client (section 11.1.4) what they wanted, to which they responded that they would like to be able to assign users different roles per courses.

Role Templates The Role Templates are used when creating a course, being simple templates that can be added to a course when it is created and which can then be edited by the admins/teachers to configure/customize the roles to fit their needs.

Role Templates were suggested by our supervisor (section 11.1.4) after we explained our new user-role-permission system based on our decision regarding Global Roles or Course Specific Roles.

Default Roles One of the last features requested by our supervisor were default roles. This would allow you to mark a role as default, which means that any person joining the course will be able to access it with this role. This would reduce the workload of the maintainer as they would no longer need to manually assign users to a course. Both the role system and templates were updated to support this feature. Each course can only have one default role at a time.

6.2 High-Level Design

This chapter will focus on a high-level overview of our system.

6.2.1 Architecture

The system follows a containerised architecture which consists of a client, server, and a database, all deployed and orchestrated using `Docker Compose`. In production requests are routed to `/api` and `/admin` to the backend, while all other requests are routed to the frontend.

The frontend communicates with the backend through a `RESTful API` over `HTTP`. For real time-updates such as submission checker progress, the backend pushes events to the frontend using `Server-Sent Events`.

The backend is the only component that communicates directly with the database. It does so using `TypeORM`. Authentication is handled through an external identity provider, as such the frontend does not interact directly with the identity provider.

6.2.2 Components

Frontend The frontend is a `Next.js` application written in `TypeScript` using the `React-19` library. It is responsible for rendering the user interface, handling client side navigation, and communicating with the back end over `HTTP`. Internally it is organised as a monorepo with three main components: the application routes and pages, a stand-alone reusable component library, and a shared theme library.

Backend The backend is a `NestJS` application written in `TypeScript`. It serves as the central `API` server and it is responsible for logic, data access, and coordination between the other components. It exposes a `RESTful API` that is called by the frontend.

The code checker is a subsystem within the backend that automatically evaluates student submissions against preconfigured `unit tests`. When a submission is received, the checker service executes a set of discrete steps, where the critical steps are sandboxed for added security.

Database The system uses `PostgreSQL-15` as data storage. The schema for the database is managed through `TypeORM` and evolved using migrations. This ensures that schema changes are versioned, repeatable, and applied automatically on deployment. Additionally, custom seeder module is in place for database population, both for the production and development environment.

6.2.3 Data Flow

The most important data flow in the system is the submission flow, which arises when a student uploads a zip file for a given assignment. This data flow involves all the major components of the system. The sequence diagram for the submission flow can be seen in figure 13 in appendix C. It illustrates the flow from the moment a student uploads a zip file until the final grade updates. When a submission is received, the backend stores the file, creates a database record, and starts the checker pipeline. As the checker pipeline progresses, the backend pushes status events to the frontend via `Server-Sent Events`. Once the checking process has completed, the grading service recalculates the students grade and a notification is issued. The checker pipeline is further outlined in Section 7.3.

6.3 Detailed Design

This chapter will show a more detailed analysis of the design per component.

6.3.1 Back-end

The Cprog-ng backend is a modern and modular application built with Nest.js, AdminJS, and Typescript. Its primary responsibilities are communication between the frontend and the database, and the code submission checker. The new backend should improve on the old CProg system by being easier to maintain and expand. Additionally, CProg-ng should support and improve upon the features deemed must haves by the requirements.

Constraints During the development of the backend, a few key constraints were taken into account:

- **Secure handling of information.** The backend processes sensitive data, such as student information, submission grades, and other authentication related data. Therefore, it was important to ensure that the information could not be accessed, leaked, or modified by unauthorized users, or parties.
- **Secure code execution.** Given that the system executes code submitted by students, the backend has to ensure that untrusted programs could not compromise the host system. Uploaded code should never be able to escape the execution environment, access server resources outside of its scope, or interfere with other system processes.
- **Fairness.** Because the backend is responsible for automatically grading student submissions and computing the final course grade, it must be consistent, reliable, and fair. Submissions should all be processed to the same set of rules and conditions to ensure fair assessment.
- **Maintainability.** The backend had to be designed in a way that makes it maintainable and expandable, especially given that the system will be handed over to future maintainers who were not involved in developing CProg-ng. The future maintainers should therefore be able to understand, modify, and expand the system without excessive difficulty.

Application Overview The backend is a single application organised in the following directories:

- **admin.** Integrations and components related to AdminJS, including custom modules, UI components, and configuration for the admin interface.
- **assets.** Non-code resources and static files used by the backend, such as templates or documentation assets.
- **auth.** All authentication logic, including controllers, services, strategies, and related exceptions and guards.
- **checker.** The core logic for the automated checker system, including its processing steps and types.
- **config.** Configuration modules and files used by the system.
- **database.** Database connection setup and TypeORM configuration, including data sources and migration logic.
- **decorators.** Custom TypeScript decorators used throughout the application.
- **grading.** Logic and utilities for grade calculation, including controllers, services, and grading algorithms.
- **guards.** Custom authorization and validation guards.
- **interfaces.** TypeScript interfaces that define data structures used across the backend.
- **logger.** Logging modules and configuration, providing structured and levelled logging throughout the application.

- **models.** TypeORM entity models, along with their related services and controllers, more detail in section 6.3.3.
- **permissions.** Enum definitions and logic for managing application permissions.
- **seeder.** Database seeders for populating the database with default or legacy data, more detail in section 6.3.3.
- **sse.** Implementation of Server-Sent Events (SSE) for real-time communication with clients.
- **storage.** Logic for handling the storage and retrieval of user submissions and related files.
- **system_setting.** Enum definitions and logic for managing system-wide settings.

AdminJS AdminJS is used in our project to give an admin the ability to view and change the system data. The AdminJS system provides default pages for most entities/models in the TypeORM system, allowing the admin to view and modify the data in the system directly. Alongside the default pages, we have also created custom pages which make certain tasks easier for the admin:

- **System Settings.** The System Settings page gives the admin control maintenance mode and checker availability for the entire platform.
- **Creating Courses.** The Course Builder page is a guided workflow to create or update a course, including assignment sets, assignments, unit tests, and role templates.
- **Archiving Courses.** The Course Archive page allows the admin to archive, restore, and delete archived data.
- **Managing Archives.** The Archive Manager page is for managing archive bundle files on the disk of the server. uploading files, listing files, exporting files, and deleting files.
- **Importing Users.** The User Import page is for importing users into the system, either individually or in bulk, with CSV previews, CSV mapping, and optionally assigning roles to users.
- **Editing Role Templates.** The Role Template Editor page is for creating and editing role templates. The page also allows role templates to be applied to courses.

Authentication In the backend, there are two ways to authenticate yourself:

- **Local Accounts.** Local accounts are authenticated directly by the backend. Authentication is done using an email and a password. Initially, passwords are hashed using Bcrypt[11], before they are stored in the database. Upon logging in, Bcrypt is used to compare the given password to the stored hash in the database.
- **SSO.** for Single-Sign On we use OpenID Connect (OIDC)[12], which is a standard way of doing authentication built upon OAuth. Single Sign-On is used to allow students to log in to the system using their student account.

Checker In the backend, the checker is implemented using the Chain of Responsibility[13] design pattern, to make a modular and extendable checker system. Currently, the checker has the following steps in order:

1. **Setup steps.**
   (a) **TmpDirStep.** Creates a temporary directory for the submission to live in whilst the checker is running.
   (b) **DownloadStep.** Downloads the submission zip file into the temporary directory.
   (c) **UnzipStep.** Unzips the zip file from the submission into the temporary directory.
   (d) **FileUsageCheckStep.** Scans the unzipped files of the submission and identifies used/unused files.
   (e) **AssetIntegrityStep.** Verifies that all required assets are present and have not been modified by the student.
   (f) **CompileStep.** Compiles the C source and header files into an executable binary.
   (g) **AssetInjectionStep.** Injects the selected runtime assets into the sandbox environment.
2. **Unit test steps.**
   (a) **UnitTestStep.** This step runs once for every unit test of the assignment. This step runs the compiled binary, writes the unit test input to the `stdin`, and then collects the `stdout` and `stderr` to compare. If the `stdout` and `stderr` match the expected output of the unit test, then the unit test passes, otherwise it fails.
   (b) **ValgrindStep.** This step runs only if all the unit tests have passed and will run once for every unit test. This step will run the compiled binary, writes the unit test input to the `stdin`, and then analyse the program using Valgrind to detect mistakes such as memory leaks. If there are any memory leaks, then the Valgrind test fails, otherwise it passes.
3. **Cleanup steps.**
   (a) **CleanupStep.** Cleans up the project after running, deleting the temporary directory and the files within.

A more detailed overview can be found in the checker sequence diagram 14

Decorators and Guards In the backend, we use decorators and guards to protect the endpoints from being accessed by unauthorized users. We have defined four decorators for this task:

- **RequireAdmin.** Requires that the user is an admin, meaning that the user has the `isAdmin` flag set to `true`. This decorator uses the `AdminGuard`.
- **RequireCoursePermission.** Requires that a user is authenticated and has a specified permission in the course to which the request pertains. If the user is not authenticated, then `401 UNAUTHORIZED` will be returned. If the user does not have the necessary permission, then `403 FORBIDDEN` will be returned. This decorator uses the `PermissionGuard`.

- **RequireCourseUnlocked.** Requires that the course to which the request pertains is unlocked. If the course is not unlocked, then `403 FORBIDDEN` will be returned.
- **RequireSystemSetting.** Requires that a system setting has a given value, currently only supports the `frontend-login-available` flag. This decorator uses the `SystemSettingGuard`.
- **RequireFrontendLoginAvailable.** Requires that the frontend login is available, meaning the `frontend-login-available` flag is set to `true`. This decorator is an alias for the `RequireSystemSetting` decorator.

Pagination In the backend, some endpoints will return a list of data. However, these lists may become large, so the returned lists are paginated to prevent a large amount of data from flooding the network at once.

Server-Sent Events (SSE) In the backend, Server-Sent Events are used to give the students live updates about their submissions whilst they are going through the checker pipeline. Every time a submission advances through the pipeline, an event is sent out to all listeners which can then update their state with the new information.

6.3.2 Front-end

The Cprog-ng frontend is a modern, modular web application built with Next.js (App Router), React 19, and TypeScript. Its primary responsibility is to present backend data in a user-friendly, efficient, and extensible interface, replacing the frontend part of the legacy PHP system. The primary objective of the new frontend is to deliver a polished, maintainable interface that supports all required features, improves usability, and is easy to extend for future needs.

Constraints Several constraints were key in shaping the new design:

- **Server-agnostic deployment.** The backend URL had to be configurable at build or runtime because the frontend is shipped as its own container which can be pointed to different environments.
- **Browser compatibility.** Since the system will be used on a variety of machines, we could not assume any particular OS or browser. Therefore, we rely on standard HTTP protocols and widely supported web technologies to ensure compatibility across all major browsers.
- **Feature parity and extensibility.** The new system had to include at least the features of the legacy system in addition to new requirements, which ruled out a rudimentary MVP approach and requiring a modular architecture from the start.
- **Maintainability.** The frontend had to be maintainable and easily extensible, given that the project will be handed over to new maintainers after development.

Architecture Overview The frontend is organised as a monorepo with three main packages:

1. **Application (`frontend/src`).** Contains the Next.js app, route groups, API clients, hooks, providers, types, and utilities. The app uses permission-based rendering for fine-grained access control.

2. **Component Library (`frontend/packages/components`).** This is the path to the stand-alone UI library being `@cprog-ng/components`. It contains the reusable visual building blocks that make up the pages; examples include Button, Profile and NavBar. Components are styled using the theme package and documented in Storybook. Automated tests are written with Vitest and Playwright verifying the expected behaviour.

3. **Theme Library (`frontend/packages/themes`).** Provides design tokens (colors, spacing, typography, etc.) and multiple themes (Light, Dark, and Windows 95). The package exports a `ThemeProvider` and utilities for theme management, ensuring consistent styling across the app and component library.

Component Library Details The component library is the main feature to make the frontend both modular and reusable. It is located in `frontend/packages/components/` and exports all components from `src/index.ts`. The library includes:

- **UI Primitives.** `Button`, `Box`, `Label`, `Avatar`, etc.
- **Navigation.** `Navbar`, `NavButton`, `Tabs`, `Pagination`, etc.
- **Data Display.** `Table`, `FileList`, `Profile`, `Notification`, etc.
- **Interactive Widgets.** `Popup`, `GradingLadder`, `Searchbar, etc`.
- **Feedback.** `LoadingSpinner`, `CircleProgressBar`, `LineProgressBar`, etc.

Components are designed to be focused and modular, with styling driven by the theme package. Storybook is used for isolated development and documentation, and all behaviour-heavy components are accompanied by automated tests. In this way, the expected behaviour and styling can be confirmed before using the components in the pages.

Theme Library Details The theme package (`@cprog-ng/themes`) defines all design tokens and supports multiple themes. It is located in `frontend/packages/themes/` and exports:

- **Themes.** `CprogTheme` (Material-inspired), `CprogDarkTheme`, and `Windows95Theme`.
- **Tokens.** Colours (with semantic and palette variants), spacing, font families and sizes, font weights, line heights, border radii and widths, box shadows, transitions, z-index, opacity, and breakpoints.
- **Theme Management.** `ThemeProvider`, `useTheme`, and utilities for theme selection and registration.

All components and pages consume these tokens via styled-components, ensuring visual consistency and easy theming. The user theme preference is read from a cookie and applied server-side to prevent flashes of incorrect themes.

Navigation and User Flows Navigation follows the routing structure of the application. The entry point is the root path `/`, where the login page is hosted and validation of existing sessions occurs before redirecting further. From there, the user is prompted to select a course, after which they are redirected to the home page `/home`. The homepage is adapted to the user. It currently supports two main modes, teachers or teaching-assistants and students. On the home page, the teacher or teaching assistant will see a course-wide overview, while students will see their own progress for a given course.

From the homepage, the user can navigate to a specific `/assignment` view. As each assignment allows for multiple attempts, you can navigate from this page to a specific `/submission`. This is the most information dense page in the system, it combines file listing, code viewing, comment threads. The teacher can also override the checker results here.

A persistent navigation bar at the top provides shortcuts to the other functional areas Groups, Users, Help, and Notifications for students, with an additional Settings and Announcement page for teachers.

Keep in mind that what the user sees and is able to interact with for each route is not decided by role names, but by permissions. The client receives a permission set which is used for fine grained conditional rendering.

State Management State is managed using a combination of React context, custom hooks, and local state:

- **App-wide state.** Managed by context providers such as `UserSessionProvider` (session, course, notifications) and `ToastProvider` (UI feedback).
- **Reusable logic.** Encapsulated in custom hooks (e.g., `useAutoRefresh`, `useLocalStorageState`, `useToast`) located in `src/hooks/`.
- **Component-local state.** Managed with `useState` and `useEffect` for isolated, screen-specific data.

State is purposefully kept close to where it is used, minimizing unnecessary re-renders and improving maintainability.

Responsiveness and Accessibility The theme package defines a standard for scales from `xs` up to `2xl`, which the components consume through styled-components rather than hardcoded pixel values. Layout sensitive components have been designed to adapt to different view port sizes, and the current implementation targets standard laptop and desktop screens. Visual components were developed and reviewed inside Storybook. This development approach helped surface colour, contrast, and styling issues early before components were integrated into the application pages. All components reference theme tokens for colours, borders, and spacing, which made it possible to adjust themes centrally without reworking individual components.

Performance Several decisions contribute to the performance of the system:

- **Server-render first paint.** Next.js renders the initial HTML on the server with both the correct theme and the appropriate layout already applied. Because layout selection happens server-side, the application can enforce permission-based layouts before anything reaches the client. This prevents flashes of incorrect themes, and exposing UI structures that the user is not authorized to see.
- **Scoped re-renders.** Because state is split across small, focused contexts and local hooks, most updates only re-render the subtree that actually depends on that state. This includes, but is not limited to, a notification arriving, a course being selected, and showing a toast.
- **API request de-duplication.** The API layer tracks in-flight requests to avoid redundant network calls.

Data Fetching and API Abstraction All data fetching is handled via API clients in `frontend/src/api/`. Components never call `fetch` directly. Instead:

- **API abstraction.** Each resource (user, course, submission, etc.) has a dedicated module exporting asynchronous functions for CRUD operations. These functions handle URL construction, credentials, and error handling.
- **Request de-duplication.** In-flight requests are tracked using maps and variables to avoid duplicate HTTP calls for the same resource. Once a request resolves, the corresponding entry is cleared.
- **Caching and revalidation.** There is no persistent cache or automatic background revalidation (as provided by libraries such as React Query). Caching is handled manually and scoped per request. Frequently used data (like user information and courses) is re-fetched on navigation or explicit refresh.
- **Error handling.** Errors are propagated via exceptions or structured error objects. Global handling is applied for common cases (e.g., redirecting to the login page on HTTP 401 responses), while user-facing feedback is provided through toasts and notifications.
- **Server vs. client.** API modules expose functions usable in both server-side rendering and client-side contexts, with explicit handling of cookies and authentication credentials.

This approach provides complete control over the data flow, error handling, and request lifecycle, at the cost of increased manual management compared to higher-level data fetching libraries.

Authentication and Permissions Authentication is managed through NestJS sessions, using secure cookies and session management in the `UserSessionProvider`. All UI rendering is permission-based, with permissions fetched on login and checked throughout the app. This enables fine-grained access control and flexible user flows.

Components corresponding to restricted actions are not rendered when required permissions are absent, which means that buttons, views, and interactions are not exposed in the user interface. This reduces the likelihood of users discovering or attempting unauthorized actions and keeps the interface focused on relevant functionality.

However, this mechanism is not trusted upon for security. All permission checks are enforced server-side by the backend API. Even if a user bypasses the frontend and issues direct requests, unauthorized actions are rejected by the backend.

6.3.3 Database

Overview The system uses `PostgreSQL-15` as its relational data storage, which is accessed through `TypeORM`. For the production environment, schema changes are applied through versioned migrations that run automatically when deployed. In the development environment, `TypeORM`'s synchronize mode is used instead, which updates the schema to line up with the entity definitions for each restart.

Entity Model The database schema can be grouped into six major areas:

1. A **Course** is the top level unit. Each course contains one or more `AssignmentSets`, where each such set contains one or more `Assignments`. Each assignment defines its own set of `UnitTests`, which specify the what the input and output should look like, and specifies the ordering which the code checker will use. Courses also store configuration information such as group sizes, publication, and lock status.

Optionally, a set of grading brackets can be set in a field of type `JSONb` (which is JSON, stored in a binary representation for fast querying).

2. **Submission** is an abstract base entity that uses single table inheritance. It has two concrete children: `IndividualSubmission`, which references a user, and `GroupSubmission`, which references a group. The base entity stores all shared fields including the uploaded files, compilation and Valgrind output, checker status, submission status, and any potential override to the grade. Each submission links to its `UnitTestResults`, which store the actual output and correctness for each test. It also links to its `SubmissionComments`, which support feedback between students and teachers/teaching assistants.

3. **Users and Access Control** A `User` is identified by a UUID and can be authenticated locally with a password or through an external provider using SSO. Users are associated with courses and groups using many-to-many join tables. Access control is handled using `Roles`. Each role belongs to a specific course and stores a list of explicit permissions. `RoleTemplates` provide reusable presets that can be applied when creating new roles for a course.

4. **Grading** Grades are computed and cached at three stages. `AssignmentPoints` records the number of points earned for a specific assignment. `AssignmentSetGrade` records the compound grade across an entire assignment set. `CourseGrade` records the overall course grade. Additionally, each grade record tracks timestamps of the latest update to avoid unnecessary recalculations.

5. **Notifications** A `Notification` is a second abstract base entity that uses single table inheritance. Its three children are `Announcement` which is course wide, `IndividualNotification` which is targeted at a single user, and `GroupNotification`, which is targeted at a group. The read statuses of notifications are tracked through many-to-many join tables between the notifications and the users. All notifications support soft deletes.

6. **System** `SystemSetting` is a simple key value store used for runtime configurations, such as if the checker is enabled. `Log` records entries for administrative actions. It stores the action, resource, and responsible user.

The relationships between these entities are illustrated in the class diagram in Figure 15 in Appendix C. This diagram provides an overview of how courses, assignments, submissions, and users are related within the system.

Key Design Decisions

- **Single Table Inheritance.** Both the submission and notification hierarchies use single table inheritance rather than separate tables per type. This was chosen to keep queries simple, as all submissions and notifications each can be fetched from a single table. This allows us to avoid the join overhead of multi table inheritance. The trade off here is that child specific foreign key constraints cannot be enforced at the database level. Instead, `BeforeInsert` and `BeforeUpdate` validation hooks are used to enforce them at the application level.

- **Composite primary keys for grades.** The three gradables all use composite keys of a user and entity rather than a surrogate auto-increment key. This ensures uniqueness at the database level and makes `upserts` straightforward.

- **Permissions as simple array.** Role permissions are stored as a simple array in a single column rather than in a dedicated join table. We chose this to keep the role

model simple to read and compare permission sets while at the cost of not being able to query individual permissions with standard SQL.

- **Grading brackets as JSONB.** Courses can override the system default grading brackets by storing them as a custom bracket configuration as a `JSONB`. If no override is set, it defaults to the system-default. This allows for per course grading rules without any additional tables.

Seeding and Migrations In production, the schema evolves through `TypeORM` migrations, which are run automatically before the application starts. This ensures that the schema changes are versioned, repeatable, and consistent across environments.

A custom seeder module is in place to handle the initial data population. In all environments it creates the default role templates and system settings. For the development mode it additionally creates an admin user and a grading test fixture. Finally a separate seed mode populates the database with assignments and unit tests.

7 Implementation

This section outlines the implementation of the CProg-ng system, giving more details about the setup, key features, code execution flow, and the challenges encountered during development.

7.1 System Setup

The CProg-ng codebase is organized as a monorepo, containing the codebase for both the frontend and backend, as well as configuration and deployment scripts. The repository is structured to promote modularity and maintainability:

- **Frontend.** Located in `frontend/`, implemented with Next.js (React 19, TypeScript), and organized into packages for the main app, a reusable component library, and a theme provider.
- **Backend.** Located in `backend/`, implemented with NestJS (TypeScript), responsible for business logic, API endpoints, and integration with the database and code checker.
- **Shared configuration.** Dockerfiles, Compose files, and environment variable templates are provided at the root and within each service directory to support local development and production deployment.

Environment setup is streamlined using Docker Compose, which encapsulates all dependencies and runtime requirements. Developers can spin up the entire stack locally with a single command, ensuring consistency across different environments.

7.2 Key Features

The implementation of CProg-ng prioritizes a modular, type-safe architecture to support adaptability and extendability. The core features of our system are as follows:

Responsive Component-Driven UI The frontend is built on a library of reusable UI components, developed in isolation using Storybook and verified with Playwright. This ensures that the interface remains accessible and responsive while maintaining visual consistency throughout the application.

Sandboxed Execution Pipeline A secure, containerized environment forms the basis of the automated checking system. By utilizing a pipeline, the system safely compiles and analyses student submissions for both logical correctness and memory safety without compromising the host infrastructure.

Granular Access Control To support different access levels, the system implements a permission-based access model. Unlike a rigid role-based systems, fine-grained approach allows for the precise assigning of access to features and data based on specific user requirements.

Standardized Data Orchestration Communication between the frontend and backend is managed through a centralized API abstraction layer. This client-side architecture facilitates efficient data fetching with request deduplication, error propagation, and optional caching strategies.

Type-Safe Modular Architecture Both the frontend and backend are designed with a strict separation of concerns. By enforcing strong typing and modular design patterns, the system remains highly extensible, allowing future maintainers to introduce new features with minimal changes to the existing functionality.

7.3 Code Execution Flow

This section outlines the code execution flow and provides a detailed explanation about the steps mentioned in the diagram below.

7.3.1 Part 1: Checker Setup

The first part of the responsibility chain is the checker setup. This phase ensures a clean, isolated environment and includes the following steps:

- **tempdirstep.** Creates the directory where the checker operations will take place. This directory is created at `/tmp/cprog-ng/{submission_id}` and is later mounted by the checker. We check each submission in its own directory to ensure no side effects occur from the presence of other files.
- **downloadstep.** Copies the student's submission from the submission storage into the newly created temporary directory.
- **unzipstep.** Extracts the student's submission, provided it meets strict security constraints: a size no larger than 20MiB, a directory depth no higher than 5, and fewer than 20 total files. These security measures were put in place to prevent potential DoS attacks using zip-bombs.
- **fileusagestep.** Check which files within the submission will be used and which are unused. This was a requirement from our supervisor.
- **assetintegritystep.** Check whether the provided files to the student are present and unaltered.
- **compilestep.** Compiles the student's submission into a single executable using `make` and `gcc`. We use `make` to target all the `.c` and `.h` files in the student's submission, and then compile it using `gcc` to a `c` target. This allows the student to name their files however they like without the checker missing them. Security measures were put in place in the `makefile` to prevent flag-injection attacks by naming files as `make`, `gcc`, or `rm` flags. The `compilestep` is also the first step that is executed within `nsjail` to prevent possible attacks using compile hooks and targets.
- **assetinjectstep.** Inject the required assets into the runtime directory of the checker. Exposing assets at `/asset/<asset>`

7.3.2 Part 2: Executing Steps

The second part of the checker chain involves executing and testing the compiled code:

- **unitteststep.** Uses the created temporary directory and executable to apply the unit tests. Each unit test features four independently configurable options: executable flags, `stdin` blocks (string inputs), expected output, and the target output stream (`stdout` or `stderr`). For each unit test of an assignment, a separate `unitteststep` is created and executed in parallel. To ensure safety, the unit test is executed within `nsjail` under very strict file access and resource limits. The executable only has write access to the temporary directory, can only read select libraries and system files, and is capped at 64MiB of RAM, 5MiB of IO activity,

10 processes, 5 seconds of clock time, and 2 seconds of CPU time. This is put in place to prevent possible DoS attacks using fork-bombs, busy-wait programs, and starvation.

- **valgrindstep.** If all the unit tests pass, this step runs `valgrind`—a memory analysis tool—against all of the unit tests. This step provides further feedback to the students on whether their program is free of memory leaks, which is an important aspect of programs written in C. The `valgrindstep` has similar read and write restrictions as the `unitteststep`, however, it is permitted more RAM, IO activity, processes, clock time, and CPU time. This is due to the overhead that `valgrind` requires in order to analyse the programs.

7.3.3 Part 3: Cleanup

The final part of the checker chain prepares the system for the next cycle:

- **cleanupstep.** Removes the temporary directory in order to free up space for another submission.

7.4 Challenges and Solutions

Several challenges arose while implementing the CProg-ng system.

Security Ensuring that untrusted code could not escape the checker environment required careful use of `nsjail`, strict resource limits, and thorough validation of all user inputs. These measures were needed in order to maintain system integrity and prevent potential abuse.

Performance Improving system responsiveness involved running tests in parallel and deduplicating API requests. While these optimizations reduced latency, they also introduced complexity in state management and error handling, which had to be addressed to ensure reliability.

Extensibility Designing the system with future maintainers in mind required significant effort in documentation, strong typing, and modular code. Although developing the API and frontend components initially slowed progress, it ultimately resulted in a more scalable and maintainable system.

Integration Integrating the different modular components (frontend, backend, authentication, etc.) required the design of clear interfaces and consistent error propagation mechanisms. Establishing well-defined communication between components was crucial to ensure system stability and ease of debugging.

These challenges were mostly introduced by the requirements of building a maintainable and scalable system. Although they required additional effort during development, they ultimately provided a solid foundation for future extensions and improvements.

8 Testing and Validation

Since our system is responsible for automated grades, thorough testing is required to ensure fair checking of student submissions. There are several layers of testing which where applied to Cprog-ng.

8.1 Test Plan

This section shows the test plan. Our test plan consists of two key features. Firstly, we want to automate as much testing as possible, this includes unit-tests and end to end (e2e) tests. This allows for a streamlined and automated testing process, which tightly integrates with the CI/CD pipeline that was setup. Secondly, we have manual test for things that cannot be automatically verified, either due to limitations of the system, host, or another reason. Manual tests include the uploading and automated checking of submissions, but also security and penetration testing.

8.2 Unit Testing

The first line of testing we have are the unit tests. For each service in the backend, which includes grading, api endpoints, submission checking, checker steps, etc. A unit test is written that ensures correct execution and handling of edge cases. There are also unit tests in the frontend for our components. These unit tests ensure correct rendering of the foundational building blocks for our frontend. Besides this, the logic in more complex frontend components is checked.

8.3 End-to-End Testing

This section details our approach to End-to-End (E2E) testing, which currently consists of two primary test suites.

The first suite focuses on the database. It verifies that TypeORM correctly interacts with the database to handle migrations, upsert models, and manage data integrity. This suite includes test cases for each model, specifically validating auto-incrementing columns, creation and update timestamps, and key constraints.

The second suite is our system test. Its scope is inherently limited due to our architectural reliance on NsJail for sandbox execution. NsJail requires elevated Linux privileges to operate; permissions that standard users and default testing environments typically lack. Because we cannot guarantee these privileges in a standard test execution environment, this E2E test is restricted to verifying the initial uploading and processing of student submissions. The actual checking and grading phases are skipped in this specific test to avoid permission errors on the host machine.

8.4 Security Testing

This section outlines our approach to security testing, which was primarily conducted manually and on a conceptual basis. The system incorporates foundational web security measures to prevent Cross-Site Request Forgery (CSRF), Remote Code Execution (RCE), and HTML/SQL injections. These baseline protections are largely provided by our underlying framework and its bundled packages.

However, our primary security focus was on the checker pipeline. The previous system contained several vulnerabilities that could lead to sandbox escapes, privilege escalation, and Denial of Service (DoS) attacks. To prevent these vectors, the new checker pipeline is rigorously locked down through a multi-layered defence strategy.

The first layer of defence relies on `NsJail`, a Linux-namespace sandboxing tool that inherently mitigates most sandbox escape and privilege escalation attempts. The second

Module	Stmts	Branch	Funcs	Lines
checker/steps	92%	82%	85%	92%
guards/admin	100%	100%	100%	100%
guards/permission	93%	82%	100%	93%
auth/strategies	100%	75%	100%	100%
auth/exceptions	100%	100%	100%	100%
grading	53%	46%	73%	52%

Table 2: Backend code coverage by module

layer operates during the download and unzip phases. Here, we enforce strict, configurable maximums on file size, directory depth, and total file count to prevent DoS attacks, such as zip bombs.

The third layer secures the compilation step. Without proper precautions, a simple Makefile can be exploited for sandbox escapes or privilege escalation via flag-injection attacks. For example, an attacker could maliciously name a C file to mimic command-line flags, such as `-rf --no-preserve-root / .c`. When the Makefile compiles this, it creates an object file named `-rf --no-preserve-root / .o`. If this filename is passed unescaped to the native Linux `rm` command during cleanup, it is treated as a destructive flag rather than a file, executing with the permissions of the `make` process. We implemented safeguards to prevent this and similar CC flag injections.

The final layer of defence involves strict resource limits enforced by NsJail. The sandbox is configured to allocate only the absolute minimum system resources required for program execution. By capping memory usage, I/O activity, clock time, CPU time, and spawned processes, the system neutralizes a wide array of threats, including starvation attacks, DoS attacks, fork bombs, busy-wait attacks, and memory exhaustion.

8.5 Results

8.5.1 Backend

The backend test suite is implemented in Jest, with tests co-located with the source files they cover and organised by module. Running `npm run test` from the `backend/` directory executes the full suite. All 33 test suites and 150 individual tests passed without failures.

Code coverage was collected with `npm run test:cov`. Several large categories of files sit at 0% unit-test coverage by design and would otherwise drag down any aggregate figure: the AdminJS admin panel (`src/admin/`), which is a configuration-driven third-party integration with no testable logic; HTTP controllers, which are thin routing layers covered by integration tests instead; DTOs and TypeORM entity classes, which contain no executable logic; and infrastructure bootstrap files such as `src/database/`, `src/logger/`, and `src/main.ts`. Excluding these, the modules containing testable logic show substantially higher coverage, as shown in Table 2.

The checker pipeline steps, which are the most security-critical part of the system, achieve 92% statement coverage. Both access-control guards reach 93–100%, providing high confidence in permission and authentication enforcement. The grading service reaches 53%; the untested paths correspond to edge cases in bulk export and administrative overrides that are exercised through the admin panel rather than unit tests.

8.5.2 Frontend

The `@cprog-ng/components` package is tested using Vitest with Playwright for browser-based rendering. Components are developed in isolation inside Storybook, with behavioural tests co-located with the component source. Running `npm run test:packages`

Component	Stmts	Branch	Funcs	Lines
`gradingLadder`	100%	100%	100%	100%
`deletionPopup`	100%	100%	100%	100%
`navbar`	100%	71%	100%	100%
`notification`	100%	88%	100%	100%
`listGroup`	99%	97%	100%	100%
`codeBlock`	100%	83%	100%	100%
`pagination`	93%	88%	93%	94%
`button`	83%	40%	100%	85%
`filterSelect`	72%	51%	90%	71%
`All files`	81%	62%	86%	82%

Table 3: Frontend component code coverage

from the `frontend/` directory executes the full suite. All 343 component tests passed across 65 test files.

Code coverage is collected for the unit test project using the V8 provider, which is supported in Chromium-based browsers via the Chrome DevTools Protocol. The overall coverage across component source files is 81% statements and 82% lines. Some selected components are shown in Table 3.

The branch coverage of 62% is lower than statement coverage since many components contain conditional rendering paths (for example: theme variants, disabled states, empty states) that are exercised visually through Storybook stories rather than tested by the unit tests. The Storybook suite does not contribute to coverage collection as it focuses on visual correctness rather than branch execution.

9 Deployment

This section shows the deployment process, environment, and operational considerations for the new CProg system.

9.1 Deployment Setup

The deployment setup for Cprog-ng is the Virtual Machine (VM) provided by our supervisor. We where assigned a Rocky Linux VM, where the SSH key of one of our team members was added. A new deployment is still partially manual. It first requires a push from GitHub to the university GitLab. This was chosen such that GitHub could function as a local testing environment and GitLab would be our deployment. We would then update the repository on the VM by syncing with GitLab. Updating the system is as simple as re-building the docker containers. The frontend can be updated mutation free, whereas the backend generates and executes new migrations when needed.

9.2 Environment

Due to our highly dockerized system, the environment for deployment has very little requirements. The only things that had to be enabled on the VM where docker and docker-compose, and the permissions to create external and internal network connections. Any other permissions and program required are all within the docker containers; requiring no further system permissions.

9.3 Container Orchestration

The system is orchestrated using Docker Compose, with different configurations for development and production environments. The main Compose file (`docker-compose.yml`) defines the core application stack, while `docker-compose-dev.yml` and `docker-compose-production.yml` provide environment-specific overrides and additional services.

The development stack launches the backend (NestJS), frontend (Next.js), and PostgreSQL database containers as defined in `docker-compose.yml`. When running in development mode, the `docker-compose-dev.yml` file is used to add Keycloak (for authentication and authorization) and a dedicated PostgreSQL instance for Keycloak. This allows for local testing of authentication flows without affecting production data. Nginx is not used in development. The frontend is served directly by Next.js.

The production stack is defined in `docker-compose-production.yml`. It launches the backend and frontend containers, each built from their respective production Dockerfiles, and a PostgreSQL database. In production, Nginx is included as a reverse proxy to handle HTTPS termination and routing between the frontend and backend. Keycloak is not launched in production by default; authentication is expected to be handled by an external or institutional provider.

Each service is defined as a separate container with an explicit network and volume configuration. Health checks and restart policies are used to ensure resilience. Environment-specific configuration files (e.g., `.env`, `.env.production`) are used to inject secrets and settings into the containers at runtime. This separation between development and production environments ensures that they are as similar as possible while still allowing for the additional tools and services needed for local development and testing.

9.4 Limitations

Even though the current deployment setup works well, there are still some limitations.

- **Manual steps.** Some parts of the deployment process, such as syncing the code and restarting containers, are still done manually. This could be improved by introducing CI/CD pipelines to automate these steps.
- **Single VM.** The system currently runs on a single virtual machine, which limits scalability and fault tolerance. Moving to a platform like Kubernetes[14] could help improve this.
- **Secrets management.** Sensitive data is currently stored using environment variables. Using a dedicated secrets management solution would improve security.

In general, the deployment process is clear and works as intended. However, there are areas where it could be improved in the future, as outlined above.

10 Future Maintenance

The future maintenance of the system heavily depends on the future vision of the system. The system is functional in its current state, and can be used for the course of Programming in C for the Bachelor of Electrical Engineering. Having all the required features for operation, archiving, and maintenance. The system will be maintained by our supervisor, who thus is already familiar with the system and its technical setup.

On the other hand, the system was built with extensibility in mind, and can easily be extended without changing the core structure if more features are needed. An example feature that might be useful to have is an option to include assets in unit tests, or to give the teachers an in-application modal to alter the current course.

11 Process and Communication

This chapter will focus on the meetings and other communication we had with our supervisor and within our group.

11.1 Supervisor Meetings

This chapter will focus on the meetings and other communication we had with out supervisor. It will also discuss the changes that meetings induced to our design and system.

11.1.1 Meeting 05-02

We had our initial meeting with our supervisor on the fifth of February. We met up with D. Abeln, and discussed what he expects from us and from the system we will be building. He gave us a high level overview of the current system and its problems, and the improvement points he had in mind. After that he also gave us access to the original system, so that we can see how it works, where it fails on technical aspects, and what we can re-use in the new system.

Furthermore, we discussed a high level overview of what the new system should entail and what it should look like. D. Abeln prefaced that he had no preferences or requirements as to style, tech-stack and other technical aspects. He empathised that *we* are responsible for what the final product must look like and that we will be there to assist only when strictly necessary.

Finally we discussed more general organisational things. We agreed that further meetings will be planned by the team when needed, and that they will be planned by requesting participation to a calendar event in D. Abeln's calendar. He also agreed to be available on email, and that we must email him if there is anything that needs to be cleared up.

11.1.2 Email chain 09-02 - 11-02

We had discussed in our first meeting (11.1.1), that we would send our requirements to D. Abeln once we had formulated them all. Upon sending them he responded with some feedback to our requirement. The main clarifications were;

- There is no need for a communication channel between Teachers/TAs within Cprog.
- There must be support for assignment templates.
- A student must not be able to change groups. Only an authorized user can move them around.

Apart from that feedback on our requirements he also mentioned three more things. Firstly, he wanted the system to migrate away from the hard-coded assignments, groups, and users. He wants everything to be configurable and editable within the Cprog-ng system. Second, he wants a very extensive user-role-permission system. He wants to be able to manage the permissions that a role has within the system, and wants to be able to manage a users' role directly. Lastly he mentioned that he would like to bring back the `valgrind` step. This step would run a memory analysis on the student's program, scanning for memory leaks and out of bound reads. Which are a common pitfall in **c** programming.

After this initial feedback we refined our requirements. We incorporated the feedback D. Abeln gave us and improved the requirements for the code checker. We then asked for his feedback again and asked to clarify the difference between a group submission and an individual submission. In the current Cprog system the user can choose to either hand in as a group or individual. This system was in place to prevent free-loading when a member of the group does not participate. He mentioned that he would like there to be

some system in place to still prevent this, but that it should not be our top priority. Apart from that he had a few small remarks on our requirements, which we re-incorporated into our requirements.

11.1.3 Meeting 11-03

On March 11, we held a midway progress meeting with our supervisor to discuss our current status, demonstrate the system, and address a few outstanding questions.

Our supervisor expressed appreciation for the meeting but noted he would have preferred more frequent updates. In response, we committed to maintaining closer communication throughout the second half of the project.

We presented the current version of the system, which was well-received. The only notable feedback was a minor issue where the code-syntax component rendered tab characters as a single space. Additionally, he requested access to the source code to familiarize himself with our tech stack, as he will be maintaining the project in the future. To provide this, the team setup a second remote at the university's GitLab which our supervisor can see.

Our primary questions revolved around whether the current user-role-permission model was comprehensive enough to meet his requirement for fine-grained access control. We also asked about the hosting environment and requested early access to the deployment virtual machine (VM). He confirmed he would provision a VM running Rocky Linux and notify the team once it was ready.

11.1.4 Email chain 12-03 - 13-03

After the meeting on 11-03 (11.1.3), we discussed what we had learned at the meeting and developed a new solution for our user-role-permission system. We then sent an email to our supervisor to confirm this approach.

Our supervisor agreed that our new user-role-permission system sounded reasonable. In addition, he suggested a new feature to complement the new system: role templates (section 6.1.4).

11.1.5 Meeting 20-03

As mentioned in the meeting on 11-03, our supervisor contacted Twan regarding the virtual machine (VM) that had been arranged. The team then decided that Twan would take on the role of system administrator and be responsible for hosting.

Following the initial email, Twan met with our supervisor on the 20th of March to discuss and set up the hosting environment. Initially, they were unable to configure the host correctly due to several misconfigurations within the VM. It was decided that Dorus would address these issues and notify Twan once they were resolved.

Later that day, Dorus emailed Twan to confirm that he could continue setting up the VM. However, there were still persistent issues with the VM's internal and external networking, which Twan was unable to resolve due to a lack of permissions. Upon request, Dorus made the necessary changes, and Twan was finally able to deploy the very first version of CProg-ng to the VM on the 24th of March.

11.1.6 Meeting 31-03

On March 31, we had a meeting with our supervisor to discuss usability and grading within the system.

A key point was that all workflows should be intuitive, assuming that users may not fully understand the system. Actions like saving should always give clear feedback, and sensible defaults should be used.

Regarding grading, it was decided that the system must be simple and intuitive for teachers, while still allowing flexibility. We chose to use the same grading formula that the university uses for exam. In addition, the system should support both mandatory assignments and a minimum number of completed assignments, with the option to use either or both.

From the teacher's perspective, only basic information is required for grading, such as name, student number, and final grade. Therefore, export functionality should support exporting per assignment set or per course.

Finally, it was decided that assignments are optional by default, with a flag to mark them as mandatory.

11.1.7 Email chain 01-04 - 15-04

On April 1st we sent out an email to our supervisor planning when to upload a full-feature version to the VS and GitLab. Additionally we asked for some clarification regarding how the final grading should be computed within the system. It was decided that a final grade should be computed, and should follow the general Exam formula provided by the university.

A full-feature version was uploaded the 10th of April, with the purpose of allowing our supervisor to evaluate and give feedback on the system and user flows. We identified the main workflows in our system for our supervisor to do usability tests on. No description of the workflows was provided, since the system should be intuitive enough to execute these flows. Following the testing, our supervisor provided us feedback on the 14th of April. He had discovered a few minor bugs, visual inconsistencies, and overall shortcomings. The team was able to promptly address and fix these issues following day.

11.2 Team Communication

Our primary communication hub was a dedicated Discord server, organized into various category-based text and voice channels. Beyond serving as our daily chat platform, these voice channels hosted all of our online meetings. To complement Discord, we utilized Trello for issue tracking and GitHub for version control. To keep workflows streamlined, specific questions regarding tasks or code were handled directly within their respective platforms via ticket comments or pull request reviews. Additionally, we integrated a Discord webhook with GitHub to provide real-time notifications for pull request activity, ensuring the team remained updated on all repository changes.

12 Evaluation

This section will evaluate the process of developing CProg, with reflection on both the process and the results.

12.1 Comparison with Old System

CProg-ng was developed as a complete redesign of the legacy CProg system, with a strong focus on maintainability, security, and extensibility. While the core functionality of automated grading remains unchanged, several improvements have been made.

Firstly, the architecture of CProg-ng is modular and based on a decoupled client-server model, in contrast to the tightly coupled structure of the original system. This improves maintainability and allows individual components to be changed independently.

Secondly, the new system introduces a permission-based access control model, replacing the rigid role-based approach of the legacy system. This enables fine-grained control over user actions and improves flexibility in course management.

Another major improvement is the redesigned checker pipeline. Unlike the previous system, which relied on less secure execution methods, CProg-ng uses NsJail to provide strong sandboxing, strict resource limits, and protection against common attacks such as sandbox escapes and denial-of-service attacks. Besides this, the checker is now triggered on a submission instead of running as a periodic job.

Additionally, the introduction of a course-based structure resolves a key limitation of the old system, where no separation between academic years existed. This allows multiple iterations of a course to coexist without manual data migration.

Finally, the frontend has been completely modernized using modern web technologies, resulting in improved usability, responsiveness, and maintainability compared to the legacy PHP-based interface.

Despite these improvements, some trade-offs were made, which are discussed in Section 12.3.

12.2 Team Performance

A major strength of our team was our willingness to engage in constructive conflict. We actively sought to critically evaluate each other's ideas and proposals in order to arrive at the best possible design. There were, however, a few misalignments that slipped past, resulting in significant changes and integration challenges within the codebase.

Database Setup At the very start of the project we needed to setup the database and its TypeORM modules. One team member opted to start implementation on this, however they were not yet comfortable with the chosen framework. This resulted in some duplicate work and generally taking longer than expected. Next to that, during this time the design of the models was also changed. This resulted in an even longer development time.

This could have been prevented by working out the design of the models earlier, allowing for a more linear implementation. Next to that, the development would have benefitted from doing that database in smaller chunks instead of everything at once. This would allow dependant parts of the project to start earlier and procedural feature rollout.

Admin CRUD An admin CRUD interface is a very useful, and also required part of our system. At first a team member was assigned to this and they slaved away for quite a while without ample feedback and communication to the team, and did not publish their branch. After some time another team member took a look at the ticket, since it was stationary for over week. They quickly found the framework of AdminJs which perfectly fit our needs. This led to the earlier work of the other team member being in vein.

Best would be that the team member communicated earlier about their slow progress, this would have resulted in the team finding a solution together, and not having to do work for nothing.

Permissions and Cleanup Nearing the end of the project we where done with most of the features and wanted to start finalising everything. This included both adding and enforcing the final permissions, and removing unused code. One teammate started working on adding the final permissions, but they were not up to date on what was to stay and what should be removed in the final cleanup, and they also did not communicate what they were working on exactly. This lead to the team member adding a lot of permissions which where no longer needed.

Simply communicating better within the team would have prevented most of the unused work, since the team member would have been up to date on what parts of the code where no longer needed.

Frontend Permission Refactor One major architectural challenge arose from the introduction of granular permissions. In our initial design, we planned for two disjoint ecosystems: a student system and a teacher system. The frontend team had built all the initial pages with this structure in mind.

During development, the backend team proposed an architectural shift: replacing the strict teacher-student split with a unified, permission-based system. In this new model, specific components of a page would require the user to have explicit permissions before viewing them, which the backend endpoints would then verify. Because of this divergence, the frontend required a full, blocking refactor that took multiple days to complete.

If we were to build a similar system in the future, we would prevent this by more clearly defining the core architecture and user-access models upfront. Additionally, establishing tighter communication between the frontend and backend teams would allow for earlier, more iterative pivots rather than requiring a large-scale system refactor late in development.

12.3 Limitations

Despite the improvements introduced by CProg-ng, several limitations remain.

Lack of horizontal scalability The current system is designed to run as a single deployment instance and does not support horizontal scaling. Although the architecture itself does not prevent scaling, no orchestration or distributed job execution mechanism has been implemented. As a result, the system may become a bottleneck under very high submission loads.

No mobile support CProg-ng was designed and developed with desktop devices in mind. Thus, aspect ratios like the ones of mobile phones are not supported. It was chosen to focus on desktops because of the time span of the project.

Role precedence The system does not enforce role precedence, meaning that lower-privileged roles are not restricted from modifying or affecting higher-privileged roles if they are granted permissions allowing this. An example would be, a teacher having role editing capabilities changing the role of the admin.

No advanced insights Some features, such as plagiarism detection and advanced analytics, were explicitly excluded from the scope of the system. Although this was a conscious design decision, it limits the system's functionality in a broader educational context.

These limitations highlight areas for future improvement and provide direction for further development of the system.

12.4 Conclusion

This report was used to demonstrate the design, development, and evaluation of CProg-ng, a modern replacement for the legacy CProg system used in the Electrical Engineering C-Programming course. Starting with the limitations of the original system, the project aimed to deliver a fully reimplemented platform. It should aim to preserve the core functionality of automated grading while improving maintainability, security, extensibility, and usability. To achieve this, the system was redesigned with a decoupled client-server architecture, a permission-based access-control model, a course-based structure, and a sandboxed checker pipeline using NsJail.

The resulting system addresses several important shortcomings of the original system. In particular, the new CProg-ng introduces a clear separation between course iterations, submission-triggered checker execution, a modernized frontend, and a more maintainable, modular and modern architecture. The implementation is verified by automated testing on both the backend and frontend, providing confidence in the correctness of key parts of the system. At the same time, the project revealed several areas for future improvement, including deployment automation, improved scalability on mobile devices, and support for more advanced educational features such as plagiarism detection.

Overall, the re-implementation of CProg can be considered a successful next-generation replacement. It contains the main functionality of the legacy system while establishing a stronger technical foundation for future maintenance and extension. Although some limitations remain, the project shows that the new system is not only a functional replacement, but also a substantial architectural improvement.

References

[1] Atlassian. *What is the Waterfall Methodology?* Atlassian. URL: https://www.atlassian.com/agile/project-management/waterfall-methodology (visited on 04/10/2026).

[2] Atlassian. *What is Agile?* Atlassian. (Visited on 04/10/2026).

[3] Agile Business Consortium. *MoSCoW Prioritisation - DSDM Project Framework Handbook*. Agile Business Consortium. URL: https://www.agilebusiness.org/dsdm-project-framework/moscow-prioritisation.html (visited on 04/10/2026).

[4] Nerd For Tech. *Client-Server Architecture Explained with Examples, Diagrams, and Real-World Applications*. Medium. 2021. URL: https://medium.com/nerd-for-tech/client-server-architecture-explained-with-examples-diagrams-and-real-world-applications-407e9e04e2d1 (visited on 04/10/2026).

[5] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Boston, MA: Addison-Wesley, 2003.

[6] mochafreddo. *Mastering Git: Commit Message Types and Git Flow Branch Naming*. DEV Community. Sept. 21, 2023. URL: https://dev.to/mochafreddo/mastering-git-commit-message-types-and-git-flow-branch-naming-1lbb (visited on 04/10/2026).

[7] Atlassian. *Software Testing Types: Unit, Integration, and End-to-End Testing Explained*. Atlassian. 2026. URL: https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing (visited on 04/10/2026).

[8] Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux Journal* 2014.239 (Mar. 2014).

[9] Python Software Foundation. *subprocess — Subprocess management*. Python 3.12 Documentation; Accessed: 2026-02-06. Python Software Foundation. 2026. URL: https://docs.python.org/3/library/subprocess.html.

[10] Samuel Laurén, Sampsa Rauti, and Ville Leppänen. "A Survey on Application Sandboxing Techniques". In: *Proceedings of the 18th International Conference on Computer Systems and Technologies*. ACM, 2017, pp. 141–148. DOI: 10.1145/3134302.3134312. URL: https://doi.org/10.1145/3134302.3134312.

[11] npm. *bcrypt*. npm, Inc. URL: https://www.npmjs.com/package/bcrypt (visited on 04/17/2026).

[12] OpenID Foundation. *What is OpenID Connect*. OpenID Foundation. URL: https://openid.net/developers/how-connect-works/ (visited on 04/15/2026).

[13] Refactoring Guru. *Chain of Responsibility*. Refactoring Guru. URL: https://refactoring.guru/design-patterns/chain-of-responsibility (visited on 04/15/2026).

[14] Kubernetes. *Kubernetes Architecture*. The Linux Foundation. 2026. URL: https://kubernetes.io/docs/concepts/architecture/ (visited on 04/14/2026).

A Workload Distribution

Week	Mart	Maarten	Marcus	Sander	Twan
Week 1 2 Feb – 6 Feb	• Design student system	• *Sick* • *Joined meeting (while sick)*	• Learning new system • Iterate class diagram	• Design student system • *Ear infection*	• Frontend Setup • Backend Setup • CI Setup • Docker Setup
Week 2 9 Feb – 13 Feb	• Design teacher system	• Learning new system • Configure TypeORM • Create database models	• Learning new system • Iterate class diagram	• Components • *Ear infection*	• Code Execution Setup • Submission Pipeline • Progress Chart Components
Week 3 16 Feb – 20 Feb	• Design Finetuning • Theme Setup • Components	• Further database work • E2e Database test • OIDC SSO Auth • Configure Keycloak for local sso testing • Frontend: help page components • Frontend: action buttons, icon library • Frontend: theme setup	• Add express session • Implement permission guard • Work on database services	• Components • Log-in page • *Ear infection*	• Logging System • Unit Test System • Theme Setup • Components
Week 4 2 Mar – 6 Mar	• Components • Theme Integration • Student Home Page • Submission Popup • Student Group Pages • Student Assignment Page	• Pagination and Search	• Add permission decorator • Work on database services	• Student Submission Page • Student users page • Components • *Surgery (affected next 2 weeks)*	• Grading System • Student Home Page • Student Submission Page • Components • Student Group Pages • Student Assignment Page
Week 5 9 Mar – 13 Mar	• Notification Page • Frontend General Page Layout • Components • Providers	• Database-agnostic search • AdminJS exploration	• Work on database services	• Components • Teacher home page • Teacher announcement page • Theme/component story refactor	• Submission Pipeline Hardening • Notification Backend • Submission SSE feedback • Documentation • Notification Page

Continued on next page...

Week	Mart	Maarten	Marcus	Sander	Twan
Week 6 16 Mar – 20 Mar	• Course Page • Teacher Group Page • Teacher Role Page • Settings Page • Components • Teacher Submission Page	• Reworked Permission System • Permission Guards / Decorators	• Work on database services • Add HTTP endpoints	• Components • Teacher users page • Dark theme • Teacher student overview page	• AdminJS • System Settings Widget • E2e tests • Teacher Group Page • Teacher Role Page
Week 7 23 Mar – 27 Mar	• Teacher Notification Page • Components • Dark theme • Bug fixes	• AdminJS: add resources • AdminJS: Course Builder	• Comment system	• Components • Teacher log page • Bug fixes	• Permission SSR frontend • Deployment • Bug fixes • Teacher Notification Page • Settings Page
Week 8 30 Mar – 3 Apr	• Bug fixes • Teacher Settings Page Update • Teacher Roles Page Update • Global Styling Refactoring • Toasts	• AdminJS: dashboard	• Add Missing Permissions	• Bug fixes • Win95 theme • Win95 component compatibility	• Group size settings • Grading system rework • Archiving system • Grading rework (WIP) • Admin Settings Functionality • Grading ladder setting • AdminJS extensions
Week 9 6 Apr – 10 Apr	• Bug fixes • Manual Testing Frontend • Specifications Poster • Refactors • User Management Admin • Report writing	• Poster Design & Creation	• Report writing	• Icon refactor • Bug fixes • Report writing	• Grading exports • University number • User CSV import • Bug fixes • Sandbox hardening • AdminJS extensions • Unit Test argv stdin
Week 10 13 Apr – 17 Apr	• Report writing • Bug fixes	• Poster (full) redesign and creation • Write SSO Guide • Bug fixes • Styled class diagram	• Report writing	• Report writing • Bug fixes • Make diagrams	• Report writing • System documentation • Auto-group creation • Bug fixes • Permission Tweaks • Database configuration

Table 4: Task distribution per week, Weeks 1–10

B System UI

This appendix contains the most relevant parts of the UI. There are many more pages available within the system, but the core functionality can be extracted from these.

B.1 Student View

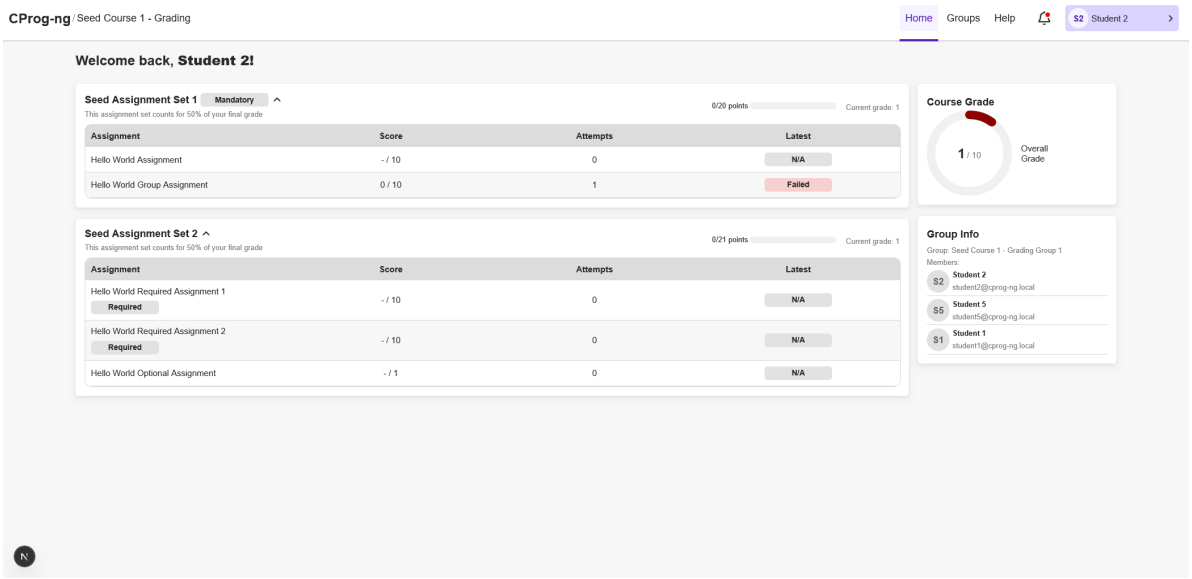

Figure 1: Student home view

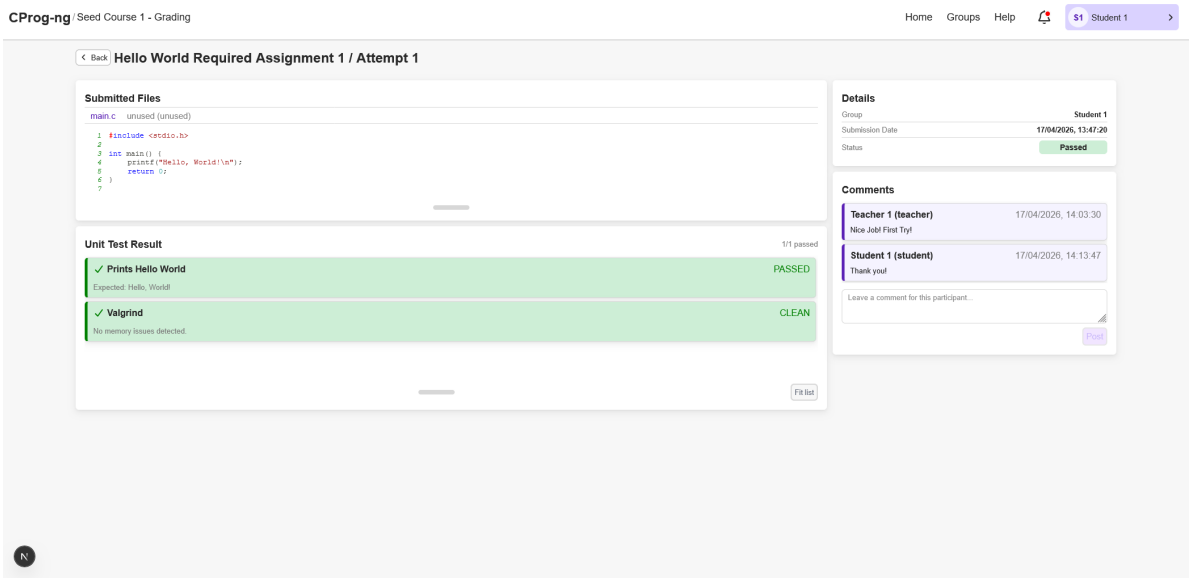

Figure 2: Student submission view

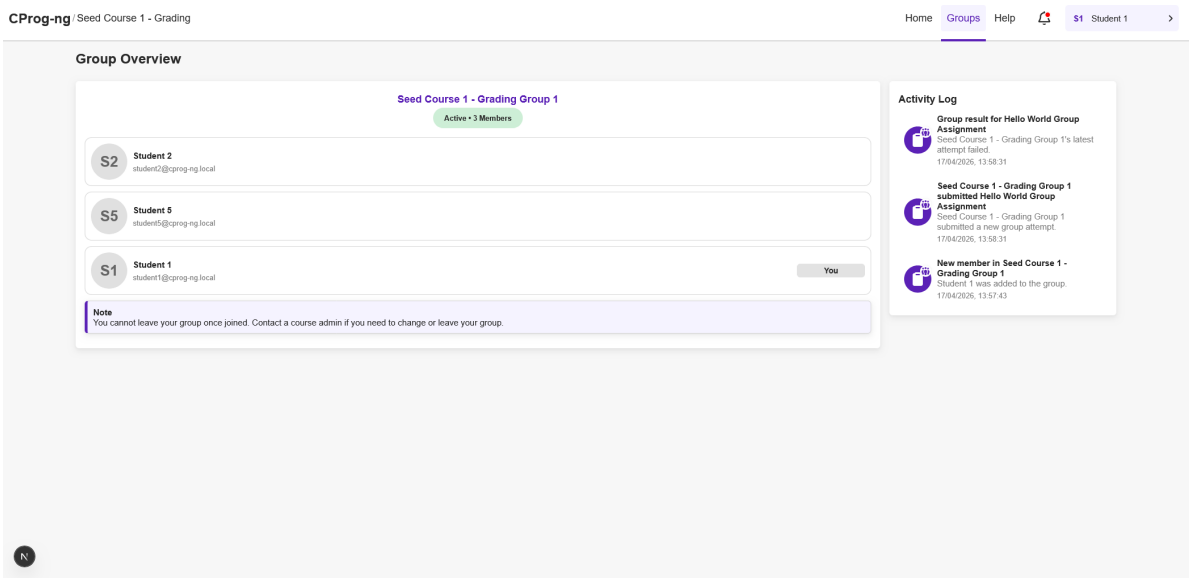

Figure 3: Student group view

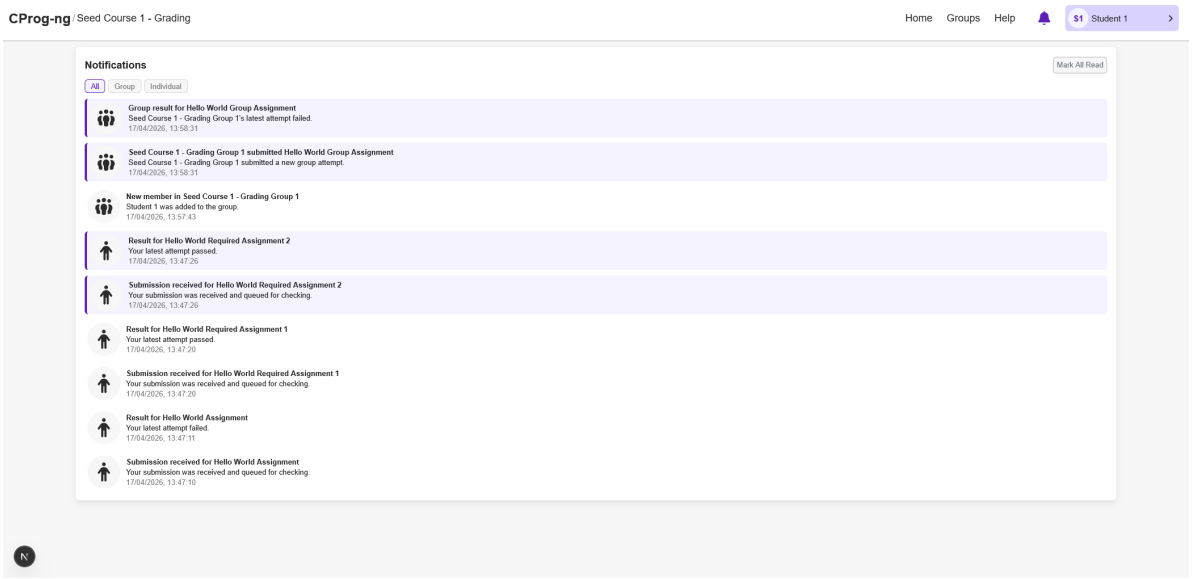

Figure 4: Student notification view

B.2 Teacher View

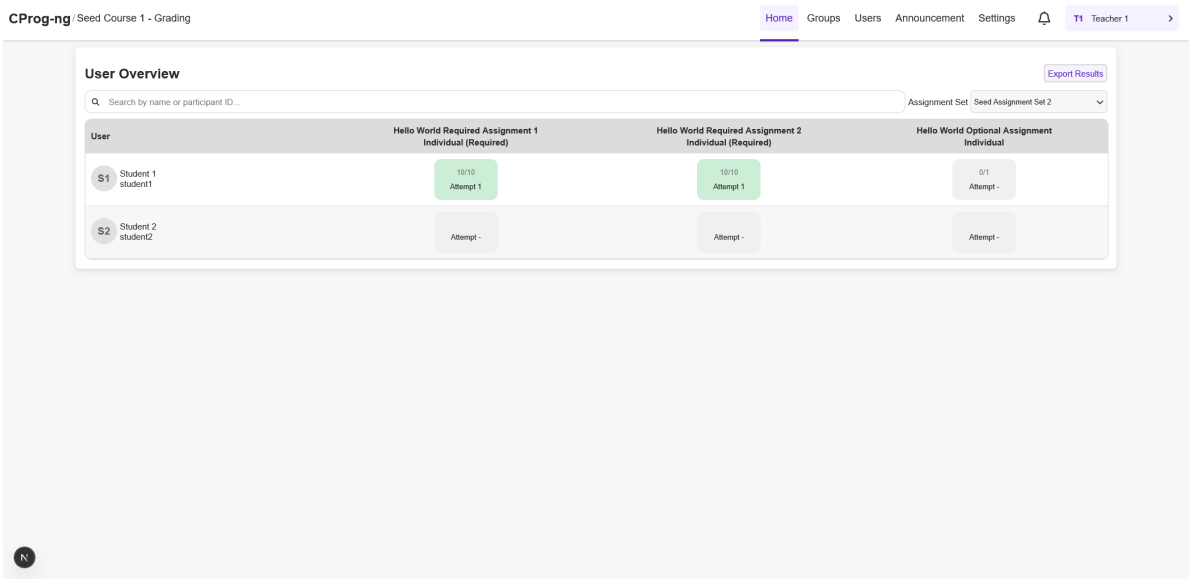

Figure 5: Teacher home view

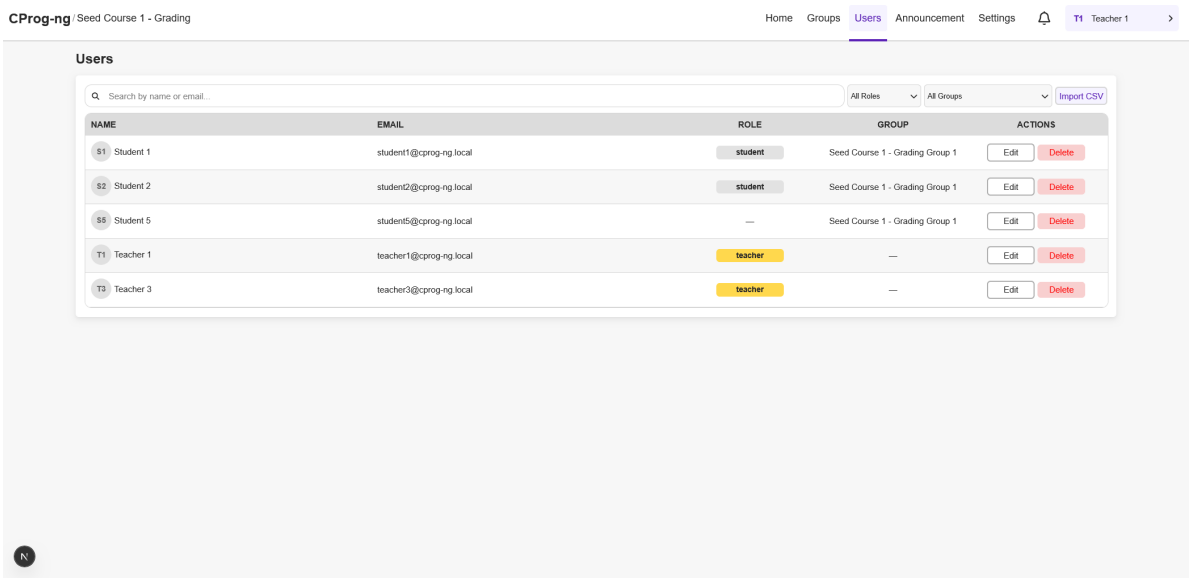

Figure 6: Teacher user management view

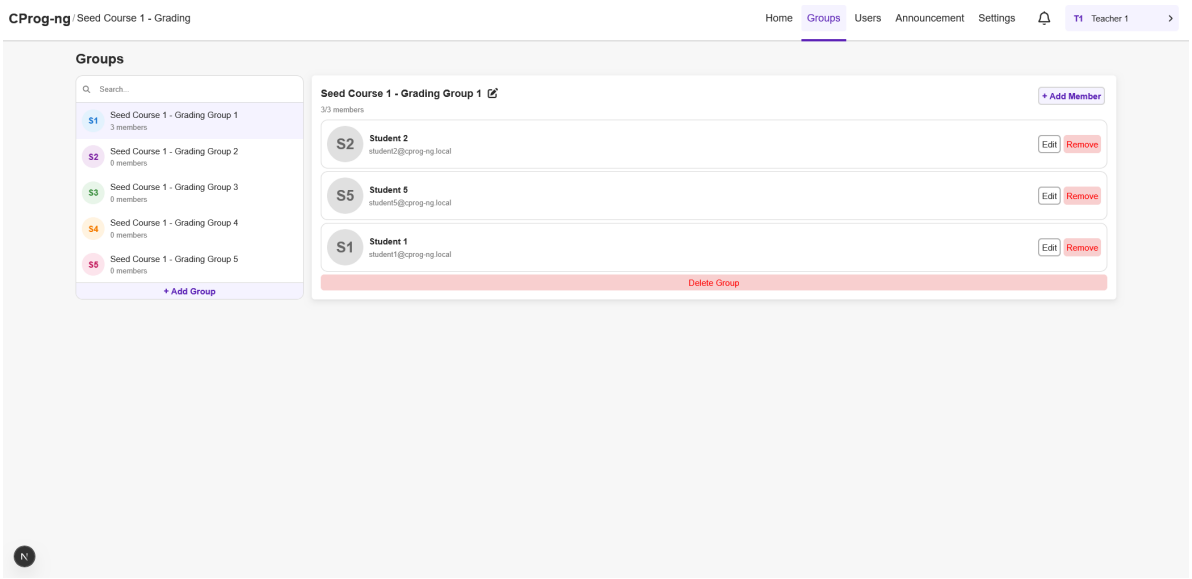

Figure 7: Teacher group view

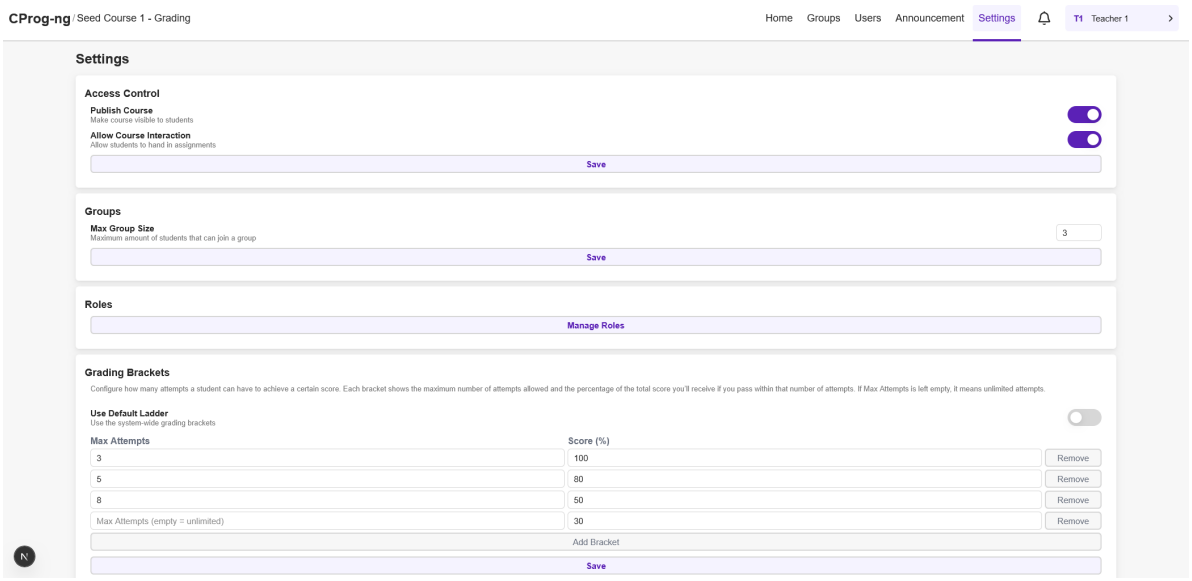

Figure 8: Teacher setting view

B.3 Admin View

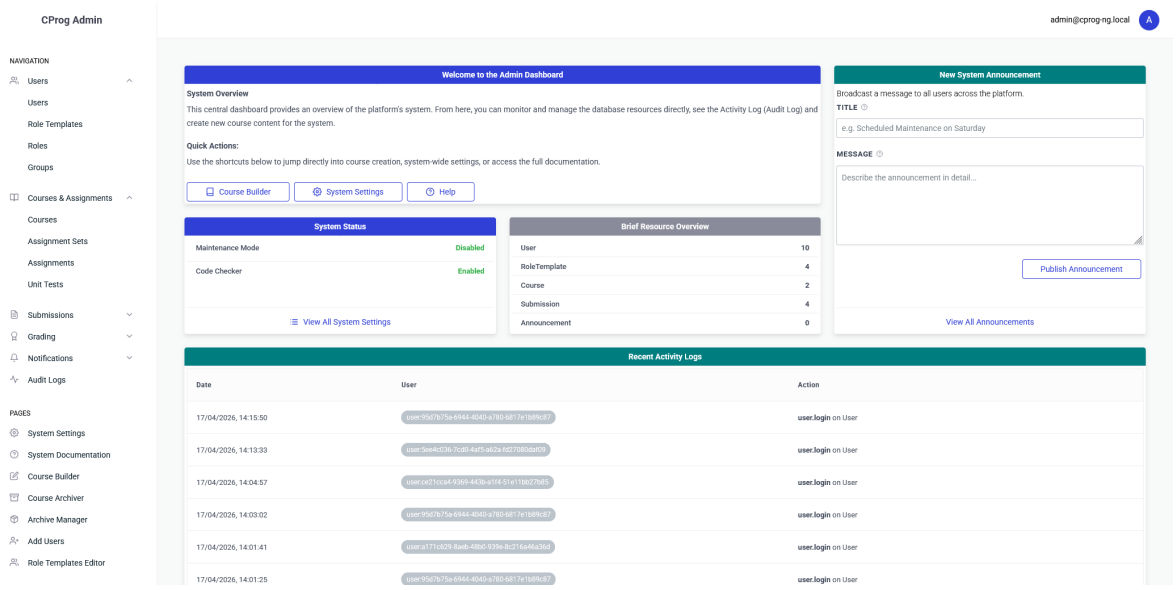

Figure 9: Admin home view

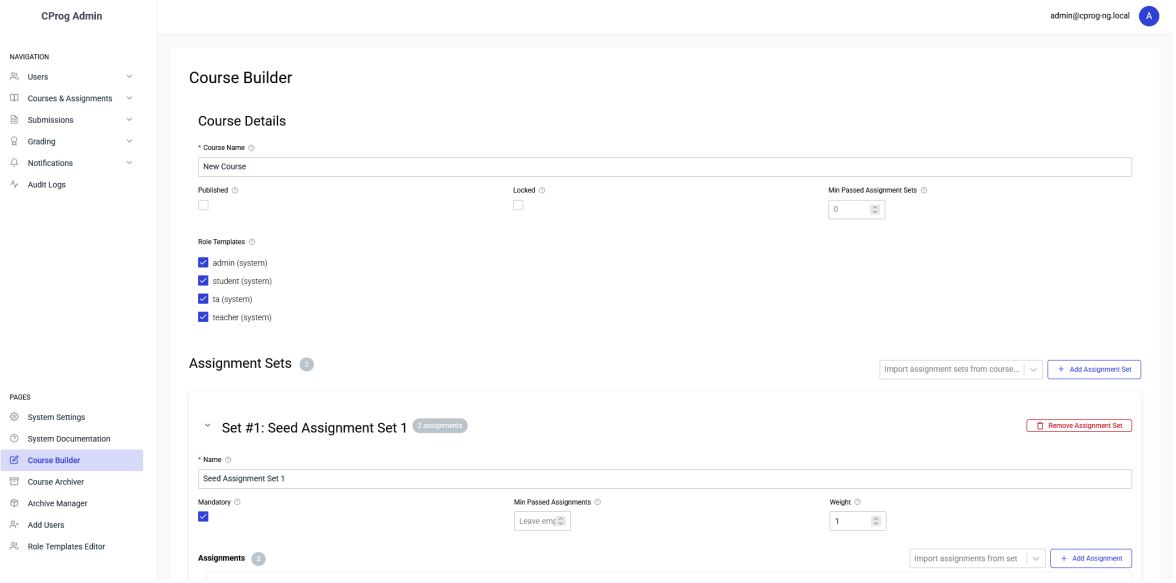

Figure 10: Admin course builder view

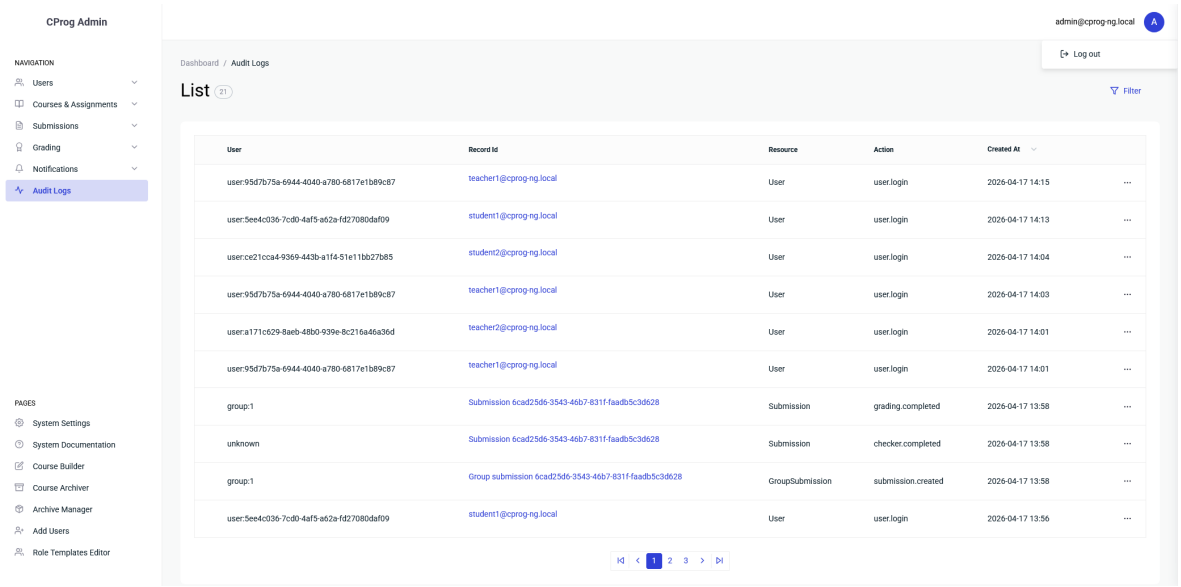

Figure 11: Admin audit logs view

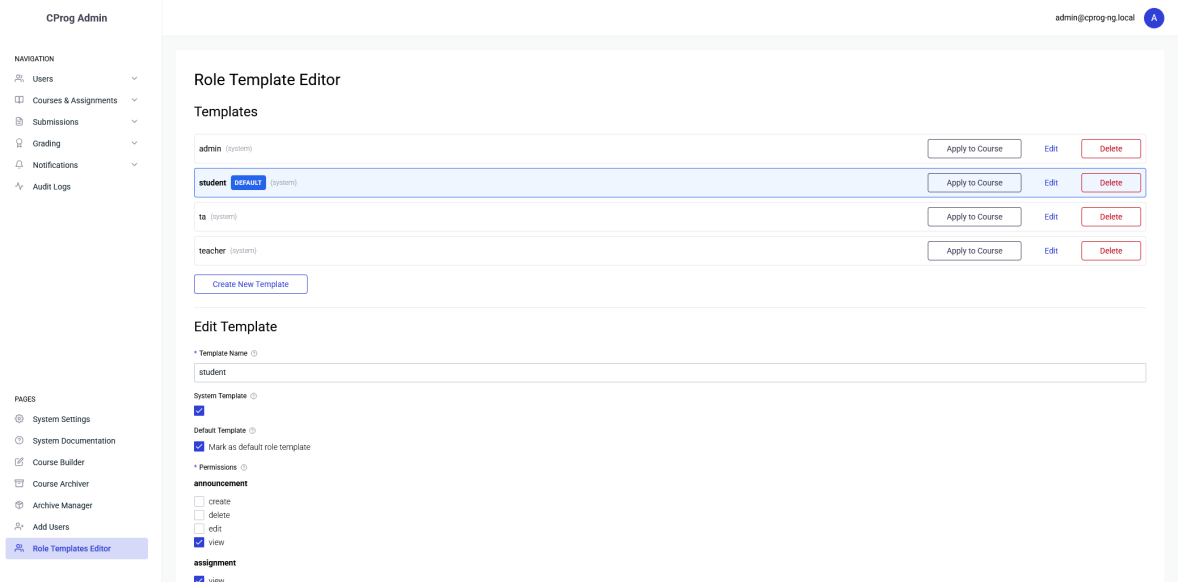

Figure 12: Admin role template view

C Diagrams

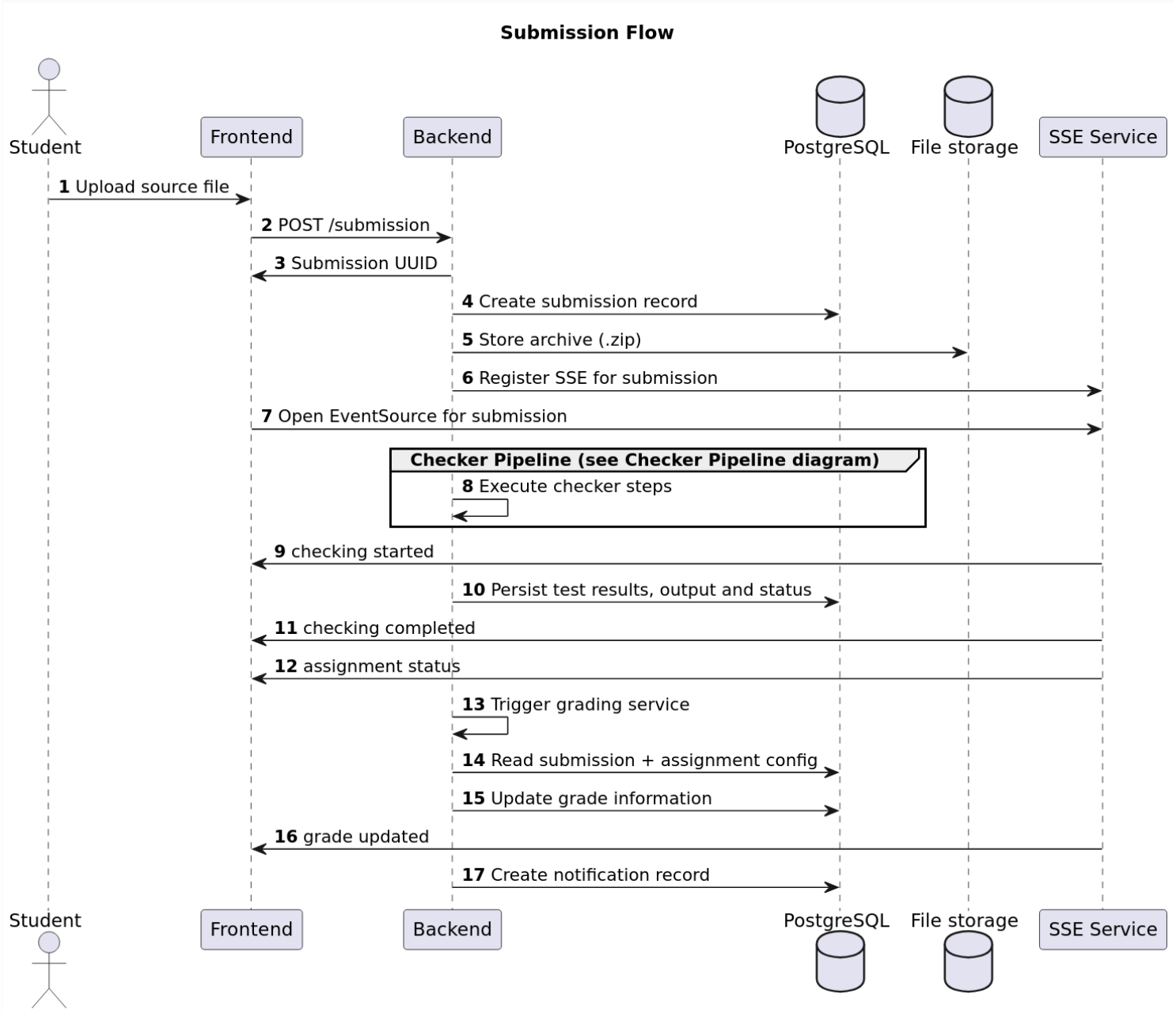

Figure 13: Data flow for starting from a students submission until new final grade is calculated and notification issued

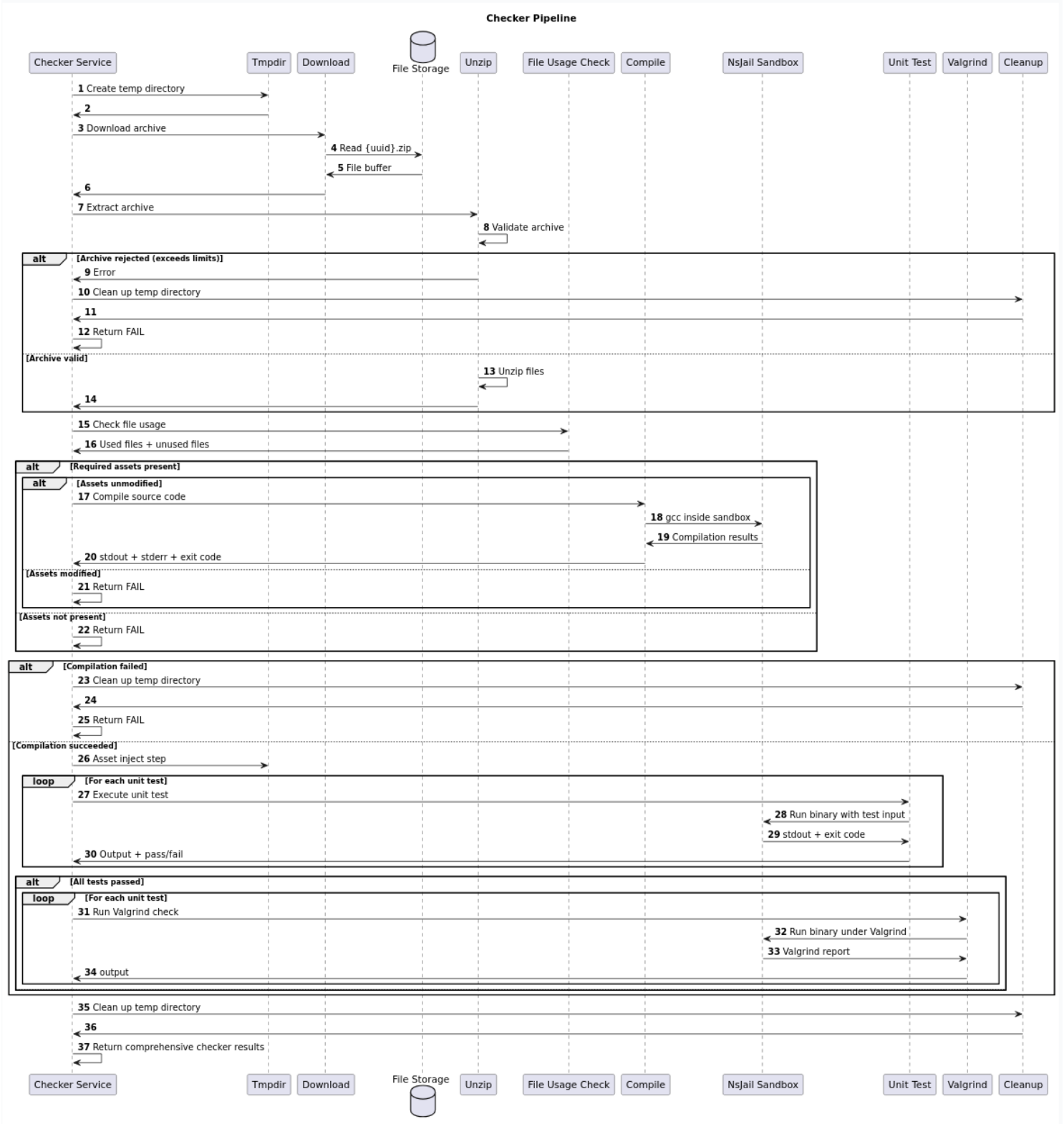

Figure 14: Data flow

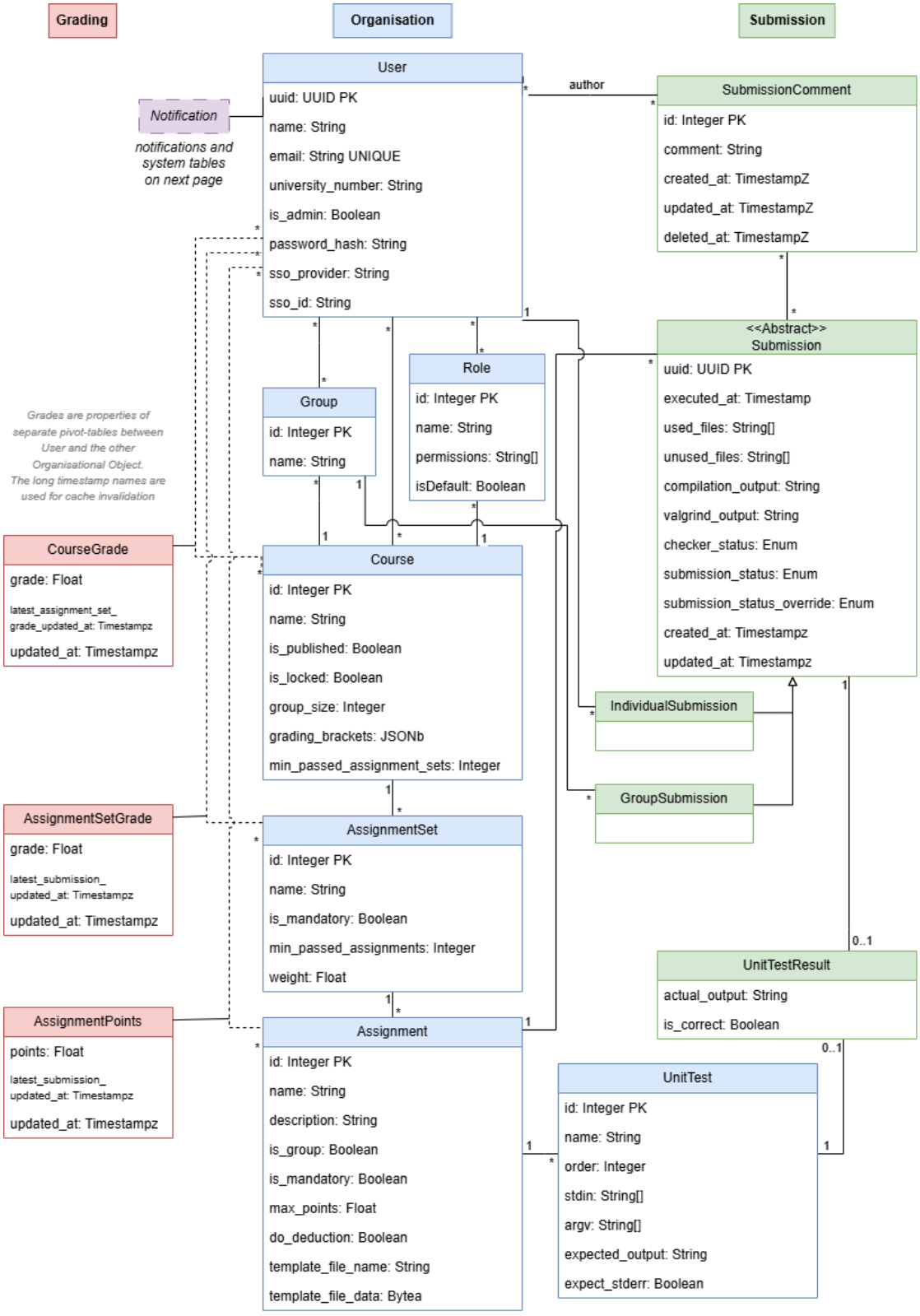

Figure 15: Class Diagram Part 1

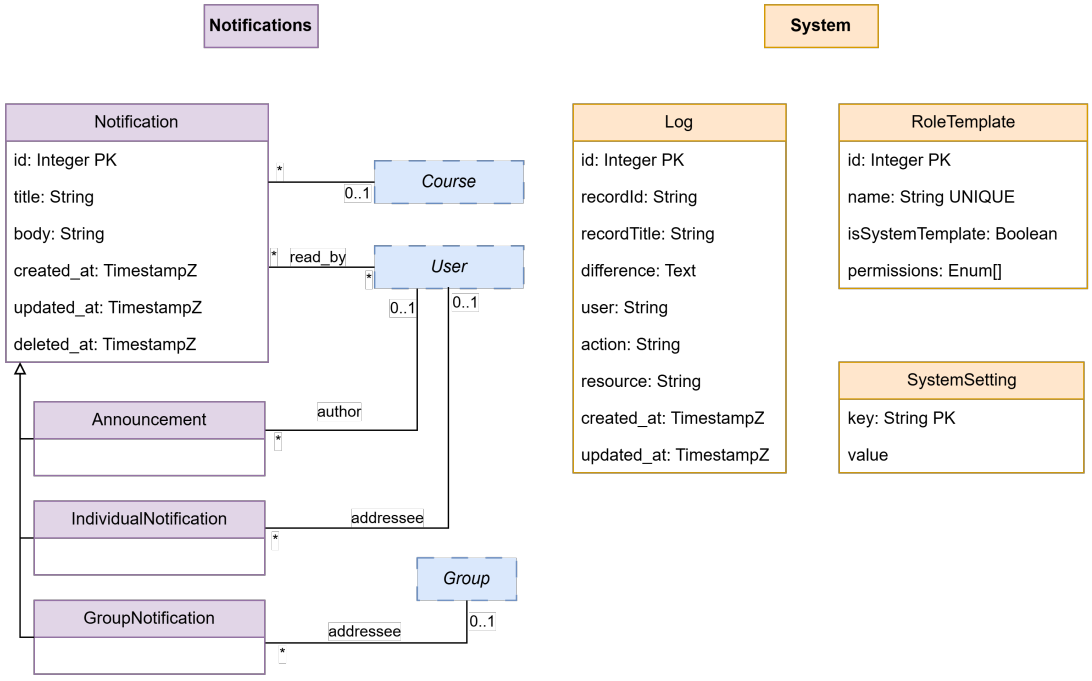

Figure 16: Class Diagram Part 2

D Ai Statement

During the preparation of this work, I and my fellow authors used LLM based AI models to review text, brainstorm ideas, and suggest grammatical revisions. After using this tool/service, we thoroughly reviewed and edited the content as needed, taking full responsibility for the final outcome.