CS3099 - Deliverable 4 Report

Sequoia Research: A Hub for Peer-Reviewed Open-Source Code

150000946, 190004947, 190007288, 190012580, 200002549

Jack Waller, Jordan Rowley, Walker Herndon, Benjamin Sanders, Sam Ball

March 31st, 2022

University of
St Andrews

Word Count: 10,195

1 Abstract

In a modern world where working remotely is becoming increasingly prevalent, the demand has increased for an online platform where open-source code can be treated like an academic paper.

We have created Sequoia Research: a journal within a larger decentralised federation with standardised protocols to allow users to submit code to multiple journals within the federation using the same credentials. Once submitted, it can be reviewed, approved and shared to create a collective hub for peer-reviewed open-source code.

2 Declaration

Contents

3 Introduction: 190012580

When the COVID-19 pandemic affected our day-to-day life in 2020, many regular daily tasks became impossible and new innovations had to be developed to adjust to daily life moving online. As a society, we became more dependent on software and as a result, the demand for research and technological advancements in that area increased. Despite this, it was still somewhat difficult to perform the simple process of getting code peer-reviewed and published in comparison to a research paper for other areas of science. Sequoia Research aimed to simplify this process by hosting a platform which combines code review and peer-reviewing into one operation and the result is a journal of published open-source code certified to be of high quality.

Our journal is a part of a federation, which has standardised protocols which enables other journals within the federation to interact with each other - users can use the same credentials to log into any journal as well as import and export submissions from them with ease. Since each journal is independently hosted and there is no central company controlling the federation, the system is decentralised and this provides further benefits - it is extremely difficult for an individual entity to gain control over what is published to the federation which helps maintain neutrality and remove bias from the platform.

When the project first began in September, we discussed the specification and what direction we wanted to go with it. We concluded that we were confident we could implement all the required functionality but we admitted at this early stage we were not set on what additional features we wanted to add were yet. Instead, all team members created user stories, where we pretended to be an end-user keen on using our platform, and wrote down what we thought an end-user would want to see our platform do. This gave us a good foundation to work from, and formed the basis for our product backlog [appendix 8] and prevented us from straying too far from our original intentions with the product.

We were able to implement all the required functionality as well as some extra features not listed in specification. However, before the details of the software development methodology and the product itself are expanded on, some of the commonly-used terminology will be defined.

3.1 Build and usage instructions

Detailed build instructions are available in the 'README.md' file in our accompanying codebase. To summarise, the 'make' command from our codebase's root directory will install, build, and run our Journal's server. You can then interact with our client side from this local host by navigating to 'localhost:22292'. Stopping the locally hosted server can be achieved by typing 'quit' into the terminal where you ran the 'make' command. To remove the installed dependencies and built files, run 'make allclean' from the same directory. Our hosted instance (if online at the time of reading) is : https://cs3099user07.host.cs.st-andrews.ac.uk/

3.2 Terminology

Post: It is used to describe one or more code files that have been uploaded to the journal by a user. Used interchangeably with submission.

User: A person who has created an account on the journal and covers all admins, reviewers, editors and regular users

Supergroup: A group of journals with agreed protocols that allow users of any journal within the group to post and log into any other journal. Used interchangeably with federation.

Endpoint: A defined URL that our front-end or, in some cases, the supergroup can query or send data to which performs a certain action based on the provided data and returns any relevant information. Used interchangeably with route.

MEAN: An acronym representing a technology stack, standing for MongoDB, Express, Angular, and Node.js.

Backend: The 'server-side' component of our Journal, i.e. the code base which is run on a well-known server, responsible for serving clients with our frontend user interface, and any other client requests.

Frontend: The 'client-side' component of our Journal, i.e. the user interface seen by clients, and the code to interact with our Backend server from those clients.

UI: An acronym: User Interface

TLS: An acronym: Transport Layer Security. This is a mechanism to provide cryptography over TCP, typically used with HTTP, and referred to as HTTPS.

School: Referring to the School of Computer Science at The University of St Andrews.

4 Software Development Methodology: 150000946

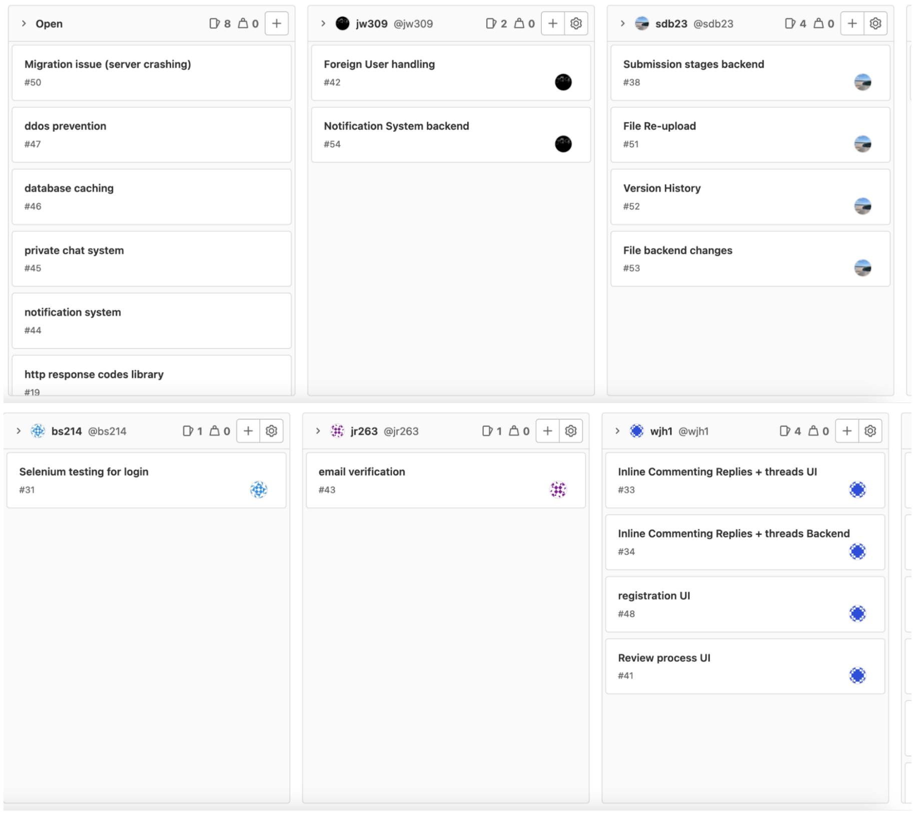
During the development of our Journal, we followed a continually evolving Scrum model to make iterative improvements to our codebase. At its core, this model consists of several 'Sprint Cycles', in which each of the increments are made.

Throughout this academic year, the exact components of these sprint cycles have evolved as we encountered issues, or found improvements. Having said this, the 'Product Owner' and 'Scrum Master' roles have remained throughout all of our sprint cycles, although we have rotated the responsibilities of these roles between team members after every sprint.

The Product Owner's role is to focus on the desired outcome of our system in terms of functionality and features. This is heavily reliant on knowing what the functionality of the system is to begin with! Therefore, before development of our Journal started, we as a team created a set of ‘User Stories’. These are desirable functionalities of our Journal from the perspective of several users of our system. This was then converted into a ‘Product Backlog’ [Appendix 5], a prioritised set of required features and functionalities for our system, created and maintained by the current Product Owner.

At the beginning of each sprint, tasks are taken from the Product Backlog by the development team, and placed into the ‘Sprint Backlog’ [Appendix 6], a list of tasks to complete by the end of the current sprint cycle. At this time, each team member estimates how long each task will take (in days), and the average value is used as an estimate of task complexity. Tasks are then assigned to team members, attempting to ensure each team member has equally complex tasks. These tasks are then replicated as gitlab issues, and organised in an issue board [Image 1] in the gitlab repository. This mirroring allows for easy issue management through commit messages, whilst still enabling a comprehensive report of the time spent on each issue, via the self reporting of time spent by developers in the Sprint Backlog. The Scrum Master is responsible for maintaining both the Sprint Backlog, and the gitlab issue board, as well as being responsible for running all types of Sprint meetings, and any issues or help development team members need throughout the Sprint cycle.

[Image 1] - Screenshots of the GitLab Issue Board



Open 8 0
Migration issue (server crashing) #50
ddos prevention #47
database caching #46
private chat system #45
notification system #44
http response codes library #19

jw309 @jw309 2 0
Foreign User handling #42
Notification System backend #54

sdb23 @sdb23 4 0
Submission stages backend #38
File Re-upload #51
Version History #52
File backend changes #53

bs214 @bs214 1 0
Selenium testing for login #31

jr263 @jr263 1 0
email verification #43

wjh1 @wjh1 4 0
Inline Commenting Replies + threads UI #33
Inline Commenting Replies + threads Backend #34
registration UI #48
Review process UI #41

The type and duration of the Sprint meetings we held throughout the development process has evolved as we found methodologies that suited our team, or to overcome issues we encountered, resulting in the following structure for Sprint 5 and 6. At the beginning of each Sprint Cycle, we hold a meeting in order to set up the following sprint. This is where the aforementioned Sprint Backlog is set up, and the Scrum Master and Product Owner roles are assigned. The duration of the Sprint Cycle is also agreed upon. During the last two sprints, we have settled on twice weekly 'stand-up' meetings (on Tuesdays and Fridays), where each team member discusses their progress so far, as well as any issues they have encountered, or if they need any help. In each of these meetings, we explicitly ask if anyone is struggling to complete a task, or will not be able to do so by the end of the Sprint. This gives us an opportunity to re-assign tasks to different team members. At the end of each Sprint, we hold a 'Sprint Review' meeting, where we discuss what went well, and what could be improved about the last Sprint.

Up until recently, we held all of our Sprint meetings on our team's Discord server. This was an invaluable piece of technology throughout this project - it enabled both voice communication and several text messaging channels to communicate between team members. In these text channels, on top of general team discussion, the minutes from each meeting have a distinct text channel, as well as channels for useful web resources from team member's individual research. These text channels are easily searchable with in-built filtering and ordering, enabling fast lookup of resources or previous conversations. Links to useful team documents are also held in a text channel, including the Product Backlog, the Sprint Backlog, and this report. These are hosted by Google's cloud based productivity suite, Google Docs and Google Sheets. These tools have been invaluable in the organisation of our Project - they offer real time multi-user editing, without the need to worry about merge conflicts or other issues, enabling a much smoother collaborative experience. During the covid restrictions, the voice communication provided by our Discord server has been very useful - it enabled us to easily communicate issues we encountered, as well as enabling a primitive version of Pair Programming via the screen share functionality. Thankfully as restrictions began to lift, in person meetings became possible, however the Tuesday meeting has remained on Discord, simply due to its utility.

Our meetings with our supervisors have taken place on Microsoft Teams however. This is due to the pre-existing channels dedicated for this communication, and easy integration with the school's pre-existing email and calendar implementation, without the requirement for our Supervisors to start using an unfamiliar technology.

When developing code, our team has followed the Feature Branch development model, where each team member works on a new feature in a separate branch, and submits a merge request once complete. These merge requests are then reviewed, tested and approved by at least 2 other team members before the feature branch is merged into the main branch. This ensures that the majority of our team of five has seen and reviewed every new piece of code that is in our final product, helping to alleviate bugs, and helping towards ensuring optimal design.

For the most part, our Sprint Cycles have been around three weeks long, resulting in a total of 6 Sprint Cycles for the development of our Journal. As we were all working part time on this project, we felt that this gave us an adequate amount of time to complete the required tasks without disrupting our other modules too heavily. The exception to this was during the winter vacation, where we tried a completely different sprint structure. To give each of us time to relax, we did not hold stand-up meetings, instead keeping each other up to date via summary messages on our team's Discord server. At the end of that Sprint, a more comprehensive review meeting was needed to explain the decisions of each team member. This worked well for the less intense vacation, however we reverted to the stand-up meetings after that.

The review process at the end of each Sprint enabled us to implement changes to our Scrum Methodology when we encountered issues. One good example of a beneficial change is the explicit opportunity during each stand-up meeting to highlight issues for team members, and to re-assign tasks when necessary. This was introduced after the first Sprint, and has been used throughout the rest of the Sprint Cycles successfully to avoid delays and help struggling team members, either by the re-assigning of tasks, or enabling the organisation of a quick call between team members to resolve a particular problem.

Task selection at the beginning of each Sprint was another problem highlighted by the Sprint Review process. During the fourth sprint, we encountered a delay due to tasks being dependent on the completion of another task in the same Sprint. The highlighting of this issue in the review enabled our team to plan to attempt to avoid these issues in the last two sprints, ensuring dependent tasks are completed in Sprint five, with their dependencies to be completed in Sprint 6.

Since Sprint 5, we have not changed our methodology, simply as we found that the structure we had created with the previous alterations has proved to be effective, resulting in the smooth running of our last two Sprint Cycles.

Our iterative Agile approach has had several benefits, mainly around our interaction with the rest of our Supergroup. As the API specification has repeatedly changed throughout the development of our project, we have needed to stay flexible: able to re-engineer changing functionalities. It has also helped with the design of our database, in a situation where we have frequently had to react and adjust to changes, Agile has allowed us to adjust our Schemas accordingly with minimal disruption - something which would be harder in a different software development methodology where a more concrete plan is set out initially.

The main downside to our iterative approach is that we have needed to rewrite a non-negligible amount of code, both as our aims shifted, and as initial design flaws were revealed. A good example of this is our re-implementation of login sessions from our use of the 'express-session' library to a json web token implementation. This perhaps could have been avoided if a more careful plan was made to incorporate the future Supergroup API requirements into our Minimum Viable Product implementation, resulting in the majority of one team member's work for Sprint three being to correct this issue.

Compared to an alternative Waterfall approach, I do think our use of the Scrum methodology has been successful - it has enabled us to stay on track, and overcome issues that we encountered effectively. A Waterfall approach would not have been flexible enough for us to keep up to date with the evolving Supergroup API, and too intolerant of delays we encountered.

In future projects, it may be useful to spend more time thinking about the specific design decisions of components before implementation, which would have eliminated the need for the aforementioned login session re-write. This could also have helped with the detection of dependent tasks, which would have prevented some of the delays we experienced. The review process at the end of Sprint Cycles has been invaluable, and we would certainly continue this practice going onwards.

A possible alternative to consider in the future would be to replace Discord with Slack. The application ecosystem could have possibly provided a much closer integration between team communication and both our Sprint resources and GitLab repository. The main reason we decided to use Discord however is the pricing - Discord's free tier offers much more in terms of channel management and permission system. Discord also has the ability for open source extensions similar to Slack, although the ecosystem for developers is not as advanced as Slack's. For longer term projects where Slack's pricing is not as big a problem, we would certainly consider it for future use.

5 Product Details: 200002549

We worked with the supergroup all throughout the project. At the beginning, this was mainly focused on agreeing upon the format to use for the collaboration whilst towards the end of the project, we mainly focused upon testing the standards and individual endpoints that are set out. This is not to say that the wiki never changed however. After devlieverable 2, the format for storing the supergroup wiki was changed as people were making changes without asking prior. It was at this point the idea for migration also changed, where you would have to fetch a zip folder instead of a link to a submission. We adhered to the supergroup wiki[11] as much as we could and, to the best of our knowledge, should return the correct data. One issue we had with the supergroup was due to a lack of testing. Testing on our own server was particularly difficult as it meant rewriting code and since we used the data from these endpoints, it would not test much. For example, if one endpoint returned incorrectly formatted data (which it did on different occasions), this form of self-testing did not highlight this bug. We had limited opportunities to test, as the testing we conducted with another supergroup was done quite late. To add to this, we often could not access the other servers as they were offline. With the servers that were online, we also encountered issues where the specification was not being properly followed by other teams and one journal hosting very large files which sometimes would crash our server.

5.1 Technology Used

We have used the MEAN technology stack for this project. In this stack, our backend is composed of a Node.js runtime with the Express library, our frontend represented by Angular, and our database by MongoDB.

5.1.1 Backend

As mentioned above, our backend runs using Node.js. This is JavaScript (JS) runtime built on top of Google Chrome's JavaScript engine. To build our backend, we decided to use TypeScript (TS). Javascript was initially designed to add support and interactivity to websites, but was never intended to be the backend foundation for enterprise-level applications. TS has been described as "JavaScript but with no surprises"[1] and is seen to have solved the ambiguity that comes with JS by using strict data types, which leads to more robust, stable code. TypeErrors are found at compilation rather than runtime, which aids development. TypeScript also offers compilation directly to JavaScript, enabling easy integration with both Node.js and the accompanying package manager, npm (Node Package Manager). Our group had a mixed knowledge of JavaScript before starting this project, which simplified the process of learning TypeScript. As our frontend framework also uses Typescript, this made it the obvious choice for our backend, preventing confusing the codebase, and enabling code reuse between our frontend and backend.

The use of a JavaScript based backend enables the use of the event driven programming paradigm. Node.js is a single threaded runtime, but provides 'asynchronous' behaviour via the 'event' abstraction, where code is executed when an 'event' occurs, rather than in other languages, where asynchronous behaviour is acquired via the use of threads. This fits well with our backend application, where our server only ever responds to client requests, which occur asynchronously at unknown times. Node.js handles this by queuing 'events', and a single threaded 'event loop' executes these queued events. This behaviour makes Node.js very scalable: the queuing system enables fair execution as load increases, resulting in a graceful slowdown.

Alongside Node.js, we also are using a nginx proxy to serve our application to the world wide web. The main reason why nginx was used was that the school already has a running instance, making the setup of our server straightforward, and providing transport layer security via the use of the school's certificates, without the requirement to configure our Node.js instance to use https. Alongside this security benefit, if we were to scale up our Journal, nginx could provide load balancing between running instances of our backend code from the same URI.

The package manager provided with Node.js (npm) provided a vast ecosystem of open-source libraries available to ease our development. Npm also provides vulnerability analysis for zero day attacks once they are detected, and automatic fixes when available. The major library we made use of for this project was express. This library wraps the included Node.js server functionality into an easy to use interface, providing event handlers for http requests made to our server. This simplified our codebase significantly, enabling the splitting of our endpoints into 'Routes' in different directories, and providing an interface for middleware code to be executed before the endpoint code. Express's middleware in particular was particularly useful for both login token handling and the logging of our server's activity.

In order to provide users with the ability to 'login' to our server, we use an implementation of

JWT's provided to the npm by the 'jsonwebtoken' library. Json web tokens provide a way for our server to 'sign' a payload representing a user's authenticated login. This token is generated with an expiry date by our server on a user's successful authentication, and returned to the user. This token can then be passed to the server by the client alongside other requests as a certificate proving that the user is who they say, certifiable by the server being able to verify that the token's signature was generated by someone who knows the same key that was originally used to generate the token. The 'jsonwebtoken' library provides an interface to generate, validate, and decode json web tokens provided to it. We use this library as an express middleware function to check if a provided token is valid, and pass the results to subsequent endpoints.

We also use two libraries to help us with log messages on our server, 'Winston' and 'Morgan'. Winston provides an interface to uniformly format log messages of different severity levels with a timestamp. It can also write these differing severity logs to different files depending on the severity level. Morgan wraps express's middleware to provide log messages (which we pass to Winston) for HTTP requests made to our server. The use of this functionality has vastly improved our ability to both debug, and inspect the historical use of our server through development, and also will help once deployed.

When it came to zipping and unzipping files, we made use of the yazl and yauzl libraries. This is a set of libraries for npm which can be performed asynchronously. Though node does have the gzip library, this was limited and would often crash and cause issues. Yazl worked fine for us, and allowed us to easily work with buffers which we often dealt with from requests.

5.1.2 Frontend

Angular is a framework used for building client web applications. We chose to use this because it is TS-based and is designed with web applications similar to ours in mind. Although no one in our group had any experience with Angular, we felt as though it fit our use case well. We felt the extra time spent learning it would be worth the potential increase in the quality of our site. Additionally, none of us had much experience with front-end development in the first place so there was not another framework we were more comfortable working with.

As a result of our lack of experience with front-end work, development was initially very slow. It took a lot of time to get the front end working seamlessly with the backend, but the more we understood it the better and more powerful it became. Angular uses components to build a site. Each component contains an HTML, CSS, and Typescript file and components can be linked together through HTML and Angular routing. Typically there is a separate component for every page, but components can also just be segments of a page. For example, the navigation bar is its own component which dynamically loads other components as different pages are selected on it. Angular can be very powerful and flexible in this way, but did not take advantage of this as much as we likely could have. For example, the post component became very bloated by the end of the project because it contained large amounts of code for loading the files of a submission, comments, permissions, and more. This could have been split into a few smaller components to keep the code more well organised.

One important aspect we focused on with the front-end was maintaining consistent styling throughout the site. This was difficult to do with multiple different people working on the front-end in various areas throughout the project. One library we found which helped greatly with this was Angular Materials. This library provides many different pre-made elements and icons which we were able to use throughout the site. By using these much of the styling was standardised by default. Anything that did not use Angular Materials was made consistent manually. We use standardised borders, buttons, colours, etc. throughout the site to keep everything looking consistent.

5.1.3 Database

Though none of us had used MongoDB before, we ended up using it for our database and connected it using mongoose in our backend. MongoDB is a popular NoSQL database program which we choose for its functionality, flexibility, and cross platform support. Though often used in conjunction with MongoDB, we did not use containers for the backend, due to our running of the system on the School's host servers, which only allow the use of podman. This would require a complex configuration to set up our MongoDB container on the same virtual 'network' as our server, or to expose either the database or server network interface to the system's local network port/s.

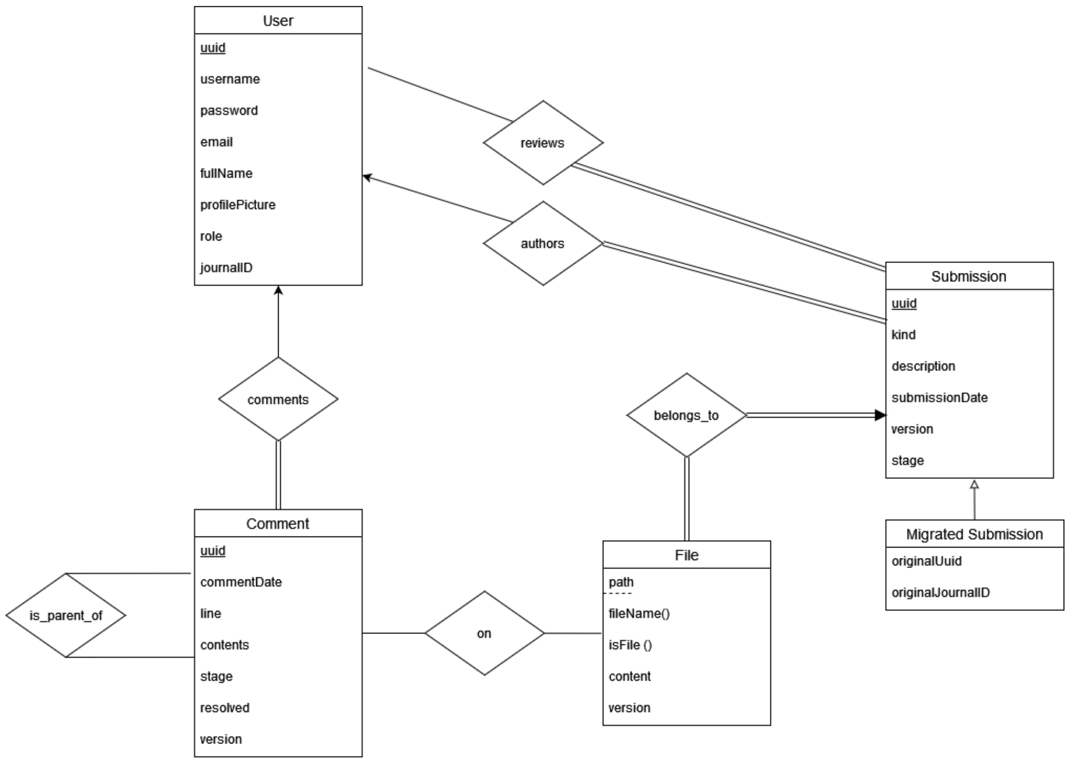
We therefore wrote several bash scripts to handle the installation, starting and shutdown of our database executable. We did not want to host anything externally to the School, as we wanted to maintain control of our codebase. However, hosting our database exclusively on the host server would cause issues as the schema was constantly changed, and since many people were working with different versions of the database, this could easily cause inconsistencies and errors. With the use of our installation script, this allowed developers to locally install and host the database alongside the server backend.

Throughout development we encountered many issues with mongoose, partly down to our inexperience with it. MongoDB and mongoose offer a lot of functionality which we did not use and often found that the documentation was not helpful so we did not make full use of the database as we could have. For example transactions were hard to set up and were not generally worthwhile, due to time constraints. Population was a feature which would allow the collections to be more organised, and reduce the number of direct queries to the database but were limited by the use of uuid as an index, instead of using mongoose's inbuilt ids, which made making use of this feature harder. Mongoose often operates with promises, which some group members struggled to use effectively which ended up causing delays in certain functionality of the database.

After our iterative development, our database has resulted in the following ER diagram [Image 2]. One abstraction in this diagram is the 'User' entity. In our implementation, we have split this entity into two database clusters for security purposes. Although the notation used is for a relational database, it is a good representation of the document based NoSQL MongoDB.

In our current implementation, once deployed, the scalability of our Journal would be directly linked to the performance of the MongoDB executable installed, which would be our major bottleneck, as multiple Node.js instances would be querying the same database. This could be solved in the future by installing MongoDB executables for each of the entities represented in [Image 2].

[Image 2] - An ER diagram which represents our current database



Silberschatz et. Al notation, CS3101

5.1.4 Building

There are downsides from using this technology stack, the first of which is the build process. Our TypeScript backend requires compiling to JavaScript before execution. Similarly, our Angular project has a build phase, where the TypeScript is compiled, and then compressed and optimised alongside the HTML and CSS into files that can be statically provided by our express server. Our Database also needs specific scripts to install, initialise, run and stop.

In order to manage the setup, running, and cleanup of these processes, we used a Makefile. This enables very simple setup and teardown of our system with a single command for each.

A simple Makefile itself does not help with the portability of our system however. Firstly, version 14 of Node.js is required for the Angular installation we are using. Secondly, installing the correct version of our Angular installation is dependent on using version 8 of npm. We solve both of these issues using the 'npx' utility provided with all of the versions of Node.js I have tested. This utility enables the installation and running of other npm libraries with specific versions of Node.js and npm for any version of Node.js installed. As Node.js and npm are themselves npm libraries, this also solves the installation problem for Angular mentioned above.

The other portability issue is our MongoDB executable, which is OS dependent. As our Makefile uses BASH commands, using a separate set of commands to handle Windows was seen as unnecessary, especially as our development team uses either Linux or MacOS to develop, and the deployed server would be running on the School Linux hosts. Therefore, we only needed to account for Linux and MacOS when installing our MongoDB executable. This was solved by checking for the currently running OS in our install bash script for MongoDB using the '$OSTYPE' environment variable.

Another problem that we needed to overcome with building was our configuration file. For ease of development with TypeScript, our main configuration file is a json file, which made parsing for use in our bash scripts problematic. This was solved with a simple custom python script, 'getKey.py', which we can call easily from our scripts.

We also wanted to be able to smoothly shut down and reload our server interactively via a CLI, and re-compile if changes were made whilst the server was running. As re-compilation was needed, this required communication between the Node.js executable and the Makefile. We therefore solve this problem with a bash script 'start.sh', and by managing the exit codes of our Node.js executable (with a simple CLI). By calling our Node.js process from start.sh, we can listen to the exit code, and either stop the process if all was ok, or rebuild and reload the server otherwise. This functionality has the beneficial side effect of minimising the downtime of our server on crashes, as if a crash were to occur, the start.sh script would simply re-launch it.

6 Currently Implemented Functionality

6.1 Required Functionality

Our website allows for users to easily create submissions and upload multiple files and .zip files. A user can then post comments on the submission or specific files, either as a general comment or on a specific line. Submissions go from an unpublished state, to a 'in review' state. Once in review, reviewers can check over the contents and suggest changes. Once the code is approved, editors can then approve the changes to submit it to the journal. Users can search the journal for specific submissions and view their contents.

6.2 Supergroup Interaction

To comply with the supergroup, we implemented a series of endpoints to return the appropriate data. We aimed to use these endpoints as much as possible in our implementation as it would reduce the amount of endpoints which were needed. It also had the additional benefit that it would allow us to use other supergroups endpoints to show data from them. An example of how we leveraged this was being able to fetch other supergroups submissions and display them like they were our own. We did use our own endpoints when required e.g. we wanted to be able to store comment threads. However, we made sure we used a fallback so that if it were a foriegn post, our front-end would still correctly fetch the comments. Another case where we could make use of the supergroup endpoint was using a zip file endpoint. This was intended to be used to help migration though we also used it to allow for a user to download a submission they could access. This also acted as a good fallback for migration if fetching the zip file did not work. If the zip file endpoint is not implemented, you could still browse the submission from a foriegn journal though reviewing it on our end would not be possible.

We interacted with the supergroup through a series of weekly meetings in a teams channel. Once most of the specification was designed, we then split into groups of 3 for testing. We had issues with the supergroup initially as the initial specification was too ambitious, aiming for things such as logging into other journals, which was agreed upon by. For MVP, most endpoints at the time were implemented, excluding the protocol for logging in.

Despite our focus on adhering to the protocol, there have been some slight deviations. It was recently discovered that most groups had used numbers instead of strings to store the issue and expiry time for JWTs, since most libraries are implemented this way. However, this was changed so that the majority of groups would not have to change their implementation. Another small deviation would be comments as they can be in a markdown format. This was not a high priority for us and, looking at most of the data from other groups, they also did not format the comments this way. Even if the data was sent in a markdown format, it would not be too bad, as we would display the raw text for the comment instead which is still generally readable.

An issue we faced initially with the supergroup was that the specification changed quickly as new ideas were proposed, and then quickly dismissed. This included specific lines for supergroup comments and replies, hence why we had to create a journal specific endpoint for this as it was something we wanted to implement. Once the specification was agreed upon, there was a meeting to decide the groups for testing. This also caused issues, as it took a while for the other groups to get back to us, with only one being there for testing in the end.

A specific example of the above was with our development of the ability to log into our Journal with user credentials from a different Journal in our supergroup. This required asking the other Journal to validate the provided credentials, and then parse the response to ask the journal for that user's details. As the other supergroups did not immediately respond, development of this was done without their initial explicit co-operation. The first challenge was finding a team whose server was online. Once this was achieved, we found that the specification for communication with other teams in our supergroup was not exactly followed by every team.

We attempted to counter this by instead comparing the keys of the json response with a regex, which would accept both the API specified by the supergroup, and a wider range of keys that could have been sent by other teams. Once we did organise a testing meeting however, we quickly found and solved the bugs with this feature.

However, working with the supergroup has mostly gone well. It has forced us to make our implementation flexible and we have been able to successfully leverage some of what it offered to expand our design.

6.3 Extended Functionality

Submissions can be reuploaded, at any given stage of the review process. In the case of published submissions, these will be then set back into review so the changes can be approved. To track the changes made to a submission, the previous files can be browsed which could give deeper insight for the review process. The idea of versions for our project came from group 22, though this was solely for the idea and not the code or implementation ideas. Comments can reply to other comments, though this is limited to a singley nested thread as it makes the UI and fetching more manageable. Threads can also be resolved, and will not show up in the usual comment location. Instead, they will appear at the bottom and have the option to be unresolved, in case the comment thread becomes relevant again.

Any browsable submission can be downloaded from the website. This relies on a user being able to access the submission. For supergroup submissions, this relies upon the other supergroups endpoint working as intended.

6.4 Accessibility

It was an important concern of ours that our website was as versatile as possible. To achieve this, we made sure that every image on our site had alt text so that users with vision impairments were able to determine what every image contained. User story 31 refers to this. We also wanted to make sure that the site worked for mobile users. The majority of users were going to be desktop users and other open source websites such as JOSS had optimised the desktop experience, so we did the same. Nevertheless, there is no loss in functionality on the mobile interface and adds to the accessibility of our website.

6.5 Security

Security was a priority for our journal and where possible, we followed the OWASP[2] Foundation's and, at times, the NCSC's[3] advice for different aspects of web development. From the beginning, we therefore ensured that all passwords stored were hashed and salted before storage in our database.

After MVP, we started working on ensuring our data was kept secure for our users. Our first step was to use the mongoose-santize library after it became apparent that a mongoose injection vulnerability was possible. This would involve passing maliciously crafted data to an endpoint which interacted with the database. This data would trick the database into returning data it should not. In extreme cases, this would provide an unscrupulous attacker access to the entire contents of our databases.

To minimise attack vectors, and to prevent some timing based attacks, we split the User database, with one half containing usernames and passwords, and the other containing the rest of the user's information. By doing this, fewer endpoints would be exposed to username and password data, ensuring the endpoints that did could have a greater portion of attention paid to them.

We also used the database to store our files as it helped with ensuring correct permissions and it helped avoid potential vulnerabilities such as a local file inclusion, where a user could maliciously access our server files, potentially leaking sensitive information e.g. passwords.

We also looked briefly into other attacks that could be discovered although we were not able to find much in our web application. By storing JWTs in session storage, it meant that tokens were protected against CSRF attacks (where a user's token can be used to perform unwanted actions). We tried made sure that our endpoints were properly secure and used validation on both front and backend to ensure that even if the user managed to bypass our client-side filters, it would not affect our backend, operating a system akin zero-trust, where we have to assume all requests are bad unless they can be verified.

Having mentioned the above, our application is still vulnerable to some vectors of attack. The main reason that we have not resolved these vulnerabilities is time, however for the expected application of our Journal, we are not expecting these attacks.

The first of these vulnerabilities is cross site scripting (XSS). This is when an attacker injects '$\langle script \rangle \langle /script \rangle'$ into strings provided in requests to the server, which the server would then unknowingly send to an unsuspecting client, possibly executing code on other client's machines. This is particularly tricky to stop, mainly as our Journal is intended to review and publish code, and these tags could be there perfectly innocently. One possible bypass is to replace these tags with a known string server side, and when the client receives this string, it could display the correct script tag, escaped.

The second vulnerability is the variety of Denial of Service (DOS) attacks. Some of these attacks will be handled by the School's nginx instance, but without testing (which we have not done for obvious reasons) this would be hard to confirm. There are npm libraries available to help with some of these attacks (mainly rate limitation methods), although we are not sure how effective some abstract attacks (i.e. Slow Loris) will be.

The final vulnerability we are aware of is the uploading of very large files to our Journal, or to other members of our supergroup. These files could quickly overwhelm the storage capabilities of the hosting server, and very large files cause our client UI to struggle to load certain projects.

This could be overcome by imposing a server side check and limit on file sizes, and the quantity of data uploaded by one user or IP address, however time concerns prevented our implementation of this.

7 Changes From Initial Plan: 190012580

To our surprise, there were few major changes of direction since the third deliverable was submitted. We feel that getting a better understanding of the project as we progressed through it helped with this. That being said, the one change we did make was regarding comment threads. We initially planned to have tree-style reply threads where there could be a chain of comments all replying to one another , but we felt that this was not fully necessary to implement as it the nature of the peer-reviewing meant that threads would rarely be used. We also did not want to clutter the interface which a large thread of replies would be capable of doing. A combination of this and the fact that more important features took priority, this did not get implemented, and we designed the page so that only top-level comments could be replied to. We were happy with this decision

In terms of changes since the second deliverable, these are all outlined in the following four paragraphs copied from the third deliverable:

A handful of changes have been made to our overall plan since the second deliverable (MVP) was submitted. Our focus was centred on ironing out issues with our features that were already implemented before implementing new ones. One of the main changes to our plan was regarding the storing of sessions - initially we were implementing authentication and sessions using the express-session module, however as the supergroup communication was done using JSON Web Tokens (JWTs), we discovered that this brought on unnecessary complexity to our system, and therefore decided to reimplement our own server sessions using JWTs. In the short-term, this meant that backtracking had to be done and elements of the system were rewritten. In the long term, however, this will simplify the implementation of further features - easier development leads to better quality code.

We also decided to invest time into cleaning up the backend typescript files in order to make them easier to navigate as they grew in size, complexity and functionality. For the second deliverable, all endpoints were contained in a singular server.ts file. Functionally, this was satisfiable for MVP, but was hard for a programmer to navigate. To solve this, we created a 'routes' folder and categorised each endpoint into TypeScript files which split the endpoints into files such as supergroup.ts and login.ts files.

Security has been more of a concern since the MVP. After discovering that our endpoints allowed for a mongodb injection, we have taken the time to ensure we sanitise data whenever it could pose a threat. We also separated user credentials from other user data into another database. This reduces the exposure of these credentials only to necessary endpoints, hence reducing potential avenues of attack on the most sensitive data. Other ways the database has changed is to do with which data is stored. We now store files in its own model, instead of in a submission, and many of the schemas have been modified for extra functionality.

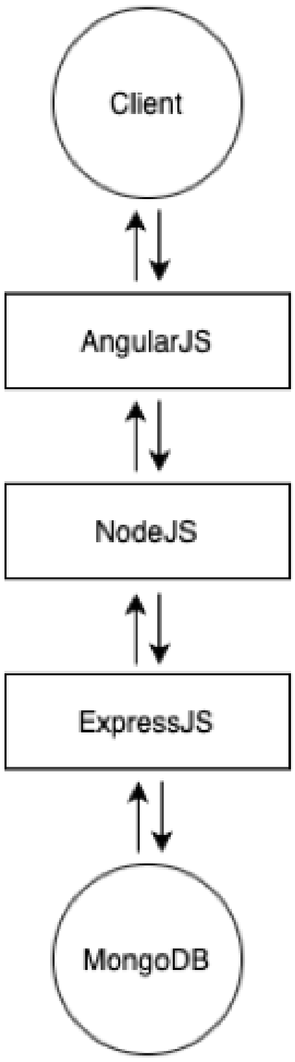
There have been many changes with the supergroup, forcing us to change our implementation. One major change we have had to implement is how migration was handled. Previously, migration would work akin to a repost, where you wouldn't store the migrated data on the journal but instead fetch it from the other journal whenever needed. This was changed so that migration requires all submission files to be transferred to and stored in the journal, so a lot of the focus this semester has been on design and implementing an endpoint to facilitate this. We have also been forced to revisit all our supergroup endpoints, to ensure they still comply with the wiki.

8 Software Architecture: 190004947

8.1 Overview

The highest level elements of our final software architecture is directly derived from our tech stack. The client interacts with our Angular components displayed on the front-end of the web application which then forwards the requests made to Node on the server-side. Node applies some business logic and then uses Express to handle requests to and from the Mongo database. When a response is given from the database, the data returned is propagated back to the client through Express, then Node, then Angular. This is depicted in [Image 3] below.

[Image 3] - Top-down view of technologies



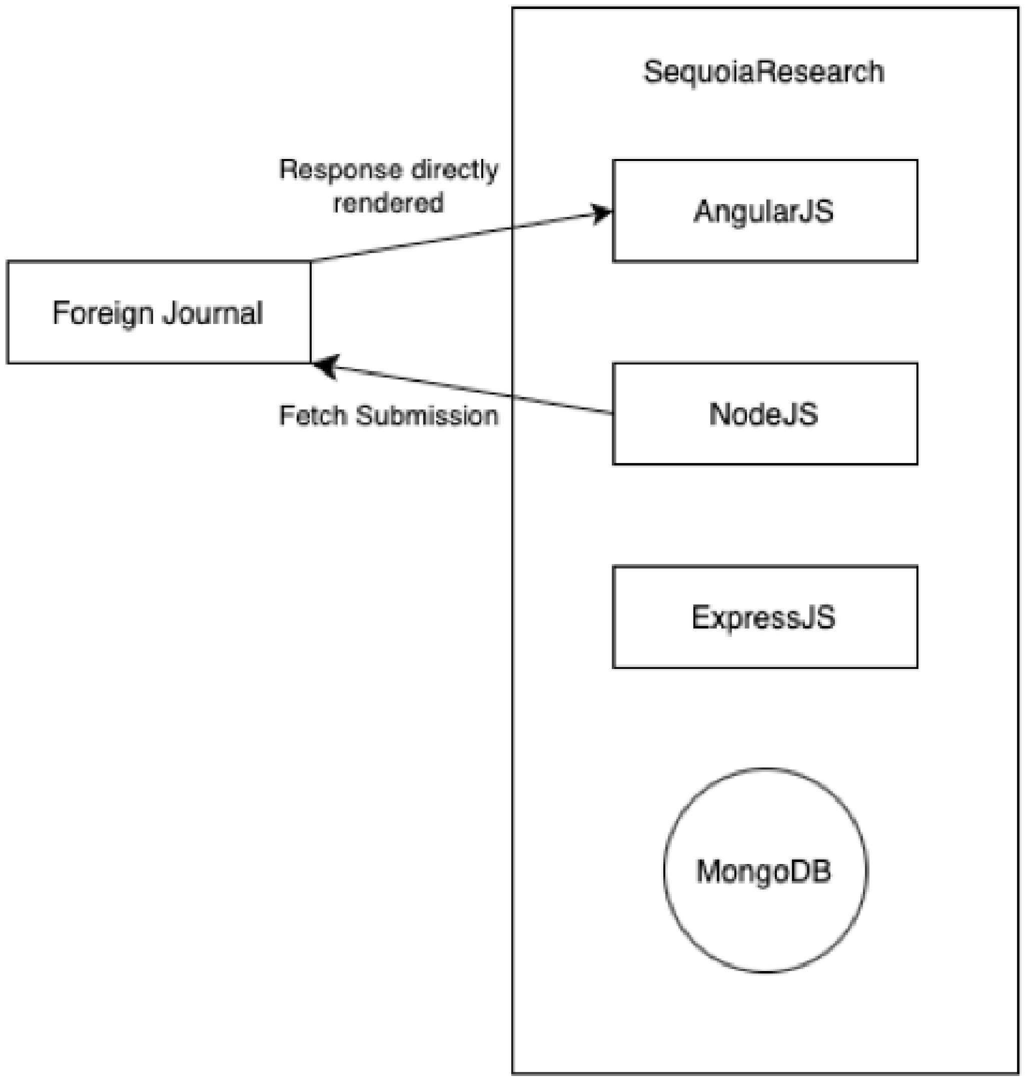
8.2 Agile Architecture

Aligned with our agile development paradigm, we employed the principle of ‘just enough architecture’ which means that our architectural design decisions were made at the beginning of each sprint rather than upfront at the start of development. This allows the architecture to evolve with the codebase in the fast-paced nature of agile implementation. It also meant that we were not restricted by ill-informed design decisions made before development even began. One disadvantage of this protocol is that we have to identify dependent parts of the system to implement first as to avoid building out the majority of a single functionality only to realise we need other parts first.

8.3 Supergroup Interaction

To display submissions from other team's journals, as portrayed in [Image 4], Node makes a fetch request through a supergroup endpoint which returns all submissions from the journal. Then, the contents and comments are taken from the supergroup submission schema and rendered in our front end.

[Image 4] - Protocol for displaying submissions from other journals



8.4 Advantages and Disadvantages

One main advantage of this architecture is that each element is written using JavaScript (or in our case the strongly typed variant TypeScript). This allows for seamless integration between the components as well as avoiding additional learning curves for our development team having to deal with multiple programming languages.

Another advantage of the MEAN stack architecture is that there are thousands of open-source libraries which we can leverage through the Node Package Manager to abstract away complexities such as server-side logic which is handled by the Express library. Each component of the stack also supports JSON for data transfers which has multiple benefits including being able to use Express to manage JSON web tokens for authentication.

An important property of our application is concurrency as multiple users across multiple journals must be able to safely perform actions on shared submissions. We have achieved reliable concurrency through NodeJS. Node uses two types of threads: the single event loop and the pool of worker threads managed by the libuv library. This pool of workers is used to handle computationally expensive tasks such as CRUD operations on the Mongo database. It works by assigning requests from connected clients to available workers which then process the data and return a response. In the case that all the workers are 'busy', the requests are added to a queue where they will remain until picked up by the next available thread. This behaviour means that we could handle every student in the university making concurrent requests without worrying about thread-safety.

One disadvantage of our architecture is that our component implementations are structured to support code re-use for efficient development. However, we have not leveraged this property and instead ended up writing out unique Angular components for each part of the system. Especially in the case of our 'post' component, we could've abstracted away the constituent sections of the submission into their own components which could've been reused elsewhere but instead our final codebase includes an overly complex post component. In future projects, we might spend more time on design decisions at the beginning of each sprint to unanimously decide on an implementation strategy instead of just allowing the assigned group member to make the development decisions alone. This would've promoted the discussed idea of component reuse while also providing more structure to our sprint cycles.

9 Evaluation and Critical Appraisal: 190007288

This section of the report will offer an evaluation of our product and how it compares to our original objectives, the specification, and other similar platforms.

The two main objectives as outlined in the introduction were (i) to create a platform for sharing and peer-reviewing code, and (ii) for the journal to function as a part of a larger federation which allows for accounts and submissions to be shared between other journals in the federation.

The first objective was fully met. Our journal offers the ability for a user to submit code to the journal and for reviewers and editors to review the code. It can then be modified as needed before finally being fully published. We succeeded in creating a journal which combines the abilities of sharing code and of peer-reviewing into one consolidated platform.

The second objective was also well met. After creating an account on any other journal in the federation a user is able to then log in with the same credentials to our own journal. Posting on another journal also allows it to be viewed on our own journal once it has been reviewed and fully published.

Beyond the general overall objectives for the platform, there are more specific basic requirements outlined in the specification.

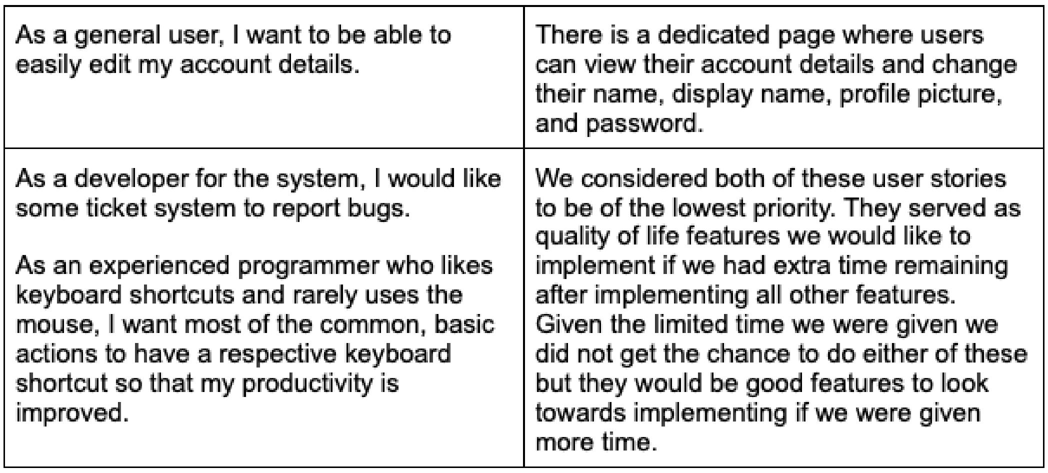
- Ability to register users.
- Posting new source code files (in plain text format).
- Posting comments on source code files.
- A simple web-based interface.
- A description of a protocol for sharing user information and migrating content between code journals.

Each of these core requirements was met by the second deliverable as part of the minimum viable product. Using our website, users are able to register an account and log in. They can then upload a submission with a title, description, and any number of files. Once that submission is put in the review stage it can be reviewed and commented on by other reviewers and editors. Finally, it can be fully published so that anyone can see it, including users of other journals in the federation through the use of the post migration API.

In the beginning of our work on this project we created 33 user stories outlining the features we wanted to implement. These go beyond the basic requirements to create a more fully fleshed out site. The table below outlines the most significant of these user stories and the extent to which each was completed.

User Story	Evaluation
As an author, I want reviewers to be able to comment and give feedback on my code which is in the publication process. As a reviewer, I want to be able to easily comment on specific lines of the submitted code, so that I can accurately convey my feedback to the author. As a reviewer, I want to be able to reply to comments made on the code. As a general viewer of the publication, I want to be able to view, make comments on, and generally discuss published code.	Each of these user stories were fully completed. Reviewers can make general comments on a submission, comment on a specific file, comment on a specific line of a file, or reply to previous comments made. Once a post is fully published any users are able to view and comment on the post.
As a reviewer, I want to be able to edit and delete comments I have made. As an administrator, I want to be able to edit and delete comments posted by other users.	This user story was modified for our implementation. Instead of having the ability to delete or edit comments, they can be resolved which hides them at the bottom of the page. This is more useful when reviewing as the comments can still be seen later if needed.
As a reviewer, I want to be able to privately ask questions to the author of the codebase I am reviewing, so that I can clarify my understanding of their code.	This was not implemented because we later decided that we wanted to provide a space with more open discussion.
As a reviewer, I want to be informed whenever the author or a reviewer responds to a comment I have made, so that I ensure I do not miss any important information. As an editor, I want to be notified when certain events happen in the reviewing process e.g. reviewers think that the code can be published.	This feature was of much lower priority compared to others as it was not a core function of the journal and required significant work to create a notification system. We had time to create a backend for notifications but did not have the chance to implement the front end for it.
As an editor, I want to be able to assign reviewers to the project As an administrator, I want to be given enough permissions to effectively manage the system, so that I can effectively manage the system. As an administrator, I want to be able to ban/unban users easily.	We created a user management page where administrators can manage all users and editors can manage reviewers and general users. This gives the ability to assign roles to users and change their display name and journal number. It also includes a UI for banning and muting users but the functionality for this was not implemented as other features had higher priority.

As an administrator, I want to be able to remove public code posted by other users.	Similar to deleting comments, this was partially implemented. Instead of having the ability to delete posts they can be unpublished which hides them from all users other than the original author and administrators.
As an author, I want adequate security so that only allowed individuals can modify the code, so that I can safely collaborate with other users before trying to publish the code.	Only users who have review, edit, or admin privileges can edit code. There is no ability to edit code directly on the website, but users are able to reupload their code in the review stage and can then see a version history.
As an author, I want to be able to have public and private unpublished code based on my needs.	Code can be put into three different stages – published, review, and unpublished. Users are able to move their code between the review and unpublished stages as necessary, and can unpublish code that has been approved and published if they would like it to be private again.
As a user from another coding journal, I want agreed upon standards for aspects like migration and authentication, so that access to published code from other journals is not difficult. As an author, I want to be able to easily migrate my code to another journal in this federation.	Code migration is very easy. We have a standardised API across our federation which makes sharing code easy. For our own journal we use this API so that submissions published on other journals can be viewed directly on our own journal.
As an author, I want to have an option to upload entire folders as well as files, so that I do not have to individually select every file.	When uploading a new submission, users can either select a single file, multiple files, or a zip file containing any number of files within it. This allows entire projects of code to be submitted all together making for a much better experience.
As a general user, I want to be able to download code that has been posted.	Once a submission is fully published, a download button appears at the bottom of the source page which allows anyone to get a local copy of the code for themselves. We also make use of the federation's standardised API so that submissions from other journals can be downloaded as well directly from our own website.
As a general user of the system, I want my account information to be kept secure, so that I can use the system safely.	User information is stored securely in the database so users do not need to worry about their information being seen by anyone.



As a general user, I want to be able to easily edit my account details.	There is a dedicated page where users can view their account details and change their name, display name, profile picture, and password.
As a developer for the system, I would like some ticket system to report bugs. As an experienced programmer who likes keyboard shortcuts and rarely uses the mouse, I want most of the common, basic actions to have a respective keyboard shortcut so that my productivity is improved.	We considered both of these user stories to be of the lowest priority. They served as quality of life features we would like to implement if we had extra time remaining after implementing all other features. Given the limited time we were given we did not get the chance to do either of these but they would be good features to look towards implementing if we were given more time.

Our platform fills a niche that other similar services are not quite suited for. There are a few popular journals which exist for research papers based on software such as JOSS and JORS. Our journal differs in that it is focused on reviewing the code directly instead of a paper which discusses code. These other sites will often link to a github repository or something similar to refer to the code while our journal merges the two together. Code can be viewed and reviewed all in one on the site.

10 Conclusions: 190012580

10.1 Key Achievements

Having detailed everything about the project and its development process, we will take a step back and look at the bigger picture of what we have achieved and outline potential next steps with a project like ours.

To conclude, we are satisfied with what we were able to implement - making sure all the requirements outlined in the specification were implemented to a high standard was paramount. We were able to do this to a standard we were content with, and test it reliably during development to all but conclude that each mandatory feature was robust. This was the most important goal of our project and we achieved it.

10.2 What we have learnt

This project taught all of us a lot of key, practical skills that we perceive to be the most useful out of anything we have learnt in other modules. We do not know what the future holds for us but becoming familiar with version control, web development and working remotely in a team are all skills we believe will be extremely valuable in the future.

Although we were happy with our development process as a whole, one aspect we could have done differently was make sure that sprint tasks were as loosely coupled and cohesive as possible - when there were many tasks dependent on one individual task, it caused a hold-up in development which is something we would want to avoid at all costs. Sometimes it is inevitable that one task will be dependent on another but it was something we should have tried to keep to a minimum.

10.3 Drawbacks and desired functionality

Email verification was a feature we were keen to add, but the restrictions on the host servers prevented us from being able to send emails. Alternatives were to use a third party client such as SendGrid[4] or generating a link which could be written to a text file, but we did not want to have a feature that was not did not have the full functionality we intended it to have, so other features were prioritised.

We implemented a backend to support notifications which would have improved our website but due to time constraints we were not able to implement a UI to make it a fully functional feature. This would obviously be leading the list of future plans.

10.4 Future plans

There are two sources for ideas on features we could add to Sequoia Research in the future:

- User stories.
- Features there were not an initial user story we attempted or considered implement but did not fully implement.

User stories are a good source as they were created at the very start of the development process, purely from an end-user's point of view. As one works on a project, bias may arise as they account for their own programming skill when it comes to deciding what features to implement, which may subconsciously sway them away from the core desired features outlined at the start.

A project like ours is and will never be fully 'complete'. There is always something which could be implemented to improve it so in a sense, there will always be something we could have done that we chose not to. We were aware this would be the case so we had to make sure that ordered the tasks in terms of priority and that features we chose to implement have the biggest positive effect on our site.

11 Acknowledgements

As a group, we would like to personally thank Olexandr Konovalov and Areti Manataki, our supervisors for Semester One and Two respectively, for their ongoing support and guidance throughout the entirety of a turbulent academic year.

12 Appendix

[1] - Sprint 1 Weekly Report: See 'Sprint 1 Weekly Report.pdf' in the 'Appendix' directory.

[2] - Sprint 2 Weekly Report: See 'Sprint 2 Weekly Report.pdf' in the 'Appendix' directory.

[3] - Sprint 2 Summary Report: See 'Sprint 2 Weekly Report (After 2nd deliverable).pdf' in the 'Appendix' directory.

[4] - Sprint 3 Report: See 'Sprint 3 Weekly Report.pdf' in the 'Appendix' directory.

[5] - Product Backlog: See 'Product Backlog.pdf' in the 'Appendix' directory.

[6] - Sprint Backlog: See 'CS3099 - Sprint Backlog - Sheet1.pdf' in the 'Appendix' directory.

[7] - Sprint 4 Report: See 'Sprint 4 Weekly Report.pdf' in the 'Appendix' directory.

[8] - Sprint 5 Report: See 'Sprint 5 Weekly Report.pdf' in the 'Appendix' directory.

[9] - Sprint 6 Report: See 'Sprint 6 Weekly Report.pdf' in the 'Appendix' directory.

[10] - Meeting Minutes: See 'Meeting Minutes.pdf' in this directory.

[11] - Supergroup Wiki: See 'Supergroup.md' in this directory.

13 Testing Summary: 150000946

During the development of this Journal, we initially wanted to implement CD, and an automated Python testing suite using the Selenium tool. As we continued development, we found that writing unit tests for our endpoints was difficult to maintain, as our project both in the backend and frontend was continually evolving (as expected with an Agile development method). We did make some progress with the Python suite, however it was quickly made obsolete.

Thankfully, our Scrum methodology involving feature branches was invaluable. As each merge request required the approval from the majority of our team, at every review at least three people looked at and tested the new functionality of the merge request, as well as a more brief check that that particular request didn't break other functionality with manual testing.

This approach suited us all much more - for the most part bugs were caught before they were pushed to the main branch, and when bugs did get through the net, they were quickly identified and fixed. We feel that this approach was much more suited to the Scrum methodology we followed.

Given the above, we do intend to complete the Python testing Suite now that our product is more finalised. Unfortunately we were not able to do this before the deadline.

14 Bibliography

[1] Pros and Cons of TypeScript. Altexsoft. 2020. Retrieved from:

https://www.altexsoft.com/blog/typescript-pros-and-cons/

[2] OWASP Cheat Sheet Series. Jim Manico, Jakub Maćkowski. 2022. Retrieved from:

https://owasp.org/www-project-cheat-sheets/

[3] Password administration for system owners. 2022. Government Digital Service. Retrieved from:

https://www.ncsc.gov.uk/collection/passwords/updating-your-approach

[4] Email Marketing Automation. Sendgrid 2022. Retrieved from:

https://sendgrid.com/solutions/email-marketing/automation/