University of Glasgow | School of Computing Science

Level 3 Project Case Study Dissertation

SH01

Maxine Armstrong
Ibrahim Asghar
Finlay Cameron
Rachel Horton
Colin Salvatore Nardo
QiKai Zhou

26 September 2024

Abstract

This case study analyses and reflects on the decisions made during the development of a web application to replace the Learning Innovation Support Unit's (LISU's) Course Content Mapping (CCM) system, from initial design to deployment and handover.

We present the necessary background information in our Case Study Background to provide context for five main themes of software development discussed in sections three through seven. In each theme, we highlight our learnings and make recommendations based on our reflection and the literature. We focus on the following themes: the Development Process and conflicting best practices, our Software Design and technical debt, Deployment and planning for external delays, the dangers of Last Minute Development, and Accessibility and addressing blind spots in review.

Our Conclusions summarise the key takeaways from each theme section, offering a set of recommendations for future team projects. These include proactive planning and avoiding taking on technical debt early in a project, the importance of regular feedback and user testing to discover blind spots, and how the costs of changes increase as a deadline approaches.

Education Use Consent

We hereby give our permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format.

1 Introduction

We present a case study of the development of the University of Glasgow's Learning Innovation Support Unit's (LISU's) Course Content Mapping (CCM) web application, from initial design to deployment and handover. We present the necessary background information to provide context for five main themes of software development discussed in sections three through seven. In each such section we investigate the core theme, discussing the steps we took and reflecting on mistakes made and better practices we could have implemented to improve the software process. In the conclusion, we present final remarks and key takeaways for the reader, summarising our learnings from each theme.

Our client, LISU, sought to develop a collaborative and scalable tool that could be easily adopted by academics across all schools within the University of Glasgow. To achieve this, we set out to create a familiar yet enhanced platform that would retain the core functionalities academics relied upon while introducing a more intuitive and streamlined user experience. The new CCM application was created to improve course design efficiency by enabling educators to quickly create, modify and share teaching workbooks, reducing administrative overhead.

Throughout the six-month development process, we held client meetings and retrospectives in the Agile methodology. Our conclusions presented throughout this case study were gathered from our retrospectives, allowing us to analyse the longitudinal effects of policies we implemented, and contrast quantitative and qualitative evidence from before, during, and after specific changes to our development process were made.

This dissertation discusses five themes in depth. Section three, Development Process, analyses how our continuous integration pipeline boosted productivity when implemented, and the unexpected detriments of long queue times. It also covers the positive effects of ticket templates, and complementary practices which can further improve their effectiveness. Technical debt is a common thread in section four, Software Design, in which we explore the evolution of our design, relating our experiences to scientific and industry views on best software development practices. Section five, Deployment, discuss the effects of deadlines on developer productivity, and steps to mitigate the problems we faced in the context of deployment. Section six, Last Minute Development, reflects on incidents related to commit frequency before deadlines, and outline strategies to mitigate the risk of last-minute changes breaking the product. Finally, section seven, Accessibility, reflects on our attempts to implement accessibility features, and tools from human centered design that we could have employed to improve our final product in this regard.

2 Case Study Background

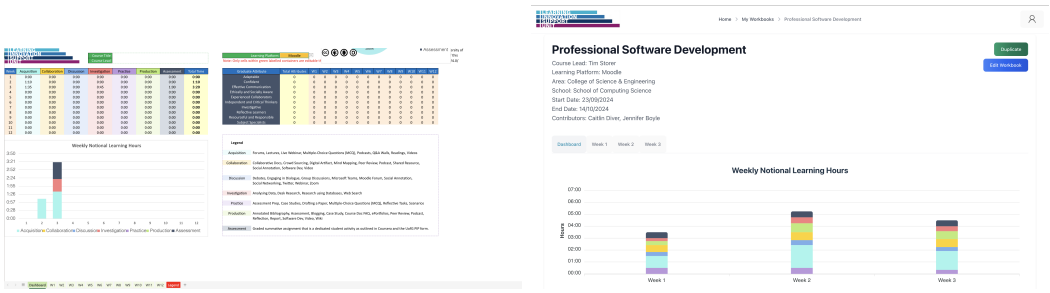
Our client, the Learning Innovations Support Unit (LISU), plays a key role in enhancing teaching and learning at the University of Glasgow by helping staff integrate

technology into their practices. They manage the development of the university's Massive Open Online Courses (MOOCs), microcredentials, expert tracks, and specialisations, ensuring high-quality online education. Our main point of contact was Learning Innovation Officer and LISU representative, Richard Johnston, who attended all team meetings and provided consistent feedback. His involvement helped ensure our design aligned with LISU's strategic priorities and addressed the practical needs of university staff.

The project goal was to replace LISU's existing Course Content Mapping (CCM) spreadsheet system. The CCM is used by academic teaching staff as an aid in course development. The information about a course is stored in a workbook in the form of an Excel spreadsheet. A workbook is structured into weeks, each of which contains a list of activities and associated information. The information associated with each activity includes the time required and the related learning outcome. Sharing workbooks is easy, as the whole system is built on top of Excel which has tools for online collaboration, is simple to download and thus share in emails or other electronic communication, and can be edited in the user's preferred environment as they deal with all the other Excel spreadsheets they use.

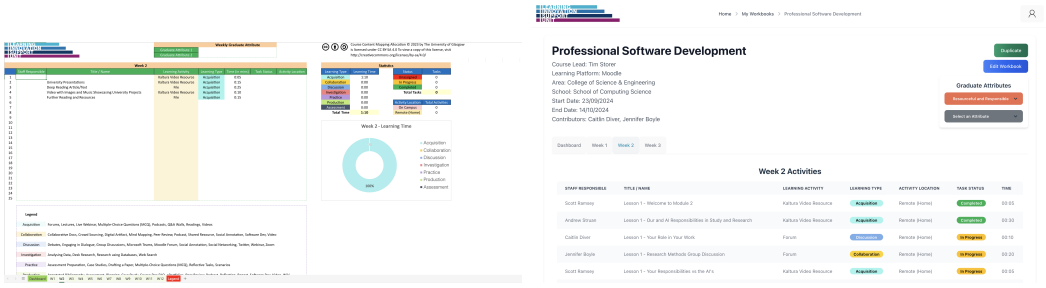
Our replacement was a dynamic, scalable web application. Our system allows academic staff to plan, modify, and share course content in a collaborative digital space. Key requirements included role-based access control (RBAC), integration with the University of Glasgow's Single Sign-On (SSO) system, search and filter functionality, and all core components of the Excel-based system. As a university application, the product stylistically integrates with the UofG brand and software suite. The needs of two main user groups shaped our design: course coordinators needed detailed curriculum planning tools that aligned with existing university platforms, and subject lecturers needed an intuitive interface that would be accessible and shareable regardless of their technical proficiency. Ensuring both types of users could engage with the system effectively was a central design priority.

Ultimately, the final product was a fully functional CCM application that met LISU's core goals. It replaced the outdated spreadsheet workflow with a modern, user-friendly web platform, deployed on the university's network and fully integrated with Azure AD SSO like all other university services.



(a) Old (b) New

Figure 1: Workbook Dashboard



(a) Old (b) New

Figure 2: Workbook Week Page

3 Development Process

Throughout the course of this project, we found that improvements to our continuous integration (CI) pipeline in linting and formatting and ticketing procedures enhanced code quality, reduced communication overhead, and accelerated development. We also discovered that the best practices are not always universally compatible, and that an effective development process requires ongoing critical evaluation.

Our early success with MyPy, a static typechecker for Python, prompted us to reflect on our priorities and implement TypeScript type checking, cashing in accrued technical debt. Due to a team member's previous experience in best Python practices, we implemented MyPy typechecking in the CI pipeline on all Python code in our fifth merge request. We frequently ran into pipeline failures at the linting stage due to type errors that would have caused runtime exceptions. This is a common experience: empirical research suggests that 15% of all defects in Python codebases are type errors that would have been caught by MyPy [13]. Typechecking the TypeScript codebase was only implemented in merge request 87, due to fears that it would unnecessarily slow development during a stage of the design cycle in which rapid iteration was critical. When we finally implemented frontend typechecking due to the benefits we saw in the backend, over 1500 lines of TypeScript had to be changed to pass linting, which held up the entire frontend development process. This led us to reflect on the effect of technical "debt", an analogy presented by Cunningham in 1992 [4, p.2], and how the small gains in productivity we made earlier in the project came at a far greater cost later on. In retrospect, we should have implemented strict typechecking from the beginning, because we ended up spending far more time in the eventual refactoring than we saved deferring it.

While static typechecking improved reliability, we also found that implementing opinionated formatting enhanced code clarity and streamlined collaboration. Similarly to typechecking, we quickly implemented the opinionated formatter Black in our Python code. However, the frontend did not implement any formatting until March, leading to inconsistent style throughout the frontend code, for example in line lengths and newline placement. We found that implementing opinionated formatting on the frontend allowed us to "concentrate on what [we were] saying rather than on how [we were] saying it" [8, Parting Words, para. 3], making bugs in logic easier to spot in

review. Because code stylers are automatic, we did not have to spend any manual effort refactoring code to pass formatting tests. The main cost for us was the loss of productivity from not having implemented it earlier, and in retrospect this should have been set up before any code was committed to grease the gears of development.

Despite these benefits, our CI pipeline also served a major challenge which required us to reevaluate best development practices. Due to the number of jobs being run on Wednesdays, when every team was working on the project in-person, pipelines were often queued for hours before running. As our merge requests (MRs) were blocked on pipeline success, we would put more and more features in them instead of making more to further clog the queue, putting the generally accepted good practice of CI [25] in conflict with that of small MRs [9]. We experienced an increase in frequency and difficulty of merge conflicts, code review overhead, and time spent chasing dead ends due to the increase in MR size caused by our slow pipeline. In light of this conflict, and realising that our current CI setup was unsustainable, we registered our personal computers as pipeline runners for our project, all but eliminating queue times: we went from merging one to three MRs every Wednesday to twelve in our final week when we registered our runners. We quickly adapted to this change by submitting smaller MRs, further increasing efficiency of code review and decreasing merge conflicts. This taught us to critically consider the different continuous integration practices we were implementing, and how they could interfere both constructively and destructively. In the future we would reflect on the conflict between these two goals and register our own runners as soon as possible to resolve it.

Throughout the development process, we made improvements to our ticketing procedures that improved the efficiency of our development process. We initially did not use Merge Requests (MRs) or issue templates, which made collaboration across issues and peer review of MRs difficult. We were unable to extract all the necessary information from these tickets, and extracting any information took longer due to a lack of a common grammar provided by templates. When the Markdown issue and MR templates introduced in MRs 11 and 12 respectively were properly used, we found a reduction in time spent on reviewing and acting on tickets as well as a reduction in the amount of asynchronous communication required to bring MRs to an acceptable state. This seems to be a common experience; research suggests that the mean number of comments on issues (asynchronous communication) decrease by approximately 13%, and the time to respond (TTR) to issues decreases by approximately 80% when Markdown issue templates are added to open source projects on Github [21]. We expected that creating the templates and applying them automatically would fix the issue of poorly formatted issues and MRs, but they were not consistently used by all developers, which greatly reduced their effectiveness. This was a point of friction, and was commented on in the December sprint meeting, and improved slowly until we were consistently using the templates in the final sprint. In retrospect we should have been consistently providing feedback on the quality of other developer's tickets in order to make the template changes stick quicker, so that we could have reaped the full benefits sooner.

In summary, our CI pipeline and tickets had a positive effect on development speed and quality throughout the project, enabling us to meet the minimal viable product

for our client. We encountered an unexpected conflict between enforcing CI and following the best MR practice, leading us to reconsider our implementation and find a solution that fit our specific constraints. We also were not aggressive enough in enforcing new changes, which allowed poor practice to propagate long into the project, which is a key point for improvement.

4 Software Design

This section delves into the evolution of our application's software architecture, focusing specifically on the critical shift from a monolithic structure to a more modular, component-based frontend design. While our initial approach allowed for rapid prototyping, it soon presented significant challenges. The subsequent refactoring effort, though demanding, provided learning experiences regarding maintainability, scalability, and the practical application of core software design principles, a central thread that underlies all sections that follow.

4.1 Initial Frontend Challenges

In the early stages of development, aiming for quick delivery of visible features, our frontend codebase organically grew into a monolithic structure with limited modularity. Prime examples were the `CreateWorkbook` and `EditWorkbook` pages. Although they shared substantial state logic, UI elements, and functionality, none of it was abstracted. Instead, code was duplicated across both pages, violating the DRY (Don't Repeat Yourself) principle [24] and inflating the codebase. The practical consequence of violating DRY is that duplications inevitably become inconsistent or outdated in different places, harming maintainability and reliability, precisely the issues our eventual refactoring needed to address.

This approach stemmed partly from our team's initial limited familiarity with React's component-based paradigm and an underestimation of the application's eventual complexity. The direct consequence was the emergence of a "God component" anti-pattern [1, p.1] within these pages, single, large components attempting to manage numerous, often unrelated, concerns. This significantly hampered development agility, even minor changes, like adding a single form field, required developers to hunt down and modify code in multiple locations, increasing the risk of introducing inconsistencies and bugs. This experience highlighted the practical cost of technical debt incurred by prioritising initial speed over structural planning, resulting in what Fowler describes as "shotgun surgery" where one small change necessitates many scattered edits [18, Payment Feature section, para. 4].

4.2 Transition to Modular Architecture

The difficulties caused by code duplication and the God components became a primary motivator for a significant refactoring initiative, internally dubbed the *Great Refactor*. The main goal was to improve the codebase's health, which can be achieved by adhering to best practices like separation of concerns and enhancing component reusability, according to [20].

However, the refactoring process itself presented considerable challenges and valuable learning opportunities. It proved far more time-consuming than initially anticipated. Rather than designing isolated components from the start, we invested significant effort in identifying, carefully extracting, and transforming duplicated logic and UI segments into genuinely reusable components [12]. This task was often complex, requiring us to disentangle logic that had become tightly coupled within the monolithic structures. We had to learn, through trial and error, how to design components that were generic enough for reuse yet customisable via props. For instance, creating a reusable warning pop-up required mastering prop-driven configuration and conditional rendering techniques within React to cater to different contexts. This demanding process deepened our practical understanding of modular design and the trade-offs between component generality and specific needs, teaching us the importance of anticipating reuse patterns earlier in the design phase. We also learnt the value of explicitly discussing and planning for refactoring time, acknowledging it not just as cleanup, but as a necessary investment for long-term project health, even amidst deadline pressures.

4.3 Enhanced Maintainability and Efficiency

Following the *Great Refactor*, the advantages of the modular architecture became immediately apparent, particularly when implementing new features. The development process felt significantly streamlined. Two examples illustrate this improvement: the `ActivityModal` and the `GraduateAttributesChart`.

Previously, activity creation and editing were handled within the separate CreateWorkbook and EditWorkbook pages, leading to the aforementioned duplication and complexity. Post-refactor, these workflows were consolidated into a single, reusable `ActivityModal` component. This component now encapsulates all related form logic, state management, validation, and submission handling. Integrating it simply requires passing appropriate props. Consequently, adding a new feature, such as activity categorisation tags, involved modifying only one component, reducing effort, ensuring consistency, and minimising the potential for bugs arising from divergent logic paths. This experience taught us the power of identifying shared workflows early and encapsulating them effectively.

The modular design profoundly improved overall code quality and maintainability. We eliminated bugs that resulted from inconsistent fixes across duplicated code sections. Developers could now pinpoint and address issues within smaller and well-

defined components, each with a clear responsibility. Furthermore, it facilitated more effective unit testing. We learnt that testing smaller, focused components is significantly easier and leads to more reliable tests, as isolating functionality like input validation or chart data transformation logic became straightforward.

The improved structure positively affected the workflow of the team. We observed a noticeable increase in feature development speed, which we attribute to clearer component boundaries, fewer merge conflicts as developers worked on more isolated parts of the codebase, and a simplified debugging process, benefits commonly associated with modular software architectures [2]. Although the refactoring required a significant upfront time investment, the long-term gains in maintainability, adaptability, and developer productivity clearly validated the decision and underscored the lesson that investing in good architecture early, or correcting it decisively, pays dividends.

Based on these learnings, our approach to future projects has evolved. We would prioritise a more upfront architectural discussion, dedicating specific time early on to explore potential patterns, identify core reusable entities, and sketch component interactions. We would also strive to build in good structure even during initial prototyping phases, rather than relying on large, potentially disruptive refactoring efforts later. We would establish a standard practice of conducting regular code and architecture reviews. Additionally, we would more actively consider established frontend design patterns earlier in the process to provide a robust structural foundation. Ultimately, our experience underscored the critical importance of thoughtful software design, not just as an academic principle but also in its applications efficiently within a team setting.

5 Deployment

In order to successfully complete this project and provide LISU with a product that they could introduce to lecturers and teaching staff, we had to ensure that the site was accessible. Additionally, one of the critical elements of the MVP for LISU was the implementation of the University of Glasgow Single Sign-On (SSO) onto the product using Azure AD. To achieve both of these goals, the site needed to be hosted publicly. The process of achieving these goals taught us about the benefits of prototyping and introduced us to the potential pitfalls of working with other teams during a project.

5.1 Prototyping

In our first term, we focused purely on developing our product and therefore only had a local version of our site. We then used this local version as a prototype to show to our customer at monthly sprint meetings to help with design validation and risk mitigation [17]. This allowed us to gain monthly feedback on the aesthetics and UI/UX in a controlled environment, but we knew that we wanted more consistent reviews of our product as regular prototype testing is beneficial to continuous development. It has been shown to both prevent design fixation and improves team morale as “failure

is reframed as an opportunity for learning" [3, p.10], so instead of feeling discouraged after monthly sprint meetings, we would already be aware of most feedback and be able to continuously adjust our product to fix any issues or meet modified criteria.

Our solution was to host our site to allow members of LISU to give us continuous feedback outside of our monthly sprint meetings, as regular customer feedback encourages a more user-centred product [14]. We prioritised hosting and had our site hosted by the January Sprint Meeting using Netlify for the frontend and Render for the backend. This allowed our customer to come to the sprint meetings more prepared with their own discussion points, and better prepared to answer our questions about different aspects of the site. Receiving better answers allowed us to better understand and meet our customer's requirements or avoid agreeing to work that would not be feasible within our time frame. In these ways, the use of a prototype that was hosted and could be accessed at any time by our customer, led to benefits both in and outside of our sprint meetings. If we were to do this project again, we would ensure that we had a live prototype that could be used by our customer earlier, to maximise these benefits.

Since this was only a prototype, we knew that this would not be the final hosting structure, and whilst this was being setup we were simultaneously discussing a more permanent way of hosting - particularly with handover in mind.

5.2 Hosting

Since our customer was a department within the University of Glasgow, they were interested in using the university's infrastructure to host, particularly because of their focus on implementing SSO. We agreed to this since by hosting the site using the university's Azure platform, it would be easier to extend that to incorporate Azure AD - the system through which we would attach the university's SSO.

As soon as we had agreed this with the customer and were ready with a deployable prototype, we requested an Azure virtual machine, which would be controlled by the Windows & Azure Team Manager, our liaison within the university's IT department. Although this was agreed upon towards the end of January, we could not immediately get to work - we had to wait for the University of Glasgow Finance Department's cost centre to approve a request funding the virtual machine.

Unfortunately, we had to wait much longer than anticipated for this and our expected timelines were delayed by a month until we were given access to the virtual machine. We reflected that when other departments or teams are involved in a development process, their timelines and potential delays also have to be taken into account within our own plans. Even when a delay is out of our control, we still have to be careful of the potential negative psychological impacts delays can have to team morale [10]. With these learnings in mind, if we were to repeat the project we would prioritise early communication with teams we were dependent on. Instead of waiting until we had a deployable app to start the deployment process, we would start the deployment process in parallel.

6 Last Minute Development

Throughout the project, we found that the quantity of code written consistently spiked just before a sprint meeting. This was normally in order to either complete aesthetic fixes or sprint goals set out in the previous meeting. Members of the team would often stay up late the night before in order to get these tasks done. At the time, staying up late felt worthwhile in order to make our product looking more finished, but working while sleep deprived has been shown to decrease the quality of code written [7]. This meant that our energy the next day was not the only factor impacted by late night coding - sometimes the code written would need to be undone or refactored later in the day. In retrospect, we recognise that we should have adjusted our practices to avoid late-night work. However, we were consistently able to make the desired fixes before our sprint meetings so were given positive reinforcement for this development practice. Hence, we continued making last-minute changes before deadlines.

Then, on the day before our presentation, we found that adding contributors to a workbook could not be done during workbook creation, rather only when editing once a workbook which was already in the database. Upon review, we discovered the root cause was a lack of modularisation which led to two potential solutions. One solution was to copy code into the file it was missing from, but this would break the Don't Repeat Yourself (DRY) software development principle [24]. Another fix was to repeat the practices described in the *Great Refactor* to separate the functionality out of the child component and ensure component reusability and improve maintainability [20]. We chose to refactor the files, fixed the issue on its feature branch, and subsequently merged into main.

However, when we attempted to move this code across to the production branch and update the deployed site, it failed to update. In our attempts to force an update, we modified the nginx and docker compose configuration files but due to our lack of expertise with these tools, it caused more issues on the public site. Having left this change so late, we then panicked to fix the issue in time. Since no one in the team was particularly experienced with nginx or docker, the bug became more complex to fix. It was solved only twenty minutes before we were scheduled to give our presentation. This meant that we did not have time to do a practice run-through of our demonstration and so were less prepared for this important, graded aspect of our project.

As one approaches a deadline, the cost of additions to the product increases since "compensating for the lost time becomes more difficult" [5, p.1]. On reflection, we should have discussed more thoroughly whether this feature was worth attempting so close to the presentation - whether the potential benefit justified the risk or if postponing would have been safer.

7 Accessibility

In this section we focus on the accessibility of the product; we will discuss the methods we used to make the program more legible and intuitive for our target demographic and the subsequent improvements we would implement based on our experiences with the project.

7.1 Design

Colour palettes were key to the product's accessibility, developed based on client requirements to differentiate learning types. Our primary meeting with the customer highlighted colour's importance, leading us to expand this approach throughout the UI with distinct palettes for workbook areas, status indicators, learning types, and graduate attributes. To ensure accessibility, the project utilises a clear font and dynamically computes text colour to enforce contrast ratios, as a 2009 study suggests that font size and contrast adjustments can increase readability by 47% for older users [6]. All text elements and interactive components are adjusted to meet or exceed the minimum WCAG-recommended ratio of 4.5:1, to account for the older users, 60% of whom were reported to have difficulty using digital devices [6]. This systematic approach not only meets accessibility standards, but also reinforces the project's visual identity consistently throughout the UI.

The project's layout is designed for both visual appeal and accessibility, using a responsive design framework. The external framework, `Tailwind CSS`, ensures the project maintains consistent margins, padding, and overall white space, which is crucial for readability and user comprehension. Research shows that properly structured white space can improve reading comprehension by 20% and reduce task completion time by 14% among older users [23]. For instance, the `WeekActivityTab` component employs standardised spacing classes that effectively segregate interactive elements and content blocks, reducing visual clutter—an improvement that addresses the frustration reported by up to 72% of users aged 55 and over when exposed to poorly designed interfaces [23], which we and the client felt was an issue with their original spreadsheet design. Additionally, the layout includes responsive table implementations with overflow handling, ensuring that content remains legible on various screen sizes. Embedding these layout principles directly into our CSS and component structure ensures a design that is both adaptable and accessible.

Data visualisation is a core functionality of the program, which was discussed from the beginning with the client, and within it we utilise a dual presentation mode; the interactive graph ensures high-contrast, legible tooltips and responsive breakpoints for seamless adaptation to different devices. In addition to the graphical view, a table view is provided which enables users to effortlessly switch between a visual chart and a textual data table, thereby catering to diverse user preferences. Notably, 68% of older adults prefer text-based data representations over solely graphical formats, and offering a dual presentation mode has been shown to increase usability by approximately 32% [22].

7.2 Improvements

The development of accessibility features is an ongoing process; although we encountered certain issues during development, the product handover highlighted key accessibility challenges that provided critical learning opportunities for future improvements.

One element which was discussed in the meeting is that the site uses a lot of visual elements, which themselves utilise colour to help users navigate the site easily. However, people with visual impairments, including colour blindness and partial or complete blindness, may struggle. Increasing the contrast between colours and allowing users to toggle between different colour schemes within their account settings would thus improve accessibility. Recent studies indicate that customisable colour schemes significantly improve legibility for individuals with colour vision deficiencies [16, 11].

In addition, enhanced support for screen reading tools can provide a better auditory experience for visually impaired users. Research indicates that implementing advanced screen reader compatibility and streamlined navigation can significantly reduce cognitive load and improve usability studies have shown that 75% of screen reader users rely on keyboard shortcuts instead of a mouse, and proper `ARIA` labelling can enhance usability by up to 48% for blind users [15, 19].

Therefore, in the future, we would implement these features alongside continuous feedback and testing from users with the appropriate accessibility needs, adopting a more inclusive, user-centred design approach that has been shown to significantly improve overall usability by involving a wider spectrum of users from our target demographic with diverse accessibility needs. In fact, research indicates that iterative usability testing and user-centred design practices can reduce accessibility issues by up to 35% with each development cycle, underlining the importance of continuous improvement in accessible design moving forward with the product [15, 19].

8 Conclusions

The development of the LISU CCM project presented several challenges and valuable learning opportunities, especially for our team, as it was one of our first experiences working on a project of this scale. Throughout the project, we gained insights into the complexities of working with external stakeholders, managing a live prototype, and ensuring the accessibility and usability of our product.

Section 3 addresses the development process, which began with a focus on improving the quality of our codebase and streamlining communication. While we found continuous integration (CI) tools boosted code quality, they came with some unexpected challenges. Typechecking and linting not only provide immediate benefits, but their early introduction can prevent technical debt from accruing. This highlighted

the importance of addressing technical debt early and considering the long-term implications of seemingly small decisions. Despite these challenges, the CI pipeline ultimately helped us deliver a more reliable and maintainable codebase.

In section 4, we explored the significant evolution of the software architecture of our application, particularly the transition from a monolithic structure to a modular, component-based design: The *great refactor*. By analysing the costs of creating new features and fixing bugs before and after the *great refactor*, we uncovered just how destructive technical debt can become. This experience reinforced the need for proactive planning, even in the early stages of development, to create a robust and adaptable codebase.

In section 5, we discuss the importance of regular feedback from the customer. Initially, our project was hosted locally, which allowed us to gather feedback during sprint meetings. As we progressed, we recognised the benefits of having a live prototype early on. It allowed us to receive continuous feedback outside of scheduled meetings. This iterative approach allowed us to address design issues promptly and made our customer feel more engaged in the development process. In a similar vein of communication, we discuss the stress caused by the time pressure of implementing SSO. Reflecting on this, we learnt the importance of early communication with external teams and adjusting our internal timelines to account for these dependencies.

In section 6, we discuss the impact of last-minute development on our team's workflow. We often worked in short bursts to meet deadlines, which led to sleep deprivation and compromised code quality, which required additional time to address issues later. One critical last-minute issue forced us to perform a refactor which broke our product at the eleventh hour. In hindsight, we recognise that this last-minute rush could have been avoided through better planning and an earlier focus on testing and deployment, and implementing a hard code-freeze shortly before critical deadlines could prevent disaster.

Section 7 highlights the importance of accessibility to our product. From the outset, we made design decisions that aimed to ensure the product was legible and intuitive for all our users. Although we implemented many accessibility features, we found the final product lacking. Our key reflection is that for domain-specific knowledge such as accessibility, getting help from experts and direct feedback from users is a necessity: a team of developers will be unable to account for every human experience in their own testing and ideation.

Overall, the LISU CCM application has been a significant learning experience. We faced technical challenges, delays, and design issues, but these were balanced by our ability to communicate as a team and adapt based on feedback. Our customer has expressed satisfaction with the final product, which meets the core requirements set out at the beginning. Moving forward, the lessons learnt in customer collaboration, prototyping, deployment, and accessibility will be invaluable as we tackle future projects.

References

[1] Pritom Saha Akash and Kevin Chen-Chuan Chang. Exploring variational graph auto-encoders for extract class refactoring recommendation, 2023.

[2] Number Analytics. 5 surprising modularity benefits in tech product design. *Number Analytics Blog*, 2024.

[3] E.J. Christie, Daniel Jensen, R.T. Buckley, D.A. Menefee, Kyle Ziegler, and Kristin Wood. Prototyping strategies: Literature review and identification of critical variables. *ASEE Annual Conference and Exposition, Conference Proceedings*, 06 2012.

[4] Ward Cunningham. The WyCash portfolio management system. OOPSLA '92 Experience Report, 1992. http://c2.com/doc/oopsla92.html.

[5] Aviv Emanuel, Maayan Katzir, and Nira Liberman. Why do people increase effort near a deadline? an opportunity-cost model of goal gradients. *J. Exp. Psychol. Gen.*, 151(11):2910–2926, November 2022.

[6] Arthur D. Fisk, Wendy A. Rogers, Neil Charness, Sara J. Czaja, and Joseph Sharit. *Designing for Older Adults: Principles and Creative Human Factors Approaches*. CRC Press, 2009.

[7] Davide Fucci, Giuseppe Scanniello, Simone Romano, and Natalia Juristo. Need for sleep: The impact of a night of sleep deprivation on novice developers' performance. *IEEE Transactions on Software Engineering*, 46(1):1–19, 2020.

[8] Google. Google python style guide, 2024. Accessed: 2025-03-27.

[9] Google. eng-practices: Small CLs, 2025. Accessed: 2025-03-27.

[10] Hannes Guenter, Ij.H. Emmerik, and Bert Schreurs. The negative effects of delays in information exchange: Looking at workplace relationships from an affective events perspective. *Human Resource Management Review*, 24:283–298, 12 2014.

[11] Brett Jordan. Accessible color choices: Best practices for colorblind-friendly design. *Universal Access in the Information Society*, 4(2):92–102, 2002.

[12] S. H. Kannangara and W. M. J. I. Wijayanayake. An empirical evaluation of impact of refactoring on internal and external measures of code quality. *CoRR*, abs/1502.03526, 2015.

[13] Faizan Khan, Boqi Chen, Daniel Varro, and Shane McIntosh. An empirical study of type-related defects in python projects. *IEEE Transactions on Software Engineering*, 48(8):3145–3158, 2022.

[14] Vineela Komandla. Enhancing product development through continuous feedback integration. *ESP Journal of Engineering & Technology Advancements*, 2:105–115, 12 2022.

[15] Jonathan Lazar, Jinjuan Feng, and Harry Hochheiser. *Research Methods in Human-Computer Interaction (2nd Edition)*. Morgan Kaufmann, 2017.

[16] Masataka Okabe and Kei Ito. Color universal design (cud): How to make figures and presentations that are friendly to colorblind people. *Color Vision Research*, 2002.

[17] Minjie Pan. Prototyping methods: Techniques and its significance. *Journal of Research and Development*, 11, 2023.

[18] J. Qiu. Modularizing react applications with established ui patterns, 2023. Section 3, para. 4. Accessed: 27 Mar. 2025.

[19] Kristen Shinohara and Jacob O. Wobbrock. In the shadow of misperception: Assistive technology use and social interactions. *CHI Conference on Human Factors in Computing Systems*, pages 705–714, 2011.

[20] IEEE Computer Society. Why software design is important, n.d. Accessed: 27 March 2025.

[21] Emre Sülün, Metehan Saçakçı, and Eray Tüzün. An empirical analysis of issue templates usage in large-scale projects on github. *ACM Trans. Softw. Eng. Methodol.*, 33(5), June 2024.

[22] Eleftheria Vaportzis, Maria Giatsi Clausen, and Alan J. Gow. Older adults perceptions of technology and barriers to interacting with tablet computers: A focus group study. *Frontiers in Psychology*, 8, 2017.

[23] Bo Xie. Older adults, computers, and the internet: Future directions. *Gerontechnology*, 6(4):223–233, 2007.

[24] Yijun Yu, Julio Cesar, Julio Leite, and John Mylopoulos. From goals to aspects: Discovering aspects from requirements goal models. *Aspect-oriented programming*, 05 2004.

[25] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. The impact of continuous integration on other software development practices: A large-scale empirical study. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 60–71, 2017.