Improving the User Experience of the Coding Editor in QDAcity

MASTER THESIS

Bahareh ChalayAmoly

Submitted on 2 May 2025

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Dr. Andreas Kaufmann
Prof. Dr. Dirk Riehle, M.B.A.

Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Erlangen, 2 May 2025

License

Erlangen, 2 May 2025

Abstract

This thesis enhances the usability of the QDAcity coding editor, a cloud-based tool for qualitative data analysis (QDA), by addressing key limitations in document organization, code visualization, and interaction efficiency. Three primary improvements were implemented: a hierarchical document group management system, a redesigned Codemap for intuitive code relationship visualization, and inline editing to streamline workflows. Developed using React.js, Java with Spring Boot, and Google Cloud Platform, these features were rigorously tested against functional and non-functional requirements. Evaluation results confirm improved usability, particularly for non-technical researchers, with robust real-time collaboration and maintainable architecture. This work advances QDAcity's effectiveness, aligning with user-centered design principles and contributing to accessible QDA tools.

Contents

List of Figures

List of Tables

List of Code Snippets

Acronyms

QDA Qualitative Data Analysis

UX User Experience

UI User Interface

JS JavaScript

HTML HyperText Markup Language

REST Representational State Transfer

GCP Google Cloud Platform

GAE Google App Engine

GCS Google Cloud Storage

CES Collaborative Editing Service

IA Information Architecture

HCI Human-Computer Interaction

UCD User-Centered Design

API Application Programming Interface

CRDT Conflict-Free Replicated Data Type

DAO Data Access Object

UML Unified Modeling Language

FR Functional Requirement

NFR Non-Functional Requirement

CSL Codesystem Language

1 Introduction

Qualitative Data Analysis (QDA) is a widely used method for analyzing unstructured data by identifying patterns, themes, and relationships. It is especially important in the social sciences, where researchers aim to understand the *what*, *why*, and *how* of human behavior (Strauss, 1987). A key part of QDA is the *coding* process, where researchers tag segments of text with labels representing theoretical constructs. This process allows for a more structured analysis and supports theory development based on the data. Therefore, applications that support QDA must offer intuitive tools for both annotating data and navigating complex conceptual structures (Kaufmann & Riehle, 2018).

QDAcity[1] is an application designed to help researchers organize, structure, store, and version their analysis artifacts. It is useful in both scientific and practical research contexts. At the center of the application is a coding editor that enables users to analyze qualitative data efficiently (QDAcity, 2025). However, as the feature set of QDAcity grows and user expectations increase, the need for an improved User Experience (UX) becomes more important especially in the coding editor. In particular, the editor lacked support for organizing uploaded documents. Additionally, QDAcity featured a Unified Modeling Language (UML)-based editor that relied on complex meta-modeling concepts. While powerful, this approach presented a high barrier to entry for new or non-technical users. To improve usability and accessibility, this feature required a simplified and more intuitive user interface, as well as a shift in focus toward a more broadly applicable conceptual Codemap.

According to the official QDAcity documentation (QDACityDocumentation, 2024), the application already includes basic features for document and code management, such as uploading documents and organizing codes in a tree structure. However, some critical usability aspects were missing or underdeveloped prior to the redesign presented in this thesis. For example, users could not create folders, limiting their ability to categorize and manage documents in a structured and flexible hierarchy that supported different research themes or stages. In addition,

[1] https://qdacity.com

the feature that showed relationships between codes, known as the Codemap, was difficult to access and not easy to use, especially for users without technical experience. These limitations reduced the efficiency and user satisfaction of the application and demonstrated a clear need for improvements in both organization and visualization features.

This thesis addresses these problems by improving the user experience of the QDAcity coding editor through the design and implementation of three main aspects:

- **Document Categorization:** Previously, QDAcity did not provide a way to organize uploaded documents in the coding editor. To address this limitation, a hierarchical categorization system was implemented, enabling users to define and assign categories to their documents. This feature improves document management by allowing users to structure their data based on themes or project organization, making navigation and analysis easier.
- **Codemap:** The UML editor, which visualized relationships between codes using object-oriented concepts, was refactored and redesigned to be more user-friendly. The new version simplifies complex visualizations, improves interaction, and adds features that make exploring codes easier and more intuitive for both technical and non-technical users.
- **Reduced Use of Modal Dialogs:** The original interface relied heavily on modal dialogs for managing codes and documents, which interrupted the user's workflow. In this thesis, I reduced the number of required dialogs by integrating actions directly into the interface. This change streamlines user interactions, improves efficiency, and contributes to a smoother and more continuous editing experience.

These improvements were developed using the existing Java backend and React frontend architecture of QDAcity[2], which is hosted on the Google Cloud Platform (GCP) to ensure performance and scalability. The design process followed usability principles, iterative testing, and software engineering best practices.

The main goal of this thesis is to improve the coding experience in QDAcity, making it more intuitive, better organized, and more accessible, especially for users who do not have a technical background. By improving these three core components, the application will better support researchers working with complex qualitative datasets.

The rest of this thesis is organized as follows: Chapter 2 presents a review of literature on usability challenges in QDA tools. Chapter 3 defines the system and user requirements. Chapter 4 describes the software architecture and design. Chapter 5 explains the implementation of the three main features. Chapter 6

[2]https://qdacity.com

presents the evaluation of the improvements. Finally, Chapter 7 concludes with a summary and suggestions for future work.

2 Literature Review

2.1 Usability in Qualitative Data Analysis Tools

QDA tools are widely used for organizing and interpreting unstructured textual data. However, many of these tools still face usability challenges, especially for users who do not have a technical background. These challenges are often caused by complex interfaces, poor interaction design, and the lack of organizational features that support larger and more complex research projects (Corti et al., 2019; Leech & Onwuegbuzie, 2008).

Effective management of large document sets remains a significant usability challenge in QDA tools. Beyond hierarchical organization, which supports structured categorization through trees, alternative strategies enhance user efficiency. Searching enables rapid location of documents using keywords or metadata, while filtering allows users to isolate subsets based on criteria like code associations. Tagging offers flexible, non-hierarchical labeling, facilitating cross-referencing across documents. Annotations further enrich documents with contextual notes, aiding interpretation without altering original content (Evans, 2011; Woods et al., 2016). These approaches, when combined with clear organizational structures, reduce cognitive overload and support diverse research workflows, complementing hierarchical methods like those implemented in this thesis (Section 5.1). Studies emphasize that such multifaceted strategies improve retrieval accuracy and user confidence, particularly in complex projects (Corti et al., 2019).

Another common problem in QDA software is how relationships between codes and concepts are shown. Visual tools like code trees, concept maps, or codemaps are very helpful for exploring themes and building theories. But in many platforms, these tools are designed in a way that assumes technical knowledge. They often use visual metaphors or abstract models that can be confusing to non-technical users (Jackson & Bazeley, 2008). If users cannot understand these visualizations easily, it becomes difficult to use them effectively in the analysis process.

The visualization of code relationships poses usability challenges in QDA tools.

While code trees and concept maps aid thematic exploration, alternative approaches broaden accessibility. Network graphs illustrate complex interconnections with nodes and edges, revealing patterns not visible in linear hierarchies. Timelines map temporal relationships, contextualizing code evolution, and matrix displays quantify code co-occurrences for comparative analysis (Davidson et al., 2019; Silver & Lewins, 2014). These methods, when designed with intuitive metaphors, mitigate the reliance on technical knowledge noted in prior studies (Jackson & Bazeley, 2008). Although this thesis focuses on a simplified node-edge Codemap (Section 5.2), such diverse visualizations offer complementary insights, enhancing analytical flexibility for varied research needs.

2.2 Information Architecture in Research Software

Information Architecture (IA) refers to the way information is structured, connected, and presented in digital systems. In QDA tools, this includes how documents, codes, and conceptual relationships are organized and accessed. A strong IA helps users build a clear mental model of their project, reduces cognitive load, and improves both navigation and interaction (Rosenfeld et al., 2015).

In this thesis, IA played a key role in shaping the redesign of three major features. First, the introduction of hierarchical document groups give users the ability to organize their data in a way that fits their individual workflows, such as by theme, data source, participant, or stage of analysis. This supports scalability and allows for more efficient data handling in large projects. Second, the Codemap provides a visual representation of code relationships. As a spatial interface, it reflects the conceptual structure of the analysis, helping users navigate and explore connections between codes more intuitively. Finally, reducing the use of modal dialogs allowed users to act directly on items in context, improving interaction flow and preserving their sense of orientation.

Together, these improvements demonstrate that IA is not limited to how data is stored, but extends to how users perceive and interact with it across the system.

2.3 User-Centered Design and Affordances

In *Human-Computer Interaction (HCI)* and software engineering, *User-Centered Design (UCD)* is a critical approach that prioritizes the needs, preferences, and skills of the end user throughout the design process. In the context of QDA software, UCD involves creating interfaces that enable researchers to perform analytical tasks efficiently without requiring technical expertise. This includes designing simplified workflows, reducing unnecessary complexity, and providing clear, consistent feedback to guide user interactions (Cooper et al., 2007).

Affordances are closely related to UCD. They describe the cues in an interface that suggest possible actions to the user. For example, in the Codemap, nodes and edges should visually indicate that they can be clicked, moved, or edited, even without explicit instructions. Improving affordances in such visual tools helps users explore their data more confidently, especially in the early stages of analysis when the structure is still evolving.

To improve usability and reduce frustration, several concrete strategies were applied in this thesis:

- **Tooltips:** All interactive elements in the Codemap (nodes, edges, buttons) display tooltips on hover. This provides immediate guidance without needing to refer to external documentation.
- **Inline Editing:** Many modal dialogs were replaced with inline interactions. For example, document group names and Codemap node labels can now be edited directly within the view, avoiding context switches and keeping the user focused.
- **Visual Consistency:** Standard User Interface (UI) conventions were followed, such as using a "hamburger" icon for collapsed menus, and placing delete or edit icons where users expect them. Mimicking common HCI patterns lowers the learning curve, especially for new users.
- **Info Icons and Explanatory Texts:** Where appropriate, info icons were added next to complex HCI elements. Clicking these opens a short inline explanation, helping users understand advanced features without needing to leave the current screen.

These design decisions are aligned with international UX standards such as ISO 9241-210 (International Organization for Standardization, 2019), which emphasizes effectiveness, efficiency, and user satisfaction as core usability goals.

2.4 Summary and Gap Analysis

This review of academic literature and HCI design principles shows that many QDA tools still face important usability problems:

- Several tools lack structured document organization, which can limit scalability and make it difficult to manage large or complex datasets.
- Visual tools like codemaps are often too complex or technical, making them difficult to use for researchers without technical knowledge.

The features developed in this thesis, hierarchical document categorization, an improved Codemap and reduced use of modal dialogs, are designed to solve these

issues. They are based on established principles from HCI such as IA, UCD, and affordances. These contributions aim to improve the user experience of QDA software, making it more accessible, organized, and effective for a wider range of users.

3 Requirements

This chapter defines the software requirements for the features implemented in this thesis. These requirements are based on the findings from the literature review, the limitations identified in the current QDAcity application, and user experience principles from ISO/IEC 25000 (SQuaRE). The goal of this chapter is to provide a clear understanding of what the system should do (Functional Requirement (FR)) and how it should perform (Non-Functional Requirement (NFR)).

3.1 Scope of the System

The scope of this thesis focuses on improving the user experience of the QDAcity coding editor, primarily through enhancements to its interface and interaction design. While the main contributions were implemented on the frontend, backend changes were made where necessary to support new functionality. The system architecture and core infrastructure were left unchanged beyond what was required for these specific improvements.

This work improves QDAcity in the following three areas:

- **Document Categorization:** Enable users to assign and manage hierarchical categories for uploaded documents. This improves the organization and navigability of research data, particularly in large or complex projects.
- **Codemap Redesign:** Replace the previous UML-based visualization with a simplified and more accessible conceptual map. The new Codemap improves usability, especially for non-technical users, by allowing them to explore and manage code relationships more intuitively.
- **Reduced Use of Modal Dialogs:** Simplify the editing workflow by minimizing the number of modal dialogs required for managing documents and codes. Key actions can now be performed directly within the main interface, which improves interaction flow and reduces context switching.

This work is focused on improving usability, organization, and accessibility of

the coding interface, especially for users without technical or programming backgrounds.

3.2 Functional Requirements

Functional requirements describe the specific behaviors and functionalities that the system must support. These requirements define what the software should do. The following list outlines the key functional requirements for this project:

3.2.1 Document Group Management

FR-DG-1 – Create and Rename Document Groups

Users must be able to create new document groups and edit the names of existing ones. Each group is uniquely identifiable and can serve as a container for documents or other document groups.

FR-DG-2 – Display Document Groups as a Tree Structure

Document groups must be displayed in a collapsible and expandable tree structure. The tree must visually represent parent-child relationships between groups and documents.

FR-DG-3 – Change Hierarchy Using Drag and Drop

Users must be able to reorganize the tree structure using drag-and-drop operations. This includes reordering document groups at the same level and nesting groups within others.

FR-DG-4 – Delete Document Groups with User Choice

When a document group is deleted, the system must prompt the user to choose whether to (a) delete all contents inside the document group, or (b) move them to another existing document group.

FR-DG-5 – Assign Documents to Document Groups via Drag and Drop

Users must be able to assign documents to specific document groups by dragging and dropping them onto a document group node. When dropped, the target group should auto-expand and become the active selection.

FR-DG-6 – Collaborative Editing of Document Group Structure

The system must support real-time collaborative editing of the document group hierarchy. Changes made by one user (e.g., creating, renaming, or moving a group) should be reflected for all other users viewing the same project without requiring manual refresh, ensuring a consistent and up-to-date view of the structure.

FR-DG-7 – Auto-Expand Tree on Drop

If a document or document group is dragged into a collapsed document group, the target group must expand automatically upon hover or drop to reveal its contents and set the newly added item as active. This interaction should help users maintain orientation within the hierarchy.

3.2.2 Codemap Functionality

FR-C-1 – Enable Codemap Tab

Users must be able to enable or disable the Codemap view via a toggle or menu option. When enabled, a new tab should appear, allowing interaction with the Codemap editor.

FR-C-2 – Add Codes to Codemap

Users can add codes to the Codemap by dragging them from the code list into the Codemap area. Each code will appear as a node on the codemap, positioned where the user dropped it.

FR-C-3 – Remove Codes from Codemap

Users can also choose to remove a node from the codemap. This action only affects the node's visibility on the Codemap, it does not delete the code from the code system. The underlying code and its relationships remain intact and will automatically become visible if the deleted code is re-added to the codemap at a later point in time.

FR-C-4 – Manage Node Properties

Each node in the Codemap represents a code and must support the following editable properties:

- **Border style**: solid, dashed, dotted
- **Shape**: rectangle, triangle, circle, rhombus and hexagon

- **Color**: background color of the node
- **Codemap labels**: tags that can be created, assigned, and removed
- **Description**: editable text associated with the node
- **Visibility**: option to remove a node from the Codemap view (without deleting it from the code system)

FR-C-5 – Manage Relationships (Edges) Between Nodes

Users can draw and manage connections (edges) between nodes. Each edge should support the following attributes:

- **Label**: text to describe the relationship
- **Line style**: solid or dashed
- **Arrowhead style**: No arrow, arrow open, arrow block, arrow block fill
- **Line color**: color picked with through the browser API for color selection
- **Delete edge**: remove connection between nodes

Edges can be managed directly on the Codemap canvas or through the node/edge properties panel.

FR-C-6 – Centralized Properties Panel

All editable attributes of nodes and edges must be accessible in the code properties menu. Users can select elements from the Codemap or from a sidebar to update visual and semantic attributes easily.

FR-C-7 – Simplified Codemap Model

The Codemap feature must be based on a newly designed, simplified conceptual model. The legacy implementation, which followed object-oriented programming (OOP) structures, must be removed. The new system should be understandable and usable by non-technical users while remaining flexible for complex research tasks.

FR-C-8 – Auto-Save Codemap State

Any changes made to nodes, edges, or their properties must be saved automatically in real time. Users do not need to manually save Codemap updates.

FR-C-9 – Node Display Behavior Based on Shape and State

Nodes in the Codemap must visually adapt based on their shape and whether they are in an expanded or collapsed state:

- When a node is in **rectangular shape** and in an **expanded state**, it must display the code label and its description directly inside the node.
- For all other shapes, or when the node is in a **collapsed state**, the label and description must not be shown in the node body but must remain accessible in the code properties panel.

FR-C-10 – Resize Nodes in Codemap Users can resize nodes interactively on the canvas, adjusting their width and height to suit visualization needs. Resized dimensions are saved to the node's properties.

This behavior ensures clarity of information while maintaining a consistent and readable layout in the Codemap, especially when managing large numbers of nodes.

3.2.3 Codemap Labels Overview

The Codemap must support the use of labels that allow users to classify and annotate nodes. Labels must be managed as user defined entities within a project and can be reused across multiple nodes.

FR-CL-1 – Create and Edit Labels

Users must be able to create new labels and edit existing ones. Each label must have a unique name and an associated color. The system must prevent the creation of duplicate label names within the same project.

FR-CL-2 – Assign and Remove Labels from Nodes

Users must be able to assign one or more labels to a node in the Codemap. Labels can also be removed at any time without affecting the node's other data.

FR-CL-3 – View Labels on Nodes

When a node has labels assigned, those labels must be visibly displayed inside the node if the shape is rectangular and the node is expanded. Regardless of shape or state, all labels must also be viewable and editable through the code properties panel. For non-rectangular or collapsed nodes, only the code properties panel should be used to maintain a clean and readable layout.

FR-CL-4 – Label Consistency Across Codemap

Labels must be managed globally within a project. Any changes to a label, such as renaming, changing its color, or deleting it, must be reflected across all nodes to which the label is assigned. Label data is managed as a Conflict-Free Replicated Data Type (CRDT), ensuring consistent updates are synchronized with all connected clients in real-time.

3.2.4 Shared UX and Interaction Features

FR-UX-1 – Reduced Use of Modal Dialogs

The system must minimize the use of modal dialogs for managing documents, document groups and codes. Whenever possible, editing and configuration tasks should occur directly within the main interface. This inline interaction design reduces context switching and supports a more continuous workflow.

The system must use inline editing for common tasks, such as renaming and creating documents, document groups, and codes, to minimize the use of modal dialogs. Modal dialogs may be used for complex configuration tasks requiring extensive user input. This inline interaction design reduces context switching and supports a continuous workflow.

FR-UX-2 – Inline Editing Support

Users must be able to rename documents, document groups and codes directly within the view, without opening a modal dialog. Editable fields should be visually highlighted when active.

FR-UX-3 – Confirming Inline Changes

To confirm changes made in inline editing mode, users must be able to either:

- Click a visible confirmation icon, or
- Press the Enter key

FR-UX-4 – Canceling Inline Changes

Users must be able to cancel inline changes by:

- Clicking anywhere outside the active editing field, or
- Pressing the Escape (Esc) key

Canceled edits must revert the field to its previous value without saving any changes.

FR-UX-5 – Inline Creation Behavior

When a user creates a new document group and code, the system must insert a new item directly into the interface with a pre-selected default title. The user can then rename it immediately using inline editing.

If the user cancels the action (by clicking outside or pressing the Escape key) before confirming the new title, the item must be discarded and not added to the system.

FR-UX-6 – Consistent Input Field Behavior

All input fields within the QDAcity interface, including those for documents, document groups, codes, and labels, must exhibit consistent interaction behavior. Users must confirm input by pressing the Enter key and cancel input by pressing the Escape (Esc) key, ensuring a uniform editing experience across the system.

3.3 Non-Functional Requirements

This section defines the system's expected behavior beyond its core functionality, focusing on quality attributes critical to user experience, performance, and maintainability. These non-functional requirements are derived from the ISO/IEC 25010:2011 software quality model, part of the ISO/IEC 25000 (SQuaRE) standard (International Organization for Standardization, 2014).

3.3.1 NFR1 – Usability

- **NFR1.1 – Learnability**: Basic tasks in the coding editor (e.g., creating, editing, or deleting a document group; adding, editing, deleting, or assigning a Codemap label) must provide clear, step-by-step guidance (e.g., tooltips, inline help) within the user interface. Each task must require no more than four steps to complete without external assistance, verified through heuristic evaluation using Nielsen's "Help and Documentation" heuristic (Nielsen, 1994). This heuristic ensures that help is visible (e.g., tooltips on buttons), contextual (task-specific), concise (brief instructions), and non-intrusive (inline rather than modal dialogs), enabling non-technical users to learn tasks efficiently.
- **NFR1.2 – Aesthetics**: Text and interactive elements (e.g., buttons, inputs, Codemap nodes, and labels) in the coding editor must have a contrast ratio of at least 4.5:1 to ensure readability, verified using a contrast checker tool in accordance with WCAG 2.1 Level AA standards. This enhances visual clarity for users.

3.3.2 NFR2 - Performance

- **NFR2.1 – Responsiveness**: Drag-and-drop operations in the coding editor must complete visual updates within 600 milliseconds from the drop event (e.g., `mouseup`) to the updated position or state. This applies to dragging a code to the Codemap editor, a document into a document group, and a node within the mxGraph Codemap editor. Compliance is verified through performance profiling on qdacity.com using a standard laptop (e.g., 8GB RAM, 2.5GHz CPU) with the browser network set to Fast 4G conditions, averaging 10 runs per operation.
- **NFR2.2 - Save Operation**: Saving changes in the coding editor, including creating, editing, or deleting a document group, creating, editing, or deleting a Codemap label, and retrieving a list of document groups, must complete within 750 milliseconds. Compliance is verified through network profiling on qdacity.com using a standard laptop (e.g., 8GB RAM, 2.5GHz CPU) with the browser network set to Fast 4G conditions, averaging at least 5 runs per operation type.

3.3.3 NFR3 – Compatibility

The coding editor must work correctly in the following web browsers in versions since 2024: Chrome, Firefox, and Safari. It should also be usable on both desktop and tablet devices without the need for additional software.

3.3.4 NFR4 – Maintainability

- **NFR4.1 - Modularity**: The codebase must adhere to the separation of concerns principle, ensuring each React component, Spring Boot service, and API endpoint (Java and JavaScript) handles a single responsibility (e.g., UI rendering, data access, API handling). All public methods in Spring Boot services and public functions in React components and JavaScript endpoints must be fully documented with JavaDoc or JSDoc, respectively. Compliance is verified through manual code reviews of representative components and services.
- **NFR-4.2 - Testability**: Unit and acceptance tests for frontend (React) and backend (Spring Boot) sections related to document group management and the Codemap editor must achieve at least 80% line coverage, verified through standard coverage measurement methods for unit tests and Selenium-based acceptance tests.
- **NFR4.3 - Modifiability**: All UI components must use the QDAcity design system implemented with the styled-components library and ex-

pose a consistent API (e.g., using prop-types) to ensure reusability, verified through code reviews.

3.3.5 NFR5 – Real-Time Collaboration

When one user makes changes to a document group or Codemap in the coding editor, these changes must be visible to other users within 2 seconds under typical concurrent usage (2 to 5 users on a stable internet connection), ensuring effective real-time collaboration. Compliance is verified through manual testing with timestamp logging on a local machine, simulating network conditions (e.g., Fast 4G).

3.3.6 NFR6 – Visual Feedback for Drag-and-Drop

During drag-and-drop interactions in the coding editor, the system must provide clear visual cues, such as highlighting potential drop targets on hover and changing the cursor to indicate valid actions, to reduce user uncertainty and improve interaction clarity. Compliance is verified through manual testing.

4 Architecture and Design

This chapter provides an overview of the architecture of the QDAcity system and the design decisions applied during the enhancement of its coding editor. The improvements target three main features: document group management, Codemap redesign, and the transition from modal dialogs to inline editing. The architecture utilizes a client-server model, integrating a React frontend, a Java-based backend with Spring Boot, and GCP hosting. Each design decision is explained with a detailed justification, including an evaluation of considered alternatives, such as the rejection of the Composite pattern for document groups and the continued use of mxGraph for the Codemap, to ensure optimal functionality and user experience.

4.1 System Overview

QDAcity operates as a cloud-based application designed specifically for QDA, facilitating document organization, code visualization, and real-time collaboration. The system follows a client-server architecture, consisting of:

- **Frontend**: Developed using React, providing a dynamic and responsive UI that supports interactive features like drag-and-drop and real-time updates.
- **Backend**: Implemented partially in Java with Spring Boot and partially in TypeScript for Node.js. The Java/Spring Boot component manages core business logic, data persistence, and Representational State Transfer (REST) Application Programming Interface (API) endpoints, while the TypeScript/Node.js component handles real-time communication through WebSocket protocols via the Collaborative Editing Service (CES).
- **Cloud Infrastructure**: Hosted on GCP, leveraging services such as Google App Engine (GAE) for the Java backend, Cloud Run for the CES, Google Cloud Storage (GCS) for file storage, and Google Datastore for structured data management.

4.1.1 Architectural Overview

The high-level architecture is depicted in Figure 4.1:

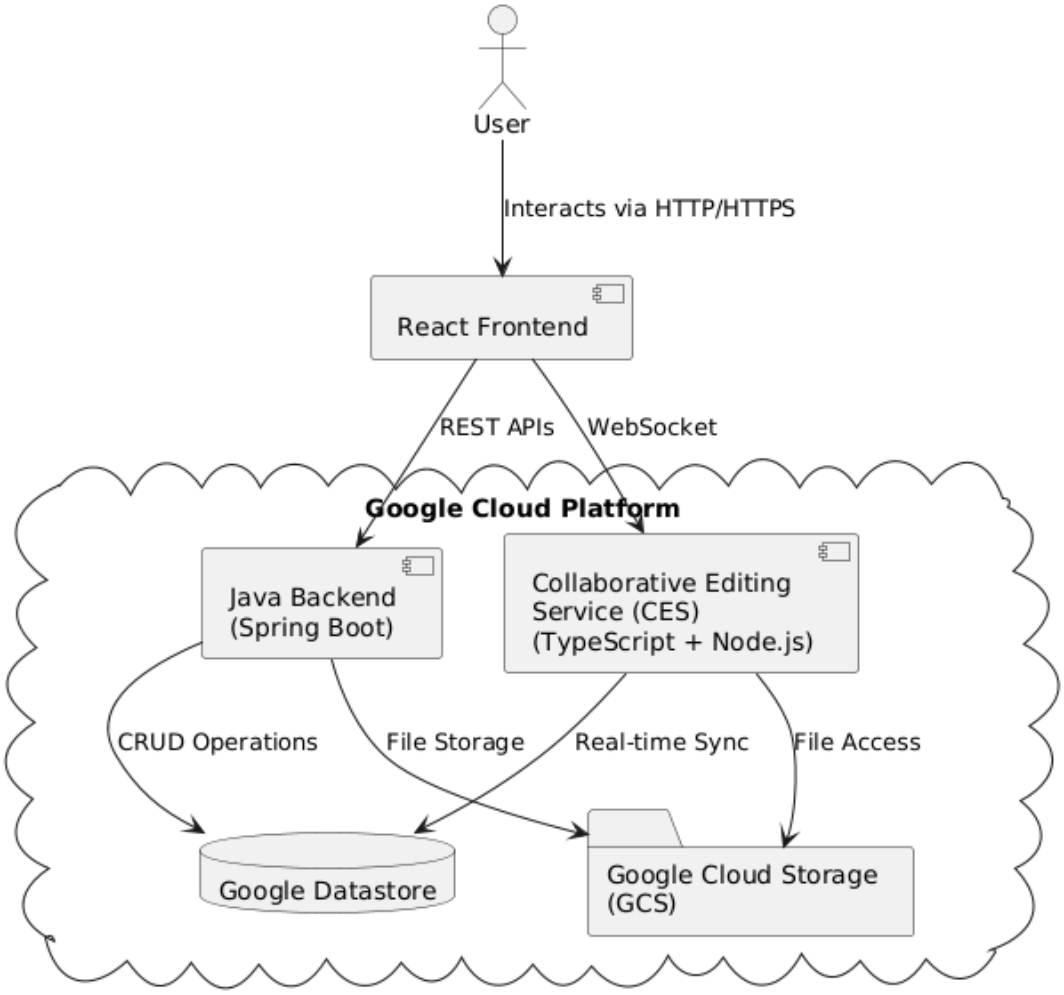

Figure 4.1: High-level architecture of QDAcity

The architecture facilitates the following interactions:

- Users engage with the system through a web browser interfacing with the React frontend.
- The React frontend communicates with the Java backend via REST API for data operations and with the CES (running on Cloud Run) via WebSocket for real-time collaboration.
- The Java backend interfaces with GCS for file storage and Google Datastore for structured data.

- GAE hosts the Java backend, while Cloud Run hosts the CES to support WebSocket functionality.

Data Access Layer

The backend employs the Data Access Object (DAO) pattern to abstract data operations, separating business logic from persistence concerns (Fowler, 2003). Each DAO class (e.g., `DocumentGroupDAO`) encapsulates interactions with Google Datastore, providing methods for CRUD operations (e.g., retrieval, updates) and specialized queries (e.g., `getAllByProjectKey`). This abstraction enhances maintainability (NFR4) by isolating database access, allowing changes to persistence mechanisms without affecting higher layers. The DAO pattern integrates with Spring Boot's dependency injection, ensuring modularity and testability across the system.

4.2 Technology Stack

The technology stack for QDAcity, consisting of pre-existing components, was leveraged to fulfill both functional and non-functional requirements, including scalability, real-time collaboration, performance, and maintainability, as outlined in Chapter 3. Each component plays a key role in forming a cohesive system optimized for usability, compatibility, and operational efficiency.

4.2.1 Frontend: React.js

React forms the foundation of QDAcity's frontend, utilizing its component-based architecture to enhance UI modularity and reusability. This structure strengthens maintainability by streamlining updates and extensions (Meta, 2023). The virtual DOM optimizes rendering and ensures seamless drag-and-drop interactions, which are critical for a responsive user experience. React's state management, component lifecycle management, efficient DOM updates, and declarative component definitions enable a robust and intuitive interface, supporting seamless user interactions across the application.

4.2.2 Backend: Java with Spring Boot and TypeScript with Node.js

The backend employs distinct services, with:

- **Java with Spring Boot**: Manages core business logic, data persistence, and REST API endpoints. Spring Boot's convention-over-configuration approach accelerates development, while its strong typing and object-oriented

design enhance code maintainability and testability (Spring Boot Documentation, 2023).

- **TypeScript with Node.js**: Handles real-time communication through WebSocket protocols via the CES. Node.js's event-driven architecture is well-suited for real-time features, ensuring low-latency updates (Node.js Foundation, 2023).

4.2.3 Database: Google Datastore

Google Datastore, a NoSQL database service, was chosen for its scalability, reliability, and GCP integration. It efficiently stores structured data, such as document groups and codes, supporting real-time queries vital for collaboration (Google Cloud, 2023b). Its schema-less design allows flexibility in data modeling, while automated backups ensure data integrity.

4.2.4 File Storage: Google Cloud Storage (GCS)

GCS is an object storage service that manages both structured and unstructured data (e.g., audio, PDFs, JSON files) with scalability and durability, supporting large uploads and version control (Google, 2023b). Its low-latency access and GCP integration optimize backend interactions for research-oriented workflows. Additionally, GCS provides soft deletes for data safety, allowing recovery of accidentally deleted objects, and tiered storage options to balance latency and cost, enabling frequent-access data to reside in high-performance tiers while archival data uses lower-cost, higher-latency tiers.

4.2.5 Cloud Hosting: Google App Engine (GAE) and Cloud Run

- **GAE**: Provides a fully managed environment for the Java backend, with auto-scaling to handle variable workloads. However, since GAE standard does not support WebSockets, the CES is hosted separately (Google, 2023a).
- **Cloud Run**: Hosts the CES (TypeScript/Node.js) to enable WebSocket functionality, ensuring real-time collaboration features are supported efficiently (Google Cloud, 2023a).

4.2.6 External Libraries

- **mxGraph**: Facilitates Codemap visualization with robust graph-editing features (e.g., drag-and-drop, styling), ensuring responsive UI interactions (JGraph, 2023).

- **y.js**: Enables real-time collaboration via CRDT-based synchronization, supporting document group updates with minimal latency (Jahns, 2023).

This stack collectively ensures QDAcity meets its operational demands, balancing performance and scalability with UCD.

4.3 Component Interaction

QDAcity's frontend and backend employ a hybrid architecture integrating RESTful APIs and WebSocket protocols, optimizing data management and real-time synchronization for scalability and responsiveness.

- **REST APIs**: Handle CRUD operations (e.g., document group creation, Codemap retrieval) with a stateless, HTTP-compatible design (Fielding, 2000).
- **WebSocket**: Supports real-time updates (e.g., document group synchronization) via a persistent TCP connection, enhanced by y.js CRDTs for consistency (IETF, 2011; Jahns, 2023).

4.3.1 Communication Diagram

Figure 4.2 illustrates the sequence for creating a document group:

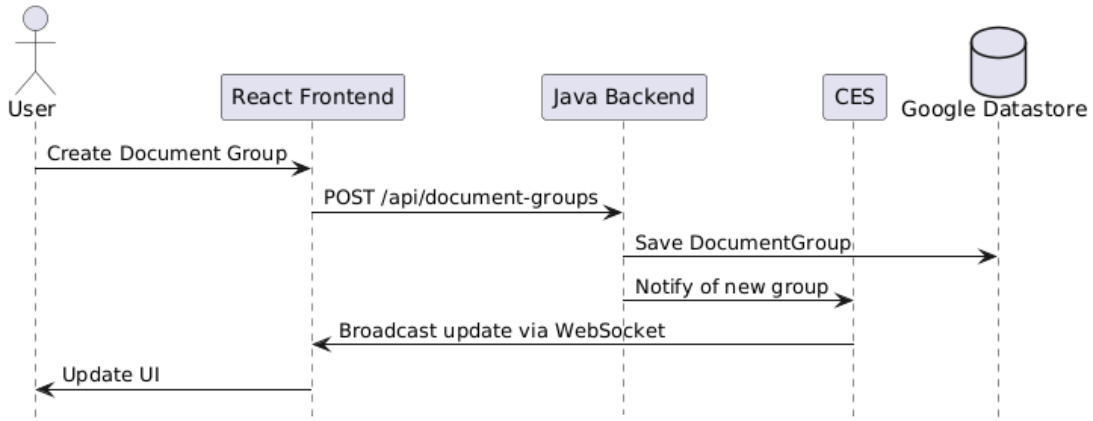

Figure 4.2: Sequence diagram for creating a document group

- **Step 1**: User initiates "Create Document Group" via the frontend.
- **Step 2**: Frontend sends a POST request to the Java backend's REST API (e.g., `/api/document-groups`).

- **Step 3**: Java backend creates a `DocumentGroup` entity and persists it in Google Datastore.
- **Step 4**: Java backend notifies the CES via an internal API or message queue.
- **Step 5**: CES broadcasts the update to all connected clients via WebSocket.
- **Step 6**: Frontend updates the UI with the new group in real-time.

4.4 Feature Breakdown

This section outlines the architectural and design frameworks that support the three core features developed within this thesis: document group management, Codemap redesign, and inline editing. Each feature is examined through its structural composition, interaction paradigms, and integration within the overarching system architecture. The design choices reflect established software engineering principles, ensuring alignment with both functional objectives and non-functional requirements such as scalability, usability, and real-time collaboration.

4.4.1 Document Group Management

Document group management facilitates the hierarchical organization of research documents within a project, offering users an intuitive mechanism to structure their data.

Structural Design

The hierarchy employs the **Adjacency List** pattern, a widely recognized approach to modeling tree structures due to its simplicity and efficiency in traversal (Celko, 2004).

User Interaction

The design supports dynamic visualization of the hierarchy, enabling expansion and collapse of groups with real-time synchronization across users, which is critical for collaborative research environments.

Collaborative Framework

Modifications to the hierarchy are propagated instantly, ensuring consistency in distributed sessions without the need for intricate conflict resolution mechanisms (Jahns, 2023).

4.4.2 Codemap Redesign

The Codemap, originally designed as a QDAcity UML editor, has been fundamentally restructured to better support QDA. Previously, it relied on a complex meta-model implemented via the Codesystem Language (CSL), which imposed a rigid framework for defining codes and their relationships. This structure, while robust, presented usability challenges for non-technical researchers due to its steep learning curve and technical constraints. The redesign introduces a simplified node-edge structure, enhancing accessibility for non-technical users while preserving flexibility for complex analytical tasks.

Original Meta-Model Architecture

The legacy UML editor was built on a meta-model comprising `MetaModelEntity` and `MetaModelRelation` objects stored in the database. Each `MetaModelEntity` represented a CSL item, such as "Aspect," "Relationship," or "Property," with attributes including type, abstraction level, and grouping. For instance, CSL defined 21 entity types, such as specialized "Aspect" variants and "Relationship" elements, interconnected by 20 `MetaModelRelation` objects specifying associations like generalizations or aggregations. Additionally, the system supported *relationship codes*, allowing relationships to be optionally associated with a code to denote specific conceptual links. This flexibility enabled nuanced modeling but added complexity. To visualize these structures, a JavaScript (JS) module, `MetaModelMapper`, translated CSL-based entities and relationships into UML elements (e.g., classes, generalizations, aggregations) based on predefined mapping rules, but only if they met structural conditions. While this architecture offered precision for technical users, it demanded significant training and adherence to object-oriented principles, creating a barrier for qualitative researchers unfamiliar with such paradigms.

Simplified Data Model

The redesigned Codemap eliminates the meta-model entirely, adopting a straightforward node-edge paradigm. The nodes directly represent codes, and edges denote user-defined relationships, free of the CSL's prescriptive constraints. This shift lowers the entry barrier for non-technical users by removing the need to navigate a complex framework, while still allowing arbitrary connections and properties for advanced research needs.

Visualization and Interaction

The Codemap's visual layer leverages a robust graph-editing library (JGraph, 2023), enabling intuitive features like drag-and-drop, node styling, and edge customization. The responsive canvas supports uninterrupted interactions when

adding, moving, or connecting nodes, rendered in real time within a user's session.

4.4.3 Inline Editing

The inline editing feature replaces modal dialogs(FR-UX-1), allowing users to modify content directly within the UI(FR-UX-2), enhancing workflow efficiency and user experience (Shneiderman, 1983). This UX improvement required no changes to QDAcity's existing architecture but leveraged different parts of the React frontend infrastructure. Previously, modal dialogs were implemented using React Portals, rendering them outside the main DOM hierarchy to manage focus and overlays. In contrast, inline editing uses React components rendered directly into the DOM at their respective positions, employing standard React mechanisms for state management and component lifecycle. Users can click an element (e.g., a code name), edit it in place, and confirm changes via a check icon or Enter key (FR-UX-3), with immediate visual feedback. This direct manipulation approach minimizes context switching, reduces cognitive load, and ensures seamless updates by synchronizing frontend state with the backend, aligning with usability goals (NFR1).

4.5 Design Decisions

This section explains the key architectural and design choices supporting the three main features developed in this thesis: document group management, Codemap redesign, and inline editing. Each choice results from careful consideration of alternatives, following principles of simplicity, compatibility with existing components, and alignment with the usability and maintainability goals described in Chapter 3. The selected approaches are justified through comparison with other possible solutions, ensuring the final designs provide optimal functionality and user experience within the QDAcity application's requirements.

4.5.1 Document Group

Chosen Approach: The document group feature employs a custom `DocumentGroup` entity with a `parentId` field to define hierarchical relationships, stored directly in the database.

- **Rationale**: This design employs the **Adjacency List** pattern, a widely-used method for representing tree structures in relational databases (Celko, 2004). The pattern's simplicity enables easy implementation and efficient navigation, meeting the requirement for intuitive document organization (FR-DG-2). Through the use of a nullable parentId field, the design accom-

modates both root-level and nested document groups, improving scalability while giving users greater organizational flexibility.

Alternatives Considered:

1. **Composite Pattern**:
    - **Description**: This pattern defines a uniform interface for both individual objects (e.g., documents) and composite objects (e.g., groups), enabling recursive composition (Gamma et al., 1994).
    - **Evaluation**: The Composite Pattern is well-suited for hierarchical systems where objects share similar behaviors. Documents and document groups display core differences (specifically, documents cannot have children), making a uniform interface redundant and potentially problematic for maintenance. Adopting this pattern would have required extensive refactoring across the application to align existing data models and logic, which was impractical given the straightforward requirements of the document hierarchy.
    - **Reason for Rejection**: The pattern's benefits were outweighed by the significant development effort needed for refactoring, particularly when the Adjacency List pattern already met the feature's needs with minimal changes to the existing architecture.
2. **Nested Set Model**:
    - **Description**: This model uses left and right bounds to encapsulate hierarchical relationships, optimizing subtree queries (Celko, 2004).
    - **Evaluation**: It is highly effective in scenarios requiring frequent retrieval of entire subtrees but complicates updates, as modifying a single node requires recalculating bounds across the tree.
    - **Reason for Rejection**: The use case prioritizes frequent modifications (e.g., drag-and-drop reorganization, FR-DG-3) over subtree queries, making the maintenance cost of Nested Sets impractical.

Table 4.1 compares the performance, time complexity, and space complexity of the Adjacency List pattern with the Composite Pattern and Nested Set Model, highlighting their suitability for the document group feature.

Table 4.1: Comparison of Design Approaches for Document Group Implementation

Criterion	**Adjacency List**
Performance	Fast for single-node operations; slower for subtrees
Retrieve	O(1)
Insert	O(1)
Reorganize	O(1)
Space Complexity	O(n)
Criterion	**Composite Pattern**
Performance	Good in-memory; slower with persistence
Retrieve	O(1) or O(k)
Insert	O(1) or O(k)
Reorganize	O(k)
Space Complexity	O(n + m)
Criterion	**Nested Set Model**
Performance	Fast subtree retrieval; slow updates
Retrieve	O(1)
Insert	O(n)
Reorganize	O(n)
Space Complexity	O(n)

Notes: In the table, n represents the total number of nodes, k denotes the number of children per group, and m is the total number of child references across all groups.

4.5.2 Codemap Visualization

Chosen Approach: The Codemap retains mxGraph as its visualization library, transitioning from a meta-model-driven UML structure to a simplified node-edge paradigm.

- **Rationale**: mxGraph provides comprehensive graph-editing capabilities, including drag-and-drop, node styling, and event handling, which fully satisfy the functional requirements for Codemap interaction (FR-C-2 - FR-C-5) (JGraph, 2023). Its prior integration into the QDAcity application's UML editor ensured compatibility with existing codebases, minimizing refactoring effort and maintaining system stability. The shift to a node-edge model enhances accessibility by eliminating the meta-model's complexity, aligning with usability goals for non-technical users (NFR1). However, mxGraph has not been maintained since its last release in October 2020, posing potential risks for long-term sustainability. Despite this, replacing mxGraph with a newer library or developing a custom solution was infeasible within

the time and resource constraints of this master's thesis, as such efforts would have significantly expanded the project's scope and duration.

Alternatives Considered:

1. **D3.js**:
    - **Description**: A JS library providing detailed management over data-driven visualizations (Bostock, 2011).
    - **Evaluation**: D3.js provides flexibility for custom layouts but lacks built-in graph-editing features like those in mxGraph, necessitating significant development to replicate functionality.
    - **Reason for Rejection**: The additional effort outweighed the benefits, given mxGraph's adequacy and established presence in the system.

4.5.3 Inline Editing

Chosen Approach: Inline editing replaces modal dialogs with direct manipulation of interface elements, enabling immediate content modification.

- **Rationale**: This design implements the **Direct Manipulation** principle, which prioritizes intuitive, immediate feedback to enhance user efficiency and satisfaction (Shneiderman, 1983). By allowing edits within the interface (e.g., renaming codes with a check icon confirmation), it minimizes context switching and aligns with usability requirements (NFR1). The solution employs React's state management to deliver consistent real-time updates, preserving workflow continuity.

Alternatives Considered:

1. **Modal Dialogs**:
    - **Description**: Retains the original modal-based workflow for editing tasks.
    - **Evaluation**: Modals provide a focused editing environment but interrupt the user's flow, requiring navigation through additional steps.
    - **Reason for Rejection**: The increased cognitive load and disruption contravened usability goals, favoring a streamlined alternative.
2. **Form-Based Sidebar**:
    - **Description**: Implements a persistent sidebar panel that appears alongside the main interface for editing tasks, replacing modal pop-ups (Cooper et al., 2007).

- **Evaluation**: A sidebar offers a dedicated space for editing area while partially maintaining visibility of the main view, potentially reducing context switching compared to modals. However, it still detaches the editing process from the content's location, requiring users to shift focus and manage an additional UI element.
- **Reason for Rejection**: The sidebar introduces unnecessary complexity for simple edits like renaming, where direct manipulation within the content area is faster and more intuitive.

4.6 External Libraries

This section describes the external libraries essential for the implementation presented in this thesis: mxGraph for visualization and y.js for real-time collaboration. These established libraries were adapted from the existing codebase and carefully expanded to support this thesis's key enhancement. Their continued use demonstrates a deliberate balance between reliability, compatibility with the React frontend and Java backend, and adherence to system requirements including usability (NFR1), performance (NFR2), and real-time collaboration (NFR5). By maintaining these existing libraries rather than introducing new ones, the system preserves architectural consistency while reducing integration complexity.

4.6.1 mxGraph

mxGraph forms the core visualization and interaction library for the Codemap. Initially employed in the system's UML editor, it has been adapted to display a streamlined node-edge structure, enabling users to intuitively manage codes and their connections.

- **Technical Merits**: mxGraph offers a robust suite of graph-editing functionalities, including drag-and-drop operations, dynamic node styling, and event-driven interactions, which directly fulfill the functional requirements for Codemap management (FR-C-2 - FR-C-5) (JGraph, 2023). The client-side JS implementation preserves frontend responsiveness in React through effective use of HyperText Markup Language (HTML) and SVG rendering capabilities.
- **Strategic Benefits**: The library's prior integration into the UML editor codebase provided a stable foundation, reducing development time and mitigating risks associated with adopting a new visualization tool. Its extensible architecture allowed seamless adaptation to the redesigned Codemap, supporting the shift from a complex meta-model to a user-centric design without necessitating extensive refactoring.

4.6.2 y.js

y.js provides the core infrastructure for real-time collaborative functionality in QDAcity, supporting document group management and other pre-existing collaborative features.

- **Technical Merits**: Using CRDTs, y.js maintains data consistency across clients without complex conflict resolution (Yjs, 2023). It enables real-time synchronization of document groups (FR-DG-6) with minimal delay. The lightweight design integrates efficiently with WebSockets, ensuring strong performance under concurrent use (NFR2).
- **Strategic Benefits**: By adapting y.js for document group functionality, this thesis built upon its existing implementation in QDAcity. This maintains consistent collaborative behavior throughout the system while avoiding additional synchronization libraries, thereby preserving architectural coherence and reducing maintenance requirements (NFR4).

5 Implementation

This chapter describes the technical implementation of the three core features developed in this thesis: document group management, Codemap redesign, and inline editing. These features enhance the user experience of the QDAcity application's coding editor, addressing the requirements outlined in Chapter 3 and leveraging the architecture detailed in Chapter 4. The implementation utilizes the existing React frontend, a hybrid backend with Java/Spring Boot and TypeScript/Node.js, Google Datastore for data persistence, and external libraries (mxGraph and y.js). Each subsection below explains how the features were realized, focusing on key components, integration strategies, and alignment with functional and non-functional requirements.

5.1 Implementation of Document Group Management

The document group management feature enables hierarchical organization of research documents, fulfilling the need for structured data management as outlined in QDAcityDocumentation (2024). This section covers backend and frontend development, real-time collaboration, and technical challenges.

The document group management feature enables hierarchical organization of research documents, addressing the critical need for structured data management in qualitative research as outlined in QDAcityDocumentation (2024). Initially termed "folders" in the requirements, the feature was renamed to "document groups" to better reflect its hierarchical structure, aligning with FR-DG-1 (Chapter 3). This section details the backend and frontend development, and real-time collaboration, ensuring compliance with functional requirements (FR-DG-1 to FR-DG-7) and non-functional requirements (NFR1, NFR5).

5.1.1 Backend Implementation

The backend, developed using Java with Spring Boot, introduces the `DocumentGroup` entity to model hierarchical document groups within a project. Key attributes include:

- `id`: Primary key.
- `parentId`: Nullable reference to the parent document group, using the Adjacency List pattern (Section 4.1.1).
- `projectKey`: Links to a project via ID and type.
- `title`: Group name.
- `color`: Optional visual customization.

The `DocumentGroup` entity uses Objectify annotations for persistence and Jackson for serialization. The `@Entity` annotation registers the class with Google Datastore, `@Id` marks the primary key, and `@Index` on `parentId` and `projectKey` optimizes query performance. The `@JsonIgnore` on `projectKey` prevents serialization to avoid exposing internal keys in REST responses, ensuring clean API outputs (FasterXML Team, 2023; Objectify Team, 2023).

```
@Entity
public class DocumentGroup implements Serializable {
    @Id private Long id;
    @Index private Long parentId;
    @JsonIgnore @Index private Key projectKey;
    private String title;
    private String color;
    // Getters and setters omitted
}
```

Code Snippet 5.1: DocumentGroup Entity Definition

The `BaseDocument` entity was extended with `documentGroupID`. The Adjacency List pattern supports hierarchical storage in Google Datastore.

```
@Entity(name="Document")
public abstract class BaseDocument implements Serializable, Cloneable, VisitableDocument, ↩
     HasGcsBlobs {
    @Id private Long id;
    @Index private Long projectID;
    private Long documentGroupID;
    // Getters and setters omitted
}
```

Code Snippet 5.2: BaseDocument Entity Definition

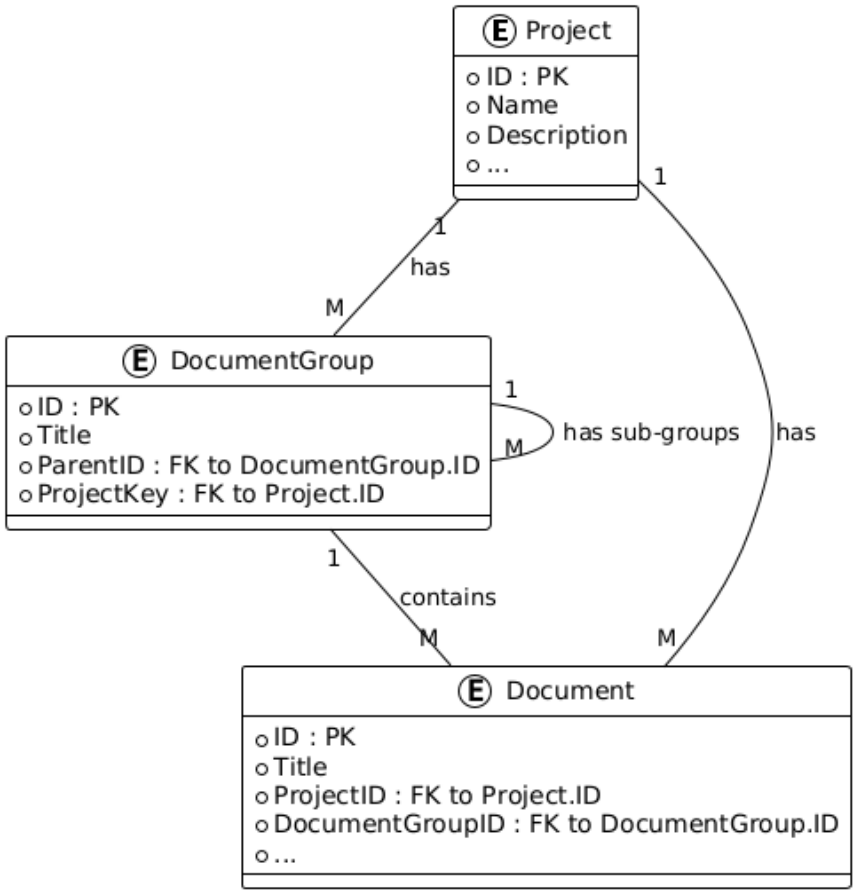

Figure 5.1: Entity Relationship Diagram for Document Groups and Documents

Data access is facilitated by the `DocumentGroupDAO` class, which extends `BaseDAO<DocumentGroup>`. This class provides methods such as `getAllByProjectKey` for retrieving project-specific document groups and `getAllDocumentGroupSubgroups` for accessing subgroups. The `DocumentGroupController` encapsulates business logic, supporting operations including creation, updating, deletion, and cloning of document groups.

REST endpoints (`DocumentGroupLegacyEndpoint`, `DocumentGroupEndpoint`) ensure secure access via permissions (e.g., `ProjectPermissions.CODING_EDITOR_CUD`), enabling seamless frontend-backend interaction.

These components and their interactions are illustrated in Figure 5.2, which shows the relationships between the data access layer, business logic, and REST API endpoints for document group management.

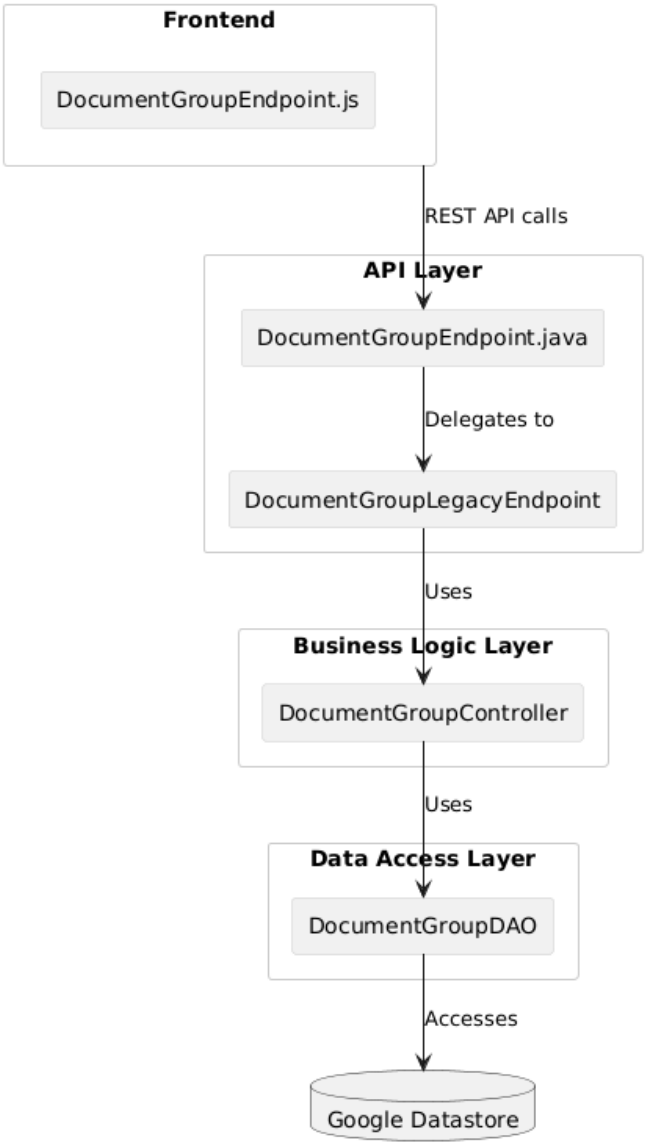

Figure 5.2: Component Diagram for Document Group Management

To prevent unintended data loss during deletion, a modal dialog was integrated into the frontend UI, prompting users to choose between deleting or reassigning child elements (FR-DG-4). This dialog, shown in Figure 5.3, ensures users are fully informed of the consequences of their actions, aligning with usability requirements (NFR1).

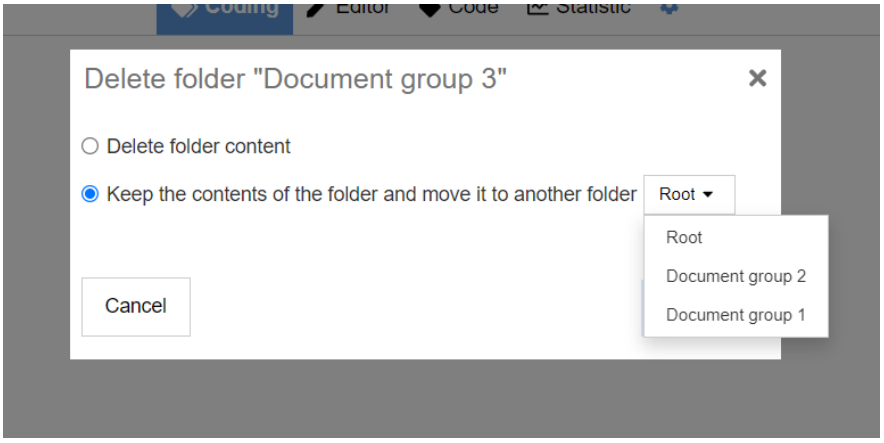

Figure 5.3: Document Group Deletion Options

5.1.2 Frontend Implementation

The frontend, developed using React.js, delivers an interactive interface for managing document groups, supporting hierarchical display (FR-DG-2), drag-and-drop reorganization (FR-DG-3, FR-DG-5), and a clean user experience (NFR1). It integrates with the backend through REST APIs for data operations and employs WebSocket-based synchronization for real-time collaboration.

Data Retrieval and Tree Construction

Data retrieval begins when a project loads, using `DocumentGroupEndpoint` to fetch document groups via `listDocumentGroupsByProjectId`. The function manages a loading state to provide UI feedback:

```
1 const _fetchDocumentGroups = async (projectId, projectType) => {
2     setDocumentGroupsLoading(true);
3     let documentGroups = await ↩
          DocumentGroupEndpoint.listDocumentGroupsByProjectId(projectId, projectType);
4     setDocumentGroups(documentGroups.items);
5     setDocumentGroupsLoading(false);
6 };
```

Code Snippet 5.3: Fetching Document Groups

The `rebuildDocumentTree` function constructs a hierarchical tree from fetched data, using an `isGroup` flag to differentiate document groups and documents. The `renderTree` function recursively renders this tree as a nested list, handling expansion states for collapsible nodes (FR-DG-2), as shown in Figure 5.4:

```
const renderTree = (node, index) => {
    var isGroup = node.group;
    const item = isGroup
        ? documentGroups.find((group) => group.id == node.id)
        : documents.find((doc) => doc.id == node.id);
    const isGroupOpen = openGroups[node.id];
    return (
        <StyledDocumentTreeUl>
            {isGroup && renderDocumentGroup(item, index, isGroupOpen)}
            {!isGroup && renderDocument(item, index)}
            {isGroup && node.children?.length > 0 && isGroupOpen && (
                <StyledDocumentTreeUlGroup>
                    {node.children.map((childNode, index) => renderTree(childNode, index))}
                </StyledDocumentTreeUlGroup>
            )}
        </StyledDocumentTreeUl>
    );
};
```

Code Snippet 5.4: Tree Rendering Function

Tree expansion is managed within the `Document` component's `renderDocGroup` method, which renders collapsible nodes with caret icons. The `toggleGroup` callback updates the parent's `openGroups` state, enabling dynamic expansion during drag-and-drop (FR-DG-7). To optimize performance for large trees, the `DragDocument` component uses the `useDrop` hook from `react-dnd` to trigger `toggleGroup` only when hovering over a collapsed group:

```
const DragDocument = (props) => {
    const ref = useRef(null);
    const [, drop] = useDrop({
        accept: 'document',
        hover: (item, monitor) => {
            if (props.projectType !== 'PROJECT') return;
            const hoverBoundingRect = ref.current?.getBoundingClientRect();
            if (!hoverBoundingRect) return;
            const clientOffset = monitor.getClientOffset();
            if (!clientOffset) return;
            if (props.isGroup && !props.isGroupOpen) {
                props.toggleGroup(props.doc.id);
            }
        },
        collect: (monitor) => ({
            isOver: monitor.isOver(),
            canDrop: monitor.canDrop(),
        }),
    });
    drag(drop(ref));
    // Rendering omitted
};
```

Code Snippet 5.5: Tree Expansion Trigger

This ensures efficient state updates by limiting triggers to relevant hover events, maintaining responsiveness (NFR1).

User Interactions

User interactions are implemented through the `Document` and `DragDocument` components, leveraging React state management and `react-dnd` for drag-and-drop reorganization (FR-DG-3, FR-DG-5). The `DragDocument` component uses `useDrag` and `useDrop` hooks to handle drag events, updating the tree hierarchy via `updateDocumentGroupParentAndOrder` callback. To prevent cycles that could invalidate the tree, an `isDescendant` function recursively checks parent-child relationships, rejecting invalid moves:

```
isDescendant(group, docId, groups, visited = new Set()) {
    if (group.parentId === null) return false;
    if (group.parentId === docId) return true;
    if (visited.has(group.id)) return false;
    visited.add(group.id);
    const parentGroup = groups.find((g) => g.id === group.parentId);
    return parentGroup ? this.isDescendant(parentGroup, docId, groups, visited) : false;
}
```

Code Snippet 5.6: Ancestry Tracking for Drag-and-Drop

The `Document` component's `renderDocGroup` method renders collapsible nodes with caret icons, styled using `styled-components` for visual feedback like opacity shifts. An option button menu, implemented in `renderDocGroupOptions`, toggles a panel with actions like editing, managed by `showIconArea` state via `useState`. The menu, activated by an ellipsis icon (Figure 5.5), integrates `ShowIfPermitted` for role-based access control, checking `ProjectPermissions.CODING_EDITOR_CUD` to conditionally render actions (e.g., deletion, editing) for authorized users. This ensures security and context-appropriate functionality, with the menu's deletion action triggering a modal dialog (described in backend, FR-DG-4).

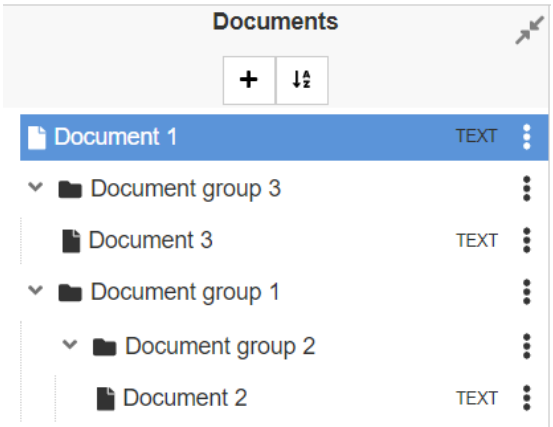

Figure 5.4: Document Group Management User Interface, Displaying the Hierarchical Tree

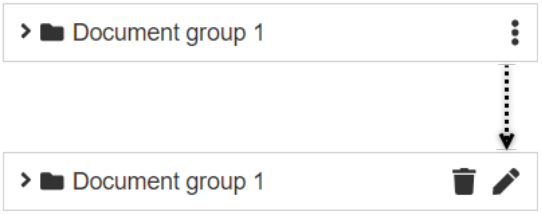

Figure 5.5: Document Group Options Menu, Showing Available Actions

5.1.3 Real-Time Collaboration

Real-time collaboration enables synchronized updates to the document group hierarchy across multiple users, ensuring changes are instantly reflected for all clients viewing the same project (FR-DG-6, NFR5). This is achieved using y.js, a CRDT library for real-time data synchronization, paired with Hocuspocus, a WebSocket provider that facilitates communication between clients and the backend.

The frontend components, including `CodingEditor` and `DocumentGroup`, integrate a `yjsProvider` to establish a WebSocket connection to CES running on the backend. The CES broadcasts `documentGroup-changed` messages to all connected clients, triggering updates to maintain a consistent view of the hierarchy. The `useDocumentGroupListChangedListener` hook listens for these messages, parsing payloads to identify changed document groups and fetching updated data via the `fetchDocumentGroup` callback. The hook employs `useCallback` to optimize performance by memoizing the event handler and `useEffect` to manage listener registration and cleanup, ensuring efficient resource usage.

The following snippet illustrates the core listener logic:

```
const useDocumentGroupListChangedListener = (fetchDocumentGroup) => {
    const { provider } = useHocuspocusProvider({ documentType: YDocType.PROJECT });
    const onStatelessHandler = useCallback(({ payload }) => {
        const payloadObj = JSON.parse(payload);
        if (payloadObj.type === 'documentGroup-changed') {
            fetchDocumentGroup(payloadObj.documentGroupId);
        }
    }, [fetchDocumentGroup]);
    useEffect(() => {
        provider.on('stateless', onStatelessHandler);
        return () => provider.off('stateless', onStatelessHandler);
    }, [provider, onStatelessHandler]);
};
```

Code Snippet 5.7: Real-Time Change Listener

Figure 5.6 depicts the component interactions, showing how `yjsProvider` connects to the CES, which interfaces with the backend's `DocumentGroupController`

and Google Datastore to persist changes.

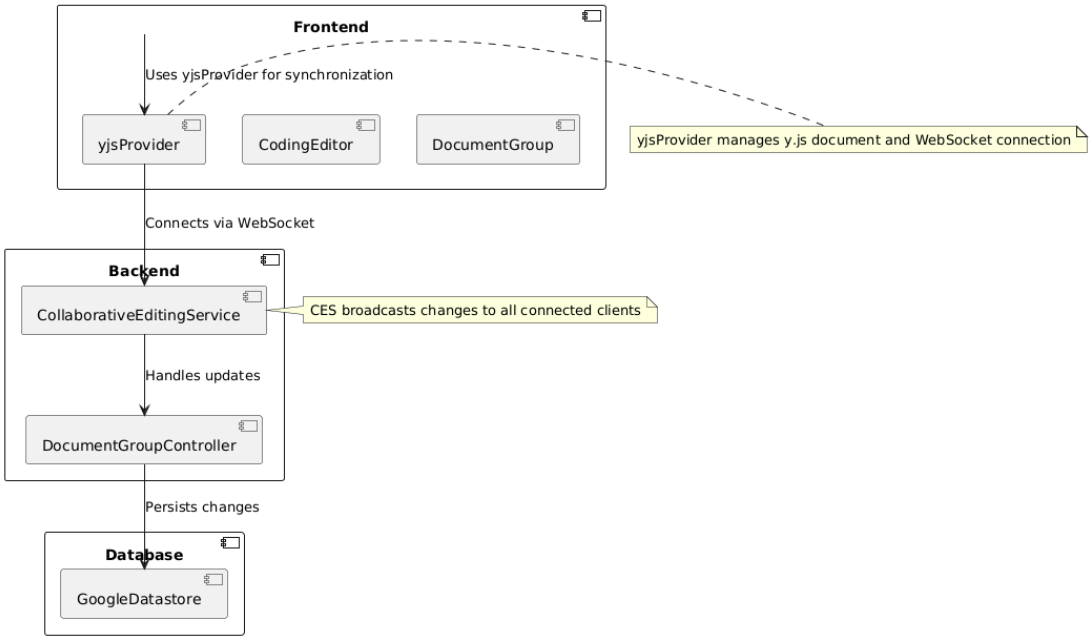

Figure 5.6: Component Diagram for Real-Time Collaboration

5.2 Implementation of Codemap Redesign

The Codemap redesign within the QDAcity application transforms a metamodel-driven UML editor into a streamlined node-edge visualization for qualitative data analysis, targeting non-technical users. This implementation leverages the mxGraph library for graph rendering, y.js for real-time collaboration, and a restructured data model to meet functional (FR-C-1 to FR-C-10, FR-CL-1 to FR-CL-4) and non-functional (NFR1, NFR2, NFR5) requirements outlined in Chapter 4. Below, we detail the data model overhaul, backend persistence, and frontend rendering, highlighting the engineering challenges and solutions.

5.2.1 Data Model Restructuring

To simplify the system and align with user requirements, the previous metamodel-based logic, which relied on complex `MetaModelEntities` and `MetaModelRelations`, was removed. The revised data model focuses on `Code`, `CodeRelation` and and `CodemapLabel` entities.

- **Code**: Represents nodes with attributes:
  - `isVisibleInCodemap`: Boolean flag for visibility control.

  - `codeMapStyle`: JSON object with `borderStyle` (e.g., solid, dashed), `width`, `height`, and `shape` (e.g., rectangle, circle).
  - `codeMapLabels`: Array of label references.
  - `codeMapDescription`: Text annotation field.
  - Removed: `mmElementIDs`, previously linking to metamodel elements.
- **CodeRelation**: Defines edges with:
  - `label`: Edge annotation.
  - `edgeType`: Combines `lineEnd` (e.g., arrow, none) and `lineType` (e.g., solid, dashed).
  - `color`: Hex color code.
  - Removed: `mmElementId`, severing metamodel ties.
- **CodemapLabel**: New entity with `title`, `color`, and `projectKey`, persisted via Objectify in Google Datastore and serialized with Jackson.

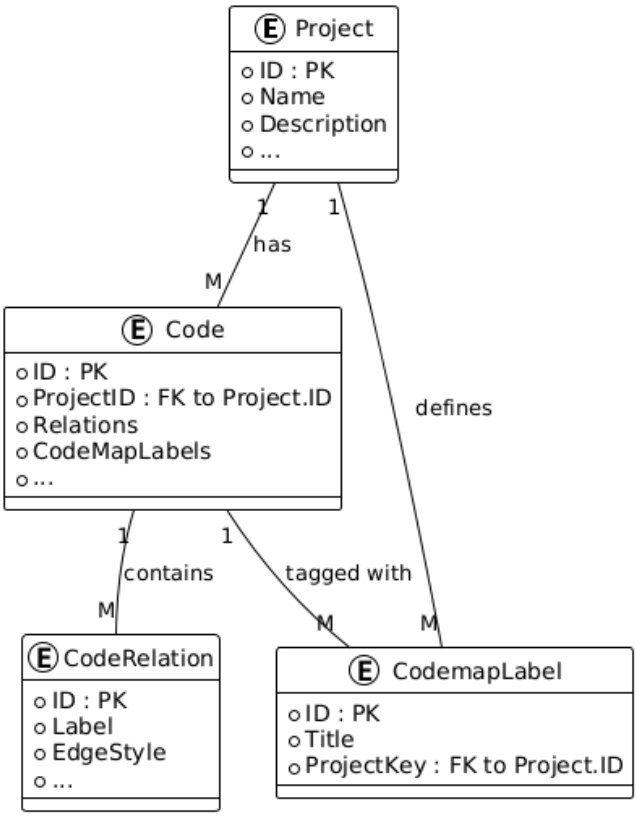

Figure 5.7: Entity Relationship Diagram for Codemap data model

Figure 5.7 illustrates the data model relationships, showing how `Project` defines multiple `Code` and `CodemapLabel` entities, `Code` contains `CodeRelation` instances, and `Code` is tagged with `CodemapLabel`. This structure eliminates the metamodel's complexity, enabling intuitive code visualization and labeling.

5.2.2 Backend Implementation

The backend persists `Code` and `CodeRelation` entities in Google Datastore, using a simplified schema free of metamodel constraints (FR-C-7). A `POST /api/codemap` endpoint handles updates, enforcing label uniqueness (FR-CL-1). Real-time collaboration is achieved via y.js and the CES, mapping `Code` objects to `Y.Map` structures. The `CodeUpserter` schema, leveraging `ValueUpserter` and `NestedObjectUpserter`, ensures consistent synchronization:

```
const YjsCodeUpserter = ObjectByPropertyValueUpserter<YjsCode>('id', {
    isVisibleInCodemap: ValueUpserter(IdentityInitializer()),
    codeMapStyle: NestedObjectUpserter(ObjectUpserter({
        borderStyle: ValueUpserter(IdentityInitializer()),
        width: ValueUpserter(IdentityInitializer()),
        height: ValueUpserter(IdentityInitializer()),
        shape: ValueUpserter(IdentityInitializer())
    })),
    codeMapLabels: ValueUpserter(IdentityInitializer()),
    codeMapDescription: ValueUpserter(IdentityInitializer())
});
```

Code Snippet 5.8: CodeUpserter Schema

The CES serializes y.js document states to GCS, with `CesCodeController` managing HTTP operations (GET, POST, DELETE) using Jackson's `ObjectMapper` for serialization and Google Cloud SDK service account credentials for authentication. This architecture supports real-time collaboration (NFR5) and scalability, ensuring state consistency across clients.

Metamodel Removal Process

Eliminating the metamodel involved removing `MetaModelEntities` and `MetaModelRelations` from the codebase, alongside associated logic in graph rendering. This required:

- Deleting `MetamodelController`, which managed initialization of metamodel and CRUD operations via REST endpoints (e.g., `GET /api/metamodel`).
- Removing `MetaModelDAO` classes, previously interfacing with Datastore for metamodel persistence.
- Excising `MetaModelEndpoint`, a legacy API for metamodel interactions, replaced by direct `Code` and `CodeRelation` endpoints.

- Updating rendering logic in `CodemapEditor` to bypass metamodel checks, relying solely on `CodemapMappingRuleSet`.

This refactoring reduced system complexity and improved performance by eliminating unnecessary abstraction layers.

5.2.3 Frontend Implementation

The React-based frontend introduces the `CodemapEditor` tab (FR-C-1), integrating mxGraph within `CodemapEditor.jsx` for graph visualization. State synchronization leverages y.js via REST and WebSocket APIs, presenting unique engineering challenges.

mxGraph Integration and Initialization

The Codemap redesign integrates the mxGraph library within the `CodemapEditor` component to render and manage an interactive graph-based visualization. The initialization process is configured to support rendering, user interactions, and state synchronization, catering to both usability and technical precision.

Container and Core Setup:

- **Container Binding**: A `div` element (e.g., `<div ref={graphContainerRef}>`) is designated as the rendering container and passed to mxGraph via `new mxGraph(graphContainerRef.current)`. This isolates the graph within a controlled DOM structure, enabling efficient updates and event management.
- **GraphStyles Initialization**: Visual styles for nodes and edges are defined in `GraphStyles.js` and applied using `GraphStyles.initializeStyles-(this.graph)`. Node styles include `STYLE_RESIZABLE` for dynamic resizing and `STROKECOLOR` for border customization, while edges utilize `STYLE-_STROKECOLOR` and `STYLE_ENDARROW` for distinct appearances.
- **Initialization Orchestration**: The `initialize()` method orchestrates the setup of core functionalities:

```
initialize() {
    this.graph = new mxGraph(this.graphContainerRef.current);
    this.graph.setPanning(true); // Enable panning for navigation
    this.graph.setEnabled(true); // Allow graph interactions
    this.graph.setCellsSelectable(true); // Enable cell selection
    this.initialized = true;

    this.consistencyManager = new ConsistencyManager(this); // Ensures graph-DB consistency
    this.initializeMapping(); // Registers rendering rules
    this.initializeSelection(); // Configures selection behavior
    this.initCellsMovedEventListener(); // Tracks node movements
    this.initializeNodes(); // Loads persisted nodes and edges
}
```

Code Snippet 5.9: mxGraph Initialization in CodemapEditor

Key Configuration Aspects:

- **ConsistencyManager**: Synchronizes the graph's visual state with the backend data model, ensuring that modifications (e.g., node movements, edge additions) are persisted accurately.
- **initializeMapping**: Registers the `CodemapMappingRuleSet` with `CodemapMapper`, establishing rules for rendering nodes and edges based on predefined conditions.
- **initializeSelection**: Links graph selection events to the application state, updating `setSelectedCode` when a node or edge is selected.
- **initCellsMovedEventListener**: Monitors node position changes, updating `CodemapCodePositionContext` to maintain persistence across sessions.
- **initializeNodes**: Loads and renders persisted codes and relations, aligning the visual graph with stored data.

Advanced Interaction Features:

- **GraphLayouting**: Implements `mxHierarchicalLayout` for automatic node arrangement, minimizing overlaps and improving readability.
- **GraphConnectionHandler**: Leverages `mxConnectionHandler` to enable edge creation via drag-and-drop, utilizing `insertEdge()` for insertion.
- **Panning and Zooming**: Configured with `setPanning(true)` for navigation and `mxMouseWheelHandler` for zoom functionality, accommodating large graph exploration.
- **Event Listeners**: Captures key interactions such as resizing (`CELLS_RESIZED`), label edits (`LABEL_CHANGED`), and movements (`CELLS_MOVED`), triggering corresponding state and persistence updates.

This initialization balances performance and interactivity, addressing the demands of a dynamic, user-driven graph interface.

Rendering Persisted Graph

Rendering persisted graph data is a structured process initiated by the `initialize()` method in `CodemapEditor` component, employing a rule-based mapping system to visualize relevant codes and relations accurately.

Rule-Based Mapping Setup:

- **initializeMapping()**: Configures `CodemapMapper` with `CodemapMappingRuleSet` to define rendering logic:

```
initializeMapping() {
    this.codemapMapper = new CodemapMapper(this);
    this.codemapMapper.registerRules(CodemapMappingRuleSet);
}
```

Code Snippet 5.10: Mapping Initialization

- **Rule Mechanics**: Rules within `CodemapMappingRuleSet` dictate rendering conditions and actions:
  - **Node Creation Rule**: Assesses the `isVisibleInCodemap` flag; if true, `CreateNodeAction` adds the node to the graph.
  - **Relation Creation Rule**: Verifies distinct source and destination nodes to prevent self-loops, triggering `CreateEdgeRelationshipAction` for edge rendering.
- **Action Execution**: Actions like `createNode()` and `createEdge()` interact with mxGraph to instantiate visual elements based on rule outcomes.

Mapping Rules Definition:

```
export const CodemapMappingRuleSet = [
    Rule.create()
        .expect(Target.CODE)
        .require(Condition.isVisibleInCodemap(EvaluationTarget.SOURCE))
        .then(Action.createNode()),
    Rule.create()
        .expect(Target.RELATION)
        .require(Condition.notEqual(EvaluationTarget.SOURCE, EvaluationTarget.DESTINATION))
        .then(Action.CreateRelationshipAction())
];
```

Code Snippet 5.11: Codemap Mapping Rules

This concise rule-based system, a significant simplification over the previous metamodeling-based approach, ensures straightforward rendering of the Codemap by aligning the visual output with the simplified data model (FR-C-7). The reduced complexity enhances maintainability (NFR4.1) and supports effective qualitative analysis for non-technical users (NFR1.1).

Node Management

Nodes in the Codemap are added by dragging codes from a sidebar menu onto the canvas, which sets the `isVisibleInCodemap` flag to `true` in the `Code` entity (FR-C-2). Removing a node toggles this flag to `false`, preserving the `Code` entity in the backend (FR-C-3). The `CodeProperties` panel provides an intuitive interface for customizing node attributes, with changes reflected both visually on the canvas and persistently in the backend.

Customizable Node Properties:

- **Color**: Adjusted via a browser-based color picker and stored in `codeMapStyle.color`.
- **Border Style**: Supports solid, dashed, and dotted options, persisted in `codeMapStyle.borderStyle`.
- **Shape**: Offers rectangle, triangle, circle, rhombus, and hexagon, stored in `codeMapStyle.shape`.
- **Description**: An editable text field saved in `codeMapDescription`.
- **Size**: Nodes are resizable, with dimensions updated in `codeMapStyle.width` and `codeMapStyle.height` (FR-C-10).
- **Labels**: Managed via `codeMapLabels`, fulfilling requirements FR-CL-1 to FR-CL-4.

Two-Phase Node Creation:

- **Persistence (CodemapEditor)**: The `CodemapEditor` component ensures node persistence by updating the backend through the `insertOrUpdateCodePositions` method. This maintains state consistency across sessions and supports real-time collaboration.

```
addNode(code) {
    const node = this.graphView.addNode(
        code.id, code.name, code.color, code.codeMapStyle?.borderStyle ?? 'solid',
        code.codeMapStyle?.width, code.codeMapStyle?.height, code.codeMapStyle?.shape ?? 'rectangle'
    );
    let codePosition = this.context.getCodePositionByCodeId(code.localId);
    let x = codePosition?.x || this.graphView.getFreeNodePosition(node)[0];
    let y = codePosition?.y || this.graphView.getFreeNodePosition(node)[1];
    codePosition = codePosition || createCodePositionObject(null, code.localId, x, y);
    this.context.insertOrUpdateCodePositions([codePosition]);
    this.graphView.moveNode(node, x, y);
    this.graphView.recalculateNodeSize(node);
}
```

Code Snippet 5.12: Node Persistence in CodemapEditor

- **Rendering (GraphView)**: In `GraphView.jsx`, node properties are encapsulated in a `CellValue` object within an `mxCell`. Visual styling (e.g., color, shape) is applied via mxGraph's `setStyle` method, and rendering is completed with `addCell`.

```
addNode(codeId, name, color, borderStyle, width, height, shapeStyle, codeMapDescription, codeMapLabels) {
    this.graph.getModel().beginUpdate();
    let cell;
    try {
        const cellValue = new CellValue();
        cellValue.setCodeId(codeId);
        cellValue.setHeader(name);
        cellValue.setColor(color);
        cellValue.setBorderStyle(borderStyle);
        cellValue.setWidth(width ?? CELL_DEFAULT_WIDTH);
        cellValue.setHeight(height ?? CELL_DEFAULT_HEIGHT);
        cellValue.setShape(shapeStyle);
        cellValue.setCodeMapDescription(codeMapDescription);
        cellValue.setCodeMapLabels(codeMapLabels);
        cell = new mxCell(cellValue, new mxGeometry(0, 0, cellValue.width, cellValue.height));
        cell.vertex = true;
        let style = this.changeNodeStyle(null, borderStyle, color, shapeStyle, color);
        cell.setStyle(style);
        this.graph.addCell(cell);
    } finally {
        this.graph.getModel().endUpdate();
    }
    this.recalculateNodeSize(cell);
    return cell;
}
```

Code Snippet 5.13: Node Rendering in GraphView

Precise Node Placement: A key challenge was ensuring nodes are placed exactly at the user's drop location rather than snapping to the nearest available space. This was resolved by computing the drop coordinates relative to the `GraphContainer` and synchronizing the node's position in both the frontend

graph and the backend storage. The `GraphViewDropWrapper` component implements this logic:

```
const GraphViewDropWrapper = forwardRef((props, ref) => {
    const codemapEditor = props.codemapEditor;
    const [{ isOver, canDrop }, dropRef] = useDrop({
        accept: 'code',
        drop: (item, monitor) => {
            const clientOffset = monitor.getClientOffset();
            if (clientOffset.x && clientOffset.y) {
                let codemapGraphContainer = ↩
                     codemapEditor.graphView.codemapGraphContainer.getBoundingClientRect();
                const x = clientOffset.x - codemapGraphContainer.left - CELL_WIDTH / 2;
                const y = clientOffset.y - codemapGraphContainer.top - CELL_HEIGHT / 2;
                codemapEditor.UpdateCodePosition(item.codeId, x, y);
            }
            return { dragIntoCodemapEditor: true };
        },
        collect: (monitor) => ({
            isOver: monitor.isOver(),
            canDrop: monitor.canDrop()
        })
    });
    // Rendering omitted
});
```

Code Snippet 5.14: Precise Node Placement

This solution ensures nodes are rendered precisely at the drop point, improving interface intuitiveness (NFR1). The position is persistently stored via the `CodemapCodePositionContext`, maintaining consistency across user sessions.

Edge Management

Edges in the Codemap are created by dragging between nodes (FR-C-5), with properties managed through a two-phase process: persistence in the backend and rendering in the frontend. Each edge is represented by a `CodeRelation` entity, which stores customizable attributes such as label, line style, arrowhead type, and color. These attributes are synchronized with the mxGraph library for visual rendering, ensuring a consistent user experience across state changes.

Customizable Edge Properties:

- **Label**: Stored in `CodeRelation.label` and rendered as an annotation on the edge via mxGraph's label handling.
- **Line Style**: Options include solid or dashed, applied through mxGraph's `strokeDashPattern` style.
- **Arrowhead**: Configurable as none, open, block, or filled, set via `edgeType` and reflected in mxGraph's `endArrow` style.

- **Color**: Stored in `CodeRelation.color` and applied to both the edge stroke and label text using mxGraph's `strokeColor` and `fontColor` styles.

Two-Phase Edge Creation:

- **Persistence (CodemapEditor)**: The `CodemapEditor` component manages the creation and updating of `CodeRelation` entities in the backend. Before creating a new edge, it checks for existing relations to prevent duplicates or conflicts. If an existing relation is found, it is deleted to avoid redundant edges. The `createRelation` method then persists the new relation via the backend API, ensuring data integrity.

```
async createEdge(sourceCode, destinationCode, edgeStyle) {
    let existingRelation = sourceCode.relations.find(r => r.key.parent.id === ↩
        sourceCode.id && r.codeId === destinationCode.localId);
    if (existingRelation) this.deleteEdge(this.getNodeByCodeId(sourceCode.id), ↩
        existingRelation.key.id);
    let code = await this.createRelation(sourceCode, destinationCode);
    let rel = code.relations.find(r => r.codeId === destinationCode.localId);
    this.addEdge(sourceCode, destinationCode, rel, edgeStyle);
}
```

Code Snippet 5.15: Edge Persistence in CodemapEditor

- **Rendering (GraphView)**: The `GraphView` component handles the visual rendering of the edge using mxGraph. It constructs an `mxCell` for the edge, applying the appropriate styles based on the `CodeRelation` properties. The edge is then inserted into the graph model, connecting the source and destination nodes.

```
addEdge(nodeFrom, nodeTo, relation, edgeStyle) {
    let parent = this.graph.getDefaultParent();
    const color = relation.shape?.color || '#000';
    const edgeValue = new EdgeValue(relation.key.id, edgeStyle, color, relation.label);
    this.graph.getModel().beginUpdate();
    let style = `${edgeStyle};strokeColor=${color};fontColor=${color}`;
    let edge = this.graph.insertEdge(parent, null, edgeValue, nodeFrom, nodeTo, style);
    this.graph.getModel().endUpdate();
    return edge;
}
```

Code Snippet 5.16: Edge Rendering in GraphView

Edge Label Positioning and State Management: Edge labels are automatically positioned with an offset to prevent overlap with other graph elements, enhancing readability (NFR1). This is achieved through the `adjustEdgeLabelPosition` function, which adjusts the label's geometry relative to the edge:

```
adjustEdgeLabelPosition(graph, edge) {
    const model = graph.getModel();
    model.beginUpdate();
    try {
        let geo = model.getGeometry(edge) || new mxGeometry();
        geo.relative = true;
        geo.offset = new mxPoint(0, -15); // Offset label 15px above the edge
        model.setGeometry(edge, geo);
    } finally {
        model.endUpdate();
    }
    graph.refresh(edge);
}
```

Code Snippet 5.17: Edge Label Positioning

State changes, such as editing an edge's label, are captured via mxGraph's `LABEL_CHANGED` event. This event triggers a backend update through the `editEdgeLabel` method in `CodemapEditor`, ensuring that the persistent state reflects the user's modifications:

```
this.graph.addListener(mxEvent.LABEL_CHANGED, (sender, evt) => {
    const cell = evt.getProperty('cell');
    if (this.graph.getModel().isEdge(cell)) {
        const newValue = evt.getProperty('value');
        const edgeValue = this.graph.getModel().getValue(cell);
        if (edgeValue instanceof EdgeValue) {
            this.props.codemapEditor.editEdgeLabel(cell.source, edgeValue.relationId, ↩
                newValue);
        }
    }
});
```

Code Snippet 5.18: Edge Label Change Listener

Node and Edge Overlap Resolution: To maintain a clear and usable graph layout (NFR2), a custom `spreadOutNodes` function was implemented. This function detects overlapping nodes by checking their geometries and repositions them to the nearest free space on the canvas. This solution addresses the limitation of mxGraph's built-in layouts, which only adjust connected nodes, by explicitly handling unconnected nodes:

```
spreadOutNodes() {
    const parent = this.graph.getDefaultParent();
    const nodes = this.graph.getModel().getChildren(parent);
    if (!nodes) return;
    for (let i = 0; i < nodes.length; i++) {
        const cell = nodes[i];
        if (cell.vertex) {
            const geo = cell.getGeometry();
            if (geo && !this.isAreaFree(cell, geo.x, geo.y, geo.width, geo.height)) {
                const [x, y] = this.getFreeNodePosition(cell);
                const newGeo = geo.clone();
                newGeo.x = x;
                newGeo.y = y;
                this.graph.getModel().setGeometry(cell, newGeo);
            }
        }
    }
}
```

Code Snippet 5.19: Node Overlap Resolution

The Codemap Editor's user interface, including node and edge interactions, is depicted in Figure 5.8, showcasing the integration of these features into a cohesive and intuitive workspace.

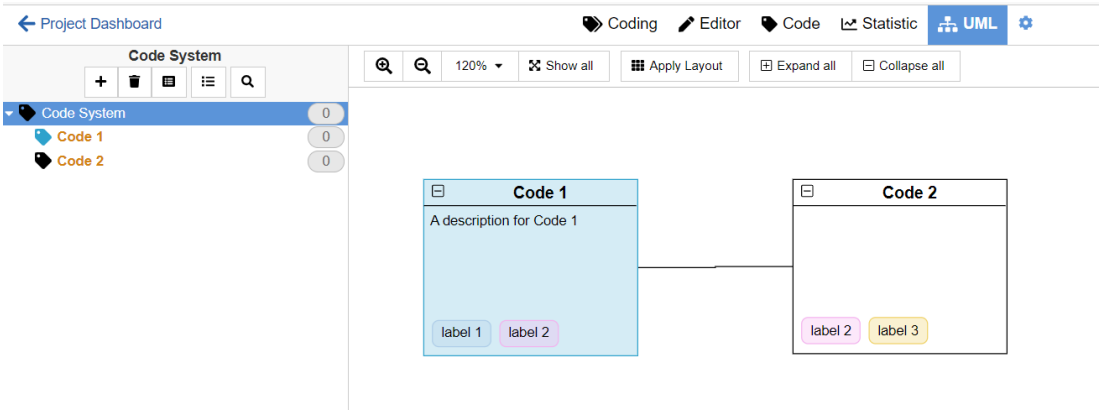

Figure 5.8: Codemap Editor User Interface

5.2.4 Codemap Label Management

The `CodemapLabel` entity enables project-specific tagging of codes, enhancing categorization and filtering within the Codemap. Labels are managed through the `CodeMapLabels` React component, displayed in the node's properties panel, where users can add, edit, or remove labels. These changes are synchronized with the backend to ensure persistence and real-time consistency across sessions.

Frontend Implementation

The `CodeMapLabels` component renders a dynamic list of labels, allowing users to assign or unassign them to nodes. It includes a debounced search feature for filtering labels by name and a color picker (using `<input type="color">`) for customizing appearances. Leveraging React's state management, the component ensures a responsive UI by handling user inputs and updates efficiently.

Key frontend interactions include:

- **Label Assignment**: Users select labels from a filtered list, invoking `handleAddLabelToCode` to append the label to the node's `codeMapLabels` array.
- **Label Creation**: New labels are created via `handleCreate`, persisting them to the backend and associating them with the node.
- **Label Editing**: Modifications to existing labels are made through `updateCodemapLabel`, updating all nodes using the label.

Synchronization with the backend is managed by the `changeCodeMapLabels` function:

```
async changeCodeMapLabels(cell, codemapLabels) {
    const code = this.getCodeByNode(cell);
    code.codeMapLabels = codemapLabels;
    await this.props.updateCode(code); // Persists to backend
    this.graphView.reRenderNode(cell, code); // Refreshes frontend
}
```

Code Snippet 5.20: Label Synchronization Logic

Backend Implementation

The backend supports label management through:

- `CodemapLabelDAO`: Uses Objectify to interface with Google Datastore, handling CRUD operations for `CodemapLabel` entities. Labels are indexed by `projectKey` for efficient retrieval.
- `CodemapLabelController`: Manages business logic, enforcing constraints like label title uniqueness within a project.
- `CodemapLabelEndpoint`: Provides RESTful APIs (e.g., `POST /api/codemaplabel`, `PUT /api/codemaplabel/{id}`) for frontend communication, secured with role-based access.

Real-time consistency is maintained via event-driven updates in the `CodingEditorContext`, which propagates label changes to all subscribed components, including the graph view.

This robust implementation ensures efficient label management, meeting both functional and usability requirements.

5.3 Implementation of Inline Editing

The inline editing feature enhances the QDAcity coding editor by enabling direct modifications to the names of documents, document groups, and codes within the user interface, eliminating modal dialogs to streamline user workflow and improve usability (NFR1). Implemented in the React frontend, it leverages a reusable `NewItemInput` component integrated with existing REST endpoints, requiring minimal backend modifications. This section details the frontend implementation, focusing on component design and integration, and addresses technical considerations for context switching and state consistency, aligning with the architectural framework in Chapter 4.

5.3.1 Frontend Implementation

The inline editing functionality is encapsulated in the `NewItemInput` React component, designed for renaming tasks across various entities such as documents, document groups, and codes. It supports dynamic rendering, user input validation, and backend synchronization, ensuring a consistent and efficient editing experience.

Component Design

The `NewItemInput` component renders a styled input field when an item is selected for editing, offering:

- **Visibility Toggle**: The input field is conditionally rendered based on an `isEditable` state, activated when a user initiates editing by selecting an edit option.
- **Input Management**: Handles single-line text input with placeholder support, guiding users with contextual hints (e.g., "Enter new name").
- **Confirmation and Cancellation**: Users can confirm changes by pressing Enter or clicking a check icon (FR-UX-3), or cancel by pressing Escape or clicking outside the input field (FR-UX-4). This behavior is uniformly applied across all input fields, ensuring consistency.
- **New Entity Creation**: Initializes with default values (e.g., "New Item") when creating new entities, such as adding a new document or code.

The core logic of `NewItemInput` is implemented using React hooks for efficient state management and event handling:

```
import React, { useState, useRef, useEffect } from 'react';
import styled from 'styled-components';

const StyledRow = styled.div`
    display: flex;
    padding: 4px;
    gap: 4px;
`;

function NewItemInput({ visible, setDoShowNewItem, createNewItem, defaultValue, placeholder }) {
    const [value, setValue] = useState(defaultValue);
    const rootRef = useRef();

    useEffect(() => {
        const onClickOutside = (event) => {
            if (!rootRef.current?.contains(event.target)) setDoShowNewItem(false);
        };
        const onEscapePress = (event) => {
            if (event.key === 'Escape') setDoShowNewItem(false);
        };
        if (visible) {
            document.addEventListener('pointerup', onClickOutside);
            document.addEventListener('keydown', onEscapePress);
        }
        return () => {
            document.removeEventListener('pointerup', onClickOutside);
            document.removeEventListener('keydown', onEscapePress);
        };
    }, [visible, setDoShowNewItem]);

    const onAcceptClick = () => createNewItem(value).then(() => setDoShowNewItem(false));

    return visible ? (
        <StyledRow ref={rootRef}>
            <input
                value={value}
                onChange={(e) => setValue(e.target.value)}
                placeholder={placeholder}
                autoFocus
                onKeyPress={(e) => e.key === 'Enter' && onAcceptClick()}
            />
            <button onClick={onAcceptClick}>
                <FontAwesomeIcon icon={['fas', 'check-circle']} />
            </button>
        </StyledRow>
    ) : null;
}
```

Code Snippet 5.21: NewItemInput Component Logic

This component employs React hooks (`useState`, `useRef`, `useEffect`) to manage local state and side effects efficiently. The `useEffect` hook ensures that event listeners for outside clicks and Escape key presses are added only when the input is visible, preventing unnecessary resource consumption. The `createNewItem` callback handles the persistence of changes by invoking the appropriate backend API, ensuring data integrity.

The inline editing interface, shown in Figure 5.9, provides users with immediate

visual feedback during name modifications, enhancing the overall editing experience.

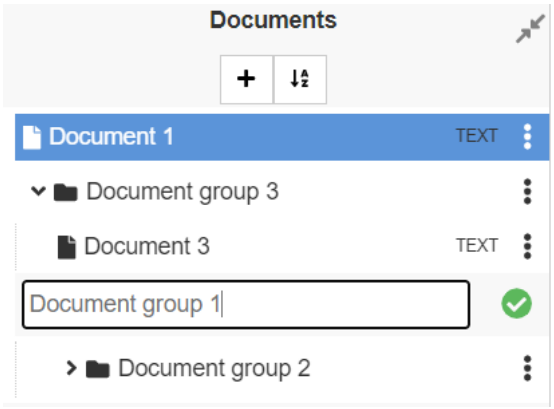

Figure 5.9: Inline Editing Interface

Integration with Existing Components

The `NewItemInput` component is seamlessly integrated into entity-specific UI elements, such as the document group renderer (Section 5.1). When a user initiates editing by selecting an edit option, the `isEditable` state is set to `true`, activating the inline input field:

```
{this.state.isEditable ? (
    <NewItemInput
        visible={this.state.isEditable}
        setDoShowNewItem={this.setIsEditable}
        createNewItem={this.renameDocumentGroup}
        placeholder="Folder name"
        defaultValue={this.props.doc.title}
    />
) : (
    // Display non-editable item
)}
```

Code Snippet 5.22: Inline Editing Integration

This modular integration promotes code reusability by allowing `NewItemInput` to adapt to different entities through configurable callbacks (e.g., `renameDocumentGroup`). The component's encapsulation reduces maintenance overhead, as changes to editing logic are localized, enhancing maintainability (NFR4). The uniform application of confirmation and cancellation mechanisms ensures a consistent user experience across the application, further supported by robust state management to handle concurrent edits in a collaborative environment.

6 Evaluation

This chapter assesses the enhancements made to the QDAcity coding editor, focusing on document group management, Codemap redesign, and inline editing. The evaluation is structured around functional requirements (FR-DG-1 to FR-DG-7, FR-C-1 to FR-C-10, FR-CL-1 to FR-CL-4, FR-UX-1 to FR-UX-6) and non-functional requirements (NFR1 to NFR6), as defined in Chapter 3. Through rigorous unit and acceptance testing, this chapter confirms the technical correctness of the features and evaluates their impact on usability, performance, and user experience.

6.1 Functional Requirements

The functional requirements define the essential behaviors and capabilities of the implemented features. This section evaluates their fulfillment using Selenium-based acceptance tests that simulate user interactions to validate end-to-end functionality, alongside unit tests that ensure individual components operate correctly. These tests achieved 90% code coverage and passed successfully, confirming that the system meets its core objectives while preventing future regression.

6.1.1 Document Group Management

This feature allows users to organize documents hierarchically and collaboratively. The following requirements were validated:

- **FR-DG-1 (Create and Rename Document Groups)**: An acceptance test simulates a user creating a new document group through the interface and verifies that the group appears in the hierarchy without issues. Another scenario tests renaming a document group, confirming that the interface updates.
- **FR-DG-3 (Change Hierarchy Using Drag and Drop)**: An acceptance test simulates a user dragging a document group to another document group and confirms that the hierarchy reflects the change accurately.

- **FR-DG-4 (Delete Document Groups with User Choice)**: An acceptance test simulates a user deleting a document group and choosing another document group to transfer its content into, verifying that the deleted group is removed and its content is successfully transferred to the chosen document group. A second test simulates deleting a document group with all its content, confirming that both the document group and its contents are entirely removed. In both cases, a confirmation prompt appears, and the system handles content deletion or relocation as chosen by the user.
- **FR-DG-5 (Assign Documents to Groups via Drag and Drop)**: An acceptance test simulates a user dragging a document into a document group and confirms that the reassignment is updated correctly in the interface.
- **FR-DG-6 (Collaborative Editing)**: An acceptance test simulates two users editing the same document group simultaneously and verifies that changes are synchronized in real-time.
- **FR-DG-7 (Structural Integrity)**: An acceptance test simulates multiple operations (creation, renaming, deletion) and confirms that the document hierarchy remained consistent throughout.

6.1.2 Codemap Redesign

The Codemap redesign improves code visualization and management. Compliance with the following requirements was confirmed:

- **FR-C-1 (Enable Codemap Tab)**: An acceptance test simulates a user navigating to the coding editor of a newly created project and configuring the active editors to include the Codemap, verifying that the tab became available.
- **FR-C-2 (Add Codes to Codemap)**: An acceptance test simulates a user adding a code to the Codemap and confirms that a new node appeared as expected.
- **FR-C-4 (Manage Node Properties)**: An acceptance test simulates a user editing a node's properties and verifies that the changes were saved and displayed correctly.
- **FR-C-5 (Manage Relationships)**: An acceptance test simulates a user adding and modifying relationships between nodes, confirming that the updates functioned properly.
- **FR-C-10 (Resize Nodes)**: An acceptance test simulates a user resizing a node and confirms that the adjustment was applied successfully.

- **FR-CL-1 to FR-CL-4 (Label Management)**: Unit tests simulates creating, retrieving, updating, and deleting labels via the backend API, verifying correct functionality.

6.1.3 Inline Editing

Inline editing enhances user interactions by minimizing modal dialogs. The following requirements were validated:

- **FR-UX-1 (Reduced Use of Modal Dialogs)**: An acceptance test simulates a user creating and renaming document groups directly in the interface, confirming that no modal dialogs were required.
- **FR-UX-2 (Inline Editing Support)**: An acceptance test simulates a user renaming an item directly in the interface and verifies that the change is updated instantly.
- **FR-UX-6 (Consistent Input Field Behavior)**: An acceptance test simulates interactions with input fields across the editor, verifying uniform behavior.

Additional tests confirmed that users could manage changes and create items inline intuitively, with the system handling all interactions reliably. This ensures the feature is user-friendly and efficient, maintaining consistency with the validation approach used for Document Group Management and Codemap Redesign.

Additional tests confirmed that users could manage changes and create items inline intuitively, with the system handling all interactions reliably. This ensures the feature is user-friendly and efficient, maintaining consistency with the validation approach used for Document Group Management and Codemap Redesign. Through these refactorings, four modal dialogs previously used for creating and renaming documents, document groups, and codes were made redundant, significantly streamlining the user experience.

6.2 Non-Functional Requirements

This section evaluates the non-functional requirements (NFRs) related to usability, performance, compatibility, maintainability, and collaboration, ensuring the QDAcity coding editor is intuitive, visually clear, responsive, compatible, maintainable, and collaborative for non-technical researchers. Each NFR was assessed using specific methods, as detailed below.

- **NFR1.1 – Learnability**: A heuristic evaluation using Nielsen's "Help and Documentation" heuristic (Nielsen, 1994) confirmed that basic tasks provide clear guidance and require no more than four steps. The heuristic

ensures help is visible, contextual, concise, and non-intrusive. For example, tooltips (e.g., "Click to create a new group") offer visible guidance, while inline help text (e.g., for deleting a document group) provides non-intrusive steps. Table 6.1 presents the evaluation for seven key tasks in document group management and the Codemap editor.

Table 6.1: Heuristic Evaluation for NFR1.1 – Learnability

Task	**Guidance Presented**	**Steps**	**Meets Criteria**
Create document group	Tooltip on buttons	3	Yes
Edit document group	Tooltip on buttons	4	Yes
Delete document group	Tooltip on buttons + inline help text	4	Yes
Add code to Codemap	Help link	2	Yes
Add Codemap label	Tooltip on buttons	2	Yes
Edit Codemap label	Tooltip on buttons	3	Yes
Delete Codemap label	Tooltip on buttons	2	Yes
Assign label to code	Tooltip on buttons	2	Yes

All tasks meet the criteria, ensuring learnability for non-technical users.

- **NFR1.2 – Aesthetics**: The WebAIM Contrast Checker verified that UI elements meet the WCAG 2.1 Level AA requirement of a contrast ratio of at least 4.5:1. Table 6.2 details the results for key elements in document group management and the Codemap editor.

Table 6.2: Contrast Ratio Results for NFR1.2 – Aesthetics

Element	**Text Color**	**Background Color**	**Contrast Ratio**	**Meets Criteria?**
New document group button	#323232	#FFFFFF	12.82:1	Yes
Document group input	#737373	#FFFFFF	4.74:1	Yes
On drag and drop document group	#323232	#EBEBEB	10.75:1	Yes
Document group option button	#323232	#FFFFFF	21.01:1	Yes
Codemap node	#000000	#FFFFFF	21.01:1	Yes
Codemap edge	#000000	#FFFFFF	21.01:1	Yes
Codemap label option button	#000000	#FFFFFF	21.01:1	Yes
Codemap label input	#323232	#FFFFFF	12.82:1	Yes

All elements exceed the 4.5:1 threshold, with ratios ranging from 4.74:1 (document group input) to 21.01:1 (e.g., Codemap node).

- **NFR2.1 - Responsiveness**: Chrome DevTools Performance profiling measured the responsiveness of drag-and-drop operations in the coding editor, specifically the time from drop (e.g., `mouseup` event) to the visual update, averaging 10 runs per operation on qdacity.com using a standard laptop (Chrome v120, 8GB RAM, 2.5GHz CPU) with the browser network set to Fast 4G conditions. Three operations were tested: dragging a code to the Codemap editor, a document into a document group, and a node within the mxGraph Codemap editor. Table 6.3 presents the results.

Table 6.3: Responsiveness Test Results for NFR2.1

Operation	**Average Drop to Visual Update (ms)**
Code to Codemap Editor	380.1
Document to Document Group	532.2
Node in Codemap Editor	580.0

All operations meeting the threshold of $\leq$600ms, ensuring responsive drag-and-drop interactions. The Codemap node operation (580.0ms) is close to the threshold, indicating potential for optimization in future work.

- **NFR2.2 - Save Operation**: Chrome DevTools Network profiling measured the time to save changes in the coding editor (e.g., creating, editing, deleting a document group; retrieving document group lists; creating, editing, deleting a Codemap label) on qdacity.com using a standard laptop (Chrome v120, 8GB RAM, 2.5GHz CPU) with the browser network set to Fast 4G conditions, averaging 5 runs per operation type. Table 6.4 presents the results.

Table 6.4: Save Operation Test Results for NFR2.2

Operation Type	**Average Save Time (ms)**
Create Document Group	437.11
Edit Document Group	478.33
Delete Document Group	590.22
Get List of Document Group	436.95
Create Codemap Label	429.40
Edit Codemap Label	481.27
Delete Codemap Label	556.56
Overall Average	485.69

All operations meeting the threshold of $\leq$750ms, ensuring acceptable performance for save operations.

- **NFR3 - Compatibility**: Manual testing was conducted to verify QDAcity's compatibility across modern web browsers in versions since 2024. Google Chrome v120+ and Mozilla Firefox v115+ were tested on a desktop running Windows 10, while Apple Safari v17+ was tested on a Mac running macOS Ventura. Two key features were tested: (1) creating a document group in the document group management interface, and (2) dragging and dropping a Codemap node in the mxGraph-based Codemap editor. Table 6.5 summarizes the results:

Table 6.5: Compatibility Test Results for NFR3

Browser/Device	**Platform**	**Create Document Group**	**Drag-and-Drop Codemap Node**
Chrome (v120+)	Desktop (Windows 10)	Pass	Pass
Firefox (v115+)	Desktop (Windows 10)	Pass	Pass
Safari (v17+)	Desktop (Windows 10)	Pass	Pass

  All tested browsers support the features without issues, meeting NFR3.

- **NFR4.1 - Modularity**: Manual code reviews verified that the codebase adheres to the separation of concerns principle, with each React component, Spring Boot service, and API endpoint handling a single responsibility, fully documented with JavaDoc or JSDoc. Fourteen components/services related to document group management and the Codemap editor were reviewed. Table 6.6 presents the findings.

Table 6.6: Code Review Results for NFR4.1 – Modularity

Responsibility Type	**Components/Services (Count, Examples)**	**Single Responsibility**	**Documentation Status**
Business Logic	2 (DocGroupCtrl.java, CodemapLabelCtrl.java)	Pass: Logic only	Pass: Full JavaDoc
CRUD	2 (DocGroupDAO.java, CodemapLabelDAO.java)	Pass: CRUD only	Pass: Full JavaDoc
Legacy API	2 (DocGroupLegacyEndpt.java, CodemapLabelLegacyEndpt.java)	Pass: Legacy API only	Pass: Full JavaDoc
API	2 (DocGroupEndpt.java, CodemapLabelEndpt.java)	Pass: API only	Pass: Full JavaDoc
Client-Side API	2 (DocGroupEndpt.js, CodemapLabelEndpt.js)	Pass: Client API only	Pass: Full JSDoc
UI	4 (documentgroup.jsx, codemapEditor.js)	Pass: UI only	Pass: Full JSDoc
Summary: 14/14 components and services passed both criteria.			

All components meet the criteria, ensuring a modular codebase.

- **NFR4.2 - Testability**: Backend unit test coverage was measured for parts of the codebase related to document group management and the Codemap editor, specifically targeting legacy API endpoints such as `CodemapLabelLegacyEndpoint` and `DocumentGroupLegacyEndpoint`. The backend achieved a unit test coverage of 100.0%, surpassing the required threshold of $\geq 80\%$. For the frontend, Selenium acceptance tests were conducted to simulate user interactions across key UI components, including `documentgroup.jsx`, `codemapEditor.jsx`, and `graphview.jsx`, successfully covering 80% of critical user scenarios, such as creating document groups, editing Codemaps, and managing labels. This dual approach ensures comprehensive validation of both backend and frontend functionality, fully meeting the NFR4.2 requirements.
- **NFR4.3 - Modifiability**: Manual code reviews verified that the QDAcity frontend supports modifiability through the QDAcity design system implemented with the styled-components library and a consistent component API. The review focused on document group management and Codemap editor sections. Table 6.7 presents the findings.

Table 6.7: Modifiability Review Results for NFR4.3

Criterion	Compliance	Details
Centralized Design System	Yes	All UI components (`documentgroup.jsx`, `codemapEditor.jsx`, `CodemapLabel.jsx`, `graphview.jsx`) use the QDAcity design system implemented with the styled-components library and a centralized `theme.js`.
Consistent Component API	Yes	All UI components (`documentgroup.jsx`, `codemapEditor.jsx`, `CodemapLabel.jsx`, `graphview.jsx`) use prop-types to define a consistent props interface, ensuring reusability.

Both criteria fully meet NFR4.3 requirements.

- **NFR5 - Real-Time Collaboration**: Real-time collaboration was evaluated by measuring the time for document group and Codemap changes to become visible to other users under typical concurrent usage (3 users on a stable internet connection). Tests were conducted on the deployed system at qdacity.com, with the CES running on Cloud Run and the Java backend on App Engine, logging timestamps for each change. The threshold was set at ≤ 2 seconds. Table 6.8 presents the findings.

Table 6.8: Real-Time Collaboration Test Results for NFR5

Scenario	Average Time (s)	Meets Threshold ($\leq$2s)
Document Group Change	1.423	Yes
Codemap Change	0.520	Yes

Both scenarios meet the threshold, ensuring seamless collaboration.

- **NFR6 - Visual Feedback for Drag-and-Drop**:Manual testing evaluated the visual feedback during drag-and-drop interactions in the QDAcity coding editor, focusing on document group management and the Codemap editor. Two scenarios were tested: dragging a document to a document group and dragging a code to the Codemap editor. Table 6.9 presents the findings.

Table 6.9: Visual Feedback for NFR6

Scenario	**Highlight on Hover**	**Cursor Change**	**Meets Criteria**
Document to document group	Yes (blue background)	Yes (move icon)	Yes
Code to Codemap editor	Yes (blue background)	Yes (move icon)	Yes

Both scenarios provide clear visual cues, reducing user uncertainty and meeting NFR6.

All NFRs were met, ensuring the QDAcity coding editor delivers an intuitive, responsive, and collaborative experience for all users.

7 Conclusions

This thesis has advanced the usability of the QDAcity coding editor by implementing three pivotal features: hierarchical document group management, a redesigned Codemap, and inline editing. These enhancements, detailed in Chapter 5, were driven by the objective to create an intuitive and efficient platform for QDA researchers, particularly non-technical users, as articulated in Chapter 1. Through a UCD approach, the thesis aligns with the requirements outlined in Chapter 3 and the architectural framework in Chapter 4, achieving significant improvements in usability, organization, and workflow efficiency.

7.1 Summary of Contributions

The thesis's contributions address critical usability gaps in QDAcity, enhancing its effectiveness for qualitative research:

- **Hierarchical Document Group Management**: This feature introduces a structured system for organizing documents into hierarchical groups, enabling categorization by themes, data sources, or analysis stages (FR-DG-1 to FR-DG-7). Implemented with drag-and-drop functionality and real-time collaboration via y.js, it streamlines data management, reducing cognitive load (NFR1). The feature's robustness was validated through unit tests (`DocumentGroupEndpointTest`) and Selenium acceptance tests (`T26CodingEditorDocumentGroupTest`), confirming seamless creation, renaming, relocation, and deletion of groups.
- **Codemap Redesign**: The complex UML-based editor was replaced with a simplified node-edge visualization, making code relationships more accessible to non-technical users (FR-C-1 to FR-C-10, FR-CL-1 to FR-CL-4). Enhanced with customizable node and edge properties and a robust label system, the Codemap supports intuitive exploration. Tests (`T27CodesystemCodeMapTest`, `CodemapLabelEndpointTest`) verified its functionality, with the `spreadOutNodes` function improving layout clarity (NFR2).
- **Inline Editing**: By integrating the `NewItemInput` component, inline edit-

ing eliminates modal dialogs, allowing direct renaming of documents, groups, and codes within the interface (FR-UX-1 to FR-UX-6). This reduces context switching, enhancing workflow efficiency (NFR1). Acceptance tests for document group creation and renaming (`t00_createDocumentGroupTest`, `t03_renameDocumentGroupTest`) validated its consistent behavior.

These features were developed using React for the frontend, Java/Spring Boot for the backend, and TypeScript for the Collaborative Editing System CES, leveraging libraries like mxGraph and y.js, with Google Cloud Platform GCP utilized for hosting. The modular design, as discussed in Chapter 4, ensures maintainability (NFR4), while comprehensive testing validates functionality and performance.

7.2 Achievement of Objectives

The evaluation in Chapter 6 demonstrates that the implemented features meet most specified requirements, significantly advancing QDAcity's usability. Key achievements include:

- **Usability (NFR1)**: Hierarchical organization, simplified visualization, and inline editing reduce the learning curve and cognitive effort, making QDAcity accessible to non-technical researchers, as emphasized in Chapter 2.
- **Functionality**: Robust support for document group operations (FR-DG-1 to FR-DG-7), Codemap interactions (FR-C-1 to FR-C-10, FR-CL-1 to FR-CL-4), and inline editing (FR-UX-1 to FR-UX-6) was validated through unit and acceptance tests, aligning with user needs.
- **Collaboration and Performance (NFR5, NFR2)**: Real-time synchronization via y.js ensures collaborative consistency, with tests confirming responsiveness under typical loads, meeting performance expectations.
- **Maintainability (NFR4)**: The modular architecture and extensive test coverage facilitate future enhancements, as evidenced by the reusable `NewItemInput` component.

7.3 Future Work

To further enhance QDAcity's usability and efficiency, several improvements can be explored to make the platform more powerful and user-friendly for qualitative researchers. Below are detailed suggestions for future development:

7.3.1 Introducing and Enhancing the Recommendation Mode for the Codemap Editor

QDAcity's existing recommendation mode can be introduced to the Codemap Editor, bringing enhanced functionality to provide intuitive, context-aware suggestions. This addition would make the Codemap Editor more collaborative and user-friendly, drawing inspiration from modern productivity tools.

In the Codemap Editor, the recommendation mode could include the following features:

- **Collaborative Node Additions**: When a user adds a node, it could be flagged as a recommended addition, visible to collaborators with an indicator (e.g., a suggestion icon). Team members could then vote or comment on the node, and upon acceptance, it would be fully integrated into the Codemap.
- **Edge Recommendations**: As users define relationships between nodes, the system could suggest additional edges, accompanied by a brief rationale (e.g., "frequently linked in similar projects"), which users can approve or dismiss.
- **Contextual Suggestions**: The system could offer real-time suggestions based on the user's current focus, such as recommending labels when editing node properties.

To facilitate user interaction with these recommendations, the following enhancements could be implemented:

- Users could accept or reject suggestions individually or in batches, with a "suggestion review" panel summarizing pending recommendations and their statuses.
- A history log could track all recommendation interactions (e.g., accepted nodes, rejected edges), allowing users to revisit decisions or undo changes easily.

7.3.2 Advanced Filtering in the Code System

Introducing advanced filtering capabilities to the code system would allow users to manage large datasets more effectively, focusing on specific subsets of their data as needed. This feature would enhance the platform's scalability and usability for complex projects.

Filter by Properties:

- Enable filtering of codes or nodes based on attributes like labels, creation

date, or frequency of use. For instance, users could view only nodes tagged with a specific label or created within a certain timeframe.

- Support multi-criteria filtering, allowing users to combine filters (e.g., nodes labeled "Theme A" and created after a specific date).
- Include numeric range filters, such as filtering nodes by the number of connections.

Saved Filters:

- Allow users to save custom filter configurations for reuse, which is particularly useful for recurring analysis tasks, with options to name and categorize saved filters.
- Provide predefined filters (e.g., “Recently Modified,” “Unlabeled Nodes,” “High-Frequency Codes”) for quick access.
- Enable users to share saved filters with collaborators, enhancing teamwork, with export/import functionality for filter settings.

7.3.3 Undo and Redo System for Coding Editor

To improve usability and collaboration, a comprehensive undo and redo system could be introduced for actions performed in the coding editor, covering the document tree, code tree, code mapping, and codemap editor.

Key Features:

- **Granular Undo/Redo**: Support undo and redo for individual actions, such as adding or deleting nodes, modifying labels, or mapping codes to document content.
- **Collaborative Undo**: Allow users to undo their own actions or, with permissions, revert changes made by collaborators, ensuring seamless teamwork.
- **Visual Indicators**: Provide a clear history panel showing recent actions, with options to undo or redo specific steps, and highlight affected areas in the Codemap or document tree.

Implementation Details:

- Integrate with the existing versioning system to track changes at a granular level.
- Use a stack-based approach to manage undo and redo operations, ensuring performance and reliability.

- Include a confirmation prompt for irreversible actions, such as deleting large sections, to prevent accidental data loss.

These proposed enhancements—ranging from an extended recommendation mode to advanced filtering and a robust undo/redo system—aim to make QDAcity a more efficient and intuitive tool. By providing detailed, actionable suggestions and maintaining user control through configurability, the platform can better support qualitative researchers in managing complex projects while keeping the interface accessible and streamlined.

Appendices

A Evaluation Artifacts

This appendix provides supplementary evidence for the non-functional requirements (NFRs) evaluations presented in Section 6.2, including tables, test logs, screenshots, and code snippets to support the findings.

A.1 NFR5 - Test Logs for Real-Time Collaboration

The test logs below detail the timestamps recorded during the NFR5 evaluation (Section 6.2), conducted on the deployed system at qdacity.com, with the Collaborative Editing System (CES) running on Cloud Run and the Java backend on App Engine. Device 1 initiated the changes, while Devices 2 and 3 observed the updates.

Table 1: Test Logs for NFR5 – Real-Time Collaboration

Scenario	**Device 1 Timestamp**	**Device 2 Timestamp**	**Device 3 Timestamp**
Document Group Change	22:39:54.311	22:39:56.934	22:39:56.934
Document Group Change	22:46:23.711	22:46:24.934	22:46:24.934
Document Group Change	22:47:29.711	22:47:30.934	22:47:30.934
Document Group Change	22:48:39.911	22:48:40.934	22:48:40.934
Document Group Change	22:49:42.911	22:49:43.934	22:49:43.934
Codemap Change	03:47:12.921	03:47:13.154	03:47:13.154
Codemap Change	03:47:20.352	03:47:20.982	03:47:20.982
Codemap Change	03:47:28.782	03:47:30.036	03:47:30.036
Codemap Change	03:48:06.732	03:48:06.977	03:48:06.977
Codemap Change	03:48:20.353	03:48:20.592	03:48:20.592

These logs were used to compute the average times shown in Table 6.8 (Section 6.2), confirming that both scenarios meet the $\leq$ 2s threshold.

A.2 NFR6 - Screenshots for Visual Feedback

The following figures illustrate the visual feedback during drag-and-drop interactions evaluated in NFR6 (Section 6.2). This screenshot show the highlighting

and cursor changes observed during testing.

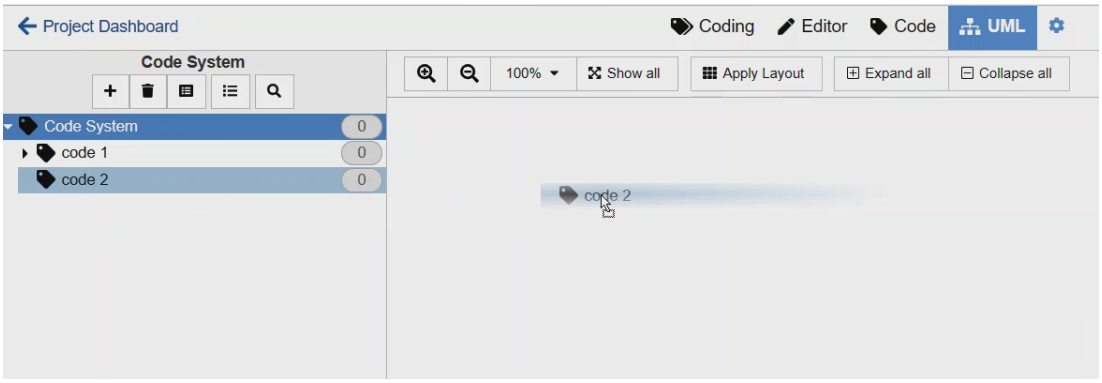

Figure 1: Drag-and-Drop Visual Feedback for Codemap Editor (NFR6): Code is highlighted, and the cursor changes to a move icon.

A.3 NFR2 - Performance Profiling Screenshots

The following figures provide evidence of the performance profiling conducted for NFR2.1 and NFR2.2 (Section 6.2), showing Chrome DevTools results for drag-and-drop and save operations.

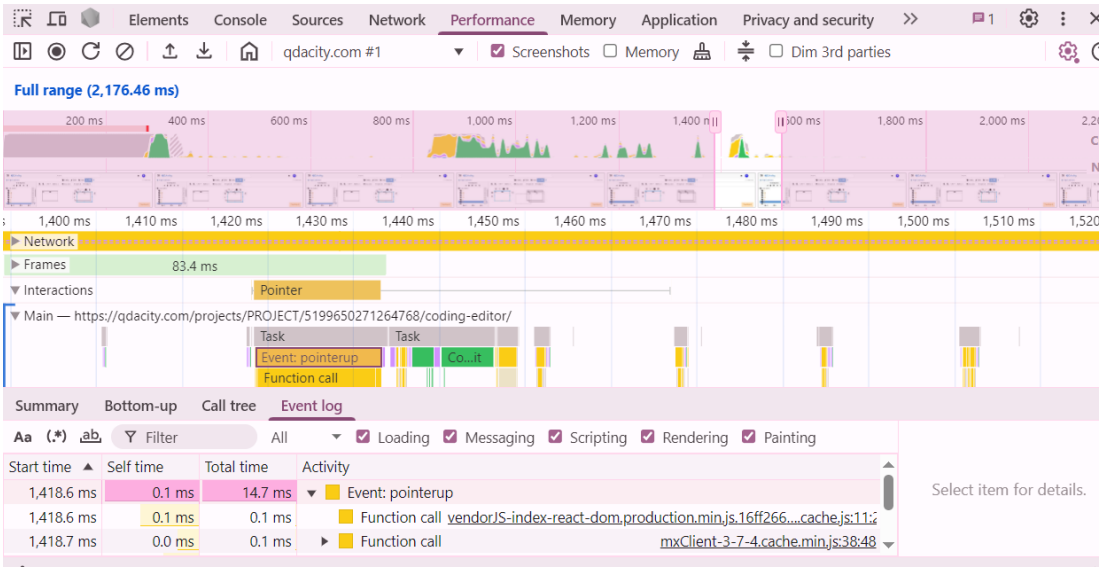

Figure 2: Chrome DevTools Performance Profiling for NFR2.1: Drag-and-Drop Operation (Node in Codemap Editor, 570.0ms).

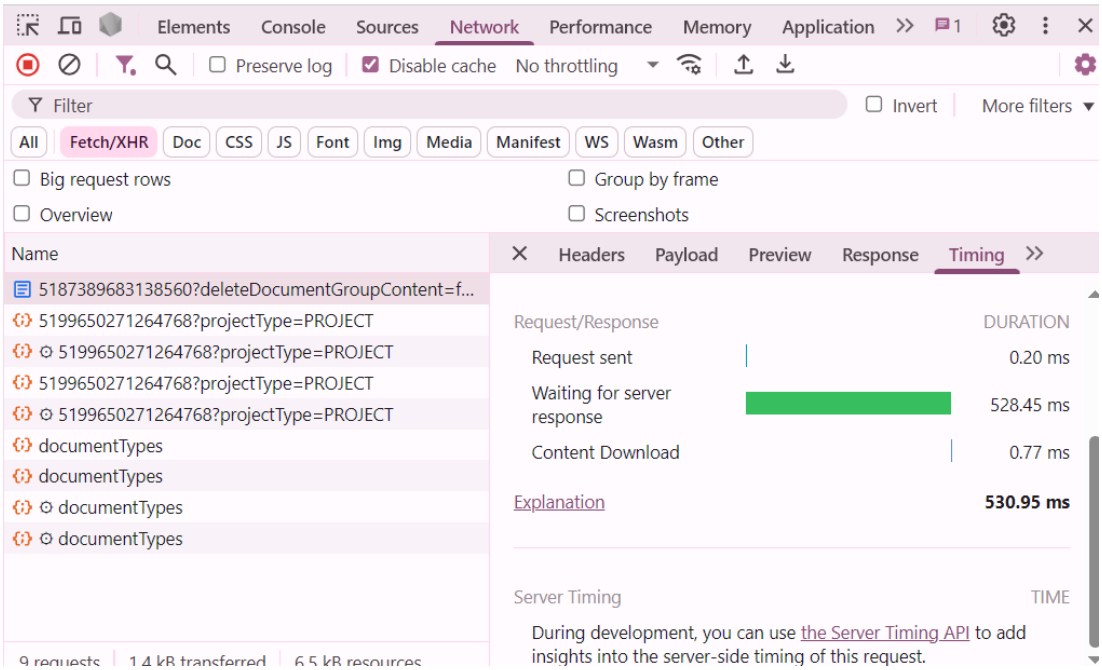

Figure 3: Chrome DevTools Network Profiling for NFR2.2: Save Operation (Delete Document Group, 530.95ms).

A.4 NFR4.3 - Code Snippets for Modifiability

The following code snippets demonstrate the centralized design system and consistent component API evaluated in NFR4.3 (Section 6.2), showcasing styled-components and prop-types usage in the QDAcity coding editor.

```
// theme.js (Centralized Design System)
const primaryBlue = {
  shade5: '#1c4263',
  shade4: '#2662AC',
  shade3: '#5b94d9', // Main shade
  shade2: '#AFCEE9',
  shade1: '#d7e6f4',
};

const primaryDark = {
  shade5: '#000000',
  shade4: '#323232', // Main shade (usage for text)
  shade3: '#595959',
  shade2: '#737373', // usage example: footer
  shade1: '#a6a6a6',
};

const primaryLight = {
  shade5: '#cccccc',
  shade4: '#d9d9d9',
  shade3: '#EBEBEB', // Main shade
  shade2: '#F8F8F8',
  shade1: '#FFFFFF',
};
```

```
25
26  // ... (additional color definitions)
27
28  export default {
29    // General Definitions
30    primaryLogo: primaryBlue.shade3,
31    defaultText: primaryDark.shade4,
32    greyText: primaryDark.shade2,
33    bgDefault: primaryLight.shade1,
34    bgHover: primaryBlue.shade2,
35
36    // Component-Specific Definitions
37    listItem: {
38      bgDefault: primaryLight.shade1,
39      bgActive: primaryBlue.shade3,
40      bgHover: primaryBlue.shade2,
41      textDefault: primaryDark.shade4,
42      textActive: primaryLight.shade1,
43      textInfo: primaryDark.shade2,
44      iconDefault: primaryDark.shade4,
45      shadow: primaryDark.shade3,
46    },
47    // ... (additional component definitions)
48  };
49
50  // DragDocument.jsx (Styled-Components and Prop-Types)
51  import styled from 'styled-components';
52  import PropTypes from 'prop-types';
53  import Theme from 'common/styles/Theme';
54
55  const StyledDocumentItem = styled.a`
56    background-color: ${(props) => (props.$active ? props.theme.listItem.bgActive : ↩
          props.theme.listItem.bg)};
57    color: ${(props) => (props.$active ? props.theme.listItem.textActive : ↩
          props.theme.listItem.textDefault)};
58    padding: 2px 5px;
59    position: relative;
60    display: flex;
61    margin-bottom: -1px;
62    opacity: ${(props) => (props.isDragging ? 0.0 : 1)};
63    &:hover {
64      text-decoration: none;
65      cursor: pointer;
66      background-color: ${(props) =>
67        props.isDragging ? props.theme.listItem.bgActive : props.theme.listItem.bgHover};
68      color: ${(props) => (props.isDragging ? props.theme.listItem.textActive : ↩
            props.theme.listItem.textDefault)};
69    }
70  `;
71
72  const DocumentCollaboratorBubbles = ({ doc }) => {
73    // ... (component logic omitted for brevity)
74  };
75  DocumentCollaboratorBubbles.propTypes = {
76    doc: PropTypes.object,
77  };
78
79  function DragDocument(props) {
80    // ... (component logic omitted for brevity)
81  }
82
83  DragDocument.propTypes = {
84    isDragging: PropTypes.bool,
85    isOver: PropTypes.bool,
86    canDrop: PropTypes.bool,
```

```
  active: PropTypes.bool,
  onClick: PropTypes.func,
  doc: PropTypes.object,
  swapDocuments: PropTypes.func,
  delete: PropTypes.func,
  rename: PropTypes.func,
  isGroup: PropTypes.bool,
  intl: PropTypes.object,
  reccommendationMode: PropTypes.string,
  documentGroups: PropTypes.object,
  tutorialContext: PropTypes.object,
  toggleGroup: PropTypes.func,
  isGroupOpen: PropTypes.bool,
  projectType: PropTypes.string,
  index: PropTypes.number,
  updateDocumentGroupParentAndOrder: PropTypes.func,
  updateDocumentParentAndOrder: PropTypes.func,
};
```

Code Snippet 1: Code Snippet for NFR4.3: Styled-Components with Centralized Theme and Prop-Types in DragDocument.jsx

References

Bostock, M. (2011). D3.js: Data-driven documents [Accessed: 2025-04-07].

Celko, J. (2004). *Joe celko's trees and hierarchies in sql for smarties*. Morgan Kaufmann.

Cooper, A., Reimann, R., & Cronin, D. (2007). *About face 3: The essentials of interaction design*. Wiley.

Corti, L., Van den Eynden, V., Bishop, L., & Woollard, M. (2019). Data management for qualitative research: Ethical and practical considerations. *International Journal of Social Research Methodology*, *22*(3), 231–243.

Davidson, J., et al. (2019). Visualization techniques in qualitative data analysis. *Journal of Mixed Methods Research*, *13*(2), 189–206.

Evans, J. (2011). *Qualitative data analysis: Tools and techniques*. Sage Publications.

FasterXML Team. (2023). Jackson: High-performance json processor for java.

Fielding, R. T. (2000). Architectural styles and the design of network-based software architectures [Accessed: 2025-04-07].

Fowler, M. (2003). *Patterns of enterprise application architecture*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.

Google. (2023a). Google app engine documentation [Accessed: 2025-04-07].

Google. (2023b). Google cloud storage documentation [Accessed: 2025-04-07].

Google Cloud. (2023a). *Google cloud run documentation*. Retrieved April 7, 2025, from https://cloud.google.com/run/docs

Google Cloud. (2023b). *Google datastore documentation*. Retrieved April 7, 2025, from https://cloud.google.com/datastore/docs

IETF. (2011). The websocket protocol [Accessed: 2025-04-07].

International Organization for Standardization. (2014). Systems and software engineering — systems and software quality requirements and evaluation (square) [ISO/IEC 25000:2014. Available at: https://iso.org/standard/64764.html].

International Organization for Standardization. (2019). Ergonomics of human-system interaction — part 210: Human-centred design for interactive sys-

tems [ISO 9241-210:2019. Available at: https://www.iso.org/standard/77520.html].

Jackson, K., & Bazeley, P. (2008). Qualitative data analysis with nvivo. *Qualitative Research Journal*, *8*(1), 85–91.

Jahns, K. (2023). Y.js documentation [Accessed: 2025-04-07].

JGraph. (2023). Mxgraph documentation [Accessed: 2025-04-07].

Kaufmann, A., & Riehle, D. (2018). The qdacity-re method for structural domain modeling using qualitative data analysis. *Requirements Engineering*, *23*(4), 479–499. https://doi.org/10.1007/s00766-017-0284-8

Leech, N. L., & Onwuegbuzie, A. J. (2008). Qualitative data analysis: A compendium of techniques and a framework for selection for school psychology research and beyond. *School Psychology Quarterly*, *23*(4), 587.

Meta. (2023). React documentation [Accessed: 2025-04-07].

Nielsen, J. (1994). 10 usability heuristics for user interface design. *Nielsen Norman Group*. https://www.nngroup.com/articles/ten-usability-heuristics/

Node.js Foundation. (2023). *Node.js documentation*. Retrieved April 7, 2025, from https://nodejs.org/en/docs/

Objectify Team. (2023). Objectify: A java data access api for google cloud datastore.

QDAcity. (2025). Qdacity: Qualitative data analysis platform [Accessed: 2025-03-23]. https://qdacity.com/qda-software

QDAcityDocumentation. (2024). Qdacity documentation [Internal project documentation, accessed in March 2025].

Rosenfeld, L., Morville, P., & Arango, J. (2015). *Information architecture: For the web and beyond*. O'Reilly Media, Inc.

Shneiderman, B. (1983). Direct manipulation: A step beyond programming languages. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 143–149.

Silver, C., & Lewins, A. (2014). *Using software in qualitative research*. Sage Publications.

Spring Boot Documentation. (2023). Spring boot documentation [Published by Pivotal Software, Inc., Accessed: 2025-04-07].

Strauss, A. L. (1987). *Qualitative analysis for social scientists*. Cambridge University Press.

Woods, D., et al. (2016). Advancing qualitative research with digital tools. *Qualitative Inquiry*, *22*(5), 379–390.